

NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

DISSERTATION

Interaction Space Abstractions: Design Methodologies and Tools for Autonomous Robot Design and Modeling

Submitted by

Carl L. Kaiser

Mechanical Engineering

In partial fulfillment of the requirements
For the Degree of Doctor of Philosophy
Colorado State University
Fort Collins, Colorado
Fall 2009

UMI Number: 3400991

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 3400991

Copyright 2010 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

COLORADO STATE UNIVERSITY

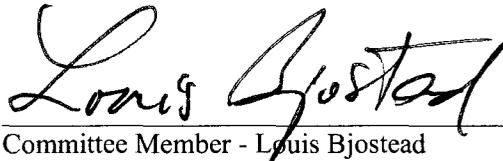
August 21, 2009

We hereby recommend that the dissertation prepared under our supervision by Carl Kaiser entitled, "Interaction Space Abstractions: Design Methodologies and Tools for Autonomous Robot Design and Modeling" be accepted as fulfilling in part requirements for the degree of Doctor of Philosophy.

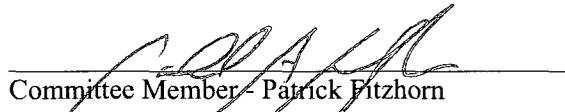
Committee on Graduate Work



Committee Member - David Alciatore



Committee Member - Louis Bjostead



Committee Member - Patrick Fitzhorn



Advisor - Wade O. Troxell



Department Head - Allan Kirkpatrick

ABSTRACT OF DISSERTATION

Interaction Space Abstractions: Design Methodologies and Tools for Autonomous Robot Design and Modeling

Current abstractions, design methodologies, and design tools are useful but inadequate for modern mobile robot design. By viewing robotics systems as an interactive and reactive agent and environment combination, and focusing on the interactions between the two, particularly those interactions that result in task accomplishment, one arrives at the interaction space abstraction.

The role of abstractions, formalisms and models are discussed, with emphasis on several specific abstractions used for robotics as well as the strengths and shortcomings of each. The role of design methodologies is also discussed, again with emphasis on several currently used in robotics. Finally, design tools and the use thereof are briefly discussed.

The concept of interaction spaces as an abstraction and a formalism is developed specifically for use in robot design. Types of elements within this formalism are developed, defined, and described. A formal nomenclature is introduced for these elements based on Simulink blocks. This nomenclature is used for descriptive models and the Simulink blocks are used for predictive models.

The interaction space abstraction is combined with the concept of exploration-based design to create a design methodology specifically adapted for use in descriptive modeling of autonomous robots. This process is initially developed around a simple wall-following robot, then is expanded around a multi-agent foraging system and an

urban search and rescue robot model, each of which demonstrates different aspects and capabilities of interaction space modeling as a design methodology.

A design tool based on iterative simulation is developed. The three specific examples above are used to perform quantitative simulation and the results are discussed with emphasis on determination and quantification of factors necessary for task accomplishment. These simulations are used to illustrate how to explore the design space and evaluate trade offs between design parameters in a system.

Carl L. Kaiser

Department of Mechanical Engineering

Colorado State University

Fort Collins, CO 80523

Fall, 2009

Acknowledgements

As with any significant undertaking, this dissertation would not have happened without the help and support of many people who are often not fully recognized. In particular I would like to thank the numerous friends as well as my parents without whom I may have given in to the temptation to lay this aside unfinished. Many people have listened sympathetically and offered both empathy and advice at times when I needed it.

I also owe a debt to my committee and in particular, my advisor Wade Troxell, for their advice, encouragement and teachings. This undertaking would also have been impossible without the financial support of the Department of Mechanical Engineering and the College of Engineering in the form of fellowships and assistantships.

Table of Contents

Table of Contents.....	1
Table of Figures.....	7
Table of Tables.....	11
Chapter 1 – Introduction.....	12
1.1 Abstractions.....	12
1.2 Abstractions for Robot Design.....	14
1.3 Interaction Spaces and Design Theories.....	15
1.5 Thesis Statement.....	16
1.6 Synopsis.....	17
Chapter 2 – Abstractions, Methodologies, and Tools for Robot Design.....	19
2.1 The Role of Abstraction in Design.....	19
2.1.1 Elements of Abstraction.....	21
2.1.2 System Dynamics.....	22
2.2 Design Methodologies.....	23
2.2.1 Top Down and Bottom Up Design.....	24
2.2.3 Exploration Based Design.....	25
2.2.4 Behavior Based Design.....	27
2.2.5 Generalized Design Methodologies.....	29
2.3 Robotic Design Abstractions.....	29
2.3.1 Task, Environment, and Agent.....	31
2.3.1.1 – TEA Definitions.....	32
2.3.2 Affordances.....	33

2.3.3 Petri Nets	34
2.3.4 Geometric Representations.....	34
2.3.5 Sense-Plan-Act	35
2.3.6 Schema.....	37
2.4 <i>Design Tools</i>	37
2.4.1 Geometric Simulations	38
Chapter 3 – Adding Structure to Interaction Spaces and Defining Notation for Interaction Space Models	40
3.1 Interaction Spaces	40
3.2 Reactive Agent/ Reactive Environment.....	44
3.3 Defining and Using Interactions and Cycles	46
3.4 Multi-Agent Systems	48
3.5 Understanding the Role of Information and Data Types.....	50
3.5.1 Binary Data.....	50
3.5.2 Discrete Data	51
3.5.3 Continuous Data	51
3.6 Basic Functional Block.....	51
3.6.1 State Blocks	52
3.6.1.1– Features.....	52
3.6.1.2 – Attributes	53
3.6.1.3 – Properties.....	53
3.6.2 Reaction Blocks	53
3.6.2.1 – Signals	54

3.6.2.2 – Information	54
3.6.2.3 – Behaviors	54
3.6.3 System Dynamic Implementation.....	55
3.6.4 Controlling Functional Blocks.....	56
3.7 Stochastic Blocks.....	56
3.7.1 Random Number Generator Blocks.....	57
3.7.2 Random Decision Maker Block.....	58
3.7.3 Noise Blocks	58
3.8 Tasks	59
3.8.1 Task Accomplishment for Perceptive Tasks	59
3.8.2 Measuring Task Accomplishment in Physical Tasks	60
Chapter 4 – Descriptive Models	61
4.1 Modeling Elements	61
4.1.1 Combining Functional Blocks	61
4.1.2 Developing Pseudo Code and Meanings	62
4.2 Steps in the Modeling Process.....	63
4.2.1 Defining the Problem.....	64
4.2.2 Creating Initial Cycles	65
4.2.3 Adding Additional Interactions	65
4.2.4 Adding Discrete Signals	66
4.2.5 Adding Analog Signals.....	67
4.2.6 Adding Uncertainty	67
4.2.7 Measuring Task Accomplishment.....	68

4.3 Refining the Model	68
4.4 Muramador Model	70
4.4.1 Defining the Problem.....	70
4.4.2 Creating Initial Cycles.....	71
4.4.3 Adding Additional Interactions to the Basic Model	74
4.4.4 Adding Discrete Signals	79
4.4.5 Adding Analog Signals.....	85
4.4.6 Adding Uncertainty to the Model	88
4.4.7 Determining Task Accomplishment	91
4.4.8 Suggested Additions for the Muramador Model	91
4.5 Multi-Agent Foraging Model	93
4.5.1 Defining the Problem.....	94
4.5.2 Addressing the Multi-Agent Issue	95
4.5.3 Foraging Model	95
4.5.4 Measuring Task Accomplishment	101
4.5.5 Improvements and Additions.....	102
4.6 Urban Search and Rescue Victim Detection Model	102
4.6.1 Defining the Problem.....	105
4.6.2 GSVD Model	106
4.6.3 Measuring Task Accomplishment	107
4.6.4 Additions and Improvements.....	109
4.7 Prototyping	109
Chapter 5 – Predictive Modeling.....	111

5.1 Developing Predictive Models	112
5.2 Implementing a Simulation.....	112
5.2.1 Simulink Implementation of Basic Functional Block	113
5.2.2 Simulink Implementation of Other Blocks.....	116
5.3 Measuring and Interpreting Results.....	117
5.4 Muramador Simulations	118
5.5 Foraging Simulations.....	121
5.6 Victim Detection Simulations.....	125
5.7 Exploration Based Design with Predictive Modeling	126
5.8 Discussion on Predictive Modeling.....	128
5.8.1 Multi-Variate Parameterized Simulations	128
5.8.2 Grounding the Simulations.....	128
5.8.3 Limitations of Simulink Simulation Environment	129
5.8.4 Comparison to Current Methods	130
5.8.5 Computational Complexity.....	130
Chapter 6 – Conclusions and Future Work	132
6.1 Summary.....	132
6.2 Conclusions.....	133
6.2.1 Interaction Spaces and Design.....	133
6.2.2 Descriptive Modeling	134
6.2.3 Predictive Modeling.....	135
6.3 Future Work.....	135
6.3.1 Real Robots.....	135

6.3.2 Dealing with Units.....	136
6.3.3 Standard Simulation Language.....	136
6.3.4 Expansion of Standard Modules.....	137
References.....	138
Appendix A – Implementation and Code for the Simulink Modeling Tools	143
A.1 – Simulink Implementation of a Stock	143
A.2 Simulink Basic Functional Block Implementation.....	148
A.3 – Simulink Uniform Random Number Generator Implementation.....	149
A.4 – Random Decision Making Block Implementation	150
A.5 – Discrete Random Number Generation Block	152
Appendix B – Code for Simulink Models	154
B.1 – Muramador Model	154
B.2 – Multi-Agent Foraging Model.....	158
B.3 – GSVD Model	165
Appendix C – PowerSim Code.....	176
C.1 – Muramador Program Listing (PowerSim)	176
C.2 Foraging Program Listing (PowerSim).....	178
C.3 Victim Detection Program Listing (PowerSim).....	184

Table of Figures

Figure 1 - Rosen Model	20
Figure 2 - Types of Abstraction.....	21
Figure 3 - Basic System Dynamic Elements	23
Figure 4 - Top Down and Bottom Up Design	25
Figure 5 - Exploration Based Design.....	26
Figure 6 - Vertical Robot Decomposition [1].....	28
Figure 7 - Level of Abstraction and Type of Design Process for Various Robot Abstractions	30
Figure 8 - Task Environment and Agent Abstraction with Interaction Spaces	32
Figure 9 - Sense, Plan, Act Abstraction.....	36
Figure 10 - Sense, Perceive, Plan, Act Abstraction.....	36
Figure 11 - Task, Environment, and Agent Abstraction with Interaction Spaces	41
Figure 12 - Level of Abstraction and Top Down vs. Bottom Up Characteristics of the Interaction Space Abstraction Relative to Other Robot Design Abstractions.....	43
Figure 13 - Quadrant abstraction of Interaction Spaces	45
Figure 14 - Interaction Space Cycles.....	47
Figure 15 - Multi-Agent Quadrant abstraction	49
Figure 16 - Basic Functional Block	52
Figure 17 - System Dynamic Implementation of a Basic Functional Block	55
Figure 18 - Uniform Random Number Block.....	57
Figure 19 - Discrete Uniform Random Number Block	58
Figure 20 - Random Decision Making Block.....	58

Figure 21 - Standard Cycle Abstraction	62
Figure 22 – Creating an Initial Interaction Space Model	64
Figure 23 - Options to Refine the Initial model.....	69
Figure 24 - Basic Cycle for the Muramador Model	72
Figure 25 - Muramador Model with Additional Interactions	75
Figure 26 - Muramador Model with Additional Interactions	77
Figure 27 - Muramador Model with Discrete Signals	80
Figure 28 - Muramador Model Additional Discrete Elements.....	83
Figure 29 - Muramador Model with Analog Signals.....	86
Figure 30 - Full Muramador Model.....	89
Figure 31 - Foraging Robots as Implemented by Krieger and Billeter [32].....	94
Figure 32 - Foraging Agent Block.....	96
Figure 33 - Foraging Agent Block Internal Structure.....	97
Figure 34 - Environmental Reaction Block	98
Figure 35 - Environmental Reaction Block Internal Structure	98
Figure 36 - Environmental Properties Block.....	99
Figure 37 - Environmental Properties Internal Structure	100
Figure 38 - Multi-Agent Foraging Model.....	101
Figure 39 - Typical RoboCup Terrain and a Typical Victim with the Good Samaritan in front of it.....	104
Figure 40 - Additional Typical RoboCup Terrain with the Good Samaritan	104
Figure 41 - Crowd at the RoboCup Competition in 2006	105
Figure 42 - GSVD Framework Model with Basic Element	107

Figure 43 - Simulink Implementation of a Basic Functional Block.....	113
Figure 44 - Simulink Implementation of a Basic Functional Block.....	114
Figure 45 - Dialog Box to Set the M-file.....	114
Figure 46 - Simulink Stock.....	115
Figure 47 - Dialog Box to Input the Initial Value of the Stock	115
Figure 48 - Random Decision Making Block.....	116
Figure 49 - Random Decision Block Dialog Box.....	116
Figure 50 - Random Number Generation Block.....	117
Figure 51 - Discrete Uniform Random Number Generator.....	117
Figure 52 - Muramador Model of the Presence of a Wall (Time Units are Arbitrary)	119
Figure 53 - Muramador Cumulative Wall Time (Units are Arbitrary but Consistent)	119
Figure 54 - Muramador Instantaneous Wall Distance (Units are Arbitrary).....	120
Figure 55 - Average Distance from the Set Point.....	121
Figure 56 - Agent Foraging Output (Units are Arbitrary).....	122
Figure 57 - Agent Object Found Output (Units are Arbitrary).....	122
Figure 58 - Individual Agent Energy Level.....	123
Figure 59 - Instantaneous Nest Energy Level for a Typical Power Sim Run.....	123
Figure 60 - Maximum Number of Times Steps (5000 possible) to Complete Nest Energy Loss (out of 20 runs)	124
Figure 61 - Average Nest Energy as a Function of Agent Energy Usage and the Value of an Energy Module	125

Figure 62 - Average Number of Victims Found Based on Environmental Noise and Sensor Effectiveness.....	126
Figure 63 - Exploration Based Design with Predictive Modeling.....	127
Figure 64 – Stock.....	143
Figure 65 - S-Function Dialog Box for a Stock.....	144
Figure 66 - Stock User Dialog Box for Setting the Initial Value	144
Figure 67 - Stock Mask Initialization.....	145
Figure 68 - Basic Functional Block Implementation.....	148
Figure 69 - Uniform Random Number Generator Block Internal Structure	149
Figure 70 - Uniform Random Number Generator Block Mask Initialization	150
Figure 71 - Internal Structure of a Random Decision Making Block.....	151
Figure 72 - Random Decision Maker Function Dialog Box.....	151
Figure 73 - Random Decision Maker Block Mask Parameter Set Up.....	152
Figure 74 - Discrete Uniform Random Number Generator Block Internal Structure	152
Figure 75 - Discrete Uniform Random Block Mask Initialization.....	153
Figure 76 - Muramador Framework Model without Basic Modeling Agent	154
Figure 77 - PowerSim Muramador Model.....	176
Figure 78 - Foraging Agent PowerSim Model	178
Figure 79 - Foraging Agent Environment PowerSim Model	179
Figure 80 - Left Side of the USAR PowerSim Model.....	184
Figure 81 - Right Side of the USAR PowerSim Model.....	185

Table of Tables

Table 1 - Muramador Basic Cycle.....	74
Table 2 - Muramador 1st Set of Additional Interactions.....	76
Table 3 - Muramador 2nd Set of Additional Interactions.....	79
Table 4 - Muramador 1st Discrete Model.....	82
Table 5 - Muramador Model Additional Discrete Elements	85
Table 6 - Muramador Model with Analog Blocks.....	88
Table 7 - Muramador Model with Uncertainty.....	91

Chapter 1 – Introduction

Robotics in its current form has been enabled by the digital computer.

Steady improvements in computing and other technologies such as sensors and actuators have led to widespread use of robotics in many tasks. Other tasks have remained relatively free of robotic involvement on any large scale; in some cases because relatively little effort has been made, and in other cases because effective robots have eluded designers despite substantial efforts.

Each robot operates within an environment; these environments can range from carefully engineered to relatively unstructured and uncertain. Generally two approaches exist to dealing with robots in complex environments. Where practical, one can seek to reduce the effective complexity of the environment. This has generally been the case with industrial robots and many research robots. In many cases, redesigning the environment is impractical or undesirable. It is with these cases that the remainder of this dissertation will be concerned.

1.1 Abstractions

If, as has been discussed, an environment cannot or should not be modified, it is necessary to find a way of understanding the environment. It is also necessary to develop an abstraction of the task if the task is significantly complex. Moreover, it is likely that any robot capable of performing a “complex” task would itself be difficult to understand without some tool to assist in description and understanding.

Abstraction is a generalized tool for understanding complex phenomenon. Formally introduced in Chapter 2, abstraction can generally be thought of as a mental

model. Abstraction is used throughout engineering design. As examples, consider the concept of current symbolizing a flow of electrons (in and of itself an abstraction of a more complicated physical reality) or the concept of enthalpy in thermodynamics, which is actually a more abstract, and for some situations more useful, way of expressing probabilistic movement and behavior of atoms. The behavior of the atoms is in and of itself a simplification of the interplay of various quarks and subatomic forces.

The examples above represent formal abstractions. It is also possible to have informal abstractions. Informal abstractions are more internalized mental models; for example, most children develop the abstraction that throwing a ball harder results in it flying farther. This has nothing to do with the formalized abstraction of projectile motion, or with the more complex abstraction of various gravitational and aerodynamic theories.

Informal abstractions are essential to everyday life. Individuals rely on generalized mental models to anticipate the effects of their actions. Likewise, modern engineering relies on more formal abstractions to predict behavior of the surrounding world and thus design devices that work. Although the devices work in the real world, it would be difficult for the engineer to deal with subatomic forces and particles while designing a building. Abstraction allows these effects to be aggregated and dealt with on a macroscopic scale.

Formal abstractions such as those most commonly used in engineering are well documented and have been evaluated experimentally to reveal limitations, such as the breakdown of Newtonian physics at speeds that are a significant fraction of the

speed of light. The risk of all abstractions, but especially informal abstractions, is applying them in situations where they are invalid. To return to the abstraction of throwing the ball, an unstated limitation of this abstraction is that the ball must leave the hand traveling in the right direction. Most adults unconsciously add this to their movements when throwing a ball, but watching a two-year-old quickly reminds one that this is a refinement of earlier childhood abstractions. More formal abstractions can also suffer from this limitation as evidenced by unexpected failures of various devices from the Tacoma Narrows bridge to the space shuttle Columbia. In general, the more informal the abstraction the more risk there is of applying it incorrectly or of two individuals applying it differently.

In addition to allowing for prediction, abstraction also facilitates communication and documentation. For well over a century, the three-view dimensioned drawing was the engineering communication tool of choice for mechanical objects. These drawings were not physically the objects but rather abstractions of the objects. The abstractions were not needed to design the objects, but rather to document and communicate the form of the object. This abstraction was only useful for communication because it was a formal abstraction with an agreed-upon relationship to the real world.

1.2 Abstractions for Robot Design

To design a complete robotic system it is necessary to consider the task, the environment, and the robot. Moreover, it is necessary to consider (and therefore in a complex system, abstract) the interactions between these three elements. After introducing some necessary concepts in Chapter 2, the remainder of this dissertation

will focus on the interactions between the task, the environment, and the agent; how these interactions can be abstracted; and how those abstractions can be used.

1.3 Interaction Spaces and Design Theories

The process of developing a theory of design in a particular field is largely related to developing correct formal abstractions and knowing when and how to apply each one systematically so that gaps are not created. The abstractions themselves often come from the physical sciences, but can also come from engineering practice; for example, the behavior-based architecture for robotics proposed by Brooks [1] is an abstraction for how to build a robot control system. Also necessary is the design process (abstractions of how to undertake a design) for a particular field. As discussed above, any of these abstractions can be either formal or informal or some combination of the two. As a design theory matures, these abstractions become more formalized. In general this process leads to more efficient and successful designs.

An interaction space is the set of all possible interactions between the robot and the environment. [2,3] The goal of robot design is to create a system that will act in that portion of the interaction space that will result in accomplishing the task. Interaction spaces focus specifically on the features and reactions of the agent and environment that trigger the desired interactions. Interaction spaces will be developed in further detail in Chapter 2.

Interaction spaces are used to more formally abstract the process of task accomplishment within a robotics system. The interaction space in and of itself is an abstraction but is primarily intended to help a designer apply other existing abstractions to a design.

As it currently exists, the interaction space abstraction only allows the designer to organize the abstractions that make up the interaction space. Interaction spaces do not currently help a designer to develop these abstractions, decide what abstractions should be used, or help to ensure a complete overall picture of the system. These tasks require a design process also sometimes referred to as a design framework. Such a process or framework does not currently exist around interaction space modeling. Much of the remainder of this dissertation will focus on developing such a framework.

Abstractions used for documentation and communication of robot design are common and often overlap with existing design fields; however, communication of the interaction between the robot and the environment and the relationship of those interactions to task accomplishment is generally not well documented in a formalized abstraction. Interaction space modeling is intended to bridge this gap.

1.5 Thesis Statement

By viewing robotics systems as an interactive and reactive agent and environment combination, and focusing on the interactions between the two, particularly those interactions that result in task accomplishment, the abstraction of interaction space models can be developed.

Interaction space modeling (based on the interaction space abstraction) can be used inside a formal framework with both an agent and an environment state represented, as well as agent and environment reactions. By defining interaction cycles between these components, a designer can formalize knowledge and assumptions about the interaction of the agent and environment as well as task

accomplishment. Standard functional blocks can be used to implement these models and should be added iteratively in a bottom up fashion to help the designer implement an exploration-based design process and provide a design methodology.

Further expansion of the concept of interaction space modeling combined with a mathematical framework provided by system dynamics can lead to predictive models that function as design tools. These design tools can provide both qualitative and quantitative insight into individual requirements necessary for system level task accomplishment.

By considering the agent and environment as equal reactive systems, and by iteratively refining the understanding of task accomplishment as an interaction between the two, the focus remains on the system level design instead of clever engineering or technology.

1.6 Synopsis

Chapter Two of this dissertation focuses on the fundamentals of formalized design. The role of abstraction is discussed, with emphasis on several specific abstractions used for robotics as well as the strengths and shortcomings of each of these. The role of design methodologies is also discussed, again with emphasis on several currently used in robotics. Finally, design tools and their use are discussed briefly.

Chapter Three of this dissertation uses the concept of interaction spaces as an abstraction, and formally develops the abstraction specifically for use in robot design. Types of elements within this abstraction are developed, defined, and described. A

standard nomenclature is introduced that is used throughout the remainder of the dissertation.

Chapter Four combines the abstraction described and developed in Chapter Three with the concept of exploration-based design to create a design methodology specifically adapted for use in descriptive modeling of autonomous robots. This process is initially developed around a wall-following robot, a multi-agent foraging system, and an urban search and rescue robot model, each of which demonstrates different aspects and capabilities of interaction space modeling as a design methodology.

Chapter Five takes the interaction space abstraction from Chapter Three and the interaction space methodology from Chapter Four and creates a design tool based on iterative simulation. The three specific examples from Chapter Four are used to perform quantitative simulation and the results are discussed with emphasis on determination and quantification of factors necessary for task accomplishment.

Chapter Six Reviews the new work presented in this dissertation, discusses the conclusions that can be drawn from this work, and suggests future avenues of research to capitalize on the beginning made here.

Finally, a number of appendices are provided to give implementation details not relevant to the general discussion of interaction spaces, but necessary to replicate or expand this work in the future.

Chapter 2 – Abstractions, Methodologies, and Tools for Robot Design

When discussing design, three broad categories of formalization are available. Design abstractions reduce the complexity of a system to a level comprehensible to a designer and allow the designer to communicate his or her ideas to others who are familiar with the abstraction. Design methodologies provide a process to assist the designer in the creative process and in accounting systematically for the steps necessary to create a functional system. Design tools provide quantitative insight into the functionality and behavior of a system and allow for reduced physical experimentation during the design process. Each of these formalizations is critical to design in the modern world of limited time and resources and global competition.

2.1 The Role of Abstraction in Design

The concept of abstraction is fundamental to design. The real world is infinitely complex, or at least so nearly infinite as to be effectively so for the purposes of current human capabilities. By contrast, an abstract model (mental or otherwise) of the real world is understandable, allows prediction, and often provides sufficient correlation with reality that conclusions derived from the model are effectively correct in the real world. The practice of design consists of the selection of an acceptable solution to a problem from among the many possible solutions. This can only be carried out via the development of a mental or physical model of the problem and solution.

A tool for understanding abstraction is the Rosen Model [4] shown in Figure 1. The Rosen Model, itself an abstraction, envisions the real or material world on the left side of an imaginary line. Within the real world, events occur due to causality, explained in other words as the normal flow of time and the laws of nature. On the right side of the same line, the abstract world exists. Within the abstract world, events “occur” based on execution of formal constructs; in other words, predictions are made according to the model that defines the abstract world. To move between the two worlds, an encoding or decoding process must take place. The encoding process is the mental process that takes place to transform the infinite complexity of the real world to the finite complexity of an abstraction; the decoding process is the application of the abstraction to infer real world results. The quality of the encoding and decoding processes represent the accuracy and precision with which predictions made in the abstract world will apply to the real world.

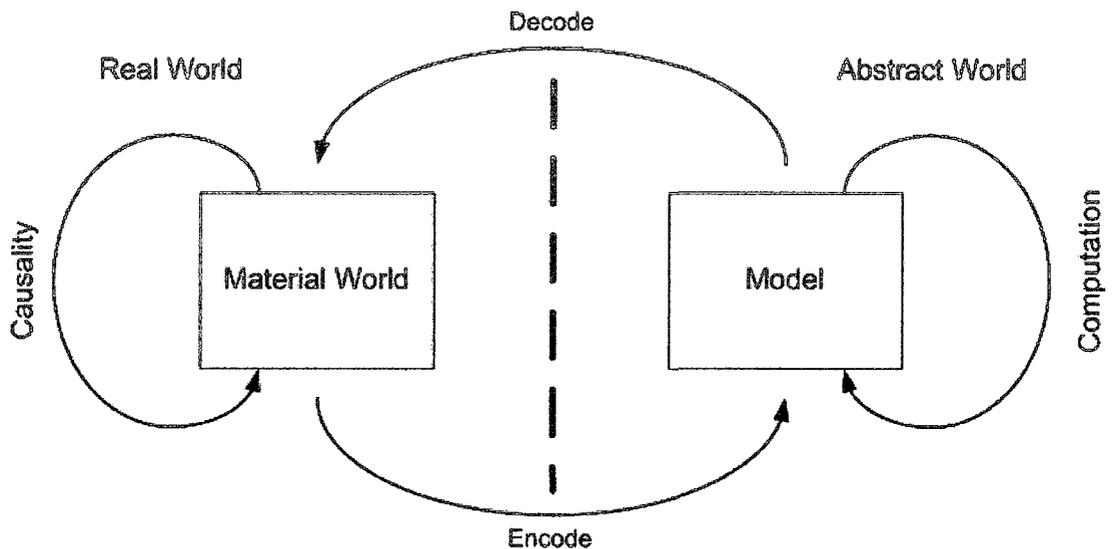


Figure 1 - Rosen Model

There are several abstractions that are currently used to develop models for autonomous systems. A brief overview of the most common, as well as those particularly relevant to the work in the remaining portion of this dissertation, is provided below.

2.1.1 Elements of Abstraction

In addition to abstraction, which is discussed above, it is also important to understand the distinction between abstraction and other related concepts such as formalisms, models, and realizations (see Figure 2). As shown in the Rosen Model, there is an encoding process from the real world to the abstract world. In some cases this encoding process can take the place of an informal removal of detail (here referred to as the process of abstraction) while in other cases a *formalism* exists that explicitly guides the move from the real world to the abstract world. In this case the process of abstraction is still being applied but it is guided by the formalism. An example of this would be the application of Newton's Second Law to abstract the motion of a projectile. Other less rigid formalisms are possible; the key aspect is that they represent a clearly communicable and documented encoding.

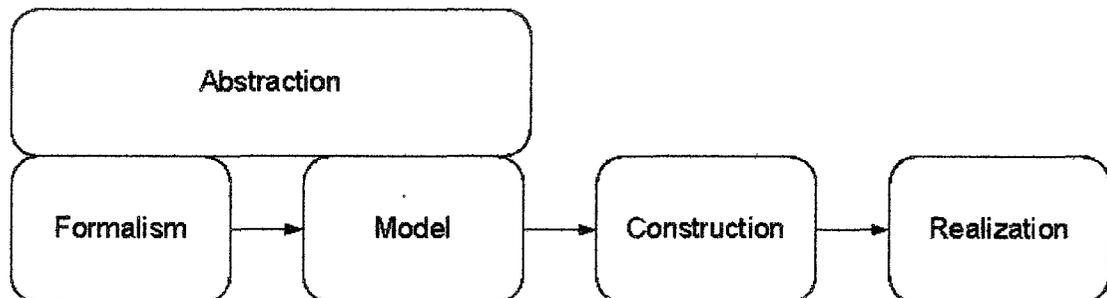


Figure 2 - Types of Abstraction

A *model* is what is created when one or more formalisms are applied to create an abstraction of a system. A model should be a representation of system behavior. Models can be both descriptive (*i.e.*, what does happen) or predictive (*i.e.*, what the system will do).

As shown in the Rosen Model, in order to move out of the abstract world and back into the real world, a decoding process is needed. In engineering, this decoding process is a multi-stage process as a system is designed and built. A physical system that has been built is a *realization* of the model.

2.1.2 System Dynamics

A formalism that will be used within this dissertation is system dynamics. System dynamics [5,6] is a feedback loop based technique for abstracting difficult-to-quantify situations, particularly in the business and economic world. In particular, system dynamics is used to model, understand, and communicate the complex interactions of related components of a system. System dynamics models contain the six basic elements shown in Figure 3. Stocks represent quantities and can most generally be thought of as real numbers. Flows represent a change in a stock. Auxiliaries are used to decompose complex logical or mathematical statements. Data arrows indicate connections between elements of a model, and denote the transmission of the value of one element to the other element. Constants are exactly that and do not change throughout the simulation. Sources and sinks can be thought of as stocks with value infinity.

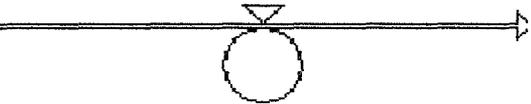
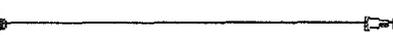
Symbol	Name
	Stock
	Source/ Sink
	Flow
	Auxiliary
	Constant
	Data Arrow

Figure 3 - Basic System Dynamic Elements

System dynamics as a formalism tends to focus on cause and effect relationships, and as such can be useful when considering the interaction space abstraction discussed below. In addition, most system dynamics texts, such as those referenced above, emphasize an iterative bottom up modeling approach.

2.2 Design Methodologies

Design is the application of an abstraction, usually through a formalism, to create a model. From this model, predictions about the efficacy of particular solutions are then evaluated in an attempt to determine an optimal solution. This description, while common, does not adequately address the issue of determining

possible solutions. Within the field of robotics, the determining possible solutions step is often addressed in a vague process of brainstorming or similar activities. While these activities are indisputably useful, they are more effectively applied as part of an overall design methodology such as the exploration based design process discussed below.

2.2.1 Top Down and Bottom Up Design

Most design methodologies can be broadly classified as either top down, or bottom up [1,7,8]. A bottom up design strategy involves getting the simplest possible element of a solution working and tested. Additional elements of the solution are then added incrementally with full testing and verification at each step. Thought is not given to the design of later increments while a particular piece is designed. By contrast, a top down strategy focuses on the simultaneous design of the entire system. In theory, all aspects of a solution would be completely known prior to construction of any element.

In practice, the top down, bottom up distinction is really a spectrum as shown in Figure 4. Bottom up design is used principally when a field is not well understood and when design tools and abstractions are poor, while top down design is more common in mature fields with well-understood abstractions, methodologies, and design tools.

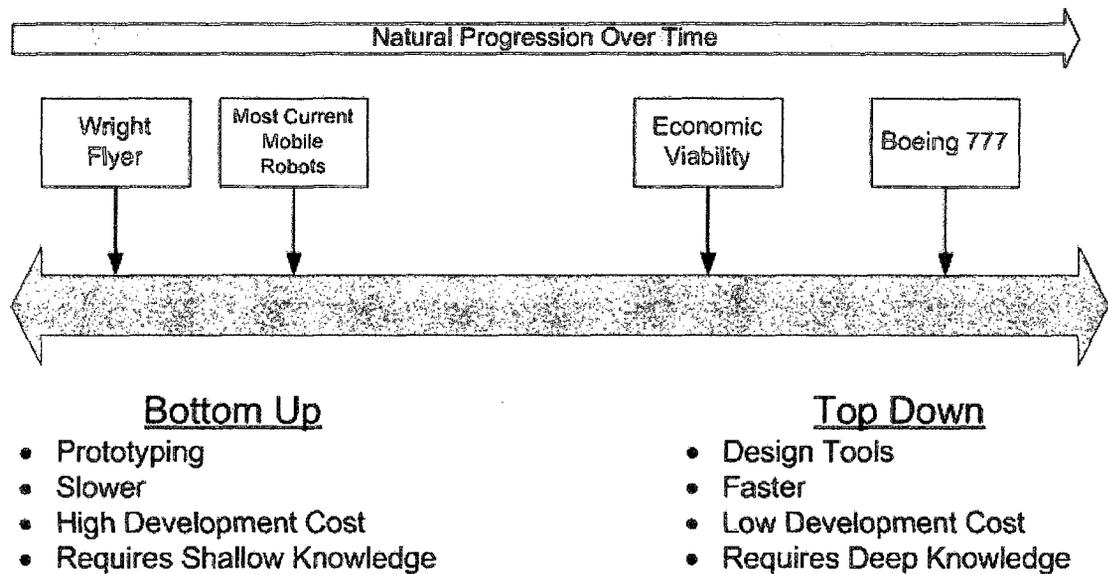


Figure 4 - Top Down and Bottom Up Design

2.2.3 Exploration Based Design

Exploration based design (EBD) [9,10] views design as a narrowing refinement of constraints. Initially, one starts out with all potential solutions to a problem (a solution space). Based on understanding of constraints and criteria, a designer is able to eliminate large portions of the solution space. Exploration through analysis, modeling, or prototyping of remaining segments of the space is used to further refine and quantify constraints and criteria in order to eliminate additional solution space regions. Gradually a designer narrows in on a single solution that best meets the constraints and criteria as they are understood.

Exploration Based Design begins with three elements: K_{dn} , K_{dm} , and R_i . K_{dm} is designer knowledge of the domain. This includes knowledge of how to perform a particular task; for example, that turning a doorknob and either pushing or pulling opens an unlocked door. K_{dn} represents knowledge of how to design in a particular field and can be broadly said to represent past experience of the designer plus any

formal methods that are to be used. R_i is a set of initial requirements, often in a qualitative form and rarely at a sufficient level of detail to begin choosing solutions. In an EBD process, illustrated in Figure 5, a designer would then use K_{dn} , K_{dm} and other properties to generate a better set of requirements. This would then eliminate a portion of the solution space, allowing a more detailed refinement loop to be subsequently implemented on the requirements. When applied in an iterative fashion this will, in theory, lead to a design that solves the problem at hand.

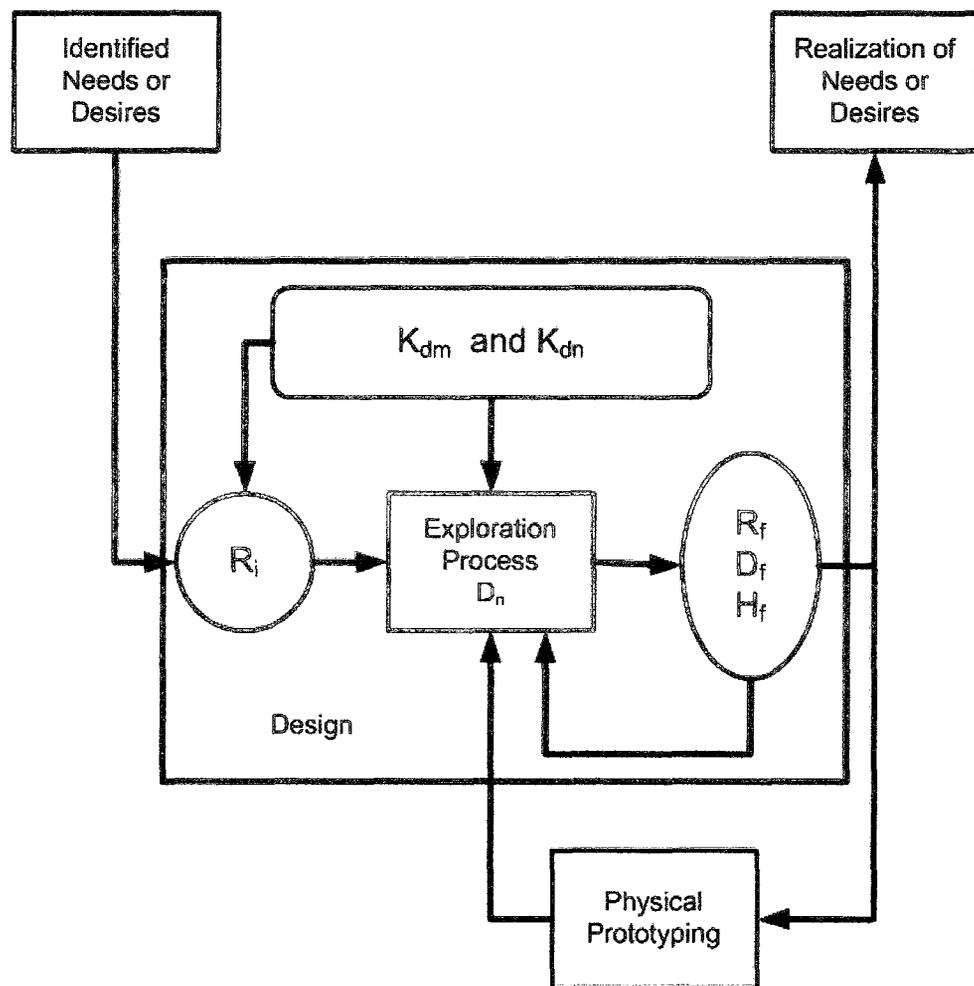


Figure 5 - Exploration Based Design

2.2.4 Behavior Based Design

The behavior based design abstraction [11,12] seeks to define a parallel set of behaviors (*i.e.*, tightly coupled reactions to environmental stimuli) that together result in emergent behavior that will lead to task accomplishment (an example from Brooks' seminal paper on the subject is shown in Figure 6.) Brooks, [1,7] the most visible practitioner of behavior based design, tends to advocate that the best way to accomplish this is through bottom up physical prototyping of successive layers of behavior. Indeed, in many cases, the concept of behavior based design is mentally linked directly to the concept of extensive physical prototyping and unpredictable emergent behaviors. This is undesirable due to the inherent cost in time, materials, and testing that is associated with design based purely on physical prototyping, particularly when used not for debugging, but for exploration, as is the case when emergent behavior is sought. This issue is discussed in further detail later in this chapter.

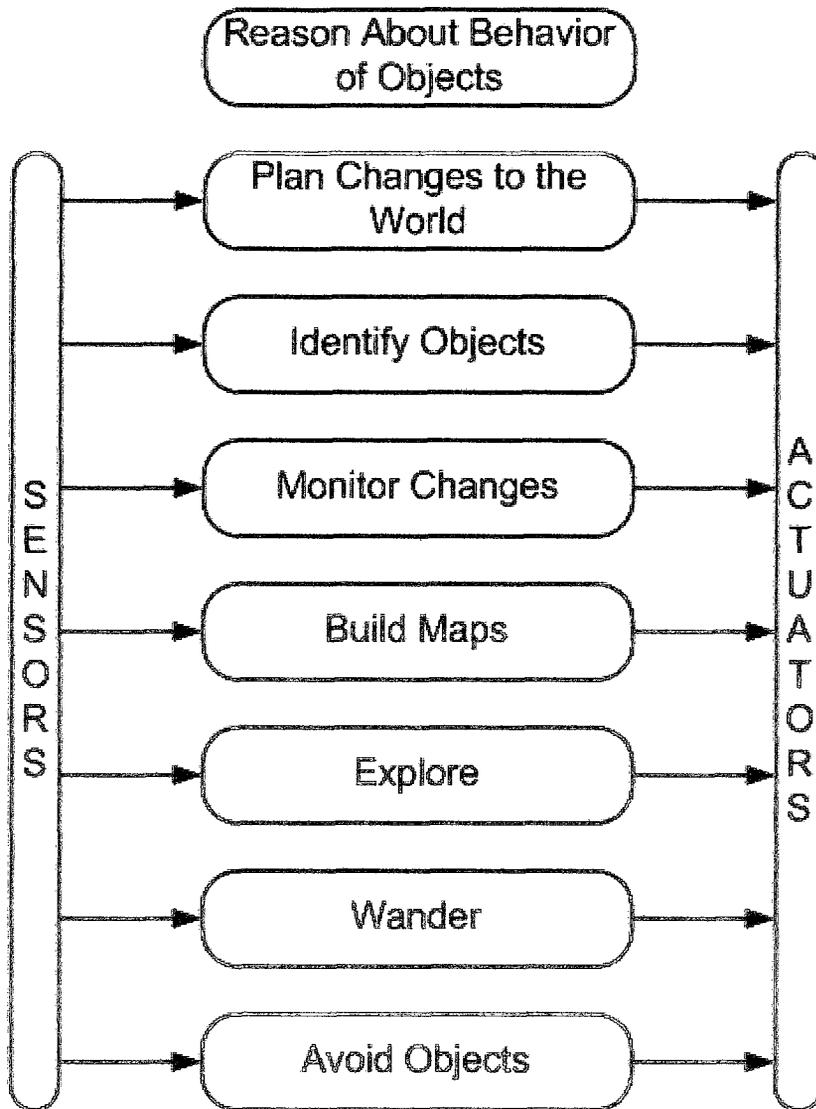


Figure 6 - Vertical Robot Decomposition [1]

Many implementations of behavior based design exist including subsumption [1], schema [13,14], and a host of others. In general each of these is useful in certain cases. It is left to the broader engineering community and the individual designer to make use of these as appropriate. In general, a design methodology should seek to allow use of as many of the tools that have been developed as possible.

2.2.5 Generalized Design Methodologies

Many generalized design methodologies exist. It is beyond the scope of this dissertation to provide a comprehensive description or explanation of these. Several comprehensive references are available in most technical libraries. Any of these generalized philosophies can be useful and relevant to robot design; however, in the parlance of exploration based design, most of these are predicated on very detailed and specific domain knowledge and extensive design knowledge within a narrow field. Given the present absence of this knowledge in many fields with potential robot application, these generalized methodologies are often insufficient for speedy and successful robot development.

2.3 Robotic Design Abstractions

Design abstractions in general, and robotic design abstractions in particular, can be classified according to both level of abstraction and the degree to which they are applied top down or bottom up. Design abstractions applied at a high level of abstraction are generally used for conceptual design and initial design definition while lower levels of abstractions become more applicable as the design process progresses. This is not a hard and fast line, but rather represents a progression.

Many robotic design abstractions have been proposed, many framed as architectures, and others specifically as design abstractions. A representative set of design abstractions is shown in Figure 7 and discussed below.

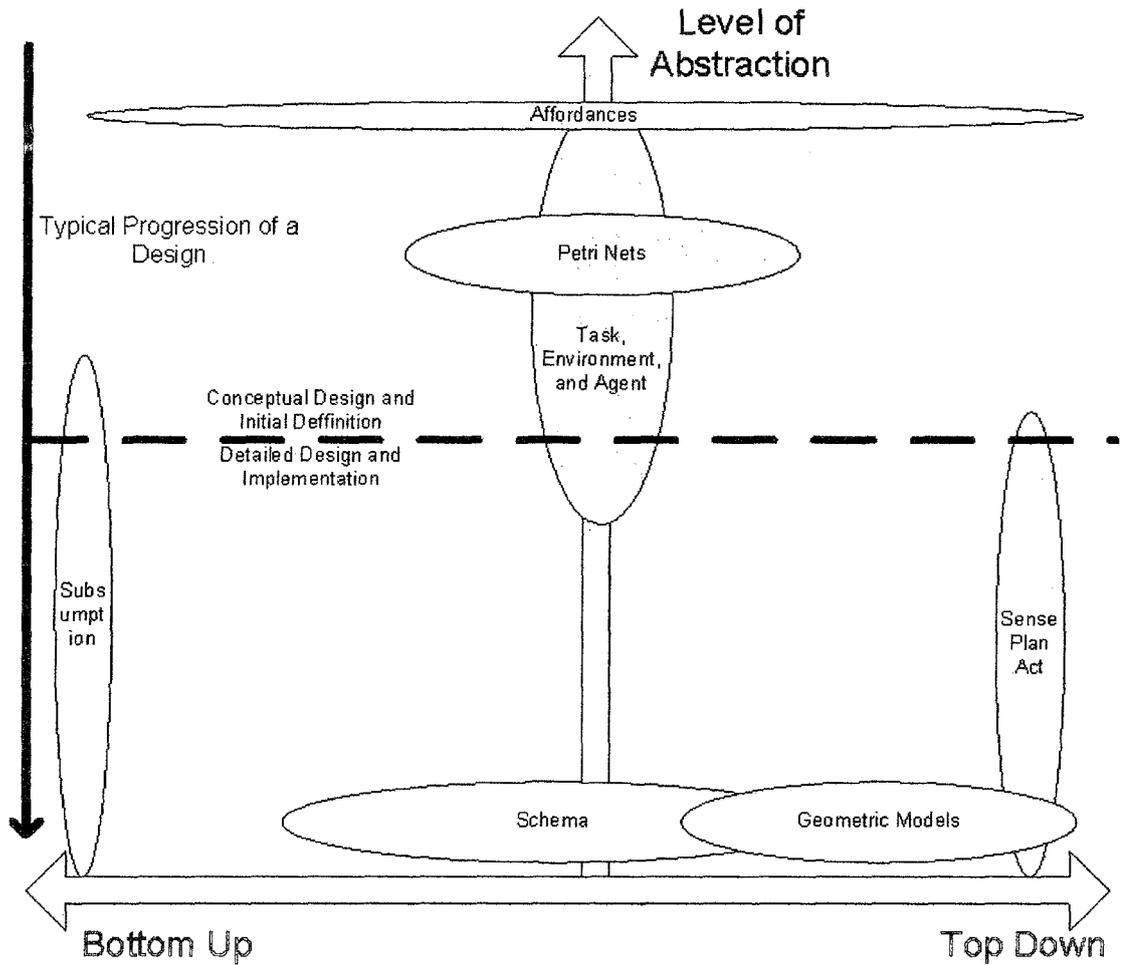


Figure 7 - Level of Abstraction and Type of Design Process for Various Robot Abstractions

Within the exploration based design model, design abstractions are typically a way of capturing domain knowledge about the system. To the extent that these abstractions have a formalized manner in which they are typically applied during design (for example, subsumption is typically applied in a particular bottom up fashion described by Brooks) they may also represent design knowledge.

For the purposes of the remainder of this dissertation, a robot will be considered any designed system that reacts to its environment and which seeks to accomplish a task.

2.3.1 Task, Environment, and Agent

In the Task, Environment, and Agent (TEA) abstraction (Figure 8), robotic systems are comprised of three parts: an agent (robot), an environment in which it is to operate, and a task or tasks that it is designed to achieve. Early robotic projects dealt predominantly with agent design using contrived tasks and contrived environments. Later work [15] views task, environment, and agent on equal terms, whereby each part of the system must be based in the real world (e.g., not contrived). Moreover each of the three elements must interact with the other two. The TEA abstraction explicitly points out the equal, if not greater, importance of the interactions between system elements as compared to descriptions of the elements themselves. As shown in Figure 7, the TEA model is generally a high level abstraction, most useful in the early stages of design. The TEA model is relatively neutral with respect to top down or bottom up design.

Unlike most of the other abstractions discussed in this chapter, the TEA abstraction does not lend itself to implementation and hence is not as clear cut a case of domain knowledge about how a system works or is constructed, nor does it provide any knowledge of how to design a system and cannot be considered design knowledge. However, the TEA abstraction is in fact a limited form of domain knowledge.

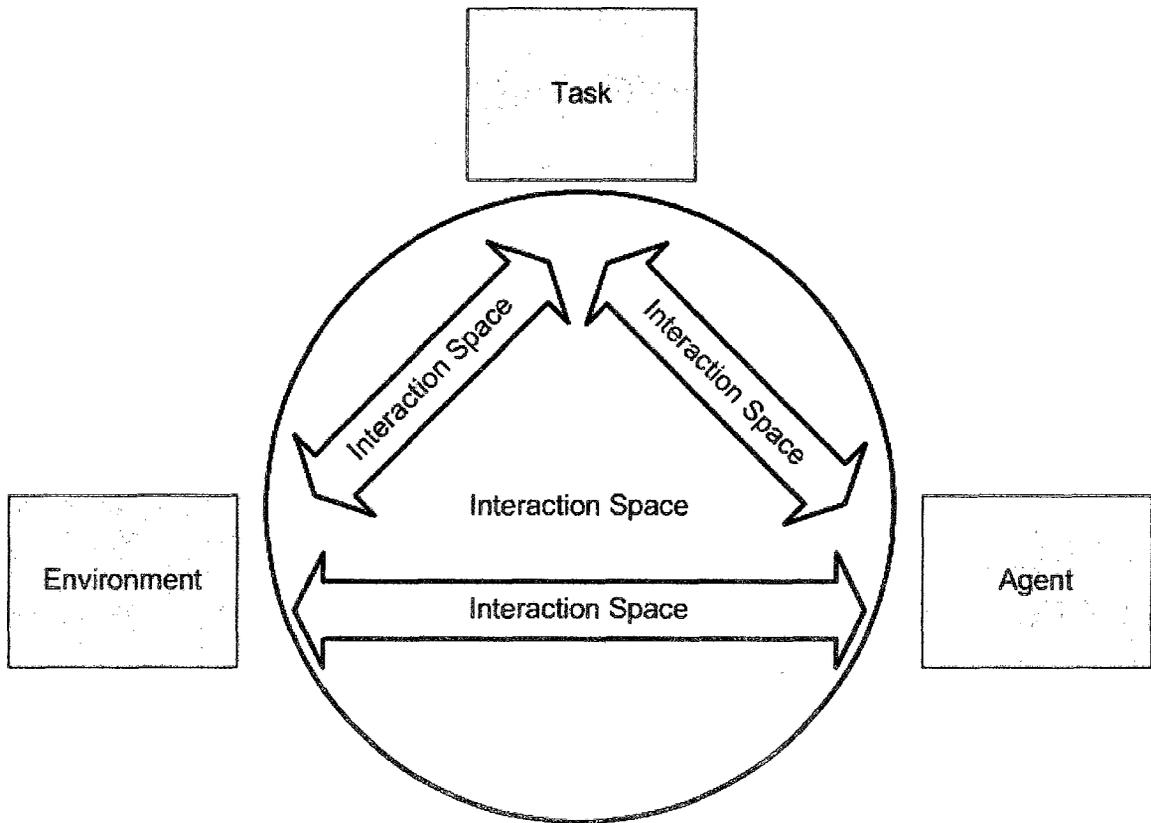


Figure 8 - Task Environment and Agent Abstraction with Interaction Spaces

2.3.1.1 – TEA Definitions

- **Task:** A measurable outcome of the interaction between agent and environment. A task must be “useful” in that it must contribute to an agent’s purpose.
- **Interaction:** A cause and effect exchange between an agent and an environment
- **Agent:** An independent device, consisting of one or more subsystems, that is designed to complete specified tasks

- Multi-agent system: A system consisting of more than one agent that is designed to carry out additional purposeful task(s) beyond the sum of the capabilities of the constituent agents.
- Environment: The entire “relevant” world, excluding the agent itself, but including other agents in a multi-agent system

2.3.2 Affordances

The theory of affordances was applied to robotics by Ford [16]. The theory of affordances postulates that an agent is able to achieve a task because certain invariants in the environment “afford” the robot the opportunity to accomplish that task. For example, a chair has invariants in that it is at approximately knee height, is able to support weight, and has a flat surface, thereby affording a person the ability to interact with the chair by sitting on it.

The theory of affordances is a step in the right direction, but has two notable limitations with respect to understanding the interaction of task, environment and agent within robot design. The first is the qualification problem [17]. The second related problem is that there is no quantification associated with this theory making it difficult to use for prediction of real system behavior. As shown in Figure 7, affordances are a very high level of abstraction and are typically used very early in the design process. Moreover, since there is no formal structured method to apply affordances, this abstraction is principally a way of capturing domain knowledge rather than design knowledge.

2.3.3 Petri Nets

Petri nets, first developed by Carl Petri [18] are one form of abstraction used to model mobile robot design. Within a Petri net, many states are defined, each of which may be either active or inactive. For each state a set of transitions is defined through which the state may either become active or inactive. An active state contains a token that must be passed to another state in order to activate that state. In some implementations, multiple tokens may be propagated from a single active state.

Current work on Petri net models of robots focus primarily on resource allocation (*i.e.*, memory, sensors, etc.) [19]. Limited work has been undertaken on creating automated software generation systems based on Petri net models [20-22], but only within significantly limited boundaries. To date the author is unaware of any work on physical robot design using Petri nets. Petri nets are limited (with respect to some types of robot modeling and design) primarily by the fact that they are limited to finite state systems. As depicted in Figure 7, Petri nets are implemented at a high degree of abstraction but are relatively neutral to top down or bottom up design. Petri nets are inherently only a method of capturing domain knowledge, but several of the examples referred to above have some degree of design process inherent in the implementation, and to this extent Petri nets have been used to capture design knowledge as well.

2.3.4 Geometric Representations

Geometric representations are those most commonly thought of in mobile robot design [23,24]. These representations are generally applied as simulations in which the agent and environment are explicitly modeled in a great deal of detail and

interactions are modeled only at the physics level if at all. Often the focus is on making the environment generate appropriate sensor data [25]. This form of representation works well for some things, but is generally too cumbersome for exploration based design processes except perhaps very late in the iterative process.

As shown in Figure 7, geometric abstractions are typically applied at a very low level of abstraction. Geometric models are almost always applied in a top down fashion as it is necessary have a reasonably complete representation of a system before useful predictions can be made from a geometric model. Geometric models do not in and of themselves capture any design knowledge, but rather are only a way of recording domain knowledge at a relatively low level of abstraction.

2.3.5 Sense-Plan-Act

The sense-plan-act process, shown in Figure 9, is one of the earliest abstractions for dealing with robotics and dates back at least to the days of STRIPS [26]. In general this abstraction can be described as the robot using its sensors to gather data about the environment, subsequently developing a plan that it is believed will result in the goal state, and then acting to a state closer to that of the goal state. This process is then repeated indefinitely until the goal state is achieved. Since the environment is dynamic, this plan must be regenerated either fully or in part during every implementation of this cycle. Given the complexity of the world and the rapid changes that are possible in most environments, computational complexity becomes a major issue in this paradigm, particularly within the real time limitation that robots necessarily operate under and the limitations of mobile computing that can be placed

on a robot. These planning systems are typically based on a geometric abstraction though others are possible.

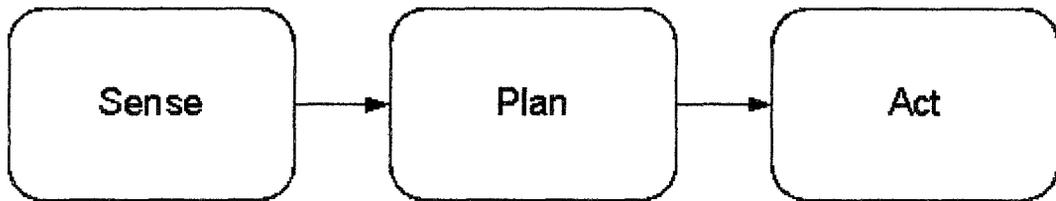


Figure 9 - Sense, Plan, Act Abstraction

Alternatively, a perceive stage is often added as shown in Figure 10. This cycle works the same way as the one above, with the exception that prior to planning, the agent attempts to classify the state of the environment around it. This is often used to select alternate planning systems in an attempt to reduce the complexity of any single planning system.

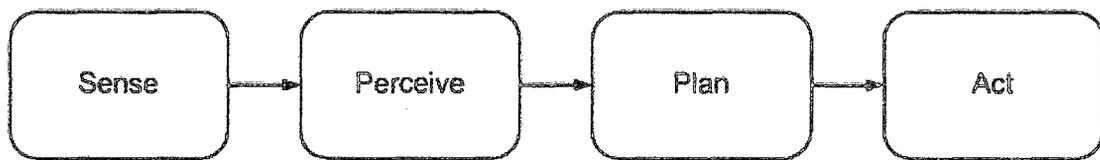


Figure 10 - Sense, Perceive, Plan, Act Abstraction

By its nature the sense, plan, act cycle is typically a top down design abstraction as most planning systems require significant detail to achieve a basic functionality. For this same reason, this abstraction is typically implemented at a low level of abstraction. Sense-Plan-Act falls firmly into the domain knowledge realm as it is both an abstraction and an implementation.

2.3.6 Schema

In a general sense the schema architecture is a functional mapping from environmental inputs to actuator outputs. Originally proposed by Ronald Arkin [13,14], there are a number of additional works that expand on this architecture. As with many of the abstractions presented here, this is a useful tool in implementing certain aspects of robot control, but does not inherently provide any systematic methodology for developing requirements or understanding the actions that will lead to task accomplishment. As with most of the combination implementation/abstractions, one is limited to a single technique for all problems.

Schema is one of the architectures that also serves as a design abstraction. As shown in Figure 7, the schema abstraction is generally a very applied abstraction and is generally applied as both a design abstraction and an implementation. The schema abstraction was originally presented as a bottom up architecture, but has generally been applied as both bottom up and top down. As with other implementation/abstractions, the schema abstraction represents primarily domain knowledge.

2.4 Design Tools

As opposed to qualitative models, design tools are generally used to provide quantitative predictions concerning the behavior of a system. Many design tools exist in other engineering domains from the general (*e.g.*, structural or thermal finite element analysis) to the very specific (*e.g.*, bridge design software or auto routing systems for PCB layout).

Design tools typically capture some degree of design knowledge and often automate some or all of the process of applying this knowledge. For example, a solid modeling finite element analysis package can capture and display domain knowledge regarding geometry, forces, stresses, deflections, and other such factors. In addition, many of the more sophisticated packages are also capable of formal optimization of geometry based on constraints or other such automated design processes. For the purposes of this dissertation, a design tool will be considered any application of a model that yields useful quantitative domain predictions regardless of the degree to which this process is automated.

2.4.1 Geometric Simulations

The majority of robotic design tools are geometric simulation engines. These range from proprietary simulations developed for research purposes to commercial products such as Robot Studio [27]. The sophistication and complexity of these models ranges from relatively simple to highly complex dynamics engines similar to those used in video games [28]

While geometric simulation certainly has a role to play in well-defined situations or in determining the physical ability of a particular system to accomplish a specified task, geometric simulation requires substantial understanding of the task and environment, and significant definition of the agent. As such it is poorly suited for use early in the design process when significant design freedom still exists. Moreover, the process of geometric simulation either requires a previously developed dynamics and physics engine, with attendant assumptions that are not apparent to the designer,

or a significant investment of time to develop these features for the particular application at hand.

Chapter 3 – Adding Structure to Interaction Spaces and Defining Notation for Interaction Space Models

Within the exploration based design process, abstractions and more specifically models are often useful as a part of the exploration process. To fully explore a design space, it is necessary to have a formal means of capturing thoughts assumptions (*i.e.*, domain knowledge) about the system under consideration. This chapter will introduce the concept of interaction spaces and discuss how this interaction fits into the task, environment, and agent model as well as the exploration based design methodology. A number of tools and notational devices will be introduced to help the reader follow proceeding chapters. Interaction space models and modeling will not be introduced until Chapter 4.

3.1 Interaction Spaces

As shown in Chapter 2, a traditional view of the task, environment, and agent abstraction has each of the three corners of the triangle on equal footing. The interaction space abstraction tends to view the agent and environment in continuous interaction. The set of all of these interactions is the “interaction space.” If these interactions are the “proper” interactions, the task will be accomplished. This is shown in Figure 11.

Within the interaction space abstraction, the goal of a designer is to create an agent that will interact with the environment in such a way as to accomplish the task(s) at hand. A key element of carrying this out involves correctly understanding the interactions between the agent and environment.

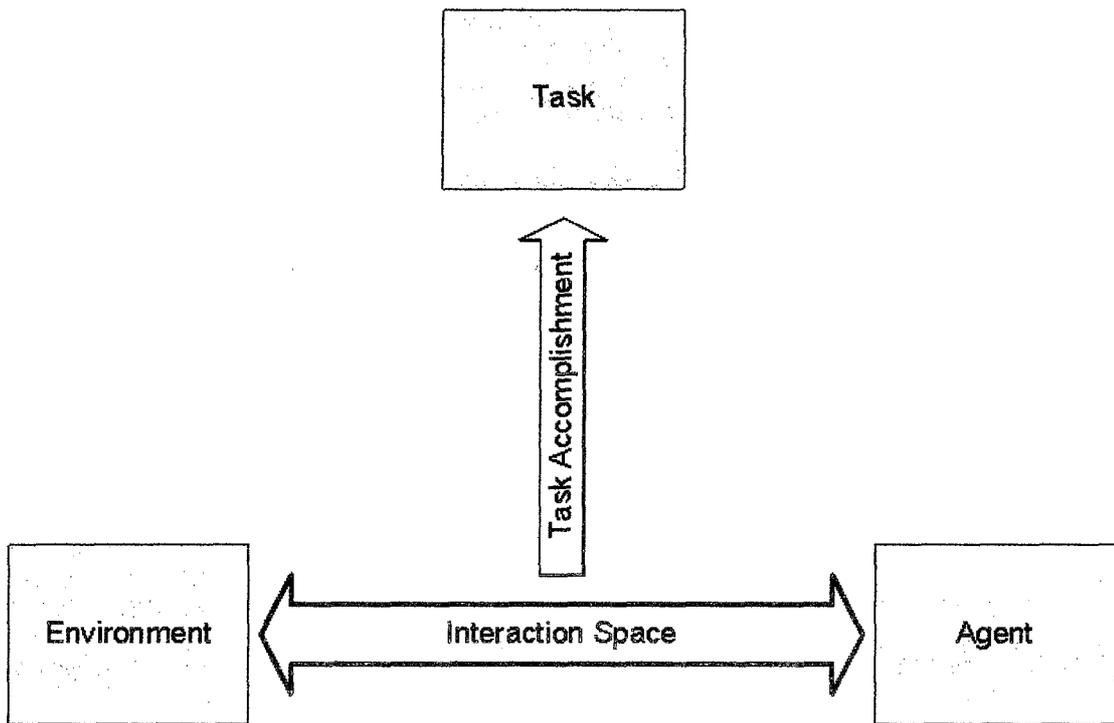


Figure 11 - Task, Environment, and Agent Abstraction with Interaction Spaces

Interaction spaces and interaction space modeling were originally introduced in previous work of the author. Additional information on interaction spaces is available in [2,3]. As implemented previously, interaction spaces as an abstraction have been limited by a lack of formal structure. Relatively few suggestions were made for developing system models for either the agent or the environment, and only a small and far from spanning set of standardized blocks were available. Because of these limitations, similar to many of the other abstractions discussed, there was no design methodology to assist the designer in developing his or her thoughts; the emphasis was on clever modeling. As a consequence, early interaction space models took several dozens of iterations and a significant amount of time to develop even a simple model with limited complexity.

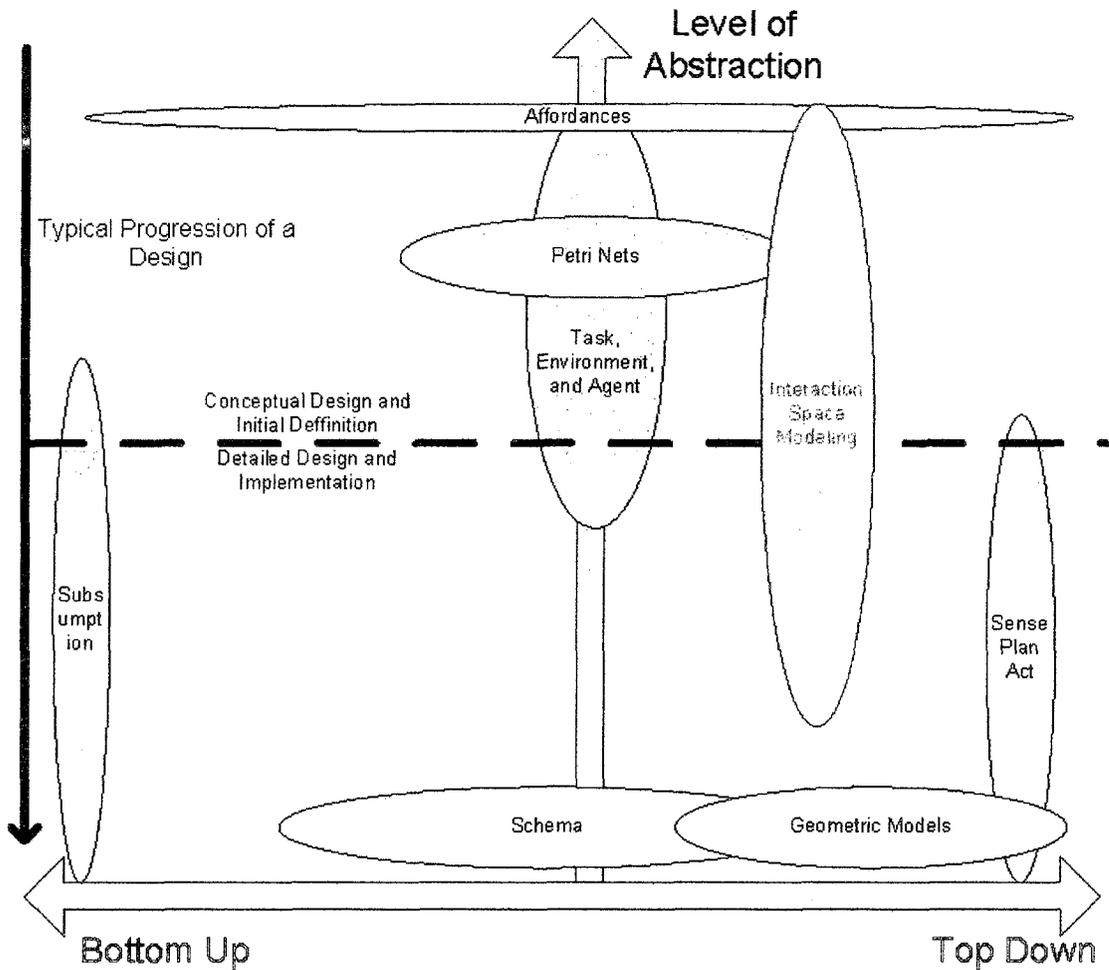


Figure 12 - Level of Abstraction and Top Down vs. Bottom Up Characteristics of the Interaction Space Abstraction Relative to Other Robot Design Abstractions

As shown in Figure 12, the interaction space abstraction can be implemented at widely varying levels of abstraction. The interaction space abstraction is intended primarily for use in the early stages of design and is not necessarily well suited to final detailed design. The interaction space abstraction is intended to help enable top down design although the models are built in a bottom up fashion.

Although interaction space models can be created in an *ad-hoc* fashion [2] to capture domain knowledge only, this dissertation will introduce interaction spaces in a different manner that incorporates significant design knowledge into the abstraction

by providing a clear process and methodology to develop models and thus explore the design space. It should be noted that this limited addition of design knowledge, while helpful to the designer, does not remove the responsibility for substantial design knowledge on the part of the designer.

While interaction space modeling does not make the designer any more intelligent or any more knowledgeable, it does provide a means of creative exploration of the design space. By forcing explicit examination of the interactions, it also forces the designer to think about different ways that the agent can interact with the environment.

3.2 Reactive Agent/ Reactive Environment

The interaction space abstraction defines the agent and environment to be equally influential in the design of a mobile robot. Interactions are explicitly shown both from the environment to the agent and from the agent to the environment. The quadrant abstraction shown in Figure 13 is an abstraction that can be used with the concept of interaction spaces to make this paradigm more explicit. Separating the state from the reaction for both the agent and the environment will become useful later in creating a systematic modeling methodology.

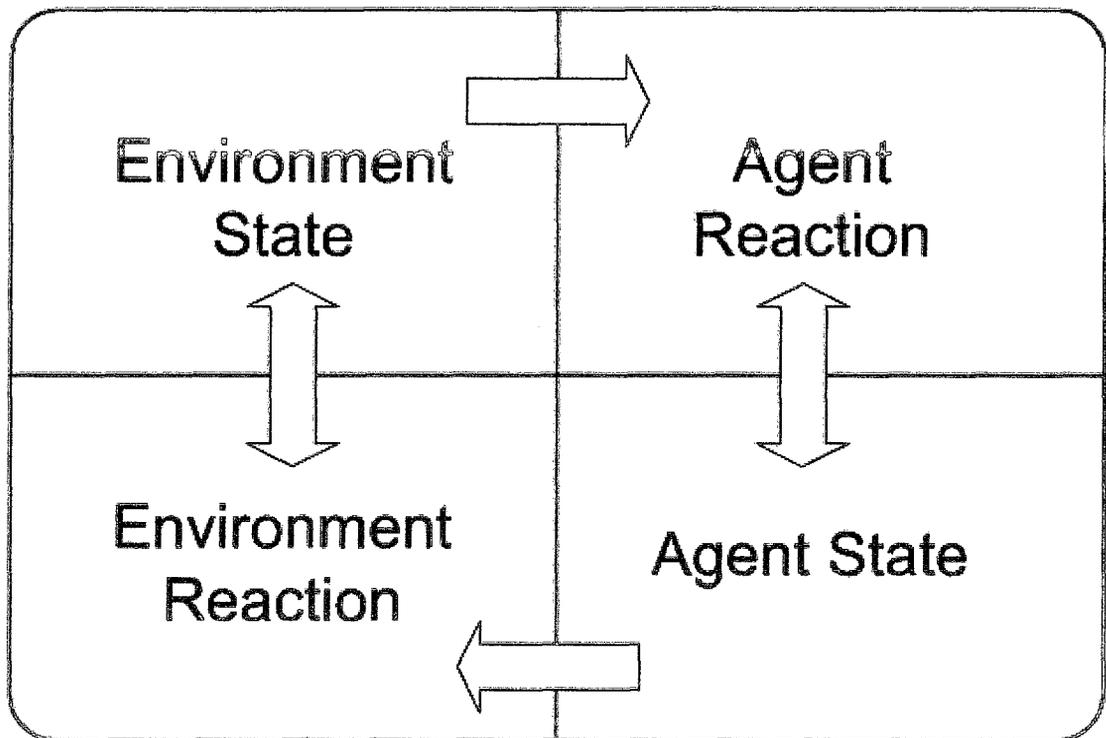


Figure 13 - Quadrant abstraction of Interaction Spaces

Starting in the upper left quadrant, the environment state stores information about the environment and, in cases where the state of the environment will change due to anything other than actions of the agent, determines what changes are needed. This could, for example, include the beginning of a wall or presence of a randomly distributed object within the environment. Action is considered a state in the same sense that position, velocity, and acceleration can all be considered states.

Moving to the right across the quadrant abstraction, the agent reaction is the response of the agent to the state of the environment. This includes the entire process from sensing to selecting a behavior via whatever control architecture the designer has selected. As indicated in Figure 13, the agent may also react to the agent state in the quadrant below, but only under certain circumstances that will be discussed in the next two sections.

Similar to the environment state, the agent state stores information about the agent and manages changes to the agent state based on agent reactions or stochastic elements.

The environment reaction box manages changes to the environment state as a result of agent actions. Two types of signals can be sent into the environment reaction section. The first is actual actions from the robot that are used to affect the environment state directly. The second are agent “perceptions”, such as whether or not the agent believes a victim to be present. The second type is used primarily to record task accomplishment when developing predictive models and really represents the passing of information from the agent to a user (who is, from the point of view of the agent, a part of the environment.) This will be discussed in more detail in the chapter on predictive modeling.

3.3 Defining and Using Interactions and Cycles

The first step in creating an interaction space model is to define basic cycles within the quadrant abstraction described above. As shown in Figure 14, there are

three basic ways in which cycles exist within the quadrant abstraction.

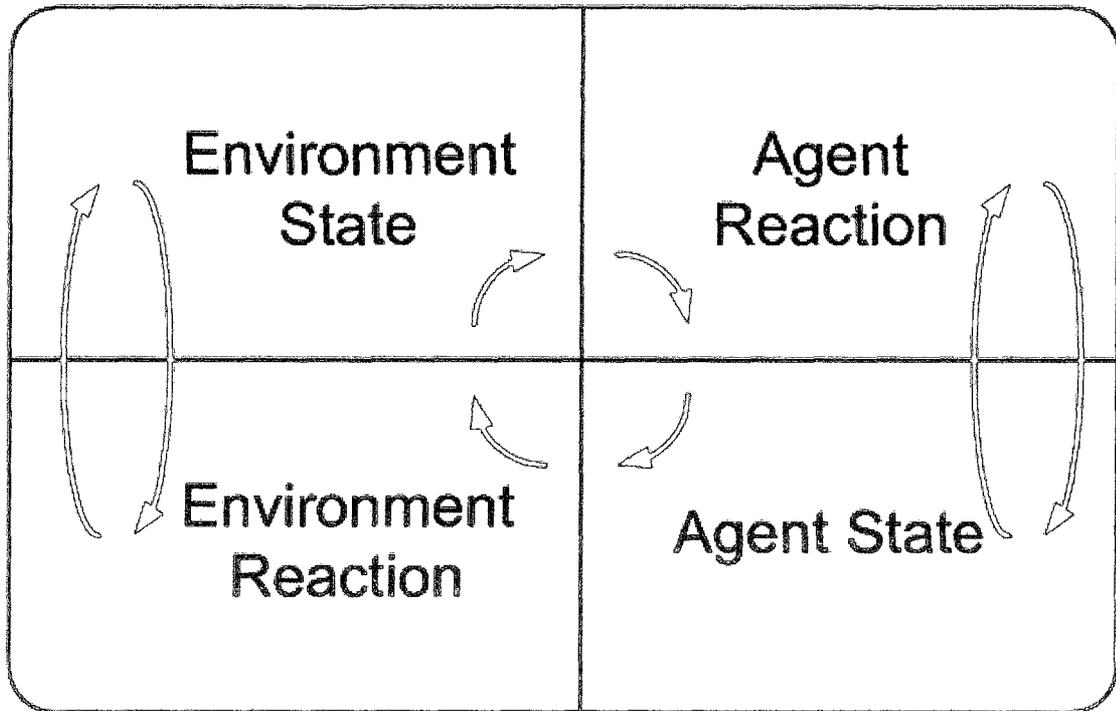


Figure 14 - Interaction Space Cycles

The most common cycle is that where information proceeds clockwise around the quadrant abstraction. In general, this represents actual interaction between the agent and the environment, and between the environment and the agent. The cycles are most easily constructed by identifying a particular task that the agent must complete. In most cases, this means that a specific environmental state must be achieved. Cycles are most easily constructed in the opposite direction of implementation. Starting in the bottom left quadrant at environmental reaction, one works counterclockwise to determine what agent state (usually actions) must be present to evoke this reaction; continuing counterclockwise, one determines what behavior or reaction of the agent would trigger this state, which then defines what

information the agent must “perceive” from the environment state, which must then in turn be updated to reflect the environment reaction.

Individual cycles should be as simple as possible at the early stages of developing a model. It is not uncommon to see only four functional blocks representing a cycle early in the process. As more cycles are added, more interaction will be required between cycles and it will be desirable to represent earlier cycles at a lesser level of abstraction, requiring that additional functional blocks, typically of more complex data types, be added.

The other two types of cycles shown in Figure 14 represent internal reactions within the agent or environment. For example, if power and consequently operation time of an agent is to be modeled, then the remaining power is a property that should be recorded in the agent state quadrant. There is not an interaction with the environment per se that causes the agent to cease to function, but rather the reactions to environmental stimuli are directly affected by the fact that the agent no longer has sufficient power. Similar situations exist in environmental modeling.

3.4 Multi-Agent Systems

The quadrant framework described above can also be applied to multi-agent systems as shown in Figure 15. In this case, the agent state for one agent acts as a part of the environment for other agents who can then react to it. Similarly, to each of the other agents, the first agent represents a part of the environment to which they can react. Thus cycles can be defined both between agents and between the rest of the environment and each agent.

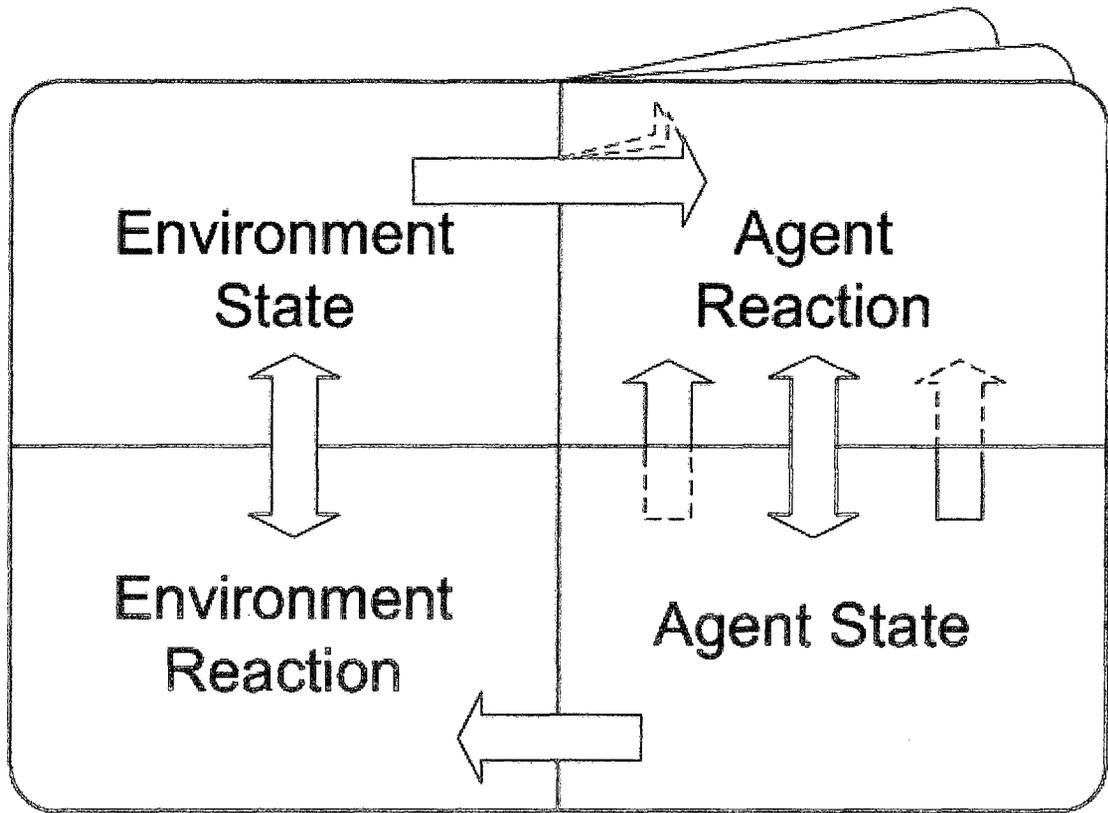


Figure 15 - Multi-Agent Quadrant abstraction

In practice, it usually makes sense to make modular subsystems out of the agent. In this case, there are defined inputs from the environment state and defined outputs to the environment reaction. In many cases, it is also useful to combine portions of the environment into modular blocks. In general, those blocks that are global in scope (*i.e.*, have the same value with respect to all agents) should be left independent, while those blocks that are local in scope (*i.e.*, that are different with respect to each agent) can usually be modularized. Specific examples of this are given in Chapter 4 in the Multi-Agent Foraging Model.

3.5 Understanding the Role of Information and Data Types

Implicit in the cycles of Figure 14 is the transfer of some type of information both in the real world and in the abstract world. Within the context of this dissertation, abstracted information will be represented in one of three ways. In the same way that units play a vital role in the correct interpretation of engineering calculations, data type management is critical to correct interaction space modeling and in fact provides a qualitative measurement of the fidelity and level of abstraction of the model. Once a cycle has been defined using information, it is essential to decide how that information will be represented. Inputs and outputs that are connected must operate on the same data type and format. In particular, for outputs from the state blocks, the degree to which the information is represented realistically largely defines the level of abstraction of the model.

While essentially any data type is feasible within interaction space modeling, the three discussed below are sufficient to create both descriptive and predictive models and are recommended as a starting point.

3.5.1 Binary Data

As the name implies, the binary data type corresponds to either one or zero. This can also be thought of as true or false, on or off, or any other two-state decision. Binary data types are the simplest to use and allow the tools of digital logic to be used. Binary data types should be used whenever possible.

3.5.2 Discrete Data

This choice is really an extension of the binary data type. This data type allows any of a finite number of states to be expressed. In implementation this is often a positive integer to distinguish between states. This data type is identical to the Unsigned INT data type in many programming languages. An example of this is the USAR model representing no movement, a small movement, or a large movement. Typically this data type would not be exchanged between the agent and the environment, but rather within the agent or the environment.

3.5.3 Continuous Data

Of the three discussed here, continuous data bears the most relation to the real world. This data type represents any analog quantity. In practice, this data type is most commonly implemented as a double precision float for predictive modeling.

3.6 Basic Functional Block

The cycles described in Figure 14 are actually modeled within the abstract space using functional blocks. Each functional block represents a stage in either generating or describing the state and reaction of both the agent and the environment. At present, there are six main types of functional blocks, which are described below. In addition, there are several other types of functional blocks that are used for the stochastic elements of the model. Throughout the remainder of this dissertation, each functional block is represented by the symbol shown in Figure 16. More information about inputs, outputs, control of the block, and implementation are provided later in this dissertation.

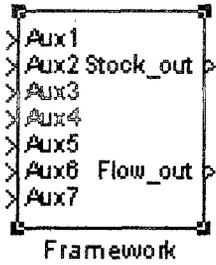


Figure 16 - Basic Functional Block

The basic functional block is used for representing both state and reaction as described above in the quadrant abstraction. Within each of these two categories, each of the data types is also represented with a particular name and implementation of the basic functional block. This yields a total of six types of basic functional blocks, which are described below.

3.6.1 State Blocks

There are three basic state blocks, which correspond to the three basic data types discussed in above. In general, a state is modeled by first defining features, then defining attributes of those features, and finally assigning properties to the attributes. However, as will be discussed in Chapter Four, this may not always be the most judicious arrangement of these blocks, particularly early in the modeling process.

3.6.1.1– Features

Here features are defined to be objects or portions of objects that are present in the environment. The feature aspect is defined to be only the presence or absence of the object; all details of the feature are defined through other types of elements.

3.6.1.2 – Attributes

An attribute is a specific aspect of a feature that gives more detail. For example, if a wall is present, an attribute of the wall may be color. However, an attribute has a finite number of states, thus “blue” would be an attribute but 780.5nm would be a property as discussed below. Attribute blocks can be implemented as stand-alone when the corresponding feature is always present and need not be modeled explicitly.

3.6.1.3 – Properties

A property is a measurable quantity associated with a feature or attribute. The quantity is always continuous (or continuous within the bounds of the numerical precision of the computational tool used) and should really be thought of as an analog signal. In this way, a property is intended to represent the real world with the highest degree of fidelity of the state blocks represented here. Similar to an attribute, a property may exist as a stand-alone entity when the associated features and attributes are known to be constant.

3.6.2 Reaction Blocks

Similar to state blocks, there are three basic reaction blocks. Once again these correspond to the basic data types discussed above. In general, most reaction models will move from signals to information to behaviors. However, as will be discussed below, there are times, particularly early in the modeling process, when other arrangements might be desirable.

3.6.2.1 – Signals

Signals are the reaction-side equivalent of properties. The signals are what is taken directly from the state. As such, the signal block can really be thought of as a sensor block; however, it is possible to have a signal block in the absence of what is traditionally thought of as a sensor (*i.e.*, in a mechanical orientation feature of an injection molded part). This is particularly true for modeling environmental reactions where there will rarely be an explicit concept of “sensor” as it is traditionally understood in the field of robotics.

3.6.2.2 – Information

An information block is intended to produce a processed finite state representation of the agent or environment’s “perception” of state. This need not be “perceived” in the classical artificial intelligence sense, but rather represents a choice from among a finite number of options; for example, whether a particular water molecule will go left or right at a Y-junction in a pipe.

3.6.2.3 – Behaviors

Behaviors are either explicitly active or inactive (*i.e.*, they have a binary data type) and are used to “decide” upon specific actions either by the agent or the environment. Examples could include an environmental reaction when a victim is found or an agent’s reaction when it believes that a victim is present. These blocks are predicated on the use of a behavior based or hybrid robot control architecture. Implementation of other control architectures may preclude the use of behavior blocks in the agent reaction.

3.6.3 System Dynamic Implementation

Regardless of specific type, the basic functional block represents a single stock and an associated flow. The flow has explicit feedback from the stock, which among other functions, is frequently used to reset the stock to zero (see below). Unlike some system dynamic implementations, this flow may be positive or negative. This could be more explicitly represented as two flows (one incoming, one outgoing); however, this is functionally simpler and mathematically equivalent.

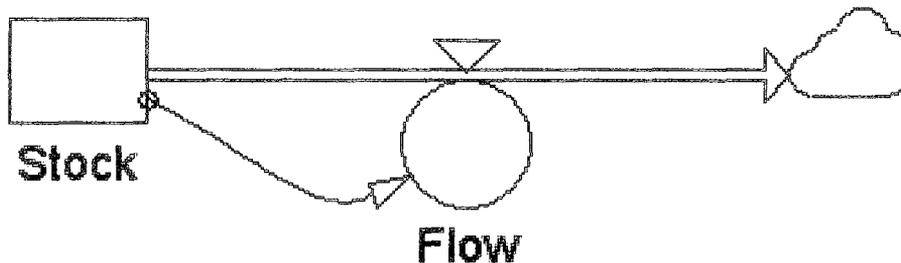


Figure 17 - System Dynamic Implementation of a Basic Functional Block

Functionally, the “Aux1” through “Aux7” inputs (shown in Figure 16) feed into the flow, and are used during each time step in calculating the flow. The “Stock_Out” function provides the value of the stock at each time step, and the “Flow_Out” provides the value of the flow at each time step. Typically only the “Stock_Out” output is used (particularly within the modeling framework), but the “Flow_Out” can be useful in cases where one flow is directly dependent on another. Additional information on the implementation of the functional block is available in Appendix A.2 Simulink Basic Functional Block Implementation.

3.6.4 Controlling Functional Blocks

Functional blocks are controlled via both an initial value and quasi-continuous control of the flow. Many strategies can be devised for control of the flow. In general, a rule-based approach has been used with significant success for modeling the situations encountered so far. As will be discussed later, when used for predictive purposes, as discussed in Chapter 5, the value of the flow is determined by an m-file allowing for the use of a wide variety of techniques, and implementation of most control architectures.

In general, a functional block will be used in one of two ways, either instantaneously or continuously. In an instantaneous block, the value of the flow is always set to a new desired value minus the current value of the stock as shown in Equation 1. In a continuous block, the previous value of the stock is retained and is only modified by the appropriate flow value.

$$\text{CURRENT VALUE} = \text{CALCULATED VALUE} - \text{PREVIOUS VALUE}$$

Equation 1

3.7 Stochastic Blocks

In addition to the six variants of the basic functional block, several other types of blocks are useful in adding uncertainty and variation into models. Both uniform and discrete random number generators are discussed below as well as a random decision-making block. In addition, various types of noise blocks are discussed. These blocks are essentially functions that would be included within a flow in a traditional system dynamic implementation. For the purposes of interaction space

modeling, the function is made explicit to help in understanding, but these blocks are fed into the inputs of the basic agent block described above such that they still control the rate of the flow.

3.7.1 Random Number Generator Blocks

The uniform random number generator block is a relatively straightforward random number generator that returns a random real number between two specified values. These blocks can be widely used, including in the generation of noise as described below or as a way to generate continuous portions of the agent or environment state. The representation shown in Figure 18 will be used throughout the remainder of this dissertation to depict this type of block. Details of the implementation of this block are given in Appendix A.3 – Simulink Uniform Random Number Generator Implementation

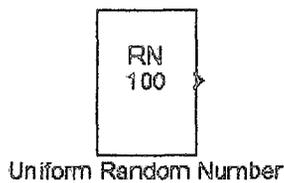


Figure 18 - Uniform Random Number Block

The discrete uniform random number block returns an integer between zero and the number of states. This block is used predominantly in discrete state models to generate attributes for the environment or the agent. The representation shown in Figure 19 will be used to depict this type of block in the remainder of this dissertation. Details of the implementation of this block are given in Appendix A.5 – Discrete Random Number Generation Block.

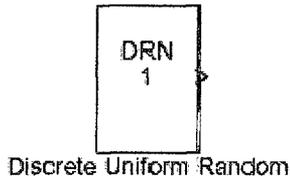


Figure 19 - Discrete Uniform Random Number Block

3.7.2 Random Decision Maker Block

The random decision maker block (shown in Figure 20) generates a “1” at approximately the percentage of time steps specified. For the remainder of the time steps this block generates a zero. This block is used primarily for stochastically determining features for the environment and agent states. The details of the implementation of this block are given in Appendix A.4 – Random Decision Making Block Implementation.

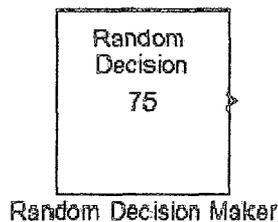


Figure 20 - Random Decision Making Block

3.7.3 Noise Blocks

Noise or uncertainty can be added to a model in many locations and is represented by many types of functions. Noise can be used to represent various types of error and uncertainty, from the uncertainty associated with a gear train, to the uncertainty associated with a sensor, an A/D converter, or other electronic device. In addition to uncertainty, noise blocks can also be used to represent outright errors. For

example, when modeling communication between two agents, a noise function could be derived to represent communication errors, perhaps due to another transmission on the same frequency, solar radiation, or some other source.

A number of noise functions are built into Simulink. In general, these have proven sufficient for most models; however, the possibilities are nearly limitless in designing noise functions to match the real world as closely as possible. There are several useful books on this topic for sensors [29], electrical and mechanical systems [30], and numerous others.

3.8 Tasks

Tasks can be broken into two categories: perceptive tasks and physical tasks. Perceptive tasks are those where the agent is asked to make a judgment about the environment, while a physical task is one where the agent is asked to manipulate the environment or its relationship to the environment in some way. Often accomplishment of one task is dependent on a number of others, which may or may not be of the same type. Under these circumstances it is generally sufficient to the evaluation of the final system to measure only the final task in the appropriate manner; however, in creating a useful design model, one should carefully observe and measure accomplishment of individual sub-tasks.

3.8.1 Task Accomplishment for Perceptive Tasks

Perceptive task accomplishment is measured by comparing the agent state (*i.e.*, what the agent “believes” to be true about the world) to the environment state (*i.e.*, what is actually true within the abstract world). In descriptive models, task

accomplishment is difficult to measure, but the conditions of accomplishment should be clearly stated. In this case, the quantities that should match between the agent state and the environment state should be defined and the degree to which they should be similar should be explicitly recorded as part of the requirements R_n . Additional information on quantitative measurement of tasks is given in Chapter 5.

3.8.2 Measuring Task Accomplishment in Physical Tasks

Physical task accomplishment is modeled and/or measured by comparing the environment state to some desired state. This ranges from simple to complex (for example, when it is difficult to define the desired physical state within a finite number of variables or when domain knowledge is insufficient to fully define the desired state of the environment). As with perceptive tasks, the quantities that must match and the degree to which they must match should be defined in the descriptive model and recorded as part of the requirements of the system.

Chapter 4 – Descriptive Models

This chapter takes the nomenclature and concepts of Chapter Three as well as the concept of exploration based design to develop a design methodology and modeling process that incorporates more design knowledge and narrows the field of options that must be considered in creating a model without such guidance. Examples of robot models are given as illustration of the process.

4.1 Modeling Elements

As was touched on in Chapter three, there are two main elements in representing an interaction space model: the blocks (with associated connections) and the functionality or definition of the blocks. Each of these is described below.

4.1.1 Combining Functional Blocks

The cycle shown in Figure 21 is a template abstraction by which all interaction space cycles can be represented. These standard cycles are combined and elaborated as described below to create interaction space models. Combination and elaboration of this basic cycle are demonstrated by example in the muramador robot models.

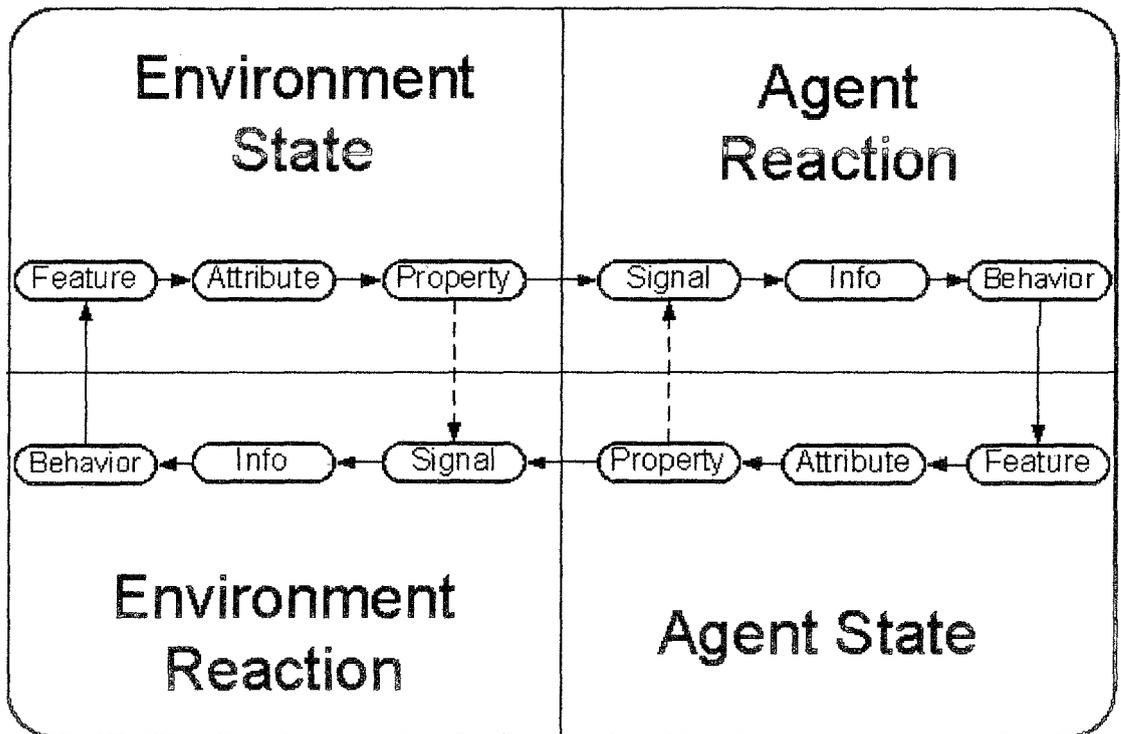


Figure 21 - Standard Cycle Abstraction

4.1.2 Developing Pseudo Code and Meanings

In addition to the blocks of an interaction space model, it is also necessary to define meaning for the blocks. In the descriptive modeling phase, this is done through the use of pseudo code and a basic explanation of the meaning of the block. As discussed in the previous chapter, each block has inputs and an output. The inputs affect the flow in the system dynamics sense while the output is the value of the stock. However, for simplicity the standard convention for a descriptive model shall be to write the pseudo code as if the inputs directly affected the stocks. Each block at every stage of the modeling process should have a pseudo code segment that defines the output as a function of the input(s). The specific coding necessary for predictive modeling will be discussed in Chapter 5.

In addition to the explicit pseudo code it is generally helpful to have a plain language description of the intended meaning of the block. Throughout the Muramador model below each stage of the modeling process will have a table with each block listed as well as the pseudo code and a physical interpretation. These aspects are just as important as the blocks themselves in developing a descriptive mode. For the sake of brevity these tables are not given for the other models, but executable code for each of the blocks can be found in the appendices.

4.2 Steps in the Modeling Process

In applying interaction space modeling to exploration based design, one first needs to create an initial model. There are seven processes that are generally used to create the initial interaction space model. In general they form a progression as shown in Figure 22. While the progression shown represents one methodology to creating interaction space models, others are possible. It is left to the individual designer to make a determination as to the optimal order if different from below.

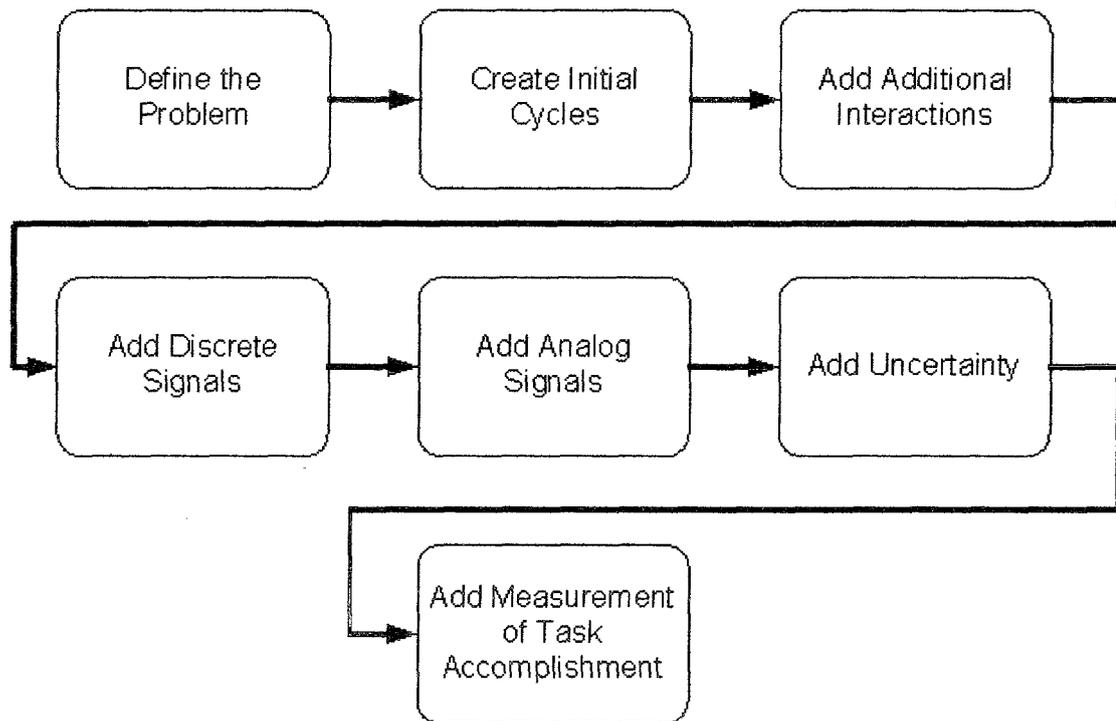


Figure 22 – Creating an Initial Interaction Space Model

4.2.1 Defining the Problem

As with any design process or any design exploration process, the first and one of the most critical steps is to begin to define the problem. For the purposes of interaction space modeling this should begin with a statement describing in plan words what the system is intended to do.

The second and more critical part of defining the problem is to create a bulleted list of tasks that it is desirable for the agent to achieve. These bullets should be specific, should be as simple as possible, and if at all possible should be phrased in a way that lends itself to asking a yes or no question about task accomplishment. It is not necessary, or at this stage desirable, to discuss the conditions necessary within the agent, environment, or both to bring about this task. Significant care should be taken

in defining this bulleted list, as it will be the basis for several subsequent steps of the modeling process and in particular for measurement of task accomplishment.

4.2.2 Creating Initial Cycles

After the initial problem definition, the first step in creating an interaction space model is to create initial cycles of the type shown in Figure 14. In general this initial cycle should be composed entirely of blocks of the binary type. To start creating initial cycles it is recommended that the designer start with a binary block to answer the yes or no question for one of the bullets developed as part of the problem statement. It is then recommended that the designer work counterclockwise around the quadrant abstraction by assessing the conditions in the 1st quadrant counterclockwise that will have an effect on the answer to the yes or no question. These factors should also be posed as yes or no questions and the process can be repeated around the model

4.2.3 Adding Additional Interactions

The purpose of this step is to successively add additional cycles for each of the primary bullets from the problem statement. This should be carried out essentially in the same fashion as the previous section with the exception that there may begin to be relationships that are defined between the cycles. All of the blocks should generally still be of the binary type.

It is critical that the designer not attempt to capture all of the subtleties of the system at this point but rather only look at the most significant one or two factors.

The goal of this step is to create a **VERY** highly abstracted model of the system. Ideally there should be no more than four times the number of blocks in the model at this stage as the number of bullets in the problem definition, although in practice this ratio is almost never maintained. Additional detail and additional factors will be added as needed in the next and subsequent steps. A general rule of thumb is to identify all important interactions that are at least one order away from the task statements from the problem definition step.

4.2.4 Adding Discrete Signals

Once a cycle has been defined for each of the bullets developed in the problem definition phase, it will often be the case that two or more yes or no questions will represent multiple discrete states for one variable. In this case, these should be condensed into a discrete block and discrete values assigned to each case. Additionally there may be cases where only a single yes or no state represents a phenomenon but there are actually more cases that are relevant; for example, on an oven thermostat one could ask if the temperature was right or not. In this step it would probably make sense to expand this to have the options of way too hot, slightly too hot, correct, slightly too cool, and way too cool. Each of these states can be represented by one variable. Depending on the designer's preferences for system modeling it may be desirable to retain the binary blocks to control behaviors or in some cases it may make sense to eliminate these. Examples of each will be given below in the Muramador model.

The addition of discrete blocks serves three basic purposes. The first is to reduce the complication of the model to make it easier to understand and follow. The

second is to allow for more options in modeling the system in order to better represent the system. The third and less obvious purpose is to pave the way for the addition of analog blocks in the next step. In general it is helpful to create a discrete block for a variable before plugging in an analog block. This process will be discussed in the next step.

4.2.5 Adding Analog Signals

The addition of analog signals removes the model from the domain of finite state models with the limitations thereof and moves the model into the realm of continuous models. Ideally the boundary between the agent and the environment in both directions should generally cross with an analog signal as the real world is analog. This may not be the most efficacious modeling method in all cases and it is left to the designer to undertake a cross of the boundary with other than an analog signal. The inherent risk is missing the details of how information is transferred from agent to environment or environment to agent but it can be useful early in the process.

It is left to the discretion of the designer whether to retain the discrete and binary blocks or to use purely analog blocks in some cycles. As with retaining the binary blocks when adding discrete blocks, the advantage is greater representation of the real world in the model, but this occurs at the expense of a more complex and difficult to follow model.

4.2.6 Adding Uncertainty

This step is where, for the first time, the model begins to become a useful representation of the real world. Stochastic blocks are added to the model to

represent uncertainty and variation in the real world. In general random decision blocks will be added to features and behaviors, discrete random number blocks to information and attribute blocks, and other types of random noise to signal and property blocks. It is beyond the scope of this dissertation to provide specific guidelines for adding uncertainty but several examples are given in the models presented below.

4.2.7 Measuring Task Accomplishment

Task accomplishment is measured against the original bullets from the problem definition. It is not always necessary to explicitly represent task accomplishment within the model while building a descriptive model but it is important to keep task accomplishment in mind. It can also be useful to write out a statement of task accomplishment for each bullet of the problem definition from time to time in the modeling process. This is demonstrated below.

When creating a predictive model, it is necessary to explicitly represent task accomplishment within the model. In some cases this can be done through existing elements and in others it may be necessary to add elements to make this measurement. This is discussed in more detail in Chapter 5.

4.3 Refining the Model

Once the initial model has been created, additional steps (shown in Figure 23) are used to create a more sophisticated model. In general this more sophisticated model allows the designer to better understand the interactions and thus to create a new iteration of D_n . Models are built in a bottom up fashion, often with many cycles

of refinement and iterations of D_n . In other words, a very simple model is first constructed and then is improved until the desired level of representation is achieved. To better explain this process a model of a wall following robot has been incrementally developed. This has the advantage of being sufficiently simple that the process can be clearly demonstrated, but yet complicated enough that several conditions must be simultaneously met for successful task accomplishment.

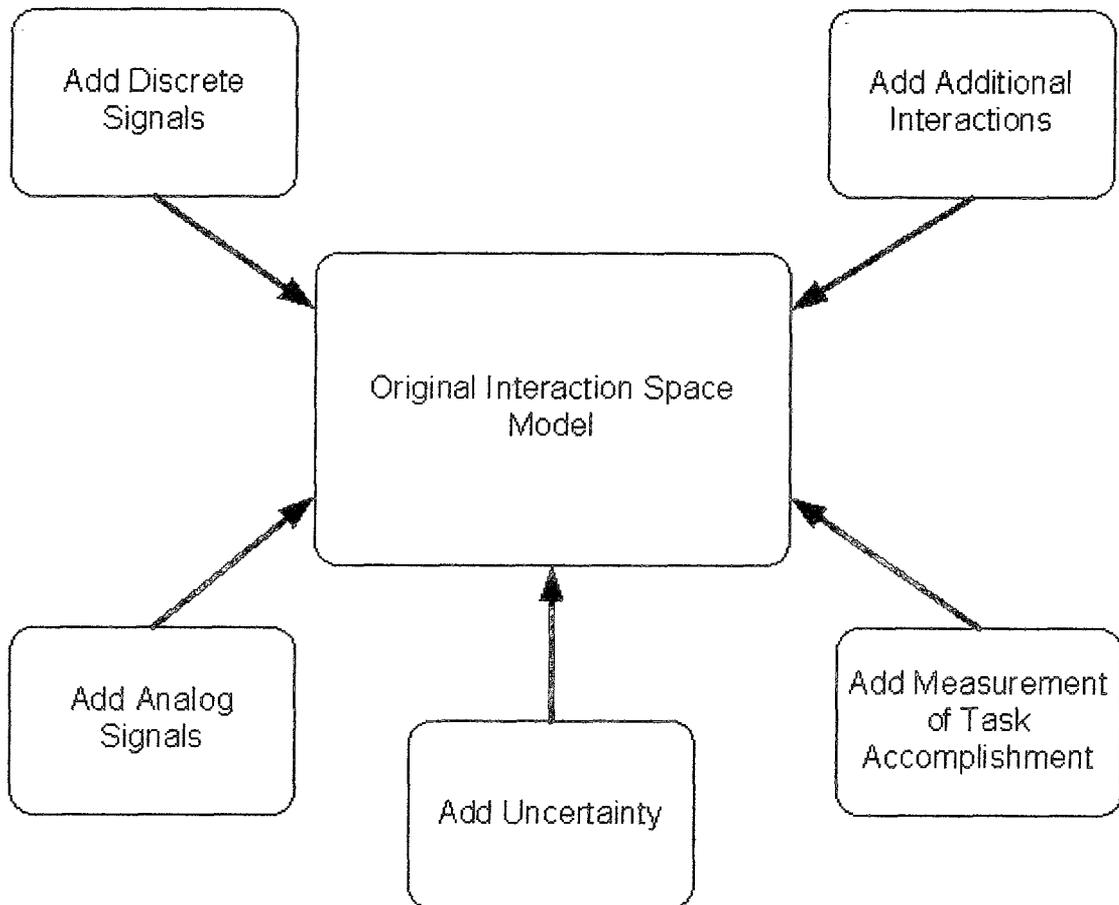


Figure 23 - Options to Refine the Initial model

The steps shown in Figure 23 represent additional complexity and sophistication that can be added to the model. As was discussed above, in the initial model it is generally only desirable to be one step removed from the bulleted tasks

statements from the problem definition phase. Once the initial model is created, it is often necessary to add additional detail. In any complex system it is likely that all relevant system elements will not be defined within one step of the primary tasks.

4.4 Muramador Model

The Muramador [16] is a simple wall-following robot. The Muramador uses a single distance sensor and a set point value to remain at a set distance from the wall. As long as the Muramador is able to move along the wall, the system will continue the wall-following behavior. When the end of a wall is reached, the Muramador will randomly turn in a new direction and proceed either until another wall is reached, or a time threshold is exceeded. If the time threshold is exceeded, the Muramador will once again change direction to a random new heading.

4.4.1 Defining the Problem

As discussed above, a problem statement should consist of a plain language description and of one or more bullets that define the task in ways that are observable and can be rephrased as yes or no questions. A general problem statement for the Muramador can be summarized as follows:

DESIGN A SYSTEM THAT WILL SEEK WALLS. UPON FINDING A WALL, THE DEVICE SHOULD PROCEED ALONG THE WALL AS CLOSELY AS POSSIBLE AT A SETPOINT DISTANCE UNTIL THE END OF THE WALL.

This statement can then be reduced to two discrete tasks:

- FIND WALLS TO FOLLOW.
- FOLLOW THE WALL WHILE REMAINING AT THE SET POINT DISTANCE FROM THE WALL

4.4.2 Creating Initial Cycles

As mentioned above, the first step in creating initial cycles is to find blocks that provide answers to the yes or no questions. In the basic model of the Muramador shown in Figure 24, the block near wall answers the bulleted question: is the agent following walls (or at least in the vicinity, true following will be added later), while the block find wall indicates that the agent has found a new wall to follow, or in this case more specifically that the environment has reacted to the agent looking for a new wall by having a new wall come into proximity with the agent.

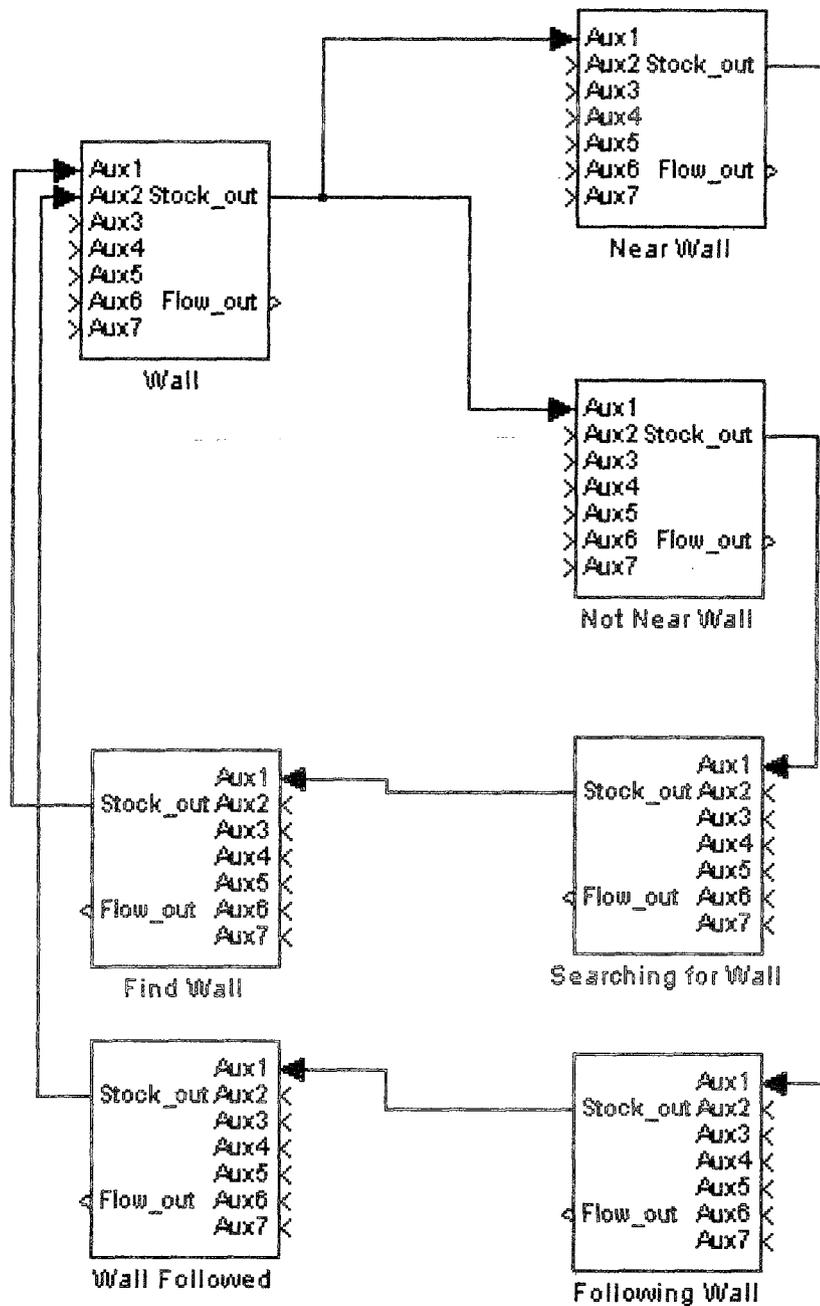


Figure 24 - Basic Cycle for the Muramador Model

Once the initial two binary blocks are added, the designer works counterclockwise to add other blocks that describe the most important conditions that affect the first two blocks. In this case, the environment state is defined by the

presence or absence of a wall. This in turn drives the agent into one of two reactions, either the agent is within sensor range of a wall, or it is not. Based on this reaction, the agent will assume one of two states, either that of following a wall, or that of looking for a wall. This in turn drives the environment to react to the Muramador's attempt to follow the wall or find a wall. Finally, the environment's reaction to the agent will drive the environment state, *i.e.*, the presence of the wall. Additional details including pseudo code and descriptions of each block are given in Table 1.

Block	Pseudo Code	Meaning/Comments
Wall	IF WALL == 1 THEN WALL PRESENT IF WALL == 0 THEN NO WALL PRESENT	BINARY INDICATION OF THE PRESENCE OF A WALL IN THE ENVIRONMENT
Near Wall	IF NEAR WALL == 1 THEN AGENT NEAR WALL IF NEAR WALL == 0 THEN AGENT NOT NEAR WALL	ROBOT'S REACTION IF THERE IS A WALL NEARBY IN THE ENVIRONMENT
Not Near Wall	IF NOT NEAR WALL == 1 THEN NO WALL PRESENT IF NOT NEAR WALL == 0 THEN AGENT NEAR WALL	ROBOT'S REACTION IF THERE IS NOT A WALL NEARBY IN THE ENVIRONMENT
Find Wall	IF NEW WALL APPEARS THEN FIND WALL = 1 IF NO CHANCE IN WALL STATUS OR WALL DISAPPEARS THEN FIND WALL = 0	ENVIRONMENT REACTION TO THE ROBOT LOOKING FOR A WALL
Searching for Wall	IF ROBOT IS SEARCHING FOR WALL THEN SEARCHING FOR WALL = 1 IF ROBOT IS FOLLOWING A WALL THEN SEARCHING FOR WALL = 0	ROBOT STATE TO LOOK FOR A WALL CAUSED BY THE ROBOT REACTION OF NOT NEAR WALL
Wall Followed	IF WALL HAS BEEN FOLLOWED THEN WALL FOLLOWED = 1 IF AGENT HAS NOT FOLLOWED A WALL THEN WALL FOLLOWED = 0	ENVIRONMENT REACTION TO THE ROBOT FOLLOWING THE WALL
Following Wall	IF ROBOT IS SEARCHING FOR WALL THEN FOLLOWING WALL = 0	BINARY AGENT STATE OF FOLLOWING A WALL

	IF ROBOT IS FOLLOWING A WALL THEN FOLLOWING WALL = 1	
--	--	--

Table 1 - Muramador Basic Cycle

4.4.3 Adding Additional Interactions to the Basic Model

Once a basic model exists, additional elements should be added to more adequately represent the interactions between the agent and the environment. The goal at this point is not to more accurately represent the interactions added in the previous step, but rather to add additional interactions that are important to the functionality of the agent but were not essential to produce a minimal model. In particular by the end of this stage every bullet from the original problem definition should be represented. As was mentioned above, the goal here is not to have a complete model but rather to make sure that the basic interactions that are directly relevant to task accomplishment are at least on the model in a highly abstracted and binary state.

Only minimal additional interactions are needed to create a basic binary model of the Muramador; namely, it is necessary to add additional behaviors for being too far or too close to the wall. This in turn will drive two behaviors: moving closer to the wall, and moving farther away from the wall. These changes are shown in Figure 25. Pseudo code and block descriptions are given in Table 2.

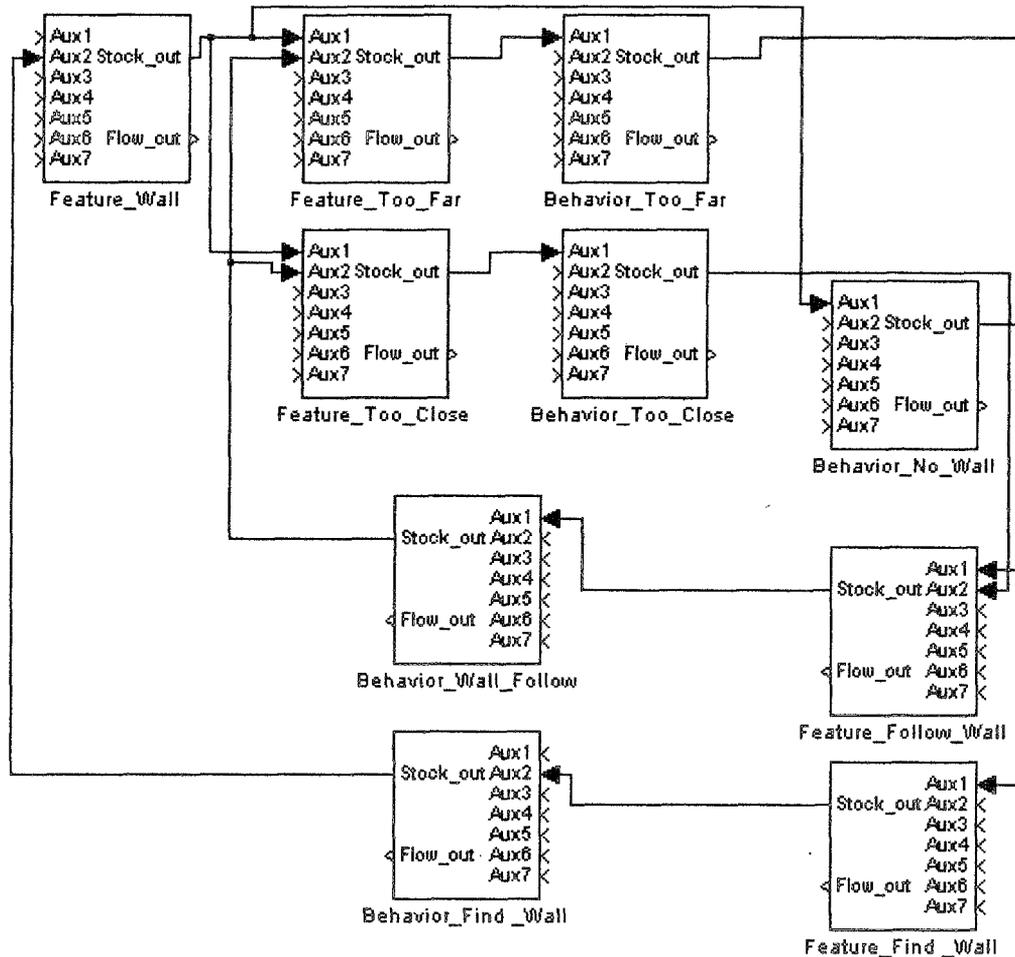


Figure 25 - Muramador Model with Additional Interactions

Block	Pseudo Code	Meaning/Comments
Feature Wall	IF WALL IS PRESENT THEN FEATURE WALL = 1 IF WALL IS ABSENT THEN FEATURE WALL = 0	RECORDS THE PRESENCE OF A WALL IN THE ENVIRONMENT
Feature Too Far	IF ROBOT IS TOO CLOSE THEN FEATURE TOO FAR = 0 IF ROBOT IS TOO FAR THEN FEATURE TOO FAR = 1 IF FEATURE WALL == 0 THEN FEATURE TOO FAR = 0	BINARY ENVIRONMENT PROPERTY ACTIVE IF THE ROBOT IS TOO FAR FROM THE WALL
Behavior Too Far	IF FEATURE TOO FAR == 1 THEN BEHAVIOR TOO	ROBOT REACTION TO BEING TOO FAR

	FAR = 1 IF FEATURE TOO FAR == 0 THEN BEHAVIOR TOO FAR = 0	FROM THE WALL
Feature Too Close	IF ROBOT IS TOO CLOSE THEN FEATURE TOO FAR = 1 IF ROBOT IS TOO FAR THEN FEATURE TOO FAR = 0 IF FEATURE WALL == 0 THEN FEATURE TOO CLOSE = 0	BINARY ENVIRONMENT PROPERTY ACTIVE IF THE ROBOT IS TOO CLOSE TO THE WALL
Behavior Too Close	IF FEATURE TOO CLOSE == 1 THEN BEHAVIOR TOO CLOSE = 1 IF FEATURE TOO CLOSE == 0 THEN BEHAVIOR TOO CLOSE = 0	ROBOT REACTION TO BEING TOO CLOSE TO THE WALL
Behavior No Wall	IF FEATURE WALL == 1 THEN BEHAVIOR NO WALL = 0 IF FEATURE WALL == 0 THEN BEHAVIOR NO WALL = 1	ROBOT REACTION TO NO WALL
Behavior Wall Follow	BEHAVIOR WALL FOLLOW = FEATURE FOLLOW WALL	ENVIRONMENTAL REACTION TO THE ROBOT FOLLOWING THE WALL
Behavior Find Wall	BEHAVIOR FIND WALL = FEATURE FIND WALL	ENVIRONMENTAL REACTION TO THE ROBOT FINDING A NEW WALL TO FOLLOW
Feature Follow Wall	IF BEHAVIOR TOO CLOSE == 1 BEHAVIOR TOO FAR == 1 THEN FEATURE FOLLOW WALL = 1 IF BEHAVIOR TOO CLOSE == 0 && BEHAVIOR TOO FAR == 0 THEN FEATURE FOLLOW WALL == 0	ROBOT STATE OF FOLLOWING THE WALL
Feature Find Wall	IF BEHAVIOR NO WALL == 0 THEN FEATURE FIND WALL = 0 IF BEHAVIOR NO WALL == 1 && A NEW WALL IS FOUND THEN FEATURE FIND WALL = 1	ROBOT STATE OF SEARCHING FOR A WALL

Table 2 - Muramador 1st Set of Additional Interactions

Although the model shown in Figure 25 has additional interactions over the initial model it does not yet contain all of the key elements. The model below shown in Figure 26 includes options for the agent to move towards or away from the wall. The details of this model are shown in Table 3.

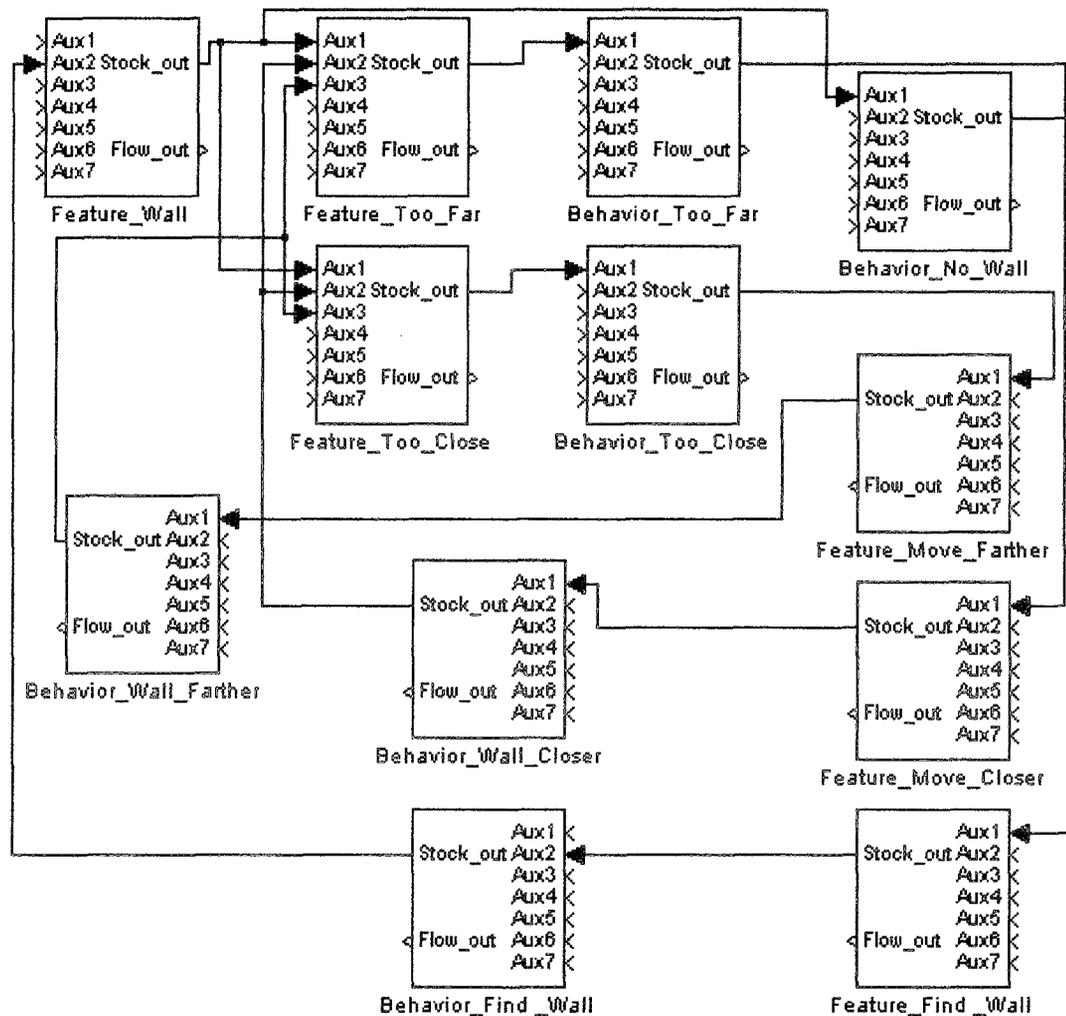


Figure 26 - Muramador Model with Additional Interactions

Block	Pseudo Code	Meaning/Comment
Feature Wall	IF WALL IS PRESENT THEN FEATURE WALL = 1 IF WALL IS ABSENT THEN FEATURE WALL = 0	RECORDS THE PRESENCE OF A WALL IN THE ENVIRONMENT

Feature Too Far	IF ROBOT IS TOO CLOSE THEN FEATURE TOO FAR = 0 IF ROBOT IS TOO FAR THEN FEATURE TOO FAR = 1 IF FEATURE WALL == 0 THEN FEATURE TOO FAR = 0	BINARY ENVIRONMENT PROPERTY ACTIVE IF THE ROBOT IS TOO FAR FROM THE WALL
Feature Too Close	IF ROBOT IS TOO CLOSE THEN FEATURE TOO FAR = 1 IF ROBOT IS TOO FAR THEN FEATURE TOO FAR = 0 IF FEATURE WALL == 0 THEN FEATURE TOO CLOSE = 0	BINARY ENVIRONMENT PROPERTY ACTIVE IF THE ROBOT IS TOO CLOSE TO THE WALL
Behavior Too Far	IF FEATURE TOO FAR == 1 THEN BEHAVIOR TOO FAR = 1 IF FEATURE TOO FAR == 0 THEN BEHAVIOR TOO FAR = 0	ROBOT REACTION TO BEING TOO FAR FROM THE WALL
Behavior Too Close	IF FEATURE TOO CLOSE == 1 THEN BEHAVIOR TOO CLOSE = 1 IF FEATURE TOO CLOSE == 0 THEN BEHAVIOR TOO CLOSE = 0	ROBOT REACTION TO BEING TOO CLOSE TO THE WALL
Behavior No Wall	IF FEATURE WALL == 1 THEN BEHAVIOR NO WALL = 0 IF FEATURE WALL == 0 THEN BEHAVIOR NO WALL = 1	ROBOT REACTION TO NO WALL
Feature Move Farther	FEATURE MOVE FURTHER = BEHAVIOR TOO CLOSE	ROBOT STATE OF MOVING AWAY FROM THE WALL
Feature Move Closer	FEATURE MOVE CLOSER = BEHAVIOR TOO FAR	ROBOT STATE OF MOVING TOWARDS THE WALL
Feature Find Wall	FEATURE FIND WALL = BEHAVIOR NO WALL	ROBOT STATE OF SEARCHING FOR A NEW WALL
Behavior Wall Closer	BEHAVIOR WALL CLOSER = FEATURE MOVE CLOSER	ENVIRONMENT REACTION TO THE ROBOT MOVING CLOSER
Behavior Wall Farther	BEHAVIOR WALL FURTHER = FEATURE MOVE	ENVIRONMENT REACTION TO THE

	FURTHER	ROBOT MOVING AWAY
Behavior Find Wall	BEHAVIOR FIND WALL = FEATURE FIND WALL	ENVIRONMENT REACTION TO THE ROBOT SEARCHING FOR A WALL

Table 3 - Muramador 2nd Set of Additional Interactions

The model above contains the key elements for both of the bullets identified in the problem definition phase. Specifically there are elements identified that can change the answer to the yes or no question posed by the problem definition bullets and these elements are formed into logically consistent cycles. It is not necessary at this point to worry about second order effects such as what causes a wall to start or what causes a wall to end. It is sufficient for the moment that it is identified that these are important interactions in the system. The next step is to add discrete signals to the model in order to improve the fidelity of the model.

4.4.4 Adding Discrete Signals

Once the first order interactions relative to the initial problem definition are included in the model as described above, it is generally time to begin to add non-binary elements. The first step in this process is to add discrete elements. Within the Muramador model shown in Figure 27, the attribute distance and the info distance blocks have been added. As mentioned previously, the boundary between agent and environment or environment and agent should always be crossed with the same data type on each side of the boundary. The distance blocks added below now have three specific states: too close, too far, and no wall. For modeling purposes these three states are assigned a number arbitrarily which then represents that state within the model. The pseudo code and descriptions for these blocks are given in Table 4.

Although the same physical system can be represented by N binary blocks where N is the number of states, the discrete block cleans up the model and makes it more manageable. In addition the discrete blocks are usually the first step on the way to adding continuous blocks as will be discussed in the next section.

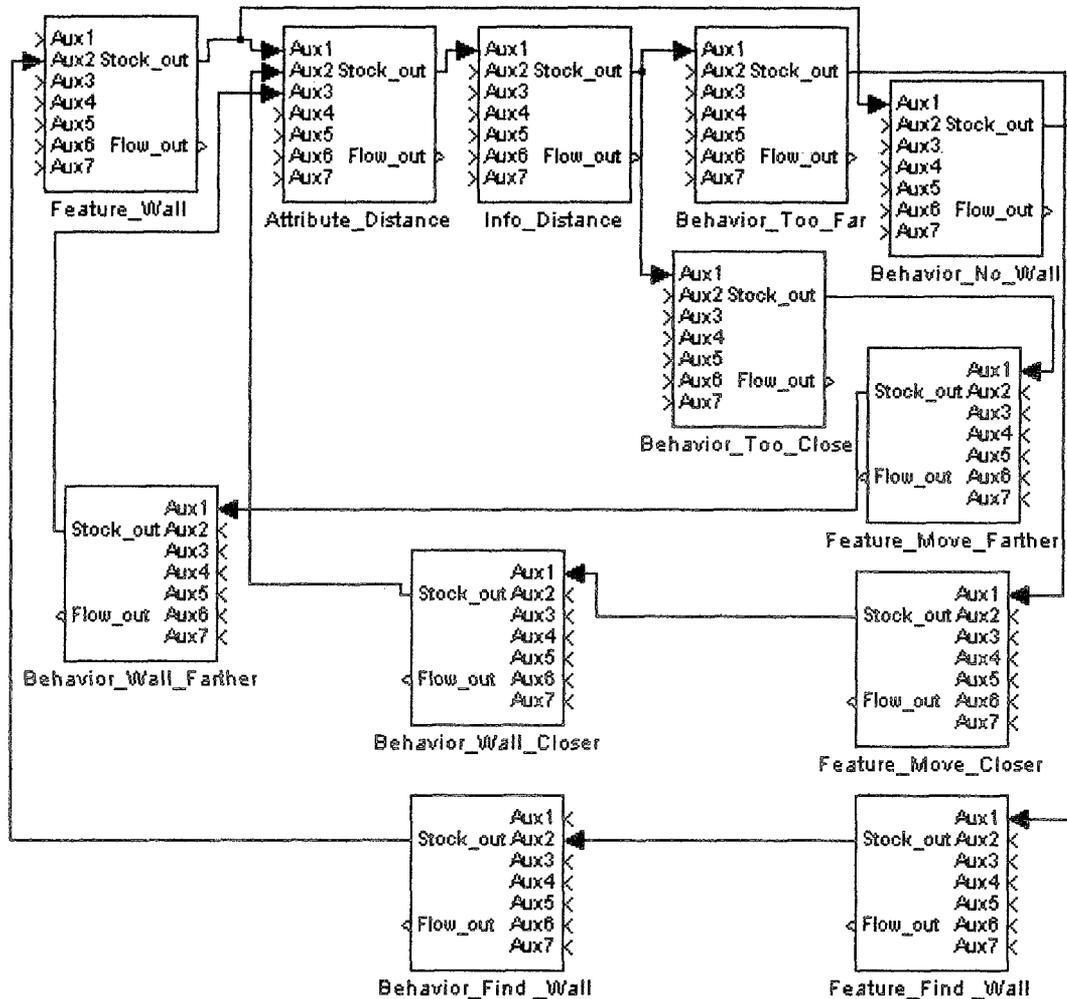


Figure 27 - Muramador Model with Discrete Signals

Block	Pseudo Code	Meaning/Comment
Feature Wall	<pre> IF WALL IS PRESENT THEN FEATURE WALL = 1 IF WALL IS ABSENT THEN FEATURE WALL = 0 </pre>	RECORDS THE PRESENCE OF A WALL IN THE ENVIRONMENT

Attribute Distance	IF ROBOT IS TOO CLOSE THEN ATTRIBUTE DISTANCE = 1 IF ROBOT IS TOO FAR THEN ATTRIBUTE DISTANCE = 2 IF NO WALL IS PRESENT THEN ATTRIBUTE DISTANCE = 0	FINITE STATE DESCRIPTION OF THE DISTANCE OF THE ROBOT FROM THE WALL
Info Distance	INFO DISTANCE = ATTRIBUTE DISTANCE	ROBOT REACTION TO THE DISTANCE FROM THE WALL
Behavior Too Far	IF INFO DISTANCE == 2 THEN BEHAVIOR TOO FAR = 1 IF INFO DISTANCE == 1 OR INFO DISTANCE == 0 THEN BEHAVIOR TOO FAR = 0	ROBOT REACTION TO BEING TOO FAR FROM THE WALL
Behavior Too Close	IF INFO DISTANCE == 1 THEN BEHAVIOR TOO CLOSE = 1 IF INFO DISTANCE == 2 OR INFO DISTANCE == 0 THEN BEHAVIOR TOO CLOSE = 0	ROBOT REACTION TO BEING TOO CLOSE TO THE WALL
Behavior No Wall	IF FEATURE WALL == 1 THEN BEHAVIOR NO WALL = 0 IF FEATURE WALL == 0 THEN BEHAVIOR NO WALL = 1	ROBOT REACTION TO NO WALL
Feature Move Further	FEATURE MOVE FURTHER = BEHAVIOR TOO CLOSE	ROBOT STATE OF MOVING AWAY FROM THE WALL
Feature Move Closer	FEATURE MOVE CLOSER = BEHAVIOR TOO FAR	ROBOT STATE OF MOVING TOWARDS THE WALL
Feature Find Wall	FEATURE FIND WALL = BEHAVIOR NO WALL	ROBOT STATE OF SEARCHING FOR A NEW WALL
Behavior Wall Closer	BEHAVIOR WALL CLOSER = FEATURE MOVE CLOSER	ENVIRONMENT REACTION TO THE ROBOT MOVING CLOSER
Behavior Wall Further	BEHAVIOR WALL FURTHER = FEATURE MOVE FURTHER	ENVIRONMENT REACTION TO THE ROBOT MOVING AWAY

Behavior Find Wall	BEHAVIOR FIND WALL = FEATURE FIND WALL	ENVIRONMENT REACTION TO THE ROBOT SEARCHING FOR A WALL
--------------------	---	---

Table 4 - Muramador 1st Discrete Model

In addition to the discrete blocks shown in Figure 27, it also makes sense and simplifies the model to create discrete blocks for the direction of the Muramador and also for the change in the direction of the Muramador within the environment. These changes are shown in Figure 28 with the corresponding pseudo code and block descriptions in Table 5.

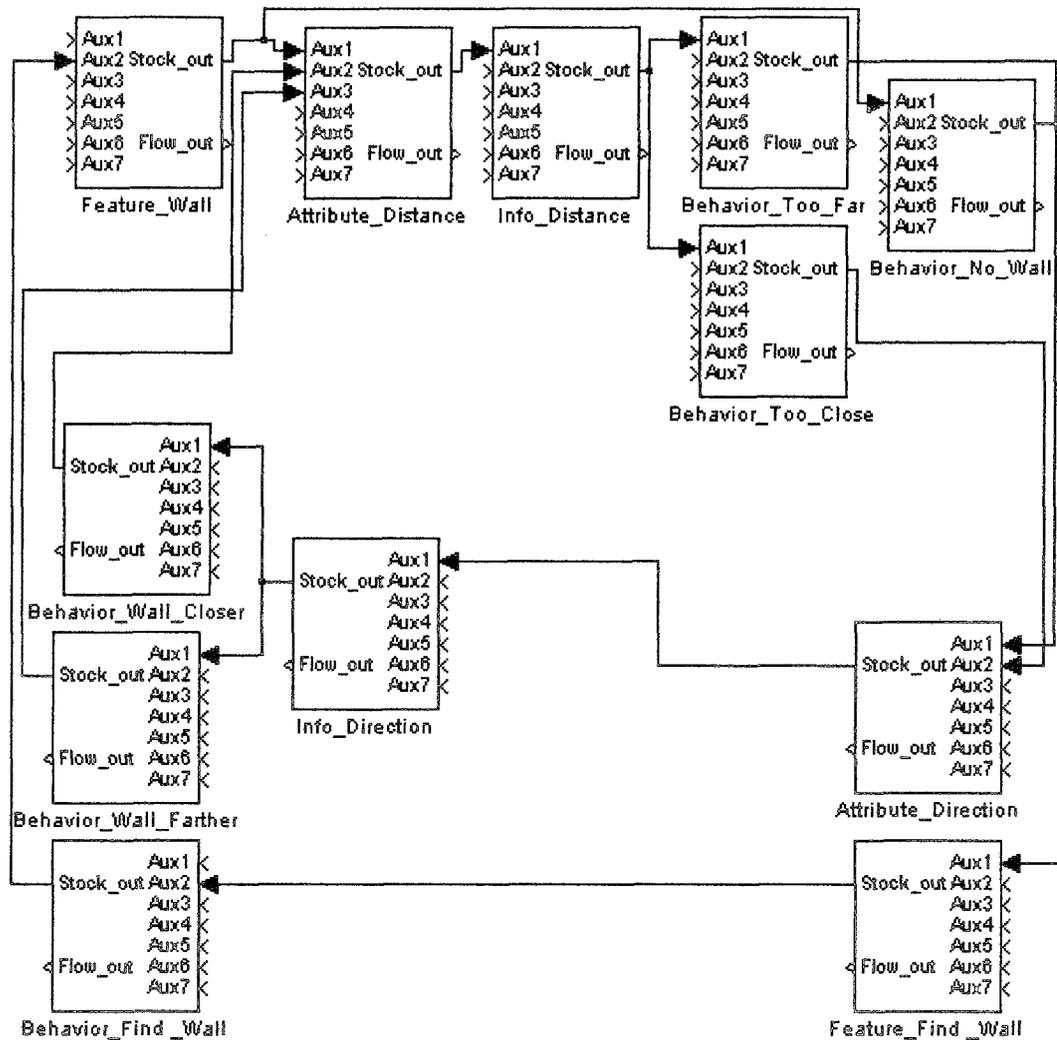


Figure 28 - Muramador Model Additional Discrete Elements

Block	Pseudo Code	Meaning/Comment
Feature Wall	<pre> IF WALL IS PRESENT THEN FEATURE WALL = 1 IF WALL IS ABSENT THEN FEATURE WALL = 0 </pre>	RECORDS THE PRESENCE OF A WALL IN THE ENVIRONMENT
Attribute Distance	<pre> IF ROBOT IS TOO CLOSE THEN ATTRIBUTE DISTANCE = 1 IF ROBOT IS TOO FAR THEN ATTRIBUTE DISTANCE = 2 </pre>	FINITE STATE DESCRIPTION OF THE DISTANCE OF THE ROBOT FROM THE WALL

	IF NO WALL IS PRESENT THEN ATTRIBUTE DISTANCE = 0	
Info Distance	INFO DISTANCE = ATTRIBUTE DISTANCE	ROBOT REACTION TO THE DISTANCE FROM THE WALL
Behavior Too Far	IF INFO DISTANCE == 2 THEN BEHAVIOR TOO FAR = 1 IF INFO DISTANCE == 1 OR INFO DISTANCE == 0 THEN BEHAVIOR TOO FAR = 0	ROBOT REACTION TO BEING TOO FAR FROM THE WALL
Behavior Too Close	IF INFO DISTANCE == 1 THEN BEHAVIOR TOO CLOSE = 1 IF INFO DISTANCE == 2 OR INFO DISTANCE == 0 THEN BEHAVIOR TOO CLOSE = 0	ROBOT REACTION TO BEING TOO CLOSE TO THE WALL
Behavior No Wall	IF FEATURE WALL == 1 THEN BEHAVIOR NO WALL = 0 IF FEATURE WALL == 0 THEN BEHAVIOR NO WALL = 1	ROBOT REACTION TO NO WALL
Attribute Direction	IF BEHAVIOR TOO FAR == 1 THEN ATTRIBUTE DIRECTION = 1 IF BEHAVIOR TOO CLOSE == 1 THEN ATTRIBUTE DIRECTION = 2	AGENT STATE REFLECTING WHETHER THE AGENT IS MOVING TOWARDS OR AWAY FROM THE WALL
Feature Find Wall	FEATURE FIND WALL = BEHAVIOR NO WALL	ROBOT STATE OF SEARCHING FOR A NEW WALL
Info Direction	INFO DIRECTION = ATTRIBUTE DIRECTION	REPRESENTS THE ENVIRONMENT REACTION TO THE ROBOT MOVING TOWARDS OR AWAY FROM THE WALL
Behavior Find Wall	IF FEATURE FIND WALL == 1 && A NEW WALL APPEARS THEN BEHAVIOR FIND WALL = 1 ELSE BEHAVIOR FIND WALL = 0	ENVIRONMENT REACTION TO THE ROBOT SEARCHING FOR A WALL

Behavior Wall Further	<pre> IF INFO DIRECTION == 1 THEN BEHAVIOR WALL FURTHER = 1 IF INFO DIRECTION == 2 THEN BEHAVIOR WALL FURTHER = 0 </pre>	ENVIRONMENT REACTION TO THE ROBOT MOVING AWAY
Behavior Wall Closer	<pre> IF INFO DIRECTION == 1 THEN BEHAVIOR WALL CLOSER = 0 IF INFO DIRECTION == 2 THEN BEHAVIOR WALL CLOSER = 1 </pre>	ENVIRONMENT REACTION TO THE ROBOT MOVING CLOSER

Table 5 - Muramador Model Additional Discrete Elements

4.4.5 Adding Analog Signals

The addition of analog elements to the model, shown in Figure 29, represents the transition to a quantitative model. This model now has the first order interactions necessary for the two task bullets from the problem definition phase. As can be seen below the discrete blocks in the environment model as well as the attribute direction block have been directly replaced with continuous elements representing the distance from the agent to the wall and the change in distance that the agent seeks to carry out respectively. The agent reaction has added the signal distance block, but for illustrative purposes the info distance and relevant behaviors for moving closer or farther from the wall have been retained. It is left to the discretion of the designer when it is appropriate to keep this detail and when it should be bypassed. In case of doubt it is recommended that the additional detail be kept. The relevant pseudo code and descriptions for this model are defined in Table 6.

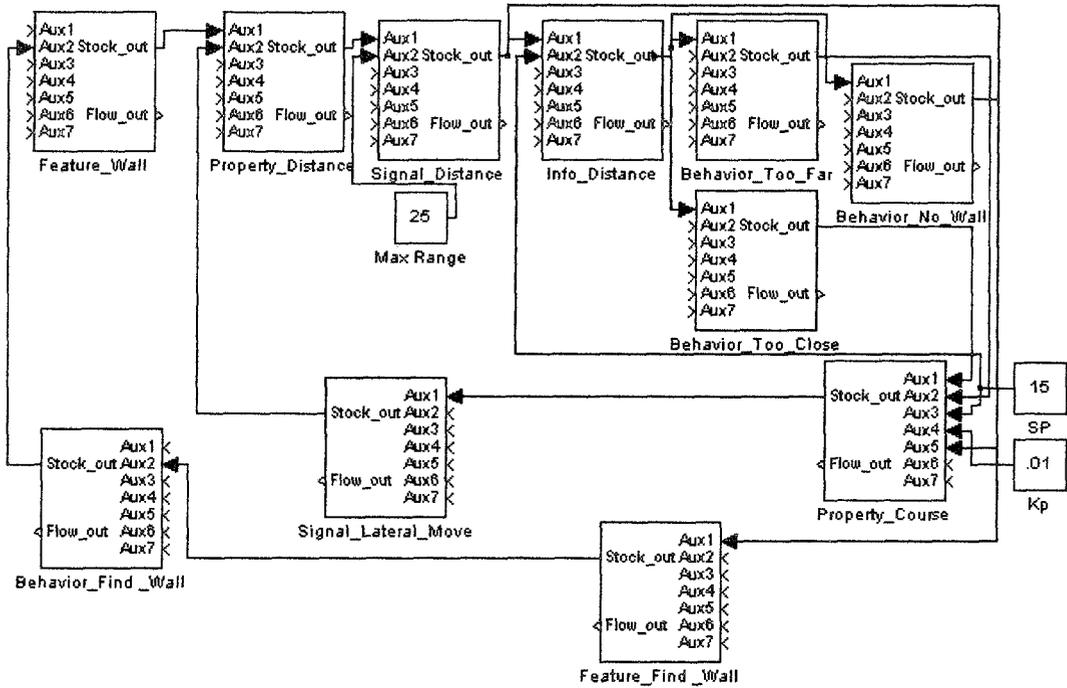


Figure 29 - Muramador Model with Analog Signals

Block	Pseudo Code	Meaning/Comment
Feature Wall	<pre> IF WALL IS PRESENT THEN FEATURE WALL = 1 IF WALL IS ABSENT THEN FEATURE WALL = 0 </pre>	RECORDS THE PRESENCE OF A WALL IN THE ENVIRONMENT
Property Distance	<pre> IF FEATUE WALL = 1 THEN PROPERTY DISTANCE (N) = PROPERTY DISTANCE (N-1) + SIGNAL LATERAL MOVE (N) ELSE PROPERTY DISTANCE = 0 </pre>	RECORDS THE DISTANCE BETWEEN THE AGENT AND THE WALL
Signal Distance	<pre> IF PROPERTY DISTANCE == 0 THEN SIGNAL DISTANCE = MAX RANGE ELSE SIGNAL DISTANCE = PROPERTY DISTANCE </pre>	ESSENTIALLY THE DISTANCE SENSOR OUTPUT FROM THE AGENT
Max Range	CONSTANT = 25	DESCRIBES THE MAXIMUM RANGE OF THE SENSOR

Info Distance	IF SIGNAL DISTANCE == 0 THEN INFO DISTANCE = 0 IF SIGNAL DISTANCE >= SP THEN INFO DISTANCE = 1 IF SIGNAL DISTANCE < SP THEN INFO DISTANCE = 2	REPRESENTS THE AGENT'S DECISION OF WHICH DIRECTION TO MOVE
Behavior Too Far	IF SIGNAL DISTANCE == 1 THEN BEHAVIOR TOO FAR = 1 ELSE BEHAVIOR TOO FAR = 0	ROBOT REACTION TO BEING TOO FAR FROM THE WALL
Behavior Too Close	IF SIGNAL DISTANCE == 2 THEN BEHAVIOR TOO FAR = 1 ELSE BEHAVIOR TOO FAR = 0	ROBOT REACTION TO BEING TOO CLOSE TO THE WALL
Behavior No Wall	IF SIGNAL DISTANCE == 0 THEN BEHAVIOR TOO FAR = 1 ELSE BEHAVIOR TOO FAR = 0	ROBOT REACTION TO NO WALL
Property Course	IF BEHAVIOR TOO FAR == 1 THEN PROPERTY COURSE = (SP-SIGNAL DISTANCE)*KP ELSEIF BEHAVIOR TOO CLOSE == 1 THEN PROPERTY COURSE = (SP-SIGNAL DISTANCE)*KP ELSE PROPERTY COURSE = 0	DETERMINES THE AMOUNT THAT THE AGENT WILL MOVE TOWARDS OR AWAY FROM THE WALL
SP	CONSTANT = 15	DEFINES THE DESIRED DISTANCE OF THE AGENT FROM THE WALL
KP	CONSTANT = 0.01	PROPORTIONALITY CONSTANT FOR THE PROPORTIONAL CONTROL SYSTEM
Feature Find Wall	FEATURE FIND WALL = BEHAVIOR NO WALL	ROBOT STATE OF SEARCHING FOR A NEW WALL
Signal Lateral Move	SIGNAL LATERAL MOVE = PROPERTY COURSE	ENVIRONMENT REACTION TO THE ROBOT MOVING LATERALLY AFFECTS THE DISTANCE TO THE WALL

Behavior Find Wall	IF FEATURE FIND WALL == 1 && A NEW WALL APPEARS THEN BEHAVIOR FIND WALL = 1 ELSE BEHAVIOR FIND WALL = 0	ENVIRONMENT REACTION TO THE ROBOT SEARCHING FOR A WALL
--------------------	--	---

Table 6 - Muramador Model with Analog Blocks

4.4.6 Adding Uncertainty to the Model

This step adds the features that control the beginning and end of walls. The end of a wall is modeled as a part of the environment state and is based on a standard random decision block as described in Chapter 3. The beginning of a wall is modeled as an environmental reaction to the agent state behavior of looking for a wall. This is also modeled as a standard random decision block, but is only activated if the agent is looking for a wall. This is shown in Figure 30, which represents the full Muramador model as implemented for this dissertation. The relevant pseudo code and block descriptions are given in Table 7.

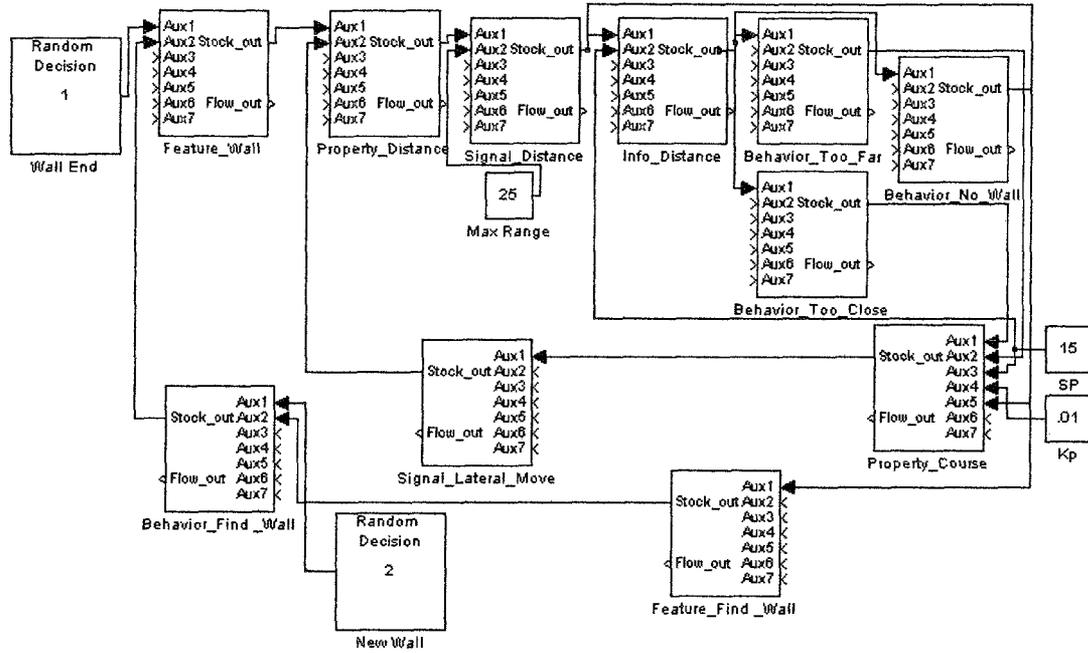


Figure 30 - Full Muramador Model

Block	Pseudo Code	Meaning/Comment
Feature Wall	<pre> IF BEHAVIOR FIND WALL == 1 THEN FEATURE WALL = 1 ELSEIF WALL END == 1 THEN FEATURE WALL = 0 ELSE FEATURE WALL (N) = FEATURE WALL (N-1) </pre>	RECORDS THE PRESENCE OF A WALL IN THE ENVIRONMENT
Property Distance	<pre> IF FEATUE WALL = 1 THEN PROPERTY DISTANCE (N) = PROPERTY DISTANCE (N-1) + SIGNAL LATERAL MOVE (N) ELSE PROPERTY DISTANCE = 0 </pre>	RECORDS THE DISTANCE BETWEEN THE AGENT AND THE WALL
Signal Distance	<pre> IF PROPERTY DISTANCE == 0 THEN SIGNAL DISTANCE = MAX RANGE ELSE SIGNAL DISTANCE = PROPERTY DISTANCE </pre>	ESSENTIALLY THE DISTANCE SENSOR OUTPUT FROM THE AGENT
Max Range	CONSTANT = 25	DESCRIBES THE MAXIMUM RANGE OF

		THE SENSOR
Info Distance	IF SIGNAL DISTANCE == 0 THEN INFO DISTANCE = 0 IF SIGNAL DISTANCE >= SP THEN INFO DISTANCE = 1 IF SIGNAL DISTANCE < SP THEN INFO DISTANCE = 2	REPRESENTS THE AGENT'S DECISION OF WHICH DIRECTION TO MOVE
Behavior Too Far	IF SIGNAL DISTANCE == 1 THEN BEHAVIOR TOO FAR = 1 ELSE BEHAVIOR TOO FAR = 0	ROBOT REACTION TO BEING TOO FAR FROM THE WALL
Behavior Too Close	IF SIGNAL DISTANCE == 2 THEN BEHAVIOR TOO FAR = 1 ELSE BEHAVIOR TOO FAR = 0	ROBOT REACTION TO BEING TOO CLOSE TO THE WALL
Behavior No Wall	IF SIGNAL DISTANCE == 0 THEN BEHAVIOR TOO FAR = 1 ELSE BEHAVIOR TOO FAR = 0	ROBOT REACTION TO NO WALL
Property Course	IF BEHAVIOR TOO FAR == 1 THEN PROPERTY COURSE = (SP-SIGNAL DISTANCE)*KP ELSEIF BEHAVIOR TOO CLOSE == 1 THEN PROPERTY COURSE = (SP-SIGNAL DISTANCE)*KP ELSE PROPERTY COURSE = 0	DETERMINES THE AMOUNT THAT THE AGENT WILL MOVE TOWARDS OR AWAY FROM THE WALL
SP	CONSTANT = 15	DEFINES THE DESIRED DISTANCE OF THE AGENT FROM THE WALL
KP	CONSTANT = 0.01	PROPORTIONALITY CONSTANT FOR THE PROPORTIONAL CONTROL SYSTEM
Feature Find Wall	FEATURE FIND WALL = BEHAVIOR NO WALL	ROBOT STATE OF SEARCHING FOR A NEW WALL
Signal Lateral Move	SIGNAL LATERAL MOVE = PROPERTY COURSE	ENVIRONMENT REACTION TO THE ROBOT MOVING Laterally AFFECTS THE DISTANCE TO THE

		WALL
Behavior Find Wall	IF FEATURE FIND WALL == 1 && NEW WALL == 1 THEN BEHAVIOR FIND WALL = 1 ELSE BEHAVIOR FIND WALL = 0	ENVIRONMENT REACTION TO THE ROBOT SEARCHING FOR A WALL
Wall End	WALL END = 1 2% OF TIME STEPS AT RANDOM ELSE WALL END = 0	ENVIRONMENT END OF A WALL AS A RESULT OF THE AGENT FOLLOWING IT TO THE END
New Wall	WALL END = 1 1% OF TIME STEPS AT RANDOM ELSE WALL END = 0	ENVIRONMENT BEGINNING OF A WALL IN RESPONSE TO AGENT LOOKING FOR A WALL

Table 7 - Muramador Model with Uncertainty

4.4.7 Determining Task Accomplishment

As discussed above, the two tasks identified for the Muramador are:

- FIND WALLS TO FOLLOW
- FOLLOW THE WALL WHILE REMAINING AT THE SET POINT DISTANCE FROM THE WALL

Both of these are physical tasks. The first task can be assessed by observing the “feature wall block”. In this particular model, the process of finding walls is relatively unaddressed, and this is reflected in the assessment of the accomplishment of that task. The second task is more carefully modeled, and hence assessment of task accomplishment is correspondingly easier. In this case, the value of the block “property distance” can be compared to the desired set point.

4.4.8 Suggested Additions for the Muramador Model

The Muramador model above represents the most important interactions and only those interactions that are within one step of the bulleted task accomplishment

statements from the problem definition phase. There are a significant number of additions that could be made. For example, the model currently deals with only straight lines. The incorporation of internal and external corners into the model will bring it into much better alignment with typical manmade environments. Another example is the addition of provisions for hitting a wall. Currently the model is limited to situations where this does not occur.

These limitations are not inherently flaws in the model, but rather represent differing levels of abstraction. These features, along with others that are desired, can be added as additional cycles in the exploration based design process if they are needed. This process is discussed in section 4.3 Refining the Model.

As additional elements are defined, the fidelity of the model increases and the level of abstraction decreases; that is, the model incorporates more and more of the complexity of how the designer sees the real world. Note that the complexity added to the model only represents how the designer sees the world and not the actual state of the world. This is true of all design processes and is the reason that no modeling method or design process can eliminate the need for real world testing.

As with any other modeling or design process, a critical aspect of the use of the process is determining when it is good enough. Although it would be nice to have a hard and fast rule, as with any other process the engineer must ultimately exercise judgment by considering the ramifications of failure, the tradeoff between modeling and testing, and the available resources for completion of the project.

4.5 Multi-Agent Foraging Model

Understanding self-organization among multi-entity and multi-agent systems is a significant field of study within biological and robotic systems. An example of this involves understanding how insect colonies such as ants and bees are able to organize insects of the worker castes apparently without centralized control, and presumably without a cognitive understanding on the part of each individual insect of the dynamics of properly running a nest or allocation of tasks amongst the insects. A popular emerging theory among a variety of insects involves variable thresholds [31]. In such a system, each individual worker is aware of certain tangible and observable properties of the collective, for example stored food or refuse within the nest or hive. Each individual within the colony will have a slightly different threshold for action to correct each particular property. For example, if stored food levels drop slightly, those individuals with a high sensitivity to lack of food will immediately begin to look for food. As stored food levels continue to drop, more and more individuals will become involved in the task of searching for food as each individual's threshold is progressively exceeded. In this way, a negative feedback loop is effectively formed that attempts to keep the food level of the colony as close as possible to a set point determined by the collective thresholds of the individuals that make up the colony.

Roboticists have become interested in this system both in an attempt to understand nature and as a possible solution technique for control of complex multi-agent systems. One such project [32] has successfully used a variable threshold technique to get a group of robots to self-organize to sustain a collective energy supply by searching for energy modules within the environment. This experiment is

sufficiently complex and well defined to serve as a good model to test a multi-agent system interaction space model. Figure 31 shows Krieger and Billeter's implementation of the multi-agent foraging system.

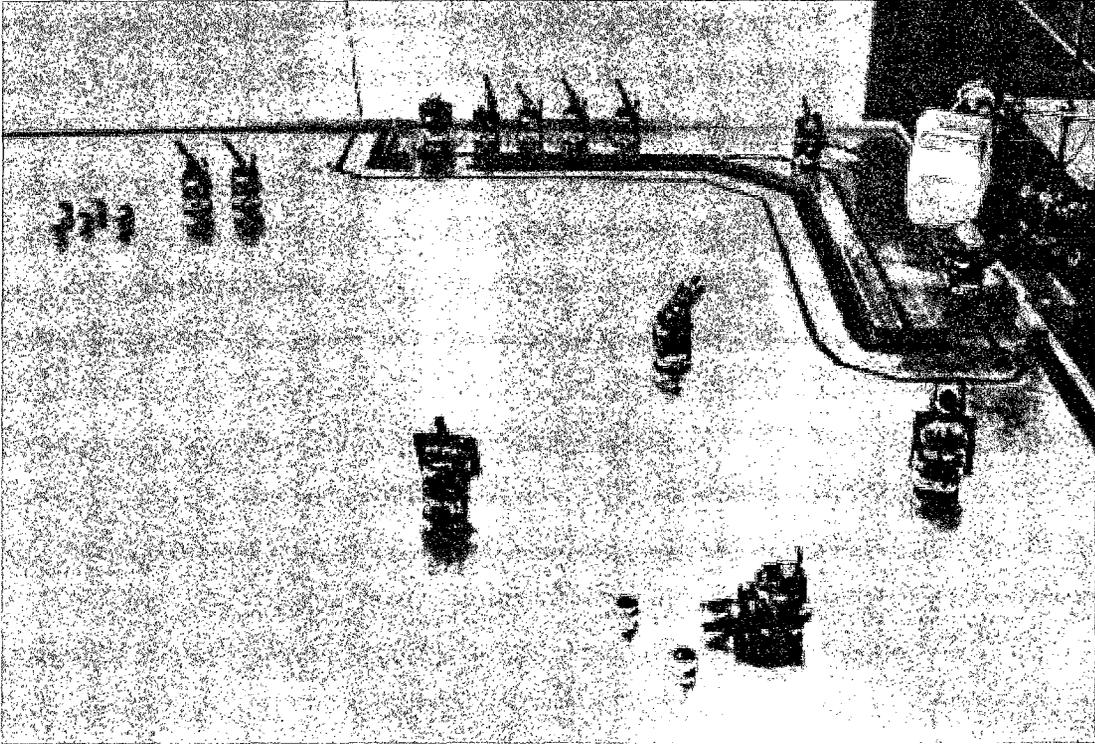


Figure 31 - Foraging Robots as Implemented by Krieger and Billeter [32]

4.5.1 Defining the Problem

A problem statement for the multi-agent foraging model can be summarized as follows:

KEEP SUFFICIENT ENERGY IN THE NEST BY COLLECTING AND RETURNING ENERGY MODULES FROM THE SURROUNDING ENVIRONMENT

This problem statement can in turn be broken down into a series of tasks:

SENSE THE NEST ENERGY LEVEL AND FORAGE WHEN NEEDED
FORAGE FOR AND LOCATE ENERGY MODULES
RETURN THE ENERGY MODULES TO THE NEST
MAINTAIN THE ENERGY LEVEL OF INDIVIDUAL AGENTS

4.5.2 Addressing the Multi-Agent Issue

As discussed in Chapter 3, in a multi-agent system, there is a single representation of the environment and its reactions, while each agent has its properties and reactions represented separately from other agents. This has both the advantage and the disadvantage that the information exchanged between the environment and the agent must be standardized. This is a disadvantage in that it limits the freedom to design custom agents, but is an advantage in the sense that the designer must carefully consider and formalize this interface, which should aid both in system understanding and in eventual agent construction.

In the event that the agents are similar or identical, it makes sense to modularize portions of the subsystem. Modularization of the agent is a natural extension of the multi-agent modeling concept shown in Figure 15. Beyond the modularization of the agent, it also is helpful to modularize those portions of the environment that must interact separately with each agent. The specific implementation of the modularization for the foraging model is discussed and shown in the next section.

4.5.3 Foraging Model

As mentioned in the previous section, much of the multi-agent foraging model is modularized. Figure 32 shows the block that is used for each agent, while Figure 33 shows the internal structure of the agent block. This block accounts for both the agent reactions (which have inputs from the environment properties on the left side of

the block) and the agent properties (which have outputs to the environment reactions on the right side of the block). Code for this model is given in Appendix B.2 – Multi-Agent Foraging Model. The specific development steps of this model do not substantially add to the understanding of the modeling process and hence are not included. For a detailed walk through of the functionality of a similar model, please see [2]

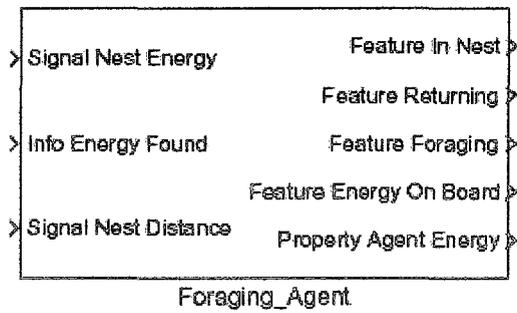


Figure 32 - Foraging Agent Block

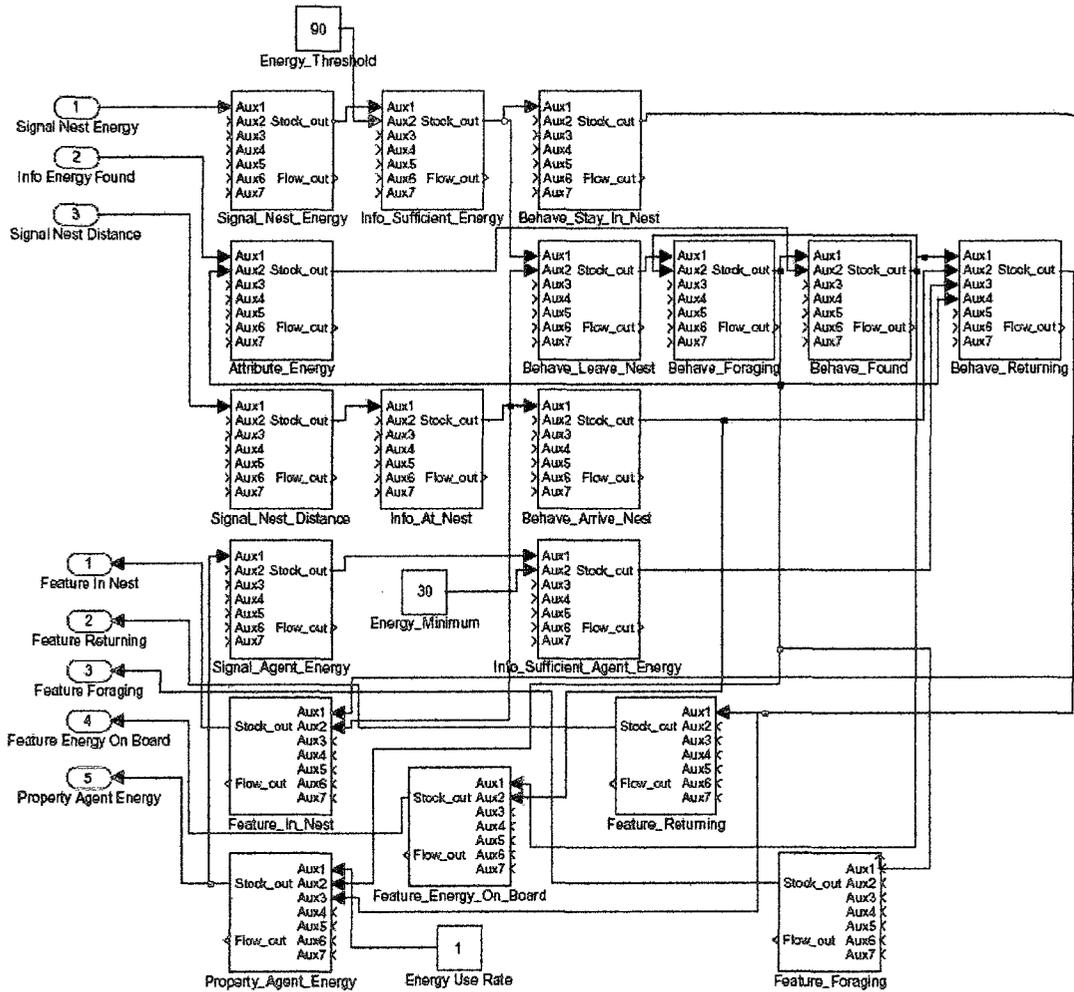


Figure 33 - Foraging Agent Block Internal Structure

Similar to the agent block above, the environment reaction block has been modularized and is shown in Figure 34 while the internal structure of this block is shown in Figure 35. The left hand side of this block provides the inputs from the agent properties, while the right hand side provides outputs to the environment properties.

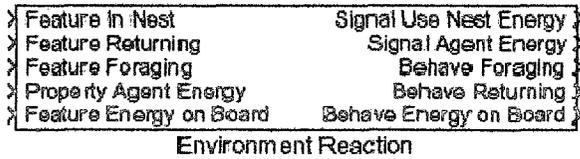


Figure 34 - Environmental Reaction Block

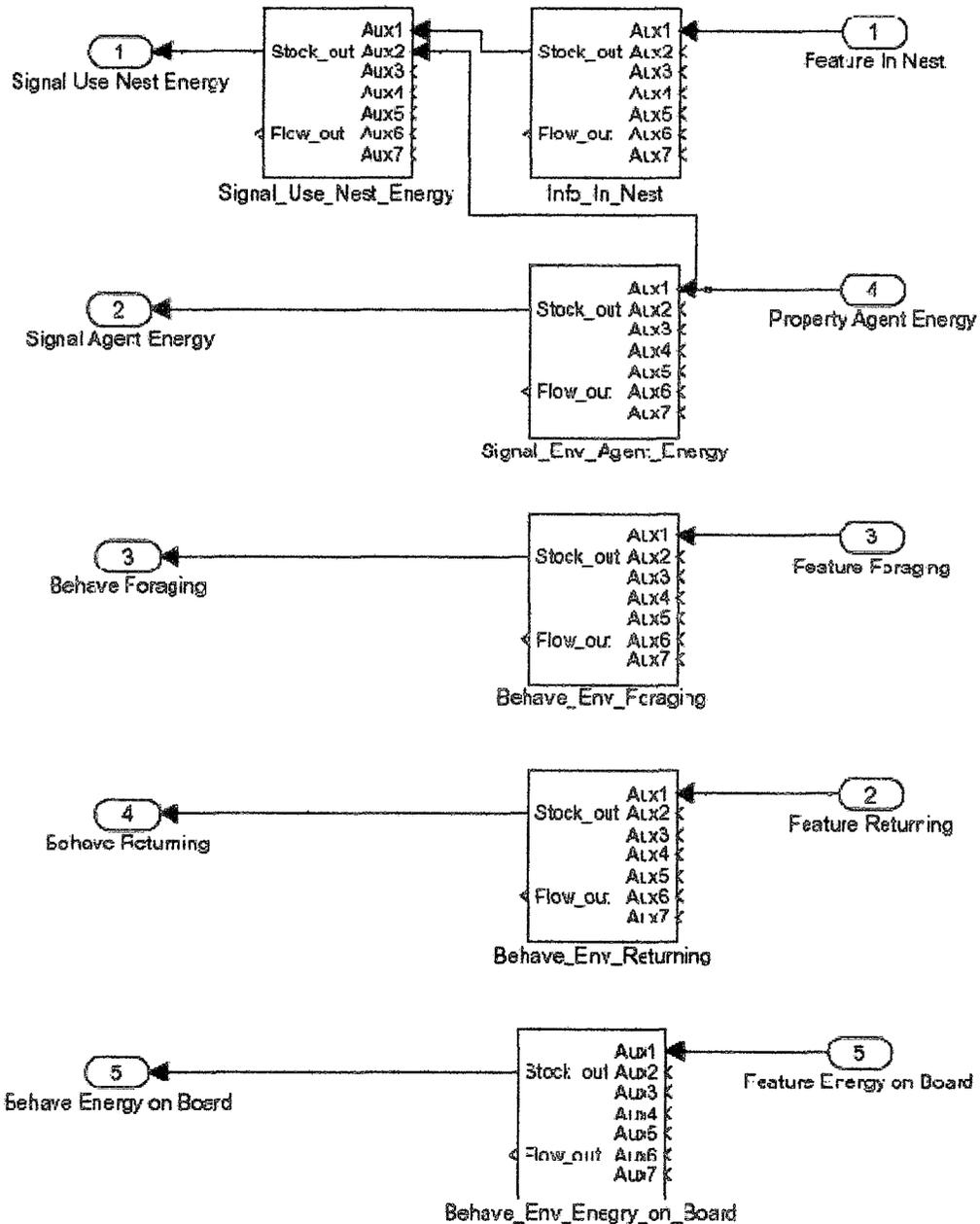


Figure 35 - Environmental Reaction Block Internal Structure

In addition to the agent and the environment reactions that should usually be modularized for a multi-agent model, it will often be helpful to modularize certain aspects of the environment properties. This is generally used when the environment properties section is being used to model a relationship between an individual agent and the environment, such as the presence of an energy module at the location of a particular agent or the distance of a particular agent from the nest. While these values are conveniently modeled as a part of the environment properties, they are clearly different for each agent.

The block for that portion of the environment properties that has been modularized is shown in Figure 36, while the internal structure of that block is shown in Figure 37. The left hand side of this block accepts inputs from the environment reaction blocks as described above, while the right hand side feeds the input section (*i.e.*, the agent reactions section) of the agent block as described above.

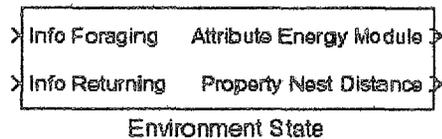


Figure 36 - Environmental Properties Block

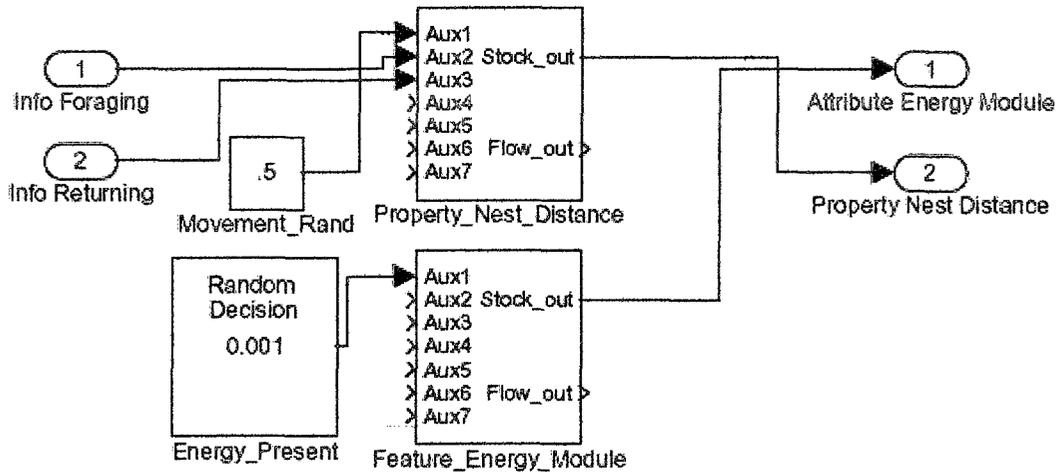


Figure 37 - Environmental Properties Internal Structure

The complete Multi-Agent Foraging model is shown below in Figure 38. The Mux and Demux blocks are used to reduce the number of traces that must be routed. Functionally Simulink uses a Mux block to convert a set of individual numbers into a vector or numbers of the same type with the order determined by the order of the graphical connection. The Demux block is the reverse. Note that despite the modularization, the basic cycle requirements of the framework are still preserved.

For simplicity this model uses only three agents, although the results shown in the next chapter on predictive modeling are obtained using a five-agent model that is implemented in a different tool. Code for the model shown above can be found in Appendix B.2 – Multi-Agent Foraging Model.

Modularization is usually best accomplished by creating a model of the desired complexity with only a single agent of any particular type, then replicating that agent an appropriate number of times. Any additional interactions between the various agents can then be added.

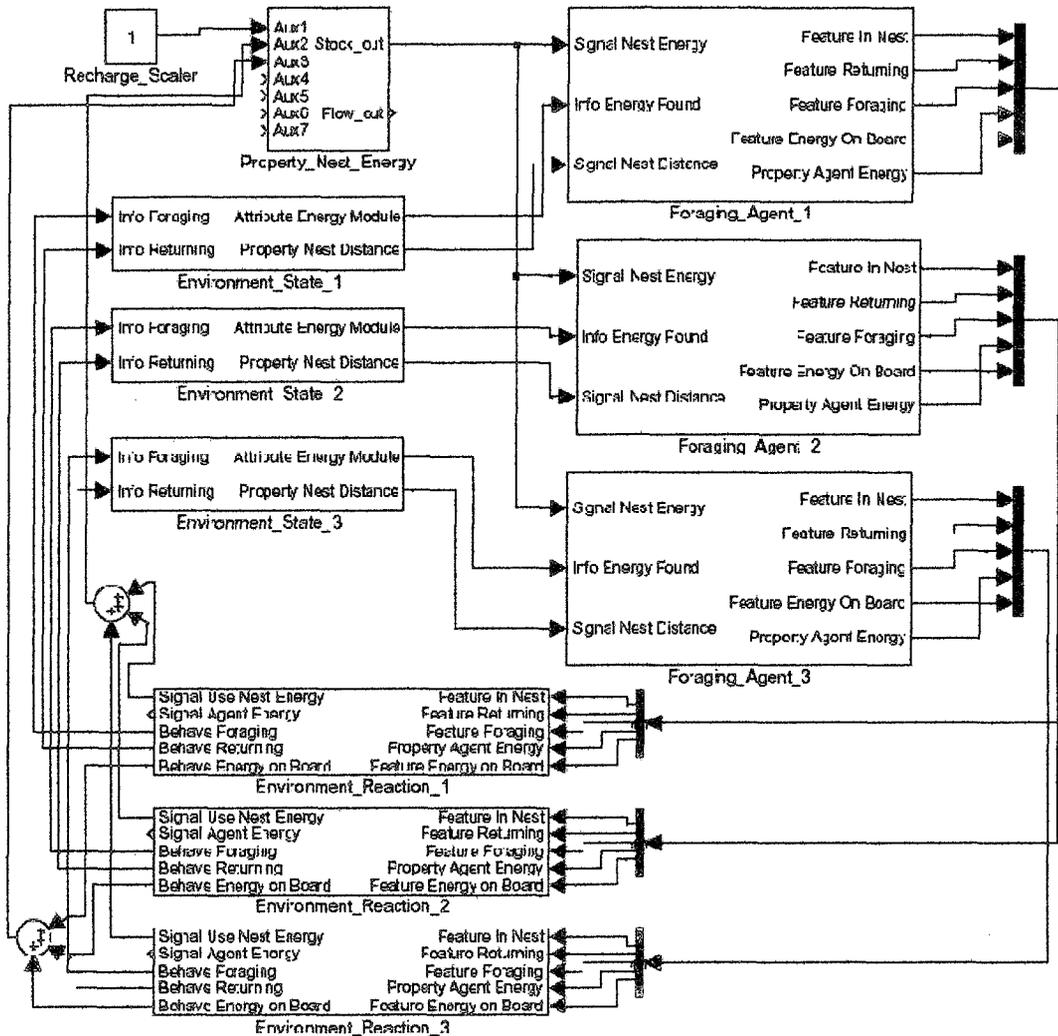


Figure 38 - Multi-Agent Foraging Model

4.5.4 Measuring Task Accomplishment

While defining the problem a list of five discrete tasks was developed:

1. SENSE THE NEST ENERGY LEVEL AND FORAGE WHEN NEEDED
2. FORAGE FOR AND LOCATE ENERGY MODULES
3. RETURN THE ENERGY MODULES TO THE NEST
4. MAINTAIN THE ENERGY LEVEL OF INDIVIDUAL AGENTS
5. MAINTAIN THE ENERGY LEVEL OF THE NEST

In measuring task accomplishment, one must now examine the model and find places to answer the questions above. Question 1 can be answered by looking at the

behave forage block of each agent as a function of the property nest energy block. Question 2 can be answered by observing the behave energy found block for each agent. Question 3 can be answered by looking for positive edges on the property nest energy block. Question 4, can be measured by observing the property agent energy block for each agent. Question 5 can be answered by monitoring the property nest energy block. In most cases, there are a number of other ways to accomplish the same result. It is left to the designer's discretion to select adequate monitoring methods given the relative importance of constraints and features.

4.5.5 Improvements and Additions

There are many opportunities for improvement and more detailed modeling in this system. In particular, the search and detection modality for the agents is not well modeled.

4.6 Urban Search and Rescue Victim Detection Model

The RoboCup Urban Search and Rescue (USAR) competition is an event within the international Robocup competition [33]. This competition and the numerous subcompetitions within it are intended to advance the state of robotics in a number of different fields to near human abilities by 2050. Specifically the RoboCup USAR competition has the stated aim:

“When disaster happens, minimize risk to search and rescue personnel, while increasing victim survival rates, by fielding teams of collaborative robots which can:

- *Autonomously negotiate compromised and collapsed structures*
- *Find victims and ascertain their conditions*
- *Produce practical maps of their locations*
- *Deliver sustenance and communications*

- *Identify hazards*
- *Emplace sensors (acoustic, thermal, hazmat, seismic, etc, ...)*
- *Provide* *structural* *shoring*

...allowing human rescuers to quickly locate and extract victims.” [33]

Within the competition, victims are simulated using five life signs: form, heat, movement, sound, and carbon dioxide emissions. Within the rules [34] of the competition, three life signs are defined as constituting a victim. Additionally, a determination of the state of the victim (fully conscious, semi-conscious, or unconscious) may be made based on the magnitude of the life sign. For example, a large movement such as an arm motion is to be categorized as a fully conscious victim, while smaller movements such as a twitching finger or a gently moving head should be interpreted as semi-conscious. A simulated victim with no movement at all, but that still satisfies the criteria of three life signs is assumed to be unconscious.

The arena of the competition varies significantly from smooth, relatively featureless terrain, to significantly obstructed and unstable terrain. Typical examples of terrain and simulated victims can be seen in Figure 39 and Figure 40. In addition, various false life signs are placed within the arena either through intention or circumstance. For example, these competitions are public, and often well attended events (see Figure 41). It is unavoidable that human life signs can be picked up from outside the arena. In addition, simulated items representing confounded environmental features such as a mostly-smothered fire are placed in the arena to generate heat signatures or other types of false life signs.

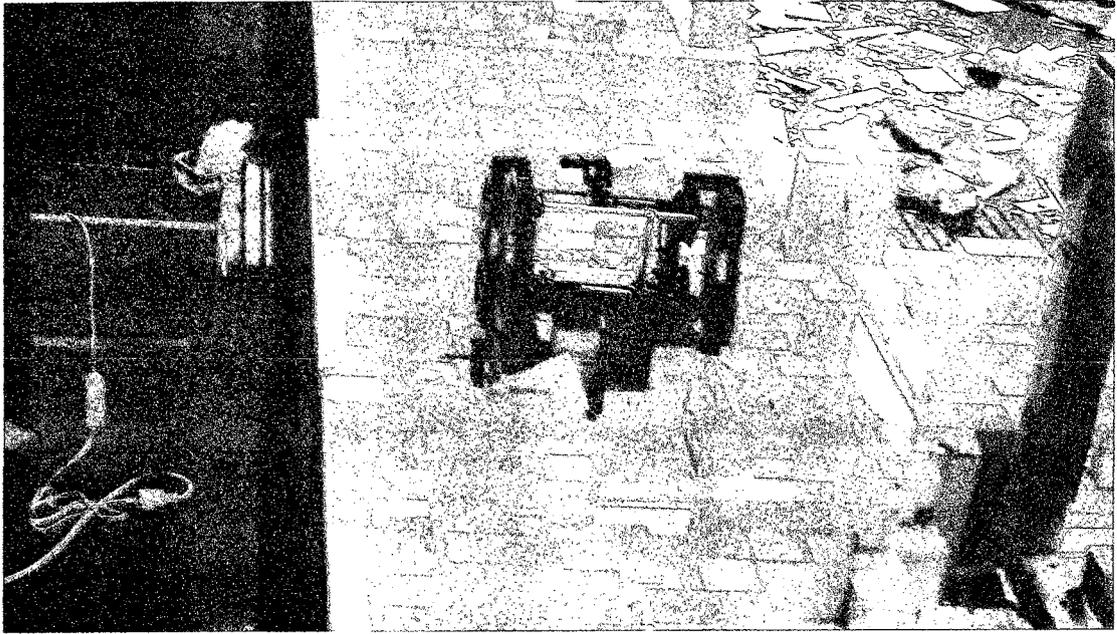


Figure 39 - Typical RoboCup Terrain and a Typical Victim with the Good Samaritan in front of it

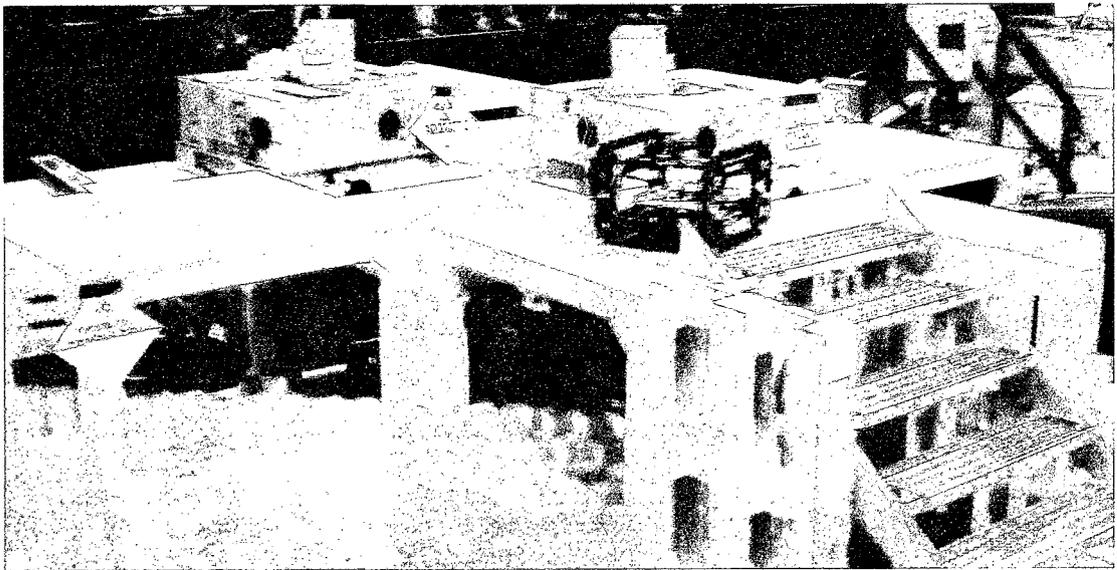


Figure 40 - Additional Typical RoboCup Terrain with the Good Samaritan

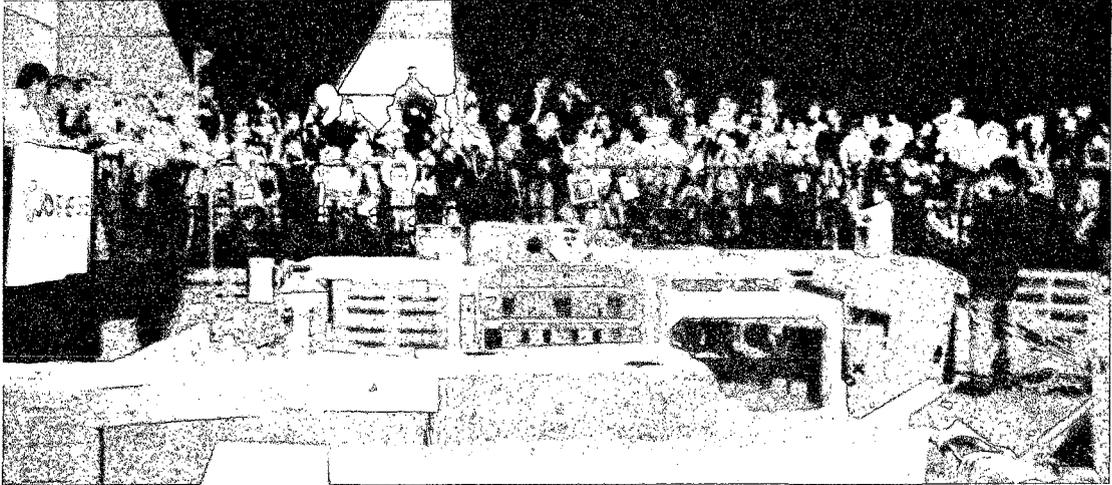


Figure 41 - Crowd at the RoboCup Competition in 2006

The Good Samaritan [35] pictured in Figure 39 and Figure 40 was a Colorado State University robot designed to compete in the RoboCup Rescue competition. The following models are highly abstracted models of this design problem and focus on the victim detection and classification aspects of this system. This model is used principally to demonstrate perceptive task modeling.

4.6.1 Defining the Problem

In this case, the problem has previously been well defined as described by the competition organizers. However, it is still up to the designer to break this problem statement into more specific tasks. The list below is incomplete and is focused predominantly on the victim detection aspect of the problem, as is the model itself.

Indentation denotes a child relationship to the parent task.

```
MOVE ABOUT THE ENVIRONMENT AND SEARCH FOR VICTIMS
DETECT VICTIM LIFE SIGNS
    DETECT HEAT
    DETECT MOVEMENT
    DETECT SOUND
    DETECT CARBON DIOXIDE
    DETECT FORM
```

DETERMINE VICTIM STATE
DISTINGUISH BETWEEN LARGE AND SMALL MOVEMENTS
DISTINGUISH BETWEEN LOUD AND QUIET NOISES

4.6.2 GSVD Model

The model for the Good Samaritan victim detection is shown below in Figure 42. The significant difference between this model and the models presented previously is the presence of perceptual tasks. This is discussed further in the section on measuring task accomplishment. Beyond the illustration of a perceptual task, this model is intended principally to illustrate that it is possible to model significantly complex tasks with relatively simple models. Full development of this model is not shown, but additional details are available in Appendix B.3 – GSVD Model.

Despite the relative complexity of the problem and the simplicity of the model, this model still provides both qualitative and quantitative insight that is useful to a designer. Qualitatively, the interactions that lead to task accomplishment are made explicit, and further exploration based design cycles can result in a more sophisticated model that is more accurate. Quantitatively, predictive modeling can yield additional insight. Please see Chapter 5 for more information.

DETECT MOVEMENT
DETECT SOUND
DETECT CARBON DIOXIDE
DETECT FORM
DETERMINE VICTIM STATE
DISTINGUISH BETWEEN LARGE AND SMALL MOVEMENTS
DISTINGUISH BETWEEN LOUD AND QUIET NOISES

This problem as stated contains the physical task of moving about the environment plus a number of perceptual tasks. The physical task is relatively poorly modeled as it appears here, and as such determination of task accomplishment is limited to recording the value of the “property searched terrain” block.

Perceptual task accomplishment is relatively explicitly modeled in the manner described in Chapter Three by comparing the agent state (*i.e.*, what the agent believes about the world) with the environment state (*i.e.*, reality within the abstracted world). For each of the life signs, as well as the magnitude of the life signs, there is a portion of the agent state that defines what the agent believes to be true about the environment. This can be directly compared with the respective blocks in the environment state to measure the accuracy of the perception of the agent.

For the higher level perceptual tasks of determining the presence of a victim and determining the state of the victim, the process is exactly the same, and once again explicit blocks exist for both the agent’s belief of the state of the environment and the actual state of the environment.

As with the other models, the conditions necessary for task accomplishment are made explicit within the descriptive model, but can only be measured within the predictive model. Measurements of task accomplishment will be discussed in additional detail in Chapter Five.

4.6.4 Additions and Improvements

It is recognized that this model represents a very simplified abstraction of the actual RoboCup USAR competition, let alone an actual USAR situation. In addition, the movement inherent in the model is not particularly representative of the Good Samaritan itself. However, this model can still be used to understand the subproblems and, as will be shown in Chapter Five, even draw some quantitative conclusions.

Many improvements to the model can be added as additional exploration based design iterations. As mentioned, the current representation of the terrain is very limited. Other significant areas for improvement include better representation and quantification of life signs, and explicit representation and modeling of the search process (to allow for accidental and intentional repeated search and the possibility of getting lost or disoriented as frequently happens even with human operators let alone under autonomous control [36]).

4.7 Prototyping

The exploration based design process makes explicit the need for prototyping. Interaction space modeling can and eventually should incorporate prototyping. As models become more sophisticated, the possibility of a mistake in the model or an incorrect assumption on the part of the modeler becomes more probable. Creating and building prototypes at appropriate points during the design/modeling process is important in creating a correct model. As with any prototyping process, it is possible to prototype the entire system, or one or more subsystems. Prototypes should be only as complex as necessary to validate the portion of the model in question. Eventually

the system as a whole will have to be prototyped as it will be necessary to check the entire model.

Chapter 5 – Predictive Modeling

Modeling efforts up to this point in this dissertation have been focused on descriptive modeling (*i.e.*, capturing the domain knowledge that the designer possesses and applying the design knowledge inherent in the modeling process). In descriptive modeling the primary task is to help the designer understand the qualitative interactions that govern system response. Even where continuous or more complex data types are employed, the purpose is predominantly to allow the designer to explicitly represent the data that will later be available rather than to explicitly represent any quantitative aspect of the system. By contrast, predictive modeling is intended to provide quantitative insight into the system behavior. This requires that the decode portion of the Rosen model be applied. If used successfully, predictive modeling will allow the designer to explore the real world design space within the abstract world and qualitatively refine R_n .

Predictive modeling is accomplished using the same models previously developed during the descriptive modeling phase by implementing these models in a numerical computing language (*e.g.*, Simulink) and iteratively simulating. Over a large number of iterations, the ability of the system to accomplish specific tasks can be correlated to system variables such as sensor accuracy, environmental parameters, physical characteristics of the agent, different control strategies, or other aspects of the model that can be changed. It is important to note that because much of a typical model, particularly the environment, relies heavily on stochastic elements, that single simulations have no meaning; trends must be looked at over a statistically significant

number of trials. A collection of ideas for rigorous implementation of this concept can be found in [37]

5.1 Developing Predictive Models

Predictive models are generally developed in the same fashion as descriptive models. The first step in developing a predictive model is the development of a good descriptive model. The major difference is in the rigor of the governing equations that must be developed. While descriptive models are best implemented with pseudo code or even verbal descriptions, predictive models require the generation of formal code to control the flows within the basic functional blocks (details on this code can be found in Appendix A – Implementation and Code for the Simulink Modeling Tools). It is important when writing this code to follow good debugging and incremental development practices. In general the predictive model should start with a basic functional model that should be adapted to predictive modeling and incrementally improved from there. An attempt to jump from a sophisticated descriptive model to a sophisticated predictive model is generally difficult.

5.2 Implementing a Simulation

As was mentioned in Chapter Two, the original predictive interaction space simulations were implemented in a system dynamics programming environment called PowerSim. Due to the limitations of this language new models have been developed in Simulink. This chapter relies interchangeably on simulation results from both new models and old models, but ties the measurement of task accomplishment explicitly to the new framework and to the design methodology

developed in Chapter Three and Chapter Four. Code and implementation of alternate models is given in Appendix C – PowerSim Code and B.1 – Muramador Model. Moving forward, it is expected that it will be advantageous to implement all simulations in Simulink; however some current limitations will have to be overcome. These are discussed below in the section on the limitations of the current Simulink implementation.

5.2.1 Simulink Implementation of Basic Functional Block

A Simulink block has been created for the basic functional block. This block, shown in Figure 43, contains seven inputs labeled “Aux1” through “Aux7” which are used to bring data into the block. Other blocks whose states need to be known are connected to these inputs in a standard data flow fashion.

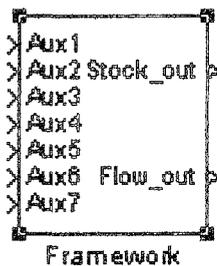


Figure 43 - Simulink Implementation of a Basic Functional Block

Figure 44 shows the internal implementation of the basic functional block in Simulink. Here the seven inputs as well as a feedback from the current value of the stock are combined into a single data stream. This data stream is sent into an m-file block which functions as a flow. Double clicking on the m-file block brings up a dialog box as shown in Figure 45 that allows the user to enter the file name. This file

written by the designer takes the inputs and reduces them to a single output value that represents the flow into or out of the stock.

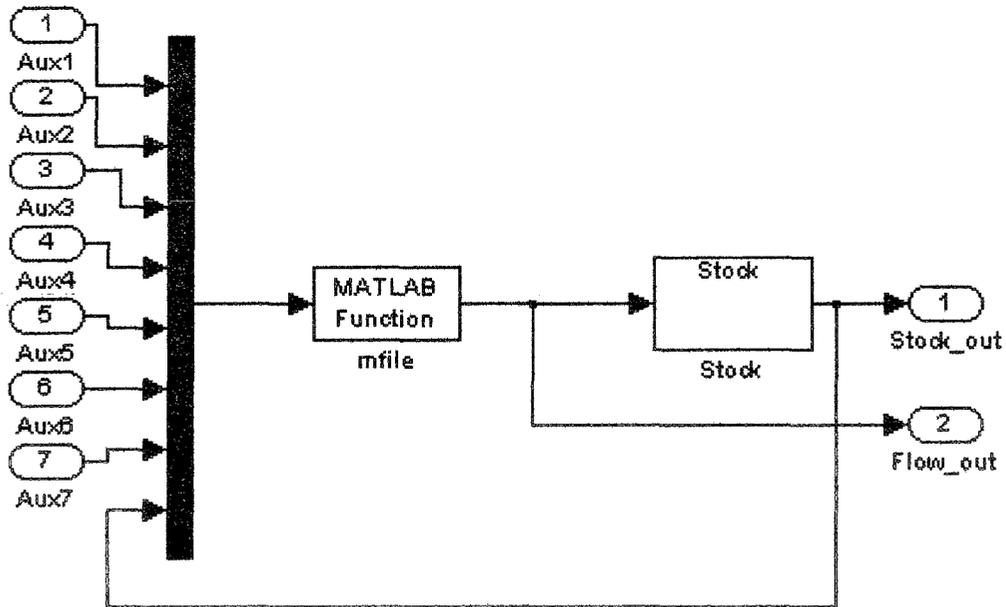


Figure 44 - Simulink Implementation of a Basic Functional Block

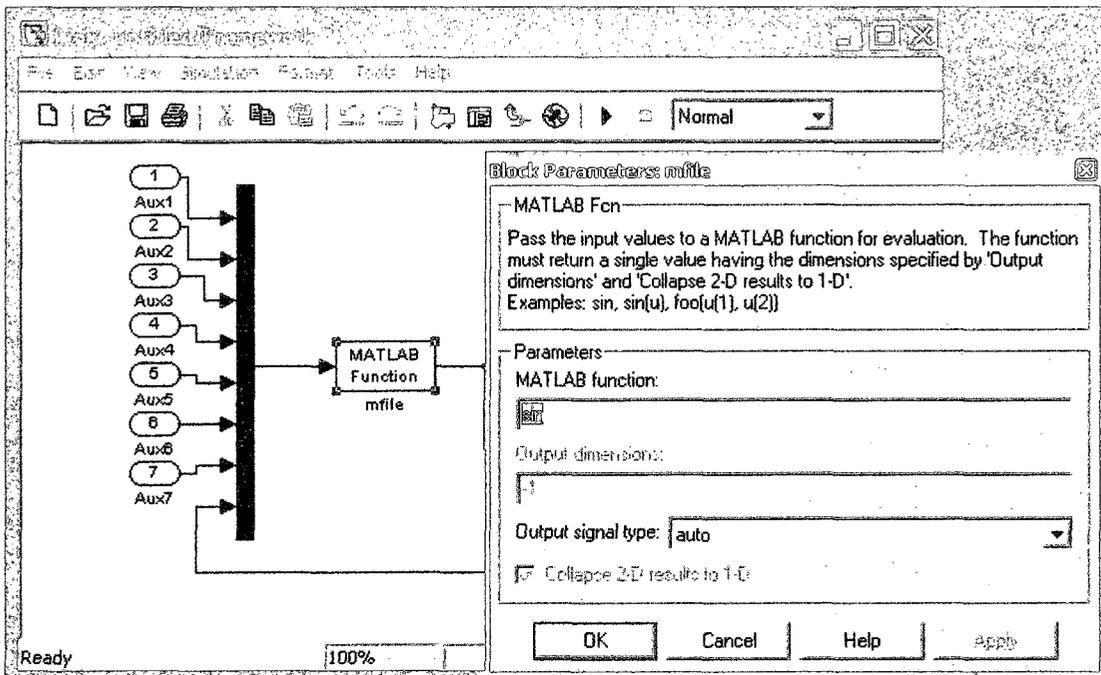


Figure 45 - Dialog Box to Set the M-file

The stock, which stores the value of the functional block and represents the actual state or reaction, is implemented as a custom S function [38]. The code for the S-function can be found in A.1 – Simulink Implementation of a Stock. The initial value of the stock must be set for each basic functional block. Double clicking on a stock will open a dialog box, shown in Figure 47, where the initial value can be set.

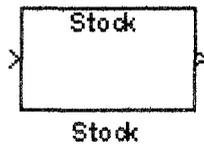


Figure 46 - Simulink Stock

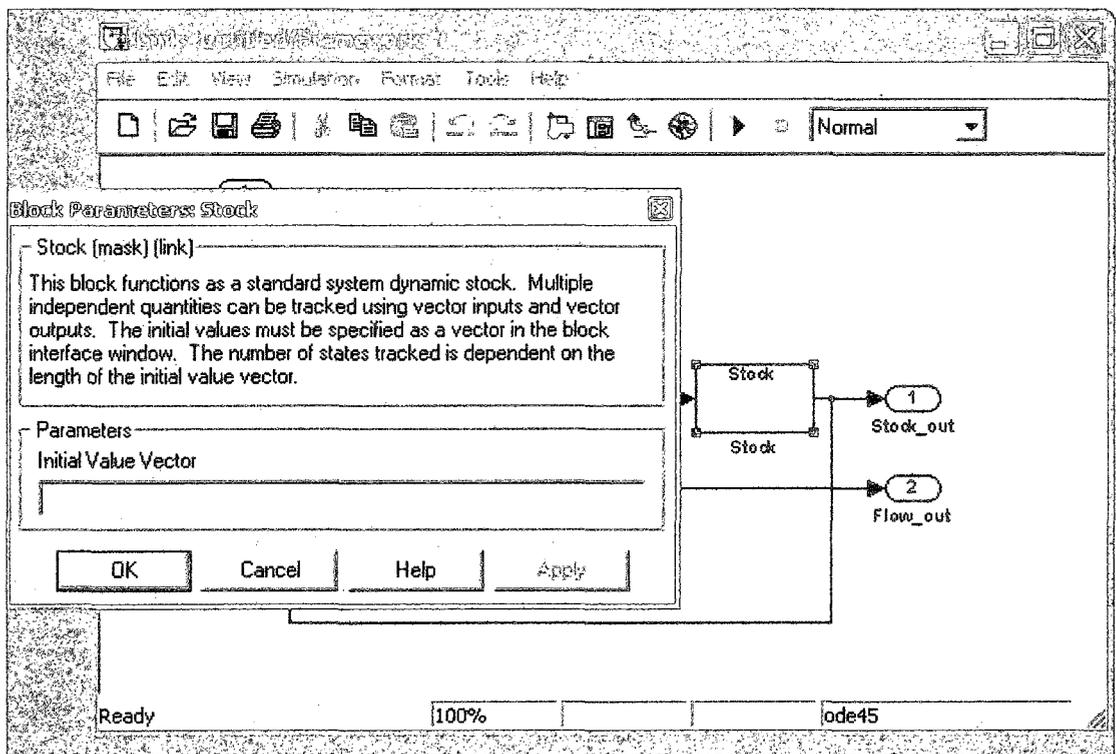


Figure 47 - Dialog Box to Input the Initial Value of the Stock

5.2.2 Simulink Implementation of Other Blocks

There are a number of additional blocks that are used for simulation of uncertainty and stochastic properties of the environment or agent. These are discussed generally in Chapter Four. The most common auxiliary block is the random decision maker shown in Figure 48. This block uses a random number generator in comparison to a threshold value entered by the user as shown in Figure 49. The output will be true (equal to one) at the specified percentage of time steps. In particular, this can be used to represent random events in the agent or environment, for example, the presence of a new wall in the Muramador model. Additional details on this block can be found in Appendix B –



Figure 48 - Random Decision Making Block

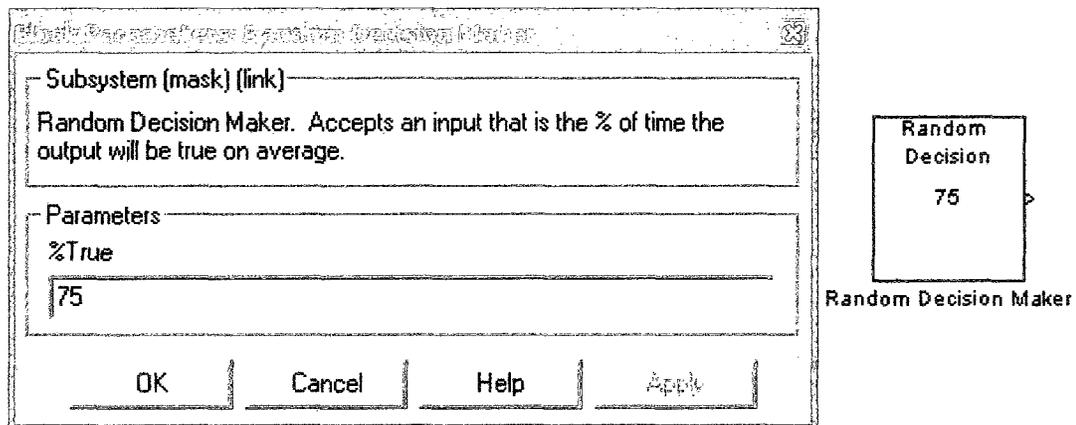


Figure 49 - Random Decision Block Dialog Box

The uniform random number (Figure 50) and the discrete uniform random number (Figure 51) blocks are also used frequently in modeling uncertainty and decisions. In particular, the discrete uniform random number generation block can be used to make discrete decisions in the same way the random decision maker block is used to make binary decisions. More details on these blocks can be found in A.4 – Random Decision Making Block Implementation.

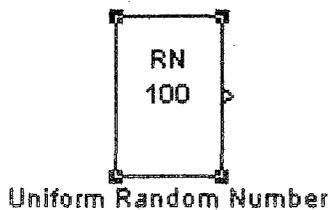


Figure 50 - Random Number Generation Block



Figure 51 - Discrete Uniform Random Number Generator

A number of standard blocks for generating noise and other stochastic signals are available within Simulink. It is beyond the scope of this dissertation to attempt a full introduction to the capabilities of Simulink, but additional information is available in [39].

5.3 Measuring and Interpreting Results

Assessment of task accomplishment is discussed extensively in Chapter Four; however, as has been previously mentioned, predictive modeling and numeric

simulation opens the possibility of quantitative measurement of task accomplishment. This can include either measuring the frequency of task accomplishment as defined by some binary criteria, or measuring the quality of task accomplishment (*e.g.*, the average deviation of the Muramador from the set point distance).

In addition to measuring task accomplishment under a fixed set of conditions, it is also possible to parameterize one or more of the system variables and observe the frequency or quality of task accomplishment that results. Such parameterizations can provide valuable insights into system requirements R_i and can subsequently lead to refinements of the model resulting in additional iterations of the exploration based design process and the ability to further quantitatively model task accomplishment.

5.4 Muramador Simulations

As presented in Chapter Four, the tasks for the Muramador are:

- FIND WALLS TO FOLLOW
- REMAIN AT THE SET POINT DISTANCE FROM THE WALL

As both of these are physical rather than perceptual tasks, task accomplishment is measured by observation of the environment state relative to some desired state. In Chapter Four, it was discussed that the first task can be measured by observing the “feature wall” block in the model. Typical results from this are shown in Figure 52. Alternatively one could measure the accumulated time that the Muramador spends near a wall as shown in Figure 53.

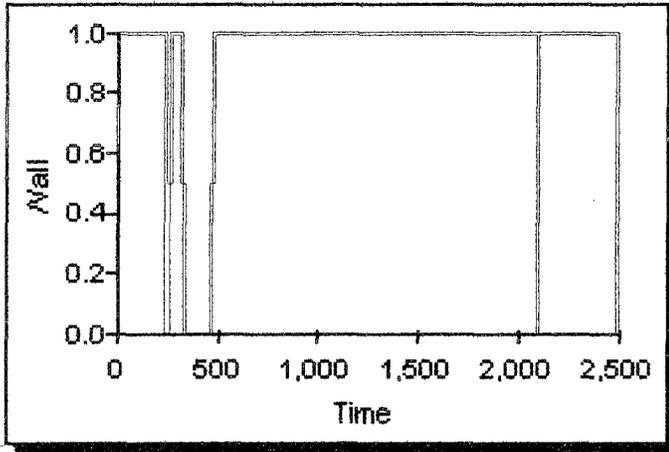


Figure 52 - Muramador Model of the Presence of a Wall (Time Units are Arbitrary)

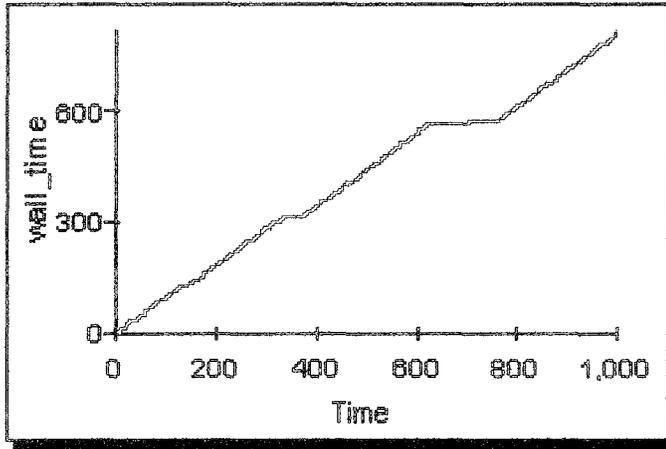


Figure 53 - Muramador Cumulative Wall Time (Units are Arbitrary but Consistent)

The second task, that of maintaining a particular distance from the wall, is more accurately modeled here and hence more realistic results are available. The distance of the robot from the set point can be measured directly, for example, as shown in Figure 54. However, as discussed above, due to the stochastic nature of the models and the consequent need for multiple runs and average values, this is valid as qualitative information (*i.e.*, the shape) only.

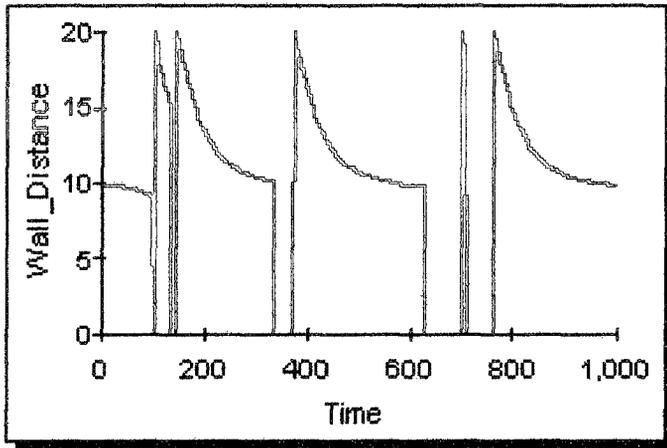


Figure 54 - Muramador Instantaneous Wall Distance (Units are Arbitrary)

In order to obtain quantitative information, multiple runs were conducted and averaged with variation in the control constant parameter. Results from this are shown in Figure 55. From this plot it can be seen that at very low values of the control constant, the average deviation is nearly equal to the difference between the set point and the maximum range of the sensor, while at very low values, the average deviation drops to essentially zero. Although not shown on this plot, at just a slightly higher value, the system becomes unstable and exponentially greater distances are reached. This is consistent with standard control theory, and in fact for this relatively simple system that could have been predicted without interaction space modeling. However, instabilities will be revealed even in very complex systems with significant uncertainty and randomness, albeit perhaps less clearly than shown here.

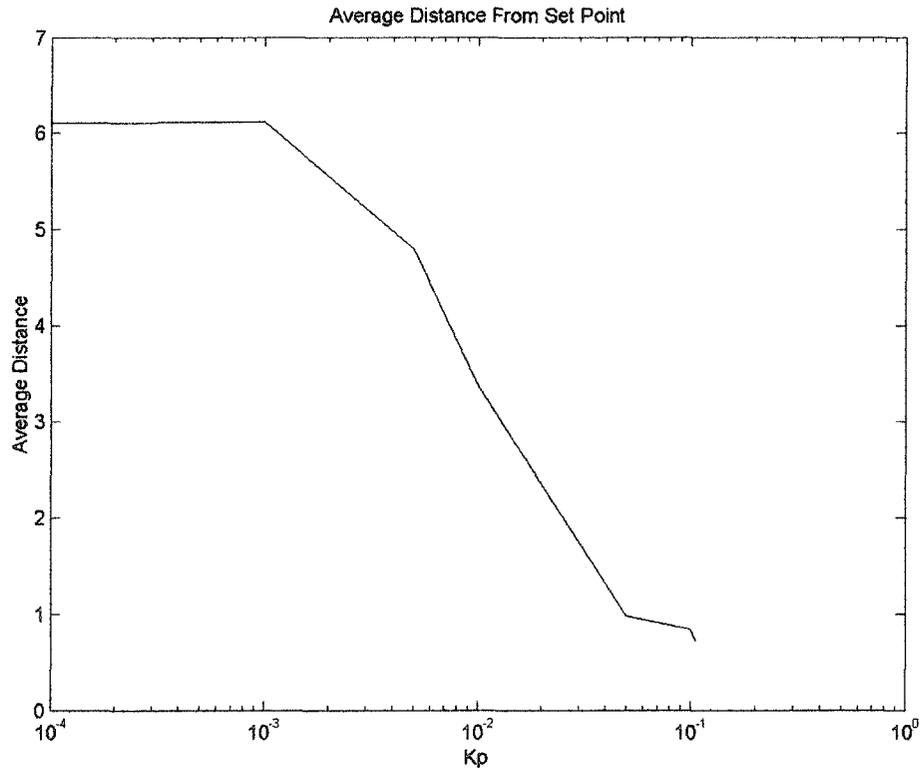


Figure 55 - Average Distance from the Set Point

5.5 Foraging Simulations

As was discussed in Chapter 4, the tasks to be accomplished by the system are:

1. SENSE THE NEST ENERGY LEVEL AND FORAGE WHEN NEEDED
2. FORAGE FOR AND LOCATE ENERGY MODULES
3. RETURN THE ENERGY MODULES TO THE NEST
4. MAINTAIN THE ENERGY LEVEL OF INDIVIDUAL AGENTS
5. MAINTAIN THE ENERGY LEVEL OF THE NEST

Task one can be observed by watching the behave foraging block of an individual agent. This can be seen in Figure 56. Task two can be measured by observation of when an agent does or does not have an object. An example of this can be seen in Figure 57. This plot can also be used to see the accomplishment of Task three as a

negative edge represents dropping off an energy module. Task four is observed by monitoring the agent energy block in the agent state. An example is shown in Figure 58. Task five is a straightforward look at the nest energy level block. An example is given in Figure 59

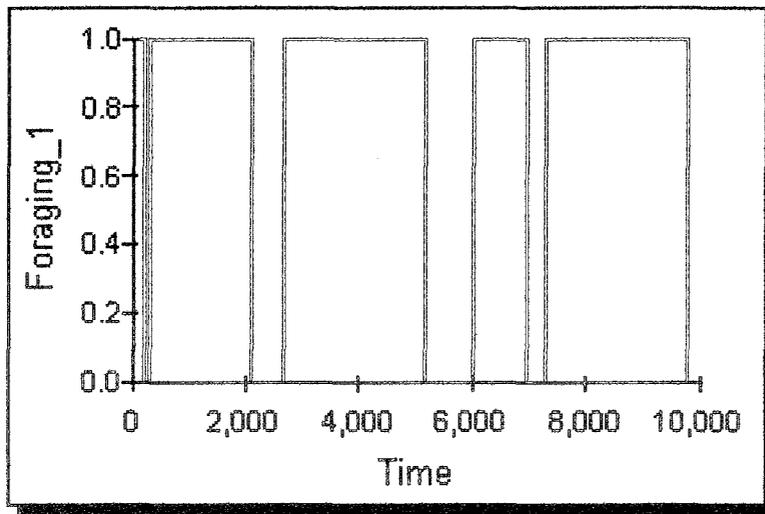


Figure 56 - Agent Foraging Output (Units are Arbitrary)

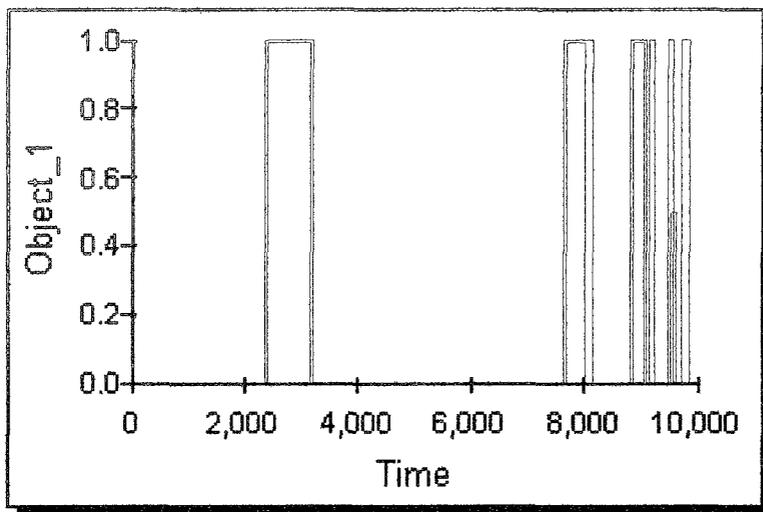


Figure 57 - Agent Object Found Output (Units are Arbitrary)

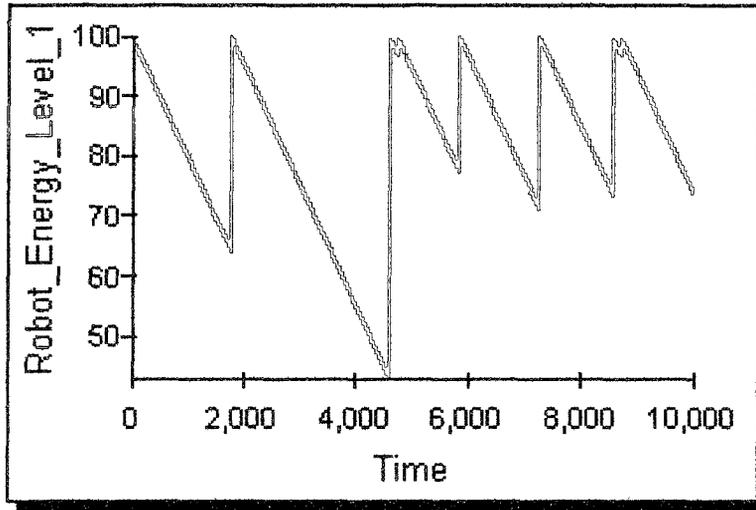


Figure 58 - Individual Agent Energy Level

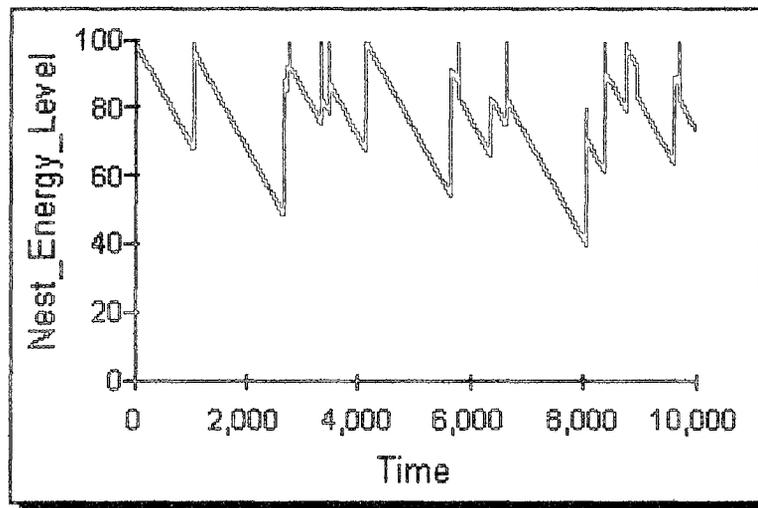


Figure 59 - Instantaneous Nest Energy Level for a Typical Power Sim Run

A more useful method of measuring task accomplishment is to repeat the simulation a statistically valid number of times and average the results. While this technique can predict the nest energy for a particular set of parameters, often to a designer the variation of task accomplishment due to the variation of more than one parameter is more interesting. This concept is illustrated in Figure 60, which

represents the simulation time that elapses before the nest energy drops to zero as a function of both the energy usage of an individual agent and the value of each packet of energy that is found and returned to the nest. Figure 61 shows the average nest energy level as a function of the same two parameters. In both cases the trend is what would be expected logically; however, here it is possible to quantify these interactions.

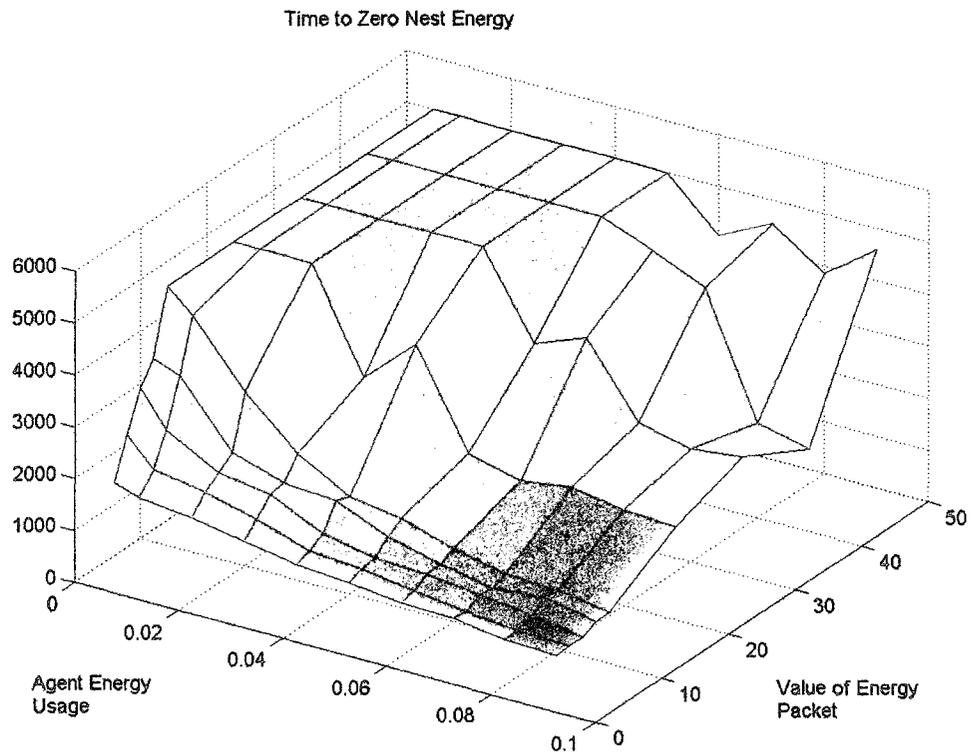


Figure 60 - Maximum Number of Times Steps (5000 possible) to Complete Nest Energy Loss (out of 20 runs)

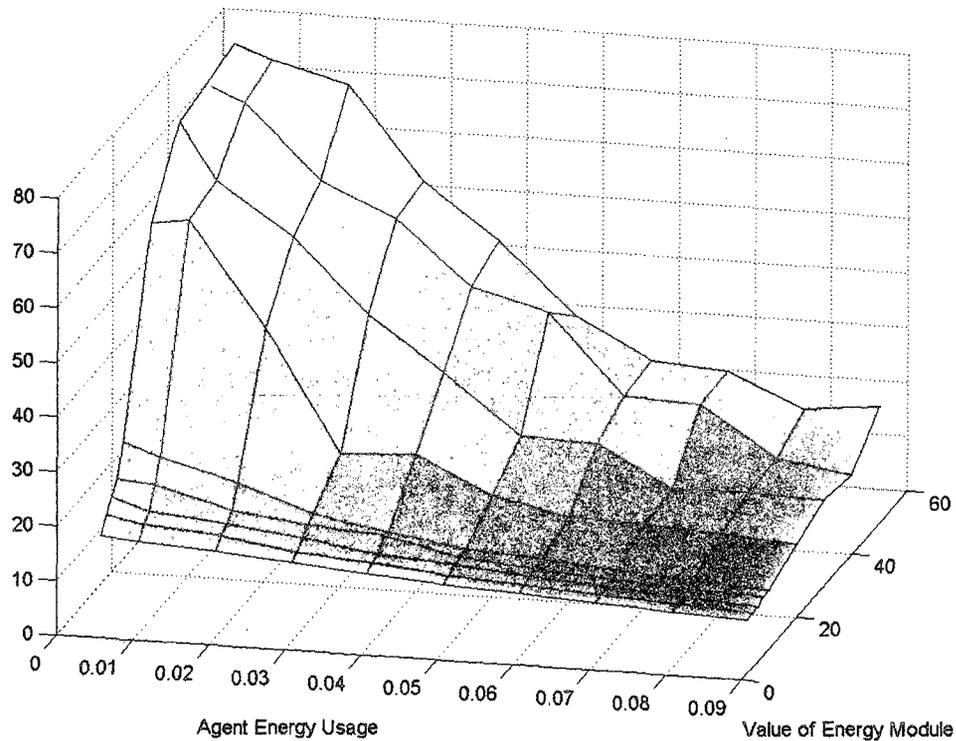


Figure 61 - Average Nest Energy as a Function of Agent Energy Usage and the Value of an Energy Module

5.6 Victim Detection Simulations

Detailed results of the victim detection simulations are not presented here, as little new work has been done on this topic for this dissertation. This is a result of the simulation environment limitation discussed above. Additional information is available in [2]. Figure 62 shows a typical example of information that can be obtained from analysis of the simulation results for the Good Samaritan Victim Detection Model. The results below are included as an example of a perceptual task as discussed in Chapter 4.

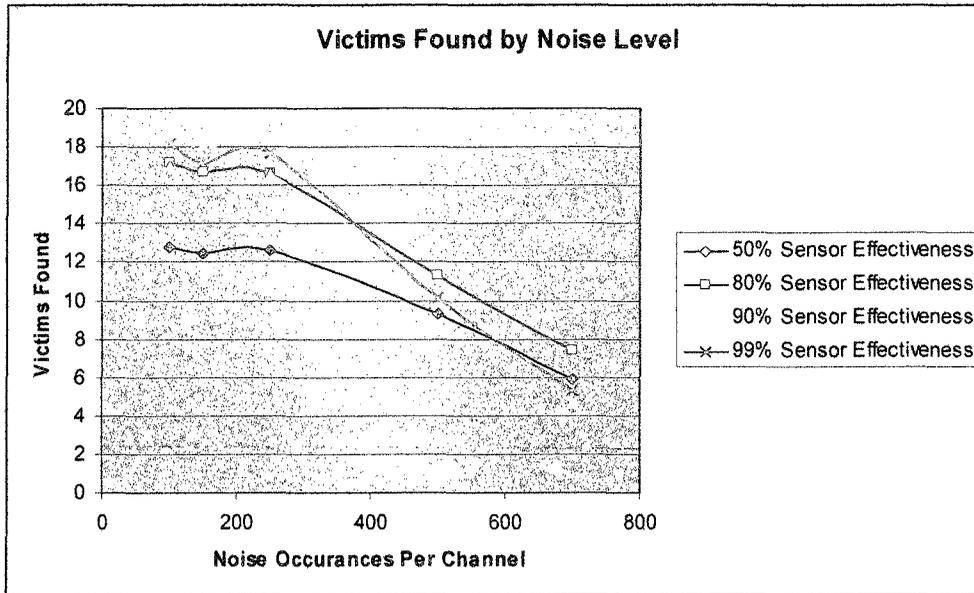


Figure 62 - Average Number of Victims Found Based on Environmental Noise and Sensor Effectiveness

5.7 Exploration Based Design with Predictive Modeling

The predictive modeling process fits into the exploration based design within the prototyping loop (see Figure 63.) The predictive modeling as with other simulations will reduce the prototyping needs in quantifying performance. As discussed in the next section it is still necessary to conduct prototyping activities; however, the emphasis can shift (at least early in the design phase) to validating assumptions and specific portions of the model rather than validating overall system response. This allows, in general, for smaller scale more contained experiments to be conducted at a lower level of sophistication than a full system model. In general this should reduce cost, cycle time or both.

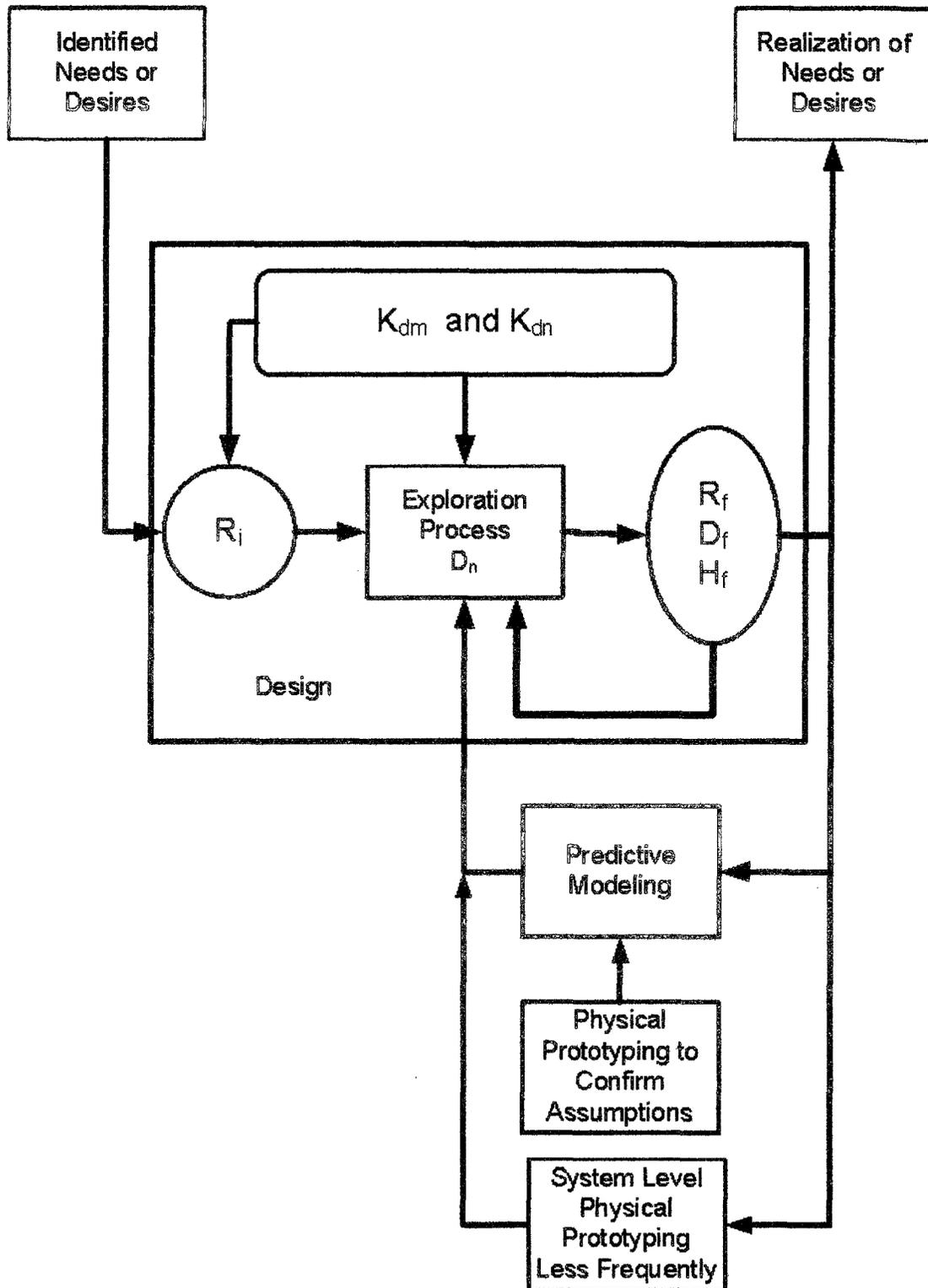


Figure 63 - Exploration Based Design with Predictive Modeling

5.8 Discussion on Predictive Modeling

Several topics related to predictive modeling bear further reflection but are not specifically applicable to the models above. Each of these topics is discussed briefly below and should be considered by the designer in the context of a specific model when implementing these techniques.

5.8.1 Multi-Variate Parameterized Simulations

In the preceding three sections, results have been presented from both single simulations and from compiled averages of many trials. Additionally parameterized results have been shown where either one or two key variables in the system are presented over a plausible range of values for either one or two variables. It should be recognized that there is no theoretical limit to the number of values that can be parameterized, particularly with the ability to script large batches of Simulink executions. However, practical limits can arise due to computation requirements of an inherently combinatorial problem.

5.8.2 Grounding the Simulations

The reader may or may not have noticed that all of the results above are presented without units. The primary reason for this is that the numbers at this point are arbitrary. While they do have relative meaning, the system model is not explicitly grounded in well-defined units; nor have the subcomponents of the simulation been validated by physical means. As with any model it is only valid after and to the degree that it has been tested. Both of these topics will be discussed in the next chapter.

5.8.3 Limitations of Simulink Simulation Environment

Due to limitations in the previous programming language that was used to implement interaction space simulations, a new set of tools was developed in Simulink for predictive interaction space models. Unfortunately, it has so far not been possible to get Simulink to update the elements in the model in an appropriate manner. Simulink solves the blocks based on the value of the predecessors during the previous time step for the entire model. Thus it takes information up to N time steps for propagate around the cycles in the model where N is the length of the longest cycle in the model.

Since much of predictive interaction space modeling relies on the comparison of information that has made a complete cycle to information at the starting point (particularly when measuring task accomplishment), this creates unmanageable models for any significantly complex system. The Muramador model is able to be implemented in Simulink because there are not time critical comparisons within the model, but more complex models such as the GSVD and the multi-agent foraging model require that blocks be computed with present time step values for the predecessors where possible. Consequently predictive models have not been fully developed for the GSVD and the multi-agent foraging system within the new framework.

The GSVD and multi-agent foraging systems have both been previously implemented in another language called Powersim[39]. These models predate the framework that has been developed in this dissertation, but are the basis from which the framework was inductively derived. Results from simulations run on these

models are discussed above interchangeable with Simulink results to better illustrate the value of predictive interaction space modeling. Complete programs for these simulations can be found in Appendix C – PowerSim Code and a complete description of the development of these models is available in [2]

5.8.4 Comparison to Current Methods

The primary change in the predictive modeling process from previous work is the additional rigor of the basis descriptive model imposed by the framework and the design methodology described in Chapter Three and Chapter Four. This rigor is additionally useful that the formal definition of task accomplishment allows for more focused assessment of task accomplishment.

In addition, the introduction of the new simulation environment will remove significant limitation imposed by older systems, particularly related to multi-dimensional data and complex functions. In addition, the implementation of standard block types, particularly for stochastic elements, provides significantly more structure to the development of predictive models compared to previous methods.

Compared to the work presented in [2], the implementation and interpretation of simulation results has not changed dramatically. The introduction of multi-variate parameterizations and the explicit connection between the tasks in the problem statement are the primary new points.

5.8.5 Computational Complexity

As implemented, the predictive models will increase in computational complexity as a high order polynomial in N function where N is the number of

elements used in the model. This complexity further increases linearly with the number of time steps, and linearly with the number of trials. Although N is relatively small in the models shown, each basic modeling element requires a large number of computations and consequently computation of even a two-parameter design analysis takes significant time.

This problem can be somewhat alleviated by more efficient coding. In particular, the use of pass-through elements adds a substantial number of computations to any simulation. By relaxing the rules of the framework, insight is lost in the descriptive model, but efficiency can be gained in the predictive model. The development of a computationally simple pass-through element could alleviate this issue.

No attempt has been made to develop efficient code, even for repetitively used elements like stocks; however, this problem is inherent in this and most other simulation methods. As computers have become more powerful this problem has been somewhat alleviated and for this method, reasonable systems can be simulated within a few hours at most with current technology.

Chapter 6 – Conclusions and Future Work

This chapter outlines the contributions and conclusions of this dissertation. Future work that either must or may be carried out to develop interaction spaces and interaction space modeling from the infant stage in which they presently exist to proven design theories is also discussed.

6.1 Summary

The core value of this dissertation is the development of a design abstraction, a formalism to go with it, and a design methodology (exploration based design) for the modeling, description, and design of autonomous robots. Interaction space modeling represents an abstraction that is capable of using most of the other design abstractions for robotics (*e.g.*, subsumption, voting, schema, etc.) but still provides the designer with a framework in which to creatively explore the design space. Additionally, the melding of the interaction space abstraction with the exploration based design methodology provides a formalism specific to autonomous robots. While this in no way relieves the designer of the need for creativity, solid technical skills, and both design and domain knowledge, it does provide a framework in which to work and decompose a problem. It also forces an explicit consideration of the interactions that drive task accomplishment, potentially leading the designer to solutions that were not previously considered.

Additionally, predictive modeling (*i.e.*, simulation) has been transformed from an *ad hoc* simulation to the rudiments of a design tool through the creation and implementation of standard blocks and the development of these blocks in Simulink.

6.2 Conclusions

Interaction spaces as a design abstraction and interaction space modeling as a design methodology continue to need significant development to be fully vetted for autonomous robot design; however, when used in conjunction with other standard engineering skills, they can provide a valuable way to gain insight into a system and arrive at a design that accounts for a systems-level view. By taking the agent and environment into equal consideration, and explicitly refining conditions for task accomplishment iteratively, emphasis remains on the system design rather than on technology or clever kludges.

Interactions space modeling has both the advantage and the limitation of freeing the designer from focus on geometry, sensors, actuators and structures and allowing a focus on good system level design. This allows for a creative exploration of the design space early in the design process and can assist the designer in finding solutions that would have been missed had the focus been on technology.

6.2.1 Interaction Spaces and Design

An interaction space as preciously introduced in [2] and further developed in this dissertation is the set of all possible interactions between the agent and the environment. Specific interactions lead to task accomplishment. The goal of a designer is to create a system that causes these interactions to occur.

One way of thinking about a design process is as an exploration. All possible solutions constitute the design space. Constraints and requirements are used iteratively to successively narrow the design space. The exploration based design

process, along with the interaction space specific process described in this dissertation, provide systematic guidance to the designer.

6.2.2 Descriptive Modeling

The nomenclature and methodology, particularly the quadrant abstraction and the cycle abstraction, presented in this dissertation can be used to develop descriptive models. A descriptive model captures and communicates the domain knowledge that the designer possesses. Specifically, this information is captured in a form that makes explicit the required interactions between the agent and the environment.

The process of developing this model also helps the designer to explore the solution space systematically and iteratively. In particular, the process of writing and modifying the pseudo code at each step provides a concrete means of forcing the designer to consider all of the information that plays a role in any given reaction and the reactions that affect any given state. Doing this via the process described in this dissertation and returning to a “complete” model after each iteration of the exploration based design process makes this task more approachable than a blank page approach and restricts the designer’s ability as well as unintentional propensity to mentally hand wave over any particular aspect of the design.

In addition, the benefits internal to a single designer, descriptive interaction space models also provide a way of communicating interactions and the abstraction of interactions. If interactions are critical in designing robotic systems, it is necessary to have a clear means of communicating one’s understanding of these interactions.

6.2.3 Predictive Modeling

Predictive interaction space modeling provides a means of quantifying the design requirements. In particular, parameterization of one or more design parameters can be varied across a feasible range and task accomplishment can be measured within the model. This provides a relatively rapid means of exploring the design space compared with physical prototyping. In addition to the design requirements, predictive modeling also changes the nature of needed prototypes early in the design process. Specifically, prototypes early in the design process are intended to confirm assumptions, abstractions, encoding, and decoding rather than full system prototypes.

6.3 Future Work

Suggestions for improvement of specific models have been presented throughout this dissertation in the same section as the model itself; however there are several suggestions for future work that span the full scope of Interaction Space Modeling. These include additional work with real robots, standardization and implementation of a better simulation language and a better graphical tool for representing descriptive models, and development of additional blocks or classes of blocks.

6.3.1 Real Robots

While the descriptive modeling process is relatively well grounded in that it is internally logical, the predictive modeling process needs significant additional work to verify both the process and individual models. In the general sense of the process,

no predictive modeling process can be fully vetted without quantitative comparison to real robots. While several models have been qualitatively compared to published real world results, this comparison has been retroactive in that models were developed that relate to existing systems. There has been no physical verification that either the descriptive or the predictive modeling process will yield information about systems that do not yet exist.

In addition to the general validation of the predictive modeling process, it will always be necessary to validate any specific predictive model with real world data. This can be done partially through validation of various blocks or subsystem models; however, the need for prototyping, testing and appropriate refinement of the model will always be necessary before considering a design to be finished.

6.3.2 Dealing with Units

In dealing with real robots, it will also be necessary to address the concern of units. At present, units are largely ignored; however, a fully quantified analysis of a system necessarily requires that units be attached to the system. It is recommended that the units be recorded explicitly for every signal (*i.e.*, every connection between two blocks), but that no specific attempt be made to carry units symbolically within a simulation language. Such units could be displayed as text above each connection.

6.3.3 Standard Simulation Language

In some manner, the limitation on simulation of perceptual tasks must be overcome. Simulink has been a useful simulation language, as has PowerSim; however, neither system is really compatible with the full framework described in this

dissertation. The primary limitation of Simulink (*i.e.*, the induced lag in the system that is relative to the path length) may be possible to overcome, possibly through the creation of a discrete delay function that would use old data based on path length. There are also a large number of computation engines built into Simulink, and it is possible that one of these will work as is. This could make Simulink a good choice. Powersim is very limited in its ability to compute complex functions, to use history, to store and represent data, and to make drop-in subsystems.

In the event that Simulink is not acceptable, it may become necessary to develop a modeling and simulation tool in a lower level language such as C or C++. In particular, the object-oriented nature of C++ would lend itself very well to implementation with this framework, in that each of the 6 basic types of blocks could be represented by a class with individual blocks representing objects.

6.3.4 Expansion of Standard Modules

Although the modules presented in this document are believed to represent a sufficient set to model the vast majority of situations, there are likely situations where these blocks will turn out to be insufficient. In addition, it is very likely that other types of blocks can be found that will improve modeling efficiency or clarity. It is likely that most such blocks would be specific to a particular domain.

References

1. R. Brooks, 1986: "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, April, pp. 14 – 23.
2. C. Kaiser, *Interaction Space Constructs and Modeling for Application in Robot Design*, MS Thesis, Colorado State University, May 2006.
3. C. Kaiser, M. Conboy, W. Troxell. "Interaction Spaces for Urban Search and Rescue Robots", *Proceedings of the 1st Joint Emergency Preparedness and Response/Robotics & Remote Systems Topical Meeting*, Salt Lake City, UT, February 11-16, 2006, pgs. 252-257.
4. R. Rosen, 1991: *Life Itself: A Comprehensive Inquiry Into the Nature, Origin and Fabrication of Life*, Columbia University Press, New York, New York.
5. J. Forrester, 1961: *Industrial Dynamics*, The M.I.T. Press Cambridge, Massachusetts.
6. J. Sterman, 2000: *Business Dynamics: Systems Thinking and Modeling for a Complex World*, McGraw-Hill Higher Education.
7. R. Brooks, 1986: "Achieving Artificial Intelligence Through Building Robots," M.I.T. Artificial Intelligence Laboratory, A.I. Memo 899.
8. J. H. Connell, 1990: *Minimalist Mobile Robotics*, Academic Press, Inc.
9. T. Smithers, W. Troxell: "Design is Intelligent Behaviour, but What's the Formalism" *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, v. 2, i. 4 pp. 89 – 98.

10. T. Smithers: "Design as Exploration: Puzzle-Making and Puzzle-Solving"
Presented at AI in Design 1992, Workshop on Search-based and Exploration-based models of Design, Engineering Design Research Center, CMU, Pittsburg, June 1992.
11. V. Braitenberg, 1986: *Vehicles: Experiments in Synthetic Psychology*, MIT Press, Boston, Massachusetts.
12. P. Maes, 1993: "Behavior Based Artificial Intelligence," *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pp.II-317 – II-323.
13. R. Arkin, 1987: "Motor Schema Based Navigation for a Mobile Robot: An approach to Programming by Behavior," *Proceedings of the IEEE conference on Robotics and Automation*, Raleigh, North Carolina USA pp. 264 – 271.
14. R. Arkin, 1989: "Motor Schema-Based Mobile Robot Navigation," *International Journal of Robotics Research*, v.8, n. 4 pp. 92 – 112.
15. U. Nehmzow, 2003: "Mobile Robotics: A Practical Approach," Springer Verlag.
16. P. Ford: *Description of a Robot, Environment and Task System Using the Theory of Affordances*, M.S. Thesis, Colorado State University, Department of Mechanical Engineering, February, 1996.
17. R. Brooks, 1991: "Intelligence Without Representation", *Artificial Intelligence Journal* v. 47, pp 139-160. Reprinted in *Cambrian Intelligence*, The MIT Press, Cambridge Massachusetts, 1999.

18. C. A. Petri, "Kommunikation mit Automaten," Bonn: Institute für Instrumentelle Mathematik 1962. English translation, "Communication with Automata," New York, Griffiss Air Force Base. Tech Rep RADC-TR-65-377 vol. 1, suppl. 1 1966.
19. M. Caccia, *et. Al.*, 2005: "Execution Control of Robotic Tasks: a Petri Net-Based Approach," *Control Engineering Practice*, v. 13 pp. 959 – 971.
20. J. Rosell, 2004: "Assembly and Task Planning Using Petri Nets: A Survey," *Proc. Instn Mech. Engrs V.* 218 part B.
21. L. Montano, *et. Al.*, 2000: "Using the Time Petri Net Formalism for Specification Validation and Code Generation in Robot Control Applications," *The International Journal of Robotics Research*, v. 19 n. 1 pp. 59 – 76.
22. W. Zhang, 1989: "Representation of Assembly and Automatic Robot Planning by Petri Net," *IEEE Transactions on Systems, Man, and Cybernetics* v. 19 n. 2.
23. R. Brooks, "From Earwigs to Humans", *Robotics and Autonomous Systems*, Vol. 20, Nos. 2–4, June 1997, pp. 291–304.
24. D. Ryan, *Robotic Simulation*, CRC Press, Boca Raton, Florida, USA, 1993.
25. M. Adams: *Sensor Modelling, Design and Data Processing for Autonomous Navigation*, World Scientific Publishing Co. Pte. Ltd, Singapore 912805, 1999.
26. R. Fikes, N. Nilsson, 1971: "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence* v.2 pp. 189-208.
27. ABB Robot Studio Web Page
<http://www.abb.com/product/seitp327/78fb236cae7e605dc1256f1e002a892c.a>
spx, accessed January 23, 2009.

28. RoboCup Rescue Virtual Robot Competition Web Page
http://www.robocuprescue.org/wiki/index.php?title=VRGeneral_Information
 accessed January 23, 2009.
29. M. Adams: *Sensor Modelling, Design and Data Processing for Autonomous Navigation*, World Scientific Publishing Co. Pte. Ltd, Singapore 912805, 1999.
30. W. J. Palm: *Modeling Analysis and Control of Dynamic Systems*, John Wiley and Sons, Inc, New York, 2000.
31. E. Bonabeau, *et. Al.*, 1998: "Fixed Response Thresholds and the Division of Labor in Insect Societies", *Bulletin of Mathematical Biology*, v.60 pp 753-807.
32. M. J. Krieger, J. B. Billeter, 2000: "The call of duty: Self-organized task allocation in a population of up to twelve mobile robots," *Robotics and Autonomous Systems* v.30 pp.65-84.
33. RoboCup Rescue Robot League Website
<http://www.isd.mel.nist.gov/projects/USAR/competitions.htm> accessed
 January 23, 2009.
34. RoboCup Rescu Robo League Rules <http://robotarenas.nist.gov/rules.htm>
 Accessed May 3, 2009.
35. M. Conboy, C. Kaiser, W. Troxell, "A Variable Geometry Tracked Robot for Urban Search and Rescue", *Proceedings of Sharing Solutions for Hazardous Environments*, Salt Lake, UT, February 2006.
36. R. Murphy, 2004: "Human-Robot Interaction in Rescue Robotics" *IEEE Transactions on Systems, Man and Cybernetics – Part C: Applications and Reviews*, v.34, May 2004.

37. G. Calariore, F. Dabbene: *Propabalistic and Randomized Methods for Design Under Unvertainty*, Springer-Verlag, London, 2006 .
38. “Writing S-functions”, Mathworks, inc. 1998.
39. Simulink User’s Manual,
<http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/index.html?access/helpdesk/help/toolbox/simulink/> accessed May 3, 2009.
40. Powersim Web Page <http://www.powersim.com/> accessed February, 2006.

Appendix A – Implementation and Code for the Simulink Modeling Tools

This appendix contains the code necessary to replicate the Simulink blocks used to run simulations. Code from earlier sections is used in later sections.

A.1 – Simulink Implementation of a Stock

For the purposes of this document, a stock is implemented by a Simulink S-function and represented by the block shown in Figure 64. This block is built by filling in the S-function dialog box as shown in Figure 65 and by implementing code for “Stock.m” as shown below.

When used for predictive modeling, the initial value of the stock can be set by the user by setting the dialog box as shown in Figure 66. This block can be used to track multiple values at once, in which case the value of “Initial Value Vector” can be set as a vector using standard vector notation for Matlab. In this case, the input and output will have vector values as well, with the output being the same dimension as the input. This must also match the dimension of the initial value vector.

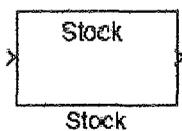


Figure 64 – Stock

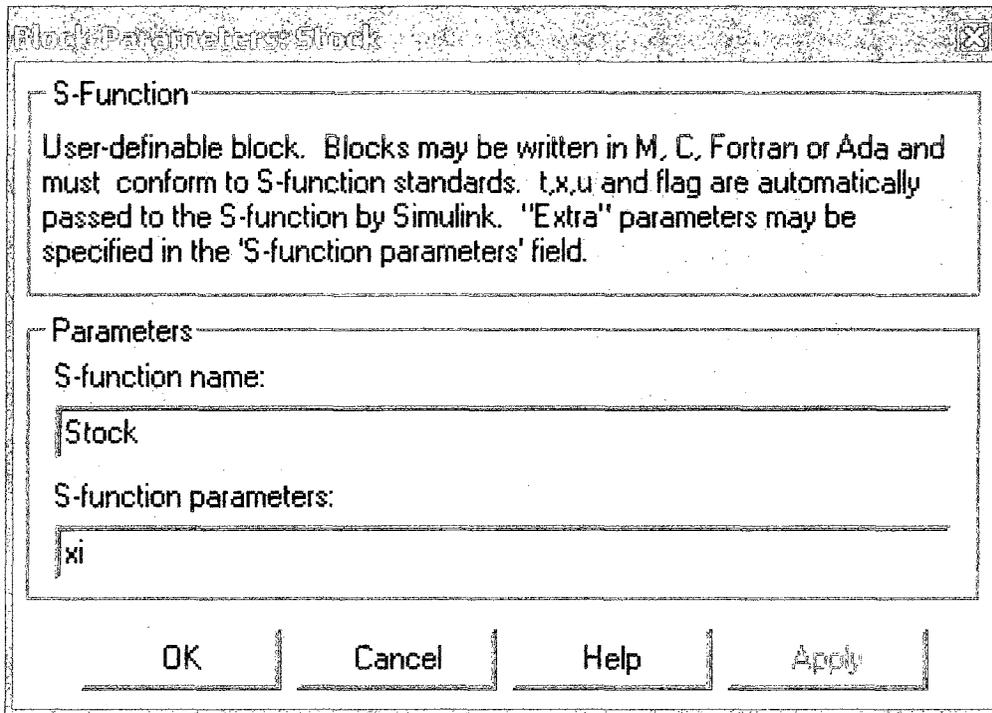


Figure 65 - S-Function Dialog Box for a Stock

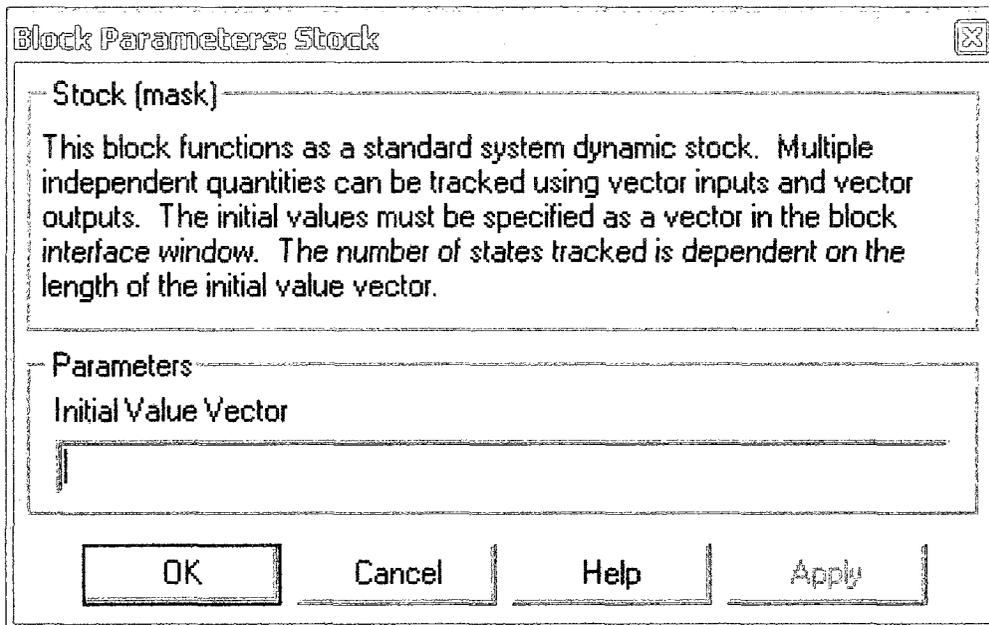


Figure 66 - Stock User Dialog Box for Setting the Initial Value

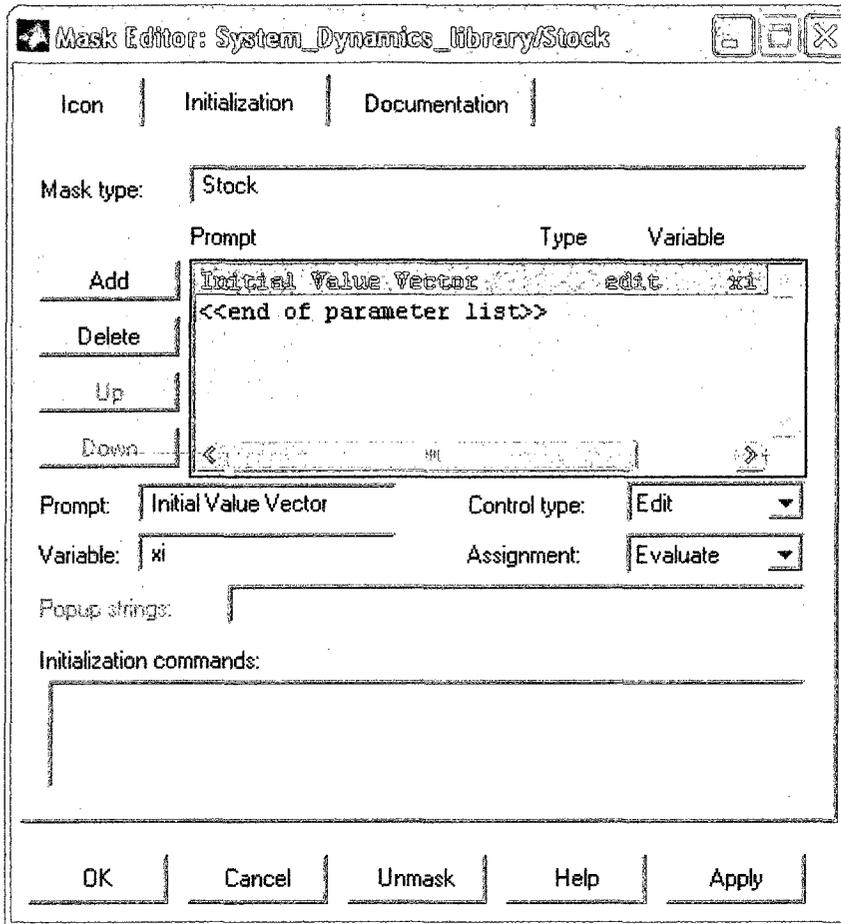


Figure 67 - Stock Mask Initialization

```

FUNCTION [SYS,X0,STR,TS] = STOCK(T,X,U,FLAG,XI)
%STOCK - A BLOCK FOR A SYSTEM DYNAMIC STOCK.
%VECTOR INPUTS ARE TREATED ELEMENTWISE AND OUTPUT
CONFIGURATION WILL MATCH INPUT CONFIGURATION
%
%
SWITCH FLAG,

%%%%%%%%%%%%%%
% INITIALIZATION %
%%%%%%%%%%%%%%
CASE 0,
    [SYS,X0,STR,TS]=MDLINITIALIZESIZES(U,XI);

%%%%%%%%%%%%%%
% UPDATE %
%%%%%%%%%%%%%%
CASE 2,
    SYS=MDLUPDATE(T,X,U);

```

```

%%%%%%%%%%
% OUTPUTS %
%%%%%%%%%%
CASE 3,
    SYS=MDLOUTPUTS(T,X,U);

%%%%%%%%%%
% UNUSED FLAGS %
%%%%%%%%%%
CASE { 1, 4, 9}
    SYS=[]; %UNUSED FLAGS

%%%%%%%%%%
% UNEXPECTED FLAGS %
%%%%%%%%%%
OTHERWISE
    ERROR(['UNHANDLED FLAG = ',NUM2STR(FLAG)]);

END

% END STOCK

%
%=====
%
% MDLINITIALIZESIZES
% RETURN THE SIZES, INITIAL CONDITIONS, AND SAMPLE TIMES FOR
% THE S-FUNCTION.
%=====
%
%
FUNCTION [SYS,X0,STR,TS]=MDLINITIALIZESIZES(U,XI)

%
%INITIALIZATION OF SIZES.  INPUTS, OUTPUTS, AND DISCRETE
%STATES ARE SET TO THE INPUT WIDTH
%
%

LU = LENGTH(U);
LXI = LENGTH(XI);
% IF (LU > LXI)
%     XI = [XI;ZEROS(LU - LXI,1)];
% END

%SHOULD STILL ADD ERROR HANDLING HERE FOR CASE OF TOO LONG AN
%XI
SIZES = SIMSIZES;
SIZES.NUMCONSTSTATES = 0;
SIZES.NUMDISCSTATES = LXI;

```

```

SIZES.NUMOUTPUTS      = LXI;
SIZES.NUMINPUTS       = LXI;
SIZES.DIRFEEDTHROUGH = 0;
SIZES.NUMSAMPLETIMES = 1;    % AT LEAST ONE SAMPLE TIME IS
NEEDED

SYS = SIMSIZES(SIZES);

%
% INITIALIZE THE INITIAL CONDITIONS
%
X0 = XI;
%
% STR IS ALWAYS AN EMPTY MATRIX
%
STR = [];

%
% INITIALIZE THE ARRAY OF SAMPLE TIMES
%
TS = [0 0];

% END MDLINITIALIZESIZES

%
%=====
%
% MDLDERIVATIVES
% RETURN THE DERIVATIVES FOR THE CONTINUOUS STATES.
%=====
%
%=====
%
% MDLUPDATE
% HANDLE DISCRETE STATE UPDATES, SAMPLE TIME HITS, AND MAJOR
TIME STEP
% REQUIREMENTS.
%=====
%
%
FUNCTION SYS=MDLUPDATE(T,X,U)

SYS = X+U;

% END MDLUPDATE

%
%=====
%
% MDLOUTPUTS

```

```

% RETURN THE BLOCK OUTPUTS.
%=====
%
%
FUNCTION SYS=MDLOUTPUTS(T,X,U)

SYS = X;

% END MDLOUTPUTS

```

A.2 Simulink Basic Functional Block Implementation

The basic functional block used throughout this dissertation is shown in Figure 68.

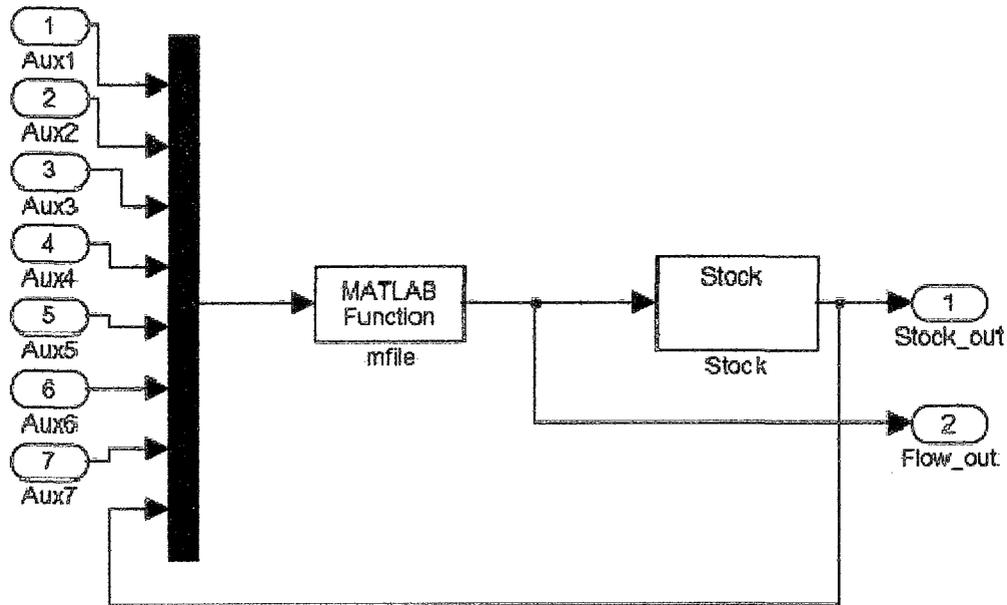


Figure 68 - Basic Functional Block Implementation

A.3 – Simulink Uniform Random Number Generator Implementation

The internal structure of the Uniform Random Number Generator is shown in Figure 69. The Matlab function block uses the code shown below in “uniform_random.m”. The mask initialization for this block is shown in Figure 70.

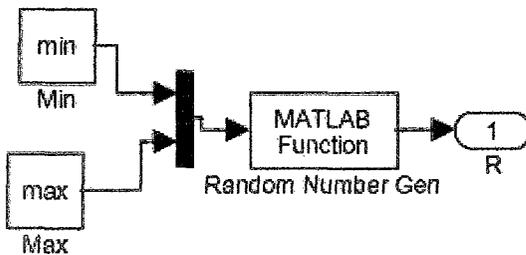


Figure 69 - Uniform Random Number Generator Block Internal Structure

```
%UNIFORM_RANDOM.M
FUNCTION RESULT=UNIFORM_RANDOM(U)
%THIS IS BLOCK TO GENERATE A UNIFORM RANDOM NUMBER BETWEEN A
%MIN AND A MAX INPUT
%AUX1 = MIN
%AUX2 = MAX
MIN = U(1);
MAX = U(2);
RESULT = RANDOM('UNIFORM',MIN,MAX,1,1);
%END UNIFORM_RANDOM
```

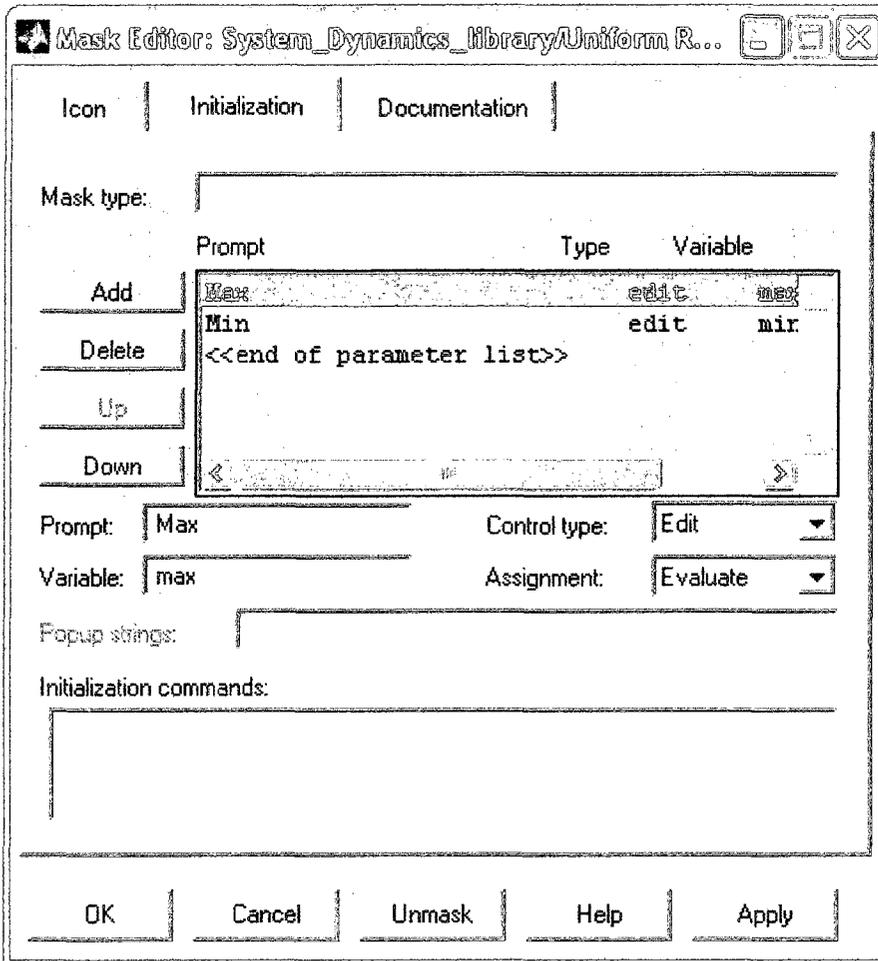


Figure 70 - Uniform Random Number Generator Block Mask Initialization

A.4 – Random Decision Making Block Implementation

Figure 71 shows the internal structure of the random decision making block. The Matlab function is implemented as shown in Figure 72. The code for the uniform random function is given in A.5 – Discrete Random Number Generation Block as it is used in multiple locations. The mask for the random decision maker block is the same as that in A.3 – Simulink Uniform Random Number Generator.

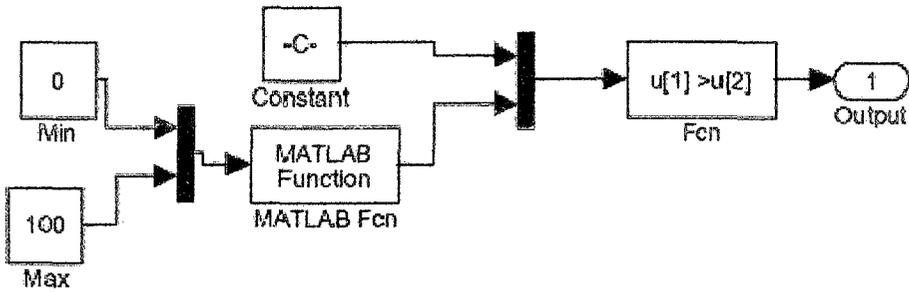


Figure 71 - Internal Structure of a Random Decision Making Block

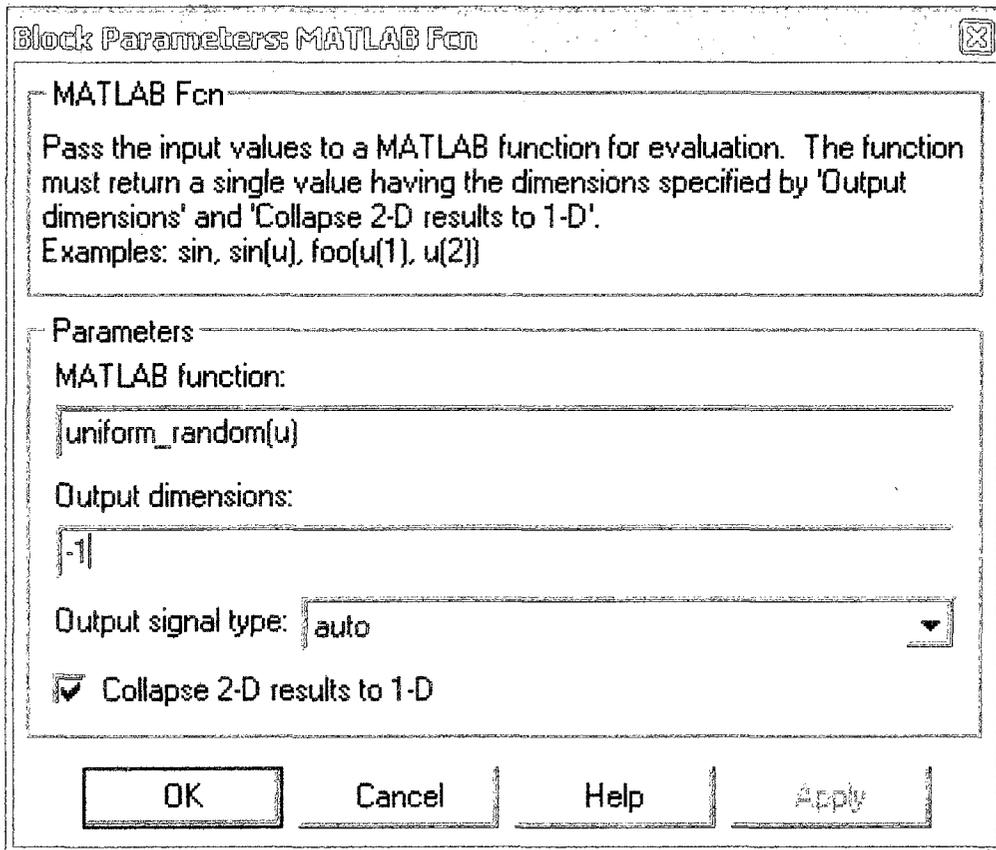


Figure 72 - Random Decision Maker Function Dialog Box

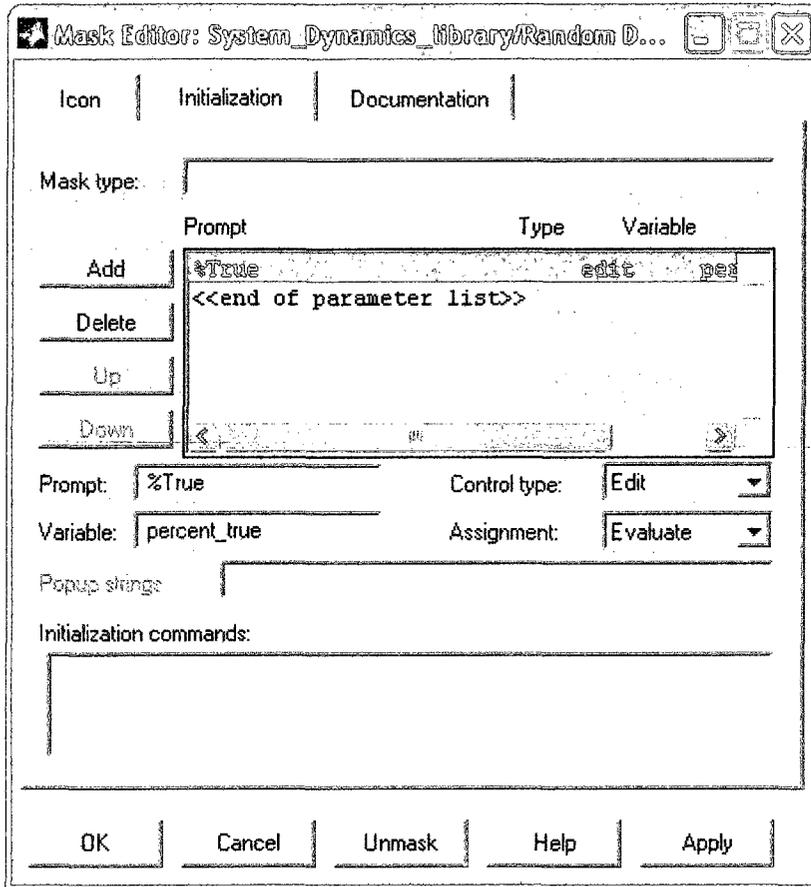


Figure 73 - Random Decision Maker Block Mask Parameter Set Up

A.5 – Discrete Random Number Generation Block

The internal structure of the discrete random number generator block is shown in Figure 74. The matlab function block uses the code shown below for “discrete_uniform_random.m”. The mask initialization dialog is shown in Figure 75.

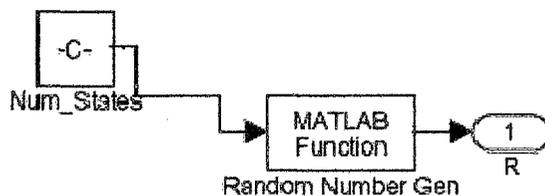


Figure 74 - Discrete Uniform Random Number Generator Block Internal Structure

```

%DISCRETE_UNIFORM_RANDOM.M
FUNCTION RESULT=DISCRETE_UNIFORM_RANDOM(U)
%THIS IS BLOCK TO GENERATE A INTEGER UNIFORM RANDOM NUMBER
BETWEEN 0 AND THE
%NUMBER OF STATES
%AUX1 = NUM_STATES
NUM_STATES = U(1);
RESULT = RANDOM('DISCRETE UNIFORM',NUM_STATES,1,1);
%END UNIFORM_RANDOM

```

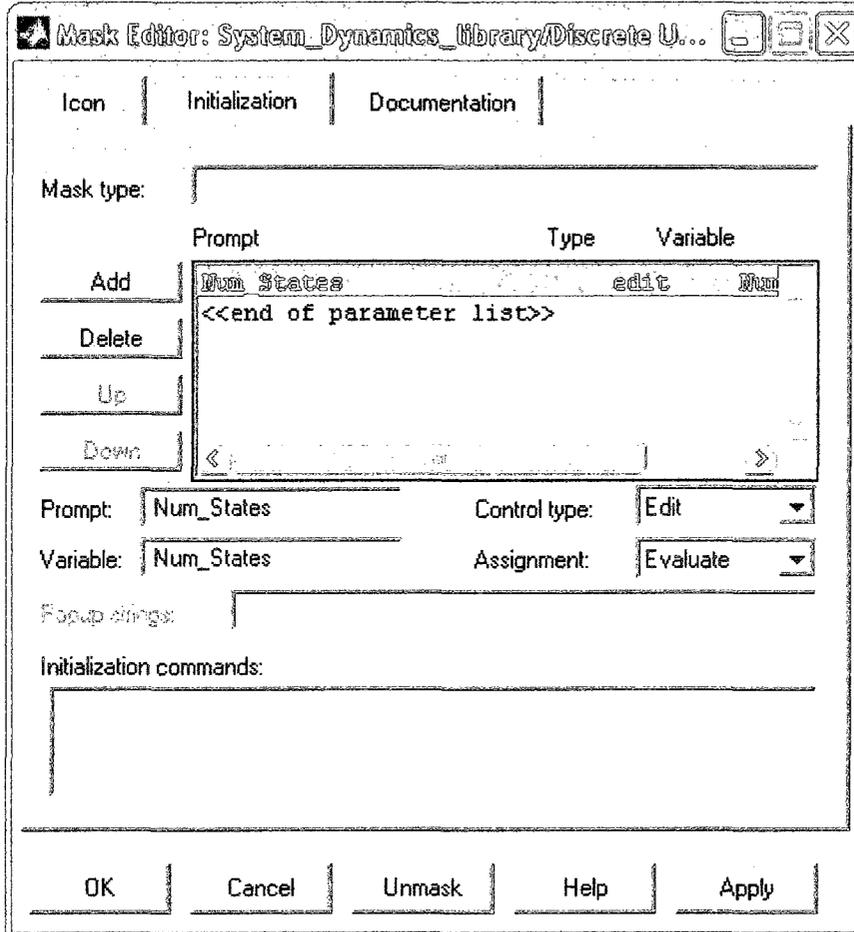


Figure 75 - Discrete Uniform Random Block Mask Initialization

Appendix B – Code for Simulink Models

This appendix contains the equations and code for the framework models from Chapter Four (or occasionally equivalent models). This code is a necessary part of the models, but in some cases is not fully debugged due to the issues presented in Chapter Five regarding the timing of the calculation of various blocks.

B.1 – Muramador Model

Note that the Actual Simulink Model for the Muramador that is used for simulations in this Dissertation is actually slightly different from the one shown in Figure 30 and is shown below in **Error! Reference source not found.** This model is substantially the same as the one shown above with the exception that the basic modeling agent had not yet been developed, but it does follow the framework.

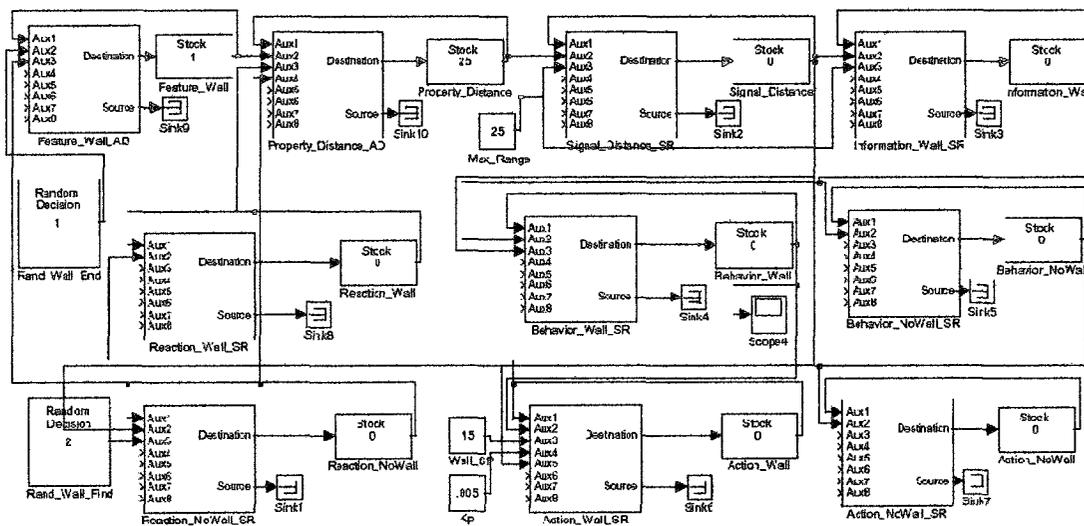


Figure 76 – Muramador Framework Model without Basic Modeling Agent

```
%MURA_ACTION_NOWALL_SR.M
FUNCTION RESULT=MURA_ACTION_NOWALL_SR(U)
%CONTROL CODE FOR THE NOWALL ACTION BLOCK
%AUX1 = ACTION_NOWALL
```

```

%           1 = LOOKING FOR A NEW WALL
%           0 = NOT LOOKING FOR A NEW WALL
%AUX2 = BEHAVIOR_NOWALL
%AUX3 = RAND_WALL_FIND
ACTION_NOWALL = U(1);
BEHAVIOR_NOWALL = U(2);
RESULT = BEHAVIOR_NOWALL - ACTION_NOWALL;
%END MURA_ACTION_NOWALL_SR.M

```

```

%MURA_ACTION_WALL_SR.M
FUNCTION RESULT=MURA_ACTION_WALL_SR(U)
%CONTROL CODE FOR THE WALL ACTION BLOCK
%AUX1 = ACTION_WALL
%           DISTANCE ROBOT TRIES TO MOVE
%AUX2 = BEHAVIOR_WALL
%AUX3 = WALL_SP
%AUX4 = KP
%AUX5 = SIGNAL_DISTANCE
ACTION_WALL = U(1);
BEHAVIOR_WALL = U(2);
WALL_SP = U(3);
KP = U(4);
SIGNAL_DISTANCE = U(5);
IF BEHAVIOR_WALL == 1
    IF SIGNAL_DISTANCE <= WALL_SP
        RESULT = (SIGNAL_DISTANCE - WALL_SP)*KP - ACTION_WALL;
    ELSE
        RESULT = -(SIGNAL_DISTANCE - WALL_SP)*KP -
ACTION_WALL;
    END
ELSE
    RESULT = -ACTION_WALL;
END
%END MURA_ACTION_WALL_SR.M

```

```

%MURA_BEHAVIOR_NOWALL_SR.M
FUNCTION RESULT=MURA_BEHAVIOR_NOWALL_SR(U)
%CONTROL CODE FOR THE NO WALL BEHAVIOR BLOCK
%AUX1 = NOWALL_BEHAVIOR
%           0 = INACTIVE
%           1 = ACTIVE
%AUX2 = INFORMATION_WALL
NOWALL_BEHAVIOR = U(1);
INFORMATION_WALL = U(2);
IF INFORMATION_WALL == 1
    RESULT = 0 - NOWALL_BEHAVIOR;
ELSE
    RESULT = 1 - NOWALL_BEHAVIOR;
END
%END MURA_NOWALL_BEHAVIOR_SR

```

```

%MURA_BEHAVIOR_WALL_SR.M

```

```

FUNCTION RESULT=MURA_BEHAVIOR_WALL_SR(U)
%CONTROL CODE FOR THE WALL BEHAVIOR BLOCK
%AUX1 = WALL_BEHAVIOR
%           0 = INACTIVE
%           1 = ACTIVE
%AUX2 = INFORMATION_WALL
WALL_BEHAVIOR = U(1);
INFORMATION_WALL = U(2);
RESULT = INFORMATION_WALL - WALL_BEHAVIOR
%END MURA_WALL_BEHAVIOR_SR

```

```

%MURA_FEATURE_WALL_AD.M
FUNCTION RESULT=MURA_FEATURE_WALL_AD(U)
%CONTROL CODE FOR THE WALL FEATURE
%AUX1 = FEATURE_WALL
%           1 = WALL
%           0 = NO WALL
%AUX2 = RAND_WALL_END
%AUX3 = REACTION_NOWALL
FEATURE_WALL = U(1);
RAND_WALL_END = U(2);
REACTION_NOWALL = U(3);
IF ((FEATURE_WALL == 1) & (RAND_WALL_END == 1))
    RESULT = -1;
ELSEIF ((FEATURE_WALL == 0) & (REACTION_NOWALL == 1))
    RESULT = 1;
ELSE
    RESULT = 0;
END
%END MURA_FEATURE_WALL_AD

```

```

%MURA_INFORMATION_WALL_SR.M
FUNCTION RESULT=MURA_INFORMATION_WALL_SR(U)
%CONTROL CODE FOR THE WALL INFORMATION BLOCK
%AUX1 = INFORMATION_WALL
%           0 = NO WALL
%           1 = WALL
%AUX2 = SIGNAL_DISTANCE
%AUX3 = MAX_RANGE
INFORMATION_WALL = U(1);
SIGNAL_DISTANCE = U(2);
MAX_RANGE = U(3);
IF (SIGNAL_DISTANCE ~= MAX_RANGE);
    RESULT = 1 - INFORMATION_WALL;
ELSE
    RESULT = 0 - INFORMATION_WALL;
END
%END MURA_SIGNAL_DISTANCE_SR

```

```

%MURA_PROPERTY_DISTANCE_AD.M
FUNCTION RESULT=MURA_PROPERTY_DISTANCE_AD(U)
%CONTROL CODE FOR THE DISTANCE PROPERTY

```

```

% AUX1 = PROPERTY_DISTANCE
% AUX2 = FEATURE_WALL
% AUX3 = REACTION_WALL
% AUX4 = REACTION_NOWALL
PROPERTY_DISTANCE = U(1);
FEATURE_WALL = U(2);
REACTION_WALL = U(3);
REACTION_NOWALL = U(4);
IF REACTION_NOWALL == 1 & FEATURE_WALL == 0
    RESULT = 24.9 - PROPERTY_DISTANCE
ELSE
    IF (FEATURE_WALL == 1);
        RESULT = REACTION_WALL;
    ELSE
        RESULT = -PROPERTY_DISTANCE
    END
END
% END MURA_PROPERTY_DISTANCE_AD

```

```

% MURA_REACTION_NOWALL_SR.M
FUNCTION RESULT=MURA_REACTION_NOWALL_SR(U)
% CONTROL CODE FOR THE NOWALL REACTION BLOCK
% AUX1 = REACTION_NOWALL
%           1 = FOUND NEW WALL
%           0 = NO NEW WALL
% AUX2 = ACTION_NOWALL
% AUX3 = RAND_WALL_FIND
REACTION_NOWALL = U(1);
ACTION_NOWALL = U(2);
RAND_WALL_FIND = U(3);
IF RAND_WALL_FIND == 1 & ACTION_NOWALL == 1
    RESULT = 1 - REACTION_NOWALL;
ELSE
    RESULT = -REACTION_NOWALL;
END
% END MURA_REACTION_NOWALL_SR.M

```

```

% MURA_REACTION_WALL_SR.M
FUNCTION RESULT=MURA_REACTION_WALL_SR(U)
% CONTROL CODE FOR THE WALL REACTION BLOCK
% AUX1 = REACTION_WALL
%           DISTANCE ROBOT ACTUALLY MOVES
% AUX2 = ACTION_WALL
REACTION_WALL = U(1);
ACTION_WALL = U(2);
RESULT = ACTION_WALL - REACTION_WALL;
% END MURA_REACTION_WALL_SR.M

```

```

% MURA_SIGNAL_DISTANCE_SR.M
FUNCTION RESULT=MURA_SIGNAL_DISTANCE_SR(U)
% CONTROL CODE FOR THE DISTANCE SIGNAL
% AUX1 = SIGNAL_DISTANCE

```

```

% AUX2 = PROPERTY_DISTANCE
% AUX3 = MAX_RANGE
SIGNAL_DISTANCE = U(1);
PROPERTY_DISTANCE = U(2);
MAX_RANGE = U(3);
IF (PROPERTY_DISTANCE ~= 0);
    RESULT = PROPERTY_DISTANCE - SIGNAL_DISTANCE;
ELSE
    RESULT = -SIGNAL_DISTANCE + MAX_RANGE;
END
% END MURA_SIGNAL_DISTANCE_SR

```

B.2 – Multi-Agent Foraging Model

```

% FORAGE_ATTRIBUTE_ENERGY.M
FUNCTION RESULT=FORAGE_ATTRIBUTE_ENERGY(U)
% AUX8 = FEEDBACK FROM STOCK
% AUX1 = ENERGY FOUND
% AUX2 = BEHAVE FORAGING
ENERGY_FOUND = U(1);
BEHAVE_FORAGING = U(2);
STOCK = U(8);
IF BEHAVE_FORAGING == 1
    RESULT = ENERGY_FOUND - STOCK;
ELSE
    RESULT = 0;
END
% END FORAGE_ATTRIBUTE_ENERGY

```

```

% FORAGE_BEHAVE_ARRIVE_NEST.M
FUNCTION RESULT=FORAGE_BEHAVE_ARRIVE_NEST(U)
% AUX8 = FEEDBACK FROM STOCK
% AUX1 = INFO AT NEST
% AUX2 = BEHAVE RETURNING
INFO_AT_NEST = U(1);
BEHAVE_RETURNING = U(2);
STOCK = U(8);
IF BEHAVE_RETURNING == 1 && INFO_AT_NEST == 1
    RESULT = 1;
ELSE
    RESULT = -STOCK;
END
% END FORAGE_BEHAVE_ARRIVE_NEST

```

```

% FORAGE_BEHAVE_ENV_ENERGY_ON_BOARD.M
FUNCTION RESULT=BEHAVE_ENV_ENERGY_ON_BOARD(U)
% AUX8 = FEEDBACK FROM STOCK
% AUX1 = FEATURE ENERGY ON BOARD
ENERGY_ON_BOARD = U(1);
STOCK = U(8);
RESULT = ENERGY_ON_BOARD - STOCK;

```

```
%END FORAGE_BEHAVE_ENV_ENERGY_ON_BOARD
```

```
%FORAGE_BEHAVE_ENV_FORAGING.M  
FUNCTION RESULT=FORAGE_BEHAVE_ENV_FORAGING(U)  
%AUX8 = FEEDBACK FROM STOCK  
%AUX1 = FEATURE FORAGING  
FEATURE_FORAGING = U(1);  
STOCK = U(8);  
RESULT = FEATURE_FORAGING - STOCK;  
%END FORAGE_BEHAVE_ENV_FORAGING
```

```
%FORAGE_PROPERTY_NEST_ENERGY.M  
FUNCTION RESULT=FORAGE_PROPERTY_NEST_ENERGY(U)  
%AUX8 = FEEDBACK FROM STOCK  
%AUX1 = RECHARGE SCALER  
%AUX2 = AGENT USE  
%AUX3 = NATIVE USE  
RECHARGE_SCALER = U(1);  
AGENT_USE = U(2);  
NATIVE_USE = U(3);  
STOCK = U(8);  
RESULT = FEATURE_RETURNING - STOCK;  
%END FORAGE_PROPERTY_NEST_ENERGY
```

```
%FORAGE_BEHAVE_FORAGING.M  
FUNCTION RESULT=FORAGE_BEHAVE_FORAGING(U)  
%AUX8 = FEEDBACK FROM STOCK  
%AUX1 = BEHAVE LEAVE NEST  
%AUX2 = BEHAVE FOUND  
BEHAVE_LEAVE_NEST = U(1);  
BEHAVE_FOUND = U(2);  
STOCK = U(8);  
IF BEHAVE_LEAVE_NEST == 1  
    RESULT = 1;  
ELSEIF BEHAVE_FOUND == 1  
    RESULT = -1;  
ELSE  
    RESULT = 0;  
END  
%END FORAGE_BEHAVE_FORAGING
```

```
%FORAGE_BEHAVE_FOUND.M  
FUNCTION RESULT=FORAGE_BEHAVE_FOUND(U)  
%AUX8 = FEEDBACK FROM STOCK  
%AUX1 = BEHAVE FORAGING  
%AUX2 = INFO ENERGY  
BEHAVE_FORAGING = U(1);  
INFO_ENERGY = U(2);  
STOCK = U(8);  
IF BEHAVE_FORAGING == 1 && INFO_ENERGY == 1  
    RESULT = 1 - STOCK;  
ELSE
```

```
    RESULT = 0 - STOCK;
END
%END FORAGE_BEHAVE_FOUND
```

```
%FORAGE_BEHAVE_LEAVE_NEST.M
FUNCTION RESULT=FORAGE_BEHAVE_LEAVE_NEST(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO SUFFICIENT ENERGY
%AUX2 = INFO AT NEST
INFO_SUFFICIENT_ENERGY = U(1);
INFO_AT_NEST = U(2);
STOCK = U(8);
IF INFO_SUFFICIENT_ENERGY == 0 && INFO_AT_NEST == 1
    RESULT = 1 - STOCK;
ELSE
    RESULT = -STOCK;
END
%END FORAGE_BEHAVE_LEAVE_NEST
```

```
%FORAGE_BEHAVE_RETURNING.M
FUNCTION RESULT=FORAGE_BEHAVE_RETURNING(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE FOUND
%AUX2 = BEHAVE ARRIVE NEST
%AUX3 = INFO SUFFICIENT ENERGY
%AUX4 = BEHAVE FORAGING
BEHAVE_FOUND = U(1);
BEHAVE_ARRIVE_NEST = U(2);
INFO_SUFFICIENT_ENERGY = U(3);
BEHAVE_FORAGING = U(4);
STOCK = U(8);
IF (BEHAVE_FOUND == 1 | INFO_SUFFICIENT_ENERGY == 0) && BEHAVE
FORAGING == 1
    RESULT = 1;
ELSEIF BEHAVE_ARRIVE_NEST == 1
    RESULT = -1;
ELSE
    RESULT = 0;
END
%END FORAGE_BEHAVE_RETURNING
```

```
%FORAGE_BEHAVE_STAY_IN_NEST.M
FUNCTION RESULT=FORAGE_BEHAVE_STAY_IN_NEST(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO SUFFICIENT ENERGY
INFO_SUFFICIENT_ENERGY = U(1);
STOCK = U(8);
RESULT = INFO_SUFFICIENT_ENERGY - STOCK;
%END FORAGE_BEHAVE_STAY_IN_NEST
```

```
%FORAGE_FEATURE_ENERGY_MODULE.M
FUNCTION RESULT=BEHAVE_FEATURE_ENERGY_MODULE(U)
```

```
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = RANDOM DECISION
RANDOM = U(1);
STOCK = U(8);
RESULT = RANDOM - STOCK;
%END FORAGE_FEATURE_ENERGY_MODULE
```

```
%FORAGE_FEATURE_ENERGY_ON_BOARD.M
FUNCTION RESULT=FORAGE_FEATURE_ENERGY_ON_BOARD(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE FOUND
%AUX2 = BEHAVE ARRIVE NEST
BEHAVE_FOUND = U(1);
BEHAVE_ARRIVE_NEST = U(2);
STOCK = U(8);
IF BEHAVE_ARRIVE_NEST == 1,
    RESULT = -STOCK;
ELSEIF BEHAVE_FOUND == 1
    RESULT = 1;
ELSE
    RESULT = 0;
END
%END FORAGE_FEATURE_ENERGY_ON_BOARD
```

```
%FORAGE_FEATURE_FORAGING.M
FUNCTION RESULT=FORAGE_FEATURE_FORAGING(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE FORAGING
BEHAVE_FORAGING = U(1);
STOCK = U(8);
RESULT = BEHAVE_FORAGING - STOCK;
%END FORAGE_FEATURE_FORAGING
```

```
%FORAGE_FEATURE_IN_NEST.M
FUNCTION RESULT=FORAGE_FEATURE_IN_NEST(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_AT_NEST
FEATURE_AT_NEST = U(1);
STOCK = U(8);
RESULT = FEATURE_AT_NEST - STOCK;
%END FORAGE_FEATURE_IN_NEST
```

```
%FORAGE_FEATURE_RETURNING.M
FUNCTION RESULT=FORAGE_FEATURE_RETURNING(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE RETURNING
BEHAVE_RETURNING = U(1);
STOCK = U(8);
RESULT = BEHAVE_RETURNING - STOCK;
%END FORAGE_FEATURE_RETURNING
```

```
%FORAGE_INFO_AT_NEST.M
```

```

FUNCTION RESULT=FORAGE_INFO_AT_NEST(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = SIGNAL NEST DISTANCE
SIGNAL_NEST_DISTANCE = U(1);
STOCK = U(8);
IF SIGNAL_NEST_DISTANCE == 0
    RESULT = 1 - STOCK;
ELSE
    RESULT = -STOCK;
END
%END FORAGE_INFO_AT_NEST

```

```

%FORAGE_INFO_ENERGY.M
FUNCTION RESULT=FORAGE_INFO_ENERGY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = ATTRIBUTE ENERGY PRESENT
%Aux2 = BEHAVE FORAGING
ATTRIBUTE_ENERGY_PRESENT = U(1);
BEHAVE_FORAGING = U(2);
STOCK = U(8);
IF BEHAVE_FORAGING == 1
    RESULT = ATTRIBUTE_ENERGY_PRESENT;
ELSE
    RESULT = 0;
END
%END FORAGE_INFO_ENERGY

```

```

%FORAGE_INFO_IN_NEST.M
FUNCTION RESULT=FORAGE_INFO_IN_NEST(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = FEATURE IN NEST
FEATURE_IN_NEST = U(1);
STOCK = U(8);
RESULT = FEATURE_IN_NEST - STOCK;
%END FORAGE_INFO_IN_NEST

```

```

%FORAGE_INFO_SUFFICIENT_AGENT_ENERGY.M
FUNCTION RESULT=INFO_SUFFICIENT_AGENT_ENERGY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = SIGNAL AGENT ENERGY
%Aux2 = ENERGY MINIMUM
SIGNAL_AGENT_ENERGY = U(1);
ENERGY_MINIMUM = U(2);
STOCK = U(8);
IF SIGNAL_AGENT_ENERGY < ENERGY_MINIMUM
    RESULT = 0;
ELSE
    RESULT = 1 - STOCK;
END
%END FORAGE_INFO_SUFFICIENT_AGENT_ENERGY

```

```

%FORAGE_INFO_SUFFICIENT_ENERGY.M

```

```

FUNCTION RESULT=FORAGE_INFO_SUFFICIENT_ENERGY(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = SIGNAL NEST ENERGY
%AUX2 = ENERGY_THRESHOLD
SIGNAL_NEST_ENERGY = U(1);
ENERGY_THRESHOLD = U(2);
STOCK = U(8);
IF SIGNAL_NEST_ENERGY >= ENERGY_THRESHOLD
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END FORAGE_INFO_SUFFICIENT_ENERGY

```

```

%FORAGE_PROPERTY_AGENT_ENERGY.M
FUNCTION RESULT=FORAGE_PROPERTY_AGENT_ENERGY(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = ENERGY USE RATE
%AUX2 = BEHAVE FORAGING
%AUX3 = BEHAVE RETURNING
%AUX4 = BEHAVE ARRIVE NEST
ENERGY_USE_RATE = U(1);
BEHAVE_FORAGING = U(2);
BEHAVE_RETURNING = U(3);
BEHAVE_ARRIVE_NEST = U(4);
STOCK = U(8);
IF BEHAVE_ARRIVE_NEST == 1
    RESULT = 100 - STOCK;
ELSE
    RESULT = (1 + BEHAVE_FORAGING + BEHAVE_RETURNING) *
ENERGY_USE_RATE
END
%END FORAGE_PROPERTY_AGENT_ENERGY

```

```

%FORAGE_PROPERTY_NEST_DISTANCE.M
FUNCTION RESULT=BEHAVE_PROPERTY_NEST_DISTANCE(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = MOVEMENT RANDOMIZATION
%AUX2 = INFO FORAGING
%AUX3 = INFO RETURNING
MOVEMENT_RAND = U(1);
INFO_FORAGING = U(2);
INFO_RETURNING = U(3);
STOCK = U(8);
IF INFO_FORAGING == 1
    RESULT = MOVEMENT_RAND;
ELSEIF INFO_RETURNING == 1
    RESULT = - 1;
ELSE
    RESULT = 0;
END
%END FORAGE_PROPERTY_NEST_DISTANCE

```

```
%FORAGE_SIGNAL_AGENT_ENERGY.M
FUNCTION RESULT=FORAGE_SIGNAL_AGENT_ENERGY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = PROPERTY AGENT ENERGY
PROPERTY_AGENT_ENERGY = U(1);
STOCK = U(8);
RESULT = PROPERTY_AGENT_ENERGY - STOCK;
%END FORAGE_SIGNAL_AGENT_ENERGY
```

```
%FORAGE_SIGNAL_ENV_AGENT_ENERGY.M
FUNCTION RESULT=FORAGE_SIGNAL_ENV_AGENT_ENERGY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = PROPERTY AGENT ENERGY
PROPERTY_AGENT_ENERGY = U(1);
STOCK = U(8);
RESULT = PROPERTY_AGENT_ENERGY - STOCK;
%END FORAGE_SIGNAL_ENV_AGENT_ENERGY
```

```
%FORAGE_SIGNAL_NEST_DISTANCE.M
FUNCTION RESULT=FORAGE_SIGNAL_NEST_DISTANCE(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = PROPERTY_NEST_DISTANCE
PROPERTY_NEST_DISTANCE = U(1);
STOCK = U(8);
RESULT = PROPERTY_NEST_DISTANCE - STOCK;
%END FORAGE_SIGNAL_NEST_DISTANCE
```

```
%FORAGE_SIGNAL_NEST_ENERGY.M
FUNCTION RESULT=FORAGE_SIGNAL_NEST_ENERGY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = ATTRIBUTE_STATE
PROPERTY_NEST_ENERGY= U(1);
STOCK = U(8);
RESULT = PROPERTY_NEST_ENERGY - STOCK;
%END FORAGE_SIGNAL_NEST_ENERGY
```

```
%FORAGE_SIGNAL_USE_NEST_ENERGY.M
FUNCTION RESULT=FORAGE_SIGNAL_USE_NEST_ENERGY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = INFO IN NEST
%Aux2 = PROPERTY AGENT ENERGY
INFO_IN_NEST = U(1);
PROPERTY_AGENT_ENERGY = U(2);
STOCK = U(8);
IF INFO_IN_NEST == 1
    RESULT = 100 - PROPERTY_AGENT_ENERGY - STOCK
ELSE
    RESULT = -STOCK;
END
%END FORAGE_SIGNAL_USE_NEST_ENERGY
```

B.3 – GSVD Model

```
%GSVD_ATTRIBUTE_MOTION.M
FUNCTION RESULT=GSVD_ATTRIBUTE_MOTION(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = ATTRIBUTE_STATE
ATTRIBUTE_STATE= U(1);
STOCK = U(8);
IF ATTRIBUTE_STATE == 0 | ATTRIBUTE_STATE == 1
    RESULT = 0 - STOCK;
ELSEIF ATTRIBUTE_STATE == 2
    RESULT = 1 - STOCK;
ELSE
    RESULT = 2 - STOCK;
END
%END GSVD_ATTRIBUTE_MOTION
```

```
%GSVD_ATTRIBUTE_SOUND.M
FUNCTION RESULT=GSVD_ATTRIBUTE_SOUND(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = ATTRIBUTE_STATE
ATTRIBUTE_STATE= U(1);
STOCK = U(8);
IF ATTRIBUTE_STATE == 0 | ATTRIBUTE_STATE == 1
    RESULT = 0 - STOCK;
ELSEIF ATTRIBUTE_STATE == 2
    RESULT = 1 - STOCK;
ELSE
    RESULT = 2 - STOCK;
END
%END GSVD_ATTRIBUTE_SOUND
```

```
%GSVD_ATTRIBUTE_STATE.M
FUNCTION RESULT=GSVD_ATTRIBUTE_STATE(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = FEATURE_VICTIM
%Aux2 = RANDOM
FEATURE_VICTIM= U(1);
RANDOM = U(2);
STOCK = U(8);
IF FEATURE_VICTIM ~= 0
    RESULT = RANDOM - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_ATTRIBUTE_STATE
```

```
%GSVD_BEHAVE_FOUND.M
FUNCTION RESULT=GSVD_BEHAVE_FOUND(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = INFO_VICTIM
```

```
INFO_VICTIM = U(1);
STOCK = U(8);
RESULT = INFO_VICTIM - STOCK;
%END GSVD_BEHAVE_FOUND
```

```
%GSVD_BEHAVE_FOUND_CORRECT.M
FUNCTION RESULT=GSVD_BEHAVE_FOUND_CORRECT(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_FIND
%AUX2 = FEATURE_VICTIM
INFO_FIND = U(1);
FEATURE_VICTIM = U(2);
STOCK = U(8);
IF (INFO_FIND == 1) & (FEATURE_VICTIM == 1)
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_BEHAVE_FOUND_CORRECT
```

```
%GSVD_BEHAVE_FOUND_WRONG.M
FUNCTION RESULT=GSVD_BEHAVE_FOUND_WRONG(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_FIND
%AUX2 = FEATURE_VICTIM
INFO_FIND = U(1);
FEATURE_VICTIM = U(2);
STOCK = U(8);
IF INFO_FIND == 1 & FEATURE_VICTIM == 0
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_BEHAVE_FOUND_WRONG
```

```
%GSVD_BEHAVE_MISSED.M
FUNCTION RESULT=GSVD_BEHAVE_MISSED(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_FIND
%AUX2 = FEATURE_VICTIM
INFO_FIND = U(1);
FEATURE_VICTIM = U(2);
STOCK = U(8);
IF INFO_FIND == 0 & FEATURE_VICTIM == 1
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_BEHAVE_MISSED
```

```
%GSVD_BEHAVE_RESEARCH.M
FUNCTION RESULT=GSVD_BEHAVE_RESEARCH(U)
```

```
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_UNEXPLORED_TERRAIN
INFO_UNEXPLORED_TERRAIN = U(1);
STOCK = U(8);
RESULT = 1 - INFO_UNEXPLORED_TERRAIN - STOCK;
%END GSVD_BEHAVE_RESEARCH
```

```
%GSVD_BEHAVE_RESET_TERRAIN.M
FUNCTION RESULT=GSVD_BEHAVE_RESET_TERRAIN(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_RESET
INFO_RESET = U(1);
STOCK = U(8);
RESULT = INFO_RESET - STOCK;
%END GSVD_BEHAVE_RESET_TERRAIN
```

```
%GSVD_BEHAVE_SEARCH.M
FUNCTION RESULT=GSVD_BEHAVE_SEARCH(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_VICTIM
INFO_VICTIM = U(1);
STOCK = U(8);
RESULT = 1 - INFO_VICTIM - STOCK;
%END GSVD_BEHAVE_SEARCH
```

```
%GSVD_FEATURE_CO2.M
FUNCTION RESULT=GSVD_FEATURE_CO2(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = ATTRIBUTE_STATE
ATTRIBUTE_STATE = U(1);
STOCK = U(8);
IF ATTRIBUTE_STATE ~= 0
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_FEATURE_CO2
```

```
%GSVD_FEATURE_EXPLORING.M
FUNCTION RESULT=GSVD_FEATURE_EXPLORING(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE_SEARCH
BEHAVE_SEARCH = U(1);
STOCK = U(8);
RESULT = BEHAVE_SEARCH - STOCK;
%END GSVD_FEATURE_EXPLORING
```

```
%GSVD_FEATURE_FIND.M
FUNCTION RESULT=GSVD_FEATURE_FIND(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE_FOUND
BEHAVE_FOUND = U(1);
```

```
STOCK = U(8);
RESULT = BEHAVE_FOUND - STOCK;
%END GSVD_FEATURE_FIND
```

```
%GSVD_FEATURE_FORM.M
FUNCTION RESULT=GSVD_FEATURE_FORM(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = ATTRIBUTE_STATE
ATTRIBUTE_STATE= U(1);
STOCK = U(8);
IF ATTRIBUTE_STATE ~= 0
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_FEATURE_FORM
```

```
%GSVD_FEATURE_HEAT.M
FUNCTION RESULT=GSVD_FEATURE_HEAT(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = ATTRIBUTE_STATE
ATTRIBUTE_STATE= U(1);
STOCK = U(8);
IF ATTRIBUTE_STATE ~= 0
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_FEATURE_HEAT
```

```
%GSVD_FEATURE_HEAT.M
FUNCTION RESULT=GSVD_FEATURE_HEAT(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = ATTRIBUTE_STATE
ATTRIBUTE_STATE= U(1);
STOCK = U(8);
IF ATTRIBUTE_STATE ~= 0
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_FEATURE_HEAT
```

```
%GSVD_FEATURE_RESEARCH.M
FUNCTION RESULT=GSVD_FEATURE_RESEARCH(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE_RESEARCH
BEHAVE_RESEARCH = U(1);
STOCK = U(8);
RESULT = BEHAVE_RESEARCH - STOCK;
%END GSVD_FEATURE_RESEARCH
```

```

%GSVD_FEATURE_UNEXPLORED_TER.M
FUNCTION RESULT=GSVD_FEATURE_UNEXPLORED_TER(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = PROPERTY_UNEXPLORED_TERRAIN
UNEXPLORED_TERRAIN = U(1);
THRESHOLD = U(2);
STOCK = U(8);
IF UNEXPLORED_TERRAIN < THRESHOLD
    RESULT = 0 - STOCK;
ELSE
    RESULT = 1 - STOCK;
END
%END GSVD_FEATURE_UNEXPLORED_TER.M

```

```

%GSVD_FEATURE_UNEXPLORED_TERRAIN.M
FUNCTION RESULT=GSVD_FEATURE_UNEXPLORED_TERRAIN(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = PROPERTY_UNEXPLORED_TERRAIN
UNEXPLORED_TERRAIN = U(1);
THRESHOLD = U(2);
STOCK = U(8);
IF UNEXPLORED_TERRAIN < THRESHOLD
    RESULT = 0 - STOCK;
ELSE
    RESULT = 1 - STOCK;
END
%END GSVD_FEATURE_UNEXPLORED_TERRAIN

```

```

%MURA_SIGNAL_DISTANCE_SR.M
FUNCTION RESULT=GSVD_FEATURE_VICTIM(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = REMAINING VICTIM DENSITY
%Aux2 = TERRAIN EXPLORATION RATE
%Aux3 = RANDOM NUMBER GENERATOR
VICTIM_DENSITY = U(1);
EXPLORATION_RATE = U(2);
RAND = U(3);
STOCK = U(8);
IF (VICTIM_DENSITY * EXPLORATION_RATE) > RAND;
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_SIGNAL_DISTANCE_SR

```

```

%GSVD_INFO_CO2.M
FUNCTION RESULT=GSVD_INFO_CO2(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = INFO_CO2
INFO_CO2 = U(1);
STOCK = U(8);
RESULT = INFO_CO2 - STOCK;

```

%END GSVD_INFO_CO2

%GSVD_INFO_FIND.M
FUNCTION RESULT=GSVD_INFO_FIND(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_FIND
FEATURE_FIND = U(1);
STOCK = U(8);
RESULT = FEATURE_FIND - STOCK;
%END GSVD_INFO_FIND

%GSVD_INFO_FORM.M
FUNCTION RESULT=GSVD_INFO_FORM(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_FORM
INFO_FORM = U(1);
STOCK = U(8);
RESULT = INFO_FORM - STOCK;
%END GSVD_INFO_FORM

%GSVD_INFO_HEAT.M
FUNCTION RESULT=GSVD_INFO_HEAT(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_HEAT
INFO_HEAT = U(1);
STOCK = U(8);
RESULT = INFO_HEAT - STOCK;
%END GSVD_INFO_HEAT

%GSVD_INFO_MOTION.M
FUNCTION RESULT=GSVD_INFO_MOTION(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_SOUND
INFO_MOTION = U(1);
STOCK = U(8);
RESULT = INFO_MOTION - STOCK;
%END GSVD_INFO_MOTION

%GSVD_INFO_RESEARCH.M
FUNCTION RESULT=GSVD_INFO_RESEARCH(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_RESEARCH
FEATURE_RESEARCH = U(1);
STOCK = U(8);
RESULT = FEATURE_RESEARCH - STOCK;
%END GSVD_INFO_RESEARCH

%GSVD_INFO_MOTION.M
FUNCTION RESULT=GSVD_INFO_SOUND(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_SOUND
INFO_SOUND = U(1);

```
STOCK = U(8);
RESULT = INFO_SOUND - STOCK;
%END GSVD_INFO_MOTION
```

```
%MURA_SIGNAL_DISTANCE_SR.M
FUNCTION RESULT=GSVD_INFO_STATE(U)
RESULT = 0;
%END GSVD_SIGNAL_DISTANCE_SR
```

```
%GSVD_INFO_UNEXPLORED_TERRAIN.M
FUNCTION RESULT=GSVD_INFO_UNEXPLORED_TERRAIN(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = SIGNAL_UNEXPLORED_TERRAIN
SIGNAL_UNEXPLORED_TERRAIN= U(1);
STOCK = U(8);
RESULT = SIGNAL_UNEXPLORED_TERRAIN - STOCK;
%END GSVD_INFO_UNESPLORED_TERRAIN
```

```
%GSVD_INFO_VICTIM.M
FUNCTION RESULT=GSVD_INFO_VICTIM(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO_VICTIM
INFO_FORM = U(1);
INFO_CO2 = U(2);
INFO_HEAT = U(3);
INFO_MOTION = U(4);;
INFO_SOUND = U(5);
STOCK = U(8);
COUNT = 0;
IF INFO_FORM == 1
    COUNT = COUNT + 1;
END
IF INFO_CO2 == 1
    COUNT = COUNT + 1;
END
IF INFO_HEAT == 1
    COUNT = COUNT + 1;
END
IF INFO_MOTION == 1 | INFO_MOTION == 2
    COUNT = COUNT + 1;
END
IF INFO_SOUND == 1 | INFO_SOUND == 2
    COUNT = COUNT + 1;
END
IF COUNT >= 3
    RESULT = 1 - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_INFO_VICTIM
```

```
%GSVD_PROPERTY_EXPLORATION_RATE.M
```

```

FUNCTION RESULT=GSVD_PROPERTY_EXPLORATION_RATE(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = SIGNAL TERRAIN DIFFICULTY
FEATURE_EXPLORING = U(1);
MAX_RATE = U(2);
SIGNAL_TERRAIN_DIFFICULTY = U(3);
STOCK = U(8);
IF FEATURE_EXPLORING
    RESULT = (1 - SIGNAL_TERRAIN_DIFFICULTY)*MAX_RATE - STOCK;
ELSE
    RESULT = 0 - STOCK;
END
%END GSVD_PROPERTY_EXPLORATION_RATE

```

```

%GSVD_PROPERTY_REMAINING_VICTIM_DENSITY.M
FUNCTION RESULT=GSVD_PROPERTY_REMAINING_VICTIM_DENSITY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = UNEXPLORED_TERRAIN
%Aux2 = UNPASSED_VICTIMS
UNEXPLORED_TERRAIN = U(1);
UNPASSED_VICTIMS = U(2);
STOCK = U(8);
RESULT = UNPASSED_VICTIMS/UNEXPLORED_TERRAIN - STOCK;
%END GSVD_PROPERTY_REMAINING_VICTIM_DENSITY

```

```

%GSVD_PROPERTY_TERRAIN_DIF.M
FUNCTION RESULT=GSVD_PROPERTY_TERRAIN_DIF(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = INFO_UNEXPLORED_TERRAIN
RANDOM = U(1);
STOCK = U(8);
IF (RANDOM + STOCK) > 1
    RESULT = 1 - STOCK;
ELSEIF (RANDOM + STOCK) < 0
    RESULT = 0 - STOCK;
ELSE
    RESULT = RANDOM;
END
%END GSVD_PROPERTY_TERRAIN_DIF.M

```

```

%GSVD_PROPERTY_TERRAIN_DIFFICULTY.M
FUNCTION RESULT=GSVD_PROPERTY_TERRAIN_DIFFICULTY(U)
%Aux8 = FEEDBACK FROM STOCK
%Aux1 = INFO_UNEXPLORED_TERRAIN
RANDOM = U(1);
STOCK = U(8);
IF (RANDOM + STOCK) > 1
    RESULT = 1 - STOCK;
ELSEIF (RANDOM + STOCK) < 0
    RESULT = 0 - STOCK;
ELSE
    RESULT = RANDOM;

```

```

END
%END GSVD_PROPERTY_TERRAIN_DIFFICULTY

```

```

%GSVD_PROPERTY_UNEXPLORED_TER.M
FUNCTION RESULT=GSVD_PROPERTY_UNEXPLORED_TER(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = BEHAVE_TERRAIN_RESET
%AUX2 = SIGNAL_TERRAIN_EXPLORATION
TERRAIN_RESET = U(1);
TERRAIN_EXPLORATION = U(2);
STOCK = U(8);
IF TERRAIN_RESET == 1
    RESULT = 100 - STOCK;
ELSE
    RESULT = -TERRAIN_EXPLORATION
END
%END GSVD_PROPERTY_UNEXPLORED_TER.M

```

```

%GSVD_PROPERTY_UNFOUNDED_VICTIMS.M
FUNCTION RESULT=GSVD_PROPERTY_UNFOUNDED_VICTIMS(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FIND_CORRECT
FIND_CORRECT = U(1);
STOCK = U(8);
IF STOCK == 0
    RESULT = 0;
ELSEIF FIND_CORRECT == 1
    RESULT = -1;
ELSE
    RESULT = 0;
END
%END GSVD_PROPERTY_UNFOUNDED_VICTIMS

```

```

%GSVD_PROPERTY_UNPASSED_VICTIMS.M
FUNCTION RESULT=GSVD_PROPERTY_UNPASSED_VICTIMS(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = RESET_TERRAIN
%AUX2 = MISSED
%AUX3 = FIND_CORRECT
%AUX4 = UNFOUNDED_VICTIMS
RESET_TERRAIN = U(1);
MISSED = U(2);
FIND_CORRECT = U(3);
UNFOUNDED_VICTIMS = U(4);
STOCK = U(8);
IF RESET_TERRAIN == 1
    RESULT = UNFOUNDED_VICTIMS - STOCK;
ELSEIF MISSED == 1 | FIND_CORRECT == 1
    RESULT = - 1;
ELSE
    RESULT = 0;
END
END

```

%END GSVD_PROPERTY_UNPASSED_VICTIMS

%GSVD_PROPERTY_VICTIM_DENSITY.M
FUNCTION RESULT=GSVD_PROPERTY_VICTIM_DENSITY(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = UNEXPLORED_TERRAIN
%AUX2 = UNPASSED_VICTIMS
UNEXPLORED_TERRAIN = U(1);
UNPASSED_VICTIMS = U(2);
STOCK = U(8);
RESULT = UNPASSED_VICTIMS/UNEXPLORED_TERRAIN - STOCK;
%RESULT = 0;
%END GSVD_PROPERTY_VICTIM_DENSITY

%GSVD_SIGNAL_CO2.M
FUNCTION RESULT=GSVD_SIGNAL_CO2(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_CO2
FEATURE_CO2 = U(1);
STOCK = U(8);
RESULT = FEATURE_CO2 - STOCK;
%END GSVD_SIGNAL_CO2

%GSVD_SIGNAL_EXPLORATION_RATE.M
FUNCTION RESULT=GSVD_SIGNAL_EXPLORATION_RATE(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = PROPERTY_EXPLORATION_RATE
PROPERTY_EXPLORATION_RATE = U(1);
STOCK = U(8);
RESULT = PROPERTY_EXPLORATION_RATE - STOCK;
%END GSVD_SIGNAL_EXPLORATION_RATE

%GSVD_SIGNAL_FORM.M
FUNCTION RESULT=GSVD_SIGNAL_FORM(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_FORM
FEATURE_FORM = U(1);
STOCK = U(8);
RESULT = FEATURE_FORM - STOCK;
%END GSVD_SIGNAL_FORM

%GSVD_SIGNAL_HEAT.M
FUNCTION RESULT=GSVD_SIGNAL_HEAT(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_HEAT
FEATURE_HEAT = U(1);
STOCK = U(8);
RESULT = FEATURE_HEAT - STOCK;
%END GSVD_SIGNAL_HEAT

%GSVD_SIGNAL_MOTION.M
FUNCTION RESULT=GSVD_SIGNAL_MOTION(U)

```
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_MOTION
FEATURE_MOTION = U(1);
STOCK = U(8);
RESULT = FEATURE_MOTION - STOCK;
%END GSVD_SIGNAL_MOTION
```

```
%GSVD_SIGNAL_SOUND.M
FUNCTION RESULT=GSVD_SIGNAL_SOUND(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_SOUND
FEATURE_SOUND = U(1);
STOCK = U(8);
RESULT = FEATURE_SOUND - STOCK;
%END GSVD_SIGNAL_SOUND
```

```
%FORAGE_SIGNAL_USE_NEST_ENERGY.M
FUNCTION RESULT=FORAGE_SIGNAL_USE_NEST_ENERGY(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = INFO IN NEST
%AUX2 = PROPERTY_AGENT_ENERGY
INFO_IN_NEST = U(1);
PROPERTY_AGENT_ENERGY = U(2);
STOCK = U(8);
IF INFO_IN_NEST == 1
    RESULT = 100 - PROPERTY_AGENT_ENERGY - STOCK
ELSE
    RESULT = -STOCK;
END
%END FORAGE_SIGNAL_USE_NEST_ENERGY
```

```
%GSVD_SIGNAL_TERRAIN_DIFFICULTY.M
FUNCTION RESULT=GSVD_SIGNAL_TERRAIN_DIFFICULTY(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = SIGNAL_TERRAIN_DIFFICULTY
SIGNAL_TERRAIN_DIFFICULTY = U(1);
STOCK = U(8);
RESULT = SIGNAL_TERRAIN_DIFFICULTY - STOCK;
%END GSVD_SIGNAL_TERRAIN_DIFFICULTY
```

```
%GSVD_SIGNAL_UNEXPLORED_TERRAIN.M
FUNCTION RESULT=GSVD_SIGNAL_UNEXPLORED_TERRAIN(U)
%AUX8 = FEEDBACK FROM STOCK
%AUX1 = FEATURE_UNEXPLORED_TERRAIN
FEATURE_UNEXPLORED_TERRAIN= U(1);
STOCK = U(8);
RESULT = FEATURE_UNEXPLORED_TERRAIN - STOCK;
%END GSVD_SIGNAL_UNEXPLORED_TERRAIN
```

Appendix C – PowerSim Code

This appendix contains the original PowerSim simulations originally developed for [2]. Some of these models were reused for some of the results in Chapter Five, as such the code and models are presented here for completeness.

C.1 – Muramador Program Listing (PowerSim)

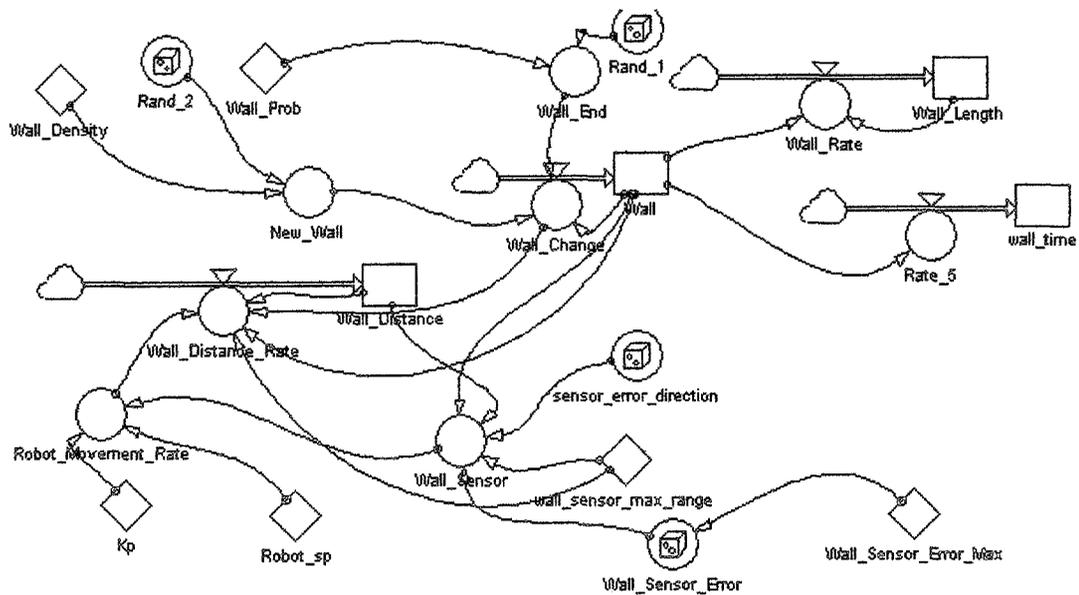


Figure 77 - PowerSim Muramador Model

Wall
 [INIT] 1
 $\rightarrow +dt * \text{Wall_Change}$

Wall_Distance
 [INIT] 10
 $\rightarrow +dt * \text{Wall_Distance_Rate}$

Wall_Length
 [INIT] 0
 $\rightarrow +dt * \text{Wall_Rate}$

wall_time
 [INIT] 0
 $\rightarrow +dt * \text{Rate}_5$

Rate_5
 = IF(Wall = 1,1,0)

Wall_Change
 = IF(Wall = 1 AND Wall_End = 1,-1,IF(Wall = 0 AND New_Wall = 1,1,0))

Wall_Distance_Rate
 = IF(Wall_Change=1,wall_sensor_max_range-5,IF(Wall_Change=-1,-Wall_Distance,IF(Wall=1,-Robot_Movement_Rate,0)))

Wall_Rate
 = IF(Wall = 1,1,-Wall_Length)

New_Wall
 = IF(Rand_2 < Wall_Density,1,0)

Rand_1
 = RANDOM

Rand_2
 = RANDOM

Robot_Movement_Rate
 = IF(Wall_Sensor < Robot_sp, -(Wall_Sensor - Robot_sp) * Kp, (Wall_Sensor - Robot_sp) * Kp)

sensor_error_direction
 = RANDOM

Wall_End
 = IF(Rand_1 < Wall_Prob,1,0)

Wall_Sensor
 = IF(Wall = 1, IF(Wall_Distance < wall_sensor_max_range, IF(sensor_error_direction < .5, Wall_Distance - Wall_Sensor_Error, Wall_Distance + Wall_Sensor_Error), wall_sensor_max_range), wall_sensor_max_range)

Wall_Sensor_Error
 = Wall_Sensor_Error_Max * RANDOM

Kp
 = .02

Robot_sp
 = 10

Wall_Density
 = .01

Wall_Prob
 = .005

Wall_Sensor_Error_Max
 = 5

wall_sensor_max_range
 = 25

C.2 Foraging Program Listing (PowerSim)

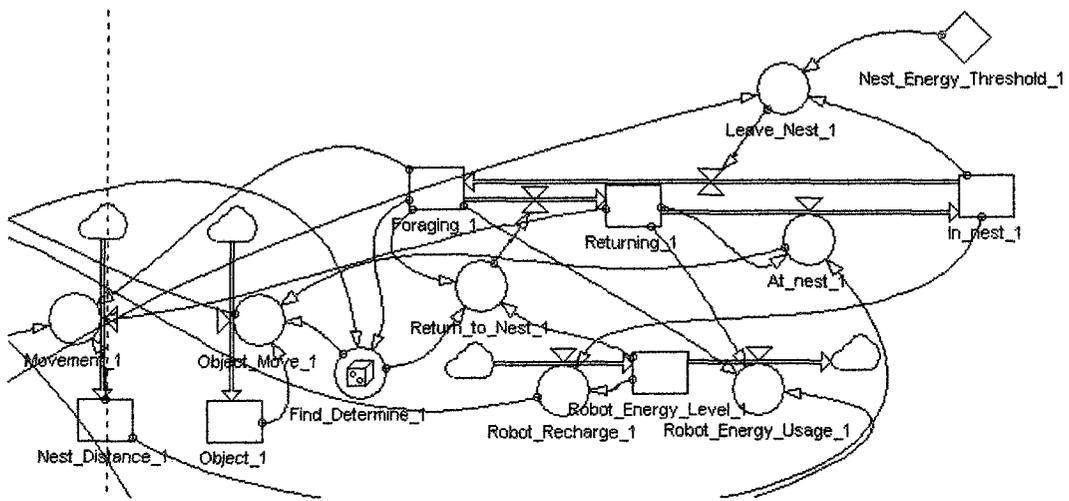


Figure 78 - Foraging Agent PowerSim Model

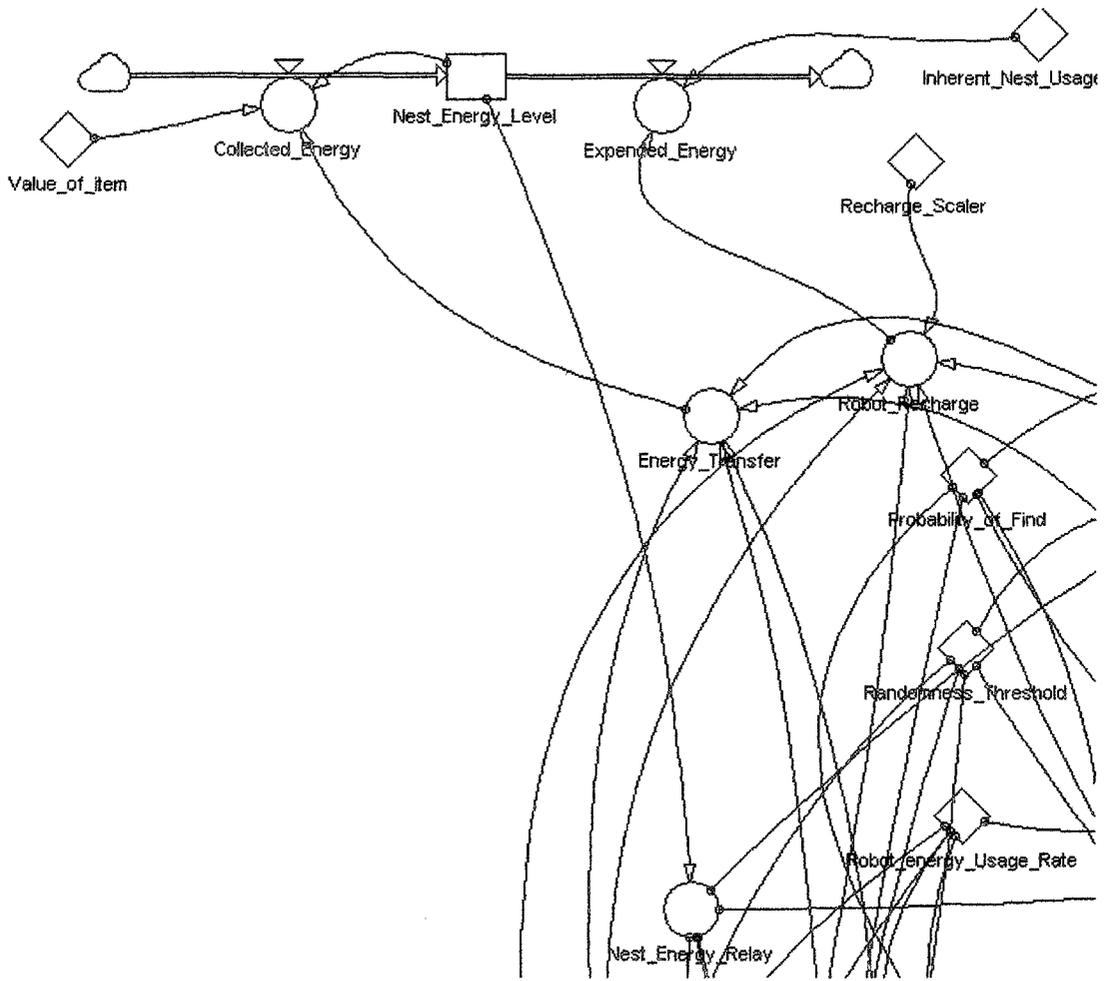


Figure 79 - Foraging Agent Environment PowerSim Model

- Foraging_1
 0
 \rightarrow +dt*Leave_Nest_1
 \leftarrow -dt*Return_to_Nest_1
- Foraging_2
 0
 \leftarrow -dt*Return_to_Nest_2
 \rightarrow +dt*Leave_Nest_2
- Foraging_3
 0
 \leftarrow -dt*Return_to_Nest_3
 \leftarrow -dt*Leave_Nest_3
- Foraging_4
 0
 \leftarrow -dt*Return_to_Nest_4
 \rightarrow +dt*Leave_Nest_4
- Foraging_5
 0
 \rightarrow +dt*Leave_Nest_5
 \leftarrow -dt*Return to Nest 5
- In_nest_1
 1
 \rightarrow +dt*At_nest_1
 \leftarrow -dt*Leave_Nest_1
- In_nest_2
 1
 \rightarrow +dt*At_nest_2
 \leftarrow -dt*Leave Nest 2
- In_nest_3
 1
 \rightarrow +dt*At_nest_3
 \leftarrow -dt*Leave_Nest_3
- In_nest_4
 1
 \rightarrow +dt*At_nest_4
 \leftarrow -dt*Leave Nest 4
- In_nest_5
 1
 \rightarrow +dt*At_nest_5
 \leftarrow -dt*Leave_Nest_5
- Nest_Distance_1
 0
 \rightarrow +dt*Movement_1
- Nest_Distance_2
 0
 \leftarrow -dt*Movement_2
- Nest_Distance_3
 0
 \leftarrow -dt*Movement_3
- Nest_Distance_4
 0
 \rightarrow +dt*Movement_4
- Nest_Distance_5
 0
 \leftarrow -dt*Movement 5
- Nest_Energy_Level
 100
 \leftarrow -dt*Expended_Energy
 \rightarrow +dt*Collected_Energy
- Object_1
 0
 \rightarrow +dt*Object_Move_1

- Object_2
 INT 0
 \Rightarrow +dt*Object_Move_2
- Object_3
 INT 0
 \Rightarrow +dt*Object_Move_3
- Object_4
 INT 0
 \Rightarrow +dt*Object_Move_4
- Object_5
 INT 0
 \Rightarrow +dt*Object_Move_5
- Returning_1
 INT 0
 \Rightarrow -dt*At_nest_1
 +dt*Return_to_Nest_1
- Returning_2
 INT 0
 \Rightarrow -dt*At_nest_2
 +dt*Return_to_Nest_2
- Returning_3
 INT 0
 \Rightarrow -dt*At_nest_3
 +dt*Return_to_Nest_3
- Returning_4
 INT 0
 \Rightarrow -dt*At_nest_4
 +dt*Return_to_Nest_4
- Returning_5
 INT 0
 \Rightarrow -dt*At_nest_5
 +dt*Return_to_Nest_5
- Robot_Energy_Level_1
 INT 100
 \Rightarrow -dt*Robot_Energy_Usage_1
 +dt*Robot_Recharge_1
- Robot_Energy_Level_2
 INT 100
 \Rightarrow -dt*Robot_Energy_Usage_2
 +dt*Robot_Recharge_2
- Robot_Energy_Level_3
 INT 100
 \Rightarrow -dt*Robot_Energy_Usage_3
 +dt*Robot_Recharge_3
- Robot_Energy_Level_4
 INT 100
 \Rightarrow -dt*Robot_Energy_Usage_4
 +dt*Robot_Recharge_4
- Robot_Energy_Level_5
 INT 100
 \Rightarrow +dt*Robot_Recharge_5
 -dt*Robot_Energy_Usage_5
- At_nest_1
 \equiv IF(Nest_Distance_1 = 0 AND Returning_1 = 1,1,0)
- At_nest_2
 \equiv IF(Nest_Distance_2 = 0 AND Returning_2 = 1,1,0)
- At_nest_3
 \equiv IF(Nest_Distance_3 = 0 AND Returning_3 = 1,1,0)
- At_nest_4
 \equiv IF(Nest_Distance_4 = 0 AND Returning_4 = 1,1,0)

- ⊖⊕ At_nest_5
= IF(Nest_Distance_5 = 0 AND Returning_5 = 1,1,0)
- ⊖⊕ Collected_Energy
= MIN(Energy_Transfer*Value_of_item,100 - Nest_Energy_Level)
- ⊖⊕ Expended_Energy
= Robot_Recharge+Inherent_Nest_Usage_Rate
- ⊖⊕ Leave_Nest_1
= IF(In_nest_1 = 1 AND Nest_Energy_Relay < Nest_Energy_Threshold_1,1,0)
- ⊖⊕ Leave_Nest_2
= IF(In_nest_2 = 1 AND Nest_Energy_Relay < Nest_Energy_Threshold_2,1,0)
- ⊖⊕ Leave_Nest_3
= IF(In_nest_3 = 1 AND Nest_Energy_Relay < Nest_Energy_Threshold_3,1,0)
- ⊖⊕ Leave_Nest_4
= IF(In_nest_4 = 1 AND Nest_Energy_Relay < Nest_Energy_Threshold_4,1,0)
- ⊖⊕ Leave_Nest_5
= IF(In_nest_5 = 1 AND Nest_Energy_Relay < Nest_Energy_Threshold_5,1,0)
- ⊖⊕ Movement_1
= IF(Foraging_1=1,Randomness_Threshold,IF(Returning_1 = 1,MAX(-1,-Nest_Distance_1),0))
- ⊖⊕ Movement_2
= IF(Foraging_2=1,Randomness_Threshold,IF(Returning_2 = 1,MAX(-1,-Nest_Distance_2),0))
- ⊖⊕ Movement_3
= IF(Foraging_3=1,Randomness_Threshold,IF(Returning_3 = 1,MAX(-1,-Nest_Distance_3),0))
- ⊖⊕ Movement_4
= IF(Foraging_4=1,Randomness_Threshold,IF(Returning_4 = 1,MAX(-1,-Nest_Distance_4),0))
- ⊖⊕ Movement_5
= IF(Foraging_5=1,Randomness_Threshold,IF(Returning_5 = 1,MAX(-1,-Nest_Distance_5),0))
- ⊖⊕ Object_Move_1
= IF(Find_Determine_1 = 1,1,IF(At_nest_1 = 1 AND Object_1 = 1,-1,0))
- ⊖⊕ Object_Move_2
= IF(Find_Determine_2 = 1,1,IF(At_nest_2 = 1 AND Object_2 = 1,-1,0))
- ⊖⊕ Object_Move_3
= IF(Find_Determine_3 = 1,1,IF(At_nest_3 = 1 AND Object_3 = 1,-1,0))
- ⊖⊕ Object_Move_4
= IF(Find_Determine_4 = 1,1,IF(At_nest_4 = 1 AND Object_4 = 1,-1,0))
- ⊖⊕ Object_Move_5
= IF(Find_Determine_5 = 1,1,IF(At_nest_5 = 1 AND Object_5 = 1,-1,0))
- ⊖⊕ Return_to_Nest_1
= IF(Foraging_1 = 1 AND (Find_Determine_1 = 1 OR Robot_Energy_Level_1 < 50),1,0)
- ⊖⊕ Return_to_Nest_2
= IF(Foraging_2 = 1 AND (Find_Determine_2 = 1 OR Robot_Energy_Level_2 < 50),1,0)
- ⊖⊕ Return_to_Nest_3
= IF(Foraging_3 = 1 AND (Find_Determine_3 = 1 OR Robot_Energy_Level_3 < 50),1,0)
- ⊖⊕ Return_to_Nest_4
= IF(Foraging_4 = 1 AND (Find_Determine_4 = 1 OR Robot_Energy_Level_4 < 50),1,0)
- ⊖⊕ Return_to_Nest_5
= IF(Foraging_5 = 1 AND (Find_Determine_5 = 1 OR Robot_Energy_Level_5 < 50),1,0)
- ⊖⊕ Robot_Energy_Usage_1
= IF(Foraging_1 = 1 OR Returning_1 = 1,Robot_energy_Usage_Rate,0)
- ⊖⊕ Robot_Energy_Usage_2
= IF(Foraging_2 = 1 OR Returning_2 = 1,Robot_energy_Usage_Rate,0)
- ⊖⊕ Robot_Energy_Usage_3
= IF(Foraging_3 = 1 OR Returning_3 = 1,Robot_energy_Usage_Rate,0)
- ⊖⊕ Robot_Energy_Usage_4
= IF(Foraging_4 = 1 OR Returning_4 = 1,Robot_energy_Usage_Rate,0)
- ⊖⊕ Robot_Energy_Usage_5
= IF(Foraging_5 = 1 OR Returning_5 = 1,Robot_energy_Usage_Rate,0)
- ⊖⊕ Robot_Recharge_1
= IF(In_nest_1 = 1,100-Robot_Energy_Level_1,0)
- ⊖⊕ Robot_Recharge_2
= IF(In_nest_2 = 1,100-Robot_Energy_Level_2,0)

- ⊕ Robot_Recharge_3
= IF(In_nest_3 = 1,100-Robot_Energy_Level_3,0)
- ⊕ Robot_Recharge_4
= IF(In_nest_4 = 1,100-Robot_Energy_Level_4,0)
- ⊕ Robot_Recharge_5
= IF(In_nest_5 = 1,100-Robot_Energy_Level_5,0)
- Energy_Transfer
= IF(Object_Move_1 = -1,1,0)+IF(Object_Move_2 = -1,1,0)+IF(Object_Move_3 = -1,1,0)+IF(Object_Move_4 = 1,1,0)+IF(Object_Move_5 = -1,1,0)
- Find_Determine_1
= IF(RANDOM<Probability_of_Find AND Foraging_1 = 1,1,0)
- Find_Determine_2
= IF(RANDOM<Probability_of_Find AND Foraging_2 = 1,1,0)
- Find_Determine_3
= IF(RANDOM<Probability_of_Find AND Foraging_3 = 1,1,0)
- Find_Determine_4
= IF(RANDOM<Probability_of_Find AND Foraging_4 = 1,1,0)
- Find_Determine_5
= IF(RANDOM<Probability_of_Find AND Foraging_5 = 1,1,0)
- Nest_Energy_Relay
= Nest_Energy_Level
- Robot_Recharge
= Recharge_Scaler*(Robot_Recharge_1+Robot_Recharge_2+Robot_Recharge_3+Robot_Recharge_4+Robot_Recharge_5)
- ◇ Inherent_Nest_Usage_Rate
= .03
- ◇ Nest_Energy_Threshold_1
= 101
- ◇ Nest_Energy_Threshold_2
= 85
- ◇ Nest_Energy_Threshold_3
= 70
- ◇ Nest_Energy_Threshold_4
= 55
- ◇ Nest_Energy_Threshold_5
= 55
- ◇ Probability_of_Find
= .0006
- ◇ Randomness_Threshold
= .33
- ◇ Recharge_Scaler
= .25
- ◇ Robot_energy_Usage_Rate
= .005
- ◇ Value_of_item
= 2

- Correct_State_Estimations
INIT 0
+dt*Correct_States_Rate
- Explored_Terrain
INIT 1
+dt*Effectiv_Terrain_Exploration_Rate
- False_State_Estimations
INIT 0
+dt*False_States_Rate
- False_Victim_Detections
INIT 0
+dt*False_Victim_Dection_Rate
- Found_Victims
INIT 0
+dt*Victim_Discovery_Rate
- Missed_Victims
INIT 0
+dt*Missed_Victims_Rate
- Power
INIT 100
-dt*Power_Consumption
- Terrain_Difficulty_Rating
INIT 1
+dt*Terrain_Difficulty_Adjustment
- Time_to_Power_Loss
INIT 0
+dt*Power_Sensor
- Unexplored_Terrain
INIT 99
-dt*Effectiv_Terrain_Exploration_Rate
- Unfound_Victims
INIT 20
-dt*Victim_Discovery_Rate
- Victims_not_passed
INIT Unfound_Victims
-dt*Victim_Pass_rate
- Victims_Passed
INIT 0
+dt*Victim_Pass_rate
- Correct_States_Rate
= IF(Arbitrator=1,State_Comparator,0)
- Effectiv_Terrain_Exploration_Rate
= IF(Power_left=1,IF(Reset=1,-Explored_Terrain,(Native_Movement_Rate/Terrain_Difficulty_Rating)),0)
- False_States_Rate
= IF(State_Comparator=0,1,0)
- False_Victim_Dection_Rate
= False_Victim_Found
- Missed_Victims_Rate
= IF(Victim_Present=1 AND Victim_Found = 0,1,0)
- Power_Consumption
= IF(Power<0,0,IF(Effectiv_Terrain_Exploration_Rate>0,Power_Consumption_Ratio*
Effectiv_Terrain_Exploration_Rate+Victim_Found_Power_Usage*Arbitrator,0))
- Power_Sensor
= IF(Power>0,1,0)
- Terrain_Difficulty_Adjustment
= IF(Terrain_Difficulty_Rating<1,ABS(Terrain_Adjustment_Level),IF(Terrain_Difficulty_Rating>50,-ABS
(Terrain_Adjustment_Level),Terrain_Adjustment_Level))
- Victim_Discovery_Rate
= IF(Power_left = 1, IF(Victim_Found=1, 1,0),0)
- Victim_Pass_rate
= IF(Reset=1,-(Victims_Passed-Found_Victims),Victim_Present)

- Arbitrator
= IF(Power<0,0,IF((CO2_Sensor+Heat_Sensor+Motion_Digitizer+Reflection_Sensor+Noise_Digitizer> Sensor_Number_Threshold),1,0))
- CO2
= IF(Victim_State>0,1,IF(Rand_4<False_CO2_Probability*Terrain_Exploration_Rate_Relay,1,0))
- CO2_Sensor
= IF(CO2=1 AND CO2_Detection_Probability > Rand_9,1,IF(CO2 = 0 AND False_CO2_Detection_Probability > Rand_9,1,0))
- False_Victim_Found
= IF((Arbitrator=1 AND Victim_Present=0),1,0)
- Heat
= IF(Victim_State>0,1,IF(Rand_5<False_Heat_Probability*Terrain_Exploration_Rate_Relay,1,0))
- Heat_Sensor
= IF(Heat=1 AND Heat_Detection_Probability > Rand_10,1,IF(Heat = 0 AND False_Heat_Detection_Probability > Rand_10,1,0))
- Motion_Digitizer
= IF(Motion_Sensor>0,1,0)
- Motion_Sensor
= IF(Movement=1 AND Small_Motion_Detection_Probability > Rand_14,1,IF(Movement = 2 AND Large_Motion_Detection_Probability > Rand_14,1,IF(False_Large_Motion_Detection_Probability > Rand_14,2,IF(False_Small_Motion_Detection_Probability > Rand_14,1,0))))
- Movement
= IF(Victim_State=1,2,IF(Victim_State=2,1,IF(Victim_State=3,0,IF(Rand_7<Terrain_Exploration_Rate_Relay* False_Large_Movement_Probability,2,IF(Rand_7<Terrain_Exploration_Rate_Relay* False_Small_Movement_Probability,1,0))))))
- Noise
= IF(Victim_State=1,2,IF(Victim_State=2,1,IF(Victim_State=3,0,IF(Rand_8<Terrain_Exploration_Rate_Relay* False_Loud_Noise_Probability,2,IF(Rand_8<Terrain_Exploration_Rate_Relay* False_Quiet_Noise_Probability,1,0))))))
- Noise_Digitizer
= IF(Noise_Sensor>0,1,0)
- Noise_Sensor
= IF(Noise=1 AND Quiet_Noise_Detection_Probability > Rand_13,1,IF(Noise = 2 AND Loud_Noise_Detection_Probability > Rand_13,1,IF(False_Loud_Noise_Detection_Probability > Rand_13,2,IF(False_Quiet_Noise_Detection_Probability > Rand_13,1,0))))
- Power_left
= IF(Power<0, 0, 1)
- Rand
= RANDOM
- Rand_10
= RANDOM
- Rand_11
= RANDOM
- Rand_13
= RANDOM
- Rand_14
= RANDOM
- Rand_2
= RANDOM(0,2)
- Rand_4
= RANDOM
- Rand_5
= RANDOM
- Rand_6
= RANDOM
- Rand_7
= RANDOM
- Rand_8
= RANDOM

- Rand_9
= RANDOM
- Reflection_Sensor
= IF(Reflective=1 AND Reflection_Detection_Probability > Rand_11,1,IF(Reflective = 0 AND False_Reflection_Detection_Probability > Rand_11,1,0))
- Reflective
= IF(Victim_State>0,1,IF(Rand_6<False_Reflective_Probability*Terrain_Exploration_Rate_Relay,1,0))
- Remaining_Victim_Density
= (Victims_not_passed-Unfindable_Victims)/(Unexplored_Terrain-unexplorable_terrain)
- Reset
= IF((Unexplored_Terrain-unexplorable_terrain)<Re_search_cutoff,1,0)
- State_Comparator
= IF(State_Estimator=Victim_State,1,0)
- State_Estimator
= IF(Arbitrator = 1,IF(Motion_Sensor=2 OR Noise_Sensor=2,1,IF(Motion_Sensor=1 OR Noise_Sensor = 1,2,3)),0)
- Terrain_Adjustment_Level
= RANDOM(-Terrain_Variability_Constant,Terrain_Variability_Constant)
- Terrain_Exploration_Rate_Relay
= Effectiv_Terrain_Exploration_Rate
- Victim_Found
= IF((Arbitrator=1 AND Victim_Present=1),1,0)
- Victim_Present
= IF(Terrain_Exploration_Rate_Relay*Remaining_Victim_Density>Rand,1,0)
- Victim_State
= IF(Victim_Present=1,Victim_State_Generation,0)
- Victim_State_Generation
= INT(RANDOM(1,3.9999))
- ◇ CO2_Detection_Probability
= .9
- ◇ False_CO2_Detection_Probability
= .01
- ◇ False_CO2_Probability
= 2
- ◇ False_Heat_Detection_Probability
= .01
- ◇ False_Heat_Probability
= 2
- ◇ False_Large_Motion_Detection_Probability
= .01
- ◇ False_Large_Movement_Probability
= 2
- ◇ False_Loud_Noise_Detection_Probability
= .01
- ◇ False_Loud_Noise_Probability
= 2
- ◇ False_Quiet_Noise_Detection_Probability
= .01
- ◇ False_Quietl_Noise_Probability
= 2
- ◇ False_Reflection_Detection_Probability
= .01
- ◇ False_Reflective_Probability
= 2
- ◇ False_Small_Motion_Detection_Probability
= .01
- ◇ False_Small_Movement_Probability
= 2
- ◇ Heat_Detection_Probability
= .9

- ◇ Large_Motion_Detection_Probability
= .9
- ◇ Loud_Noise_Detection_Probability
= .9
- ◇ Native_Movement_Rate
= .1
- ◇ Power_Consumption_Ratio
= .2
- ◇ Quiet_Noise_Detection_Probability
= .9
- ◇ Re_search_cutoff
= 2
- ◇ Reflection_Detection_Probability
= .9
- ◇ Sensor_Number_Threshold
= 2
- ◇ Small_Motion_Detection_Probability
= .9
- ◇ Terrain_Variability_Constant
= .2
- ◇ unexplorable_terrain
= 15
- ◇ Unfindable_Victims
= INT(INIT(Rand_2)*(INIT(Unfound_Victims))*INIT(unexplorable_terrain)/100)
- ◇ Victim_Found_Power_Usage
= .2