A Development System For Unmanned Untethered Submersibles

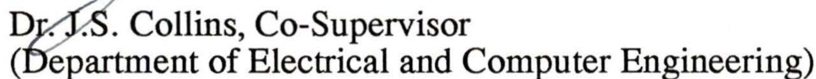
by

Daniel J. Wall
B.A.Sc., University of Waterloo, 1991

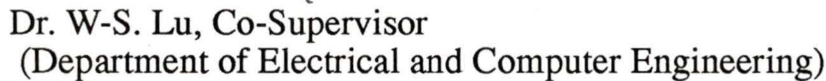
ACCEPTED
DEAN
17 May 94

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
MASTER OF APPLIED SCIENCE
in the Department of Electrical and Computer Engineering

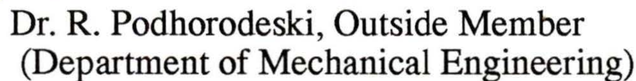
We accept this thesis as conforming to the required standard



Dr. J.S. Collins, Co-Supervisor
(Department of Electrical and Computer Engineering)



Dr. W-S. Lu, Co-Supervisor
(Department of Electrical and Computer Engineering)



Dr. R. Podhorodeski, Outside Member
(Department of Mechanical Engineering)

Dr. M. Nahon, External Examiner
(Department of Mechanical Engineering)

© Daniel J. Wall, 1994

University Of Victoria

All rights reserved. Thesis may not be reproduced in whole or part, by photocopy or other means, without the permission of the author

Supervisor: Dr. James S. Collins

ABSTRACT

This thesis presents a computer-based environment which assists in developing unmanned, untethered submersibles and intelligent controllers which are capable of guiding them. This thesis also describes the use of the development system in a specific application - the development of a vehicle and intelligent controller which together are capable of docking to an underwater dock.

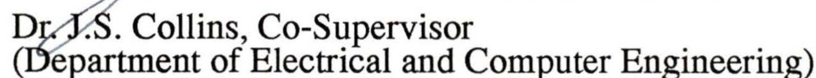
The development system offers a three-stage approach to modelling unmanned, untethered submersibles. In the first stage - source code editing - the designer can change the vehicle dynamics, sensors and controller functions. A block-diagram based design tool (Boeing's EASY5) is provided to ease the task of changing the vehicle dynamics and sensors. The controller design tools take advantage of architectural commonalities between intelligent controllers. These tools include a perception/reasoning/action module, a knowledge base for the physical environment (map) and a knowledge base for general processes (fuzzy inference engine).

The second stage - mission execution - allows the changes made in stage one to be tested in a simulated underwater environment. In this stage, an unmanned, untethered submersible attempts to reach a destination while avoiding obstacles which may be moving. During the mission, the designer can observe the vehicle's state and ensure that the submersible and controller responses are appropriate.

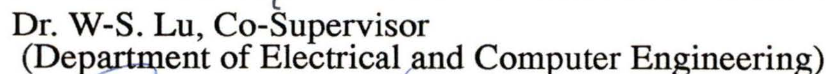
In stage three - mission display - the developer can display the completed mission history from any perspective and at any scale. The history of the completed mission can be animated or represented by line segments.

The submersible and its intelligent controller designed as an example of using this development system, are capable of docking to a subsea dock. The controller architecture has one perception/reasoning/action module, one map and one fuzzy inference engine. The submersible has four sensors to sense linear and angular position and velocity. A lookahead sonar senses changes to the vehicle's immediate surroundings, providing the intelligent controller with a way to recognize, assess and prepare for changes in its subsea environment.

Examiners



Dr. J.S. Collins, Co-Supervisor
(Department of Electrical and Computer Engineering)



Dr. W-S. Lu, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. R. Podhorodeski, Outside Member
(Department of Mechanical Engineering)

Dr. M. Nahon, External Examiner
(Department of Mechanical Engineering)

Table Of Contents

Table Of Contents....................iii
List Of Tables....................vi
List Of Figures....................vii

Chapter 1 Introduction....................1
1.1 Research Goals....................1
1.2 Background....................1
1.3 Related Work....................2
1.4 Development System Characteristics....................4
1.5 Design Stages In The AUV Development System....................6
1.5.1 Source Code Editing Stage....................8
1.5.2 Mission Execution Stage....................10
1.5.3 Mission Display Stage....................11
1.6 Thesis Layout....................11

Chapter 2 Submersible Design Tool....................12
2.1 Medium Submodel....................13
2.1.1 Coordinate System Transformations....................14
2.2 Autonomous Underwater Vehicle Submodel....................16
2.2.1 AUV Dynamics Block....................17
2.2.2 Internal And External Sensors Block....................20
2.2.3 Intelligent Controller Block....................21
2.3 Objects Submodel....................21
2.4 Using The Submersible Design Tool In Simulations....................21

Chapter 3 Intelligent Motion Controller Design Tools....................23
3.1 Background....................23
3.1.1 Perception/Reasoning/Action Module....................23
3.1.2 Knowledge Bases....................26
3.1.3 Mission Management....................27
3.1.4 Summary....................27
3.2 Perception/Reasoning/Action Module....................28
3.2.1 Perception Task Class....................30
3.2.2 Reasoning Task Class....................34
3.2.3 Action Task Class....................35
3.2.4 Summary....................36
3.3 Knowledge Base for Physical Environment - Map....................36
3.3.1 Map Data Structures....................37
3.3.2 Map Coordinate System....................39
3.3.3 Map Indexing....................40
3.3.4 Transformations....................41
3.3.5 Path Planning....................41
3.3.6 Path Validation and Path Boundary Violation Functions....................44
3.3.7 Map Updating....................45
3.4 Knowledge Base For General Processes - Fuzzy Inference Engine....................45
3.4.1 Membership Function Class....................47

3.4.2 Fuzzy Variable Mapping Class 49
3.4.3 Fuzzy Rule Class 49
3.4.4 Fuzzy Associative Memory Class 50
3.4.5 Hyper-FAM Class 51
3.4.6 Fuzzy Encoding, Inference and Defuzzification Procedures 53
3.4.7 Summary 57
3.5 Mission Manager 57
Chapter 4 Design Of A Docking-Capable AUV 59
4.1 Motivation 59
4.2 Work Related To Docking 60
4.3 Submersible Dynamics 61
4.4 Sensors 64
4.5 Intelligent Controller 66
4.5.1 Mission Manager 68
4.5.2 Perception Task Object 69
4.5.3 Reasoning Task Object 72
4.5.4 Action Task Object 75

Chapter 5 Performance Evaluation Of Docking-Capable AUV 80
5.1 Path Following Algorithm 81
5.2 Cornering Algorithm 82
5.3 Dynamic Obstacle Avoidance And Static Unmapped Obstacles 84
5.4 Subgoal Under/Overshooting 86
5.5 Map Saturation and Dock Avoidance 87
5.6 Multiple Dynamic Obstacles and Obstacle/Dock Proximity 89
5.6.1 Summary 91

Chapter 6 Conclusions And Future Work 92
6.1 Conclusions 92
6.2 Future Work 93

Appendix A - Intelligent Controller Design Tools - Code 100
A.1 PRA Module 100
A.2 Map Class 104
A.3 FIE Class 108

Appendix B - Knowledge Base Configurations For The Docking-Capable Intelligent Controller 113
B.1 Map Configuration 113
B.2 FIE Configuration 114
B.2.1 FIE Definition File - fiedef.inp 114
B.2.2 Hyper-FAM Definition File - fuzzhype.inp 114
B.2.3 FAM Definition File - fuzzam.inp 115
B.2.4 Fuzzy Rule Definition File - fuzzrule.inp 118
B.2.5 Fuzzy Variable Mapping File - fuzzvars.inp 120
B.2.6 Fuzzy Subset/Degree Of Membership File - fuzzsubs.inp 121

Appendix C - User Manual....122
C.1 Installation....122
C.2 Running VAVE....122
C.2.1 Modifying Source Code....123
C.2.2 Mission Execution....124
C.2.3 Mission History Display....126

List Of Tables

TABLE 1. Program Modules In The Development System 6
TABLE 2. FIE For Low-Level Control 75
TABLE 3. Vehicle Parameters 80
TABLE 4. Additional Knowledge for Improving the DOAA 85

List Of Figures

FIGURE 1. The Development System 7
FIGURE 2. Simulated Scenario In The Mission Execution Stage 10
FIGURE 3. High-Level Design Tool Representation in EASY5 12
FIGURE 4. Three Coordinate Systems In The Submersible Design Tool 13
FIGURE 5. Medium Submodel Contents 14
FIGURE 6. Roll, Pitch, and Yaw Euler Angles in Fixed-Axis Form 15
FIGURE 7. Autonomous Underwater Vehicle Submodel Contents 17
FIGURE 8. Object Submodel Contents 21
FIGURE 9. Functional And Behavioural Controller Architectures 23
FIGURE 10. A Possible Instance Of An Intelligent Motion Controller 28
FIGURE 11. Task Classes Within The PRA Module 29
FIGURE 12. Functions Of The Perception Task Class 31
FIGURE 13. Functions Of The Reasoning Task Class 34
FIGURE 14. Functions Of The Action Task Class 36
FIGURE 15. Data Structures In The Map Object 38
FIGURE 16. Coordinate System For The Map Class 40
FIGURE 17. Tangential Points On An Obstacle 42
FIGURE 18. T-Graph Method 44
FIGURE 19. Configuration of the FIE Knowledge Base 46
FIGURE 20. Data Structures In The FIE 47
FIGURE 21. Fuzzy Subsets 48
FIGURE 22. Fuzzy Variable Mapping 49
FIGURE 23. Fuzzy Rule 50
FIGURE 24. Fuzzy Associative Memory 51
FIGURE 25. Hyper-FAM For Motion Control Of An Unmanned Submersible 52
FIGURE 26. Fuzzy Inference Process (from [Kosko 91], page 321) 54
FIGURE 27. Lookahead Sonar Pattern Function Characteristics 66
FIGURE 28. Architecture Of The Intelligent Motion Controller 67
FIGURE 29. Flow Of Control 68
FIGURE 30. Decision-Making In The Map-Update Routine 71
FIGURE 31. The Cornering Algorithm 77
FIGURE 32. Regions Of Concern For Dynamic Obstacle Avoidance 78
FIGURE 33. Lateral Response of Vehicle (xy plane) 81
FIGURE 34. Longitudinal Response of Vehicle (xz plane) 82
FIGURE 35. Effect Of The Cornering Algorithm (3D, xy plane) 83
FIGURE 36. Dynamic Obstacle Avoidance Algorithm (3D, xy plane, xz plane) 84
FIGURE 37. Waypoint Undershooting (3D, xy-plane, zx-plane) 86

FIGURE 38. Dock Avoidance and Map Saturation (3D, xy plane, xz plane)....88
FIGURE 39. Mission History (3D, xy plane, xz plane)....89
FIGURE 40. VAVE Window And Console....123
FIGURE 41. Mission Execution Window - Pre-Initialization....124
FIGURE 42. Mission Execution Window - Post-Initialization....125
FIGURE 43. Build Instruction Window....125

Chapter 1 Introduction

This chapter describes the motivation for the work presented in this thesis. It also presents an overview of the development system and summarizes related work.

1.1 Research Goals

The primary goal of this thesis is to:

Build a development system for designing unmanned, untethered submersibles and their intelligent control systems.

To prove the viability of the development system and to initiate investigation into the docking process, a secondary goal of this thesis is to:

Use the development system to construct a submersible and intelligent motion controller which can dock to an underwater dock.

1.2 Background

This work was prompted by a lack of research into docking techniques for unmanned, untethered submersibles (commonly referred to as an Autonomous Underwater Vehicles or AUVs). Literature reviews and discussions with engineers at International Submarine Engineering Research (ISER)[1] confirmed that docking was an important but neglected aspect of subsea-vehicle activity. Consequently, initial research focused on developing effective docking procedures.

As work continued, it became apparent that docking encompassed many different aspects, all of which were related to intelligent motion control. Thus, to examine docking specifically, the research group required a framework for examining intelligent motion control in general. Control of the submersible's motion also depends on the dynamics of the submersible to be controlled. The framework subsequently expanded to include a facility for creating and modifying vehicle dynamics as well as intelligent controllers. The desired characteristics of this framework - modularity, economy and flexibility - favoured a computer-based realization over a real-world one.

1. Dr. James McFarlane, Vince den Hertog and Landy Shupe

This thesis presents the resulting framework: an environment which facilitates the design and evaluation of submersibles and their intelligent motion controllers. This thesis also describes an AUV, designed in the development system, which is capable of docking with a targeted dock.

1.3 Related Work

This section discusses work related to large-scale development systems for AUVs. A discussion of work related to docking takes place in Chapter 4, Section 2 which discusses the design of a docking-capable AUV as an example of using the development system. It should be noted that the main focus of this thesis is the AUV development system, not the design of a docking-capable AUV.

In the late 1950s, interest in submarine stability and control prompted the David W. Taylor Naval Ship R&D Center to develop a set of standardized equations of motion for submersibles [Gertler 67]. These 6 non-linear equations described the vehicle's angular and linear velocities with respect to a vehicle-based reference frame. A revised set of these equations were published later when Feldman correlated the previous equations with full-scale trial data [Feldman 75] [Feldman 79]. Other derivations have since appeared [Aucher 81] [Fossen 91].

Over the same time period, the integrated circuit has given control systems new capabilities, allowing them far more autonomy than their analog predecessors. Controllers - perceiving, reasoning and acting - have started to predict and respond in ways that resemble human intelligence. The field of "computational intelligence" now encompasses hardware and software developments which attempt to design intelligent behaviour into machines.

With a wide variety of potential scientific, industrial and military applications, submersibles are a prime candidate for intelligent control. Most intelligent control research related to submersibles is performed in a virtual environment where mistakes are less costly since dynamic equations replace a real vehicle.

A research team at Texas A&M is developing an intelligent controller for an AUV. The controller architecture is a distributed control system based on the Fault Tolerant Multi-Processor (FTMP) system [Hess 92]. The FTMP approach uses hardware-based fault tolerance over a network of 16 Sun Sparcstations. The 16 Suns form a distributed intelligence network which controls the vehicle. The performance of the controller and vehicle is analysed through a series of graphical user interfaces [Nelson 92]. Five separate interfaces display different aspects of a simulated mission. One user interface allows designers

to plan missions, constructing maps of targets, obstacles and threats and creating mission orders. A fault-tolerance interface monitors the status of 16 Suns. Other interfaces display subsystem gauges and show the vehicle's path through the subsea medium.

Another group at the Naval Postgraduate School combines simulation with real-world testing of its NPS AUV II vehicle [Brutzman 92] [Healey 92]. Here, newly developed software is tested in a "pre-mission" simulation phase. The "pseudo-mission" phase tests performance of the software onboard the dry-docked vehicle. Finally, in the "post-mission" phase (after the real AUV has executed the mission), the simulator plays back a history of the vehicle's state during the mission. In each case, the simulator is an integral part of the controller design.

Also at the Naval Postgraduate School, attention has focused on automated construction of AUV controllers using the Computer-Aided Prototyping System [Lee 91]. With a Prototype Description Language, a designer can enter the specifications of the controller. The environment then retrieves the necessary code segments from a base of reusable components.

The Charles Stark Draper Lab in conjunction with MIT/Sea Grant has an on-going AUV technology program. Recent publications have examined fault tolerant design optimization [Babcock 90], layered control [Bellingham 90] and trends in controller architectures [Hall 92].

With a few exceptions, work in this field has ignored the object-oriented approach as an option [Ornulf 91][Zheng 91]. Reasons for this may be that most projects which are at the publication stage may have started before the object-oriented methodology became popular.

The above development systems focus on the design of a particular vehicle or controller architecture for a specific task. Because of this, the systems do not offer the capability to study generalized AUV control problems. While an ideal solution may be found for a specific vehicle and mission type, the problems which apply generally to all AUVs are not necessarily treated. A more general development system is required which allows the designer to test general approaches for different types of vehicles.

Related to the fact that the development systems are not generalized is the fact that they do not offer any tools which can be repeatedly used in the design of new configurations. Each new design thus has to start at a primary design stage or borrow parts from an existing design. Borrowing components often leads to compromises which restrict the new design.

Generalized design tools are required which the designer can fine tune to meet the specifications for a new design.

1.4 Development System Characteristics

Economic Operation

In general, a simulated testing environment is less expensive (faster to configure and test) than its real-world equivalent. The expenses and delays associated with real-world testing often mean that more resources are spent on maintaining a real-world operation than researching new techniques.

Simulation cannot take the place of the real-world however. The intent here is to use the simulated environment until such time as an acceptable design has been reached and then transfer the design concepts to a real vehicle.

The development system is computer-based since, in initial design phases, simulated environments are more economical than their real-world equivalents.

Component Modularity

A modular environment allows specialists to work on one aspect of the environment without requiring broad-based knowledge of how the entire system functions. For instance, in a modular environment, a dynamics expert will be able to alter the submersible dynamics without having to understand the intricacies of the vehicle controller.

The development system is modular so that changes to one component do not propagate into other components.

Block Diagram-Based Tool For Submersible Design

Generally, models are simpler to analyse and design when presented in a graphical format. The development system uses EASY5, a dynamic system simulator from Boeing, as a platform for designing the physical aspects of the submersible, including the onboard sensors.

This platform also houses the dynamics and kinematics of the obstacles, targets and the medium with which the submersible interacts. After designing a submersible, the developer can "configure" the environment to test the newly changed submersible. It is impor-

tant to note that modelling of the obstacles, targets and the medium is secondary. The development system focuses on improving submersibles *not* obstacles and targets.

The development system offers a graphical, block-diagram-based tool for designing the physical aspects of the submersible.

Rapid Construction and Reconfiguration of Intelligent Motion Controllers

Intelligent control systems are defined as those control systems which, if a human to perform the same task, would require human intelligence. Despite having significantly different architectures, functions, and data structures, different controllers share some fundamental characteristics. These elemental similarities form the "building blocks" of all intelligent controllers. To facilitate easy construction and reconfiguration of intelligent controllers, the development system takes advantage of these similarities - offering tools (or building blocks) which are based on these fundamental similarities and are therefore applicable to the design of *all* intelligent controllers. This allows work on controllers to start at a more advanced stage, freeing the designer from having to create basic control system entities.

The development system offers intelligent controller design tools that are based on commonalities of all intelligent motion controllers.

Performance Evaluation of AUVs

After modifying the intelligent controller and/or vehicle, the designer will want to evaluate the changes. The development system therefore provides a performance evaluation mechanism. This entails simulating a subsea medium in which an AUV with an onboard intelligent motion controller can execute test missions. With the subsea simulation, the designer is able to define, execute and display test missions quickly.

A common argument against simulated environments is that they cannot effectively reproduce the uncertainties inherent in a real-world environment. This implies that developments in a simulated environment cannot be transferred to the real world without significant redesign. For devices that interface directly to the environment such as sensors and actuators, this may be a valid concern if the devices are affected by complex aspects of the real-world which cannot be modelled accurately. However, for devices that work on a more symbolic level (including intelligent motion controllers), low-level interactions with the environment are secondary. Emphasis is on acting appropriately (and quickly) given an approximate model of the environment rather than acquiring a perfectly accurate

model before acting. However, this does not preclude the use of the development system to refine AUV dynamics as well.

The development system allows the designer to define, execute and display test missions in a simulated subsea medium.

1.5 Design Stages In The AUV Development System

Through a graphical user interface (GUI), the development system offers a three-stage approach to all aspects of AUV design. In the first stage - source code editing - the designer modifies the intelligent controller, vehicle dynamics and/or onboard sensors. After compiling the new source code, the designer enters stage two, mission execution, in which the changes are tested on customized missions. The third stage, mission display, displays completed mission histories, allowing the designer to inspect them from any perspective. The designer can display a completely animated version of the missions or a line segment representation of the vehicle's trajectory during the missions. Figure 1 summarizes the different stages in the system.

The code which implements the development system is contained in four distinct programs which interact through process forking[2], inter-process signalling and file manipulation. Table 1 outlines the characteristics of the five program modules. All code runs under UNIX in the X-Windows environment.

TABLE 1. Program Modules In The Development System

Module	Language	Application Used In/Containing Module
User Interface	C	Devguide (OpenWindows Application Development Guide)
Submersible Design Tool	Fortran	EASY5 (Dynamic System Simulator)
Intelligent Motion Controller Design Tools	C++	None
Mission Execution Program	C, C++, Fortran	EASY5/DevGuide/C++ are components
Mission History Display Program	C	AutoCAD (Computer Aided Design and Display)

2. In *forking* a process, a process invokes a new program. The newly created program runs in parallel with the process which forked it.

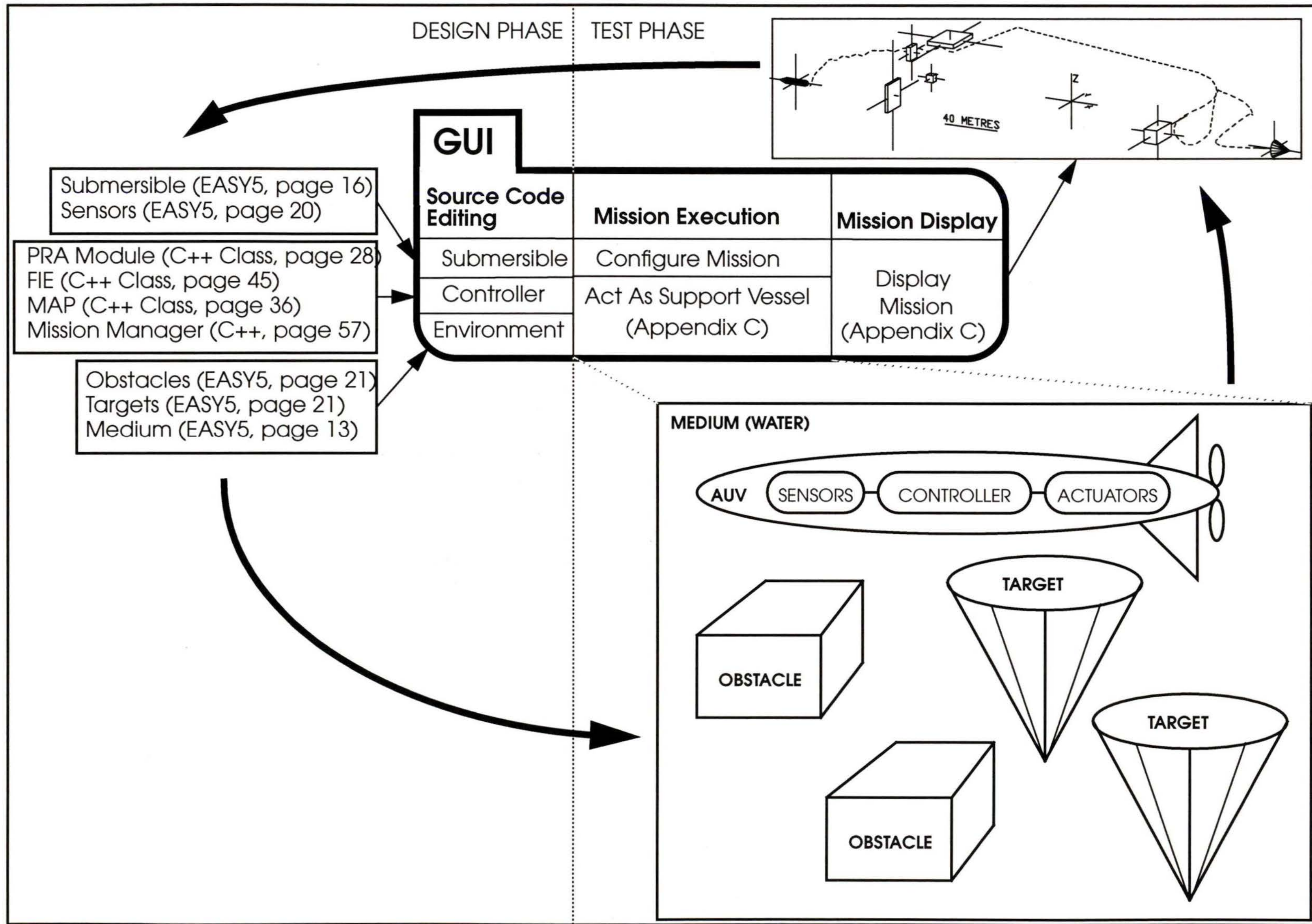


FIGURE 1. The Development System

1.5.1 Source Code Editing Stage

Modifications to the vehicle and its controller are made in the source code editing stage. All modifications are made within the structure provided by the design tools, however, the design tools do not restrict the developer to particular theories or algorithms. Instead, they provide a modular framework which allows development to occur in a structured, flexible setting. The design tools also offer some built-in functions which the designer may use as desired.

Submersible Design Tool

The submersible design tool is a dynamics system simulator called EASY5. EASY5 allows the user to work in a block-diagram environment creating and modifying parameters related to the physical aspects of the submersible.

Submersible design is separated into submersible dynamics design and sensor design. The submersible design component takes control surface inputs from the controller and outputs linear and angular velocities. Vehicle dynamics form the "plant" between the inputs from the controller and the outputs to the environment. The dynamics of the submersible can be constructed from EASY5's libraries of numerical computation components (integrators, etc.) or can be hand-coded in Fortran.

Also modelled in the submersible design tool is the submersible's interaction with the medium, obstacles and targets. These aspects are intrinsically coupled with the AUV dynamics in that the AUV dynamics depends on the characteristics of the medium.

Within the sensor design component, sensors are further distinguished as internal or external sensors. Internal sensors are defined as those sensors which sense some aspect of the submersible's state such as linear velocity or fuel level. External sensors on the other hand, sense some aspect of the environment. A sonar is an example of an external sensor. Like the submersible dynamics, sensors can also be constructed from numerical computation component libraries or hand-coded in Fortran.

Intelligent Motion Controller Design Tools

An intelligent motion controller contains *data structures* and *functions* within its *architecture*. The functions interact with one another and operate on the data structures according to the constraints of the architecture. The performance of an intelligent controller depends on an effective configuration of these three elements.

The design tools cut across the function/data structure/architecture boundaries taking advantage of commonalities between the functions, structures and architectures of all intelligent controllers. Each tool is a code template intended as a starting point for further enhancement. There are four design tools: A Perception/Reasoning/Action task class, a Map class, a Fuzzy Inference Engine Class and a Mission Manager.

Perception/Reasoning/Action Task Class: Regardless of how an architecture packages the functions and data structures, every intelligent controller continuously cycles through the tasks of perception, reasoning and action. The perception task transforms sensor inputs to representations which are comprehensible to the reasoning and action tasks. The reasoning task assesses situations and plans responses which in turn are executed by the action task. Intelligent controller architectures often contain several sets of these three-task cycles, running concurrently and accessing different data structures and functions. The design tool here, a Perception/Reasoning/Action class, captures the essence of the perception, reasoning, and action tasks and makes them available to the designer for duplication and modification.

Map Class: All intelligent controllers contain maps which record and produce information about the physical state of their surroundings [Brooks 86]. Both functions and data structures are associated with these maps. Therefore, a map class composed of functions and data structures is included as a design tool. Since intelligent controllers often use several different kinds of maps, the development system permits several different versions of the map to exist in a single controller.

Fuzzy Inference Engine Class: Intelligent controllers have a capability to make decisions in vague and varying situations. To make knowledgeable decisions in uncertain conditions, intelligent controllers encode inexact knowledge and infer from it[3]. The development system offers a tool which encodes and infers knowledge about any process. The fuzzy-logic based tool stores rules, outputting an inferred result according to the inputs and the fuzzy rule set. Again, a single controller can represent diverse sets of knowledge by incorporating several instances of the tool.

Mission Manager: The Mission Manager contains the high-level functions of an intelligent controller which generally coordinate the lower-level modules. Due to the abstract nature of the requirements at this level, the format of the Mission Manager is left to the

---

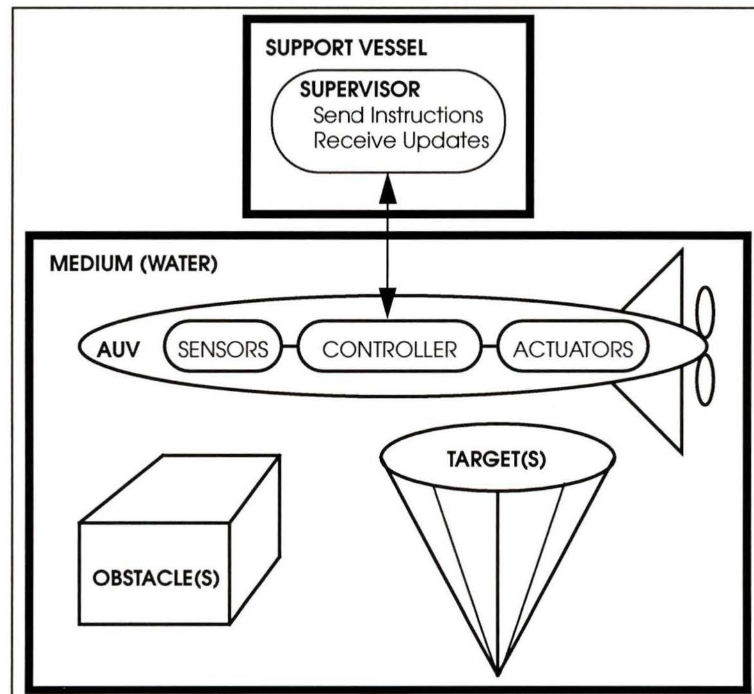
3. This knowledge takes many forms - low-level control functions, sensor fusion algorithms, obstacle avoidance, map updating algorithms, etc.

designer's discretion. A template Mission Manager is provided which gives access to all the other design tools.

1.5.2 Mission Execution Stage

After designing an AUV and its intelligent controller in the source code editing stage, the developer enters the mission execution stage to evaluate the design. The mission execution stage simulates a scenario where an AUV executes a mission requiring it to move through water, avoiding obstacles, to reach a destination. A manned support vessel supervises the mission, infrequently communicating with the AUV to send instructions and receive status reports (via a simulated acoustic link). The user interface simulates the computer onboard the manned support vessel. Figure 2 illustrates the entities in the simulated scenario.

FIGURE 2. Simulated Scenario In The Mission Execution Stage



Initially, the mission execution stage requires the designer to "configure" the AUV's environment (true locations of the AUV, targets and obstacles) and to initialize the knowledge bases in the submersible controller. The contents of these knowledge bases (fully described in Appendix B) are inference rules for decision making as well as knowledge of the submersible's physical surroundings.

After configuring the submersible's environment and knowledge bases, the designer activates the simulated AUV (vehicle model and controller together) and its physical surroundings (the targets and obstacles). Having assumed a supervisory role, the designer/supervisor communicates with the simulated vehicle through the user interface.

At all times, the user is able to monitor the vehicle's status and record its performance. If desired however, all interactions between the vehicle and its physical environment can be hidden, creating a strong illusion that the user is supervising (rather than "controlling") a remotely operating vehicle.

1.5.3 Mission Display Stage

At this stage, the submersible has completed one or more missions and the designer is interested in analyzing the path of the submersible throughout the mission. With visual feedback on the performance of the modified vehicle, the designer will return to the first stage and start the development cycle again.

The upper right-hand corner of Figure 1 illustrates sample output from the mission display stage.

1.6 Thesis Layout

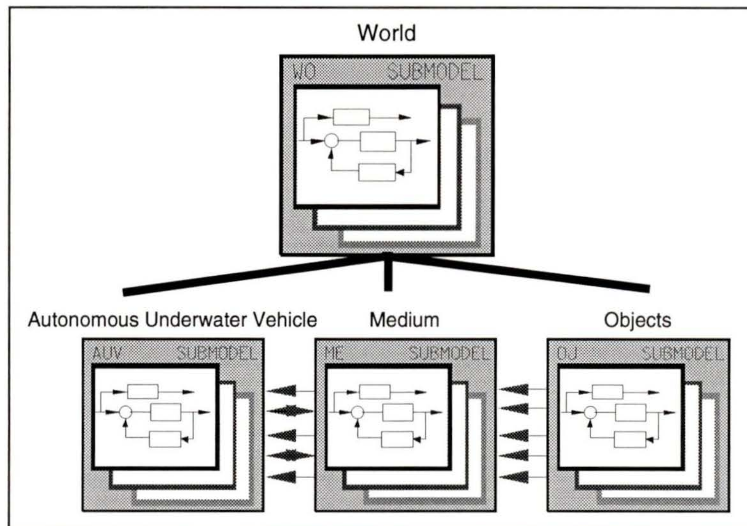
This thesis focuses primarily on the source code editing stage, discussing the design tools and the underlying theory in detail. Chapter 2 describes the submersible dynamics design tool, while Chapter 3 discusses the intelligent controller design tools. Chapter 4 presents the design of an AUV which is capable of docking to dock to an underwater dock. Chapter 5 evaluates the performance of the control algorithms in the docking-capable AUV. Appendix A presents some of the class definitions for the intelligent controller design tools. Appendix B presents the knowledge base configuration files for the AUV and intelligent controller discussed in Chapters 4 and 5. Since the mission execution and mission display stages are not central to the "research" aspects of this work, they are discussed in Appendix C - User Manual.

Chapter 2 Submersible Design Tool

The submersible design tool is the module which represents and allows changes to the physical behaviour of and interactions between the submersible, medium, obstacles and targets. Submersible dynamics are an integrated part of this tool rather than the only component. The design tool is implemented on EASY5 (Engineering Analysis System), a block diagram-based, dynamic system simulator from Boeing Computer Services. EASY5 was chosen over other dynamic system simulators because it was compilable rather than interpretive. It was also free where other products had substantial costs.

To construct a model, the user adds components and connections, forming a block-diagram representation of some real-world system. Following this, the user compiles the block-diagram model into a Fortran program whose executable simulates the physical behaviour of the AUV, medium, obstacles and targets. Figure 3 illustrates the higher level submodels of the design tool as EASY5 displays them.

FIGURE 3. High-Level Design Tool Representation in EASY5



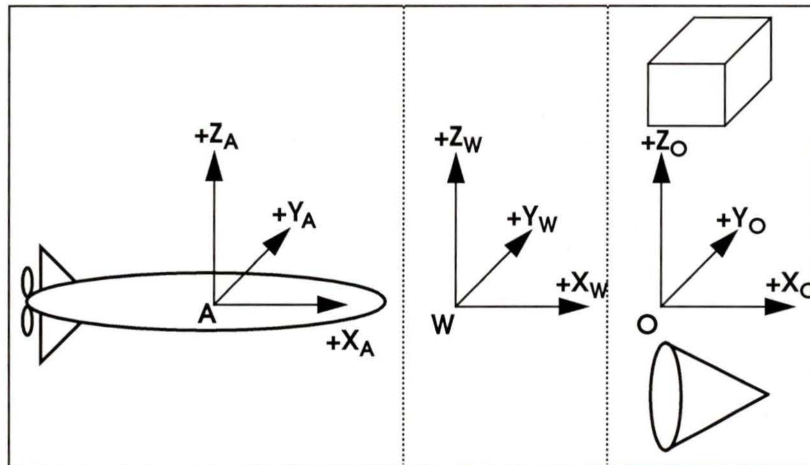
The submodel labelled Autonomous Underwater Vehicle consists of the dynamics of the AUV and its sensors. *All the submersible design work is done in this submodel; the other submodels simply support the development of the submersible design and provide a means to evaluate it.* The Medium submodel contains kinematic transformations between coordinate systems as well as the dynamics of the medium. The Objects submodel contains the obstacle and target dynamics which the submersible interacts with during a test mission.

Each of the three submodels contain further levels of detail which are discussed in the next sections. The Medium Submodel is discussed first since it contains information about the coordinate systems and the transformations between them.

2.1 Medium Submodel

The medium submodel carries out transformations between coordinate systems. The model has three right-hand coordinate systems. A coordinate system fixed to the world (W) relates AUV, obstacle and target motion to an earth-bound observer. To reduce computational complexities in the submersible dynamics module, another coordinate system (A) is fixed to the body of the AUV (specifically, the origin of A is the AUV's mass center) with axes oriented along the principle axes of the submersible. Finally, a coordinate system for obstacles and targets (O) allows object movement to be specified with respect to a reference frame which has been translated some distance from W. Each object can move independently with respect to O, however to reduce computational complexity, the axes of O and W are always parallel. Figure 4 shows the configuration of the three systems.

FIGURE 4. Three Coordinate Systems In The Submersible Design Tool

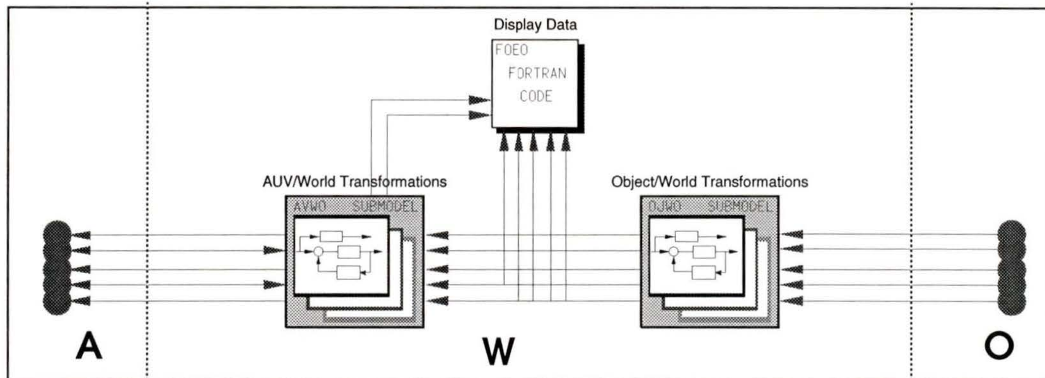


The Medium submodel contains the transformations which map the velocities in the submersible's coordinate system (A) to the world coordinate system (W). Also included are transformations which map points from W to points in A. The AUV/World Transformations block in Figure 5 contains these two transformations.

This submodel also contain transformations which map object positions and orientations in O to W. These are represented in Figure 5 by the Object/World Transformations block.

Mappings from the World coordinate system to the Object coordinate system are possible but are not currently necessary.

FIGURE 5. Medium Submodel Contents



Mapping an object in O to the submersible's coordinate system, A, requires two transformations, one from O to W and a second from W to A. These mappings are necessary when modelling external sensors on the submersible.

The Display Data block in the Medium submodel writes the positions of all entities to a file which is subsequently used to display mission histories.

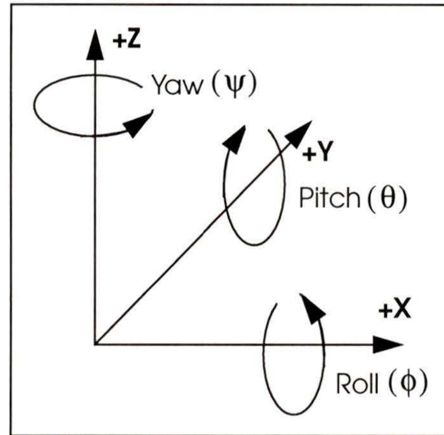
Although none are currently in place, models of current action, wave action and the acoustic channel are feasible enhancements to the Medium submodel.

2.1.1 Coordinate System Transformations

Transformations between coordinate systems require the definition of kinematic relationships between systems. The transformation between W and O is a simple translation since axes of W and O are always parallel. However, A is not similarly restricted - it rotates relative to W. Transforming between A and W requires a translation followed by a rotation. The rotation uses the Euler angles roll (ϕ), pitch (θ), and yaw (ψ) (illustrated in fixed-axis form[4] in Figure 6).

4. *Fixed-axis* operations imply that successive rotations are performed about fixed axes. *Current- axis* operations imply that successive rotations are performed about the new, rotated axes [Spong 89].

FIGURE 6. Roll, Pitch, and Yaw Euler Angles in Fixed-Axis Form



Conversion of a vector from the vehicle-fixed frame (A) to the earth-fixed frame (W) requires rotating A by a roll angle about the X_A axis, a pitch angle about the Y_A axis and a yaw angle about the Z_A axis. The rotations are represented by the rotation matrix, $\boldsymbol{J_1}(\phi,\theta,\psi)$, in the following equation:

$$\vec{v}_W = \boldsymbol{J_1}(\phi,\theta,\psi)\,\vec{v}_A \tag{EQ 1}$$

where $\vec{v}_W$ and $\vec{v}_A$ represent the same vector expressed relative to the orientations of W and A respectively.

The rotation matrix, $\boldsymbol{J_1}(\phi,\theta,\psi)$, is calculated as:

$$\boldsymbol{J_1}(\phi,\theta,\psi) = \boldsymbol{R}(Z_A,\psi)\cdot\boldsymbol{R}(Y_A,\theta)\cdot\boldsymbol{R}(X_A,\phi) \tag{EQ 2}$$

where $\boldsymbol{R}()$ is the basic rotation matrix [Spong 89]. The final form of $\boldsymbol{J_1}(\phi,\theta,\psi)$ is:

$$\boldsymbol{J_1}(\phi,\theta,\psi) = \begin{bmatrix} c\psi c\theta & (-s\psi c\phi + c\psi s\theta s\phi) & (s\psi s\phi + c\psi c\phi s\theta) \\ s\psi c\theta & (c\psi c\phi + s\phi s\theta s\psi) & (-c\psi s\phi + s\theta s\psi c\phi) \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \tag{EQ 3}$$

where $sx \equiv \sin x$, $cx \equiv \cos x$. Calculating $\boldsymbol{J_1}(\phi,\theta,\psi)$ with current-frame rotations yields an identical result [Spong 89]. Note that rotation occurs in the reverse direction however (in other words, a yaw followed by a pitch and roll).

Equation 1 transforms vector quantities between orientations of frames of reference. In the simulation, this equation transforms the linear velocities of the vehicle expressed with respect to the orientation of the body-fixed frame A, to linear velocities with respect to the

earth-fixed origin W. The integration of these earth-relative velocities yields the AUV's location in the three-dimensional medium.

The three Euler angles indicate the AUV's orientation relative to the world coordinate system. However, while $\boldsymbol{J_1}(\phi,\theta,\psi)$ is able to transform vector quantities between coordinate systems, it cannot transform angular data. A different matrix, $\boldsymbol{J_2}(\phi,\theta,\psi)$, is required to transform the angular velocity vector, $\omega_A = (p, q, r)^T$, to the Euler rate vector $\omega_W = (\dot{\phi},\dot{\theta},\dot{\psi})^T$. p, q and r are angular velocities about the x, y and z axes of an inertial frame oriented as A, respectively. The rate of change of the Euler angles is determined as:

$$\vec{\omega}_W = \boldsymbol{J_2}(\phi, \theta, \psi)\,\vec{\omega}_A \qquad \text{(EQ 4)}$$

where,

$$\boldsymbol{J_2}(\phi, \theta, \psi) = \begin{bmatrix} 1 & s\phi t\theta & c\phi t\theta \\ 0 & c\phi & -s\phi \\ 0 & \dfrac{s\phi}{c\theta} & \dfrac{c\phi}{c\theta} \end{bmatrix} \qquad \text{(EQ 5)}$$

where $tx = \tan x$. Written compactly, the kinematic equations for linear and angular velocity transformations between W and A are:

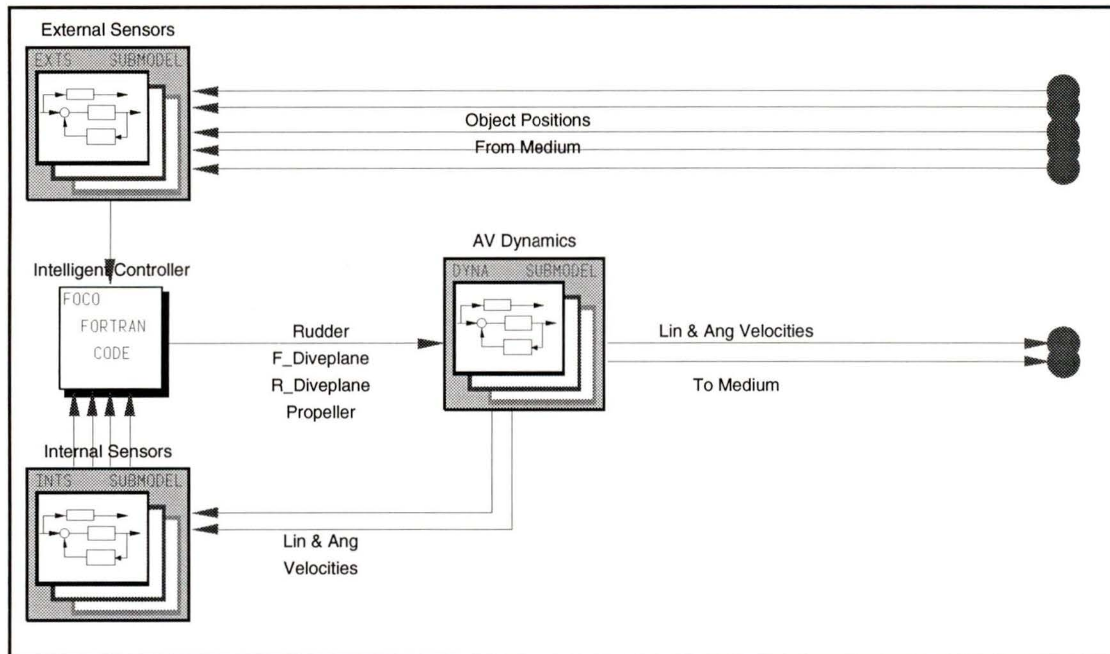
$$\begin{bmatrix} \vec{v}_W \\ \vec{\omega}_W \end{bmatrix} = \begin{bmatrix} \boldsymbol{J_1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{J_2} \end{bmatrix} \begin{bmatrix} \vec{v}_A \\ \vec{\omega}_A \end{bmatrix} \qquad \text{(EQ 6)}$$

Integration of the earth-relative velocities yields the AUV's location and orientation in the world coordinate system.

2.2 Autonomous Underwater Vehicle Submodel

The Autonomous Underwater Vehicle Submodel contains the AUV dynamics, sensor models and an interface to the intelligent controller. Figure 7 illustrates the contents of the Autonomous Underwater Vehicle submodel.

FIGURE 7. Autonomous Underwater Vehicle Submodel Contents



2.2.1 AUV Dynamics Block

The AUV Dynamics block contains the vehicle's equations of motion. Several publications present detailed derivations of the dynamic equations for rigid bodies [Fossen 91] [Aucher 81] [Beer 84]. This section presents the framework in which AUV dynamics models can be developed. Beyond a few fundamental assumptions and theories which provide a standardized approach, the design tool imposes no restrictions on the development of submersible dynamics.

Assumptions

The design tool assumes the following standard assumptions which ease the task of modelling the dynamics of the submersible. The simplifications are:

1. The vehicle is a rigid body, therefore the model can ignore forces acting on the AUV's individual mass elements.
2. The earth is fixed in space, therefore the model can ignore forces acting on the vehicle due to the earth's rotation about a star-fixed reference frame.
3. The vehicle is axisymettric, therefore all its products of intertia are zero.
4. The vehicle stays sufficiently below the sea surface and above the seafloor so that wave motion and ground effects can be ignored.

Derivation

Newton's Second Law applied to the underwater vehicle is:

$$\vec{F}_W = m\vec{a}_W \tag{EQ 7}$$

where $\vec{F}_W$ is the resultant of the external forces acting on the vehicle, m is the vehicle mass and $\vec{a}_W$ is the acceleration of the mass center relative to W. Since the vehicle acceleration, $\vec{a}_A$, is expressed in a rotating frame, $\vec{a}_W$ and $\vec{a}_A$ are related by:

$$\vec{a}_W = \vec{a}_A + \vec{\Omega} \times \vec{V}_A \tag{EQ 8}$$

where,

$\vec{a}_A = \dot{u}\hat{i}_A + \dot{v}\hat{j}_A + \dot{w}\hat{k}_A$ is the linear acceleration of the mass center (relative to an inertial frame) expressed as components in A,

$\vec{\Omega} = p\hat{i}_A + q\hat{j}_A + r\hat{k}_A$ is the angular velocity of the submersible (about an inertial frame) expressed as components in A[5] and,

$\vec{V}_A = u\hat{i}_A + v\hat{j}_A + w\hat{k}_A$ is the linear velocity of the mass center (relative to an inertial frame) expressed as components in A[6].

If,

$$\vec{F}_W = F_x\hat{i}_A + F_y\hat{j}_A + F_z\hat{k}_A \tag{EQ 9}$$

substituting 9 into 7, equating with 8 and collecting terms yields:

$$\begin{aligned} F_x &= m(\dot{u} + qw - rv) \\ F_y &= m(\dot{v} + ru - pw) \\ F_z &= m(\dot{w} + pv - qu) \end{aligned} \tag{EQ 10}$$

The analysis now focuses on the moments created due to the external forces:

5. $\vec{\Omega}$ is equivalent to $\vec{\omega}_A$ on page 18.
6. $\vec{V}_A$ is equivalent to $\vec{v}_A$ on page 18.

$$\vec{M}_A = \dot{\vec{H}}_{A'} \qquad \text{(EQ 11)}$$

where M_A is the net moment about the vehicle mass center expressed in terms of A and $\dot{\vec{H}}_{A'}$ is the time rate of change of angular momentum about A' (whose origin is the vehicle's mass center and whose axes are oriented with W). $\dot{\vec{H}}_{A'}$ is written as:

$$\dot{\vec{H}}_{A'} = (\dot{\vec{H}}_{A'})_{Axyz} + \vec{\Omega} \times \vec{H}_{A'} \qquad \text{(EQ 12)}$$

where,

$(\dot{\vec{H}}_{A'})_{Axyz.}$ is the rate of change of $\dot{\vec{H}}_{A'}$ with respect to an inertial frame instantaneously coincident with A and,

$\vec{\Omega}$ is the angular velocity of the submersible (about an inertial frame) expressed as components in A (defined in equation 8)

Since the x, y, and z axes correspond to the principal axes of inertia of the axisymettric body, simplified relations can be used to determine the components of $(\dot{\vec{H}}_{A'})_{Axyz}$:

$$(\dot{\vec{H}}_{A'})_{Axyz} = \bar{I}_x \dot{p} \hat{i}_A + \bar{I}_y \dot{q} \hat{j}_A + \bar{I}_z \dot{r} \hat{k}_A \qquad \text{(EQ 13)}$$

where,

$\bar{I}_n$ is the principal centroidal moment of inertia about axis n

Substituting 13 into 12 and equating the result with 11 yields the following component equations:

$$\begin{aligned} M_x &= \bar{I}_x \dot{p} - (\bar{I}_y - \bar{I}_z)\, qr \\ M_y &= \bar{I}_y \dot{q} - (\bar{I}_z - \bar{I}_x)\, pr \\ M_z &= \bar{I}_z \dot{r} - (\bar{I}_x - \bar{I}_y)\, pq \end{aligned} \qquad \text{(EQ 14)}$$

Assigning u, v, w, p, q and r as the state variables, the state equations become:

$$
\begin{aligned}
\dot{u} &= rv - qw + F_x/m \\
\dot{v} &= pw - ru + F_y/m \\
\dot{w} &= qu - pv + F_z/m \\
\dot{p} &= \frac{M_x + (\bar{I}_y - \bar{I}_z)\, qr}{\bar{I}_x} \\
\dot{q} &= \frac{M_y + (\bar{I}_z - \bar{I}_x)\, pr}{\bar{I}_y} \\
\dot{r} &= \frac{M_z + (\bar{I}_x - \bar{I}_y)\, pq}{\bar{I}_z}
\end{aligned}
\qquad \text{(EQ 15)}
$$

These six differential equations have a unique solution for a set of initial conditions, therefore the vehicle's motion can be analytically generated. The six state variables are the components of the two vectors $\vec{v}_A$ and $\vec{\omega}_A$. Using equation 6, these two vectors can be transformed to represent velocities in the original (non-rotated) coordinate system W. Integration of the resulting $\vec{v}_W$ and $\vec{\omega}_W$ determines the AUV's position and orientation relative to an earth-based, inertial frame.

At this point, the framework is set for a designer to specify the derivation of the forces and moments. These consist of forces and moments due to gravity, buoyancy, hydrodynamics, and control surface inputs. Since there are many derivation methods for these forces and moments, the framework does not impose any further constraints. Essentially, this is where the development of the vehicle dynamics begins.

2.2.2 Internal And External Sensors Block

The model differentiates between *internal* and *external* sensors. Internal sensors are those which sense some aspect of the AUV's state while external sensors monitor some aspect of the environment. When editing the block diagram model in EASY5, this division becomes more than a formality - internal sensors have no connections to submodels outside the vehicle while external sensors connect to external submodels only. Since the position of the vehicle is always relative to a reference frame, the internal sensors are not truly excluding the environment, but the definitions work well for this development.

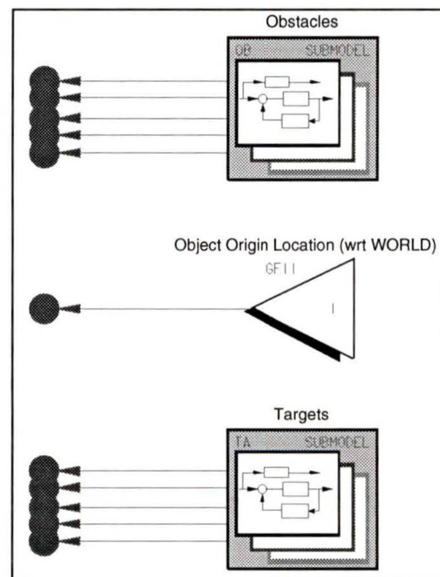
2.2.3 Intelligent Controller Block

The block representing the intelligent controller is only an interface to the true controller (a separate entity described in Chapter 3). This isolates the dynamics model from the controller and maintains a modular structure. The Fortran code in the Intelligent Controller block simply reads a control signal file, passes the values to the AUV Dynamics block and writes sensor data to a file which the intelligent controller (a separate module) interprets.

2.3 Objects Submodel

The Objects submodel contains the obstacles, targets and an object origin whose axes are parallel to the world origin. Currently, the model supports up to five obstacles and up to three targets. Obstacles are boxes defined by an initial location with respect to the object coordinate system (O), a three-dimensional linear velocity and x, y and z dimensions. Obstacles have a fixed orientation in order to simplify transformations between O,W and A. Targets resemble hollow cones with an initial angular and linear position and an angular and linear velocity (all relative to O). Figure 8 illustrates the components in the Object submodel.

FIGURE 8. Object Submodel Contents



2.4 Using The Submersible Design Tool In Simulations

After making changes to the submersible model, the developer will want to test the changes. At this point, the Medium and Object blocks become significant. Together, the three submodels are compiled into one executable Fortran program. In a simulation, the

medium and object submodels form the "environment" with which the submersible interacts.

During a test mission (in the mission execution stage), the intelligent motion controller, implemented in C++, and the submersible design tool (now representing the physical world), coordinate each other in lockstep to give the appearance of real-time operation.

Currently, the incremental time (T_{INC}) for the entire physical model is 0.1 seconds. Thus, with each activation, the physical model simulates one tenth of a second of the physical world. This value was arrived at by compromising simulation speed with accuracy in vehicle dynamics. T_{INC} below 0.1 seconds tended to result in overly slow simulations, while T_{INC} over 0.1 seconds did not allow the controller to read sensor data (or update the control signals) frequently enough.

Controller response times in the real-world vary according to the controller complexity however response times on the order of micro-seconds would be likely. The fact that the controller in the simulation updates more slowly than this indicates that real-world performance would probably be better than that of the simulated performance.

Under UNIX, each coded program is an independent entity called a *process*. After the intelligent controller process has generated the actuator signals for the vehicle, it stores them in a file and activates the physical model process (the compiled Fortran program). The physical model process, on activation, "freezes" the intelligent controller process and retrieves the actuator signals. It then runs the physical model for a simulated tenth of a second. Sensor data is stored in a file. After simulating a tenth of a second, the physical model reactivates the intelligent controller process. The intelligent controller process subsequently "freezes" the physical model (stopping simulated time), reads the sensor data from the physical model and deliberates. After a simulated 0 seconds, it reactivates the physical model process with new actuator signals. This cycle continues until the AUV completes the test mission.

Variations of the above interactions are possible and depend mainly on the architecture of the intelligent controller.

Chapter 3 Intelligent Motion Controller Design Tools

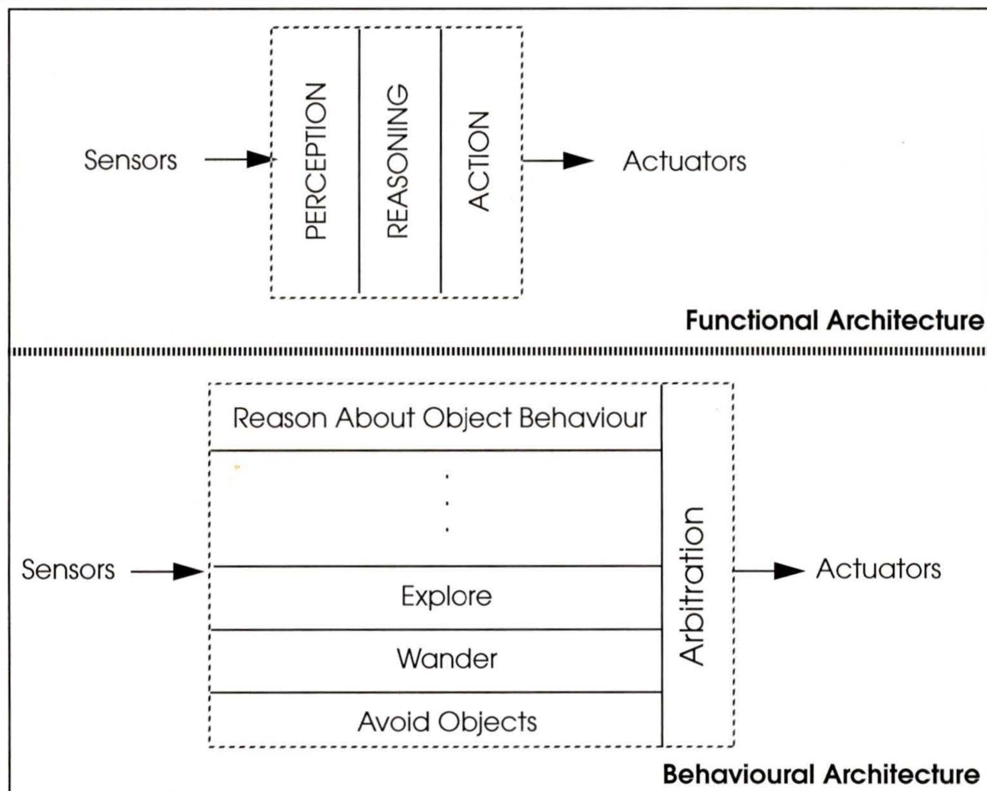
The design tools consist of a Perception/Reasoning/Action Module, a Fuzzy Inference Engine, a Map and a Mission Manager. This chapter examines the characteristics of intelligent controllers and describes how the design tools take advantage of similarities between them.

3.1 Background

3.1.1 Perception/Reasoning/Action Module

Brooks' seminal paper argues that traditional functional architectures (illustrated in Figure 9) are inefficient because each stage in the system must process new information, whether or not it is related to the current task [Brooks 86]. He proposes a new paradigm for an architecture based on *behaviour modules* which operate in parallel, processing only the data which is relevant to them. The net amount of processed data is still the same but the job of processing is distributed among behaviour modules. Figure 9 illustrates the characteristics of the two extremes.

FIGURE 9. Functional And Behavioural Controller Architectures



Subsequent to Brooks' paper, new controller architectures have generally fallen into a spectrum whose bounds are the enhanced functional[7] and behavioural architectures [Randell 93][Hall 92]. Most common are hybrid versions which attempt to combine the best features of both architectures.

Since the research in this thesis aimed to provide a framework for testing various control techniques and architectures, the challenge was to create tools that were abstract enough to apply to the design of any architecture yet concrete enough to implement in software. Unfortunately in control systems, abstraction (implying generalization) and implementation (implying customization) often conflict.

The solution to this dilemma is to identify the underlying similarity between the behavioural and functional architecture: In the behavioural architecture, each behaviour is composed of three tasks - perception, reasoning and action. Thus, **each behaviour is a mini-replica of a functional architecture**. The three tasks can be grouped together into a Perception/Reasoning/Action (PRA) module. Between different architectures, the only variables are the number of modules, their scope, and the nature of their interactions.

Immediately, the object-oriented[8] paradigm appears feasible. If it is possible to objectify the structures and functions of a single PRA module, then it is also possible to produce an object-oriented tool which supports research over the entire spectrum of intelligent controller architectures.

Having established that all intelligent controller architectures share the PRA module as an elemental component, the next step is to objectify the tasks of perception, reasoning and action. Attention now focuses on a lower level of abstraction which reveals commonalities within each task in the module.

Perception Task

Key functions of the Perception task are to acquire data from sensors, fuse it with any other relevant data, and update dependent objects. Intelligent controllers implement these sub-tasks to a greater or lesser degree, depending on hardware and architectural specifica-

---

7. Enhanced functional (or hierarchical) architectures replace the reasoning module in the functional architecture with a chain of modules which handle the "mission management" aspects such as fault tolerance, contingency planning, prediction, etc.

8. The object-oriented paradigm is one way of classifying knowledge. A class consists of data structures (structural knowledge) and functions which operate on those data structures (procedural knowledge). An *object* (or instance) is an entity declared from a class type. It has a state and an identity.

tions. For instance, sensor fusion is only a concern if the Perception task has access to more than one sensor. The Perception task in the PRA module contains template functions for each sub-task mentioned.

Reasoning Task

The Reasoning task assesses the current situation and prepares a response, updating objects accordingly. While these sub-task headings are very general, they capture the essential behaviour of the Reasoning task. Code within a "situation assessment sub-task" varies between Reasoning tasks, however it is always present in some form. The PRA module provides the space for this code and a framework which allows it to interact with other sub-tasks.

Action Task

Common sub-tasks of the Action task are: error signal calculation (indicating the vehicle's deviation from a goal state), control signal generation to minimize the error signals, and actuator control. These are standard functions of any low-level controller. The Action task (within the PRA module) offers a template for expansion of these low-level functions. Again, implementation details will differ between different Action tasks.

Inter-Task Coordination

To review, a single PRA module contains the functions and data structures which make up a Perception task, a Reasoning task, and an Action task. The previous sections define the components of the tasks without resolving the issue of inter-task communication. Inter-task communication includes inter-module communication because different tasks across modules may also communicate with each other. As with architectures, there is a spectrum of communication options ranging from indirect to direct.

Indirect task communication implies that tasks communicate with each other through a secondary entity. One example is a blackboard-based approach [Hall 92]. Here, tasks exchange information through shared data structures (the blackboard). Apart from possible data dependencies, there are few constraints imposed on one task by another. An essential aspect of the PRA design tool is that it provides a mechanism for indirect communication among tasks. Indirect communication alone may be inadequate for architectures with stricter protocols (such as master-slave or lockstep configurations). These may require direct inter-task communication.

Direct communication implies that tasks have a queued message-passing capability. Thus, one of the desirable features of the PRA module design tool is the ability to set up queues between tasks. For a behavioural architecture, Bellingham cautions that the combinational complexity of the software increases with the square of the number of interconnections between behaviours (or PRA modules) [Bellingham 90]. This indicates that although the PRA modules support queues, they should be used sparingly.

3.1.2 Knowledge Bases

A knowledge base refers to a collection of structural and procedural knowledge related to a system. Structural knowledge is knowledge about the underlying structure of a system. Procedural knowledge is knowledge of the behaviour and functions of a system. The term knowledge base thus applies to many different types of systems. Maps, classical control and signal processing systems, expert systems and neural networks can all be considered as knowledge bases.

An examination of current intelligent motion controllers leads to the conclusion that intelligent motion control systems rely on two distinct types of knowledge bases: those which maintain and operate on a physical representation of the outside environment (maps) and those which encode and infer information about processes in general [Hall 92]. The environment therefore provides two distinct knowledge base tools for use in the design of intelligent motion controllers. A single controller can incorporate multiple versions of the tools.

The map design tool encodes structural and procedural knowledge about the world outside the AUV. Encoded structure can have any form from abstract representations of obstacles and targets to image-enhanced representations of edges and vertices. Depending on the purpose of the map knowledge base, its encoded procedural knowledge may vary from high-level functions such as path planning or lower-level (though equally complex) functions such as edge-detection. A design tool which encompasses the fundamental characteristics of a map knowledge base is therefore essential. The designer is free to interface the map design tool with the PRA modules in any way.

In addition to maintaining a map knowledge base and making decisions based on it, an intelligent controller will make decisions about general processes. For instance, the Perception task in a PRA module may decide how to fuse data, while the Action task in the same PRA module may determine appropriate control signal magnitudes to null error signals. Thus another central design tool is a decision-making engine that produces an output

(based on logical rules) when presented with a set of inputs. An intelligent controller, expected to deal with vague and varying information, requires a method that is capable of dealing with uncertainty. This suggests fuzzy knowledge representation and inference as a suitable[9] candidate for a generalized decision-making design tool [Lee 90] [Kosko 91]. The Fuzzy Inference Engine (FIE) is a design tool which is able to encode and act on knowledge about *any* process. Again, the method of interaction between the PRA module and the FIE is up to the designer.

According to Hall's taxonomy, the map and fuzzy inference engine knowledge bases can be viewed as a collective Knowledge Base [Hall 92]. Both the map and inference engine are configurable as globally accessible or locally accessible knowledge[10]. As well, both knowledge bases can assume the role of a blackboard for indirect communication between tasks.

3.1.3 Mission Management

Most intelligent controllers require some type of encompassing "mission manager" to initiate and coordinate the activities of the PRA modules. Mission management tends to be the most abstract feature of intelligent controllers. Depending on the architecture, mission managers can be any combination of arbitrators, coordinators, initializors or supervisors. The design tool offered here is simply a file which is capable of linking and accessing any controller entities.

3.1.4 Summary

The intelligent controller design tools consist of:

- A Perception/Reasoning/Action Module
- A Knowledge Base for the Physical Environment - Map
- A Knowledge Base for General Processes - Fuzzy Inference Engine (FIE)
- A Mission Manager

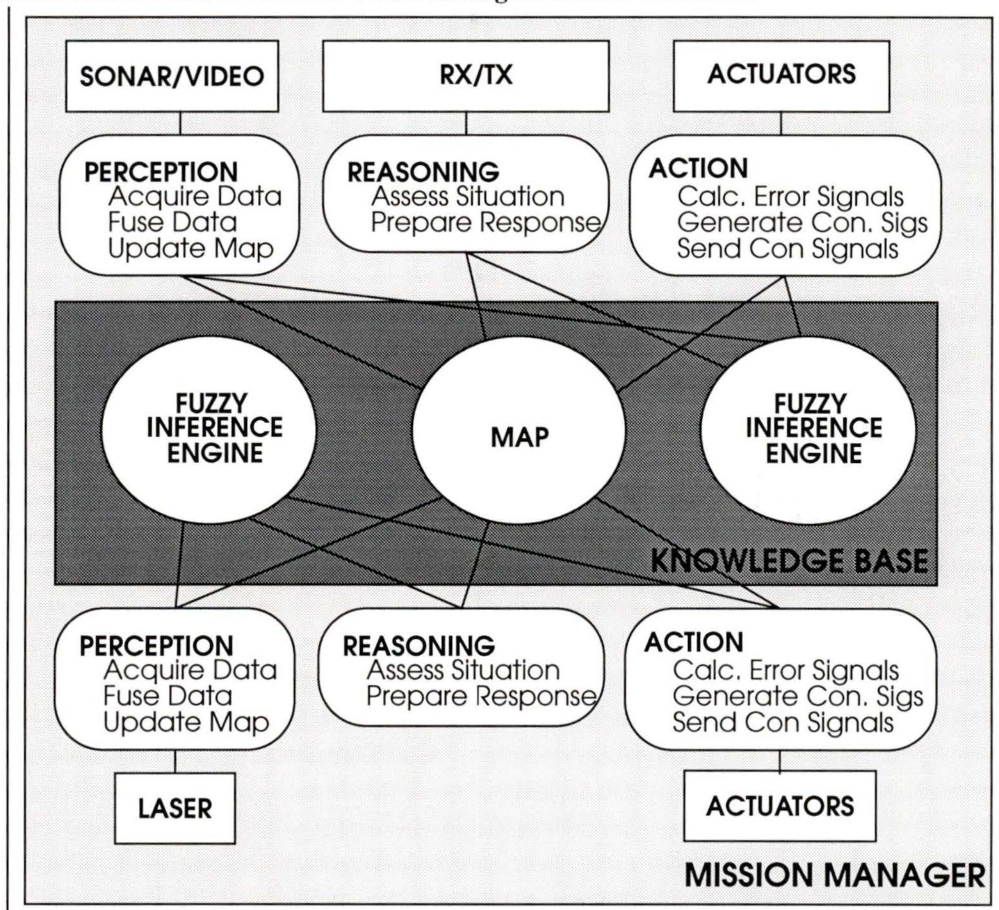
Sensors and actuators are not part of the intelligent controller but the Perception and Action tasks provide interfaces to them. Figure 10 illustrates a possible configuration for the intelligent motion controller showing two PRA modules with locally accessible fuzzy

9. The fuzzy logic-based design tool attempts to be useful in as many decision-making scenarios as possible. In this virtual environment, designers who prefer other methods must create them.

10. *Globally accessible knowledge* is knowledge that all controller functions can access. *Locally accessible knowledge* is accessible only by a specific function [Hall 92].

inference engines and a globally accessible (shared) map. The two PRA modules are analogous to two behaviour modules using the mission manager as an arbitrator. This is just one example of an intelligent controller designed using the tools.

FIGURE 10. A Possible Instance Of An Intelligent Motion Controller

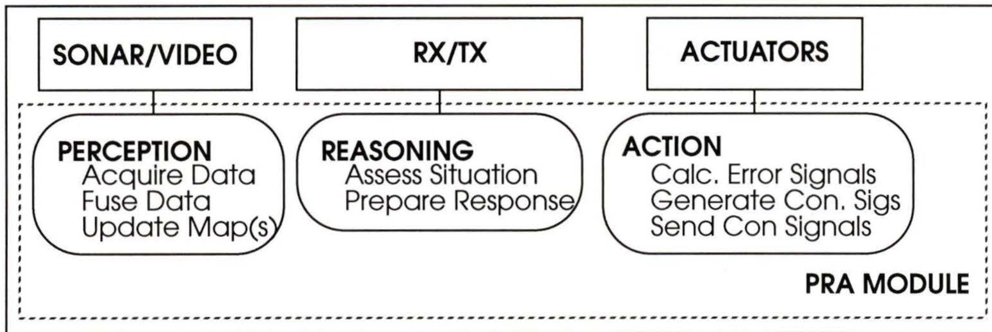


All the design tools for the intelligent motion controller are implemented using AT&T C++, Release 2.1. The perception, reasoning and action task classes within the PRA module take advantage of an object-oriented task library (described in the following sections).

3.2 Perception/Reasoning/Action Module

The PRA module consists of Perception, Reasoning and Action task *objects* (illustrated in Figure 11).

FIGURE 11. Task Classes Within The PRA Module



Each task object is an instance of a task class. For instance, the Perception, Reasoning and Action task objects shown in Figure 11 are instances of Perception, Reasoning and Action task classes, respectively. AT&T's C++ Library Reference describes the rationale for task classes and their features:

> "Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Each activity, here called a *task*, has its own locus of control, a program to execute, and its own private data. Tasks can communicate by explicit sharing of data, by messages, or by data pipes." [AT&T 89]

This description matches the characteristics of perception, reasoning, and action tasks. All three are relatively independent activities, communicating to achieve a common goal - a behaviour. The task library also simulates parallelism and supports interrupts so that tasks can respond to real-time events. An important limitation on task classes is that they must be an absolute base class[11].

Task classes allow only one level of derivation, meaning that classes cannot be inherited from classes which have already inherited the task class. Task classes must be referred to distinctly rather than as elements of a larger class. Fortunately, this constraint does not limit the effectiveness of the Perception, Reasoning, and Action tasks. At a software level however, this restriction prevents them from being combined into an single elegant entity (the ideal design tool would be a self-contained PRA class). Conceptually though, the PRA module is still valid and it will be continually referred to.

To allow for multiple PRA modules, the three task classes must contain *all* data structures and member functions[12] relevant to the intelligent control system, not just data relevant to

11. A *base class* is a class that other classes are derived from. An *absolute base class* does not inherit any traits from other classes.

a specific behaviour[13]. Each PRA module will have the same set of data structures and member functions but will access a subset of these depending on the value of arguments passed to its constructors[14]. One of these arguments will indicate which behaviour the PRA module is to portray. The value of the argument will determine which subset of member functions and data structures the PRA module will access.

Intelligent controllers designed with these tasks require large amounts of memory because each instance reserves space for all the data in its class. However, in the development system, the emphasis is on producing intelligent behaviour rather than optimizing memory. Once an intelligent controller is at the stage where it can be implemented on a real vehicle, the code can be streamlined and memory requirements reduced.

Since task operation only simulates parallelism, the time required to iterate once through the control cycle will increase approximately linearly with the number of PRA modules. Each task runs sequentially, releasing the processor only when explicitly instructed to or when it encounters a data dependency. Thoughtfully designed missions will reduce the simulation time by testing key parts of the controller while traversing minimal distances.

3.2.1 Perception Task Class

Figure12 illustrates the four primary functions of a Perception task: acquire data, standardize data; fuse data and update map(s). The constructor for the Perception task is a loop which calls these four functions sequentially (parallelism is not required here). Although the complexities of each function may vary, every Perception task processes sensory data in this manner.

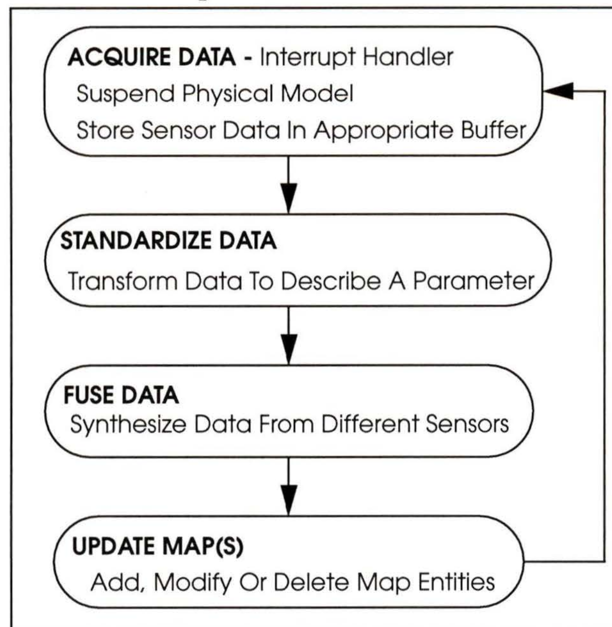
---

12. A *member function* is a function declared within an class definition [Borland 90].

13. Since a PRA module is the software equivalent of a behaviour module, the two terms are interchangeable.

14. A *constructor* is an operation that creates an instance of a class and/or initializes its state [Booch 91].

FIGURE 12. Functions Of The Perception Task Class



The Perception interrupt handler activates on the SIGQUIT signal from the operating system. Currently, the physical model sends the SIGQUIT signal to the controller when the vehicle has new sensor data. The interrupt handler suspends the physical model, updates sensor data buffers in the controller and informs the perception task that new data is available for processing.

A sample definition of a perception task class for an architecture with three PRA modules (corresponding to Explore, Wander and AvoidObstacles behaviours) is:

```
// PERCEPTION TASK CLASS DEFINITION
//   inherits from task and Interrupt_handler classes
class Perception : public task,public Interrupt_handler
{
 private: // declare all data structures
  // declare a pointer to 2 globally accessible maps
  Map *ObstacleTarget_Map("otMAP");
  Map *Image_Map("imageMAP");
  // declare 3 locally accessible FIEs for Sensor
  // Fusion for 3 different behaviours (explore,wander
  // and avoid obstacles)
  FIE ExploreFuseData_FIE("exploreFIE");
  FIE WanderFuseData_FIE("wanderFIE");
  FIE AvoidObsFuseData_FIE("avoidobsFIE");
  // declare a queue tail for the AvoidObs behaviour
  // which sends messages to the AvoidObs Action task
```

```
    qtail *ObstaclePresentQ;
    // declare structures for the vehicle sensors
    Sensor LookaheadSonar("lasSENSOR");
    Sensor Laser("lSENSOR");
    Sensor Camera ("camSENSOR");
    Sensor Compass("comSENSOR");
    Sensor Gyro("gSENSOR");
  public: // declare member functions
    Perception(int BehaviourID, Map *otMap, Map *imap,
    qtail *opQ);              // perception constructor
    void interrupt();         // interrupt handler
    int ExploreStdizeData();// Explore-data stdize fcn
    void ExploreFuseData(); // Explore-data fusion fcn
    float *ExploreMapUpdate();// Explore-map update fcn
    int WanderStdizeData(); // Wander-data stdize fcn
    void WanderFuseData();  // Wander-data fusion fcn
    float *WanderMapUpdate();// Wander-map update fcn
    int AvoidObsStdizeData();// AvoidObs-data stdize fcn
    void AvoidObsFuseData();// AvoidObs-data fusion fcn
    float *AvoidObsMapUpdate();//AvoidObs-map update fcn
};
```

The Sensor objects are composed of a character string ID, a floating point data buffer and an integer indicating the buffer length. These objects are not formal design tools but they exist for the designer to experiment with.

It is important to note that the above code defines the Perception task *class* for all PRA modules - it does not declare an instance (object) of the perception task class. As an example of how objects are created from classes, consider the intelligent controller with three PRA modules - Explore, Wander and AvoidObstacles. To declare the PRA modules, corresponding Perception, Reasoning and Action task objects must be declared:

```
// PRA MODULE (BEHAVIOUR) DELARATIONS
//-------------------------------------------------------
// Declare PRA Module for EXPLORE Behaviour
//-------------------------------------------------------
Perception  Explore_Perception (ExploreID,...);
Reasoning   Explore_Reasoning (ExploreID,...);
Action      Explore_Action (ExploreID,...);
//-------------------------------------------------------
// Declare PRA Module for Wander Behaviour
//-------------------------------------------------------
Perception  Wander_Perception (WanderID,...);
Reasoning   Wander_Reasoning (WanderID,...);
```

```
Action       Wander_Action (WanderID,...);
//-------------------------------------------------------
// Declare PRA Module for AvoidObs Behaviour
//-------------------------------------------------------
Perception  AvoidObs_Perception (AvoidObsID,...);
Reasoning   AvoidObs_Reasoning (AvoidObsID,...);
Action      AvoidObs_Action (AvoidObsID,...);
```

This creates three Perception, three Reasoning and three Action task objects and invokes their constructors. The PRA modules contain the same code, however each one executes a different behaviour due to the differing values of the BehaviourID variable.

To illustrate how the same code can execute different behaviours, consider the Perception task objects of the three PRA modules. The Perception constructors (three identical Perception constructors are running) use the "switch" statement to direct flow of control. The code segment to fuse sensor data would appear in the mainline of the Perception constructor as:

```
// Section to call Fuse Data Function
switch (BehaviourID)
{
 case (ExploreID)
 // call fusion fcn associated with Explore behaviour
 {
  ExploreFuseData();
  break;
 }
 case (WanderID)
 // call fusion fcn associated with Wander behaviour
 {
  Wander_FuseData();
  break;
 }
 case(AvoidObsID)
 // call fcn associated with AvoidObs behaviour
 {
  AvoidObs_FuseData();
  break;
 }
}
```

This implementation is similar to the "forking" concept where a process reads a value to see if it is referring to itself. `ExploreFuseData()`, `WanderFuseData()` and `AvoidObsFuseData()` are member functions of the Perception task class defined pre-

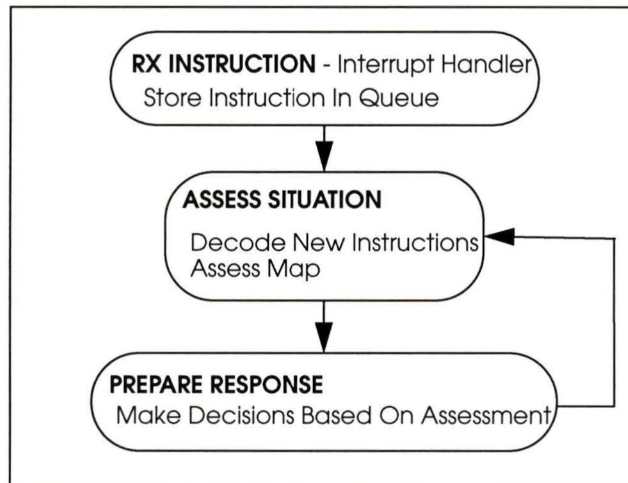
viously. While this example shows how different instances of the Perception task can access different sets of functions, the same principles can be applied to instances of the Reasoning and Action task classes[15].

After building several behaviours or PRA modules, the three task classes will accumulate large libraries of member functions. This continuously growing catalogue of previously coded functions will expedite later research.

3.2.2 Reasoning Task Class

The Reasoning task class, like the Perception task, consists of pointers to maps, FIEs, queues, custom data structures and associated member functions. Figure 13 outlines the functions of the Reasoning task.

FIGURE 13. Functions Of The Reasoning Task Class



Invoked by the SIGINT signal from the operating system, the interrupt-driven Receive function loads the buffers of an Instruction object (not formally supported) and places it on a FIFO instruction queue. Multiple Reasoning task objects will likely designate one instance to process instructions and either set the others to ignore the interrupt or use it for a different purpose.

15. Another way to create multiple task objects which access different sets of functions is to pass pointers to externally defined functions to the task constructor. This approach violates the object-oriented nature of the code since the external functions would not be associated with a particular class but may be more efficient in terms of computation time and memory.

An abbreviated version of a definition for the Reasoning task class might be:

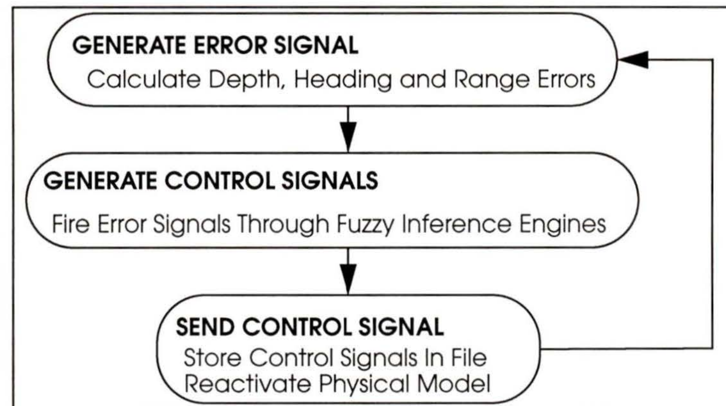
```
// REASONING TASK CLASS DEFINITION
//   inherits from task and Interrupt_handler classes
class Reasoning : public task,public Interrupt_handler
{
 private: // declare all data structures
        ...
   Instruction CurrentInst; // current instruction
        ...
 public: // declare member functions
   Reasoning(int BehaviourID, Map *otMap, Map *imap,
   qtail *opQ);             // reasoning constructor
   void interrupt();        // interrupt handler
   int ExploreAssessSituation();// assess situation fcn
   void ExplorePrepareResponse();// prepare resp. fcn
   int WanderAssessSituation(); // assess situation fcn
   void WanderPrepareResponse();// prepare resp. fcn
   int AvoidObsAssessSituation();//assess situation fcn
   void AvoidObsPrepareResponse();// prepare resp. fcn
};
```

The coding techniques described in the previous section would allow for differently behaving instances (objects) of a Reasoning task class. Code within the `AssessSituation()` and `PrepareResponse()` function depends on the desired behaviour and the specifications of the designer.

3.2.3 Action Task Class

Similar to the other task classes, the Action task class consists of references to maps, fuzzy inference engines, queues, other custom data structures and member functions. Figure 14 illustrates the primary functions of the Action task class.

FIGURE 14. Functions Of The Action Task Class



Since signals are a dominant entity in the Action task, a class of Signals exists for the designer to use. The Signal class contains floating point variables which can be used to represent control or error signals for converting depth, heading and range parameters into rudder, diveplane and propeller speed parameters. A "weight" floating point variable is also included which allows the signal to be weighted for averaging purposes.

Methods of defining, creating objects of and operating the Action task class are similar to those examined in the previous two sections.

3.2.4 Summary

A PRA module is the software manifestation of a behaviour. Since they are classes, the Perception, Reasoning and Action task classes must contain all data and functions relevant to their respective functions within the intelligent controller. *Instances* of the classes (objects) access subsets of these structures and functions depending on the behaviour they are portraying.

An intelligent motion controller may be composed of one or more PRA modules depending on the design specifications. The designer is also free to make a controller without the PRA modules, they are merely supplied as tools.

3.3 Knowledge Base for Physical Environment - Map

Brooks outlines nine principles for the design of mobile robots, two of which are paraphrased:

> The robot must model the world as three-dimensional and map making is ... of crucial importance [to the autonomy of a mobile

> robot] even when idealized blueprints of an environment are available [Brooks 86].

The reasoning behind these assertions is sound. Intelligent behaviours such as learning and planning require an internal representation of the external world. Without one, the vehicle acts with no regard for the state of the environment beyond its immediate surroundings - decidedly non-intelligent behaviour.

Intelligent beings (such as humans) use several maps in their daily lives. Humans for instance, have geographic maps for long-range activities, memorized maps for route planning around buildings or busy streets and a vision-based map for close-range tasks. Whether these maps are discrete (in the case of a road map and mall directory) or continuously linked (the maps in the brain), the fundamental concept is that *different tasks require maps of different resolutions*. Extending this idea to artificially animated vehicles, it becomes evident that the intelligence of an autonomous vehicle depends on having multiple sufficiently scaled maps covering appropriate areas. These maps will either be reusable or easily reproduced and scaled.

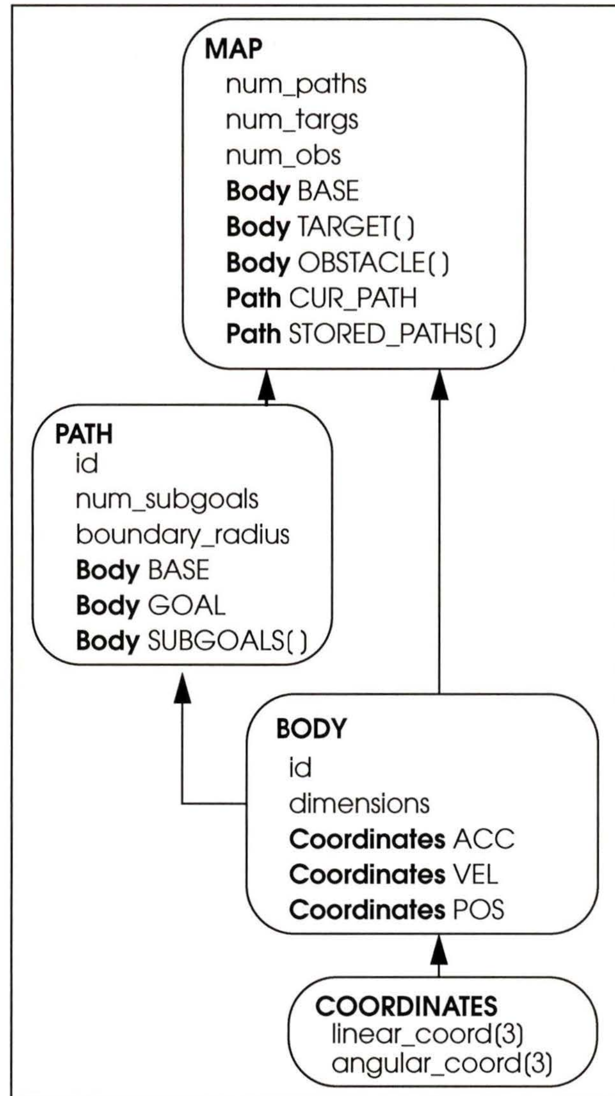
The object-oriented approach is ideal for encoding these internal maps. Different instances of a map *class* will provide maps of varying coverage area and resolution to match the changing characteristics of an intelligent controller's task.

The map design tool provided by the environment is a map at an obstacle/target level of abstraction. Image maps are not explicitly supported but they may be based on some of the characteristics of the existing map tool.

3.3.1 Map Data Structures

To objectify the concept of a map, one needs to determine the common elements of all maps. These structures were identified as fundamental: a base (the AUV), targets (docks), obstacles, and paths. Figure 15 illustrates the derivation of the map class from these structures.

FIGURE 15. Data Structures In The Map Object



The following entries define each class in the map:

Coordinate Class- A class of two 3-element vectors which represent linear and angular coordinates with respect to a frame of reference.

Body Class- A class of entities that are described by a position, velocity, acceleration and possibly dimensions in three dimensional space relative to a frame of reference.

Path Class - A class of sequences of subgoal bodies which terminate at a goal body. Depending on the path planning and path following algorithms, the designer may give the subgoals and goal a velocity if the goal happens to be a moving target.

A path boundary radius turns a straight line path into a path of cylinders whose radii are defined about line segments joining two consecutive subgoals. The boundary radius gives some flexibility to the intelligent controller in generating control signals. For example, if the vehicle encounters a new obstacle which requires a deviation from the current path, the controller can weight the importance of staying inside the path boundary (urgency is proportional to the vehicle's distance from the path boundary) with the "repulsive" effect of the new obstacle.

Map Class - A class composed of instances of a base, obstacles, targets, a current path and stored paths as explained below:

Base Object - An instance of the Body class. The base is usually the AUV however, depending on the application and purpose of the map, the intelligent controller can define any object as the base.

Obstacle Objects - Instances of the Body class. Obstacles are boxlike having dimensions in the x, y, and z planes (identical to the obstacles represented in the physical model in Chapter 2).

Target Objects - Instances of the Body class. Targets are conical having a base radius of 15 metres.

Current Path Object- An instance of a path. The current path is the one that the base is currently following.

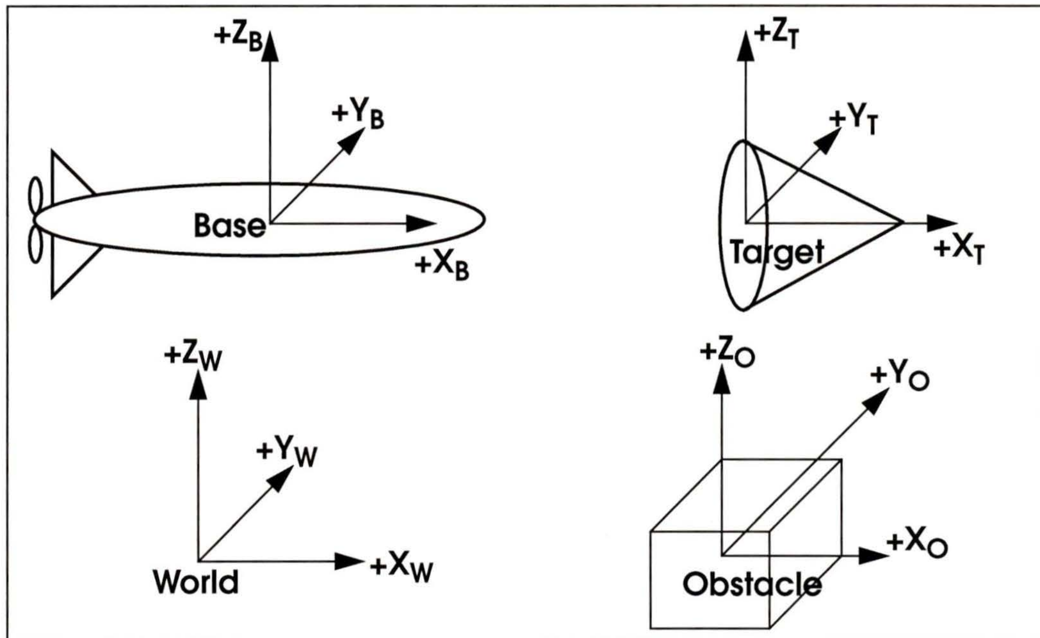
Stored Path Objects- Instances of paths that may be recalled when the base has previously traversed a path from the same start to the same goal. For example, if the AUV routinely traverses the same route, it will not plan a new path for each new traverse. Instead it stores the commonly traversed path, referring to it when necessary. If the map has changed since the path was stored it will be marked as invalid and removed from the array of stored paths.

3.3.2 Map Coordinate System

The map's coordinate system uses an absolute origin with relative locations for all other entities within the map (including the vehicle). This conflicts with Brooks' recommendation to use relational maps (where the vehicle is the sole reference point) to minimize cumulative errors [Brooks 86]. Future work may change this aspect of the map however, it

was simpler to begin with an absolute coordinate system. Figure 16 illustrates the coordinate system for the map class.

FIGURE 16. Coordinate System For The Map Class



The base origin is at the mass center of the base (the AUV in this case). The obstacle origin is at the geometric center of the obstacle. The target or dock origin is in the plane of the base of the cone-shaped dock. The coordinate system allows all entities in the map to have non-zero angular coordinates relative to the world origin. With the map structures and coordinate systems defined, the discussion turns to map class member functions. These are explained in the following sections.

3.3.3 Map Indexing

An essential feature of effective maps is that they contain an index function. The index function, given an identifier, searches the map for the entity corresponding to the identifier. If the entity exists, the index function indicates that the requested object is present, otherwise it indicates that the map does not contain the object.

The preliminary step in path planning (or map analysis of any kind) is to consult the map index and ensure that the objects of interest are present. Map analysis without an index is inefficient.

3.3.4 Transformations

Sensors on board the AUV will return data whose locations in space are relative to the AUV. Before the sensor data can be useful in the map, it is necessary to transform this data to the map's absolute frame of reference. Transformation functions in the map class perform this function. They are of the form discussed in section 2.1.1.

3.3.5 Path Planning

The problem definition in path planning is: given an environment containing a moveable object, a start node, a goal node and obstacles, compute an optimal, collision-free path from the start node to the goal node. Optimality is in the sense of a path characteristic such as distance, time or smoothness. *A path is defined as a series of subgoals or waypoints connected by straight line segments with the start node and goal node as beginning and endpoints in the path.*

Path planning algorithms are classified into graph generation and artificial potential field techniques [Randell 92]. Recent methods in the two classes map the real world to a *configuration space* (Cspace) before planning a path. Akman describes the rationale for configuration space [Akman 87]: "...the Cspace method maps an object movement problem to a point movement problem by simultaneously growing the obstacles in the workspace and by shrinking the object to be moved to a *reference point.*" The path planning problem is thus reduced to one of either graph searching or steepest-descent optimization (depending on the technique).

Currently, the path planner supplied with the Map class is a simplified version of the Tangent Graph (T-Graph) technique [Liu 91]. The T-Graph approach is based on the Visibility Graph (V-Graph) technique [Lozano 79]. The V-Graph method requires the map to model all obstacles as convex polyhedra so that in Cspace, the planner can characterize them by their vertices. After linking *visible* vertices[16] together, the planner connects the links to produce a graph of linked nodes which form many paths including the shortest path whose end node is the goal.

Liu outlines the deficiencies of the V-Graph method [Liu 91]. The V-Graph method requires memory space proportional to the square of the total number of vertices in the map. As well as taking up space, this lengthens the time required to search a completed V-

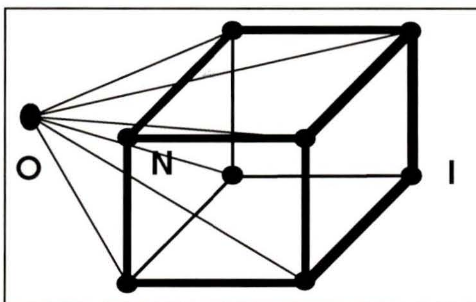
---

16. There is no obstacle boundary between two vertices which are *visible* to each other. Lee provides a more rigorous definition and a method for determining the visibility of any 2 points [Lee 83].

Graph. The V-Graph also imposes limitations on obstacle representation by requiring all obstacles to be modeled as convex polyhedra. In comparison, memory requirements in Liu's T-Graph are proportional to the number of convex segments of obstacle boundaries in the map. Obstacles can also have concave and curved surfaces instead of being restricted to convex polyhedra.

Although the Map class represents obstacles as convex polyhedra (boxes), the path planner uses the T-Graph concept of tangentiality[17] to ignore vertices that it would otherwise consider (if it were based on the V-Graph method). Figure17 illustrates tangentiality and visibility as they apply to obstacles in the Map class. From point O, vertex I is the only non-visible vertex on the obstacle because there is an obstacle boundary between it and point O. Vertex N is non-tangential because a line to it (and extending past it) from Point O intersects an obstacle boundary near point N (after point N, the line is inside the obstacle). Vertex I is also non-tangential but is ignored because it is invisible to point O.

FIGURE 17. Tangential Points On An Obstacle



To plan a path from a start node to a goal node, the planner in the Map class first reduces the environment to Cspace, builds a connected graph of tangential, visible links and then searches the graph to find the minimum distance path to the goal node. Reducing the environment to Cspace requires "growing the obstacles" - increasing obstacle boundaries to account for the vehicle size and dynamics - and "shrinking the vehicle" - reducing the vehicle to a point which can travel (without deviation) along a line segment. Significant complexities arise when growing obstacles to account for vehicle dynamics [Brooks 83]. The current planner compensates for these intricacies simply by growing the obstacles by a large safety margin - obstacle dimensions are increased by 60 metres. Obstacles are assumed to be spaced far enough apart so that grown obstacles will not overlap. If they do overlap, the path planning algorithm may break down. The reason for this is that the visi-

17. If a line contacts the boundary of an obstacle at point P but it does not intersect an obstacle boundary near point P, the line is said to be tangent to the obstacle at point P and P is called a tangent point [Liu 91].

bility of links inside an obstacle is undefined and depends on whether the algorithm treats obstacles as hollow or solid. This problem could be solved by treating all obstacles as hollow and allowing links originating from inside an obstacle and terminating at a grown vertex to be visible.

Potential paths to the goal are composed of a series of nodes (grown obstacle vertices) connected by straight lines. The waypoints of the chosen path become subgoals for the controller.

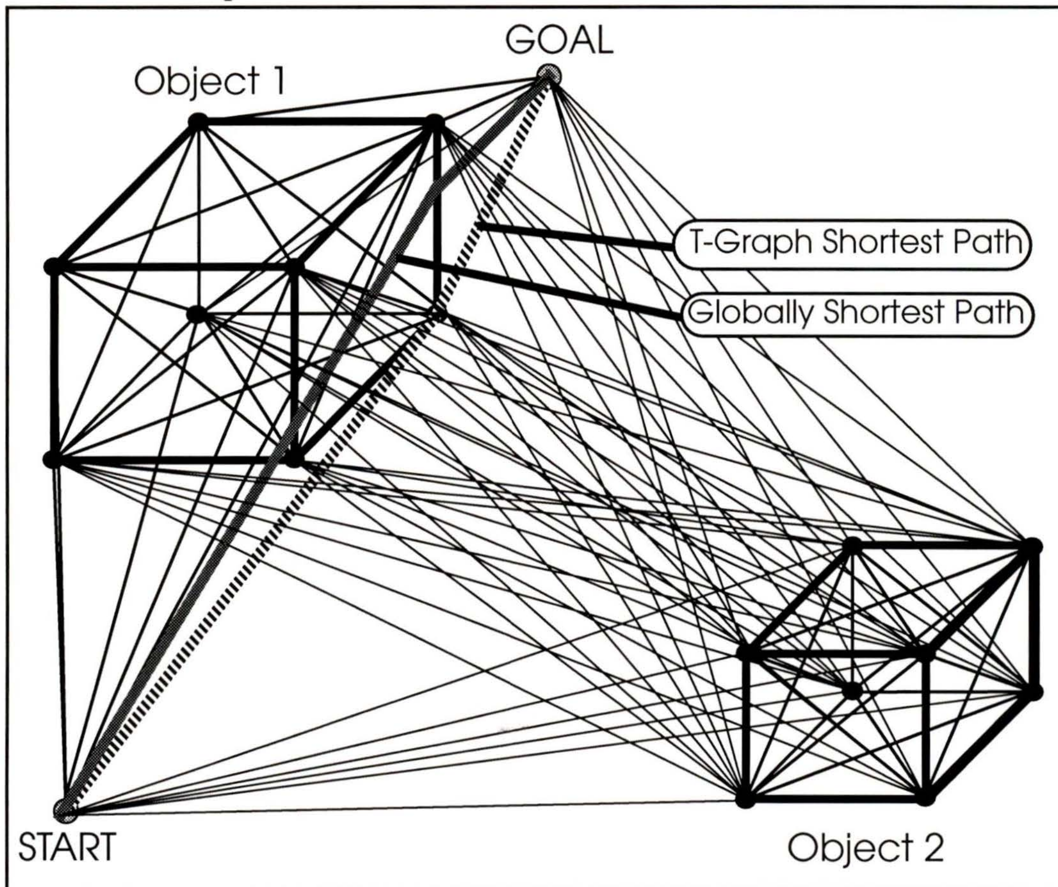
In planning a path, the planner compensates only for stationary (or static) obstacles. Al moving (or dynamic) obstacles are ignored because a planned path which includes dynamic obstacles will eventually become sub-optimal as the dynamic obstacle move out of the path. The processing time required to plan a path which includes dynamic obstacles is thus wasted. It makes more sense to plan paths around static obstacles and let the Action class take care of dynamic obstacle avoidance.

An enhanced path planner could incorporate predicted obstacle movements into its calculations. For instance, an advanced planner might plan a path around an obstacle moving in a fixed direction by estimating where the obstacle is likely to be at specific times and treating it like a static obstacle at those times. Moving targets would requires similar calculations. These additional calculations would significantly increase the time to compute possible vehicle trajectories and obstacle and target locations.

Graph-based techniques are not ideal for three-dimensional path planning because links of the globally shortest path may run across the surfaces of obstacles without intersecting *grown* obstacle vertices (the graph-based techniques generally do not consider non-vertex points) [Lozano-Perez 79]. Therefore these methods cannot guarantee to find the absolute shortest path in a three-dimensional space. However, previous research had sparked an interest in designing a graph-based planner capable of finding shortest global paths using projections on obstacle surfaces of the straight line between the start node to the goal node. To assist future work in this area, the graph-based planner was developed.

Figure 18 illustrates the visible links generated by the T-Graph approach and the shortest path found to the goal node. Note that the path may not be the globally shortest one since the algorithm ignores any planar aspects of the obstacle when generating potential routes. For instance in Figure18, the planner does not find the true shortest path.

FIGURE 18. T-Graph Method



The planner finds the shortest path by searching the connected graph with the A* (pronounced "ay-star") algorithm. The A* algorithm searches a connected graph by concentrating on nodes (representing obstacle vertices) that are *most likely* to be on the shortest path to the goal. Each node in the graph contains an estimate of its distance to the goal node. From its current node, the algorithm chooses the node whose distance estimate, h, indicates that if it is on the path, the path is the shortest one from the current node. The distance estimate, h, is the straight-line distance from the current node to the goal node, which guarantees that the A* algorithm will find the shortest path between the start node and the goal node[18].

3.3.6 Path Validation and Path Boundary Violation Functions

Path validation provides a method for determining whether new map data invalidates a previously planned path. This simple routine checks each link in a path to ensure that it is

18. This is known as the *admissibility* of the A* algorithm [Tanimoto 90].

still visible. If any link has become invisible due to a change in the map, the path is declared invalid, otherwise the path is valid. The path validation algorithm ignores previously traversed links.

During the traversal of a pre-computed path, the vehicle is likely to stray off course due to environmental disturbances and/or vehicle performance limitations. The path boundary violation function checks to see that the vehicle is within the path boundary, defined by a radius about a line segment joining consecutive subgoals. If the vehicle has violated the path boundary, the function returns a violation.

3.3.7 Map Updating

Updating a map consists of adding, modifying or deleting a map entity. The Map class provides functions to handle each case. The Perception task object of a PRA module usually updates map entities such as the base (vehicle), docks, and obstacles. Path objects however, are usually updated by a Reasoning task object. Function arguments to the map updating functions consist of the entity ID, the modified property, and if necessary, the new value.

3.4 Knowledge Base For General Processes - Fuzzy Inference Engine

Having determined that knowledge bases are common to all intelligent controllers, attention turns towards choosing an implementation of a knowledge base to offer as a design tool. Candidate implementations can be evaluated according to the requirements for knowledge bases in an intelligent controller.

A knowledge base tool which encodes knowledge and infers conclusions about general processes for an intelligent controller should meet the following criteria:

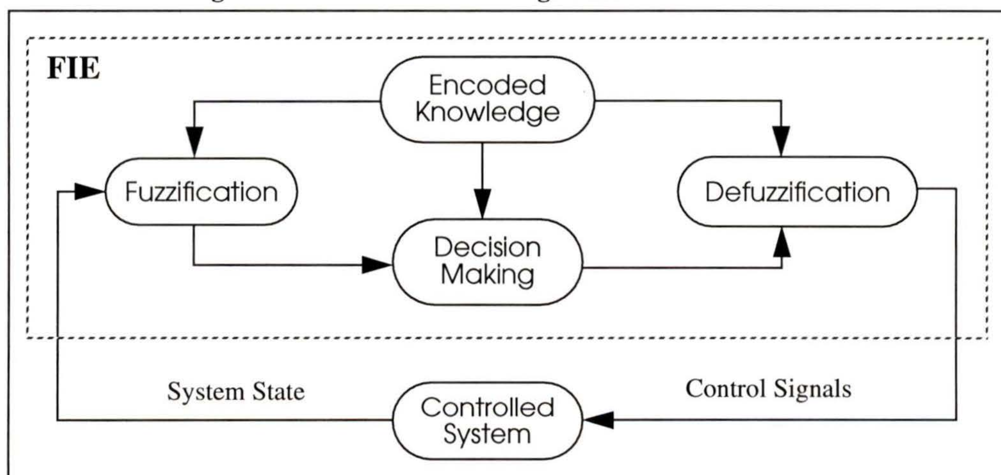
- The intelligent controller has to make knowledgeable decisions in an environment which is full of incomplete information, disturbances and varying conditions.
- The knowledge base tool must be applicable to a wide variety of processes.
- Many processes will be complex and difficult to model. The knowledge base tool should allow for model-free encoding and inference.
- The knowledge base tool should be quickly configurable so that a designer does not spend an inordinate amount time creating a knowledge base.

Among some of the possible implementations, expert systems, neural networks and fuzzy logic-based methods stand out. Model-based methods such as classical control systems cannot cover the wide variety of systems which may require encoding. Adaptive methods are possible but may also have problems incorporating all conceivable processes.

Fuzzy methods of control and analysis are useful when the system to be controlled or analyzed is difficult to model mathematically (Kosko gives a detailed treatment of fuzzy logic and its applications [Kosko 91]). Rules can be encoded quickly in contrast to neural networks where "rules" require long training periods before being reliably encoded. Expert systems tend to work well on symbolic levels (non-numeric) but do not perform well when numerical computation is required. These factors point to a fuzzy-logic based knowledge base as a suitable tool for the development system. This design tool is called a Fuzzy Inference Engine (FIE) to emphasize that the knowledge base does more than encode structural knowledge - it applies encoded procedural knowledge to encoded structural knowledge to infer new information.

The FIE takes crisp (real numbered) input values, converts them into fuzzy values, fires them through a set of fuzzy rules and produces a defuzzified (crisp) output. Figure19 (adapted from [Lee 90]) illustrates the general behaviour of the FIE.

FIGURE 19. Configuration of the FIE Knowledge Base

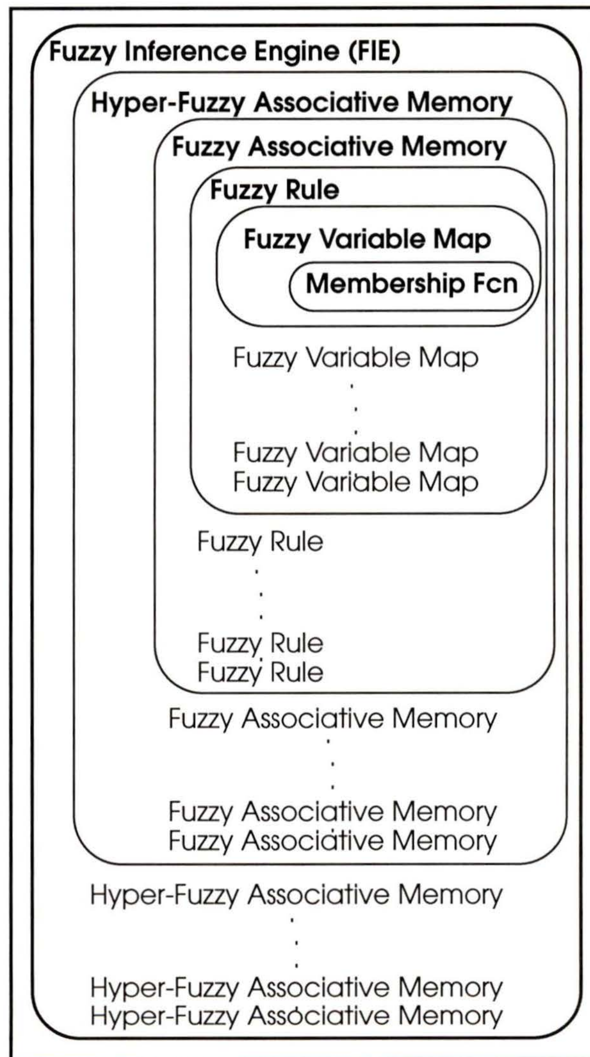


Due to its modular nature, the FIE is a perfect candidate for object-oriented coding. Because it is object-oriented, the FIE can perform a wide variety of duties within the controller where knowledgeable decision-making is required. However, while object-orientedness improves the controller's performance, it complicates a discussion of fuzzy inferencing and control. For this reason, the FIE's structural and functional components

are presented separately. The next sections describe FIE's structural aspects, followed by an examination of the functional aspects.

Figure 20 illustrates the nested nature of structures in the FIE. This figure portrays only the structural relationship between the classes and does not include object functions.

FIGURE 20. Data Structures In The FIE



3.4.1 Membership Function Class

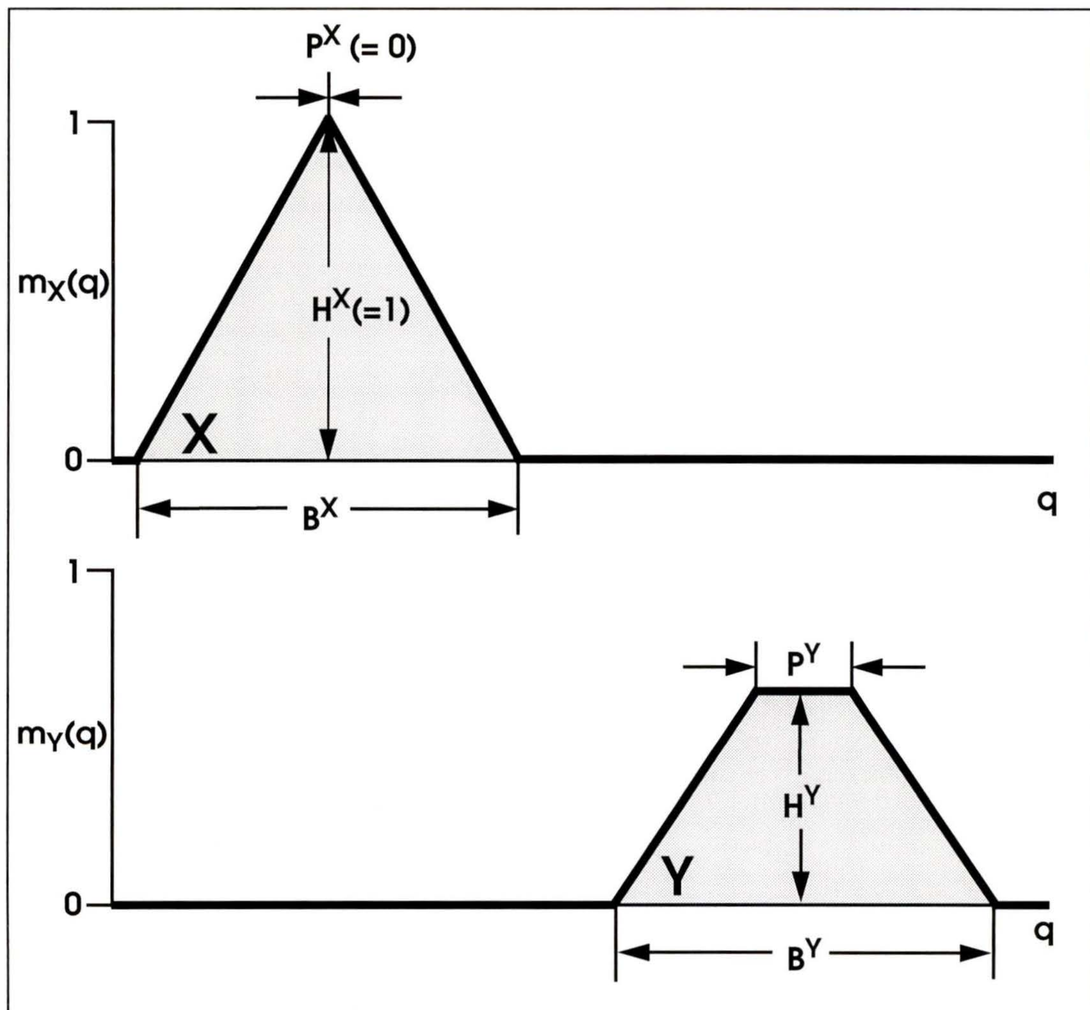
The Membership Function object is at the core of the FIE with all other classes derived from it. A membership function (m(x)) defines the degree to which a range of values of a particular variable (x), belongs to a particular fuzzy set (A):

$$m_A(x) = \text{Degree}(x \in A) \tag{EQ 16}$$

For this work, a fuzzy set is characterized by a continuous membership function over the useful range of a variable together with an identifier that distinguishes it from other fuzzy sets. If the fuzzy set A is on the real line, then $m_A : \mathrm{R} \to [0,1]$. If $m_A(x) = 0$, then x does not belong to the fuzzy set A. If $m_A(x) = 1$, x belongs completely to A. Values of $m_A(x)$ between 0 and 1 imply fuzzy degrees of membership to the set A.

In control problems, membership functions are often represented geometrically using symmetric triangles or trapezoids [Kosko 91]. Figure 21 displays two different membership functions (X and Y) of the variable q. Parameters required by the object are the Peak Width of the function, the Base Width and the Maximum Height. All membership function values outside the Base Width are equal to 0.

FIGURE 21. Fuzzy Subsets

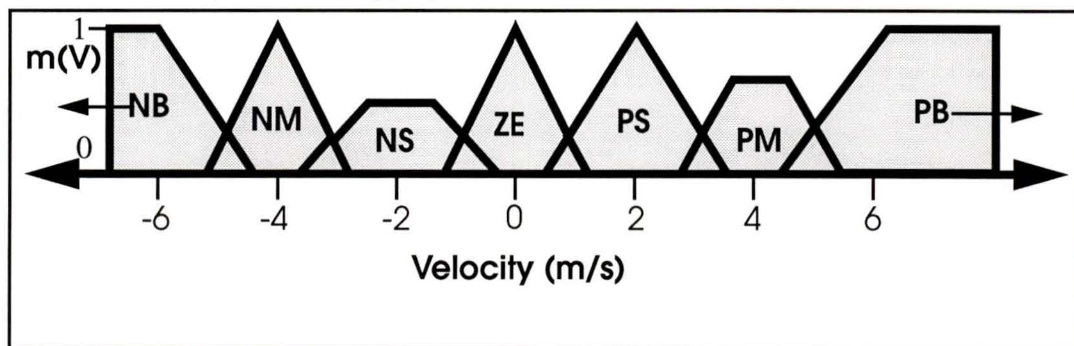


After membership functions have been defined, they can be "upgraded" to fuzzy sets by applying them to a variable. The next section looks at how degrees of membership can be mapped to a fuzzy variable.

3.4.2 Fuzzy Variable Mapping Class

A real-valued variable is mapped to fuzzy subsets by assigning membership functions to various regions throughout the range of the variable. Typically, the non-zero (symmetric triangular or trapezoidal) region of the membership function is centred about a representative value in the range and assigned an identifier which indicates the qualitative aspect of the function. The non-zero areas of adjacent fuzzy sets should overlap by approximately 25% [Kosko 91]. Figure 22 illustrates the concept of a fuzzy variable mapping.

FIGURE 22. Fuzzy Variable Mapping



In the above figure, the identifier **PM** represents the fuzzy set **Positive Medium**, whose centroid (centre of the symmetric triangle) is at 4 m/s. Other identifiers are similarly configured. Fuzzy sets at the extremes of the variable's range extend to infinity. The FIE functions take this into account in the calculations as explained later.

3.4.3 Fuzzy Rule Class

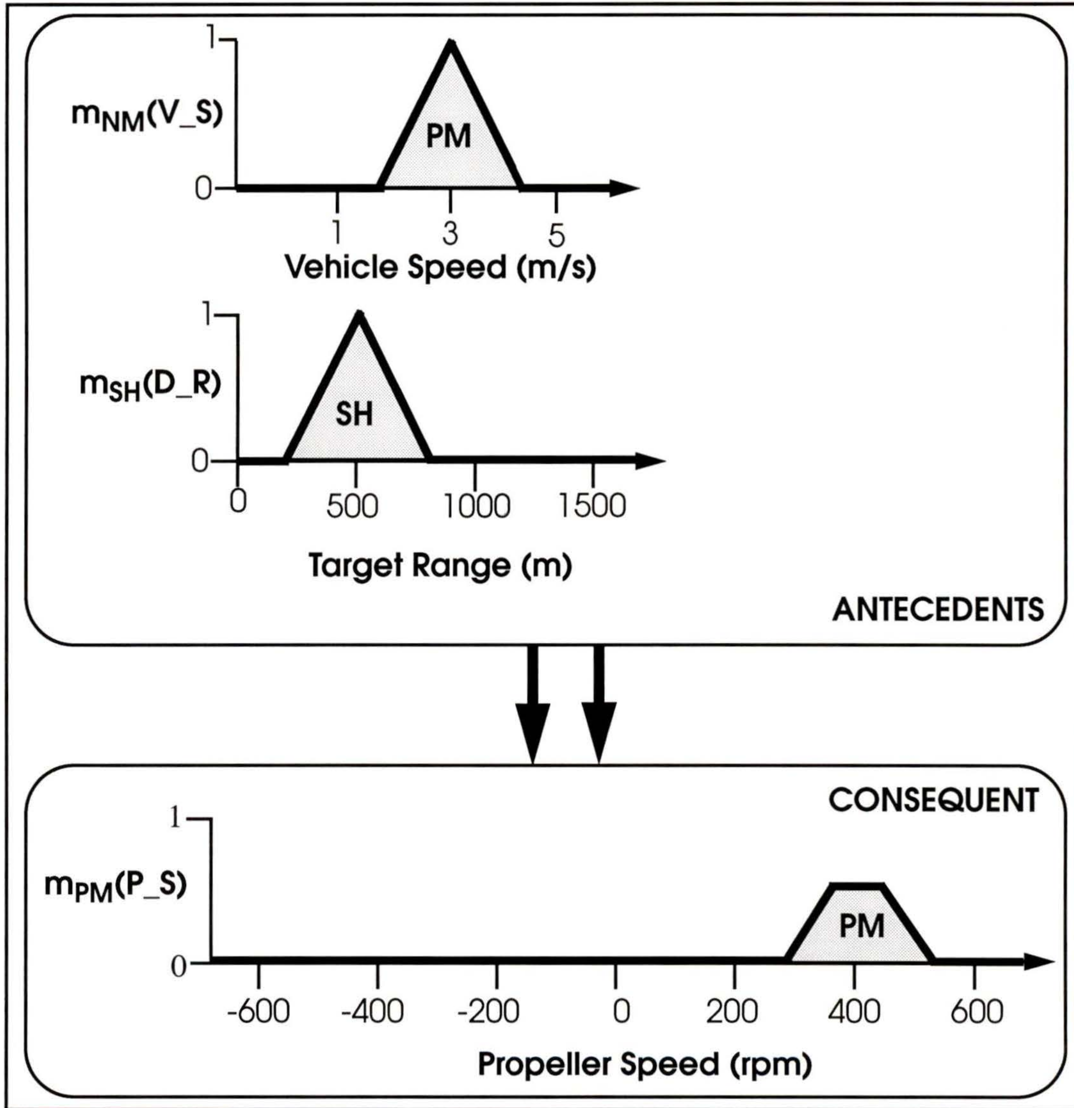
Conceptually, a fuzzy rule encodes imprecise, structural knowledge. Fuzzy rules store common-sense, linguistically represented knowledge for use in a computational manner.

Mathematically, fuzzy rules are mappings between fuzzy sets. The form of a fuzzy rule is as follows: **IF** *<antecedent(s)>* **THEN** *<consequent(s)>*, where the antecedent and consequent clauses consist of one or more instances of: *fuzzy variable = fuzzy set*. Figure 24 illustrates the following fuzzy rule:

> **IF** *Vehicle_Speed = Positive_Medium & Dock_Range = Short*
> **THEN** *Propeller_Speed = Positive_Medium*

This fuzzy rule maps collections of fuzzy sets in the antecedents to a fuzzy set in the consequent. The degree to which the consequent set fires depends on the degree of membership of the inputs (Dock_Heading and Dock_Range) to the antecedent fuzzy sets and on how the antecedent sets are combined (AND or OR). Kosko provides details [Kosko 91].

FIGURE 23. Fuzzy Rule



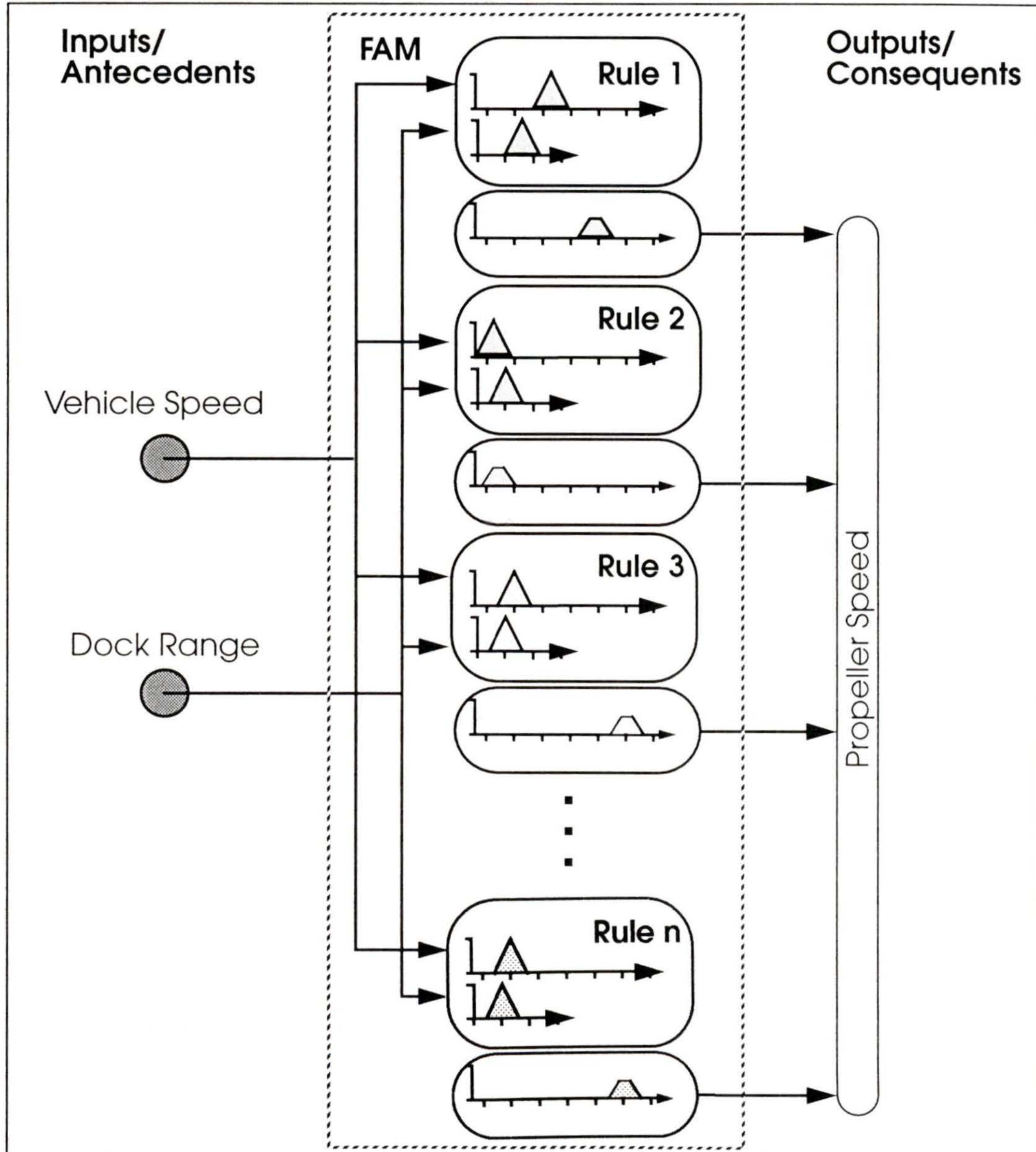
3.4.4 Fuzzy Associative Memory Class

A Fuzzy Associative Memory bank[19] (FAM) is a collection of fuzzy rules sharing common inputs (antecedents) and outputs (consequents). A well-designed FAM contains com-

19. Kosko refers to one rule as a Fuzzy Associative Memory [Kosko 91]. Here, FAM implies a collection of rules. For consistency with Kosko, FAM may also be interpreted as Fuzzy Associative *Memories.*

plete knowledge of the mapping between inputs and outputs. Note that a FAM alone is not a controller, it is simply encoded knowledge and does not provide any of the computational tools required by a fuzzy logic controller. Figure 24 illustrates the FAM concept.

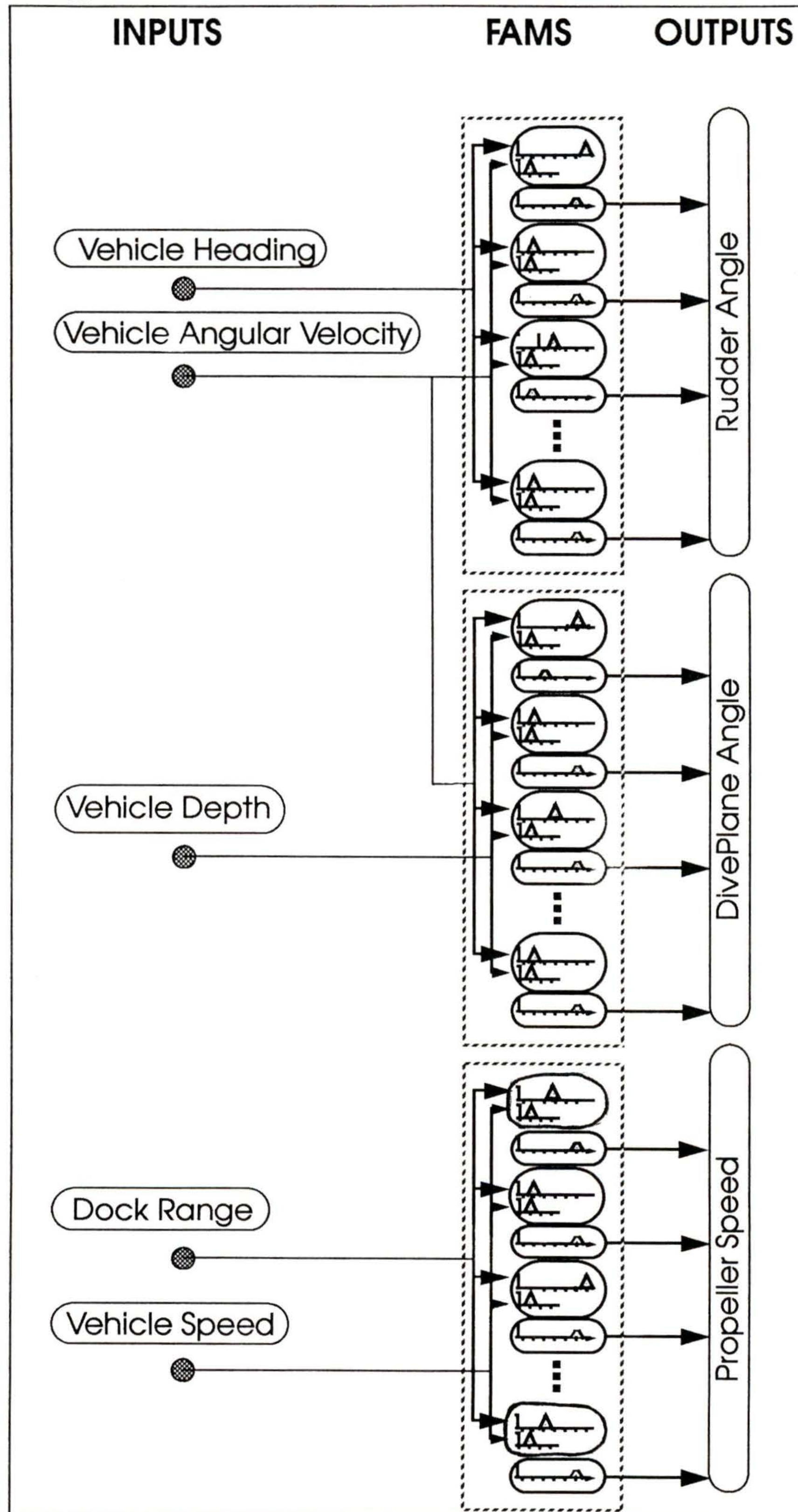
FIGURE 24. Fuzzy Associative Memory



3.4.5 Hyper-FAM Class

A Hyper-FAM is a collection of FAMs which contain all the knowledge needed to completely control a process or a device. Apart from sharing some common inputs, all FAMs within a Hyper-FAM operate independently and thus maintain the FIE's modular aspect. A Hyper-FAM which controls AUV motion might look like the one shown in Figure 25.

FIGURE 25. Hyper-FAM For Motion Control Of An Unmanned Submersible



3.4.6 Fuzzy Encoding, Inference and Defuzzification Procedures

This section examines how the FIE uses the data structures just described to infer knowledge and control or analyse a process.

Introduction

Fuzzy inferencing occurs at the FAM level where fuzzy rules are fired in parallel. New information (output) is "inferred" from the degree to which the current information (input) activates rules within the FAM.

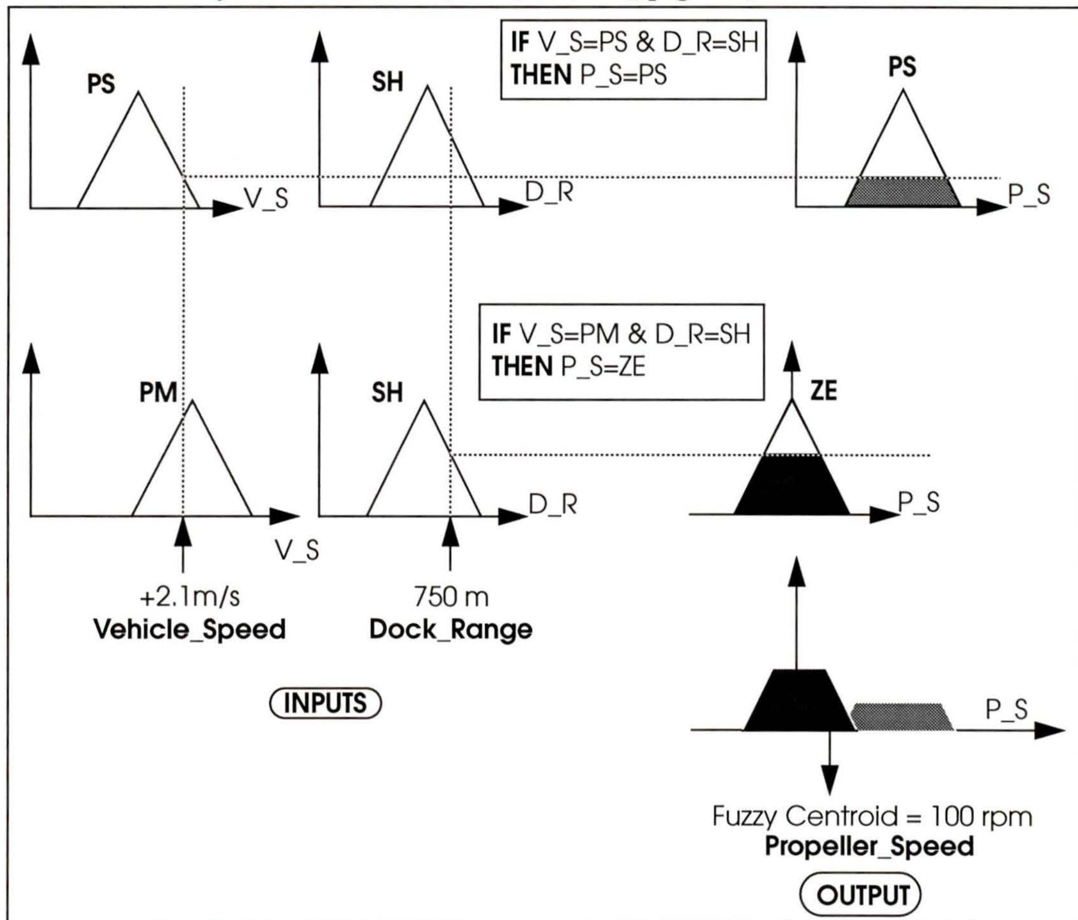
Figure 26 from [Kosko 91], illustrates the fuzzy inferencing process for a simple two-rule FAM composed of double-antecedent, single-consequent rules. FAMs with more rules and higher dimensions operate using principles identical to those shown in Figure 26.

The two rules are:

> **IF** *Vehicle_Speed =Positive_Small & Dock_Range = Short*
> **THEN** *Propeller_Speed = Positive_Short*
>
> **IF** *Vehicle_Speed = Positive_Medium & Dock_Range = Short*
> **THEN** *Propeller_Speed = Zero*

The crisp input values are a forward vehicle speed of +2.1 m/s and a range to the dock of 750 m and the inferred output is a Propeller Speed of 100 rpm.

FIGURE 26. Fuzzy Inference Process (from [Kosko 91], page 321)



Fuzzy Encoding

Fuzzy rules must be *encoded* before information can be inferred from them. Encoding a fuzzy rule requires discretization of the antecedent and consequent variables in the rule. For example, consider the antecedent variable X. The range of X is discretized into *n* elements, so that $X = \{x_1,...,x_n\}$. In applications which use continuous variables, quantization occurs since machine precision limits *n*.

The fuzzy subset A, defined on the variable X, is now a point in the unit hypercube $I^n = [0,1]^n$. Elements in the point A (a fuzzy subset) encode a membership function over the discretized range of the variable X. Referring to Figure 21, the n-dimensional “hyper” point A would represent a discretized version of the bold dark line which represents the membership function.

The matrix M encodes the rule, **IF** *X is A* **THEN** *Y is B* as:

$$M = A^t \bullet B \qquad \text{(EQ 17)}$$

where the • operator implies,

$$m_{ij} = min(a_i, b_j) . \qquad \text{(EQ 18)}$$

The matrix M is defined for the input variable X and the output variable Y. The encoded matrix M can be represented as:

$$M = \begin{bmatrix} a_1 \wedge B \\ \dots \\ a_n \wedge B \end{bmatrix} \qquad \text{(EQ 19)}$$

where $a_i \wedge B = (min(a_i, b_1), \dots, min(a_i, b_n))$

Therefore, rows in M are clipped sets of B corresponding to elements in the fuzzy subset A. In other words, rows in M correspond to elements over the discretized range of X.

Fuzzy Inference

Now that M encodes a rule, fuzzy inference can occur. The input to M is a binary vector, I^i_X, with a single element equal 1 and all others equal to 0. This binary vector "represents the occurrence of the crisp measured datum x_i" [Kosko 91] (x_i is an element in the discretized range of X).

With the input of the binary vector A', M outputs a vector B', which is the row in M (clipped fuzzy subset) corresponding to x_i. This inferred output is a fuzzy subset *not* a crisp value. Mathematically, inference is represented by:

$$B' = A' \bullet M \qquad \text{(EQ 20)}$$

where,

$$b'_j = max_{1 \le i \le n}(min(a'_i, m_{ij})) \qquad \text{(EQ 21)}$$

Encoding and Inference For Multi-antecedent Rules

For rules with multiple antecedents, the process differs. Here the rule is decomposed into components whose outputs are later recomposed. For instance, the rule

IF *X is A* **AND** *Y is B* **THEN** *Z is C*

is decomposed into the two rules,

IF *X is A* **THEN** *Z is C* and **IF** *Y is B* **THEN** *Z is C*

Each component rule is now encoded normally as,

$$M_{AC} = A^t \bullet C$$
$$M_{BC} = B^t \bullet C \qquad \text{(EQ 22)}$$

Inference from each rule occurs normally:

$$C'_A = A' \bullet M_{AC}$$
$$C'_B = B' \bullet M_{BC} \qquad \text{(EQ 23)}$$

The output fuzzy subsets of the two rules are recomposed using set-theory intersection or union depending on the composition of the two rules (**AND** or **OR**). Using the previous example, C_A' and C_B' are recomposed as:

$$C' = C'_A \cap C'_B \qquad \text{(EQ 24)}$$

For example, if x_i and y_j are the crisp inputs, they are input to M_{AC} and M_{CB} as unit bit vectors I^i_X and I^j_Y. C' becomes:

$$C' = [I_X \bullet M_{AC}] \cap [I_Y \bullet M_{BC}] \qquad \text{(EQ 25)}$$

$$= a_i \wedge C \cap b_j \wedge C \qquad \text{(EQ 26)}$$

$$= min\,(a_i, b_j) \wedge C \qquad \text{(EQ 27)}$$

Refer to Figure 26 for a graphical interpretation.

Defuzzification

For real systems (with multiple rules), the next step is to merge the outputs (B') of each rule inference and defuzzify them into a single, crisp, usable value. The defuzzification procedure used here is the fuzzy centroid algorithm [Kosko 91].

The fuzzy centroid defuzzification algorithm produces a crisp output by computing the average of the geometric centroids of all the output fuzzy subsets. The algorithm can be simplified to:

$$O = \frac{\sum_{j=1}^{v} y_j S_j m(y_j)}{\sum_{j=1}^{v} m(y_j) S_j} \quad \text{(EQ 28)}$$

where, v is the number of fuzzy subsets defined on the consequent variable

y_j is the horizontal component of the centroid of fuzzy subset j

S_j is the area of fuzzy subset j

$m(y_j)$ is the value of the membership function of fuzzy subset j at y_j

The FIE uses this algorithm to compute the crisp output of a FAM. Figure 26 graphically summarizes the sequence of procedures for a two-rule FAM.

3.4.7 Summary

The FIE is a design tool composed procedures which infer information using one or more Hyper-FAMs and associated inference procedures. It is a self-contained, model-free control and analysis tool which performs well (compared to model-based methods) when the system under consideration is difficult to model mathematically.

3.5 Mission Manager

At both ends of the spectrum of intelligent controller architectures, there is a need for high-level management which oversees operation of the intelligent controller.

Functional architectures often have a hierarchical management chain where data and instructions move up, down and across different levels of control. While it is possible to implement this hierarchical management in the Reasoning task of a PRA module, the resulting code may be very inefficient.

Behaviour-based intelligent controllers tend to require arbitrators. Purely behaviour-based controllers perform poorly without external arbitrators, especially as the number of behaviours increase [Bellingham 91]. Arbitration-free systems tend to thrash about in local minima of a control state space as competing behaviours repeatedly drive the system towards and away from an undesirable location. However, it is difficult to implement an arbitrator with the PRA modules alone because they operate as self-contained, independent entities. Alone, they do not have the global knowledge required to arbitrate among themselves.

Hybrid mixes of functional and behavioural architectures also contain encompassing management routines which coordinate activities of the PRA modules.

The Mission Manager is a file which declares (makes instances of) tasks, maps, FIEs, queues, and other variables. It can also coordinate PRA modules, arbitration and other high-level functions. Because the high-level aspects of a controller vary widely, there is little structure imposed on the Mission Manager.

In this thesis a rudimentary Mission MAnager has been made for the mission of docking a vehicle. Details of this example can be found in Chapter 4, Section 4.5.1.

Chapter 4 Design Of A Docking-Capable AUV

This chapter describes an AUV and its intelligent motion controller built using the development system. The controller is capable of docking the AUV in a subsea environment which contains static and dynamic obstacles. For this work, most attention in the design was given to the intelligent controller. The purpose of designing the submersible and intelligent controller was to initiate some research into docking techniques and to verify that all aspects of the development system functioned properly.

4.1 Motivation

In many applications, the potential capabilities of unmanned, *untethered* submersibles (AUVs) far exceed those of unmanned, *tethered* submersibles. Future AUVs will be able to perform extended duration, long range missions with minimal support. However, some unresolved limitations imposed by tetherless operation have restricted their commercial success. Tetherless operation restricts the AUV's **power**, **computation** and **communication rates** because all resources for these sub-systems must be self-contained rather than (as with tethered submersibles) remotely supplied [Wall 91].

Within the constraints of limited power, computation and communications, most AUV research has concentrated on developing procedures and devices which allow the AUV to navigate from one waypoint to another and allow it to perform various tasks at a work site. Since most AUVs are either physical or simulated testbeds, the practical considerations of releasing, retrieving and more generally docking the vehicle, have been neglected [Gwin 92]. Docking has also been ignored because unlike spacecraft, unmanned submersibles do not carry a human payload whose life depends on an ultra-robust docking procedure. This indirectly devalues the role of docking in the design stages of the AUV and its support structure. If dealt with at all, docking procedures are an afterthought.

There are several practical reasons to study docking more closely and to design intelligent motion controllers which incorporate docking. Docking is a fundamental component of every mission and as AUV missions grow more frequent, inefficient docking procedures will delay the mission completion time. As AUV missions grow more complex and lengthy, remote sites for refueling and data transfer will become necessary. Docking procedures will play an important role in any remote site rendezvous. Robust docking techniques will also allow AUVs to refuel remotely with unmanned "fuel drogues" in a manner similar to airborne jet refueling[20].

4.2 Work Related To Docking

Among the recent research which has recognized the importance of AUV docking, the most comprehensive study, from a theoretical AI perspective, comes from Gwin and Smith [Gwin 92]. Following a criticism of the lack of research into AUV docking, the authors rigorously analyse the AI aspects of docking between an AUV and a manned submarine. Since both vehicles are autonomous (humans are autonomous), a Distributed Artificial Intelligence (DAI) solution is deemed most suitable. Three DAI architectures are investigated and a hybrid is suggested as optimal. This study examines docking at a theoretical level and provides no implementation details.

Rae and Smith have developed a fuzzy rule-based docking procedure (although it has few intelligent control aspects) [Rae 92]. They define similar docking stages and present a fuzzy controller which enables the AUV to dock with a static dock. Sensors and the environment are assumed to be noise-free. The fuzzy logic controller uses the concept of cell mapping to generate the control signals. In cell mapping, fuzzy control signals are associated with cellular regions in space. The controller response depends on which cells the vehicle is in and near. Apart from conceptual differences, the two controllers are similar at a low-level. However, the fuzzy control system lacks any reasoning or intelligent aspects, such as obstacle avoidance and path planning.

Kosko examines the differences in performance between a fuzzy controller and a neural network controller for truck loading dock parking [Kosko 91]. Although the vehicle model is substantially different, the example provides a good introduction to using fuzzy logic in control systems.

The Naval Ocean Systems Center has developed a human-in-the-loop docking procedure for the Free Swimmer Testbed vehicle and a static underwater platform [Bryan 91]. Trailing a fiber optic tether, the vehicle serves as a high bandwidth data link between a remote data gathering platform and a surface vessel. During the homing stage, the vehicle autonomously drives towards the beacon. However, a human operator pilots the vehicle (with video feedback) during the approach and arrival stages. This indicates the difficulties associated with fully autonomous docking. Again, emphasis is on the physical and hardware aspects of the task. No details of the homing controller are given.

---

20. ISER expressed interest in this idea since current methods for refueling AUVs are time consuming and require a crew and support vessel.

Ditang and Ghignone describe the physical and hardware aspects of launch and recovery systems for various underwater vehicles [Ditang 92][Ghignone 92]. They do not discuss the control algorithms in detail. Lebans et al. discuss a crane and cable based method for recovering the DOLPHIN AUV [Lebans 93]. The manually operated device is capable of launching and retrieving the vehicle in heavy seas, up to sea state 6. There are, as of yet, no apparent autonomous features in this docking procedure. The report outlines the hardware and procedure for docking the DOLPHIN.

Watkinson applies aerospace rendezvous and intercept guidance techniques to underwater vehicles [Watkinson 91]. The terminal objectives of aerospace rendezvous (zero difference in target-relative position and velocity) coincide with those of submersible docking. Using passive and active acoustic sensor models aboard an unmanned submersible, Watkinson analyses the parameters affecting successful underwater interception and rendezvous. Two guidance laws are tested: Proportional Navigation (with a passive sensor only) and Augmented Proportional Navigation (with an active sensor). Watkinson concludes that although sensor capabilities are of primary importance for successful missions, other significant areas are: submersible performance; guidance laws and target dynamics.

Watkinson's guidance algorithms do not use features such as obstacle avoidance (except in a rendezvous mission, to avoid head-on collisions with the target), path planning or internal map making. Instead, the algorithms form a reactive loop between sensors and actuators with a filter/estimator providing estimations of target parameters. This type of motion control would be most applicable to the case where the dock is moving. An encompassing, intelligent controller could incorporate the output of this rendezvous guidance system into its control strategies depending on the scenario (especially if it had to rendezvous with moving targets).

This research benefits substantially from work performed at University of Karlsruche [Zhang 92]. Zhang et al. discuss their mobile land-robot and the techniques it uses to navigate in partially known environments. The path planner uses the Tangent-Graph method and the low-level controller is based on fuzzy logic. Some of the controller features are based on those of Zhang et al.

4.3 Submersible Dynamics

The submersible is loosely based on ISER's THESEUS vehicle. Control surfaces on the THESEUS are fore and aft dive planes and a rudder with a single rear propeller driving the vehicle. Although the current model in EASY5 requires enhancements before it com-

pletely represents the THESEUS, imperfections in the vehicle dynamics will always exist. Refinements to the vehicle model will be an ongoing aspect of research. The purpose of this AUV design thesis was not to develop a precisely accurate submersible model, but for the purposes of designing an intelligent controller.

Assumptions

This version of the vehicle model contains assumptions which simplify the control problem and which overcome complexities in determining the forces and moments acting on the vehicle. These simplifications, while not realistic for a final design specification, are adequate for the initial design phase. As interest permits, the vehicle model may be improved. The simplifications are:

1. Roll and pitch are zero ($p = q = 0$) so that the controller does not have to control these variables.
2. Applied forces and moments are proportional to control surface inputs and vehicle velocities only, simplifying the calculation of the forces and moments.
3. Non-zero accelerations are proportional to the applied forces and moments only, removing complexities which arise from the dependence of the accelerations on the velocities.

All units conform to the “metres, seconds, degrees, newtons” convention unless otherwise indicated.

Derivation

Recall from Chapter 2 that the submersible design tool provides a framework which is based on the following state equations:

$$\dot{u} = rv - qw + F_x/m$$
$$\dot{v} = pw - ru + F_y/m$$
$$\dot{w} = qu - pv + F_z/m$$
$$\dot{p} = \frac{M_x + (\bar{I}_y - \bar{I}_z)\,qr}{\bar{I}_x}$$
$$\dot{q} = \frac{M_y + (\bar{I}_z - \bar{I}_x)\,pr}{\bar{I}_y} \qquad \text{(EQ 29)}$$
$$\dot{r} = \frac{M_z + (\bar{I}_x - \bar{I}_y)\,pq}{\bar{I}_z}$$

Due to assumption 1, the fourth and fifth equations are not required. Due to assumption 3, the mass and moments of inertia are not required.

The next step in the development of the THESEUS submersible model is to expand on the forces (F_i) and moments (M_i). With the aforementioned assumptions, the forces and moments are proportional to control surface inputs and vehicle velocities. Equations to define the forces and moments are thus:

$$F_x = K_1 PropellerSpeed - K_2 u$$
$$F_y = 0$$
$$F_z = (K_3 DiveplaneAngle - K_4 w) \cdot \frac{u}{4}$$
$$M_x = 0 \qquad \text{(EQ 30)}$$
$$M_y = 0$$
$$M_z = (K_5 RudderAngle - K_6 r) \cdot \frac{u}{4}$$

where, *PropellerSpeed, DiveplaneAngle* and *RudderAngle* are control inputs. It would have been more realistic to make the drag terms proportional to the square of the velocities however, this was overlooked in the model. Note that F_y is set equal to zero indicating that there are no control surfaces which result in a force being applied to the side of the vehicle. The rudder's effect is manifested in M_z.

The constant values, chosen by experimentation so that the model dynamics resembled a real vehicle, are:

$$
\begin{aligned}
K_1 &= 0.2\frac{N}{rpm}\\
K_2 &= 0.2\frac{Ns}{m}\\
K_3 &= 0.7\frac{N}{deg}\\
K_4 &= 0.55\frac{Ns}{rpm}\\
K_5 &= 0.1\frac{N}{deg}\\
K_6 &= 0.3\frac{Ns}{m}
\end{aligned}
\qquad \text{(EQ 31)}
$$

Ranges for the control surface inputs are:

$$
\begin{aligned}
0rev/\text{second} \le PropellerSpeed \le 15rev/\text{second}\\
-30° \le DiveplaneAngle \le 30°\\
-60° \le RudderAngle \le 60°
\end{aligned}
\qquad \text{(EQ 32)}
$$

Since the submersible's pitch is zero at all times, both sets of diveplanes are constrained to operate together. To prevent instability of the submersible, the velocities that are not constrained to be zero are bounded as follows:

$$
\begin{aligned}
0.0m/s \le u \le 4.0m/s\\
-2\,(0\dot{m})\,/s \le v \le 2.0m/s\\
-0.5rad \le r \le 0.5deg/\text{second}
\end{aligned}
\qquad \text{(EQ 33)}
$$

The above equations define the submersible dynamics used in the development of the docking-capable AUV.

4.4 Sensors

Internal sensors on board this vehicle are a linear position sensor, an angular position sensor, a linear velocity sensor and an angular velocity sensor. The sensor's output is expressed in terms of the frame of reference A. The controller is responsible for trans-

forming this measurement into the World coordinate system. Implementation details of these sensors are ignored - it is assumed that some combination of compasses, gyros and accelerometers provide the information. Each sensor is ideal in the sense that it outputs an error-free signal. Future work may incorporate non-ideal sensors.

The single external sensor on-board the vehicle is an active, single-beam, lookahead sonar. A square, two-dimensional, uniformly weighted hydrophone array, it has the following pattern function [Burdic 84]:

$$\boldsymbol{G}(\psi, \theta, \lambda) = \left[\frac{\sin(\pi N d \sin(\psi/\lambda))}{\sin(\pi d \sin(\psi/\lambda))}\right]\left[\frac{\sin(\pi N d \cos(\psi) \sin(\theta/\lambda))}{\sin(\pi d \cos(\psi) \sin(\theta/\lambda))}\right] \quad \text{(EQ 34)}$$

where,

G is a three-dimensional vector defined by its magnitude and the angles θ and ψ
θ is the angle between the z-axis[21] and **G**'s projection in the zy plane
ψ is the angle between the zy-plane projection of **G** and **G**
λ is the signal wavelength
d is the inter-element spacing
N is the number of hydrophones in the array

The endpoints of the collection of vectors G, generated by sweeping through $0° \geq \theta > °360$ and $-90° \geq \psi \geq 90°$, represent the gain of a spatial filter (the sonar) in three-dimensional space. Note that the angles θ, ψ are not the pitch and yaw angles associated with the submersible kinematics and dynamics. In the vehicle, the sonar is rotated +90 degrees about the y-axis so that the beam is oriented out in front of the vehicle nose. Figure 27 summarizes the characteristics of the lookahead sonar.

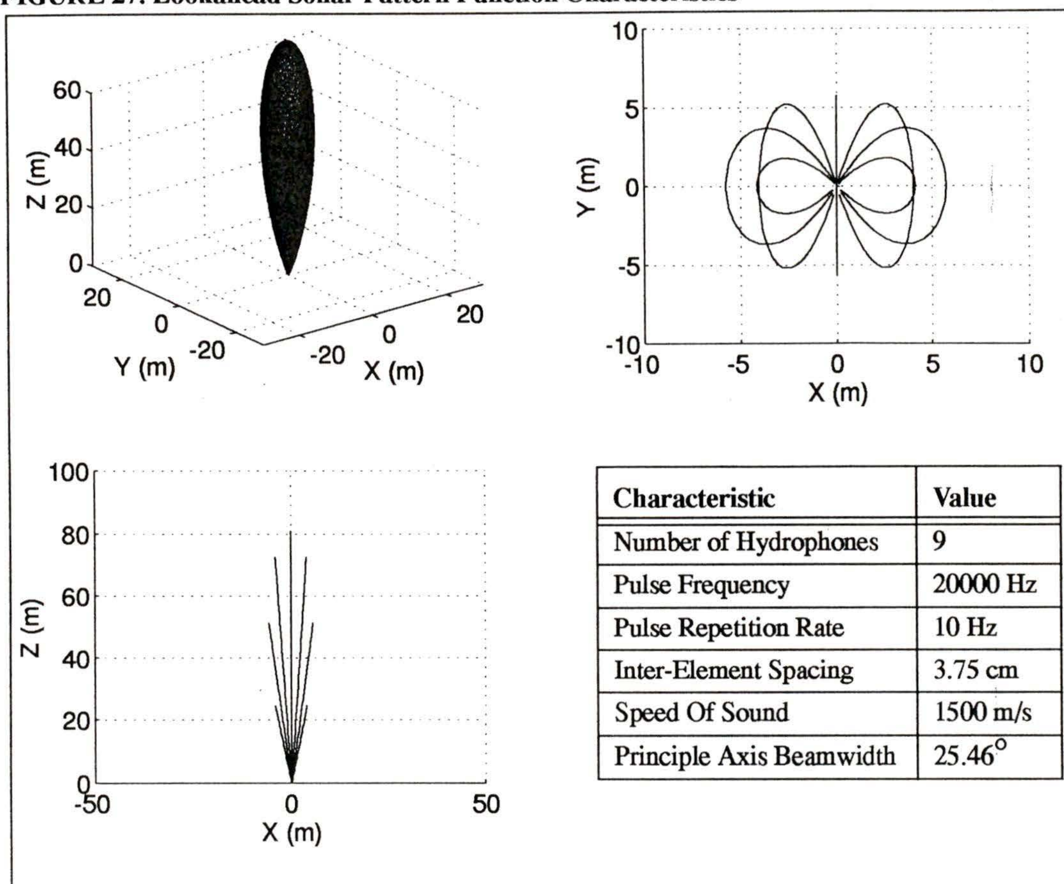
There are several restrictions which limit the realism of this sonar sensor:

- A "contact" is made only if the pattern function of the sonar encompasses an object origin. Object surfaces do not register as contacts.
- All contacts are deterministic - if an object origin is within the pattern function, the returned ping has a total travel time equal to exactly twice the distance to the object origin divided by the speed of sound in water
- Side lobes are ideally suppressed with no main-lobe distortion

---

21. The axes here refer to the axes of A, the body-fixed frame of reference

FIGURE 27. Lookahead Sonar Pattern Function Characteristics



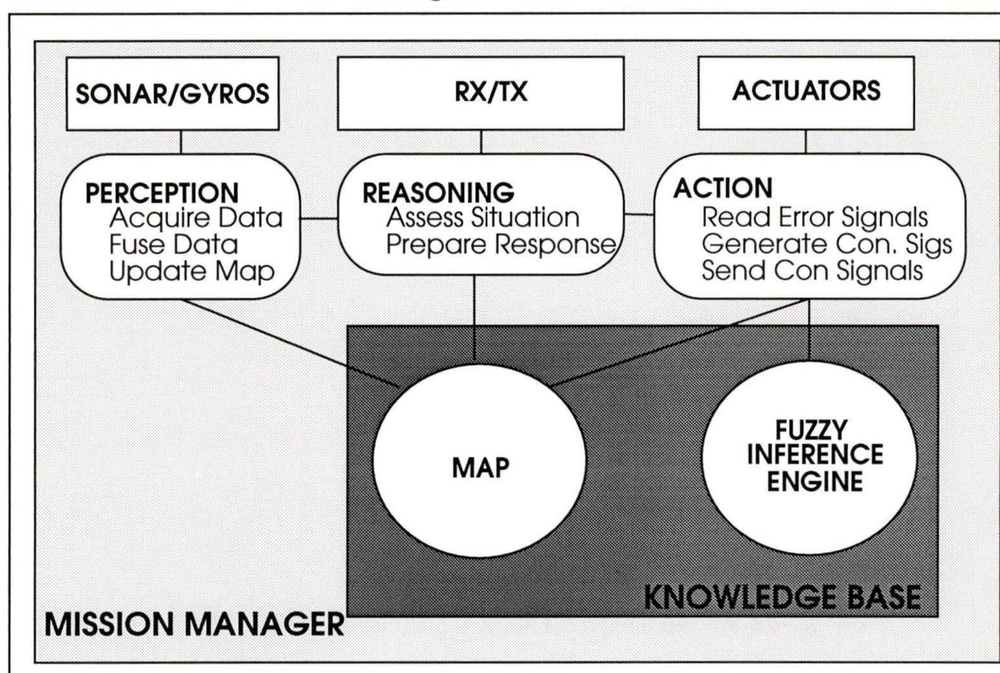
Characteristic	Value
Number of Hydrophones	9
Pulse Frequency	20000 Hz
Pulse Repetition Rate	10 Hz
Inter-Element Spacing	3.75 cm
Speed Of Sound	1500 m/s
Principle Axis Beamwidth	25.46°

The completeness of the virtual environment comes at the expense of these kind of simplifications. As time and research interests permit, the sensors can be made more realistic.

4.5 Intelligent Motion Controller

The intelligent controller is a classic functional controller with a single PRA module. Figure 28 illustrates its configuration.

FIGURE 28. Architecture Of The Intelligent Motion Controller

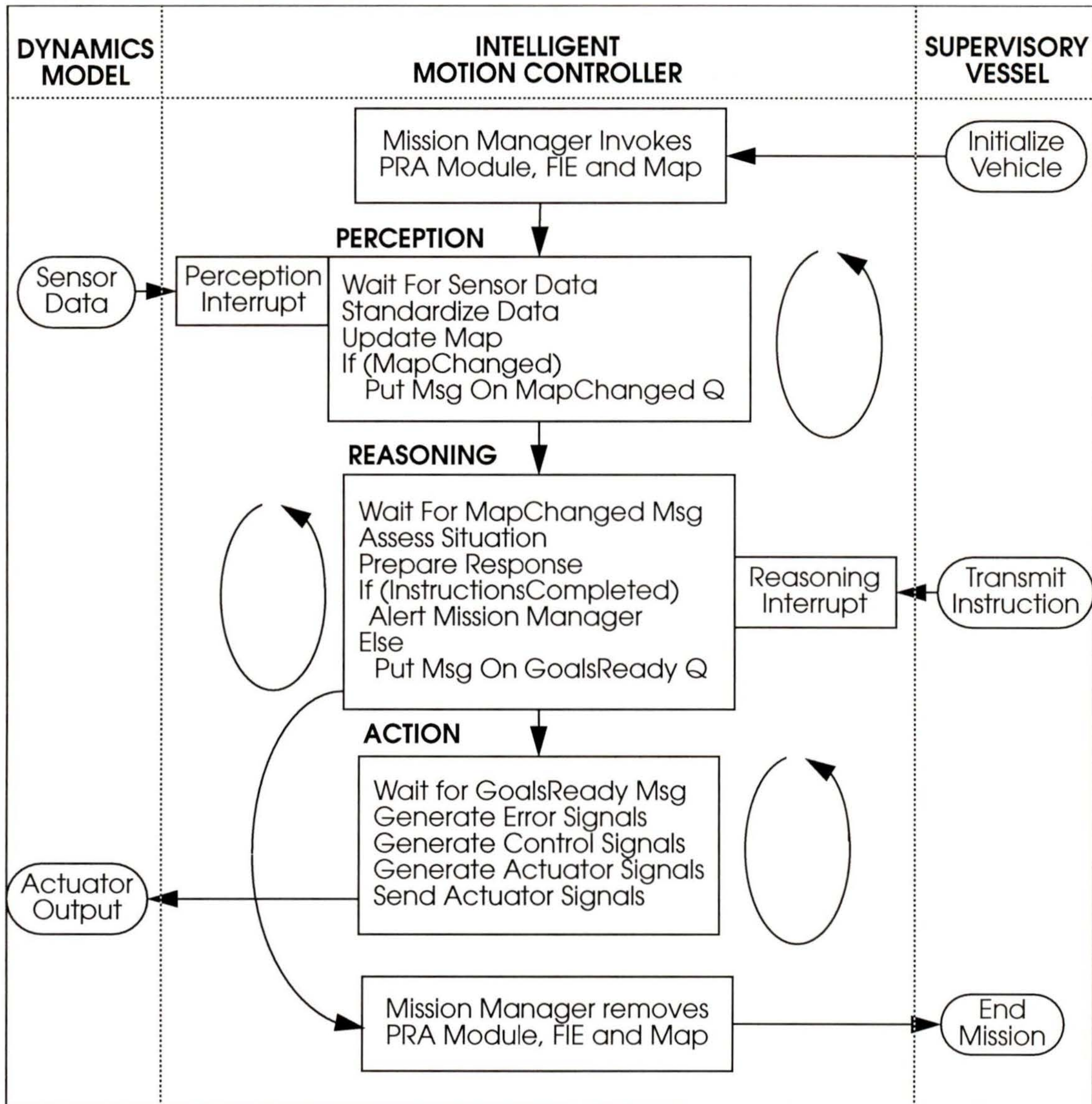


This relatively simple architecture uses all the controller design tools provided and is able to successfully execute docking missions. Special features of this intelligent motion control system are its cornering algorithm which generates smooth trajectories around subgoals in a path and its ability to compensate for moving obstacles without continuously replanning paths. However, the controller also has some shortcomings. For instance, in waters with multiple moving objects, the controller's response may be sub-optimal. As well, the map-making logic in the Perception task object is simplistic and can produce an internal map which (after encountering a few obstacles) poorly represents the real world. The single lookahead sonar is also responsible for this susceptibility because the Perception functions must build the map with limited sensor data.

With a single PRA module, the architecture is a functional one. The perception, reasoning and action task objects sequentially seize and relinquish control using queues (indicated by the interconnecting lines between tasks in Figure 28. Figure 29 illustrates how control flows between tasks during a mission. Although the implementation of the architecture is a serial one, parallel implementation is feasible[22]. This would allow the three tasks to operate with more independence with only the map (blackboard) required for inter-task communication. Tasks run serially mainly for simplicity and to make use of the queuing mechanism (the lines between the tasks indicate the serial nature of the tasks).

22. The task library in this version of C++ can simulate parallelism but cannot truly run tasks in parallel.

FIGURE 29. Flow Of Control



The following sections describe the characteristics of the intelligent controller in more detail.

4.5.1 Mission Manager

The Mission Manager declares and stores the FIE and single map which the three task objects access through pointers. Since all three tasks access the map and rely on it for information, the map fits the description of a blackboard.

To construct the single PRA module, the Mission Manager constructs the Perception, Reasoning and Action task objects. Once all three tasks are running, the Mission Manager

enters a wait state until the Reasoning task is complete. Completion of the Reasoning task indicates to the Mission Manager that all instructions have been completed and that the mission has been successfully executed. The Mission Manager then destroys the Perception and Action tasks and terminates the controller program - essentially, turning off the AUV.

This Mission Manager has a relatively passive role in the intelligent controller. Its main function is to act as a storage location for objects and to initialize and terminate the intelligent controller.

4.5.2 Perception Task Object

The Perception task object has access to data from five sensors. Externally, the lookahead sonar (described in Section 2.2.2) returns time delays between pulse transmission and reception. There are four internal sensors (an assumed set of gyroscopes and compasses) which return the vehicle's sensed location, orientation, linear velocity, and angular velocity.

After the Mission Manager invokes its constructor, the Perception task is in a *running* state and immediately accesses the `Data_Ready` queue to see if there is any sensor data available. If no data is available, the Perception task enters a *wait* state and relinquishes control of the CPU.

Data Acquisition

The Perception interrupt activates on a signal from the physical model. The physical model signals the Perception task object when the sensors have new data. It reads sensor data (created by the physical model) from a file and loads it into the buffers of the LINPOS, ANGPOS, LINVEL, ANGVEL, and LA_SONAR sensor objects. These buffers correspond to sensors for vehicle location, orientation, linear velocity, angular velocity and sonar, respectively. After loading the buffers, the interrupt puts a message on the `Data_Ready` queue, alerting the Perception task to the availability of new data. The Perception task, previously in a wait state, now enters a *pending* state which indicates that it is ready to run when the CPU becomes free.

Data Standardization

In this Perception task, the Standardize Data routine simply converts the sonar sensor's time delay to a distance (assuming that sound travels 1500 metres/second in water). As

well, a Full/Empty flag is set depending on whether the transmitted pulse returned. If no pulse was received (time delay is infinite) the flag is set to Empty. Data from the four internal sensors needs no standardization because it is already in the desired units.

Data Fusion

Since all five sensors measure different parameters, the Data Fusion function in the Perception task is essentially empty. Fusion of sequentially acquired data from the same sensor is possible but was not implemented. As the control system adds other sensors, data standardization and sensor fusion will become more prominent since multiple external sensors will measure a wider variety of characteristics with varying degrees of confidence.

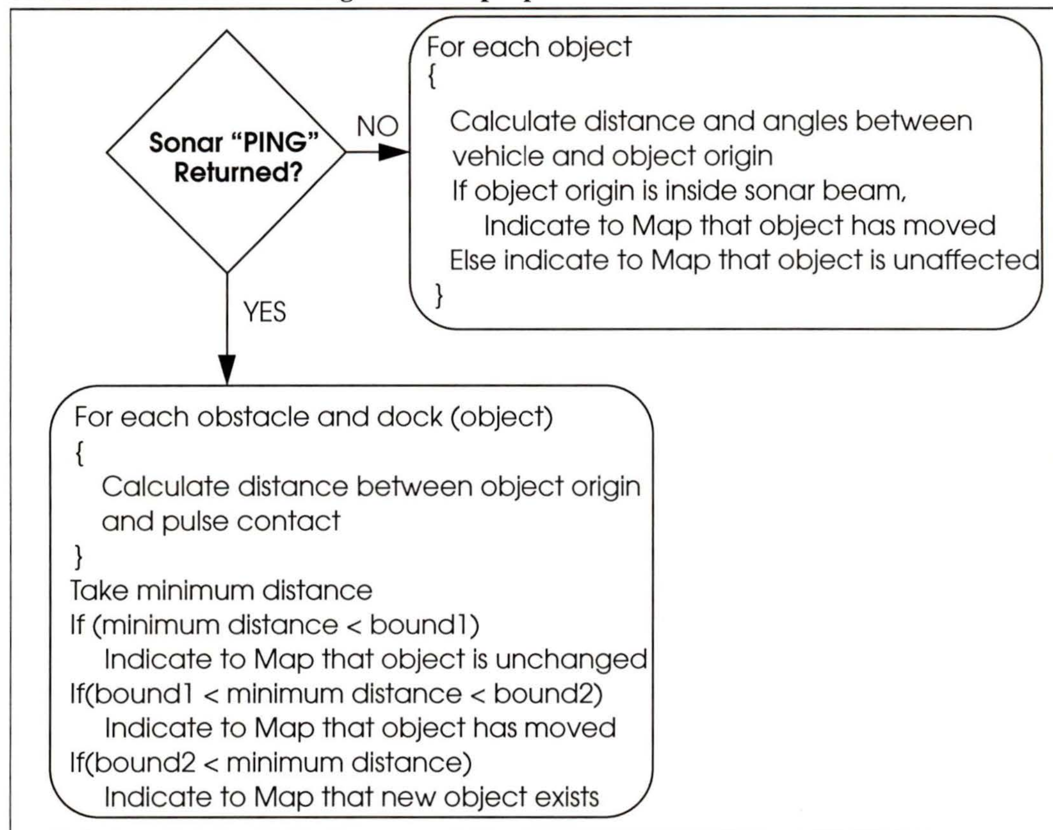
Map Updating

The map takes buffer data from the four internal sensors (location, orientation, linear and angular velocities) and directly modifies the parameters of the Base object (in the map) on each cycle through the Perception task. This allows the Reasoning and Action tasks to be aware of the vehicle's position.

Sonar data processing depends on the state of the Full/Empty flag. If the flag is Full (meaning the pulse contacted an object), the Perception task iterates through the map objects checking to see if the contact occurred near an existing obstacle or dock. If the flag is Empty, map objects are still checked to see if the path of the sonar implies that an object has moved.

If, after analyzing the sonar return, the updating routine determines that no objects changed, it returns the BASE_MOVED flag if the vehicle has moved. This ensures that the Perception task sends a message to the Reasoning task even though nothing significant occurred in the environment. If the vehicle does not move and there are no changes to the environment, the Perception task will not signal the Reasoning task, it will simply continue to sense the environment until it perceives a change. Figure 30 illustrates the details of the decision-making in the map updating routine.

FIGURE 30. Decision-Making In The Map-Update Routine



Of note in the above figure is that proximity calculations are based on object origins rather than object surfaces. This approximation adds some uncertainty to the decision-making process and helps to simulate real-world returns where sensed data may not reflect the true situation. When obstacles are large relative to the sonar beam, the approximation breaks down.

From the decision-making logic, it is obvious how the quality of internal map can degrade. as the mission progresses. Depending on the size of the error bounds, the map-updating logic can easily mistake moving objects for new ones and vice versa. The map's representation is especially likely to become corrupted when objects in the real-world are close together.

Relinquishing Control

If the Perception task has updated the map, it puts a message on the `Map_Changed` queue and loops back to check the Data_Ready queue for new sensor data. The Reasoning task object, waiting for a message on its `Map_Changed` queue now enters a pending

state. If the map has not changed, the Perception task loops to check the `Data_Ready` queue without putting a message on the `Map_Changed` queue.

4.5.3 Reasoning Task Object

After the Mission Manager invokes its constructor, the Reasoning task object immediately checks the Map_Changed queue to assess the current situation.

Instruction Processing

The user transmits instructions via the user interface (thereby simulating supervisory control aboard the support vessel). Currently, only the Dock and End Mission instructions are supported. On the transmit request, the user interface loads the instruction file and sends the SIGINT signal to the intelligent controller.

Instructions are handled by the interrupt function of the Reasoning task object. The interrupt reads the instruction file, adds the new instruction to a FIFO queue of instructions, increments an integer variable representing the number of instructions and then exits.

Situation Assessment

Situation Assessment in this controller checks two circumstances: the status of the instruction execution and the vehicle's progress along its path. These are represented in pseudocode as:

```
// Check current instruction and queue
if (Instruction_Queue != EMPTY &&
    Current_Instruction == COMPLETED)
{
  Indicate that a new instruction should be executed
  Situation = NOT_AS_PREDICTED
  return(Situation)
}
else Situation = AS_PREDICTED

// Check path and vehicle's position relative to it
if (Current_Path == VALID && Path_Boundary != VIOLATED)
{
  if (AUV has reached a subgoal)
  {
    Indicate that a new subgoal should be chosen
    Situation = NOT_AS_PREDICTED
  }
```

```
  else Situation = AS_PREDICTED
}
else
{
  // Either the current path is no longer valid or the
  // vehicle is outside the current path boundary
  Indicate that a new path must be planned
  Situation = NOT_AS_PREDICTED
}
return(Situation)
```

The Assess_Situation function returns either AS_PREDICTED or NOT_AS_PREDICTED depending on the circumstances. It also manipulates an internal variable to more precisely indicate the nature of the (unpredicted) circumstance.

If the situation is as predicted, no is further work required because the Reasoning task will have already planned for the predicted situation. If however, the situation is not as predicted, the Prepare Response function is called to deal with the new information.

Response Preparation

The Prepare Response function examines the variable set by the Situation_Assessment function. If this variable indicates that a new instruction is required (meaning the previous instruction is complete), the function decodes the next instruction on the queue and makes it the current instruction.

If the variable indicates an invalid path or a path boundary infraction, the Prepare Response function will replan the current path. When the variable indicates that the current subgoal has been attained, the function replaces the attained subgoal with the next unattained subgoal.

As is typical with high-level functions, both the Situation Assessment and Response Preparation functions are relatively simple. The more complex aspects of their operation are carried out by lower-level subroutines which concentrate on numerical relationships rather than symbolic ones.

Relinquishing Control

After preparing an appropriate response, the Reasoning task returns to the top of the main control loop and waits for the internal map to change. This is logical because the Reasoning task has accounted for the current situation which will not change unless the map

changes. If a new instruction arrives, it will be processed only when the Perception task registers a map change.

Interaction between the Reasoning and Action is in the form of the `Goals_Ready` queue and two shared objects, CurGoal and NexGoal. The CurGoal is the current subgoal in the path that the vehicle is trying to reach while the NexGoal is the next subgoal in the path. The Reasoning task sets the CurGoal and NexGoal goals depending on the location of the vehicle relative to subgoals in the current path. With these two goals, the Action task calculates actuator signals.

After preparing a response or verifying that the situation is as predicted, the Reasoning task will have assigned locations to the CurGoal and NexGoal. It then puts a message on the `Goals_Ready` queue which will send the Action task from a waiting to a pending state.

Considerations Specific To Docking

Thus far, none of the algorithms have been explicitly tailored for a docking-capable controller. This supports the idea that research into docking requires research into general motion control algorithms. However, one constraint due specifically to docking, is that the vehicle must approach the goal (dock) from a specified direction. Its alignment with the dock entrance cannot be arbitrary.

To force the vehicle to approach the dock from the correct direction, the path planner sets as the last subgoal, a safety location 50 metres in front of the dock. This is an attempt to ensure that the vehicle correctly aligns itself with the dock before approaching it. For example, assume the vehicle were approaching the dock from a head-on direction so that the nose of the dock was closer to the vehicle than the dock entrance. Without a predefined safety location to aim for, the vehicle would collide with the dock nose. Instead, it reaches the safety location and orients itself so that it approaches the dock entrance properly.

The other feature of the intelligent controller specific to docking is the "dock avoidance" algorithm, which is described in more detail in the next section.

4.5.4 Action Task Object

Until it receives a message from the Reasoning task on the Goals_Ready queue, the Action task is in a wait state. When a message arrives, the Action task examines the two updated goals (CurGoal and NexGoal).

The Action task controls the vehicle's rudder, diveplanes and propeller. Although there are both fore and aft diveplanes, this controller moves both together. A controller which controls the two planes independently is feasible but is not required since the vehicle does not pitch.

Three HyperFAMs which contain 74 fuzzy rules and 12 fuzzy variables make up the low-level control FIE in the Action task (as summarized in Table 1).

TABLE 2. FIE For Low-Level Control

HyperFAMs	FAMs	FAM Antecedents	FAM Consequents	# Of Rules
PATH_HFAM	p_Head2Rudder p_Depth2Divep p_Range2Speed	Yaw Velocity & Heading Error Forward Speed & Depth Error Range Error	Rudder Angle Diveplane Angle Propeller Speed	7 21 4
DYNAMIC_ OBS_HFAM / DOCK_ AVOIDANCE_ HFAM	d_Head2Rudder d_Depth2Divep d_Range2Speed d_HR2Weight	Heading Error Forward Speed Depth Error Range Error Heading Error Range Error	Rudder Angle Diveplane Angle Propeller Speed Dynamic Obstacle Weight	7 7 3 21
CORNERING_ HFAM	p_Range2Weight	Range Error	Cornering Weight	4

The FIE's output is three "final control signals" - one signal each for the rudder angle, diveplane angle and propeller speed. The final control signals are a weighted average of path following/cornering control signals, dynamic obstacle avoidance control signals and dock avoidance control signals. The following paragraphs explain in more detail how the final control signals are obtained.

Path Following/Cornering Control Signals

The Path HFAM generates control signals to nullify the error signals between the vehicle's current state and a goal state. Rules encoded in the Path HFAM are designed to respond in

a timely, stable manner to the vehicle's progress towards well-defined and well-planned goals.

Working in tandem with the Path HFAM, the Cornering HFAM allows the Action task to generate smooth transitions (cornering) between subgoals. Zhang et al. use a cornering method in their controller for a two-dimensional domain [Zhang 92]. This controller successfully extends their idea to three dimensions.

To generate *path* control signals (allowing the AUV to follow a path), the Action task first fires <the vehicle's error state relative to the CurGoal>[23] through the Path HFAM. The error state (as outlined in Table 2) consists of: the vehicle's yaw velocity, its heading error relative to the CurGoal, its forward speed, its depth error relative to the CurGoal and its range error relative to the CurGoal. It then repeats this process using <the AUV's error state relative to the NexGoal>. This generates two sets of control signals: one set relative to CurGoal and another set relative to NexGoal. Each set contains three control signals - one each for the rudder angle, diveplane angle and propeller speed.

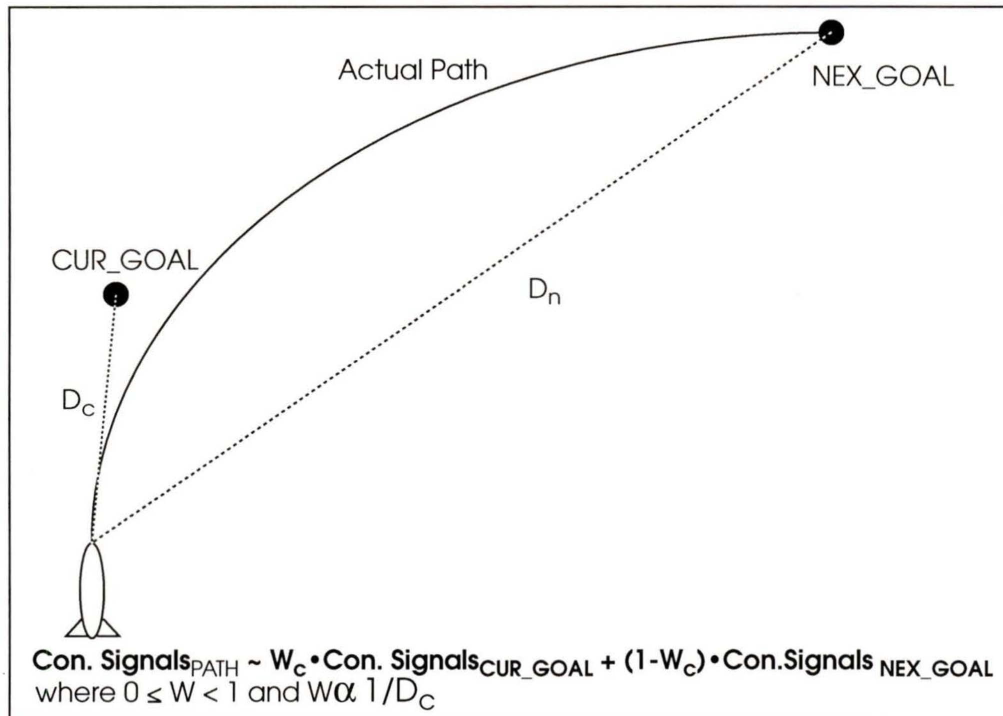
Now, the distance error between the vehicle and CurGoal is fired through the Cornering HFAM. The Cornering HFAM generates a weight which is proportional to the distance between the vehicle and CurGoal. The path control signals are a weighted sum of the two sets of control signals (corresponding to the CurGoal and the NexGoal). The pseudo-equation representing this calculation is:

$$\mathrm{ConSigs}_{PATH} = W_C \cdot \mathrm{ConSigs}_{CURGOAL} + (1 - W_C)\,\mathrm{ConSigs}_{NEXGOAL}$$

where W_C is the cornering weight determined from the Cornering HyperFAM. Figure 31 illustrates the cornering concept in two dimensions. Cornering in three dimensions is essentially the same.

23. The <...> notation group a concept to reduce the complexity of the sentence.

FIGURE 31. The Cornering Algorithm

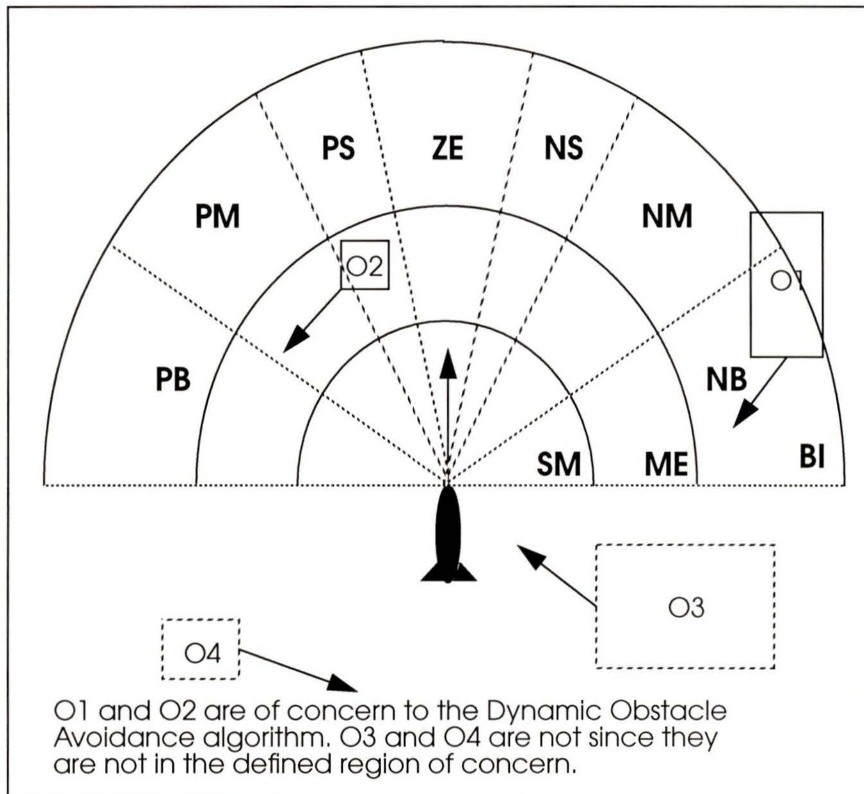


Dynamic Obstacle Avoidance Control Signals

At this point in the calculation of the final control signals, the Action task has taken into account all aspects related to the path. However since the path is based only on static obstacles, moving obstacles have been ignored. To account for dynamic obstacles, the Action task must separately generate control signals to avoid them.

Not all dynamic obstacles require attention from the dynamic obstacle avoidance algorithm (DOAA). Figure32 illustrates the concept of "regions of concern" in two-dimensions. The regions of concern are fuzzily defined over an area where moving obstacles could pose a danger to the AUV. The heading of the obstacle from the AUV combined with the range to the obstacle gives a good measure of the risk associated with it. Obstacles outside all regions of concern are ignored (this may not be an ideal solution, but it was implemented as such for this controller). In three-dimensions, the regions of concern are semi-circular wedges.

FIGURE 32. Regions Of Concern For Dynamic Obstacle Avoidance



In Figure 32, the heading is separated into seven zones (Positive Big (PB), Positive Medium (PM), Positive Small (PS), Zero (Z), Negative Small (NS), Negative Medium (NM), Negative Big (NB)). The range variable is separated into Small (SM), Medium (ME) and Big (BI).

The DOAA focuses on only one obstacle at a time.Thus if more than one dynamic obstacle is in the region of concern, the Action tasks must decide which obstacle is "more important" to avoid. The "importance" of avoiding an obstacle increases as its relative range decreases and its relative heading approaches Zero. After identifying the "most important" dynamic obstacle according to the above figure, the DOAA bases its control signals only on that obstacle.

Dock Avoidance Control Signals

To avoid head on collisions with the dock, the intelligent controller applies the DOAA to the dock when the AUV comes within 50 metres of it. The dock avoidance control signals are generated identically to the dynamic obstacle avoidance control signals.

When the CurGoal equals the NexGoal, the controller realizes that the AUV is ready to dock and therefore inhibits the dock avoidance mechanism.

Final Control Signals

The last operation in the calculation of the final control signals is to perform a weighted average of the three sets of HFAM-generated control signals.

Under the assumption that obstacles and target are widely spaced (as determined in performance evaluation), the dock avoidance signals when present, preempt the DOAA signals. Represented in equation form, the final control signals become:

$$\mathbf{ConSigs}_{\mathbf{Final}} = W_N \cdot \text{ConSigs}_{DockAvoidance} + (1 - W_N)\,\text{ConSigs}_{Path} \qquad \text{(EQ 35)}$$

where W_N, the weighting factor, is inversely proportional to the distance between the vehicle and the dock. If dock avoidance is not required, the final control signals are:

$$\mathbf{ConSigs}_{\mathbf{Final}} = W_N \cdot \text{ConSigs}_{DynamicObs} + (1 - W_N)\,\text{ConSigs}_{Path} \qquad \text{(EQ 36)}$$

where W_N is inversely proportional to the distance between the vehicle and the "most important" dynamic obstacle. After sending the final signals to the actuators, the Action task returns to the top of the main control loop to wait for a message from the Reasoning task.

Chapter 5 Performance Evaluation Of Docking-Capable AUV

With the AUV and intelligent motion controller designed, attention turns to its performance. Analyses of low-level controller components, vehicle dynamics and vehicle responses to environment/map discrepancies are all potentially important. Here, the emphasis is on controller components and macroscopic vehicle behaviour. Beyond the fact that the vehicle dynamics "look right", the many simplifications lead to a vehicle model which is very responsive in comparison to actual submersible vehicles. Future research may focus on improving the realism of the vehicle dynamics.

Performance evaluation takes place in the context of the dynamics of the vehicle. Therefore, the controller should be evaluated according to the vehicle's physical limitations. Table 3 summarizes the vehicle characteristics.

TABLE 3. Vehicle Parameters

Parameter	Max/Min Value
Forward Speed	0-4 m/s (= 14.4 km/h)
Dive Speed	0-2 m/s (= 7.2 km/h)
Yaw Rate	0-28°/sec (= 0.5 rad/s)
Turning Radius	Infinite-12m

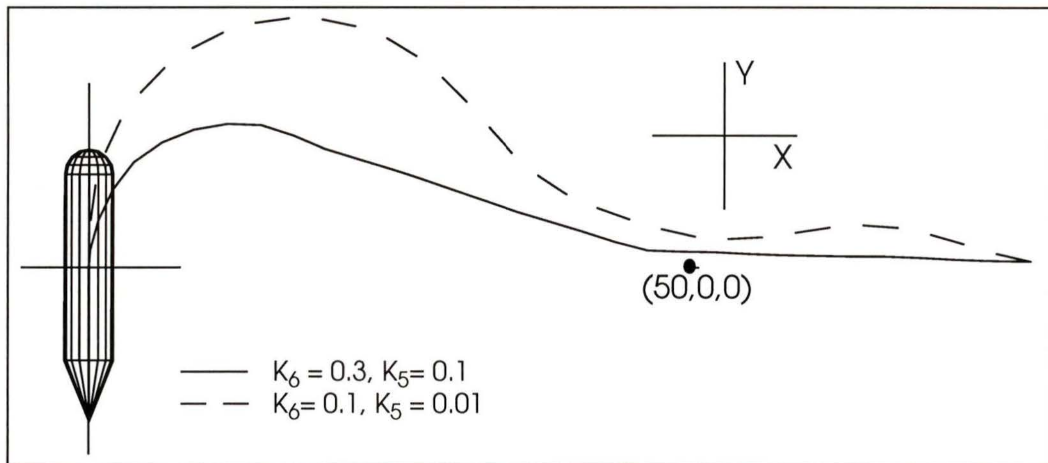
The velocities, slightly greater than in a real-world situation, were chosen to reduce the mission simulation time. On a SparcStation IPX, the AUV has a real-world speed of about 3 km/h. In other words, if a user starts the submersible running at full speed and goes away for one hour, when she comes back, the vehicle will have moved about 3 km.

The following sections examine the performance of the controller and vehicle from an algorithmic perspective. The path following/cornering, and dynamic obstacle avoidance algorithms of the Action task are examined independently before system level evaluation takes place. Although they are not explicitly investigated, reasoning, perception and map-related functions are involved in each test.

5.1 Path Following Algorithm

This section examines the robustness[24] of the low-level fuzzy rule-based path following algorithm. In Figure 33, the vehicle is instructed to head to the subgoals (50,0,0) followed by (100,0,0) from its initial location at (0,0,0) (all coordinates are expressed in metres). Initially, is has a 90° yaw angle. The two lines represent the vehicle's trajectory (in the lateral plane) under different conditions as outlined in the figure.

FIGURE 33. Lateral Response of Vehicle (xy plane)



A subgoal is "attained" as soon as the distance between the AUV and the subgoal is under 5 metres. For this reason, the vehicle does not precisely contact (50,0,0). At 5 metres from the first subgoal, the vehicle starts to head for the new subgoal of (100,0,0). Rather than indicating poor performance, the slight bypass indicates successful subgoal attainment.

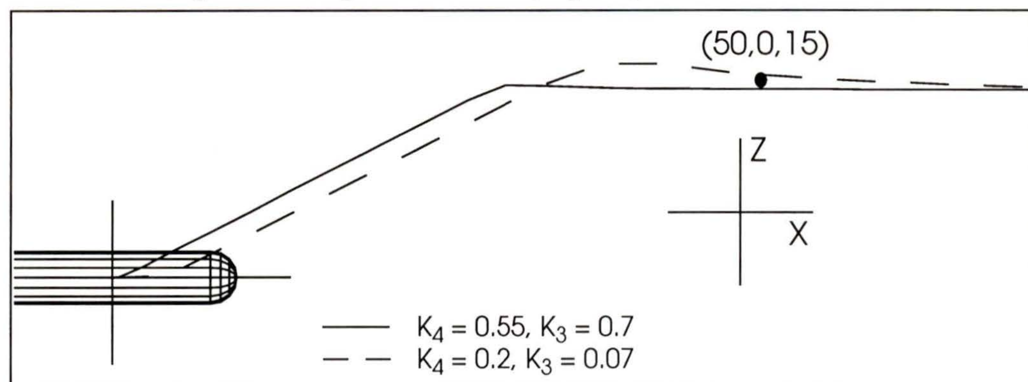
It is difficult to discriminate between vehicle limitations, sensitivity to initial conditions and controller deficiencies. However it appears that the fuzzy rules for heading control are relatively robust. An order of magnitude change in rudder gain combined with a reduced yaw drag coefficient approximately doubles the maximum overshoot but has little effect on the response time. The increased settling time is unacceptable for these subgoals but may be adequate when subgoals are separated by larger distances. As is typical with fuzzy controllers, there is a small (<2°), oscillatory, steady state error from the heading control component of the path following algorithm [Kosko 91]. This oscillatory behaviour, while not clearly distinguishable in the above figure, becomes obvious when the vehicle travels longer straight line distances.

---

24. Robustness in this case implies a study of the effects of perturbations on a subset of vehicle dynamics parameters.

Figure 34 shows an equivalent scenario which illustrates the controller's response in the diving (longitudinal) plane. In this case, the dock is located at (100,0,15) and the vehicle starts from (0,0,0) with a pitch angle of 0°.

FIGURE 34. Longitudinal Response of Vehicle (xz plane)



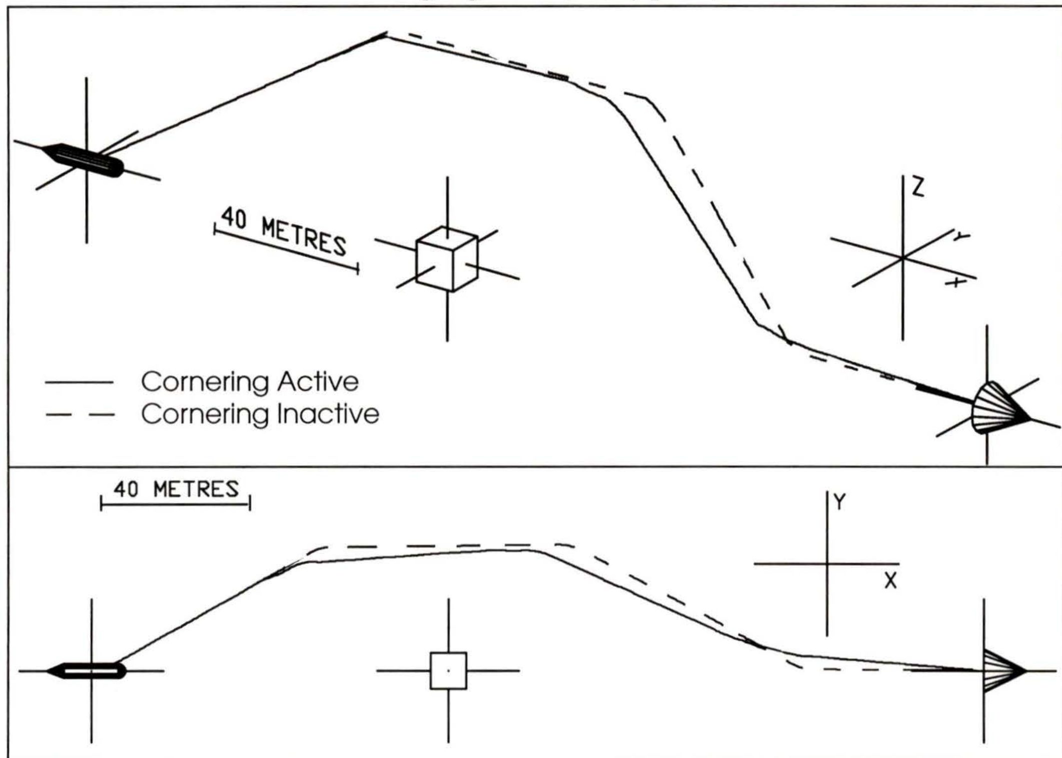
The default configuration is essentially critically damped with a constant steady state error of about -0.4 metres. The modified configuration, using the same rule base, yields an small overshoot with an oscillating steady-state error. Again, the controller appears reasonably robust to changes in vehicle dynamics.

Continued tuning would reduce the magnitude of the steady state errors however, they are acceptable under the assumption that obstacles and targets are widely spaced. Other tests for controller robustness, such as the removal or alteration of fuzzy rules, are possible, but are not useful for this high level examination.

5.2 Cornering Algorithm

Figure 35 shows a scenario with the vehicle, a single static obstacle, a dock, and a mission to reach the dock (250 m directly ahead of the AUV). The two trajectories show the effect of the cornering algorithm.

FIGURE 35. Effect Of The Cornering Algorithm (3D, xy plane)



With the cornering algorithm inactive, the vehicle proceeds to the current subgoal until it attains it and the heads to the next subgoal. This behaviour results in a "line-segment" trajectory from the start to the finish. Cornering smooths the trajectory and adds to the intelligent, pro-active nature of the controller. Modifying various parameters within the cornering algorithm can smooth the trajectory even more, however there is a trade-off - the greater the deviations from the planned path the higher the risk of obstacle collisions.

Cornering effects become more prominent[25] as the relative heading and depth differences between <the AUV and the current subgoal> and <the AUV and the next subgoal> increase. This is because the control signals for each subgoal become polarized and result in a trajectory which more sharply curves towards the next subgoal before reaching the current subgoal.

From observing both trajectories it is obvious that the path planner does not necessarily find the shortest distance path. Instead it plans a path whose intermediate subgoals (if any)

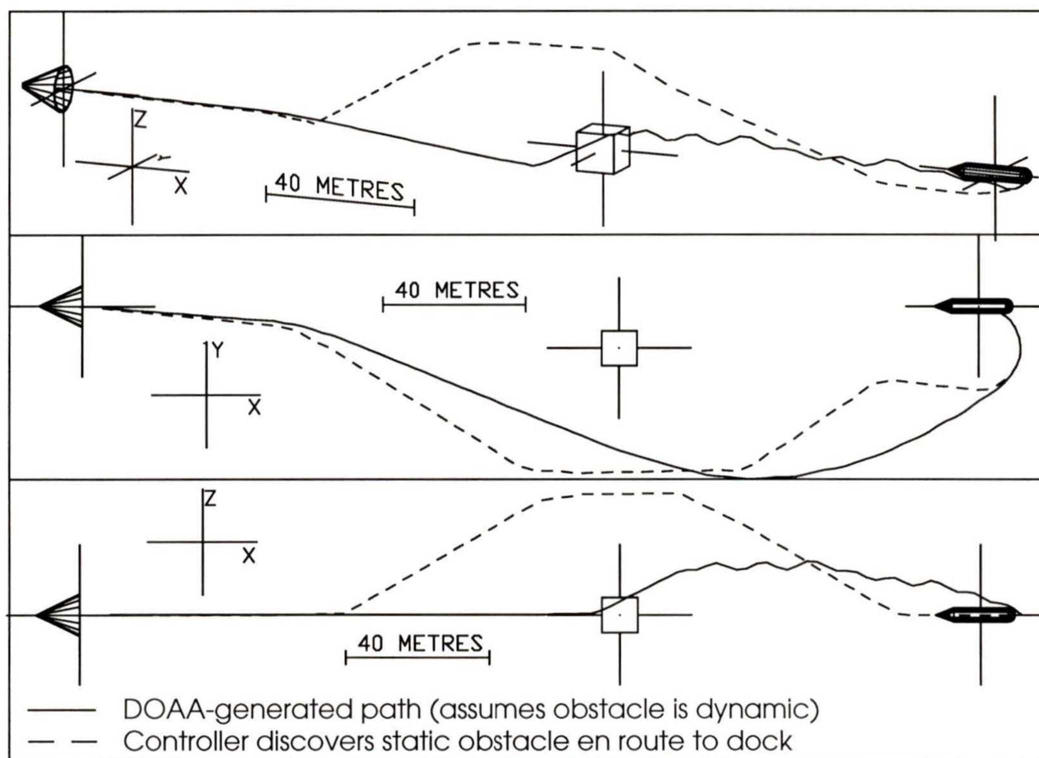
25. A z-plane projection of the trajectory is not included because the first two subgoals are at the same depth. The cornering effect was thus unobservable. In cases where the depth of consecutive subgoals differs, the effect of the cornering algorithm is evident.

are obstacle vertices. An improved path planner will either use a different technique (such as artificial potential fields) or refine the existing algorithm.

5.3 Dynamic Obstacle Avoidance And Static Unmapped Obstacles

The motion controller ignores dynamic obstacles when it is planning a path to a dock. Instead, a separate Hyper-FAM in the FIE ensures that the AUV stays a safe distance from moving obstacles. To test the avoidance algorithm, a scenario was chosen where the map has misrepresented a static obstacle as a moving obstacle with a speed of 1 m/s along the X-axis[26] (in reality, the obstacle is not moving). The mission required the vehicle to dock to a dock which was located 250 metres behind the vehicle. Figure 36 displays the outcome.

FIGURE 36. Dynamic Obstacle Avoidance Algorithm (3D, xy plane, xz plane)



If the dynamic obstacle avoidance algorithm is inhibited, the trivial outcome is that the vehicle collides with the obstacle which it thinks is moving. Thus for comparison, Figure

26. Due to the simplistic nature of the lookahead sonar, the controller assigns a velocity of 0 m/s to all new objects it encounters. The only way to test the dynamic obstacle avoidance algorithms is to manually alter the map prior to executing a mission. New sensors with greater sensitivity may allow the controller to estimate an obstacle's velocity.

36 also includes another trajectory, generated by having the vehicle's internal map contains no obstacles at all. Thus, the AUV is totally unaware of the presence of the obstacle. The resulting planned path is a straight line to the dock. When the sonar returns a contact, the Perception task object adds a new obstacle, with default dimensions of 10x10x10 metres, to the internal map which consequently invalidates the current path. The controller must then plan a new path, taking into account the newly discovered obstacle. The dashed-line trajectory in Figure 36 results from this sequence of events.

The DOAA-generated trajectory illustrates the vehicle's behaviour when the internal map represents the stationary obstacle as a moving one. At first, the moving obstacle (as perceived by the controller) is outside all regions of concern (defined in Section 4.1.4), allowing the vehicle to follow its preplanned path. As the obstacle moves into a region of concern, the weighting formula forces the AUV to leave the path.

The oscillatory behaviour in the zx-plane (longitudinal) is likely due to the narrow width of the fuzzy subsets belonging to the DOAA-Depth rules. Adjacent fuzzy subsets which overlap by less than 25% can produce a chatter effect where the control signals oscillate between discrete values rather than changing smoothly.

One significant limitation of the DOAA is that it ignores obstacle dimensions in generating control signals. Instead it reacts to the proximity of the obstacle origin. This simplification makes the DOAA response useless when obstacle dimensions become large (>25 metres). To solve this problem, the map can represent a large, moving obstacle as multiple, smaller moving obstacles.

A better way to solve the problem is to configure the DOAA so that it avoids the *closest obstacle vertex* as well as the obstacle origin. Configuring the DOAA to respond like this is a simple matter of adding more rules (or knowledge) to the FIE. The matrix below could generate the additional rules:

TABLE 4. Additional Knowledge for Improving the DOAA

	Obstacle Origin Above Closest Vertex	**Obstacle Origin Below Closest Vertex**
Obstacle Origin to Left Of Closest Vertex	head to the right of the vertex & dive down	head to the right of the vertex & dive up
Obstacle Origin To Right Of Closest Vertex	head to the left of the vertex & dive down	head to the left of the vertex & dive up

Interestingly, if all obstacles are represented as moving, the dynamic obstacle avoidance algorithm may generate a shorter path to the goal than the path planning algorithm (which

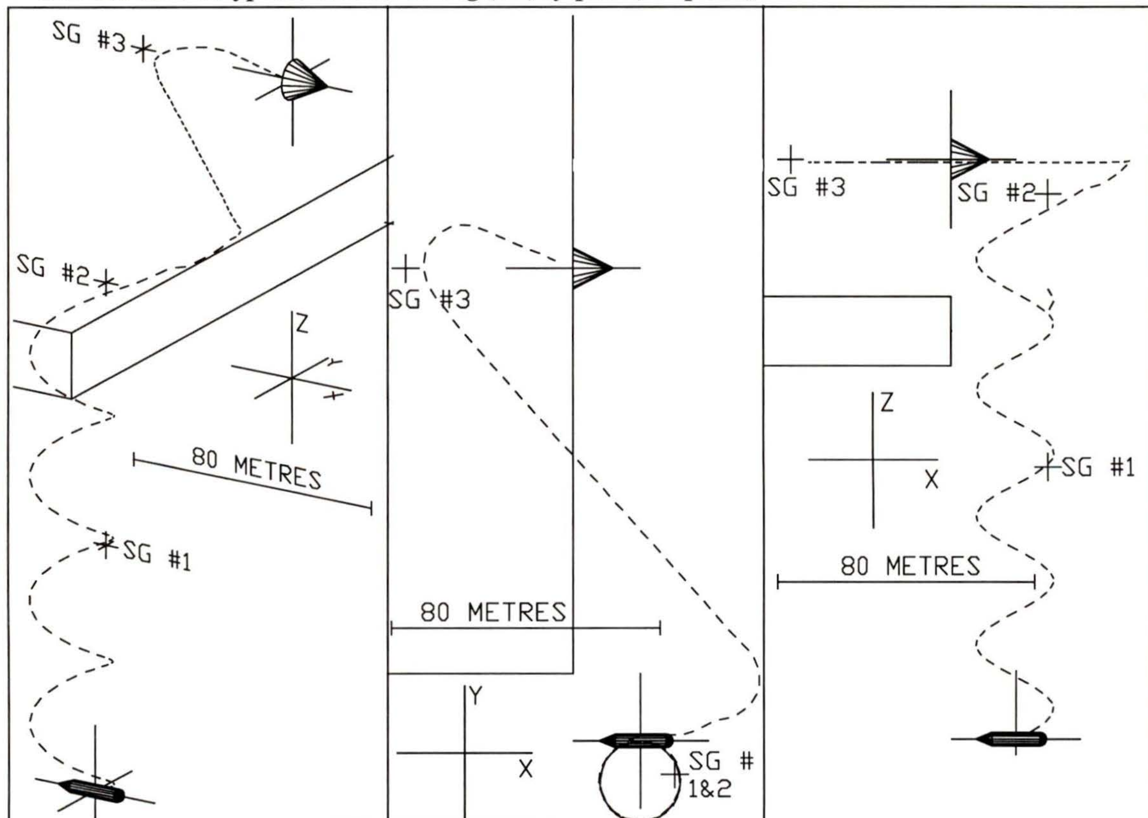
is based on static obstacles). This is due to the fact that the DOAA does not constrain the vehicle to pass through grown obstacle vertices - in fact, it has a similar effect as an instantaneous artificial potential field planner since it applies a fuzzily repulsive field to the obstacle.

5.4 Subgoal Under/Overshooting

Subgoal overshooting occurs when the AUV reaches the (x,y) coordinates of its goal without attaining the required depth (z). To attain the goal, it must intelligently maintain its (x,y) location and continue to pursue the required z coordinate. Waypoint overshooting is also observable when the vehicle reaches (z,x) or (z,y) coordinate pairs first, but the response is not as interesting (the vehicle stops diving and proceeds directly to the goal).

Figure 37 shows a mission in which the vehicle undershoots its first subgoal (SG #1). Starting from below the large obstacle, the AUV must first proceed out and up to a grown vertex. The vehicle cannot rise quickly enough to the first subgoal to satisfy all 3 constraints (x, y, and z) at once. It reacts by circling about the (x,y) location, continuously rising until it attains the first subgoal.

FIGURE 37. Waypoint Undershooting (3D, xy-plane, zx-plane)



Since the vehicle cannot pitch, it must move forward to ascend or descend. Therefore its circling actions are appropriate. If it had the capability, a better course of action would have been to pitch upwards and head straight to the subgoal. In general, a pitching capability would increase the control options available to the fuzzy low-level controller.

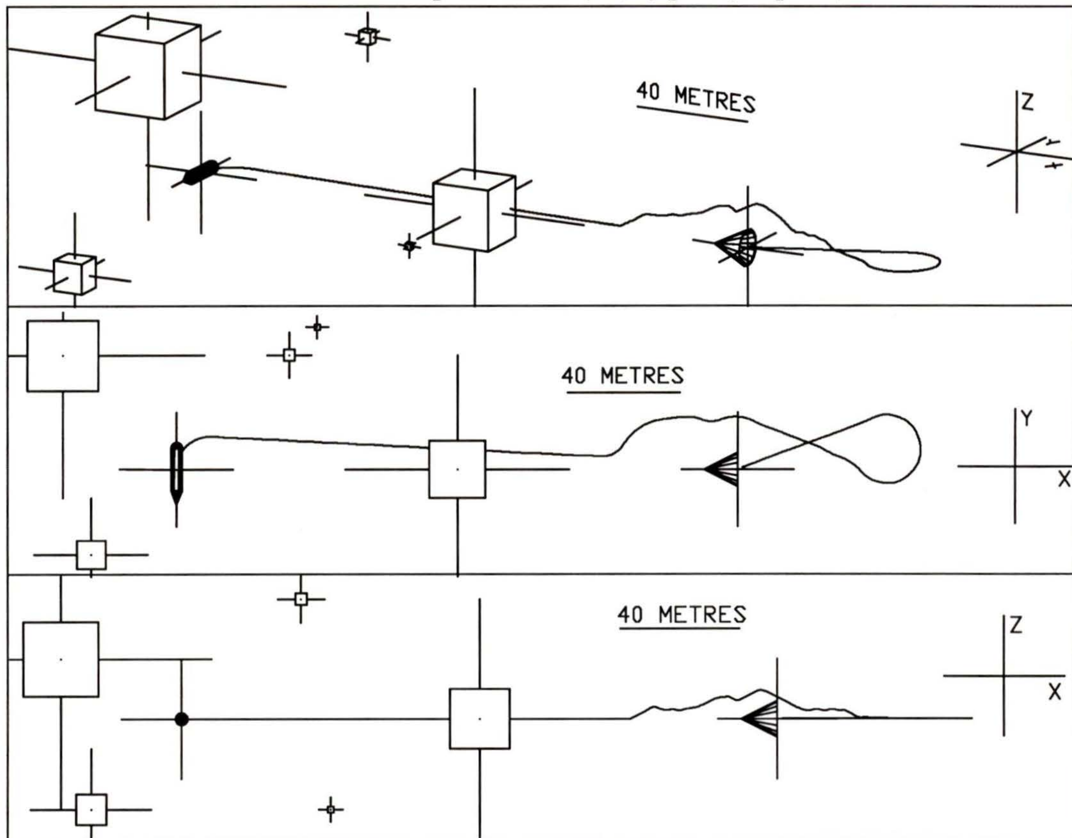
With the second subgoal (SG #2), the same phenomenon occurs. The submersible reacts in the same manner, circling up towards the subgoal. From the xy view in Figure 37, it is apparent that the grown obstacle vertices provide clearance to account for a tight turn by the vehicle. However, the margin is not exceptionally large and turns of a larger radius this close to a grown obstacle vertex would probably result in a collision.

An overshoot occurs at the safety location (SG #3). In this case, the AUV meets all 3 spatial constraints at the same time, but its forward velocity carries it past the subgoal. Due to the cornering algorithm, the vehicle started to turn towards the dock before reaching the safety location and avoided drastically overshooting the subgoal. Without the cornering algorithm, the overshoot would have increased, causing the vehicle to violate its path boundary. The vehicle would then have had to circle about the safety location until it improved its alignment with the safety location and the dock.

5.5 Map Saturation and Dock Avoidance

Figure 38 displays a trajectory which results when the map has been saturated with obstacles before beginning its mission. Map saturation occurs when the controller senses a new object but has no memory left in its map to accommodate the new information. While en route, the Perception task senses a new obstacle but is unable to update the internal map. The Reasoning and Action Tasks continue to operate, unaware of the new obstacle in front of the AUV and the vehicle eventually collides with the obstacle. Luckily for the vehicle, the obstacle's density is similar to the density of the water and the vehicle continues along unharmed (and unaware that it has collided). The next abnormal event occurs when the vehicle nears the dock on its way to the safe docking subgoal. The vehicle takes special measures to avoid a head-on collision with the dock (discussed below).

FIGURE 38. Dock Avoidance and Map Saturation (3D, xy plane, xz plane)



Map Saturation

Map saturation is a combination of poor sensor capabilities and limited map memory. The best way to solve this problem is to improve the sensors on board the vehicle. This will reduce the number of "false" obstacles residing in the map's memory. When there are many obstacles in the physical model, the memory limitation becomes more important. Swapping objects in and out of the map memory depending on locality to the vehicle is an option but this implies that the system has available unused memory somewhere else.

Dock Avoidance

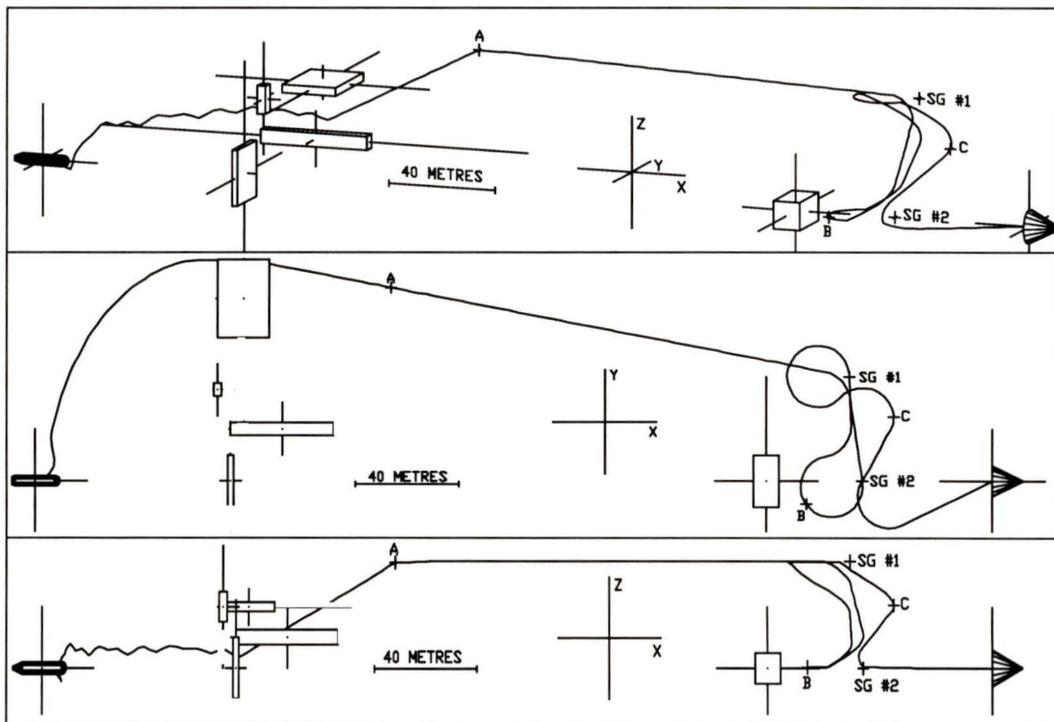
Currently, the vehicle avoids collisions with the dock by applying the DOAA to the dock origin. The DOAA disengages when the vehicle is approaching the dock with intent to dock (CurGoal is set to the dock entrance). Another way to avoid collisions with the dock would be to surround the dock with an obstacle-like box which the path planner had to avoid when finding a path to the dock.

The xy-plane view shows that the vehicle approaches the dock from a poor angle. A way to eliminate this wide angle approach is to reduce the size of the path boundary radius. If the radius were smaller, the vehicle would violate the path boundary as it was turning around and would have to return to the safety location rather than head straight for the dock. However, reducing the path boundary radius has implications in other areas. For instance, path violations (requiring new paths to be planned) would occur more often around dynamic obstacles. This would be undesirable in a real-time environment where processing speed is critical.

5.6 Multiple Dynamic Obstacles and Obstacle/Dock Proximity

Figure 39 shows a mission where the vehicle has attempted to avoid four obstacles it has classified as dynamic (though they are static in reality) and reach a dock which is close to a static obstacle.

FIGURE 39. Mission History (3D, xy plane, xz plane)



Response to Multiple Moving Obstacles

Initially, the vehicle dives down and to the left of the “dynamic” obstacle directly in front of it. This behaviour appears (in poor resolution) in the XZ plane view in Figure 39. However, as the vehicle is turning to the left, the path subgoal (with a +z coordinate) combined

with a new more important[27] obstacle (with a +z coordinate as well), causes the vehicle to regain some of its lost altitude. After this initial, transient behaviour, it begins a relatively stable trajectory around the outside of and underneath the outermost obstacle.

The DOAA focuses on the single most important dynamic obstacle and ignores other moving obstacles when it generates control signals to avoid a collision. Intuitively, this is not a robust or optimal algorithm for scenarios involving multiple dynamic obstacles. Depending on the geometry of the situation, the vehicle may avoid one obstacle but may collide with one or more of the others. One way to improve the DOAA is to have it output a weighted sum of the responses to *each* moving obstacle rather than just the response to the most important obstacle. A formula such as:

$$\mathrm{ConSig}_{DynamicObs} = \sum_{i}^{D} w_i \cdot \mathrm{ConSigs}_i \qquad \text{(EQ 37)}$$

where D is the number of dynamic obstacles, would improve the algorithm, but would not perfect it. For instance, using the above enhancement, if four obstacles approach the AUV from equal but opposite headings and depths, the control signals for each obstacle will cancel each other out and the AUV will continue on its path. The fact that the algorithm is vulnerable to certain situations does not imply that it is not useful - it is always possible to choose a scenario which confounds a controller, no matter how advanced it is. This is a possible topic for improvement in the future.

The DOAA in this controller thwarted many attempts to make it collide into four average-sized[28], static obstacles it had classified as dynamic. While the DOAA produces a non-optimal path (it should have initially headed to the right of the group of "dynamic" obstacles), it is quite robust. This is due to the three dimensional environment (which gives the vehicle many options for avoidance) and the large weight given to the DOAA when moving obstacles enter a region of concern.

Response To Static Obstacle/Dock Proximity

After avoiding the "dynamic" obstacles, the vehicle resumes its path towards the dock which is partially obstructed by a single static obstacle. Point A represents the time when

---

27. The dynamic obstacle's importance is defined in Section 4.3.4.

28. The DOAA avoids only obstacle origins, not obstacle surfaces or vertices. This means that large dynamic obstacles could (trivially) easily cause a collision. Table 4 suggests a simple solution to this problem. Obstacles in these simulations did not exceed 30 metres in length, width or height.

the submersible has attained the height of the first subgoal. It cruises to the first subgoal (SG #1). Once it attains the first subgoal, the AUV must descend quickly to reach the safety location (SG #2). On the first pass, the AUV is too far above the safety location (although the x and y coordinates are correct) and must reorient itself to approach the dock from a safe angle. To do this, it turns to its right with the aim of circling down to the safety location.

Here, as the vehicle is circling down to meet its subgoal, it enters the "grown" region of the static obstacle (Point B). At this point the current path becomes invalid and the controller must plan a new path. The problem arises when the path planning algorithm finds itself inside the grown obstacle: Since a grown obstacle surfaces bound the submersible in every direction, there is no valid path from the AUV to the safety location. The path planning algorithm breaks down and cannot supply the vehicle with a valid path. Consequently, the vehicle continues heading to the goal it had before the path became invalid, namely the safety location. This defect in the path planning algorithm should be rectified in future development.

At some point (slightly after Point B), the path planning algorithm is able to find a valid path from its location inside the grown obstacle to the dock. The resulting path must go through a vertex of the grown obstacle surface (as explained in Section 3.3.5). Since, relative to the center of the obstacle, the vehicle has positive x, y, and z components, the shortest path goes through the vertex represented by SG #1. On the second pass by SG#1, the submersible is too low to attain the subgoal. Consequently, it circles up to meet it.

After reaching SG #1 for the second time, the submersible turns and heads back down to SG #2, the next subgoal in its path to the dock. This time, due to the extra distance travelled in the sweeping turn (Point C), the submersible is able to descend long enough to attain SG #2 on the first pass. After attaining the safety location, it proceeds successfully to the dock.

5.6.1 Summary

The evaluation indicates that the intelligent motion controller is capable of docking an AUV in many differing scenarios. It is able to incorporate new information from the environment and act on it. It behaves "intelligently" in situations which are potentially confusing to it.

Chapter 6 Conclusions And Future Work

6.1 Conclusions

The AUV development system assists designers in modelling and evaluating the dynamics, sensors and intelligent controllers of unmanned submersibles. Its modular architecture allows experts to work on any aspect of AUVs without requiring expertise in other areas related to AUVs. The design tools included give the developer a head start by implementing common aspects of all AUVs as pre-programmed components. The designer is therefore able to start at an advanced stage in the design cycle.

The design tools provided in the development system will often be useful to developers. There will be cases however, where the developer bypasses the tools because their foundations (fuzzy logic in the FIE) or components (graph-based path planning in the map knowledge base or boxlike obstacles in the mission execution stage) are not deemed suitable. If the foundations of a tool are rejected, the developer will need to create all new functions and data structures. If only certain tool components are not desired, the template of the design tool is still available and the developer will be able to change parts of it as necessary.

Even if the design tools are not used, the framework for AUV modelling and evaluation will still be available. The existing code, in the form of the docking submersible and its controller, provides a working example for coordinating the intelligent controller with the submersible dynamics for a simulation.

The AUV designed as an example of using the development system is able to dock successfully in a wide variety of environments. Given its limited suite of sensors and actuators, the actions and reactions of the AUV are relatively intelligent. Nevertheless, it is not difficult to arrange a test environment which confuses the vehicle enough to make it behave inefficiently or dangerously.

Since the development system was successfully used to create an AUV which is capable of docking to an underwater dock, both the primary and secondary goals of this thesis have been met.

6.2 Future Work

The work in this thesis sets the framework for further study of all areas related to AUVs. As such, the development system gives a broad treatment to a wide variety of topics. While a good base for development now exists, several topics deserve closer attention.

Extend Fuzzy Inference Engine So That It Is Adaptive

Currently, the FIE class is not adaptive. Since it does not "learn", this tool neglects a significant aspect of intelligent control - the ability to learn from experience. Algorithms for adaptive fuzzy rules are plentiful.

Kosko describes an adaptive fuzzy logic controller which creates weights for a set of fuzzy rules by examining the firing frequency of rules in the original set [Kosko 91]. Rules which fire frequently are assigned higher weights than those which fire infrequently. In the fuzzy inference procedure, the rule outputs are weighted accordingly in the determination of the final output.

Improve Realism Of Simulation In Mission Execution Stage

One area which could benefit from future work is the modelling of the obstacles and medium in the mission execution stage. Currently, obstacles are restricted to be boxlike entities with sides parallel to the planes of the world axes. The medium is equally simplistic with no wave motion, current effects or other sources of noise. These limitations inhibit the realism of the simulation in the mission execution stage.

Future work could improve obstacle representations by expanding obstacles to be general polyhedra capable of having any orientation relative to a fixed axis. Contemporary wave and current action models could also be added to the Medium submodel.

Increased Generalization Of Map Knowledge Base

The current map knowledge base is tailored strongly to the vehicle's local surroundings with functions and data structures which are relevant to entities whose size is the same order of magnitude of the vehicle's. The map class is not general enough to be used in all instances requiring a map knowledge base. For instance, a camera image - also a type of map containing edges, vertices, and solid regions, cannot be easily represented with the current map tool. The functions to operate on these entities are also not available.

Also global maps, with continents, oceans and reefs, do not fit in well with the current map class (although in theory they can be represented).

One solution is to create new knowledge base tools which can process image information and global scale information. Ideally, there will be few additional tools though, as specialization may start to defeat the purpose of the design tool. A second option is to redesign the map knowledge base class so that it encompasses more of the characteristics of all types of maps.

Refinement Of The User Interface

At present there are some "bugs" in the user interface which can cause temporary interruptions in development. They are not serious bugs but are inconvenient. For instance, occasionally in the mission execution stage when the supervisor pushes the transmit button to transmit an instruction to the AUV, the simulation freezes and a new mission must be started. The reason for this may be that the controller receives the instruction (in the form of an interrupt) when it is in a sleep state, waiting for a message in its queue. The interrupt handler may not respond properly to interrupts when it is in a sleep state.

Another small bug is that a temporary file, created by the EASY5-generated Fortran program during the simulation, is not removed following the completion of the simulation. Although the file size is 0 bytes, the temporary file is a nuisance.

Reduction Of The Time And Memory Required For Animated Mission Display

To display an animated mission, the animation routine first runs through each 0.1 time increment in the mission, updating the positions of the AUV, targets and obstacles. With each update, it removes hidden lines and saves the resulting image as an AutoCAD slide file. Depending on the length of the mission, the number of entities in the environment, and the perspective of the view, hidden line removal can be very time consuming. As well the space required by each slide is non-trivial.

The solution is to move the animated mission display to an application which is designed for specifically for animation. AutoCAD offers some accessory programs which facilitate animation. Other packages are also available.

Graphical-Based Editing Of Knowledge Bases

Currently, to modify the knowledge bases prior to executing a mission, the user must edit ASCII files. These ascii files can be cryptic to a new user. Ideally, knowledge base editing

could be carried out with tools which presented and recorded information in a graphical format. For instance, fuzzy rules in the FIE could be graphically altered by lengthening or narrowing fuzzy subsets and map entities in the map knowledge base could be placed in space with a mouse. The overhead required for this improvement makes it a low-priority task.

Refinement Of Docking-Capable AUV

Since the dynamics and sensors of the docking-capable are extensively simplified, the simulations are not nearly ready for transfer to the real-world. The dynamics equations of the AUV should be refined substantially so that the vehicle behaviour more closely resembles the THESEUS vehicle. The low-level control aspects of the intelligent controller will also have to change correspondingly.

Bibliography

[Aucher 81] Aucher, M., translated by Lawrence D., Dynamique Des Sous-Marins, Sciences Et Techniques De L'Armement, Paris, 1981.

[AT&T 89] AT&T, AT&T C++ Language System Library Manual, Part Number 800-5148-10, Revision A of 22 February 1991.

[Akman 87] Akman, V., Unobstructed Shortest Paths in Polyhedral Environments, Springer-Verlag, 1987, ISBN 0-387-17629-2.

[Babcock 90] Babcock, P., Zinchuk, J., "Fault-Tolerant Design Optimization: Application To An Autonomous Underwater Vehicle Navigation System", *Proceedings of the Symposium on Autonomous Underwater Vehicles*, 34-43, 1992.

[Beer 84] Beer, F. P., Johnston, E.R., Vector Mechanics For Engineers, McGraw-Hill, ISBN 0-07-004438-4, 1984.

[Bellingham 90] Bellingham, J.G., Consi, T.R., Beaton, R.M., Hall, W., "Keeping Layered Control Simple", *Proceedings of AUV '90*, 3-8, 1990.

[Bellingham 91] Bellingham J.G., Consi, T.R., "State Configured Layered Control", *Proceedings of the 1st Workshop on Mobile Robots For Subsea Environments - International Advanced Robotics Programme*, 75-80, 1991.

[Booch 91] Booch, G., Object Oriented Design With Applications, Benjamin/Cummings Publishing Co, CA, ISBN # 0-8053-0091-0, 1991.

[Borland 90] Borland International Inc., Getting Started - Borland Turbo C++ Manual, 1800 Green Hills Road, Scotts Valley, CA, page 137, 1990.

[Bronshtein 85] Bronshtein, I., Semendyayev, K., Handbook Of Mathematics, Van Norstrand Reinhold Company, 1985.

[Brooks 86] Brooks, R.A., "A Robust Layered Control System for a Mobile Robot", *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, 14-23, 1986.

[Brooks 83] Brooks, R.A., "Solving the Find-Path Problem by Good Representation of Free Space", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 13, No. 3, 190-197, 1983.

[Brutzman 92] Brutzman, D., Kanayama, Y., Zyda, M., "Integrated Simulation For Rapid Development Of Autonomous Underwater Vehicles", *Proceedings of the Symposium on Autonomous Underwater Vehicle Technology*, 3-9, 1992.

[Bryan 91] Bryan, D., Gillchrist, B., Reich, R., "The Underwater Docking and Transfer of High Bandwidth Data Between An Unmanned Submersible And A Remote Underwater Platform", *Proceedings of Oceans '91*, 1275-1281, ISBN #0-7803-0202-8.

[Burdic 91] Burdic W.S., Underwater Acoustic System Analysis, Prentice Hall, ISBN 0-13-947607-5, 1991.

[Ditang 92] Ditang, W., Shouquan, K., Yulin, G., Yang, L., Dalu, L., "A Launch and Recovery System For An Autonomous Underwater Vehicle 'Explorer'", *Proceedings IEEE*, 1992.

[Feldman 79] Feldman, J., DTNSRDC Revised Standard Submarine Equations of Motion, David W. Taylor Naval Ship Research And Development Center, 1979.

[Feldman 75] Feldman, J., "State-Of-The-Art For Predicting the Hydrodynamic Charactersitcs of Submarines", *Proceedings of the Symposium on Control Theory and Navy Application*, U.S. Naval Postgraduate School, Monterey, CA, 1975.

[Fossen 91] Fossen T.I., Nonlinear Modelling And Control Of Underwater Vehicles, PhD Thesis, Norwegian Institute of Technology, 1991.

[Gertler 67] Gertler, M., Hagen, G., Standard Equations of Motion For Submarine Simulations, U.S. Naval Ship Research and Development Center, June 1967.

[Ghignone 92] Ghignone, S., Podesta, C., "Robot Crane For Underwater Vehicles Automatic Launching and Retrieval", *Proceedings of IEEE*, 282-290, 1992.

[Gwin 92] Gwin, R., Smith, J., "A Distributed Launch And Recovery System For An AUV And A Manned Submersible", citation undetermined.

[Hall 92] Hall, W.D., Adams, M.B., "Autonomous Vehicle Software Taxonomy", *Proceedings of the Symposium on Autonomous Underwater Vehicle Technology*, 49-64, 1992.

[Healey 92] Healey, A., Marco, D., "Experimental Verification Of Mission Planning By Autonomous Mission Execution and Data Visualization Using The NPS AUV II", *Proceedings of the Symposium on Autonomous Underwater Vehicle Technology*, 65-72, 1992.

[Hess 92] Hess D., Safford, D., Pooch, U., Williams, G., "FTMP: A Protocol for Distributed Fault Tolerant Operating System Services", *Proceedings of Symposium Autonomous Underwater Vehicle Technology*, 148-151, June 1992.

[Kosko 91] Kosko, B., Neural Networks and Fuzzy Systems, Prentice-Hall, Englewood Cliffs, NJ 07632, ISBN #0-13-611435-0.

[Lebans 93] Leban, G., Dinn, D., Brooke, B., "Launch & Recovery System for Untethered, Air-Breathing AUVs", Sea Technology Magazine, 19-23, July 1993.

[Lee 91] Lee Y, McGhee, R., "Automating The Construction Of Real-Time Software For An Autonomous Underwater Vehicle Through Prototyping", *Proceedings of the Seventh Symposium on Unmanned Untethered Submersible Technology*, 564-574, 1991.

[Lee 90] Lee, C., "Fuzzy Logic in Control Systems: Parts I & II", *IEEE Transactions on Systems, Man, And Cybernetics*, Vol. 20, No. 2, 404-435, March/April 1990.

[Lee 83] Lee, D.T., "Visibility of a Simple Polygon", *Computer Vision, Graphics, and Image Processing*, Vol. 22, 207-221, 1983.

[Liu 91] Liu. Y-H, Arimoto, S., "Proposal of Tangent Graph and Extended Tangent Graph for Path Planning of Mobile Robots", *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 1, 312-317, ISBN 0-8186-2163-X, April 1991.

[Lozano-Perez 79] Lozano-Perez, T., Wesley, M.A., "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles", *Communications of the ACM*, Vol. 22, No. 10, 560-570, 1979.

[Nelson 92] Nelson, E., et. al., "User Interface Design Strategies for AUVC Software Development", *Proceedings on Symposium on Autonomous Underwater Vehicle Technology*, 152-157, June 1992.

[Ornulf 91] Ornulf, J.R., "Object Oriented Software System For AUV Control", *Proceedings of the 1st Workshop on Mobile Robots For Subsea Environments - International Advanced Robotics Programme*, 15-24, 1991.

[Rae 92] Rae, G., Smith, S., "A Fuzzy Rule Based Docking Procedure for Autonomous Underwater Vehicles", *Proceedings of Oceans '92*, Vol. 2, 536-546, ISBN #0-7803-0838-7.

[Randell 93] Randell, C., A Review Of Autonomous Underwater Vehicle Software Control Architectures, Report For University Of Victoria, Dept. Of E&CE, 1993.

[Spong 89] Spong, M.W., Vidyasagar, M., Robot Dynamics & Control, John Wiley & Sons, ISBN 0-471-61243-X, 1989.

[Tanimoto 90] Tanimoto, S.L., The Elements Of Artificial Intelligence, Computer Science Press, 1990, ISBN 0-7167-8230-8.

[Wall 91] Wall, D.J., Collins, J.S., A Review of Technologies and Applications for Autonomous Underwater Vehicles, Report for Dr. J.S. Collins, University of Victoria, 1990.

[Watkinson 91] Watkinson, K.W., "Application of Aerospace Rendezvous and Intercept Guidance Techniques to Underwater Vehicles", *Proceedings of the Seventh International Symposium on Unmanned Untethered Submersible Technology*, 104-115, 1991.

[Yamaguchi 92] Yamaguchi, T., Takagi, T., Mita, T., "Self-Organizing Control Using Fuzzy Neural Networks", *International Journal of Control*, Vol. 56, No. 2, 415-439, August 1992.

[Zhang 92] Zhang, J., Bohner, P., "A Fuzzy Control Approach for Executing Subgoal Guided Motion of a Mobile Robot in a Partly-Known Environment", *To appear in a conference.*

[Zheng 91] Zheng, X., Jackson, E., "Object-Oriented Software Architecture For Mission Configurable Robots", *Proceedings of the 1st Workshop on Mobile Robots For Subsea Environments - International Advanced Robotics Programme*, 63-73, 1991.

APPENDIX A - Intelligent Controller Design Tools - Code

A.1 PRA Module

The PRA Module consists of C++ header files (shown below) and files containing member functions. Member functions are not shown, but they exist separate files. The tools have been adapted to the controller described in Chapter 4.

```
//========================================================================
// PRA.H - header file for PRA Module
// Note: Include "fie.h" before including this header
//========================================================================
#ifndef pra_h
#define pra_h
// Q_MSG CONSTANTS
#define MSG_BUF_SZ 10
// INSTRUCTION CONSTANTS
// maximum length of instruction component string
#define MX_INS_COM_LE 10
// PERCEPTION CONSTANTS
enum ObsDecision {NO_UPDATE,MOVING_O,GROWN_O,NEW_O,MOVED_O,BASE_MOVED};
enum sonar_stat {EMPTY, FULL};
// REASONING CONSTANTS
enum sit_awar {NOT_AS_PREDICTED, AS_PREDICTED};
enum av_stat {PASSIVE, ACTIVE};
enum sit_specific {NORMAL, PATH_INVALID, NEW_INSTR, REACHED_SGOAL};
// ACTION CONSTANTS
enum hfam_type {PATH_FOLLOW, DYNAMIC_OBS_AVOID, WEIGHT};
// COMM_RECEPTION CONSTANTS
enum mess_status {IN_MSG, NO_MSG, OUT_MSG};
// max number of instructions in buffer
#define MX_N_INSTR 10
//========================================================================
// Queue Message Class - defines a message for use on the queues
//------------------------------------------------------------------------
class Q_Msg : public object
{
 private:
 char c[MSG_BUF_SZ]; // character data buffer
 int i[MSG_BUF_SZ]; // integer data buffer
 float f[MSG_BUF_SZ]; // floating point data buffer
};
//========================================================================
// Instruction class - used by reasoning to rx msgs from support vessel
//------------------------------------------------------------------------
class Instruction : public object
{
```

```
 public:
 char action[MX_INS_COM_LE]; // DOCK, HOVER, WAY_TRANS, STOP, completed
 char location[MX_INS_COM_LE]; // dock 123, loc XYZ
 char extra1[MX_INS_COM_LE]; // possibly a time component or another
 char extra2[MX_INS_COM_LE]; // space component
};
//=========================================================================
// Perception class - part of the PRA Module
//-------------------------------------------------------------------------
class Perception : public task, public Interrupt_handler
{
 private:
 Map *PMap; // pointer to AUV's internal map

 qhead RTDataQh; // Sensor Data Queue head
 qtail *RTDataQt; // pointer to Sensor Data Queue tail
 Q_Msg Data_Present[5]; // array of queue messages
 Q_Msg *MAYNOTNEED; // pointer to queue message

 qhead *MapChangedQh; // Map Changd Queue head
 qtail *MapChangedQt; // Map Changed Queue tail
 Q_Msg Map_Chg[5]; // array of map changed messages

 int *sim_pid; // pointer to physical world simulation process id

 Sensor LA_SONAR; // lookahead sonar declaration
 float obj_range; // object range
 float beamwidth; // sonar beamwidth
 float loc_AV[4], loc_WO[4];// location wrt AUV and World coordinate sys-
tems
 float sos_h2o; // speed of sound in h20
 float max_range; // max range of sonar sensor
 float range2obj; // range to object
 float new_obj, moving_obj, old_obj;// flags

 Sensor LINPOS; // declarations for other sensors
 Sensor ANGPOS;
 Sensor LINVEL;
 Sensor ANGVEL;

 public:
//-----
// perception contructor
//-----
 Perception(Map *MAP, qtail *qt, int *t);
//-----
// real-time read sensor data
//-----
 void interrupt();
```

```
//-----
// load sensor package/interface into software
//-----
 void Recall_Sensors();
//-----
// standardize data
//-----
 void Stdize_Data();
//-----
// fuse data
//-----
 void Fuse_Data();
//-----
// compare data to map
//-----
 int MapData_Compare(float loc[4],int* bodtype,int* bodnum,int* trait,-
float* new_vals);
};
//=========================================================================
// Reasoning class - part of the PRA Module
//-------------------------------------------------------------------------
class Reasoning : public task, public Interrupt_handler
{
 private:
 Map *REA_MAP; // pointer to internal map
 Body *CUR_GOAL; // pointer to current goal
 Body *NEX_GOAL; // pointer to next goal
 Instruction I_BUFF[MX_N_INSTR]; // instruction buffer
 Instruction CUR_INSTR; // current instruction
 qhead *Instr_Qh; // instruction queue head
 Q_Msg *INSTR_MSG; // instruction message
 int num_ins; // number of instructions in queue

 int vehicle_status; // vehicle states indicator
 int situation; // situation indicator
 int SGNum; // subgoal number

 qhead *MapChangeQ; // pointer to Map Changed queue head
 qtail *GoalReadyQ; // pointer to Goal Ready Queue tail
 Q_Msg GoalReady; // Goal Ready message
 Q_Msg *MapChange; // pointer to Map Changed message

 public:
//-----
// reasoning constructor
//-----
 Reasoning(Map *tmap, qhead *qh, qtail *qt, Body *CUR, Body *NEX);
//-----
// instruction handler
```

```
//-----
 void interrupt();
//-----
// situation assessment and response preparation function declarations
//-----
 int Situation_Assessment();
 void Prepare_Response();
//-----
// path planner
//-----
 Path Gen_Waypoints_To(char *go_id);
//-----
// connected graph searcher (A* algorithm)
//-----
 void Srch_Path(struct link1 *link2, int *wp_vert);
};
//=====================================================================
// Action class - part of the PRA Module
//---------------------------------------------------------------------
class Action : public task
{
 private:
 FIE *MOCON; // pointer to FIE for low level motion control
 Map *CMap; // pointer to internal map

 qhead *GoalReadyQ; // pointer to Goal Ready Q head

 Body *CurGoal; // pointer to current goal
 Body *NexGoal; // pointer to next goal

 int my_pid; // controller process id
 int *sim_pid; // physical world simulation process id

 int num_actuators; // number of actuators
 Actuator RUDDER; // declare rudder actuator
 Actuator DIVEP;
 Actuator PROP;

 Signal GOAL_E; // error signal for goal
 Signal NGOAL_E; // error signal for next goal
 Signal DYNO_E[M_OBS]; // error signal for dynamic obstacle avoidance
 Signal DOCK_E; // error signal for dock avoidance

 Signal GOAL_C; // control signal for goal
 Signal NGOAL_C; // control signal for next goal
 Signal PATH_C; // control signal for path
 Signal DYNO_C[M_OBS]; // control signal for dynamic obs avoidance
 Signal DOCK_C; // control signal for dock avoidance
 Signal TOTAL_C; // final control signal
```

```
 public:
//-----
// action constructor
//-----
 Action(Map *tmap, FIE *f, qhead *qh, Body *CUR, Body *NEX, int *s);
//-----
// generate error signals
//-----
 void GenErrSig(Body *DES, Body *NEXT_DES);
//-----
//generate control signals
//-----
 void GenConSig(float depthvel, float yawvel);
//-----
// multiply control signals by gains
//-----
 void GenActSig();
//-----
// send actuator signals
//-----
 void SendActSig();
};
#endif
//=======================================================================
```

A.2 Map Class

The code below is the header file for the map class. Again, member functions are not shown.

```
//=======================================================================
// MAP.H - Header file for map class
//=======================================================================
#ifndef map_h
#define map_h
#define PI 3.14159
//-----------------------------------------------------------------------
enum bod_traits {DIMENSIONS, POSITION, VELOCITY};
enum dim {x, y, z};
enum status {absent = 999, present = 1};
enum rng {min, max};
enum dire {left, on, right};
enum lin_ang {ang,lin};
// BODY CONSTANTS
enum safety_factor {ZERO, SMALL, MEDIUM, BIG};
enum body_types {PATH, TAR, OBS};
enum quadrant_heading_flag {DONTCARE, FR2TO3, FR3TO2};
// maximum length of body id
```

```
#define M_BID_LE 3
// PATH CONSTANTS
enum bou_stat {EXCEEDED, INTACT};
enum vis {NONVISIBLE,VISIBLE};
enum path_validity {INVALID, VALID};
// maximum path id length
#define M_PID_LE 3
// maximum number of subgoals
#define M_SGOALS 20
// MAP CONSTANTS
enum map_stat {ASSIMILATED, CHANGED};
// maximum number of paths in map
#define M_PATHS 10
// maximum number of targets in map
#define M_TARGS 10
// maximum number of obstacles in map
#define M_OBS 10
// maximum number of links which can be searched in the tree graph
#define M_LINKS 300
// maximum number of visible vertices for a single node
#define M_NODE_VX 72
//========================================================================
// Coordinate class - 3 linear and 3 angular coordinates
//------------------------------------------------------------------------
class Coordinate
{
 public:
 float lin[3];
 float ang[3];

 public:
 float *Get_Lin(void); // get linear coordinate
 void Set_Lin(float *ptr); // set linear coordinate
 float *Get_Ang(void); // get angular coordinate
 void Set_Ang(float *ptr); // set angular coordinate
};
//========================================================================
// Body Class - defines an entity in three dimensional space
//------------------------------------------------------------------------
class Body
{
 public:
 char id[M_BID_LE]; // identifier
 Coordinate ACC; // acceleration coordinate
 Coordinate VEL; // velocity coordinate
 Coordinate POS; // position coordinate
 float dime[3]; // dimensions
 struct traits // possible traits of a body
 {
```

```
 float radius;
 float turn_rad;
 int virt;
 };
 public:
//-----
// test for non-zero velocity of body
//-----
 int IsMoving();
//-----
// get body vertices
//-----
 void Vertices(int SafetyFactor, float *vptr);
//-----
// return differences in x,y,z dimensions between two bodies
//-----
 void Delta(Body bod1, int HeadFlag, float *delta_ptr);
//-----
// return the closestvertex of one body to the calling body's origin
//-----
 float *ClosestPointOf(Body *REL_BOD);
//-----
// return relative depth,heading & range of one body origin to another
//-----
 void DeHeRa(Body *REL_BOD, float *DeHeRa_ptr);
};
//======================================================================
// Path Class - sequence of subgoals starting from a base, ending at goal
//----------------------------------------------------------------------
class Path
{
 public:
 char id[M_PID_LE]; // path id
 Body BASE; // base location of path
 Body GOAL; // path goal
 Body SG[M_SGOALS]; // path subgoals
 int num_sgoals; // number of subgoals in path
 float bound_radius; // path boundary radius
 float max_subgoal_sep; // maximum subgoal separation (not used)
 public:
//-----
// subgoal generator for moving targets (not used in this controller)
//-----
 void Gen_SG(Body Cur_Body, Body Goal_Body);
//-----
// indicate if vehicle is outside path boundary
//-----
 int Boundary_Status(Body TBOD);
};
```

```
//=============================================================================
// Map class - an entity composed of a base, targets, obtacles and paths
//-----------------------------------------------------------------------------
class Map
{
 public:
 Body BASE;
 Path CUR_PATH;
 Path STORED_PATH[M_PATHS];
 Body TARGET[M_TARGS];
 Body OBSTACLE[M_OBS];
 int n_s_paths, num_targs, num_obs; // counters for map entitities
 float obj_sep_dist; // min separ. dist between map entities
 float V2W[3][4]; // transformation matrix from AUV
// coordinates to world coordinates
 int data_status; // indicates if data changed map
 typedef struct link1 // link declaration for path planning
 { // see member function for details
 int num;
 float len[M_NODE_VX];
 int vx[M_NODE_VX];
 int gonogo;
 int pred;
 float f, g, h;
 };
 public:
//-----
// map constructors
//-----
 Map();
 Map(char m_id[7]);
//-----
// transformation function fron a body-based coordinates system to the
world
//-----
 void FromBody2World(Body *B, float f[4], float *t);
//-----
// Store new path
//-----
 void Store();
//-----
// index lookup function
//-----
 int Index(int type, char id[M_BID_LE]);
//-----
// function to check the visibility of a line in the map
//-----
 int ChkVis(int SafetyFactor, float start[3], float end[3]);
//-----
```

```
// returns a property of a map entitiy as requested
//-----
 float *Read(Coordinate ASPECT, int typ);
//-----
// adds, deletes, modifies a map object
//-----
 void Add(int bodtype, float pos[3]);
 void Delete(int bodnum, int type);
 void Modify(int bodtype, int bodnum, int trait, float new_val[3]);
//-----
// determines if path is still valid
//-----
 int Path_Validity(Path TEST_PATH, int SubGoalNum);
//-----
// determines if vehicle has violated the path boundary
//-----
 int Path_Boundary(Coordinate testpt, Path Testpath);
};
#endif
//======================================================================
```

A.3 FIE Class

The code below is the header file for the FIE class. Member functions are not shown.

```
//======================================================================
// FIE.H - Header file for Fuzzy Inference Engine class
//======================================================================
#ifndef fie_h
#define fie_h
//----------------------------------------------------------------------
enum rang {low, high};
// SUBSET CONSTANTS
// max subset id length + 1
#define M_SID_LE 6
// VARIABLE MAPPING CONSTANTS
// max number of subsets defined on a fuzzy variable
#define M_SUBS 7
// max subset name length + 1
#define M_SNAM_LE 8
// max variable map id length + 1
#define M_VID_LE 20
// RULE CONSTANTS
// max number of antecedents in a rule
#define M_ANTE 3
// max number of consequents in a rule
#define M_CONS 1
// max combination string length + 1 (the and/or's)
#define M_COMB 4
```

```
// max rule id length + 1
#define M_RID_LE 30
// FAM CONSTANTS
// max number of rules in a fam
#define M_RULES 30
// maximum FAM id length + 1
#define M_FID_LE 20
// HFAM CONSTANTS
// max number of FAMs in a HFAM
#define M_FAMS 4
// max length of HFAM ID + 1
#define M_HID_LE 20
// FIE CONSTANTS
// max number of HFAMS in an FIE
#define M_HFAMS 3
// max FIE ID length
#define M_FIEID_LE 30
// MISC CONSTANTS
//buffer length for skipping a line
#define LINE 130
//==================================================================
// Fuzzy Subset - a triangle with a base width, a max height and a peak
// width. Defines the characteristics of a fuzzy subset which
// will be applied to a fuzzy variable.
// must not be clipped to use Kosko's algorithm for fuzzy centroid
// defuzzification algorithm (see thesis)
//-------------------------------------------------------------------
class Fuzzy_Sub
{
 public:
 float base_width;
 float peak_width;
 float max_height;
 public:
//-----
// trivial constructor declaration is required if a non-trivial
// constructor is declared (such as the second member function)
//-----
 Fuzzy_Sub();
 Fuzzy_Sub(char s_id[M_SID_LE]);
//-----
// Deg_Of_Mem() returns the degree of membership which a "normalized"
// crisp input has to the Fuzzy Subset (see code for more details)
//-----
 float Deg_Of_Mem(float crisp_input);
//-----
// Display is a debugging Function (partially implemented)
//-----
 void Display();
```

```
};
//=================================================================
// Fuzzy Variable Mapping - Maps a variable into a set of fuzzy subsets
// (defined above). Parameters used are: the range of
// the variable, number of subsets in the mapping,
// and the subset itself along with a structure that
// contains a subset id and centroid for each subset.
// NOTE: Only one type of subset can be defined for the entire mapping.
//-----------------------------------------------------------------
class Fuzzy_Var_Map
{
 public:
//------
// range[2] is 2 dimensional to define range's upper and lower boundaries
//-----
 float range[2];
 int num_subs;
//-----
// fuz_val[] is an array of all subsets in the mapping. name is the
// subset id (NS, PB, etc) and xcentroid is the centroid of the subset in
// the range of the variable
//-----
 struct
 {
 char name[M_SNAM_LE];
 float xcentroid;
 } fuz_val[M_SUBS];
//-----
// One subset type can be defined for the variable.The above structure
// distributes instances of this type over the range of the
// variable, BUT there can only be one subset type.
//-----
 Fuzzy_Sub FUZ_SUB;
 public:
 Fuzzy_Var_Map();
 Fuzzy_Var_Map(char var_id[M_VID_LE]);
//-----
// Activation takes crisp input from the variable's range and the id of a
// fuzzy subset in the variables' range & returns the deg of membership
// of that crisp input to that fuzzy subset
//-----
 float Activation(char fuz_subset[M_SNAM_LE], float crisp_input);
};
//=================================================================
// Fuzzy Rule - a mapping between fuzzy variables. Antecedents
// and consequents are fuzzy variables. Antecedents can be combined in
// AND/OR fashion. Rule is of the form:
//
// IF ANTECEDENT[0] = ante_fuz_val[0] ante_combo[0]
```

```
// ANTECEDENT[1] = ante_fuz_val[1] ante_combo[1]
// .
// .
// THEN CONSEQUENT[0] = cons_fuz_val[0] AND
// CONSEQUENT[1] = cons_fuz_val[1]
// .
// .
//-----------------------------------------------------------------------
class Fuzzy_Rule
{
 public:
 Fuzzy_Var_Map ANTECEDENT[M_ANTE];
 char ante_combo[M_ANTE-1][M_COMB];
 char ante_fuz_val[M_ANTE][M_SNAM_LE];
 Fuzzy_Var_Map CONSEQUENT[M_CONS];
 char cons_fuz_val[M_CONS][M_SNAM_LE];
 int num_ante, num_cons;
 public:
 Fuzzy_Rule();
 Fuzzy_Rule(char rule_id[M_RID_LE]);
//-----
// Ante_Fire returns the _min_ degree of membership of a multidimensional
// crisp input to each inputs fuz value as defined in the rule. The _min_
// constraint implies, as seen in the member function code, that the OR
// antecedent combination is not supported
//-----
 float Ante_Fire(float *crisp_inputs);
};
//=======================================================================
// Fuzzy Associative Memory -(see Kosko for nomenclature) a bank of fuzzy
// rules all sharing common antecedents and consequents.
//-----------------------------------------------------------------------
class Fuzzy_AM
{
 public:
 Fuzzy_Rule RULE[M_RULES];
 int num_rules;
 public:
 Fuzzy_AM();
 Fuzzy_AM(char fam_id[M_FID_LE]);
//-----
// Ante_Fire returns a pointer to a list of degs of memberships to rules
// in the FAM. The input is a pointer to a list of crisp inputs.
//-----
 float *Ante_Fire(float *input);
//-----
// Defuzzify returns a crisp output value determined using the centroid
// algorithm by Kosko. The input is a pointer to a list of degrees of
// memberships to rules.
```

```
//-----
 float Defuzzify(float *deg_of_mem);
};
//==========================================================================
// Hyper FAM - collection of FAMS whose union of inputs
// is all the inputs of a device or a process
// and whose union of outputs are all the
// outputs of a device or a process. (!)
//--------------------------------------------------------------------------
class Hyper_FAM
{
 public:
 Fuzzy_AM FAM[M_FAMS];
 int num_fams;
 public:
 Hyper_FAM();
 Hyper_FAM(char h_id[M_HID_LE]);
//-----
// Ante_Fire returns and array of degs of memberships. One dimension is
// for all the FAMs in the HyperFAM,other dimension is for all the rules
// in each FAM
//-----
 void Ante_Fire(float input[M_FAMS][M_ANTE],float arr[M_FAMS][M_RULES]);
//-----
// Defuzzify returns a list of cristp outputs corresponding to each FAM
// in the HyperFAM.
//-----
 void Defuzzify(float arr[M_FAMS][M_RULES], float out[M_FAMS]);
//-----
// Fire is a compact way of calling Ante_Fire and Defuzzify together
// as is often required in Fuzzy Logic Control)
//-----
 void Fire(float input[M_FAMS][M_ANTE], float out[M_FAMS]);
};
//==========================================================================
// Fuzzy Inference Engine- collection of HFAMS whose outputs are combined
// in some manner to generate the FIE's final output
//--------------------------------------------------------------------------
class FIE
{
 public:
 Hyper_FAM HFAM[M_HFAMS];
 int num_hfams;
 public:
 FIE();
 FIE(char FIE_id[M_FIEID_LE]);
};
#endif
//==========================================================================
```

Appendix B - Knowledge Base Configurations For The Docking-Capable Intelligent Controller

B.1 Map Configuration

Below is the initialization file for the intelligent controller's internal map described in Chapter 4. The user initially configures this file through the user interface. After the mission begins, the map changes depending on what the intelligent controller experiences during the mission. The added comments do not normally appear in the file.

```
// map id (there may be more than one)
MAP1
// initial base location and orientation (one number for each axis)
000020.00 -00020.00 -00070.00 000003.00 000000.00 000090.00
// initial base linear and angular velocity (one number for each axis)
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
// number of known obstacles in the environment (restriction <=5)
1
// initial obstacle location and orientation (restriction:all angles =0)
-00150.00 000150.00 000050.00 000000.00 000000.00 000000.00
// obstacle linear and agular velocity(restriction:angular velocities=0)
000002.00 -00023.00 -00001.00 000000.00 000000.00 000000.00
// obstacle dimensions
000300.00 000300.00 000020.00
//buffer space for other obstacles when required
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00
// number of known docks in the environment (restriction <=3)
1
// dock id (there may be more than one)
123
// initial dock location and orientation
000000.00 000120.00 000100.00 000000.00 000000.00 000000.00
// dock linear and angular velocity
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
// buffer space for other docks
000
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
```

```
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
000000.00 000000.00 000000.00 000000.00 000000.00 000000.00
```

B.2 FIE Configuration

The files below constitute the FIEs in the intelligent controller. In this case, the files contain "knowledge" for one FIE, the low-level motion controller in the intelligent controller described in Chapter 4.

B.2.1 FIE Definition File - fiedef.inp

```
========================================================================
===== FUZZY INFERENCE ENGINE DEFINITION FILE
===== Syntax:
===== FIE ID, Number of HyperFAMS
===== HyperFAM ID1
===== :
===== HyperFAM IDN
========================================================================
===== Definition for Motion Control FIE
========================================================================
MOTION_CONTROL, 3
PATH_HFAM
DYNAMIC_OBS_HFAM
WEIGHT_HFAM
========================================================================
===== EOF
========================================================================
```

B.2.2 Hyper-FAM Definition File - fuzzhype.inp

```
========================================================================
===== HYPER-FUZZY ASSOCIATIVE MEMORY INPUT FILE
===== Syntax:
===== HyperFAM ID, number of FAMs
===== FAM ID1
===== :
===== FAM IDN
========================================================================
===== Weight HyperFAM -- Range to Dynamic Obstacles -> Weight
========================================================================
WEIGHT_HFAM,1
p_Range2Weight
========================================================================
===== Path HyperFAM -- Generates signals to stay on precomputed path
========================================================================
PATH_HFAM, 3
```

```
p_Head2Rudder
p_Depth2Divep
p_Range2Speed
=========================================================================
===== Dynamic Obstacles HyperFAM -- Generates signals to avoid moving
obstacles
=========================================================================
DYNAMIC_OBS_HFAM, 4
d_Head2Rudder
d_Depth2Divep
d_Range2Speed
d_HR2Importan
=========================================================================
===== EOF
=========================================================================
```

B.2.3 FAM Definition File - fuzzam.inp

```
=========================================================================
===== FUZZY ASSOCIATIVE MEMORY INPUT FILE
===== Syntax:
===== FAMID, number of rules
===== RuleID1
===== :
===== RuleIDN
=========================================================================
===== Path -- Yaw Velocity & Heading -> Rudder Position
=========================================================================
p_Head2Rudder, 17
p_VnHnsRpm
p_VnHzeRps
p_VnHpsRps
p_VnHpmRpm
p_VnHpbRpb
p_VzHnbRnb
p_VzHnmRnm
p_VzHnsRns
p_VzHzeRze
p_VzHpsRps
p_VzHpmRpm
p_VzHpbRpb
p_VpHnbRnb
p_VpHnmRnm
p_VpHnsRns
p_VpHzeRns
p_VpHpsRnm
=========================================================================
===== Path -- Depth Velocity & Depth -> Diveplane Position
=========================================================================
```

```
p_Depth2Divep, 21
p_VnDnbDnb
p_VnDnmDnm
p_VnDnsDns
p_VnDzeDps
p_VnDpsDpm
p_VnDpmDpb
p_VnDpbDpb
p_VzDnbDnb
p_VzDnmDnb
p_VzDnsDns
p_VzDzeDze
p_VzDpsDps
p_VzDpmDpb
p_VzDpbDpb
p_VpDnbDnb
p_VpDnmDnb
p_VpDnsDnm
p_VpDzeDns
p_VpDpsDps
p_VpDpmDpm
p_VpDpbDpb
========================================================================
===== Path -- Range -> Propeller Speed
========================================================================
p_Range2Speed, 4
p_RbiSbi
p_RmeSme
p_RsmSsm
p_RzeSze
========================================================================
===== Path -- Range -> Weight
========================================================================
p_Range2Weight, 4
p_RbiWze
p_RmeWsm
p_RsmWme
p_RzeWbi
========================================================================
===== Dynamic Obstacle -- Heading -> Rudder
========================================================================
d_Head2Rudder, 7
d_HnbRze
d_HnmRpm
d_HnsRpb
d_HzeRnb
d_HpsRnb
d_HpmRnm
d_HpbRze
```

```
==========================================================================
===== Dynamic Obstacle -- Depth -> Diveplane
==========================================================================
d_Depth2Divep, 7
d_DnbDze
d_DnmDpm
d_DnsDpb
d_DzeDpb
d_DpsDnb
d_DpmDnm
d_DpbDze
==========================================================================
===== Dynamic Obstacle -- Range -> Propeller Speed
==========================================================================
d_Range2Speed, 3
d_RsmSbi
d_RmeSme
d_RbiSsm
==========================================================================
===== Dynamic Obstacle -- Heading & Range -> Importance
==========================================================================
d_HR2Importan, 21
d_RbiHpbIze
d_RbiHpmIms
d_RbiHpsImb
d_RbiHzeIbi
d_RbiHnsImb
d_RbiHnmIms
d_RbiHnbIze
d_RmeHpbIms
d_RmeHpmImb
d_RmeHpsIbi
d_RmeHzeIbi
d_RmeHnsIbi
d_RmeHnmImb
d_RmeHnbIms
d_RsmHpbIbi
d_RsmHpmIbi
d_RsmHpsImx
d_RsmHzeImx
d_RsmHnsImx
d_RsmHnmIbi
d_RsmHnbIbi
==========================================================================
===== EOF
==========================================================================
```

B.2.4 Fuzzy Rule Definition File - fuzzrule.inp

```
===========================================================================
===== FUZZY RULE INPUT FILE
===== Syntax:
===== RuleID, NumAnte, NumCons, Ante Combo..., Cons..., AnteFuzVal,...,-
ConsFuzVal
===========================================================================
===== Path -- Yaw Velocity & Heading -> Rudder Position
===========================================================================
p_VnHnsRpm, 2, 1, yawvel and headi, rudde, NE, NS, PM
p_VnHzeRps, 2, 1, yawvel and headi, rudde, NE, ZE, PS
p_VnHpsRps, 2, 1, yawvel and headi, rudde, NE, PS, PS
p_VnHpmRpm, 2, 1, yawvel and headi, rudde, NE, PM, PM
p_VnHpbRpb, 2, 1, yawvel and headi, rudde, NE, PB, PB

p_VzHnbRnb, 2, 1, yawvel and headi, rudde, ZE, NB, NB
p_VzHnmRnm, 2, 1, yawvel and headi, rudde, ZE, NM, NM
p_VzHnsRns, 2, 1, yawvel and headi, rudde, ZE, NS, NS
p_VzHzeRze, 2, 1, yawvel and headi, rudde, ZE, ZE, ZE
p_VzHpsRps, 2, 1, yawvel and headi, rudde, ZE, PS, PS
p_VzHpmRpm, 2, 1, yawvel and headi, rudde, ZE, PM, PM
p_VzHpbRpb, 2, 1, yawvel and headi, rudde, ZE, PB, PB

p_VpHnbRnb, 2, 1, yawvel and headi, rudde, PO, NB, NB
p_VpHnmRnm, 2, 1, yawvel and headi, rudde, PO, NM, NM
p_VpHnsRns, 2, 1, yawvel and headi, rudde, PO, NS, NS
p_VpHzeRns, 2, 1, yawvel and headi, rudde, PO, ZE, NS
p_VpHpsRnm, 2, 1, yawvel and headi, rudde, PO, PS, NM
===========================================================================
===== Path -- Depth Velocity & Depth -> Diveplane Position
===========================================================================
p_VnDnbDnb, 2, 1, depvel and depth, divep, NE, NB, NB
p_VnDnmDnm, 2, 1, depvel and depth, divep, NE, NM, NM
p_VnDnsDns, 2, 1, depvel and depth, divep, NE, NS, NS
p_VnDzeDps, 2, 1, depvel and depth, divep, NE, ZE, PS
p_VnDpsDpm, 2, 1, depvel and depth, divep, NE, PS, PM
p_VnDpmDpb, 2, 1, depvel and depth, divep, NE, PM, PB
p_VnDpbDpb, 2, 1, depvel and depth, divep, NE, PB, PB

p_VzDnbDnb, 2, 1, depvel and depth, divep, ZE, NB, NB
p_VzDnmDnb, 2, 1, depvel and depth, divep, ZE, NM, NB
p_VzDnsDns, 2, 1, depvel and depth, divep, ZE, NS, NS
p_VzDzeDze, 2, 1, depvel and depth, divep, ZE, ZE, ZE
p_VzDpsDps, 2, 1, depvel and depth, divep, ZE, PS, PS
p_VzDpmDpb, 2, 1, depvel and depth, divep, ZE, PM, PB
p_VzDpbDpb, 2, 1, depvel and depth, divep, ZE, PB, PB

p_VpDnbDnb, 2, 1, depvel and depth, divep, PO, NB, NB
```

```
p_VpDnmDnb, 2, 1, depvel and depth, divep, PO, NM, NB
p_VpDnsDnm, 2, 1, depvel and depth, divep, PO, NS, NM
p_VpDzeDns, 2, 1, depvel and depth, divep, PO, ZE, NS
p_VpDpsDps, 2, 1, depvel and depth, divep, PO, PS, PS
p_VpDpmDpm, 2, 1, depvel and depth, divep, PO, PM, PM
p_VpDpbDpb, 2, 1, depvel and depth, divep, PO, PB, PB
========================================================================
===== Path -- Range -> Propeller Speed
========================================================================
p_RbiSbi, 1, 1, range, speed, BI, BI
p_RmeSme, 1, 1, range, speed, ME, ME
p_RsmSsm, 1, 1, range, speed, SM, SM
p_RzeSze, 1, 1, range, speed, ZE, ZE
========================================================================
===== Path -- Range -> Weight Factor
========================================================================
p_RbiWze, 1, 1, shrang, CgNgW, BI, ZE
p_RmeWsm, 1, 1, shrang, CgNgW, ME, SM
p_RsmWme, 1, 1, shrang, CgNgW, SM, ME
p_RzeWbi, 1, 1, shrang, CgNgW, ZE, BI
========================================================================
===== Dynamic Obstacles -- DynO_Heading -> Rudder
========================================================================
d_HnbRze, 1, 1, dyhea, rudde, NB, ZE
d_HnmRpm, 1, 1, dyhea, rudde, NM, PM
d_HnsRpb, 1, 1, dyhea, rudde, NS, PB
d_HzeRnb, 1, 1, dyhea, rudde, ZE, NB
d_HpsRnb, 1, 1, dyhea, rudde, PS, NB
d_HpmRnm, 1, 1, dyhea, rudde, PM, NM
d_HpbRze, 1, 1, dyhea, rudde, PB, ZE
========================================================================
===== Dynamic Obstacles -- DynO_Depth -> Diveplane
========================================================================
d_DnbDze, 1, 1, dydep, divep, NB, ZE
d_DnmDpm, 1, 1, dydep, divep, NM, PM
d_DnsDpb, 1, 1, dydep, divep, NS, PB
d_DzeDpb, 1, 1, dydep, divep, ZE, PB
d_DpsDnb, 1, 1, dydep, divep, PS, NB
d_DpmDnm, 1, 1, dydep, divep, PM, NM
d_DpbDze, 1, 1, dydep, divep, PB, ZE
========================================================================
===== Dynamic Obstacles -- DynO_Range -> Propeller Speed
========================================================================
d_RbiSsm, 1, 1, dyran, speed, BI, SM
d_RmeSme, 1, 1, dyran, speed, ME, ME
d_RsmSbi, 1, 1, dyran, speed, SM, BI
========================================================================
===== Dynamic Obstacles -- DynO_Range & DynO_heading -> DynO_Weight
========================================================================
```

```
d_RbiHpbIze, 2, 1, dyran and dyhea, dywgt, BI, PB, ZE
d_RbiHpmIms, 2, 1, dyran and dyhea, dywgt, BI, PM, MS
d_RbiHpsImb, 2, 1, dyran and dyhea, dywgt, BI, PS, MB
d_RbiHzeIbi, 2, 1, dyran and dyhea, dywgt, BI, ZE, BI
d_RbiHnsImb, 2, 1, dyran and dyhea, dywgt, BI, NS, MB
d_RbiHnmIms, 2, 1, dyran and dyhea, dywgt, BI, NM, MS
d_RbiHnbIze, 2, 1, dyran and dyhea, dywgt, BI, NB, ZE

d_RmeHpbIms, 2, 1, dyran and dyhea, dywgt, ME, PB, MS
d_RmeHpmImb, 2, 1, dyran and dyhea, dywgt, ME, PM, MB
d_RmeHpsIbi, 2, 1, dyran and dyhea, dywgt, ME, PS, BI
d_RmeHzeIbi, 2, 1, dyran and dyhea, dywgt, ME, ZE, BI
d_RmeHnsIbi, 2, 1, dyran and dyhea, dywgt, ME, NS, BI
d_RmeHnmImb, 2, 1, dyran and dyhea, dywgt, ME, NM, MB
d_RmeHnbIms, 2, 1, dyran and dyhea, dywgt, ME, NB, MS

d_RsmHpbIbi, 2, 1, dyran and dyhea, dywgt, SM, PB, BI
d_RsmHpmIbi, 2, 1, dyran and dyhea, dywgt, SM, PM, BI
d_RsmHpsImx, 2, 1, dyran and dyhea, dywgt, SM, PS, MX
d_RsmHzeImx, 2, 1, dyran and dyhea, dywgt, SM, ZE, MX
d_RsmHnsImx, 2, 1, dyran and dyhea, dywgt, SM, NS, MX
d_RsmHnmIbi, 2, 1, dyran and dyhea, dywgt, SM, NM, BI
d_RsmHnbIbi, 2, 1, dyran and dyhea, dywgt, SM, NB, BI
=======================================================================
===== EOF
=======================================================================
```

B.2.5 Fuzzy Variable Mapping File - fuzzvars.inp

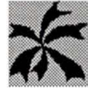
```
=======================================================================
===== FUZZY VARIABLE MAPPING INPUT FILE
===== Syntax:
===== VarID, RANGE[lo], RANGE[hi], FuzzySubsetID, NumSubsets, SubsetName: SubsetXCentroid
=======================================================================
===== Path Error Mappings ====
headi, -9.0, 9.0, ang, 7, NB: -9.0, NM: -6.0, NS: -3.0, ZE: 0.0, PS: 3.0, PM: 6.0, PB: 9.0
depth, -6.0, 6.0, dep, 7, NB: -6.0, NM: -4.0, NS: -2.0, ZE: 0.0, PS: 2.0, PM: 4.0, PB: 6.0
yawvel, -0.25, 0.25,yve, 3, NE: -0.25, ZE: 0.0, PO: 0.25
depvel, -1.0, 1.0, dve, 3, NE: -1.0, ZE: 0.0, PO: 1.0
range, 0.0, 60.0, ran, 4, ZE: 0.0, SM: 20.0, ME: 40.0, BI: 60.0
shrang, 0.0, 15.0, lra, 4, ZE: 0.0, SM: 10.0, ME: 20.0, BI: 30.0
CgNgW, 0.0, 1.0, gwe, 4, ZE: 0.0, SM: 0.33, ME: 0.67, BI: 1.0
===== Dynamic Obstacle Error Mappings =====
dyhea, -90.0, 90.0, dya, 7, NB: -90.0, NM: -60.0, NS: -30.0, ZE: 0.0, PS: 30.0, PM: 60.0, PB: 90.0
dydep, -30.0, 30.0, dde, 7, NB: -30.0, NM: -20.0, NS: -10.0, ZE: 0.0, PS: 10.0, PM: 20.0, PB: 30.0
dyran, 0.0, 30.0, dra, 3, SM: 10.0, ME: 20.0, BI: 30.0
dywgt, 0.0, 1.0, wei, 6, ZE: 0.0, SM: 0.2, MS: 0.4, MB: 0.6, BI: 0.8, MX: 1.0
===== Actuator Mappings =====
speed, 0.0, 15.0, spe, 4, ZE: 0.0, SM: 5.0, ME: 10.0, BI: 15.0
```

divep, -30.0, 30.0, div, 7, NB: -30.0, NM: -20.0, NS: -10.0, ZE: 0.0, PS: 10.0, PM: 20.0, PB: 30.0
rudde, -60.0, 60.0, rag, 7, NB: -60.0, NM: -40.0, NS: -20.0, ZE: 0.0, PS: 20.0, PM: 40.0, PB: 60.0

```
==========================================================================
===== EOF
==========================================================================
```

B.2.6 Fuzzy Subset/Degree Of Membership File - fuzzsubs.inp

```
==========================================================================
===== FUZZY SUBSET INPUT FILE
===== Syntex:
===== SubsetID, Basewidth, Peakwidth, Height
==========================================================================
ang, 3.4, 0.0, 1.0
dep, 2.3, 0.0, 1.0
dve, 1.4, 0.0, 1.0
yve, 0.45, 0.0, 1.0
ran, 25.0, 0.0, 1.0
lra, 12.5, 0.0, 1.0
wei, 0.3, 0.0, 1.0
gwe, 0.6, 0.0, 1.0
dya, 40.0, 0.0, 1.0
dra, 8.0, 0.0, 1.0
dde, 13.5, 0.0, 1.0
spe, 7.5, 0.0, 1.0
div, 12.5, 0.0, 1.0
rag, 25, 0.0, 1.0
==========================================================================
===== EOF
==========================================================================
```



APPENDIX C - USER MANUAL

This appendix describes how to use the Virtual Autonomous Vehicle Environment (VAVE). It assumes you are familiar with the topics discussed in this thesis.

C.1 Installation

To load the programs which make up VAVE into your home directory, read the file called install.README in the directory where VAVE currently resides. In the file .openwin-menu, add the following item to your Workspace menu (using the same syntax as the other items):

```
"VAVE"     exec $XTERM -n "VAVE Console" -title "VAVE Console" -e VAVE
```

Also create the following script file called VAVE (no extension) in your home directory:

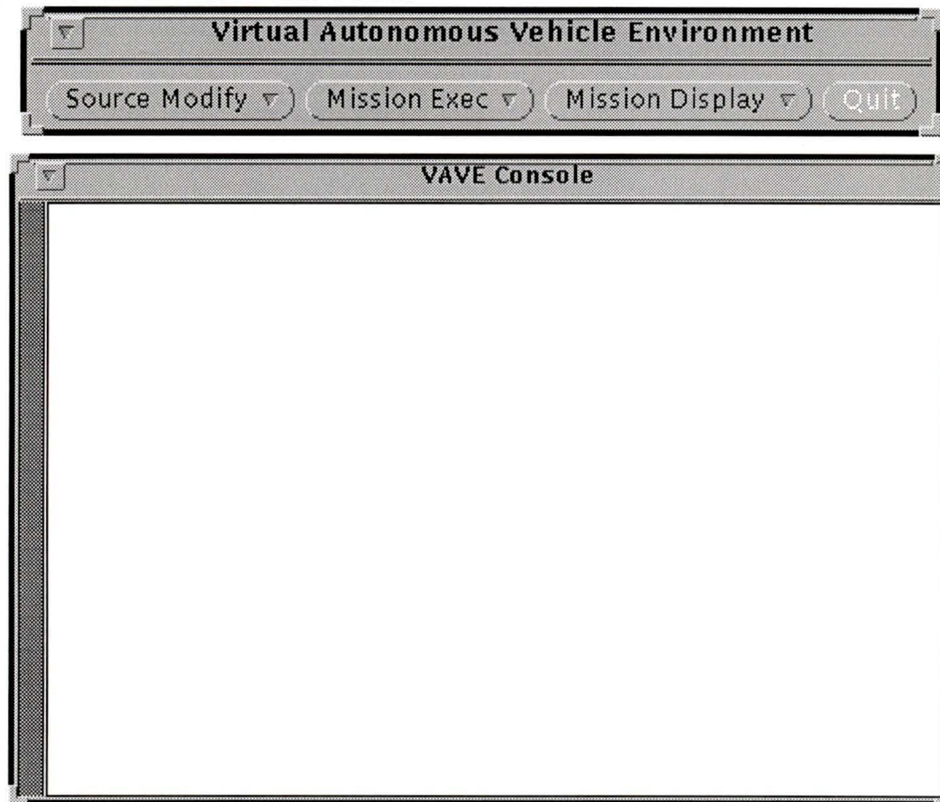
```
#! /bin/csh
# set up environment for VAVE
#===========================
setenv VHOME $HOME/Research
setenv VOVER $VHOME/Over
setenv VMODEL $VHOME/Model
cd $VMODEL
$VOVER/world1
```

This file also exists in the home directory of the current directory containing the VVE program. You may copy it over instead of creating the above script file by hand.

C.2 Running VAVE

To invoke the VAVE, click open the Workspace Menu (using the right mouse button in a free part of the background screen) and choose "**VAVE**". The VAVE Window and Console will appear. You enter commands to the VAVE through the Window. During a mission, VAVE outputs data to the VAVE Console. Figure 40 illustrates the VAVE Window and Console.

FIGURE 40. VAVE Window And Console



VAVE functions are grouped into the three design phases discussed in Chapter 1 - Source Code Modify, Mission Execution and Mission History Display. The following sections explain how to move through the three phases to design and test an intelligent controller.

C.2.1 Modifying Source Code

In this phase, you can edit and compile any aspect of the four program modules: the intelligent controller, the physical model, the user interface and the mission display module. Click the right mouse button on **Source Modify** in the VAVE Window to access the source code.

The Edit Physical World item is not supported in the user interface. However, to edit the physical world, you can invoke EASY5 - Dynamic System Simulator and load in the file - world1. User manuals for EASY5 can be obtained from Roger Kelly at the University Of Victoria.

C.2.2 Mission Execution

You enter this phase after compiling your source code changes. There are two steps involved in this phase - mission definition and mission execution.

Mission Definition

There are three steps to defining a mission. First you must configure the physical world. This is the world that the AUV moves around in. Click the right mouse button on the **Mission Execution** button and choose the **Edit Physical World** item. You have access to the parameters for the AUV, five obstacles and three targets. After applying the changes, you have configured the physical world. Be sure to hit <Return> if you manually enter a parameter, otherwise the user interface will not register your change.

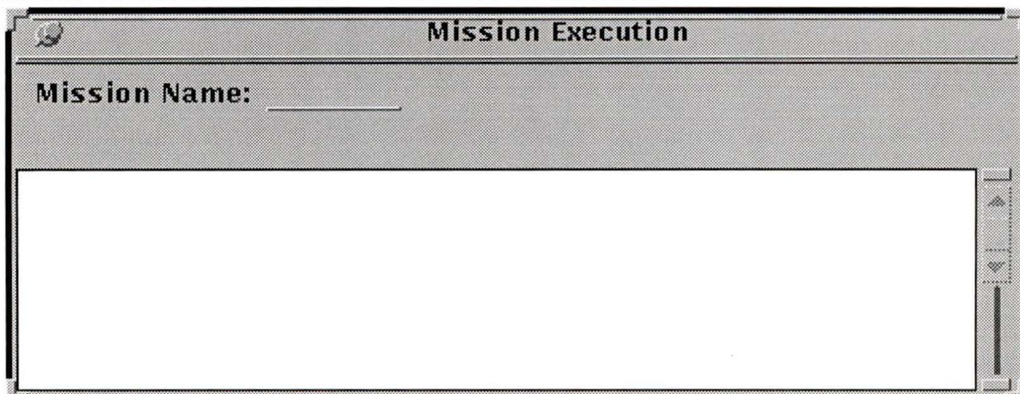
The second step is to configure the internal map(s) onboard the vehicle. Currently, the user interface allows the configuration of only one map, however you may change the user interface to configure more than one map. Using the same techniques that you used to configure the physical world, you can configure the internal map(s).

The third step is to configure the Fuzzy Inference Engines. Here, you can configure as many FIEs as necessary. Be sure to follow the syntax as outlined in the FIE files (which appear after you click on the **Edit FIE(s)** item). If you have FIE-related problems, check the syntax of your rules *very* carefully.

Mission Execution

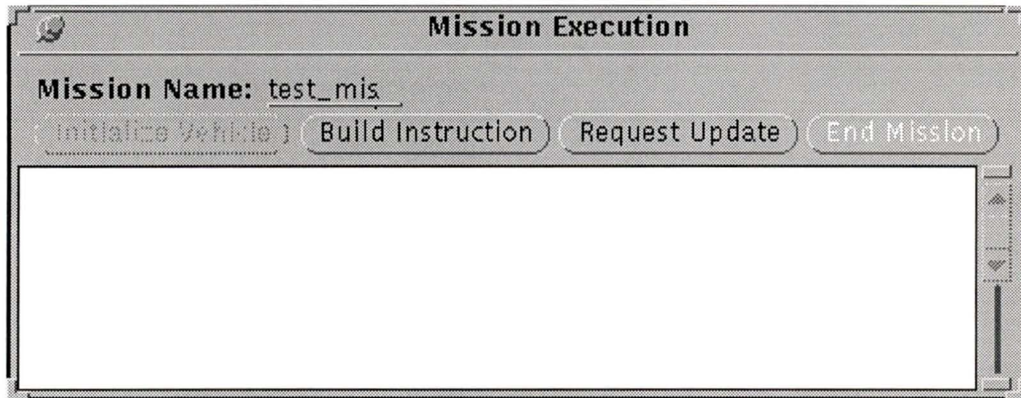
Once you have defined a mission, you are ready to execute it. Click the **Execute Mission** item in the Mission Execute Menu. The Mission Execute Window now appears (shown in Figure 41).

FIGURE 41. Mission Execution Window - Pre-Initialization



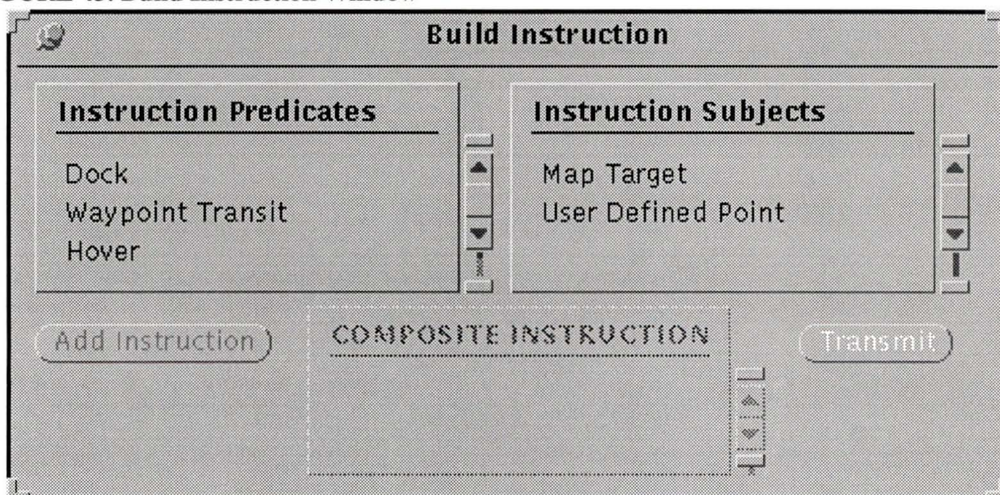
Enter the mission name in the space provided and hit <Return>. The **Initialize Vehicle** Button now appears. Click on it to start the simulated mission. At this point, data begins to scroll up in the VAVE Console. The data that appears depends on the diagnostic code that you have put in the physical model and controller code. When all systems on the vehicle are ready, the Initialize Vehicle Button will become inactive and three new buttons will appear in the Mission Execution Window. Figure 42 illustrates the new configuration.

FIGURE 42. Mission Execution Window - Post-Initialization



The vehicle will idle until you send it an instruction. To send the AUV an instruction, click on the Build Instruction button. The Build Instruction Window now appears (shown in Figure 43).

FIGURE 43. Build Instruction Window



The only supported feature in this window is the Transmit button. The Transmit button sends the instruction “DOCK 123, END” to the vehicle. “123” is a dock ID and it is essential that you have encoded a dock with id “123” in AUV’s internal map. If you have not

done this, the controller will not be able to execute the instruction. Click on the Transmit button to send the instruction to the AUV. The Build Instruction menu disappears.

The vehicle has now received the Dock instruction and will attempt to dock will the dock which has an id "123". Real-time output of the vehicle's progress will scroll up on the VAVE Console. When the vehicle completes the instruction, the simulation will end. Click on the End Mission button in the Mission Execution Window to stop the simulation. It is important that you formally end the mission if you wish to display the mission history.

C.2.3 Mission History Display

After a mission has been executed, you can display the mission history. To display the mission history, click right mouse button on the Mission Display Button on the VAVE Window and slide into the Mission Trajectory submenu (the Map History submenu is not currently supported).

You have the choice of displaying a quick history which shows a line segment version of the mission history, or a long history which shows the true vehicle and obstacle motion. The quick and long adjectives refer to the time it takes to display the histories. The long mission display takes an extremely long time and is not recommended unless you have a few days and a few gigabytes of memory. It will be useful for demonstrations to administrative people, but is not useful if you just want to evaluate the performance of the intelligent controller.

Choose the QUICK History option. VAVE invokes AutoCAD and runs a script file to display the mission history. After the mission is displayed, you have the option of returning to VAVE or editing the display. If you are done you evaluation of the mission, choose to exit the display program. If you would like to view the history from different perspectives, hit "E".

If you have chosen to edit the display, there are many ways to get different views of the mission history. The easiest way is to pull down the Display menu at the top of the AutoCAD Graphics Screen Window. Choose the VPoint 3D... option, followed by the upper right selection that resembles a crosshair. Choose a perspective by pushing the left mouse button when you have a suitable view. Now the mission is displayed from the new perspective. To view the history from another perspective, repeat the process. AutoCAD provides many more display options. Consult the AutoCAD reference manuals for more details.

To save the size of your GraphScreen and Text Screen, choose the Configure AutoCAD option in the Main Menu. Next, choose the Configure System Console Option. Position the two windows to your preference and follow the rest of the instructions.

VITA

Surname: Wall **Given Names:** Daniel John

Place Of Birth: Burlington, Ontario, CAN **Date Of Birth:** April 19, 1968

Educational Institutions Attended:

University Of Victoria	1991 to 1994
University Of Waterloo	1986 to 1991

Degrees Awarded:

B.A.Sc (First Class Honours)	University Of Waterloo

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title Of Thesis: A DEVELOPMENT SYSTEM FOR UNMANNED UNTETHERED SUBMERSIBLES

Author Signature:

Author Name: Daniel J. Wall

Date: April 29, 1994