

FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

A REFLECTIVE, 3-DIMENSIONAL BEHAVIOR TREE APPROACH
TO VEHICLE AUTONOMY

By
JEREMY HATCHER

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2015

© 2015 Jeremy Hatcher

Jeremy Hatcher defended this dissertation on April 14, 2015.

The members of the supervisory committee were:

Daniel Schwartz

Professor Directing Dissertation

Emmanuel Collins

University Representative

Peixiang Zhao

Committee Member

Zhenghao Zhang

Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

My most sincere gratitude belongs to Dr. Daniel Schwartz for taking the time to analyze my work and direct me toward its completion. When difficulties arose, his advice and recommendations were concise and helped to overcome any obstacles. I would also like to thank my family, whether immediate, extended, or my church family, who constantly encouraged me and repositioned my sight toward the light at the end of the tunnel. To all of the professors on my committee: Drs. Peixiang Zhao, Zhenghao Zhang, and Emmanuel Collins, I truly appreciate the time you set aside to meet with me and to understand my objectives.

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
LIST OF ACRONYMS	xiv
ABSTRACT.....	xvi
1 INTRODUCTION	1
1.1 Underwater Acoustic Sensor Networks	1
1.2 Underwater Robotic Behaviors – Automation vs. Autonomy	11
1.3 Common Graph Searching Algorithms	14
1.4 Behavior Trees	17
1.5 ECMA-335 and ISO/IEC 23271 - Common Language Infrastructure	31
1.6 ECMA-334 and ISO/IEC 23270:2006 – C#	33
1.7 Modular Programming.....	34
1.8 Case Based Reasoning	39
1.9 XML.....	44
1.10 Reflection.....	51
2 RELATED WORK	54
2.1 Behavior Architectures	54
2.2 Underwater Autonomy.....	60
2.3 Modular Open Systems.....	62
3 DEVELOPING A TEST PLATFORM	73
3.1 Behavior Tree World (BTWorld)	74
4 DEVELOPING WORLD ACTORS.....	81
4.1 Vehicle Simulator (Traveler)	81
4.2 Acoustic Node Simulator (Communicator)	83
5 BEHAVIOR TREE FRAMEWORK IMPLEMENTATION	85
5.1 Fundamental Interfaces	86
5.2 Asynchronous Operation	88
5.3 Composite Nodes	89
5.4 Decorator Nodes	90
5.5 Action / Assertion Nodes	91
5.6 Generic Counterparts	93
5.7 Action and Assertion Repositories.....	95
6 COMPOSING BEHAVIOR TREE ALGORITHMS	100
6.1 Breadth First Search.....	100
6.2 Dijkstra.....	104
6.3 A*.....	107
6.4 Reflective Behavior Tree Instantiation	109

7	BEHAVIOR TREE APPLICATION RESULTS	115
7.1	Path Planning	115
7.2	Acoustic Node Mapping	119
7.3	REMUS Simulator Integration	122
7.4	Performance Cost.....	130
7.5	Algorithm Representation Potential.....	132
8	CONCLUSION AND FUTURE RESEARCH POTENTIAL.....	134
8.1	Conclusion	134
8.2	Future Research Potential	134
	REFERENCES	136
	BIOGRAPHICAL SKETCH	140

LIST OF TABLES

Table 1 - A simple model of an actor in a game of life example.....	40
Table 2 - A simple model of the world in a game of life example	41
Table 3 - A collection of cases in the case-base	41
Table 4 - A CBR query based on current world variables	41
Table 5 – A world actor’s rationale for choosing a case from the case base	42
Table 6 - A modified CBR query taking into account a recent hardy meal.....	43
Table 7 - The resulting case base coefficients after taking the previous query into account	43
Table 8 –XML escape characters.....	46
Table 9 – TDM Usability Questionnaire	59
Table 10 – Alignment of TaskStatus enum to NodeStatus enum	89
Table 11 – Alignment of Boolean values to NodeStatus enum	98
Table 12 - A line by line analysis of the Breadth First Search pseudo code revealing necessary behavior tree nodes	101
Table 13 - A line by line analysis of Dijkstra's algorithm pseudo code revealing necessary behavior tree nodes	105
Table 14 - A line by line analysis of the A* algorithm pseudo code revealing necessary behavior tree nodes	108
Table 15 - Additional nodes needed to enable custom functionality on BTWorld	115
Table 16 –The descriptions of equations needed to solve for the intersection of two circles. ...	121
Table 17 - The underwater vehicle behaviors available for use in the following example	124
Table 18 - The results of an operational time comparison between a sequential A* algorithm, a hardcoded behavior tree representation, and a reflective instantiation of a behavior tree.....	130

LIST OF FIGURES

Figure 1 - Multi-path and refraction causing distorted signals at the receiver in a shallow and deep environment	2
Figure 2 – These are the different types of nodes in an Underwater Acoustic Sensor Network: (a) attached to a gateway buoy (b) attached to an autonomous underwater vehicle (c) resting on sea floor (d) suspended from water craft (e) moored to sea floor and suspended in the column by a flotation device.....	3
Figure 3 – The three basic topologies of an underwater network. (a) centralized (b) distributed (c) multi-hop. Double arrows indicate communication paths.	5
Figure 4 - A basic PC/104 stack http://www.rtd.com/PC104	7
Figure 5 –A REMUS 100 vehicle with associated equipment (from www.km.kongsberg.com)...	8
Figure 6 –A Bluefin Robotics Bluefin-9 vehicle (from www.bluefinrobotics.com).....	9
Figure 7 - Low resolution 900 kHz sonar image compared to a high resolution 1800 kHz image. (Image from oceanexplorer.noaa.gov).....	10
Figure 8 – Left: A sonar image of an interesting object. Right: A closer look at the object with an underwater camera. (Images from NATO Undersea Research Centre)	11
Figure 9 –A typical ‘mow-the-lawn’ type search pattern as performed by an AUV.....	13
Figure 10 - A 10m x 10m land mass broken down into 10 square-cm cells.....	14
Figure 11 - A sample of a modified Breadth First Algorithm used to search an area.	15
Figure 12 - Sample code demonstrating an implementation of Dijkstra's Algorithm	16
Figure 13 - Sample code demonstrating an implementation of the A* algorithm.....	17
Figure 14 - Behavior tree traversal method	18
Figure 15 - A Behavior Tree Node	19
Figure 16 –The fundamental building blocks of a behavior tree: sequence, selector, decorator, action, & assertion.	20
Figure 17 - An example of a sequence success and failure	20
Figure 18 - An example of a selector success and failure.....	21
Figure 19 - An example of a decorator being used as a timer to determine whether or not to call its child node.....	22

Figure 20 - An example of action nodes calling proprietary code.....	22
Figure 21 - An example of an assertion checking if depth is 0 before getting a GPS fix.....	23
Figure 22 - The dining philosophers	24
Figure 23 –A simplified behavior tree solution to the ‘dining philosophers’ problem. Statements with questions marks at the end are assertions and those with exclamation points are actions. ..	24
Figure 24 – The dining philosopher’s behavior trees after one complete traversal.....	25
Figure 25 - Sequence diagram for a positive and negative timer decorator	26
Figure 26 - Complete 'dining philosophers' behavior tree representation	27
Figure 27 - The INode implementation for the examples in this discussion	27
Figure 28 - The possible values of a NodeStatus enumeration.....	27
Figure 29 - An example of a Behavior Tree Sequence	28
Figure 30 - An example of a Behavior Tree Selector Update method.....	29
Figure 31 - An example of a Behavior Tree Decorator Update method.....	29
Figure 32 - A behavior sub-tree representing the actions to get a GPS fix	30
Figure 33 - Sample code that checks for a vehicle's depth	30
Figure 34 - Sample code that commands a vehicle to get a GPS fix	31
Figure 35 - Sample code that commands a vehicle to go to the surface	31
Figure 36 - Hello World in C#.....	33
Figure 37 - An example of a C# interface describing an underwater vehicle.....	34
Figure 38 - An example of an interface describing a vehicle behavior	35
Figure 39- A class titled ‘SearchBehavior’ that takes advantage of the previously defined IVehicle interface to achieve a goal without having a priori knowledge of the interface implementation’s underlying instructions.....	35
Figure 40 - Example of a delegate calling a method to modify a string.....	36
Figure 41 - A class method example alongside its Lambda expression equivalent.....	37
Figure 42 - A Lambda function capturing an outer variable	37

Figure 43 - Using Lambda expressions instead of methods to modify a string.....	38
Figure 44 - Taking advantage of the Action and Func delegates	39
Figure 45 - The four fundamental stages of a case-based reasoning exercise (The four REs).....	40
Figure 46 - An example XML file showing elements, attributes, and comments	45
Figure 47 –A sample XML schema as displayed by Microsoft Visual Studio 11.....	48
Figure 48 - The XML schema describing the relationship of the tables in Figure 26	48
Figure 49 - XML snippet for XPath example (filename xml.xml).....	50
Figure 50 - Utilizing XPath with C# XMLNodes.....	50
Figure 51 - An IVehicle interface implemented by a SampleVehicle class.	51
Figure 52 - An example of a program opening a DLL and inspecting its types and interfaces ...	52
Figure 53 - Instantiating a type / interface from a DLL and utilizing its methods.	52
Figure 54 - The Emotionally GrOunded (EGO) Architecture	54
Figure 55 - Behavior modules organized into conceptual levels.....	56
Figure 56 - Results of Target-Drives-Means Framework Usability Benchmarks (from [35]).....	59
Figure 57 - AUV AVALON Control Architecture	60
Figure 58 –The structure of a JAUS system. The highest level is system and the lowest level is instance. In between are subsystem, node, and component.	64
Figure 59 - Joint Architecture for Unmanned Systems (JAUS) Header that accompanies all JAUS messages	66
Figure 60 –A simple JAUS configuration comprising only a system commander and primitive driver.	67
Figure 61 - A simple JAUS configuration comprising a system commander, reflexive driver, and primitive driver, capable of obstacle avoidance.	68
Figure 62 –A block diagram of the major CARACaS system.....	68
Figure 63 - The R4SA Architecture comprising an Application Layer, a Device Layer, and a Device Driver Layer. The System Layer coordinates the activities of the other layers.	69
Figure 64 –Some of the key components of the device driver layer of R4SA	70

Figure 65 - Some of the key components of the device layer of R4SA.....	70
Figure 66 - Some of the key components of the application layer of R4SA	71
Figure 67 - Some of the key components of the system layer of R4SA	71
Figure 68 - Graphical representation of proposed software framework	73
Figure 69 - A sample terrain (right) generated by a bitmap image (height map, left).....	74
Figure 70 - A close-up look across the horizon of the behavior tree world, showing its 3-dimensional nature.	75
Figure 71 - The terrain.xml file used to configure the BTWorld terrain.	77
Figure 72 - The wireframe mesh representing traversable paths in the Behavior Tree World.....	79
Figure 73 - The contents of the ICell interface	79
Figure 74 - A representation of the Fog-of-War capability of BTWorld. The terrain in the left image is completely unexplored. The terrain on the right has a wide swath cut through it that shows where the vehicle has been.	80
Figure 75 - The Travelers.xml file used to dynamically create 'travelers' at runtime in our Behavior Tree World.....	82
Figure 76 - The contents of Communicator0.xml.....	84
Figure 77 –The fundamental building blocks of a behavior tree: sequence, selector, decorator, action, & assertion. Also known as ‘nodes’	85
Figure 78 - The contents of the interface INode	86
Figure 79 - The contents of the interface IComposite	86
Figure 80 - The contents of the interface IDecorate	86
Figure 81 - A graphical representation of the fundamental interfaces and base classes of the behavior tree framework	87
Figure 82 - A sequence diagram showing a behavior tree's operation whenever RunAsync is set to true	88
Figure 83 - The contents of the Sequence Update method	89
Figure 84 - The contents of the Selector Update method	89
Figure 85 - The Update method of a while-true decorator	90

Figure 86 - The contents of the ExceptionDecorator's Update method	91
Figure 87 - The code behind an ActionNode.....	92
Figure 88 - The code behind an AssertionNode	92
Figure 89 - The instantiation of an ActionNode using a regular class method.....	93
Figure 90 - Suggested interfaces for a generic behavior tree framework	94
Figure 91 - A Generic Behavior Tree Framework.....	94
Figure 92 - The requirements of the IStateNodeArgs interface.....	95
Figure 93 - The Update method implementation of state selector composites.....	95
Figure 94 - A potential implementation of the necessary ActionRepository.....	97
Figure 95 - A potential implementation of the static BTRepositories class.	97
Figure 96 - The additional methods necessary to accommodate assertions in our repository.....	98
Figure 97 - Pseudo code for a breadth first search algorithm	100
Figure 98 - The contents of BTLoopArgs which contains the necessary values for our breadth first search	101
Figure 99 - The GStateComposer's argument that provides the generic section of the behavior tree its operable state.....	102
Figure 100 - A behavior tree representation of a for loop	103
Figure 101 - A behavior tree representation of the Breadth First Search	104
Figure 102 - Pseudo code for Dijkstra's algorithm.....	105
Figure 103 - A behavior tree representation of Dijkstra's algorithm	106
Figure 104 - Pseudo code for our implementation of the A* algorithm	107
Figure 105 - The contents of BTLoopArgs which contains the necessary values for implementation of the A* algorithm	108
Figure 106 - The GStateComposer argument that provides the generic section of the behavior tree its operable state.....	108
Figure 107 - A behavior tree representation of the A* algorithm.....	109
Figure 108 - A list of assemblies necessary to load all of the types in our behavior tree.....	110

Figure 109 - A list of Types necessary craft our behavior tree.....	111
Figure 110 - The XML representation of a Breadth First Search behavior Tree. This XML code is used by the reflective loader to instantiate the tree.....	112
Figure 111 - The XML representation of Dijkstra's algorithm behavior Tree. This XML code is used by the reflective loader to instantiate the tree.....	113
Figure 112 - The XML representation of the A* behavior Tree. This XML code is used by the reflective loader to instantiate the tree.....	114
Figure 113 - The CanCross functionality needed for the Breadth First Search to check nodes for non-traversable conditions	116
Figure 114 - A modified GStateComposer argument that sets the Breadth First Search current node to yellow.....	117
Figure 115 - The MarkNeighbor functionality needed for the Breadth First Search to give color coded feedback to users	117
Figure 116 - The modified sequence necessary to add the CanCross and MarkNeighbor functionality to Dijkstra's algorithm	117
Figure 117 - The modified sequence necessary to add the CanCross and MarkNeighbor functionality to the A* algorithm.....	118
Figure 118 - The A* algorithm performing a search with fog of war enabled (left) and disabled (right)	118
Figure 119 - The A* path planning algorithm providing a path through unexplored water (left). Once explored it plans around it properly (right)	119
Figure 120 - The makeup of the BTLocator component used to localize nodes	120
Figure 121 - A graphical depiction of solving for the intersection of two circles.	121
Figure 122 - When observed from top left to bottom right (1) The vehicle begins a mow the lawn pattern around the map (2) The vehicle gets a message from the static node and builds a circle using the center point of the received communication and the distance received. The vehicle begins randomly traveling inside of this circle (3) The vehicle receives a second communication from the static node. It will now calculate intersections and check for the static node at those locations. It initially searches the wrong intersection (4) The vehicle checks the second intersection location where it will find the static node and move into the 'complete' state.	122
Figure 123 - Entering the actions and assertions into the demo repository	125
Figure 124 - Behavior tree sketch for the REMUS waypoint following test.....	126
Figure 125 - The Behavior Tree Factory returning a waypoint following behavior tree	127

Figure 126 - WaypointFollowing.txt: The waypoint following demo behavior tree file output	128
Figure 127 – The vehicle simulator output screen after a successful experiment with the proposed waypoint following behavior tree	129
Figure 128 – A chart representing the data shown in Table 18	131
Figure 129 - A graphical representation of an if-else-if structure using behavior tree nodes	133
Figure 130 - A behavior tree representation of a for loop	133

LIST OF ACRONYMS

AI	Artificial Intelligence
ASF	Altitude Above Sea Floor
ASL	Altitude Above Sea Level
AUV	Autonomous Underwater Vehicle
CARACaS	Control Architecture for Robotic Agent Command and Sensing
CASPER	Continuous Activity Scheduling Planning Execution and Re-planning
CBR	Case-based Reasoning
CIL	Common Intermediate Language
C ⁴ ISR	Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COTS	Commercial off the Shelf
CTS	Common Type System
DAG	Directed Acyclic Graph
DoD	Department of Defense
ECMA	European Computer Manufacturers Association
HTML	Hyper Text Markup Language
ISA	Industry Standard Architecture
ISO	International Organization for Standardization
JAUS	Joint Architecture for Unmanned Systems
JSF	Joint Strike Fighter
MAC	Medium Access Control
MOSA	Modular Open Systems Approach

OSA	Open Systems Architecture
OSJTF	Open Systems Joint Task Force
PCI	Peripheral Component Interconnect
PCMCIA	Personal Computer Memory Card International Association
PID	Proportional-Integral-Derivative
R4SA	Robust Real-Time Reconfigurable Robotics Software Architecture
REMUS	Remote Environmental Measuring UnitS
SAE	Society of Automotive Engineers
TCP	Transmission Control Protocol
TDOA	Time Difference Of Arrival
TOF	Time Of Flight
UASN	Underwater Acoustic Sensor Network
UDP	User Datagram Protocol
UUV	Unmanned Underwater Vehicle
UW-ASN	Underwater Acoustic Sensor Network
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

ABSTRACT

Many of today's underwater vehicles have a limited set of pre-planned behaviors that are of varying utility. This is due, in part, to very low underwater communication rates and difficulties observing the vehicle's underwater behavior pattern. The video game industry is a multi-billion dollar enterprise constantly investing in high quality, high performance frameworks for controlling intelligent agents. One such framework is called Behavior Trees. This project proposes a novel autonomy framework enabling easily reconfigurable behaviors for both land based and underwater vehicles to discover and map acoustic nodes using a modular open systems approach based on behavior trees and action repositories.

CHAPTER 1

INTRODUCTION

1.1 Underwater Acoustic Sensor Networks

Difficulties encountered with the underwater transmission medium have subdued widespread underwater research for many years. In fact, 95% of the world's underwater realm remains unexplored [1]. Advancements in digital signal processing (DSP) technology that enable high-rate, reliable communications have enabled scientists and engineers to set up underwater acoustic networks for monitoring and wide-area communication applications [2]. Scientific journals have produced a wealth of information regarding acoustic layer protocols designed at optimizing point-to-point communications among nodes. One notable example of this is *Seaweb* (see section 1.1.2).

1.1.1 Underwater Communication

Underwater communications pose problems to researchers less evident in terrestrial networks. Lower bandwidth, extended multi-path (Figure 1), and large Doppler shifts all contribute to the already complex issue [3]. Also, MAC layer conflicts¹ in the air are usually resolved faster than a human can comprehend. Underwater signal conflicts are very time consuming to rectify and waste precious battery life in the recovery due to the much slower propagation of carrier waves.

Underwater networks cannot use radio frequency communication methods due to the very high absorption rate of these frequencies in water. Even high powered lasers are completely absorbed

¹ MAC layer conflicts arise when multiple transmitters attempt to transmit simultaneously.

within 1 km [4] and are thus unsuitable for sparse underwater networks. Currently, only acoustic signals provide the range and bandwidth necessary for their communication [5]. These signals, traveling roughly 1500 meters per second (varying according to pressure, temperature, and salinity), provide lower bandwidth, higher latency, and strict energy constraints. Trade-offs between data rate and communication distance have guided many commercial underwater products into the range of 7 kHz and 30 kHz providing data rates between 80 and 38400 bits per second up to 6 km [6] [7] [8] [9]. Higher data rates are generally achieved through focusing of the beam direction. This would be unsuitable for searching nodes in unknown locations, however. Omni directional beams query in all directions but suffer more from signal attenuation.

Underwater multi-path occurs when a loosely focused transmission is received after taking different paths to the receiver. These paths are due mainly to reflections at the surface and the bottom, as well as refraction in the water, mainly at greater depths [10]. These divergent, faded signals arrive at the receiver out of phase with the original signal and may cause reinforcement or cancellation.

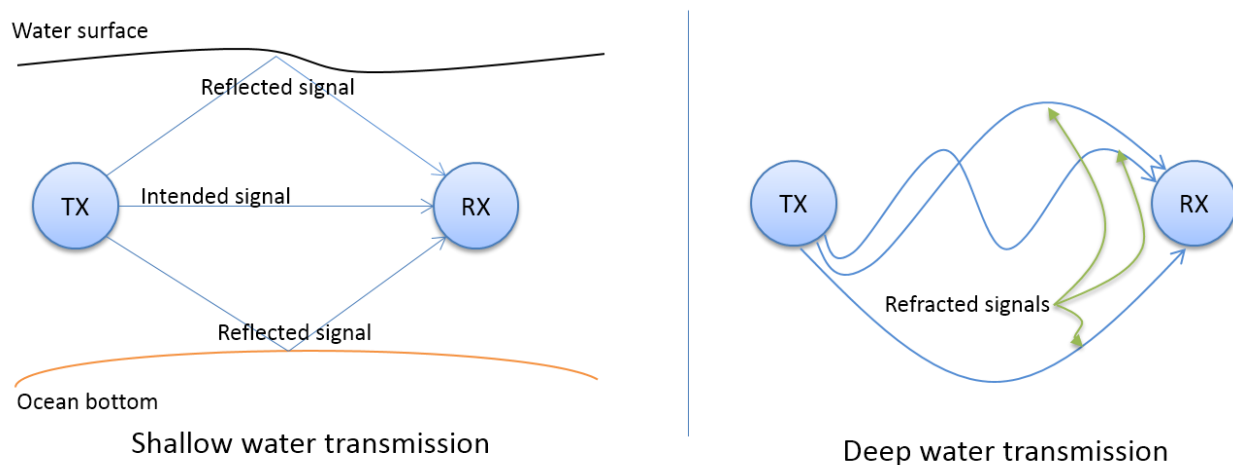


Figure 1 - Multi-path and refraction causing distorted signals at the receiver in a shallow and deep environment

1.1.2 Seaweb

Seaweb is an underwater wireless networking concept conceived in the late 90s by the Space and Naval Warfare Systems Center, San Diego, comprising a scalable number of underwater stationary nodes, peripherals, and gateway buoys [11] that employ spread spectrum modulation for asynchronous multiple access to the physical medium². Its original intent was for naval command, control, communications, computers, intelligence, surveillance, and reconnaissance (C⁴ISR). Several experiments on commercial off the shelf (COTS) telesonar modems have proven

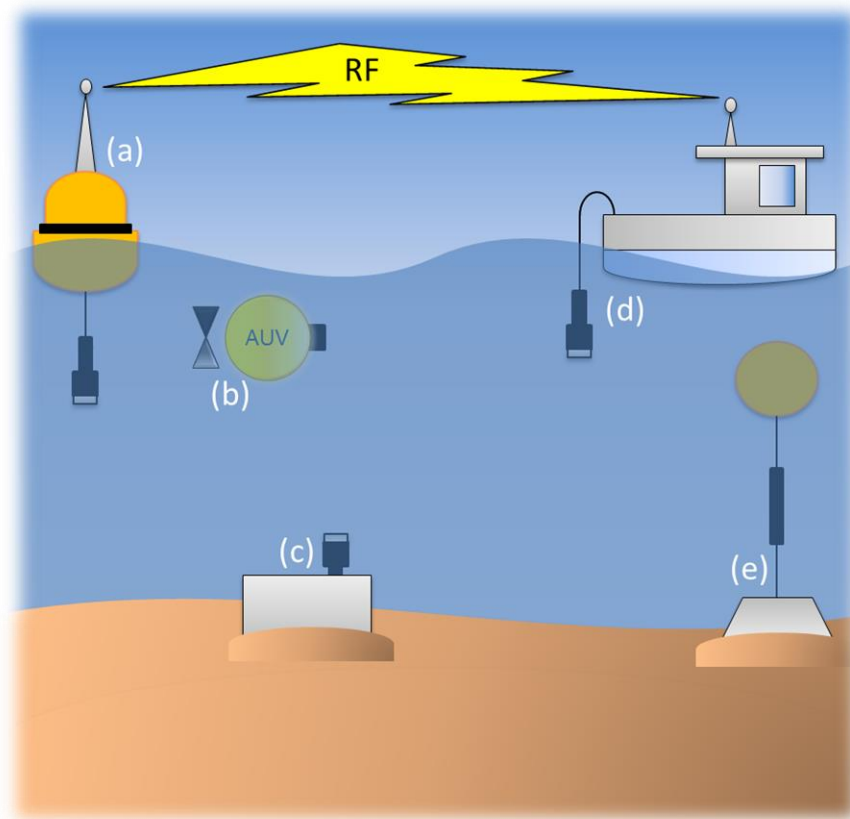


Figure 2 – These are the different types of nodes in an Underwater Acoustic Sensor Network: (a) attached to a gateway buoy (b) attached to an autonomous underwater vehicle (c) resting on sea floor (d) suspended from water craft (e) moored to sea floor and suspended in the column by a flotation device

² In this case, the physical medium is the water.

the capabilities of the program to self-organize message routes and optimize modulation parameters for each message in the clear-to-send (CTS) response of a node.

Figure 2 shows several of the different types of underwater communication nodes. Stationary nodes may be placed directly on the seafloor or moored. Moored nodes are anchored to the seafloor and suspended in the water column by a buoy so as to maintain a certain altitude above sea floor (ASF). They provide more reliable communication but are also more susceptible to damage by passing ships, curious passersby, or vandals. Mobile nodes may be of the moored type, with mechanisms to raise or lower depth, or an unmanned underwater vehicle (UUV) type, capable of traversing the node field and returning to a retrieval point. Nodes may also be directly attached to floating buoys or to small boats. A buoy mounted node, like moored nodes, introduces more risk into the design [12].

1.1.3 Localization

The localization of nodes underwater is a difficult task due to the high sensitivity of acoustic channels to multi-path, Doppler shift, refraction, and, among other things, extremely low bandwidth. Research efforts are ongoing and are applied at the acoustic modem level of a design. Modern underwater modems, like the WHOI MicroModem and Benthos 900 modems, are very effective at quickly determining the distance between two nodes. However, having the simple range between two underwater nodes is insufficient to provide accurate localization. Techniques must be employed to handle the three dimensional nature of the underwater environment. Multilateration is a range-based localization scheme where sensor nodes measure distances to known nodes, or anchors, by signal time-of-flight (TOF) [13]. Since the underwater domain is three dimensional, 4 sets of linearly independent readings are required to solve the equation: $(x -$

$x_i^2 + (y - y_i)^2 + (z - z_i)^2 = d_i^2$. Since many nodes can determine their depth (z component) from onboard pressure sensors, ranges between nodes can be projected onto a single two dimensional plane and well established 2D localization schemes can be used instead [14].

1.1.4 Topologies

There are three basic types of underwater network topologies: Centralized, distributed, and multi-hop [2] (see Figure 3). Centralized topologies rely on one or more ‘master’ nodes that direct all of the other nodes. Both distributed and multi-hop topologies rely on a peer-to-peer connection through direct communication or through a network route, respectively. Centralized topologies afford a great deal of control over the transmission medium by directing which nodes should speak and when. However, they also introduce critical nodes which can shut down very large portions of a network given only a single failure. They are also very limited in scale. The nominal range of an acoustic transponder is 6 kilometers [15].

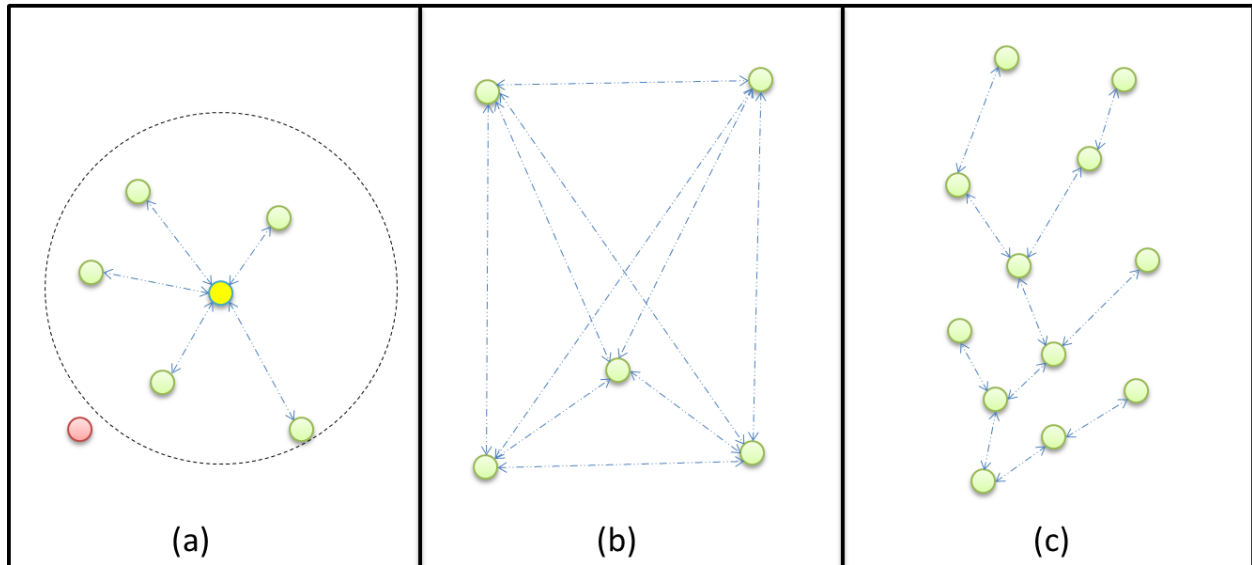


Figure 3 – The three basic topologies of an underwater network. (a) centralized (b) distributed (c) multi-hop. Double arrows indicate communication paths.

1.1.5 Lost Nodes

When underwater sensor networks are deployed for extended lengths of time, possibly several years, lost nodes become more and more likely. Underwater currents have a tendency to shift the contents of the seafloor and thereby move the nodes with it. In the presence of a hurricane, it is highly likely that previously placed nodes will be displaced by several meters and perhaps in a condition unfavorable to receive or transmit acoustic signals. Regardless of the underwater routing protocol chosen for this network topology, other underwater sensors will have to be utilized to acquire and identify the lost node.

1.1.6 Mobile Nodes

While terrestrial networks may rely on densely packed, inexpensive nodes, a more economical underwater approach must be pursued. Underwater sensor nodes are very expensive and the desired coverage space is usually very large [16]. Since the mid-1950s, mobile nodes, specifically autonomous underwater vehicles, have helped to overcome many of the traditional underwater network problems such as battery life and a poor communication path.

POWER

Undersea modems usually rely on batteries for their power and recharging or replacing them is not trivial. Since many underwater acoustic sensor networks (UASNs) are in place for years at a time, energy consumption must be closely monitored. Also, since transmit power is typically 100 times greater than receive power [16], transmit time should be severely limited. On the other hand, AUV power usage is dominated by propulsion efforts. Their batteries, however, can be easily recharged or replaced at an underwater docking station or at surface maintenance area.

COMPUTING

A typical underwater vehicle will contain several PC/104 stacks (see Figure 4) which is the standard for compact embedded pc modules. Each board performs a unique capability required for the system to operate as a whole. For example, in Figure 4, one board is the main CPU, while others may be SATA drive controllers, analog I/O boards, Ethernet control boards, or PCMCIA adapters. These stacks are desirable due to their rigid use of standard measurements, connections, and interoperability. The PC/104 specifications are maintained by the PC/104 Consortium³.



Figure 4 - A basic PC/104 stack
<http://www.rtd.com/PC104>

MANEUVERABILITY

Poor communications paths may be due to natural obstructions or poor placement of the transducer relative to the desired transmission direction. AUVs freely overcome this obstacle simply by

³ Visit the PC/104 Consortium website at <http://www.pc104.org/>

repositioning themselves in the most optimal location for communicating. Engineers all over the world dedicate themselves to improving the art of underwater vehicle design. Worthy of note are two vehicles used heavily by the United States Navy for myriad underwater missions. The Remote Environmental Measuring UnitS (REMUS) vehicle was developed by the Woods Hole Oceanographic Institution and is manufactured by Hydroid. The Bluefin vehicle class shares a name with its developer, Bluefin Robotics.

1.1.6.1 REMUS

REMUS vehicles are designed at Woods Hole Oceanographic Institute's Oceanographic Systems Lab and come in variants 100, 600, 3000, and 6000. These numbers indicate the max depth in meters at which the vehicle is rated to operate [17]. The REMUS 100 is a light vehicle, weighing only about 100 pounds depending on sensor configuration, and can be launched easily from a small boat with just 2 people. This makes it suitable for quick, low cost mapping and characterization surveys of up to 10 hours [18]. Available payload sensors include a dual 900/1800 kHz side scan sonar and a forward looking camera.



Figure 5 –A REMUS 100 vehicle with associated equipment (from www.km.kongsberg.com)

The REMUS vehicle comes with a control program called the Vehicle Interface Program (VIP). It is a PC based graphical interface program that issues user commands to the vehicle and displays the vehicle's internal status back to the user. It performs both pre- and post-mission analysis. Missions are programmed through a scripting language and are limited to a few preprogrammed behaviors. These small behaviors can be cobbled together to form more complex behaviors. The built in behaviors favor reliability and repeatability over finesse and as such do not provide any dynamic or reactive decision making capability to the user. Some variants of the REMUS include an onboard embedded computer capable of overriding the main computer's control. This board is networked into the vehicle's onboard switch and gives behavior developers the ability to create far more complex behaviors but with strict controls built in. There is both a time and a distance constraint on each override instance. One obvious rationale for an override would be for an obstacle avoidance controller. Given forward looking sonar capabilities the vehicle would detect upcoming objects and make adjustments to travel over or around them before coming back to the preprogrammed track.

1.1.6.2 Bluefin



Figure 6 –A Bluefin Robotics Bluefin-9 vehicle (from www.bluefinrobotics.com)

Bluefin Robotics develops and supports an array of vehicles known as Bluefin-9, 12, and 21. The number indicates the diameter of the vehicle. The Bluefin-9 is a small, highly maneuverable vehicle weighing about 140 pounds. This makes it a viable candidate for quick, low cost mapping and characterization just like the REMUS 100. With the added weight come slightly higher specs than the REMUS 100. The Bluefin-9's stated max speed is 5 knots, it can reach depths of 200 meters, and its endurance is about 12 hours at 3 knots [19].

1.1.6.3 Search Based Sensors

Typical underwater vehicles are equipped with sidescan sonars. These devices use sound instead of light to map surroundings. While hardly comparable to modern high-definition cameras, images from these devices can be analyzed and determined to be of further interest, in which case a vehicle can be redeployed and use a higher resolution underwater camera to capture a photo of the object of interest. The REMUS 100 uses a dual 900/1800 kHz sidescan for low resolution, long range imaging and high resolution, short range imaging, respectively.

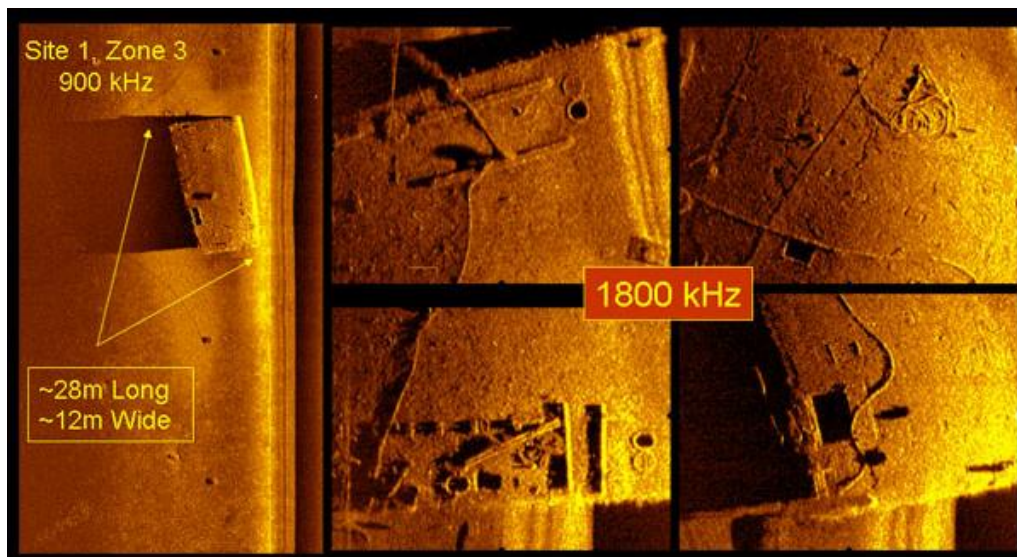


Figure 7 - Low resolution 900 kHz sonar image compared to a high resolution 1800 kHz image. (Image from oceanexplorer.noaa.gov)

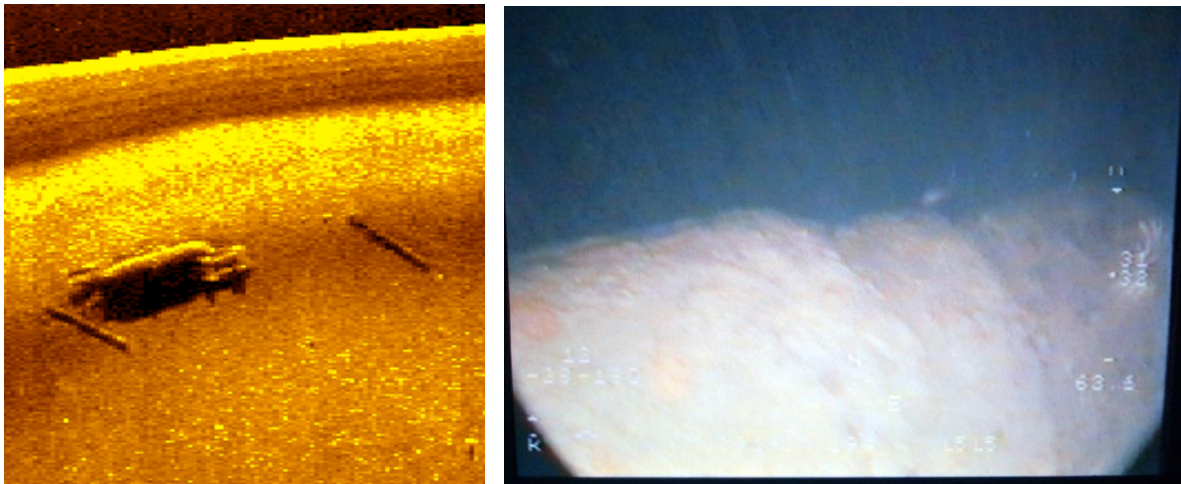


Figure 8 – Left: A sonar image of an interesting object. Right: A closer look at the object with an underwater camera.
(Images from NATO Undersea Research Centre)

1.2 Underwater Robotic Behaviors – Automation vs. Autonomy

Underwater missions are inherently difficult to observe. This puts programmers of underwater vehicle control algorithms under heightened pressure to incorporate vehicle safety into their behaviors. In many cases this has led, understandably, to a very limited set of highly automated yet severely restrictive underwater vehicle behaviors. These behaviors are typically preplanned, scripted movements with sections of code operating in sequential order until mission completion. Very little, if any room for vehicle borne decision making, or autonomy, is allowed. Some of the tension involved with allowing vehicles to operate out of sight and autonomously can be relieved by viable communication techniques to maintain vehicle location and mission status. One such implementation is the Portable Acoustic RADIo Geo-referenced Monitoring system (PARADIGM). It was developed by the Woods Hole Oceanographic Institution to satisfy these vehicle navigation and tracking requirements [20]. The implementation of this system requires radio buoys, acting as anchor nodes, to be deployed. Underwater vehicles acting within the radius of the buoys' communication range (2 km to 6km) can 'ping' them and determine its location and make navigational corrections. Similarly, the buoy may ping the vehicle to determine its location

and report it back to a user's tracking station. While effective in controlled areas, discreet operations require more discreet means of localization. Onboard GPS antennas allow surfaced vehicles to gather their location quickly before submerging and operating via inertial measurement units. These vehicles can then discreetly send their positions acoustically to a listening node attached to the mission control station.

One reason for the slant towards automation instead of autonomy is that sophisticated robotic systems can be very expensive. For example, even a lightly equipped REMUS 100 vehicle from Hydroid is several hundred-thousand dollars. It therefore becomes a major component of every mission to protect the device from harm or loss. With that in mind, the REMUS control software developers have placed strict limits on many of the vehicle's capabilities by only allowing a few very limited behaviors to be performed. These behaviors are formed mostly by commanding the vehicle to follow multiple straight line combinations to form a more complex pattern. For example, a mow-the-lawn pattern is formed by giving the vehicle a rectangular area to search and a track spacing parameter. The track spacing will usually be determined by the swath width of the sidescan sonars. If the sidescan sonar can map the ground thirty feet in both directions then, depending on the amount of overlap desired, a track spacing of sixty feet might be chosen as shown in Figure 9. These types of behaviors are usually employed to do a quick, economical search of a particular area. If something of interest is found in the returned images the vehicle may be sent out to further investigate. These investigations are usually performed at a higher resolution and the vehicles will typically make multiple passes at the intended target to ensure that a decipherable image is returned.

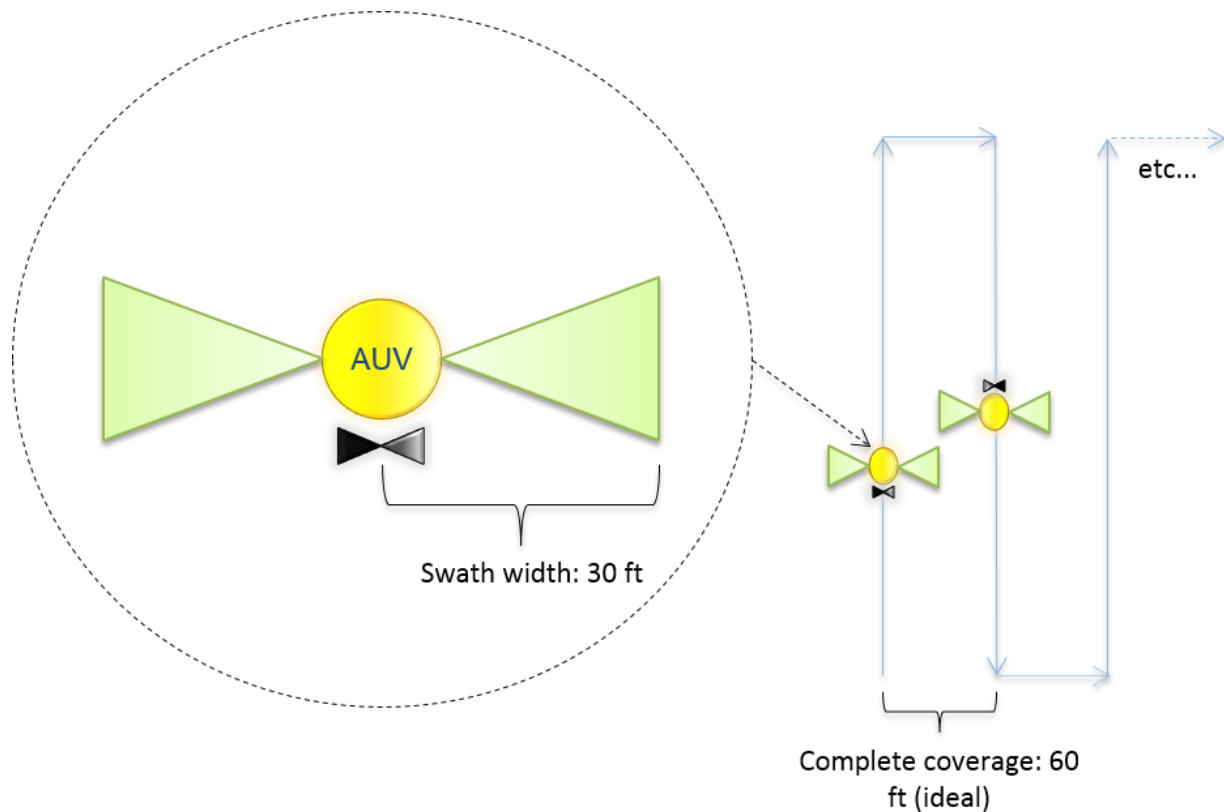


Figure 9 –A typical ‘mow-the-lawn’ type search pattern as performed by an AUV.

To accommodate underwater network research, the engineers at Hydroid have provided a ‘back door entry’ into the REMUS’s controller called the Recon protocol. This override functionality allows a behavior developer a much finer degree of control over the vehicle’s speed, depth, and heading thus allowing the implementation of user defined autonomous behaviors onboard the REMUS vehicle. It is also, however, bound by time and distance restrictions.

The typical preplanned missions focus mainly on area of coverage and vehicle safety. Communications do take place but are usually secondary objectives. Most information is stored onboard and retrieved after mission completion.

1.3 Common Graph Searching Algorithms

Utilizing computer algorithms to search a continuous space requires partitioning of that space into discrete chunks that are operable by those algorithms. Typically, large areas are broken down into grids of varying cell sizes. These cells represent the aggregate data of the land contained within that cell (e.g. land type or elevation – see Figure 10). In this way, an algorithm might break up a 10 meter by 10 meter sized land mass into 10,000 – 10 square-centimeter cells. Then, in a logical manner, proceed to search each one individually for relevant information about it. This information might lead to the detection of an object being searched for or simply a path from one cell to another. In this section we will discuss three closely related methods for searching these discrete cells: Breadth First, Dijkstra, and A*.

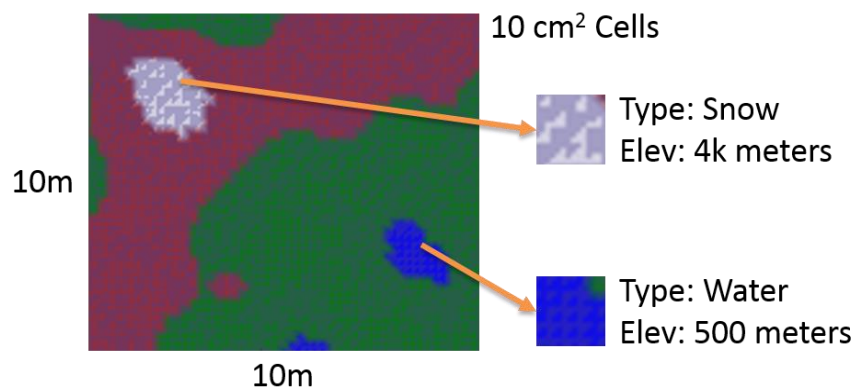


Figure 10 - A 10m x 10m land mass broken down into 10 square-cm cells

1.3.1 Breadth First Search

The Breadth First Search is not strictly a path finding algorithm but rather a very thorough approach to searching through a graph of cells. With only a minor addition to the process during each iteration it is guaranteed to provide the shortest Manhattan Distance between two of our cells. To begin, it is provided with a starting cell which it annotates. Then, it gathers all of that cell's

neighbors into a queue. After gathering the information needed from the starting cell it pops the next cell off of the queue where the process starts all over again. When the traversal is complete, the shortest path from any cell on the map to the starting cell is known. Sample code for this simple algorithm is shown in Figure 11.

```
private void BreadthFirstSearch(Cell start, Cell end)
{
    _frontier.Enqueue(start);
    while (_frontier.Count > 0)
    {
        Cell current = _frontier.Dequeue();
        List<Cell> neighbors = GetNeighbors(current);
        for (int i = 0; i < neighbors.Count; i++)
        {
            Cell neighbor = neighbors[i];
            if (neighbor.Visited == false)
            {
                _frontier.Enqueue(neighbor);
                neighbor.Visited = true;
                neighbor.CameFrom = current;
            }
        }
    }
}
```

Figure 11 - A sample of a modified Breadth First Algorithm used to search an area.

1.3.2 Dijkstra

Dijkstra's algorithm was created by Edsger Dijkstra in 1956 and published in 1959 in *Numerische Mathematik* [21]. The algorithm itself runs very similarly to the modified Breadth First Search algorithm proposed in section 1.3.1. Dijkstra, however, also tracks distances from one cell to another. In our simple terrain example of Figure 10 this might mean that travelling uphill might cost more than travelling downhill. Similarly, travelling across water might be much more costly than travelling across flat ground or impossible altogether. With a little bit of ingenuity, influences can be placed on cells that cause a travelling vehicle to avoid locations due to the necessity of concealment or other mission requirements. A sample code block is shown in Figure 12. The frontier collection is now a list sorted by priority. That means that the search will continue along

the currently shortest path. With an extra line of code that causes the search to terminate when the goal is found, we can begin executing vehicle movement much quicker.

```
private void DijkstraSearch(Cell start, Cell end)
{
    _frontier.Add(0, start);
    while (_frontier.Count > 0)
    {
        Cell current = _frontier[0];
        _frontier.RemoveAt(0);
        if (current == end) break;
        List<ICell> neighbors = GetNeighbors(current);
        for (int i = 0; i < neighbors.Count; i++)
        {
            ICell neighbor = neighbors[i];
            float newCost = CalculateDistance(current, neighbor);
            if (neighbor.Visited == false || newCost < neighbor.DistanceSoFar)
            {
                _frontier.Add(newCost, neighbor);
                neighbor.DistanceSoFar = newCost;
                neighbor.Visited = true;
                neighbor.CameFrom = current;
            }
        }
    }
}
```

Figure 12 - Sample code demonstrating an implementation of Dijkstra's Algorithm

1.3.3 A*

The A* algorithm runs much like Dijkstra's algorithm. The difference is in the way the algorithms prioritize which cell to traverse next. In Dijkstra's algorithm, the cells are prioritized in the frontier according to the currently traversed distance. In A*, the cells are placed in the frontier by that same value plus a heuristic. This heuristic may be determined by any group implementing the algorithm but a commonly used one is the straight line distance between the current cell and the goal cell. For example, if the algorithm has travelled through a number of cells equaling 3 meters so far and the next neighbor in line is 3.5 meters away from the goal, then a value of 6.5 is used as the priority. This cell will be searched before another neighbor, whose current distance is only 1, but whose straight line distance to the goal is 6 (i.e. priority 7). Sample code for the A* algorithm is shown in Figure 13.


```

private void AStarSearch(Cell start, Cell end)
{
    _frontier.Add(0, start);
    while (_frontier.Count > 0)
    {
        ICell current = _frontier.ElementAt(0).Value;
        _frontier.RemoveAt(0);
        if (current.Visited == false)
        {
            current.Visited = true;
            if (current == end) break;
            List<Cell> neighbors = GetNeighbors(current);
            for (int i = 0; i < neighbors.Count; i++)
            {
                Cell neighbor = neighbors[i];
                if (neighbor.Visited == false)
                {
                    float dist = CalculateDistance(current, neighbor);
                    float newCost = dist + current.DistanceSoFar;
                    if (newCost < neighbor.DistanceSoFar)
                    {
                        float priority = newCost + Heuristic(neighbor, end);
                        _frontier.Add(priority, neighbor);
                        neighbor.DistanceSoFar = newCost;
                        neighbor.CameFrom = current;
                    }
                }
            }
        }
    }
}

```

Figure 13 - Sample code demonstrating an implementation of the A* algorithm

1.4 Behavior Trees

Software behavior trees are a programming paradigm that came to the forefront of video game AI in 2005 through a talk given by Damian Isla at the Game Developer's Conference (GDC) 2005. Damian was involved in the AI development for the extremely popular game Halo 2 for XBOX. His talk began by addressing common issues when dealing with AI: poor run-time, poor scalability, a lack of direct-ability, and random vs. intentional behaviors. It then progressed through ways to address each problem. It concluded with the statement that, "hard problems can be rendered trivial through judicious use of the right representation," which, in this case, was a behavior tree. The concept worked so well that it was reused in the development of Halo 3, another very popular title for the Xbox 360.

Behavior trees are organized as directed acyclic graphs. During each update cycle the graph is traversed from top to bottom and left to right (Figure 14). During this traversal each node will assume one of a number of node states. These will typically comprise ‘success’ and ‘failure,’ along with any others that the designer feels necessary (e.g. running, uninitialized, etc...). A possible NodeStatus enumeration will be shown in section 1.4.8. Each successful traversal represents one time slice and constitutes a performance parameter of the system which can be adjusted up or down. For example, a video game developer may shoot for one complete traversal every 16ms (60 frames / second) while a vehicle autonomy developer may or may not require such a high traversal rate.

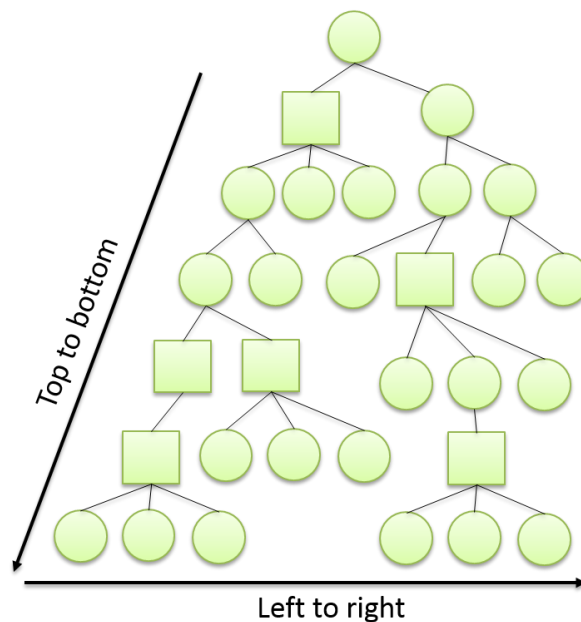


Figure 14 - Behavior tree traversal method

Inner nodes of the graph can be any of a collection of sequences or selectors, called composites, and decorators. Leaf nodes are conditions (assertions) or actions. Composites contain multiple child nodes and will generally assume the NodeStatus of their last running child node but are not

required to do so. Decorators, too, will typically assume their child's status but a designer may use any clever method to set it differently. For example, a decorator may run a successful child node and instead return failure due to the length of time that the node took to run.

The utility of behavior trees only manifests itself if strict interfaces are followed. This means that as long as developers adhere to the standard usages, collections of behaviors can easily be developed and used across a wide range of applications. The easiest way to understand this will be through the use of an example followed by a sample code implementation. But first, we will define basic behavior tree terminology.

1.4.1 Nodes

The atomic unit of a behavior tree is a node. It will contain at a minimum one public function, `Tick()`, which will return a `NodeStatus`. A node's `NodeStatus` will indicate whether the update was a success or failure. The necessity and utility of such a simple interface will become evident as we proceed through our example.

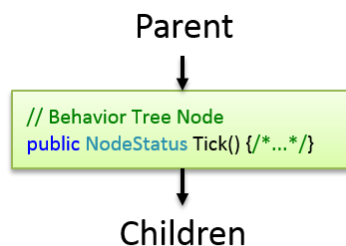


Figure 15 - A Behavior Tree Node

The implementation of a node can take the form of a sequence, selector, decorator, action, or assertion. Sequences and selectors are called composite nodes. This is because they will contain

multiple child nodes. Decorators will always contain only one child node. Actions and assertions will be leaf nodes and contain zero children. The unofficial but generally accepted depiction of these blocks is shown in Figure 16. When placed in a tree, each of these blocks is still considered a node. Simple modifications to the makeup of a behavior tree can cause many different behaviors to emerge.

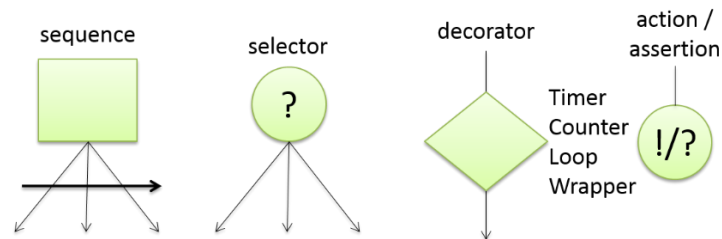


Figure 16 –The fundamental building blocks of a behavior tree: sequence, selector, decorator, action, & assertion.

1.4.2 Sequences

Sequences are like database transactions. They run their children one after the other and either pass or fail as a group like a logical AND statement. If a sequence fails to complete during an update, the last running node is called immediately on the next update. There are special variations of sequences called sequence loops. These variants run in order but instead of reporting to their parent they continue to restart execution at the first child node for a set number of iterations.

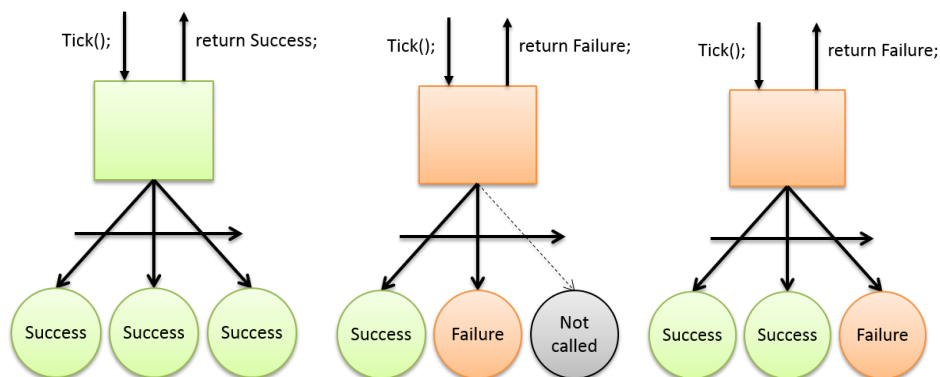


Figure 17 - An example of a sequence success and failure

1.4.3 Selectors

Selectors are the complement of sequences and act like logical OR statements. Each child of a selector is tested in order. Failures are ignored but the first success is reported back up the tree, short circuiting subsequent nodes. Priority selectors are a special case in which each node is assigned a priority and run in descending priority order. During each traversal of the graph, higher priority selectors may override the continuation of a lower priority node. Random selectors are another special case in which a child node is chosen at random. This equips the behavior tree with more variety of character.

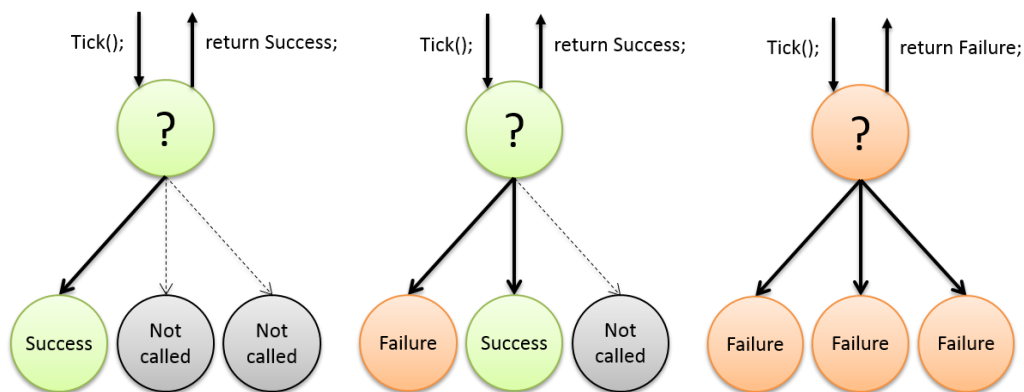


Figure 18 - An example of a selector success and failure

1.4.4 Decorator Nodes

Decorator nodes contain only one child and are used to enforce certain constraints. They might contain counter variables to maximize the number of times a behavior will run, a loop to ensure a certain number of runs, a timer to enforce time lengths in-between runs, or code to handle exceptions. If resources are shared, a decorator node is a natural place to put locks, mutexes, and semaphores. In Figure 19, the image on the right shows how a timer can return failure without running its child node since its last activation was too recent.

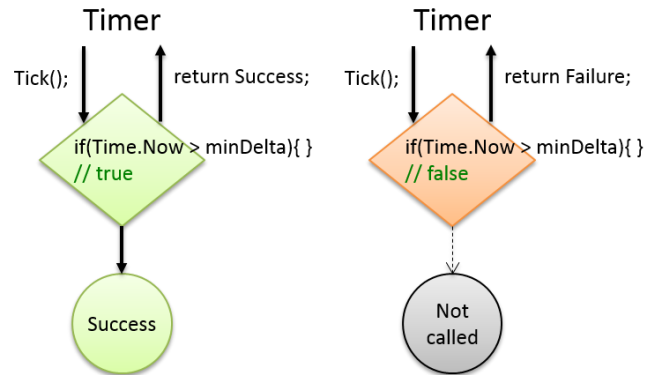


Figure 19 - An example of a decorator being used as a timer to determine whether or not to call its child node

1.4.5 Actions

Actions, as opposed to composite sequences and selectors, cause the actor to effect a world change. Outside of a few common use cases, most of these nodes will be unique to a software project and are what differentiate it from others. Many times these nodes will make calls into a proprietary library like a vehicle control class. In an underwater mission project, this might be to communicate with a network sensor, plan a path to collect sonar images, or get a GPS fix.

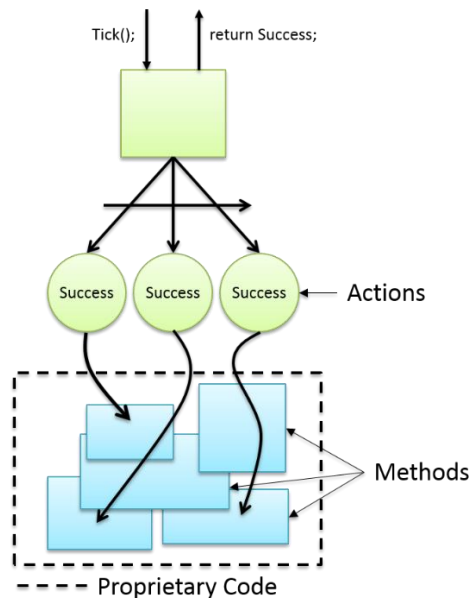


Figure 20 - An example of action nodes calling proprietary code

1.4.6 Assertions

Assertions ensure that particular world states hold true before allowing its subsequent peer nodes to run. These would be placed in the hierarchy just prior to an action or another assertion, thus allowing or preventing the action to be executed. Like actions, these nodes will be uniquely coded for a particular project and will probably make method calls into proprietary libraries.

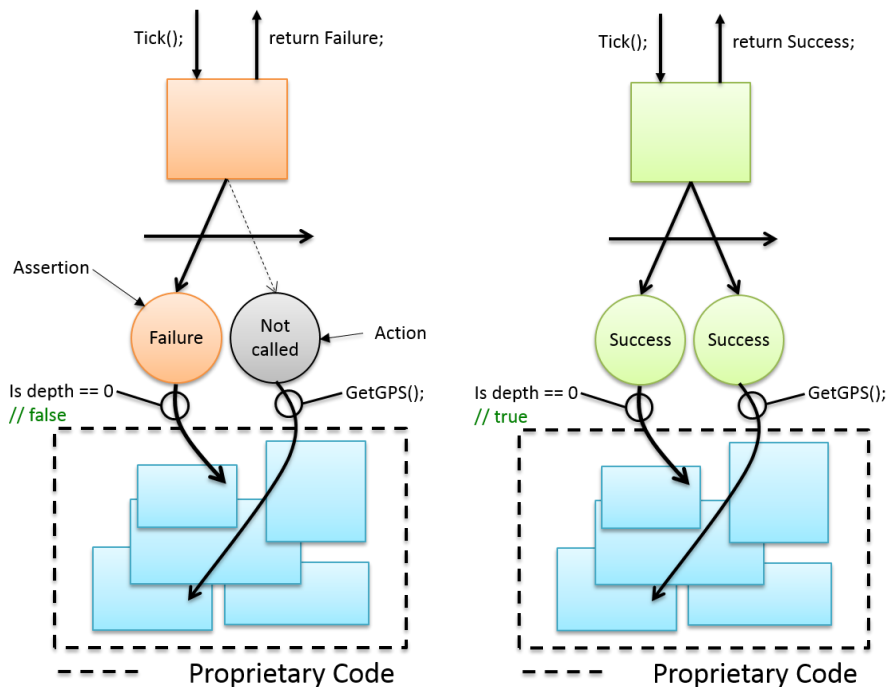


Figure 21 - An example of an assertion checking if depth is 0 before getting a GPS fix

1.4.7 Behavior Tree Example

The ‘dining philosophers’ problem is a classic example of how actors in a software solution can politely contend for resources. Let’s pretend, for simplicity, that we have only two philosophers sitting at a table. There are two forks and a bowl of pasta in the middle of the table. Each philosopher needs to hold both forks to be able to eat. If each philosopher picks up one fork and simply waits for the other one to become available we will have a deadlock. The philosophers will

wait an infinite amount of time for the other fork. There are many different solutions to avoid this outcome. The Resource Hierarchy solution [22] states that each fork should be ranked. Therefore, we have fork 0 and fork 1 as shown in Figure 22. It states that each philosopher must pick up the lower ranked fork first and the next highest ranked fork next before eating. If the philosopher cannot eat he just leans back and thinks instead.

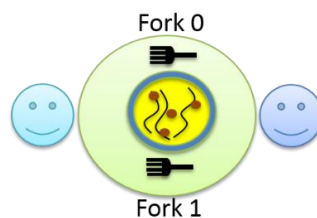


Figure 22 - The dining philosophers

The root of our example behavior tree will be a selector with two children. The first child will be a sequence to determine the availability and appropriateness of eating. The second child will be a sequence that just ‘leans back and thinks.’ We will assume for simplicity that the ‘eat’ action handles replacing the forks on the table after eating is complete.

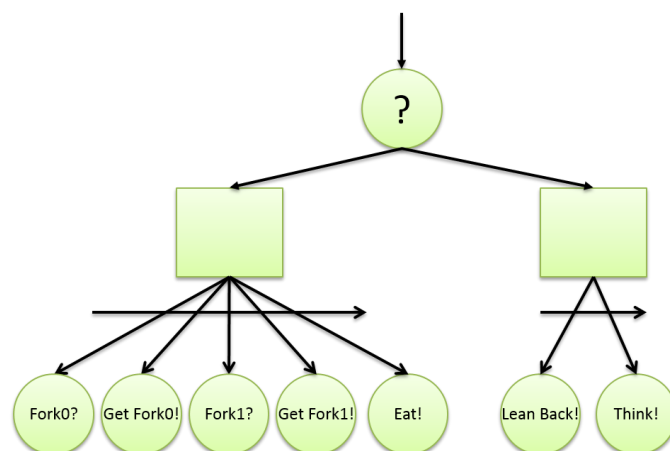


Figure 23 –A simplified behavior tree solution to the ‘dining philosophers’ problem. Statements with question marks at the end are assertions and those with exclamation points are actions.

If we assume that philosopher 0 has just a slight head start over philosopher 1, the behavior trees after one complete traversal will look like Figure 24.

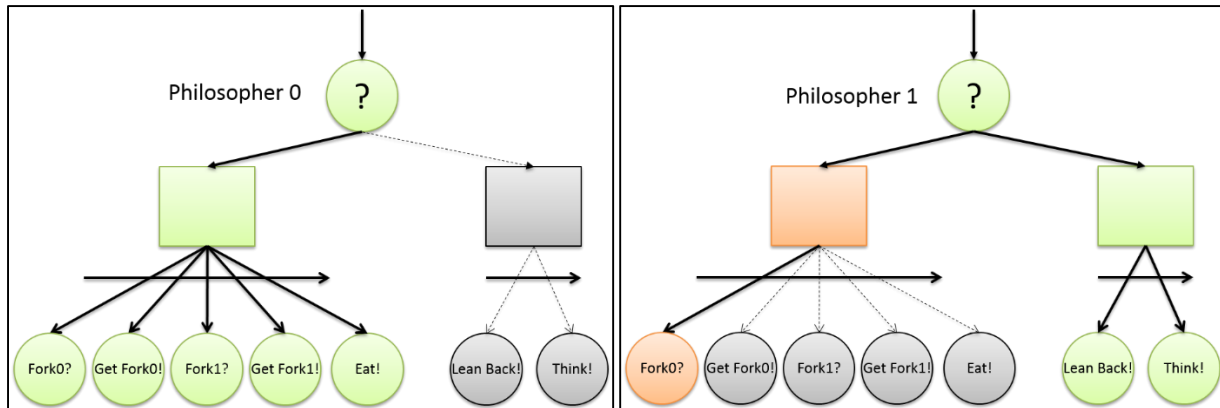


Figure 24 – The dining philosopher's behavior trees after one complete traversal

Without any length of study, obvious areas of improvement in the behavior trees present themselves. As the following optimizations will show, this is where the utility of behavior trees lies. We will completely refactor them with very little effort and yet obtain much more functionality.

Current behavior tree observations and shortcomings:

- When philosopher 0 puts his forks down they will immediately be available and, unless philosopher 1 is quick enough to pick them up, he will pick them up and begin eating again.
- The philosophers will eventually become full and tired of thinking. There is no fallback action to accommodate this.

The necessary improvements to the behavior trees are these: (1) Add a 'sleep' behavior, (2) add a negative timer decorator to the 'eat' and 'sleep' behaviors to prevent them from activating too often, and (3) add a positive timer decorator to the 'eat' and 'sleep' behaviors to ensure they take

an appropriate amount of time to run. The difference between a positive and a negative timer is this: A positive timer causes an action to continue for a set period of time by reporting `NodeStatus.Running` while receiving a `NodeStatus.Success` from its child. A negative timer prevents an action from occurring for a set period of time after it receives its first `NodeStatus.Success`. A sequence diagram of this effect is shown in Figure 25. Our new behavior tree is shown in Figure 26.

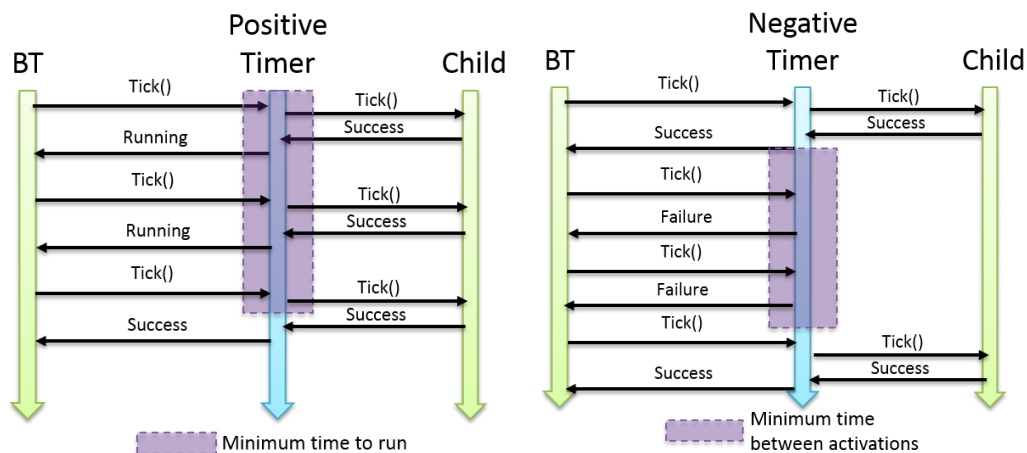


Figure 25 - Sequence diagram for a positive and negative timer decorator

You'll notice in Figure 26 that our 'think' action has been replaced by a 'think' triangle. This is to imply that each abstract action like eat, sleep, or think can be replaced by a fully functional behavior sub-tree of its own. For example, the 'think' action might contain a random selector that decides what to think about along with a positive timer to ensure it thinks about it for a certain period of time. Also, since every component in a behavior tree is equally considered a node, they can be shuffled in any fashion to achieve different behaviors. In Figure 26, the eat and sleep behaviors could easily be swapped to give more priority to the sleepy philosopher.

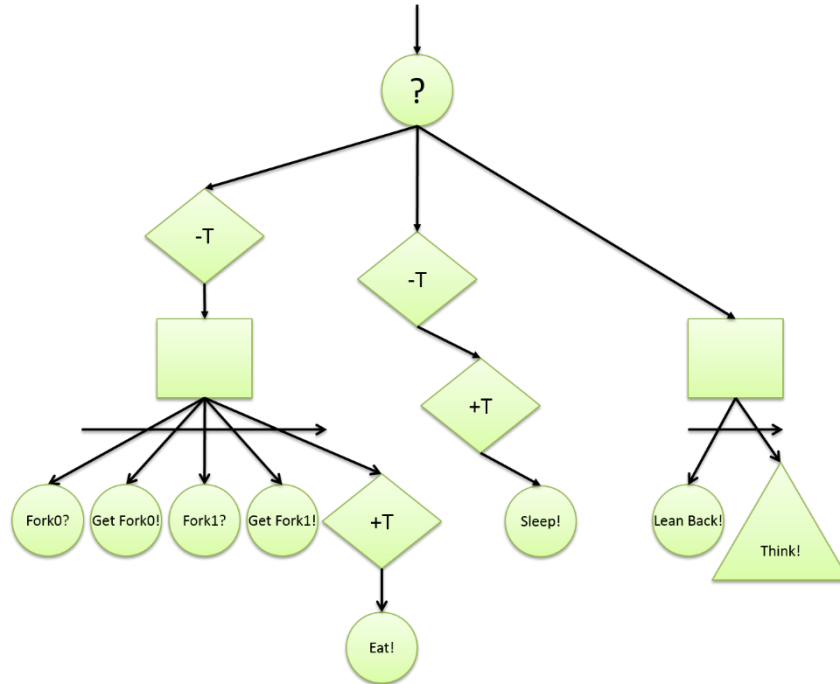


Figure 26 - Complete 'dining philosophers' behavior tree representation

1.4.8 Sample Software Implementation

```

public interface INode
{
    NodeStatus Tick();
}
  
```

Figure 27 - The INode implementation for the examples in this discussion

```

public enum NodeStatus
{
    Failure,
    Invalid,
    Ready,
    Running,
    Success,
}
  
```

Figure 28 - The possible values of a NodeStatus enumeration

The diagrams and abstract functionality of behavior trees should now be clear. Here I will discuss a very simple implementation of the basic building blocks. In this sample implementation every node must implement the INode interface as shown in Figure 27, above. This implementation is not an accepted standard. Each individual or company is at liberty to devise their own

implementation. The only requirement is that each node contain a Tick method that returns a NodeStatus enumeration. The NodeStatus possible values are shown in Figure 28.

1.4.8.1 Sequences & Selectors

```
public class ExampleSequence : INode
{
    protected ExampleSequence()
    {
        Status = NodeStatus.Invalid;
    }
    public NodeStatus Status;
    private List<INode> _children = new List<INode>();
    public List<INode> Children
    {
        get { return _children; }
    }
    public void AddChild(params INode[] nodes)
    {
        foreach (INode node in nodes)
        {
            _children.Add(node);
        }
    }
    public void Initialize()
    {
        _status = NodeStatus.Ready;
    }
    protected void Update()
    {
        foreach (INode n in Children)
        {
            Status = n.Tick();
            if (Status != NodeStatus.Success)
            {
                break;
            }
        }
    }
    public NodeStatus Tick()
    {
        if (Status == NodeStatus.Invalid)
        {
            Initialize();
        }
        Update();
        return Status;
    }
}
```

Figure 29 - An example of a Behavior Tree Sequence

Sequences and selectors are composites. They must contain an enumerable list of children and some method of populating that list. In this implementation an AddChild(params INode[] nodes) method is available. This method will accept an array of any size as long as it contains a type

derivable from INode. A potential implementation of a sequence is shown in Figure 29, above. Notice the simplicity in the Update method. It simply iterates through its children and calls their Tick method, halting if it reaches one that returns other than success.

The only difference in the coding of a selector is in the Update method (Figure 30). Now the Update method iterates through its children and halts if it reaches one that does report success.

```
protected void Update()
{
    foreach (INode n in Children)
    {
        Status = n.Tick();

        if (Status == NodeStatus.Success)
        {
            break;
        }
    }
}
```

Figure 30 - An example of a Behavior Tree Selector Update method

1.4.8.2 Decorators

Once again, the only difference between decorators and other INodes is in the Update method. One implementation is shown in Figure 31. This implementation simply wraps the _child.Tick call inside of a try/catch block and handles appropriate exceptions.

```
protected void Update()
{
    lock (this) // lock shared resources
    {
        try
        {
            Status = _child.Tick();
        }
        catch (Exception /* unique exceptions */)
        {
            //Handle the exception
        }
    }
}
```

Figure 31 - An example of a Behavior Tree Decorator Update method

1.4.8.3 Actions & Assertions

Actions and assertions will typically be implemented as needed for a specific project. For example, if the behavior of an underwater vehicle is being developed, an assertion may call `MyVehicle.Depth` to determine the depth of the vehicle. Based on the success / failure of this assertion, a subsequent action in the tree may call `MyVehicle.GetGPS()` or `MyVehicle.GoToSurface()` respectively.

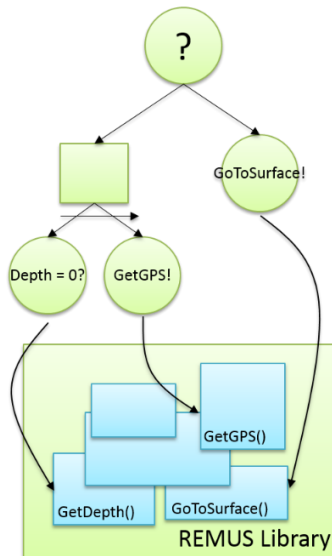


Figure 32 - A behavior sub-tree representing the actions to get a GPS fix

```
public class MyCheckDepthAssertion : INode
{
    private REMUS _remus;
    public MyCheckDepthAssertion()
    {
        _remus = REMUS.GetInstance();
    }
    public NodeStatus Tick()
    {
        if (_remus.Depth <= 0)
        {
            return NodeStatus.Success;
        }
        else
        {
            return NodeStatus.Failure;
        }
    }
}
```

Figure 33 - Sample code that checks for a vehicle's depth

```

public class MyGetGPSAction : INode
{
    private REMUS _remus;
    public MyGetGPSAction()
    {
        _remus = REMUS.GetInstance();
    }
    public NodeStatus Tick()
    {
        GPSFix fix = _remus.GetGPS();
        return NodeStatus.Success;
    }
}

```

Figure 34 - Sample code that commands a vehicle to get a GPS fix

```

public class MyGoToSurfaceAction : INode
{
    private REMUS _remus;
    public MyGoToSurfaceAction()
    {
        _remus = REMUS.GetInstance();
    }
    public NodeStatus Tick()
    {
        if (_remus.Depth > 0)
        {
            _remus.GoToSurface();
            return NodeStatus.Running;
        }
        else
        {
            return NodeStatus.Success;
        }
    }
}

```

Figure 35 - Sample code that commands a vehicle to go to the surface

1.5 ECMA-335 and ISO/IEC 23271 - Common Language Infrastructure

The Common Language Infrastructure (CLI) was designed to allow programmers the ability to write software in different languages and on different platforms while still targeting the same underlying infrastructure. It is an open specification that anyone may adhere to. Two notable examples of this are Microsoft's Common Language Runtime and Xamarin's MONO implementation. The standard describes all of the necessary functionality a language must provide

to be considered CLI compliant, namely the Common Type System, Metadata, the Common Language Specification, and the Virtual Execution System (VES) [23] [24].

The Common Type System (CTS) specifies each of the possible data types and constructs supported by the standard and how they interact [25]. Adherence to these rules is what allows software written in different languages to access the same data. For example, a dynamically linked library may be written in C# may be linked at runtime to a program executing C#, Visual Basic, or C++ without any modifications to the code. The two categories of specified types are value (int, float, etc...) and reference (class1, class2, etc...) types. While the objective of the CLI is a high level of interoperability and high performance execution, the CTS is what makes it possible.

The metadata used by the CLI describes the types defined in the CTS. It is a higher level definition of the type and the CLI uses it to locate and load classes, lay out their instances in memory, resolve method invocations, translate common intermediate language (CIL) to native code, enforce security, and setup runtime context boundaries [23].

The Common Language Specification is a set of rules that are a subset of the Common Type System. It further enhances the interoperability of languages by precisely specifying the set of features that must exist for a type.

The Virtual Execution System is responsible for loading and running programs that have been programmed against the CLI specification. Its input is a CLI assembly. The CLI assembly, comprising CIL instructions, is passed through a just-in-time (JIT) compiler that translates them into native code. This common intermediate language is what enables code written on one machine to be transported to another machine and execute the same behavior.

1.6 ECMA-334 and ISO/IEC 23270:2006 – C#

The ECMA-334 and ISO/IEC 23270 standards both describe the programming language known as C#. Its main proponent was Microsoft but standardization was proposed along with Hewlett-Packard and Intel [26]. C# includes a garbage collector and implicit memory management. Therefore it is considered a managed language. The syntax is very similar to C/C++ and experienced developers should be able to begin coding in only a short time. The ECMA standard contains a Getting Started section that introduces the reader to the ubiquitous hello world notation in C#:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello world");
    }
}
```

Figure 36 - Hello World in C#

1.6.1 Mono

The Mono Project, sponsored by Xamarin, is an open source implementation of Microsoft's .NET Framework [27]. While used mainly for C# development on Linux computers, it allows Windows, MAC, and Linux users to develop software in a familiar environment and deploy their software to any of the other operating systems.

1.6.1.1 MonoGame

By leveraging the efforts of the Mono team, a new 2 and 3-dimensional cross platform programming framework was developed. This framework allows C# developers to harness both

OpenGL and DirectX backend graphic, input, and audio support. In 2013, even Microsoft Studios published titles for Windows 8 and Windows Phone 8 using this framework [28].

1.7 Modular Programming

Following the concepts of modular programming means isolating functionality into small, reusable blocks of code that can be duplicated where needed or shuffled to achieve desired functionality. Several programming constructs lend themselves to this end.

1.7.1 Interface Based Programming

The concept of programming to an interface promotes loosely coupled interacting modules as well as ease of extension. The main benefit of interface based programming is that interaction with an object doesn't imply knowledge of the underlying code, only the method of using it. For example, sports car engines are fabricated differently than luxury sedans but any experienced driver could intuitively grab the steering wheel and press the gas pedal. The output is the same but the mechanisms to achieve it may be much different. An interface describing common functions of an underwater vehicle could be implemented using C# as shown in Figure 37.

```
public interface IVehicle
{
    void TurnLeft();
    void TurnRight();
    void Surface();
    void Dive();
    // etc...
}
```

Figure 37 - An example of a C# interface describing an underwater vehicle

If this interface were published and freely available then programmers at any underwater vehicle manufacturer could implement it and submit their designs to the framework and it would integrate

seamlessly. Another useful type of interface would be a vehicle behavior. A behavior would naturally need a begin and end method:

```
public interface IBehavior
{
    void Begin();
    void End();
}
```

Figure 38 - An example of an interface describing a vehicle behavior

A hypothetical, greatly simplified class belonging to the core framework and utilizing the vehicle and behavior interfaces, described above, might be declared like the one in Figure 39. The ease of extensibility should be apparent. Programmers wishing to extend the list of behaviors available to the system need only indicate their intent to implement the interface (similar to inheriting a base class) and then do so.

```
public class SearchBehavior : IBehavior
{
    private IVehicle _vehicle;
    // constructor
    public SearchBehavior(IVehicle vehicle)
    {
        _vehicle = vehicle;
    }

    public void Begin()
    {
        _vehicle.Dive();
        // code to evaluate for appropriate depth
        _vehicle.TurnLeft();
        // code to decide appropriate distance to travel
        _vehicle.TurnRight();
        // etc...
    }

    public void End()
    {
        _vehicle.Surface();
    }
}
```

Figure 39- A class titled ‘SearchBehavior’ that takes advantage of the previously defined IVehicle interface to achieve a goal without having a priori knowledge of the interface implementation’s underlying instructions.

1.7.2 Delegates

Delegates are objects that can be used to call functions [29]. A good C++ analog would be that of a function pointer. When defining a delegate, you must state the return type and the parameter list of the function that this delegate will be compatible with. Once this is done, the delegate type can be initialized with the function it should call in the same way that an 'int' can be initialized with an integer value. In subsequent lines of code, the delegate type can be called in exactly the same way that the original method would be called. What this allows for is a general process to be developed with the specifics left up to users of that process. In Figure 40, the example involves a string modification method. The process involves making modifications to each character in that string. In this example there are two options: MakeUppercase or MakeLowercase. The constructor of the Program demonstrates the use of each of these methods using the delegate as the method caller.

```
public delegate char CharModifier(char c);
public Program()
{
    string theString = "This Is A String";
    CharModifier mod = MakeUppercase;
    string newString = StringModifier(theString, MakeLowercase);
    Console.WriteLine(newString); // Prints: this is a string
    newString = StringModifier(theString, MakeUppercase);
    Console.WriteLine(newString); // Prints: THIS IS A STRING
    Console.ReadLine();
}
public string StringModifier(string s, CharModifier mod)
{
    // StringBuilders are arrays used to construct strings efficiently
    StringBuilder sb = new StringBuilder();
    foreach (char c in s)
    {
        sb.Append(mod(c));
    }
    return sb.ToString();
}
public char MakeUppercase(char c)
{
    return char.ToUpper(c);
}

public char MakeLowercase(char c)
{
    return char.ToLower(c);
}
```

Figure 40 - Example of a delegate calling a method to modify a string

1.7.2.1 Lambda Expressions

Lambda expressions are unnamed, or anonymous, functions. In C#, they operate exactly like regular class methods when called. The only difference is in how they are declared. A normal class method comprises five parts: access modifier, return type, name, argument list, and method body. A Lambda expression comprises just two parts: argument list and method body (See Figure 41, below).

Regular class method	Lambda expression equivalent
<pre>//[access modifier] [return type] //[name][(parameter list)] private int Sum(int a, int b) { return a + b; // [method body] }</pre>	<pre>//[(parameter list)] (int a, int b) => { return a + b; // [method body] };</pre>

Figure 41 - A class method example alongside its Lambda expression equivalent

Lambda expressions, having the same nature as methods, are also compatible with delegates. In fact, the C# compiler converts lambda expressions into either (a) a delegate, or (b) an expression tree that can be examined and executed at runtime. Thus, when declaring an anonymous function it makes sense to assign it to a delegate instance that can be referenced and called later (the Func delegate is discussed in Section 1.7.2.2 below).

```
int b = 5;
// Func delegate
// parameters <input type, output type>
Func<int, int> lambda =
(int a) =>
{
    return a + b; // b is captured
};
int result = lambda(1); // result is 6
```

Figure 42 - A Lambda function capturing an outer variable

As you can see in Figure 42, Lambdas also have the ability to capture outside variables (variables not defined within their scope). Captured variables do not need to be explicitly declared in C# as in some programming languages, but rather they are inferred from the logic itself. The power of

Lambda expressions is in their ability to be passed around like data as if they were variables. For instance, we could modify our previous example of string modifications to use Lambda expressions instead. See Figure 43.

```
public delegate char CharModifier(char c);

public Program()
{
    string theString = "This Is A String";
    // c => char.ToLower() is our Lambda expression
    string newString = StringModifier(theString, c => char.ToLower(c));
    Console.WriteLine(newString); // Prints: this is a string
    // c => char.ToUpper() is our Lambda expression
    newString = StringModifier(theString, c => char.ToUpper(c));
    Console.WriteLine(newString); // Prints: THIS IS A STRING
}

public string StringModifier(string s, CharModifier mod)
{
    // StringBuilders are arrays used to construct strings efficiently
    StringBuilder sb = new StringBuilder();
    foreach (char c in s)
    {
        sb.Append(mod(c));
    }
    return sb.ToString();
}
```

Figure 43 - Using Lambda expressions instead of methods to modify a string

1.7.2.2 Actions and Funcs

Actions and Funcs are predefined, generic, general use delegates. Since they are built in types they allow developers to more easily build general purpose code able to cross application domains. The main difference between the two is just that Actions return no value. They both accept an arbitrary number of arguments defined at compile time. For example, our previous string modification program could be changed to use Action and Func delegates as shown in Figure 44. This example does not adequately make the case for using Actions and Funcs. That purpose requires more sophisticated scenarios which will be explored in my Behavior Tree section. It does, however, demonstrate the most valuable ability to pass methods as data. This concept will become a vital part of the modular behavior tree structure described later.

```

public Program()
{
    Func<string, Func<char, char>, string> stringMod =
        new Func<string, Func<char, char>, string>((theString, mod) =>
        {
            // StringBuilders are arrays used to construct strings efficiently
            StringBuilder sb = new StringBuilder();
            foreach (char c in theString)
            {
                sb.Append(mod(c));
            }
            return sb.ToString();
        });

    Action<string> printAction =
        new Action<string>(stringToPrint =>
        {
            Console.WriteLine(stringToPrint);
        });

    string s = "This Is A String";
    // c => char.ToLower() is our func
    string newString = stringMod(s, c => char.ToLower(c));
    printAction(newString);
    // c => char.ToUpper() is our func
    newString = stringMod(s, c => char.ToUpper(c));
    printAction(newString); // prints: this is a string
}

```

Figure 44 - Taking advantage of the Action and Func delegates

1.8 Case Based Reasoning

Case-Based Reasoning (CBR) is a system based on using results of previous experiences to decide on approaches to new problems. It could be said that this approach mimics the way humans decide how to overcome unfamiliar tasks, by using memories of similar tasks undertaken and the methodology that eventually lead to success. There is not a single definition of the process to achieve CBR but a general, popularly accepted design was proposed by Aamodt and Plaza in [30]. Their design comprises four stages, known as the “four REs.”

1. Retrieve the most similar case or cases
2. Reuse the information and knowledge in that case
3. Revise the proposed solution
4. Retain the parts of this experience likely to be useful in the future

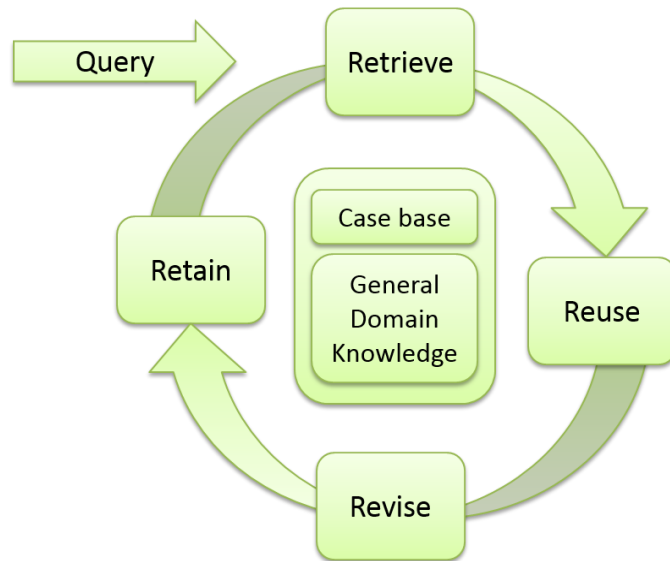


Figure 45 - The four fundamental stages of a case-based reasoning exercise (The four REs)

A case-based reasoning exercise begins with a query, proceeds to an action, and ends with newfound knowledge of the domain (see Figure 45). Queries are not prescribed but tend to take the form of mathematical formulas modeling the similarity of the cases in the base to the current situation.

The following sections describe the four stages of a case-based reasoning cycle using a simple ‘Game of Life’ example. The example will include an actor and a world model.

Table 1 - A simple model of an actor in a game of life example

Actor	Description
Health	The general well-being of the actor. A high value indicates good health. A low value would indicate some sort of sickness. A 0 would indicate death.
Hunger	An indicator of how hungry the actor is. A 0 would mean not hungry at all and a 10 would mean extremely hungry.
Energy	The level of stored energy the actor contains. A 0 would mean the actor needed rest badly. A 10 would mean that the actor was ready to participate in some kind of activity.
Mood	The level of contentment that the actor feels with its current situation. A 0 would indicate very sad and a 10 would indicate very happy. A sad actor is more likely to change the current activity it is participating in.

Table 2 - A simple model of the world in a game of life example

World	Description
Time	The current 24-hour time of day. 0 would mean midnight (am) and 2359 would mean midnight (pm).
Weather	An enumerated value of the current weather condition. {raining, sunny, snowing, sleeting, sprinkling, windy, ... }
Season	{spring, summer, fall, winter}

Table 3 - A collection of cases in the case-base

Eat		Sleep		Exercise		Heal		Relax	
Health	5	Health	5	Health	6	Health	2	Health	5
Hunger	7	Hunger	3	Hunger	4	Hunger		Hunger	3
Energy	5	Energy	3	Energy	8	Energy		Energy	4
Mood	5	Mood	5	Mood	5	Mood		Mood	5
Time	5<t<22	Time	22<t<5	Time	6<t<18	Time		Time	5<t<22
Weather		Weather		Weather		Weather		Weather	
Season		Season		Season		Season		Season	

1.8.1 Retrieval

In the retrieval stage, mathematical similarity functions are typically used to compare cases in the base to the current situation. Any number of features may be analyzed. In a game, health and difficulty may play a role. In navigation, traffic and orientation may be accounted for. Once analysis is complete, the top-k results are chosen and passed to the next phase for further evaluation or possible refinement. For our example, we start out with a query, Q, based on current world and actor variables, and a set of cases, C_k , from the case base.

Table 4 - A CBR query based on current world variables

Query (Q)	
Health	7
Hunger	5
Energy	5
Mood	5
Time	12
Weather	Sunny
Season	Summer

Then, a mathematical formula is applied to determine the most appropriate action to take. Each case in the case base is evaluated.

$$sim(Q, C) = \sum_{d \in D(Q, C)} w_d * sim_{loc}(Q_d, C_d)$$

$$D(Q, C) = Q.descriptors \cap C.descriptors$$

$$sim_{loc}(Q_d, C_d) = 1 - \frac{|Q_d.value - C_d.value|}{size_d}$$

Where w_d is a weight factor allowing priority of certain attributes under certain circumstances. In our example, it is 1 in all cases. Also, to clarify, the time attribute becomes either a 1 or a 0 for ‘true’ and ‘false’ respectively. Application of the formula yields:

Table 5 – A world actor’s rationale for choosing a case from the case base

	Eat	Sleep	Exercise	Heal	Relax
Health	0.8	0.8	0.9	0.5	0.8
Hunger	0.8	0.8	0.9		0.8
Energy	1	0.8	0.7		0.9
Mood	1	1	1		1
Time	1	0	1		1
sim(Q,C)	0.92	0.68	0.9	0.5	0.9

In this instance, the actor would choose the ‘Eat’ activity. If we presume the actor eats a healthy meal that energizes him, the next instance of the world query might yield a lower hunger attribute and a higher energy attribute:

Table 6 - A modified CBR query taking into account a recent hardy meal

Query	
Health	7
Hunger	3
Energy	8
Mood	5
Time	13
Weather	Sunny
Season	Summer

This, in turn, would result in a case-base result of:

Table 7 - The resulting case base coefficients after taking the previous query into account

	Eat	Sleep	Exercise	Heal	Relax
Health	0.8	0.8	0.9	0.4	0.8
Hunger	0.6	1	0.9		1
Energy	0.7	0.5	1		0.6
Mood	1	1	1		1
Time	1	0	1		1
sim(Q,C)	0.82	0.66	0.96	0.4	0.88

Now our actor would choose to exercise. A further refinement of the result might be to account for season and weather conditions. Since it is summertime and sunny, our actor might choose to go for a run. If it were winter and rainy, our actor might head to the gym. Following this activity it is easy to foresee the actor with a very low energy rating and pursuing the relax activity.

1.8.2 Reuse

During reuse, the selected cases are either applied directly (transformational reuse) or adapted (derivational reuse) to conform to user supplied guidelines. The transformational approach was already demonstrated in our example of a retrieval as the actor chose to execute the ‘eat’ and

‘exercise’ activities. Opportunities for the derivational approach would have emerged in a more sophisticated approach to determining which food to eat or which exercise to undertake.

1.8.3 Revision

During the revision phase, the result of applying the case is evaluated as to whether or not success was achieved. If the case was successful it may be retained (section 1.8.4 below). If the case was not successful, it may be discarded or ‘repaired’ through a fault identification and remediation process determined by the developer.

1.8.4 Retention

If a particular case successfully delivers the desired result, the underlying knowledge base (case base) may be updated with a new case for future retrieval. On the other hand, if a chosen case was unsuccessful in supplying the desired result, it may be modified to portray a less desirable solution to the query during the next retrieval process.

1.9 XML

The eXtensible Markup Language was developed by the World Wide Web Consortium (W3C). It was derived from an older standard format called SGML (ISO 8879) to be more web friendly [31]. Since its inception, it has emerged as the de facto standard for information exchange across working boundaries. It relates data in a self-describing manner so that users and machines can interpret the data [32]. Numerous tools have been developed for almost any language to parse XML documents. Unlike HTML, XML follows a strict schema. That is, the syntax does not grant conformity to files that contain errors. It is very verbose, carrying a lot of metadata about the

information which is represented. More efficient data transfer formats are easily discovered but the bridges of interoperability begin to fall down.

An XML document usually begins with a header declaring its version and encoding. It is followed by any number of elements, attributes, and comments.

```
<?xml version="1.0" encoding="utf-8" ?>
<University>
  <!--This is a comment-->
  <!--Below is an element called 'name' that has an attribute called 'type'-->
  <Name type="string">Florida State University</Name>
  <City type="string">Tallahassee</City>
  <State type="string">Florida</State>
  <ZipCode type="uint" maxLength="5">32306</ZipCode>
  <Mascot type="string">Seminole</Mascot>
</University>
```

Figure 46 - An example XML file showing elements, attributes, and comments

1.9.1 Tags & Elements

<Tag></Tag>

An XML tag is an identifier surrounded by an opening ‘<’ and a closing ‘>’ (e.g. <University>). Tags may define the opening of an element, the closing of an element, or they may be empty element tags (e.g. <EmptyElementTag/>). Closing tags must match the opening tag except for a ‘/’ just prior to the tag name (e.g. </ClosingTag>). Elements are composed of an opening tag, the element content, and a closing tag. The content of an XML element may contain any combination of text and whitespace. Certain characters, however, are used by the XML parsers to delineate tags and are thus unavailable for use in standard text. These elements must be ‘escaped’ by the entities shown in Table 8. An element may also contain attributes further describing the markup.

Table 8 –XML escape characters

Character	Entity
<	<
>	>
“	"e
‘	'
&	&

1.9.2 Attributes

```
<ZipCode type="uint" maxLength="5">32306</ZipCode>
```

XML attributes are key/value pairs contained with the opening tag of an element. They can be used to add additional information about the element they are contained in. For example, in Figure 46, the ZipCode element has attributes type and maxLength to give the XML consuming code an ability to recognize improper values.

1.9.3 Comments

```
<!--This is a comment-->
```

XML comments are placed in a document by denoting them with an opening <!-- and closing them with a -->. These characters, like in many software coding languages, are ignored by the parser and are used by developers to give insight to the meaning of the markup.

1.9.4 Schema

XML schemas define valid XML documents using the same rules that govern those XML documents. That means that all of the usual XML parsers can also parse an XML schema as a regular XML document. Suppose we wanted an intuitive way to store information about a particular underwater mission. A database representation would certainly work. XML files have

the capability to describe tables, primary keys, foreign keys, data types, and other relationships inherent to a database structure. In this example we have 5 tables.

Mission Types (MissionTypes): This table has one column (Name) that describes what type of missions we typically execute. For example, economic search, communication characterization, etc...

Missions: This table assigns a unique identifier to each mission that we send our vehicle on. It also records the start time, end time, and mission type. Mission type has a foreign key relationship to the MissionTypes.Name column.

Target Queries (TargetQueries): This table maintains a record of each range measurement we take with an underwater node. It records the current time, the vehicle's location (lat, long), the node's id, the node distance, and the mission number we are currently executing. This table has a foreign key relationship to the Mission.MissionNumber column.

Probable Locations (ProbableLocations): At the end of each mission, we use multi-lateration techniques to determine each node's most likely location. The table records the mission number, target number, and target location (lat, long). This table has a foreign key relationship to the Mission.MissionNumber column.

Communication Checks (CommunicationChecks): When characterizing a node's communication capabilities, we need to determine the bit errors that occur during each transmission. This will help optimize future communications with the node since we'll have a record of its highest success path. This table has a foreign key relationship to the Mission.MissionNumber column.

The table representation of our simple database is shown in Figure 47, below. The text based XML schema is shown in Figure 48. All of the necessary constructs to describe primary keys, foreign keys, and the other constraints are valid XML elements and attributes.

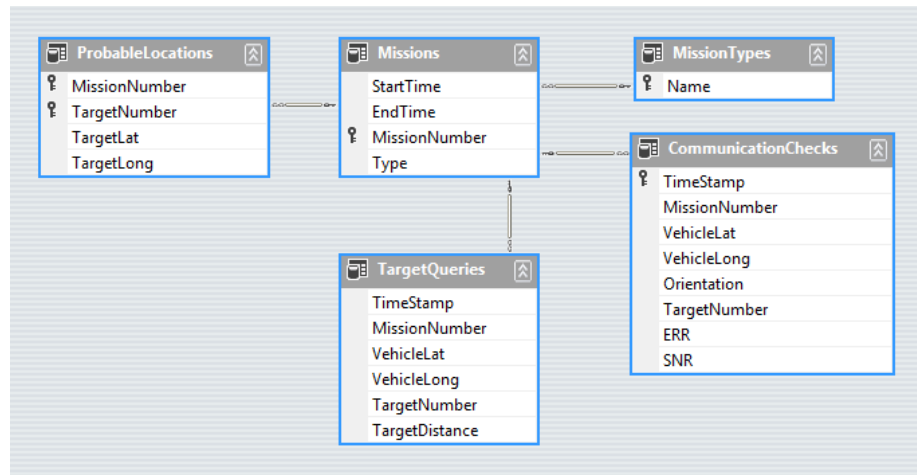


Figure 47 –A sample XML schema as displayed by Microsoft Visual Studio 11

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="Sample_x0020_Data_x0020_Set" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-
msdata">
  <xs:element name="Sample_x0020_Data_x0020_Set" msdata:IsDataSet="true"
msdata:UseCurrentLocale="true">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="MissionTypes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Name" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Missions">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="StartTime" type="xs:string" minOccurs="0" />
              <xs:element name="EndTime" type="xs:string" minOccurs="0" />
              <xs:element name="MissionNumber" msdata:AutoIncrement="true" type="xs:int" />
              <xs:element name="Type" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="CommunicationChecks">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="TimeStamp" type="xs:string" />
              <xs:element name="MissionNumber" type="xs:int" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 48 - The XML schema describing the relationship of the tables in Figure 26


```

        <xs:element name="VehicleLat" type="xs:double" />
        <xs:element name="VehicleLong" type="xs:double" />
        <xs:element name="Orientation" type="xs:double" />
        <xs:element name="TargetNumber" type="xs:int" />
        <xs:element name="ERR" type="xs:int" minOccurs="0" />
        <xs:element name="SNR" type="xs:double" minOccurs="0" />
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="TargetQueries">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="TimeStamp" type="xs:string" />
            <xs:element name="MissionNumber" type="xs:int" />
            <xs:element name="VehicleLat" type="xs:double" />
            <xs:element name="VehicleLong" type="xs:double" />
            <xs:element name="TargetNumber" type="xs:int" />
            <xs:element name="TargetDistance" type="xs:double" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProbableLocations">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="MissionNumber" type="xs:int" />
            <xs:element name="TargetNumber" type="xs:int" />
            <xs:element name="TargetLat" type="xs:double" />
            <xs:element name="TargetLong" type="xs:double" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
<xs:unique name="Constraint1" msdata:PrimaryKey="true">
    <xs:selector xpath="."//MissionTypes" />
    <xs:field xpath="Name" />
</xs:unique>
<xs:unique name="Missions_Constraint1" msdata:ConstraintName="Constraint1"
msdata:PrimaryKey="true">
    <xs:selector xpath="."//Missions" />
    <xs:field xpath="MissionNumber" />
</xs:unique>
<xs:unique name="CommunicationChecks_Constraint1" msdata:ConstraintName="Constraint1"
msdata:PrimaryKey="true">
    <xs:selector xpath="."//CommunicationChecks" />
    <xs:field xpath="TimeStamp" />
</xs:unique>
<xs:unique name="ProbableLocations_Constraint1" msdata:ConstraintName="Constraint1"
msdata:PrimaryKey="true">
    <xs:selector xpath="."//ProbableLocations" />
    <xs:field xpath="MissionNumber" />
    <xs:field xpath="TargetNumber" />
</xs:unique>
<xs:keyref name="FKProbableLocationMissionNumberMatch" refer="Missions_Constraint1"
msdata:ConstraintOnly="true">
    <xs:selector xpath="."//ProbableLocations" />
    <xs:field xpath="MissionNumber" />
</xs:keyref>
<xs:keyref name="FKTargetQueriesMissionNumberMatch" refer="Missions_Constraint1"
msdata:ConstraintOnly="true">
    <xs:selector xpath="."//TargetQueries" />
    <xs:field xpath="MissionNumber" />
</xs:keyref>
<xs:keyref name="FKMissionNumberMatch" refer="Missions_Constraint1"
msdata:ConstraintOnly="true">

```

Figure 48 – continued

```

<xs:selector xpath="./CommunicationChecks" />
<xs:field xpath="MissionNumber" />
</xs:keyref>
<xs:keyref name="FKMissionTypeMatch" refer="Constraint1" msdata:ConstraintOnly="true">
  <xs:selector xpath="./Missions" />
  <xs:field xpath="Type" />
</xs:keyref>
</xs:element>
</xs:schema>

```

Figure 48 - continued

1.9.5 XPath

```

<?xml version="1.0" encoding="utf-8" ?>
<documentElement name="docElement">
  <child0 name="child0">
    <child0-1 name="child0-1"/>
    <child0-2>
      <name>child0-2</name>
    </child0-2>
  </child0>
</documentElement>

```

Figure 49 - XML snippet for XPath example (filename xml.xml)

XPath is the standard defined by W3C that provides a common syntax for querying XML documents [29]. The specification itself can be found at <http://www.w3.org/TR/xpath/>. C# fully supports XPath queries through its System.XML namespace classes. Given the XML snippet in Figure 49, we can query its elements and attributes like so:

```

using System.Xml;
// etc
private static void XPathExample()
{
  XmlDocument xdoc = new XmlDocument();
  xdoc.Load("xml.xml");
  XmlNode docElm = xdoc.DocumentElement;
  XmlNode docElmNameAttribute = docElm.SelectSingleNode("@name");
  Console.WriteLine("docElm name=" + docElmNameAttribute.Value); // docElement
  XmlNode child0 = docElm.SelectSingleNode("child0");
  Console.WriteLine("child0 name=" + child0.SelectSingleNode("@name").Value); // child0
  XmlNode child0_1 = child0.SelectSingleNode("child0-1");
  Console.WriteLine("child0-1 name=" + child0_1.SelectSingleNode("@name").Value); // child0-1
  XmlNode child0_2 = child0.SelectSingleNode("child0-2");
  Console.WriteLine("child0-2 name=" + child0_2.SelectSingleNode("name").InnerText); // child0-2
}

```

Figure 50 - Utilizing XPath with C# XMLNodes

1.10 Reflection

VehicleInterface.cs	SampleVehicle.cs
<pre>namespace LibraryExamples { public interface IVehicle { Distance GetDepth(); void SetDepth(Distance dist); Speed GetSpeed(); void SetSpeed(Speed speed); } }</pre>	<pre>namespace SampleVehicle { public class UserVehicle : IVehicle { private Distance _depth; private Speed _speed; public Distance GetDepth() { return _depth; } public void SetDepth(Distance depth) { _depth = depth; } public Speed GetSpeed() { return _speed; } public void SetSpeed(Speed speed) { _speed = speed; } } }</pre>

Figure 51 - An IVehicle interface implemented by a SampleVehicle class.

Reflection, simply put, is the capability of software to inspect code metadata and operate on it. For example, reflection can be used to open a dynamically linked library at runtime, inspect the classes therein, and instantiate them. The experimental behavior tree framework used in this research will make use of XML files and reflection to allow users the ability to define which files contain the necessary components to make the framework activate properly. As an example, Figure 51 shows the code contained in two files: VehicleInterface.cs defines the necessary methods of a vehicle and SampleVehicle.cs defines a class that implements them. The usefulness of interfaces and their implementations was described in section 1.7.1 above.

Each of the files in Figure 51 will eventually be compiled into their own DLL (VehicleInterface.dll and SampleVehicle.dll, respectively). Now at runtime, if a component of the behavior tree

framework has need of an IVehicle interface and the XML configuration file that was setup by the user specifies that SampleVehicle.dll is the proper file to retrieve it from, it can do so as shown in Figure 52.

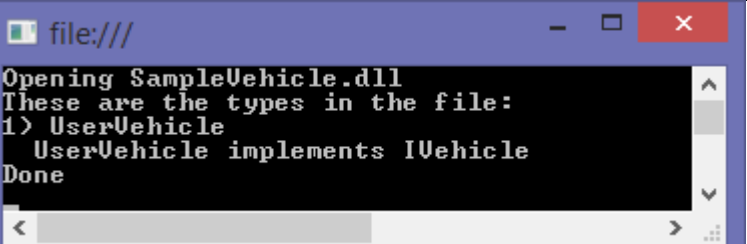
ConsoleProgram.cs <pre>private Program() { Console.WriteLine("Opening SampleVehicle.dll"); FileInfo fi = new FileInfo("SampleVehicle.dll"); Assembly a = Assembly.LoadFile(fi.FullName); Type[] types = a.GetTypes(); Console.WriteLine("These are the types in the file:"); for (int i = 0; i < types.Length; i++) { Console.WriteLine((i + 1) + ") " + types[i].Name); Type[] interfaces = types[i].GetInterfaces(); for (int j = 0; j < interfaces.Length; j++) { Console.WriteLine(" " + types[i].Name + " implements " + interfaces[j].Name); } } Console.WriteLine("Done"); }</pre>
Output of ConsoleProgram when executed 

Figure 52 - An example of a program opening a DLL and inspecting its types and interfaces

Furthermore, if the examination of the file sufficiently establishes its usefulness as an IVehicle, an instance of an IVehicle can be created and utilized as seen fit by the programmer. In Figure 53, an IVehicle is instantiated, the depth and speed are set, and the values are verified by writing them to the console window.

ConsoleProgram.cs <pre>private Program() { Console.WriteLine("Opening SampleVehicle.dll"); FileInfo fi = new FileInfo("SampleVehicle.dll"); Assembly a = Assembly.LoadFile(fi.FullName); Type[] types = a.GetTypes(); IVehicle vehicle = Activator.CreateInstance(types[0]) as IVehicle;</pre>
--

Figure 53 - Instantiating a type / interface from a DLL and utilizing its methods.

```
if (vehicle != null)
{
    vehicle.SetDepth(Distance.FromFeet(10));
    vehicle.SetSpeed(Speed.FromMetersPerSecond(5));
    Console.WriteLine("Depth: " + vehicle.GetDepth().Feet);
    Console.WriteLine("Speed: " + vehicle.GetSpeed().MetersPerSecond);
}
else
{
    Console.WriteLine(types[0].Name + " does not implement IVehicle");
}
}
```

Output of ConsoleProgram when executed

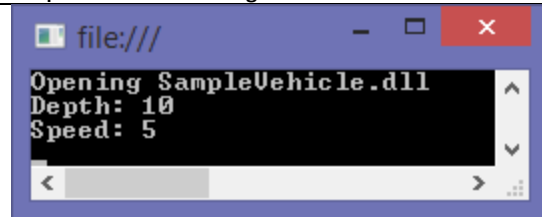


Figure 53 - continued

CHAPTER 2

RELATED WORK

2.1 Behavior Architectures

The following architectures are modern designs being used as research platforms to develop sophisticated behaviors. The EGO architecture uses behavior modules in a tree structure similar to the behavior trees proposed in this framework. Another design, Query-Enabled Behavior Trees, uses the same proposed behavior tree foundation as the basis for a case-based-reasoning (CBR) extension. The usability benchmarks paper, described below, simply makes the case for developing behavior frameworks that are targeted at the end user.

2.1.1 EGO – Emotionally GrOunded Architecture

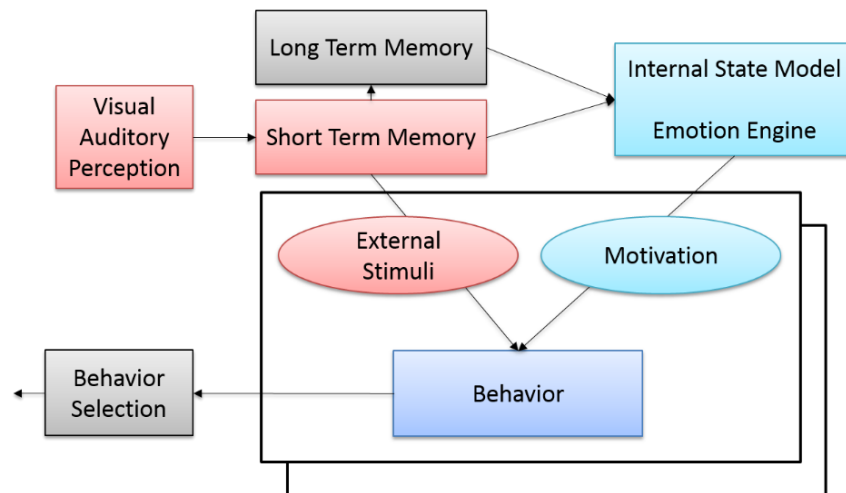


Figure 54 - The Emotionally GrOunded (EGO) Architecture

The Emotionally GrOunded module based selection architecture shown in Figure 54 was designed to be used in personal interaction robots [33]. These robots would potentially have auditory and

visual perception sensors lending external stimuli, along with an emotional engine, to the behavior selection component. This component is then responsible for choosing the appropriate behavior, protecting the integrity of hardware resources, and maintaining an acceptable state.

One of the motivations for this new approach to behaviors was “to provide a simple description, employ a simple strategy for behavior design, and make it easy to reuse behaviors.” They accomplished this by developing ‘behavior modules’ and integrating them into a tree structure. This tree structure is broken down into conceptual levels which can be independently evaluated for suitability of execution. Each of the modules on this tree is capable of being an executable behavior or the parent of other child modules which can be executed. Figure 55 shows an example of a grouping of behavior modules into these conceptual levels. The parent module of both ‘Play’ and ‘Interaction’ determines which child should be executed based on a behavior value determined by the equation

$$B_v = \beta M_v + (1 - \beta) R_v$$

Equation 1 - EGO Architecture - Behavior value

where B_v is the behavior value, M_v is the motivation value, and R_v is the releasing value, representing an expected merit value. R_v is calculated by

$$R_v = \alpha \Delta S + (1 - \alpha)(S + \Delta S)$$

Equation 2 - EGO Architecture - Releasing value

where S is the satisfaction value derived from the current internal status and ΔS is the expected change of S . ΔS is calculated by

$$\Delta S = f(\Delta I)$$

Equation 3 - EGO Architecture – Expected change of satisfaction value

where ΔI is the expected change of internal value based on external stimuli.

Then, once 'Play-Soccer' is chosen, the search, approach, and kick behaviors are evaluated for appropriate execution.

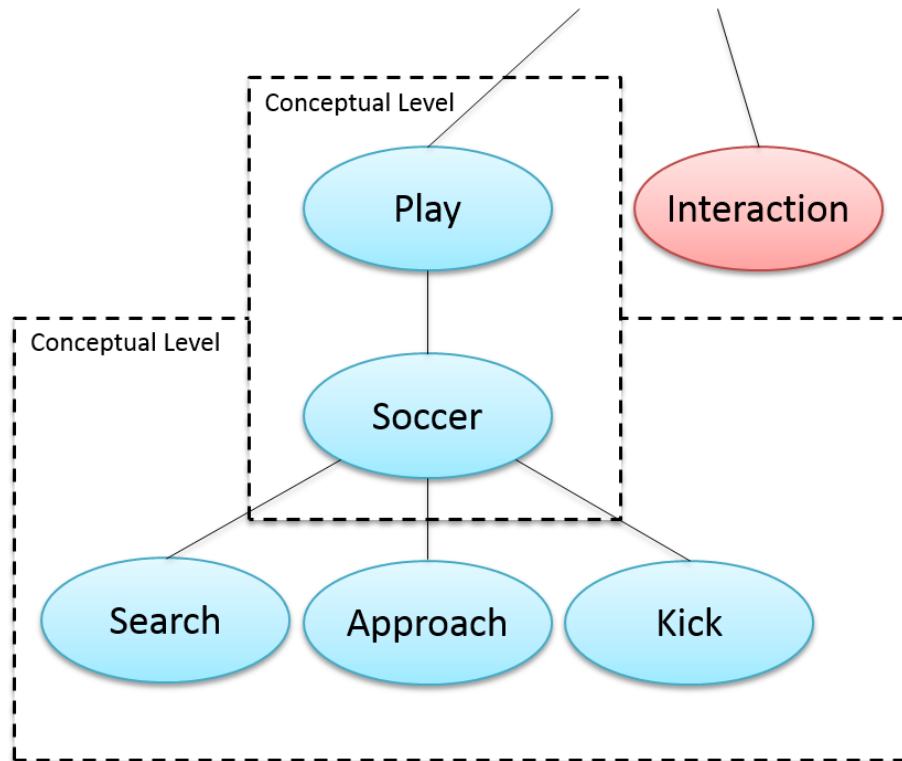


Figure 55 - Behavior modules organized into conceptual levels

Monitoring hardware resource availability during parallel execution is done with a set of keys. These keys are distributed by parent modules to their children based on a selected distribution strategy. When the execution phase begins, behavior modules are activated in parallel until a conflict of execution key is detected. When this scenario arises, lower priority modules are stopped first and then the higher priority modules are activated.

By utilizing the described architecture, researchers were able to successfully implement soccer, charging, and chatting behaviors. The suitability of the architecture was confirmed and future research plans in the area of conceptual layer decomposition were discussed.

2.1.2 Query Enabled Behavior Trees

In [34], the authors inject some planning capabilities based on case-based reasoning. Their reasoning behind the utilization of behavior trees was simple. Instead of developing a new AI framework from scratch, they would use “industry-trusted techniques” and simply extend them with academic research results. Similar justification is used for the reflective extension proposed in this paper.

Basic behavior trees were discussed in section 1.3 above. This extension proposes a new node called a query node. This new node, unlike the static sequences and selectors previously described, uses the current state of the game to match an appropriate ‘case’ from a case base. Not all cases are appropriate at each tick. Therefore cases are chosen based on a similarity function. Given a query ‘Q’ and a case ‘C’ from the case base:

$$sim(Q, C) = \begin{cases} Q.domain \not\subseteq C.class \Rightarrow 0 \\ The\ restrictions\ on\ parameters\ in\ Q\ do\ not\ hold\ in\ C \Rightarrow 0 \\ Otherwise \Rightarrow sim_{atr}(Q, C) \end{cases}$$

$$sim_{atr}(Q, C) = \sum_{d \in D(Q, C)} w_d * sim_{loc}(Q_d, C_d)$$

$$D(Q, C) = Q.descriptors \cap C.descriptors$$

$$sim_{loc}(Q_d, C_d) = 1 - \frac{|Q_d.value - C_d.value|}{size_d}$$

The weight, w_d , is an importance value for a descriptor set by the designer. The value *size_d* is the size of the interval of valid values for a descriptor.

The attraction to this method is that behaviors developed late in a development cycle have as much utility in earlier developed behavior trees as late ones. That is, as opposed to statically formed trees, where transitions to behavior nodes are explicitly typed and must be updated as new behaviors are developed, now only the case base needs to be updated. If the new case is properly added to the query domain it will be evaluated appropriately during the next run. Based on the authors' calculations, this framework would have had the effect of automatically revising 6700 nodes during the Halo 2 development cycle.

2.1.3 Usability Benchmarks of Targets-Drives-Means Robotic Architecture

In [35], the authors present an approach to behavioral programming targeted toward end developers. The architecture is called Target-Drives-Means (TDM) and is meant to promote rapid understanding and development of behaviors. The development studio for TDM enables programming of reusable components (small, discrete behavioral parts) and complete behaviors units which comprise several of the reusable components. The behavior itself is observed by activation of these components in the arrangement given by the developer using a graphical IDE. Experiments included watching a robot perform a behavior and then subsequently studying the behavior unit. Next, a behavioral unit was studied first and then observed on a mobile robot. The performance metric was how long it took for a new developer to understand the operation of the behavior. Later experiments included comparison of the time it took to develop a desired behavior using TDM and the older method of flowcharts. No individual components were developed, only complete behavioral units by means of arranging the smaller, preprogrammed components. Results

of this study show that while the initial learning curve is higher, intuitive measures can increase the behavioral programming capacity of informed developers (see Figure 56).

Table 9 – TDM Usability Questionnaire

Question Number	Question Text
1	I think this product would be easy to use
2	I think I would need the support of a technical person to be able to use this product.
3	I would need to learn a lot of things before I could get going with this product.

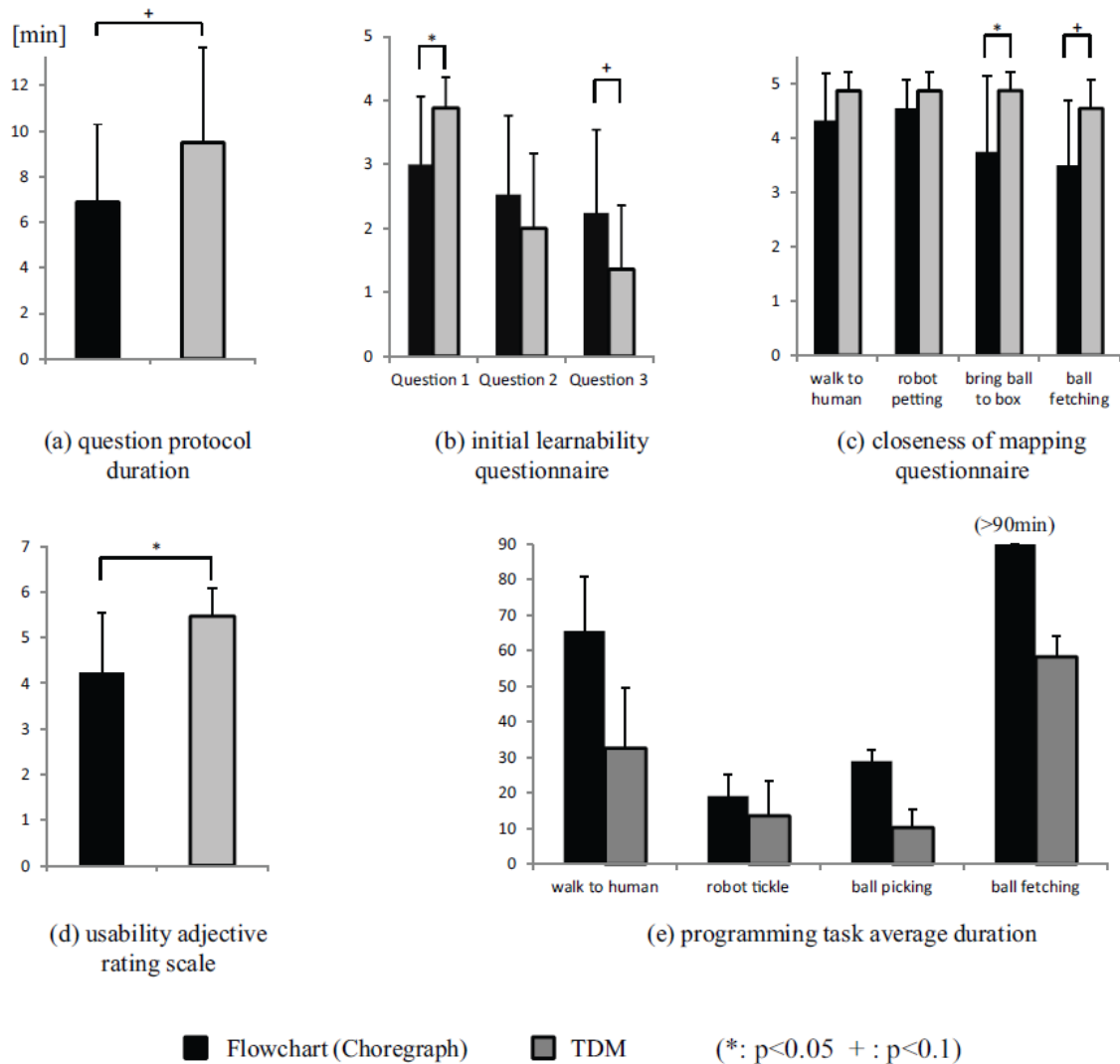


Figure 56 - Results of Target-Drives-Means Framework Usability Benchmarks (from [35])

2.2 Underwater Autonomy

2.2.1 AUV AVALON

The changing climate and the encroachment of human development into ocean environments increases the need to monitor these areas. Autonomous Underwater Vehicles are the obvious tools needed to accomplish this task. However, since underwater vehicles lack a reliable, high-speed communication channel their missions are pre-planned and very restrictive. The authors of [36] recognize this need for more reactive behaviors and propose a plan management approach that they hope will replace planning-based approaches to underwater autonomy. They document that almost all architectures are developed to solve a very specific mission. One exception was a T-REX architecture build on top of the NASA EUROPA (Extensible Universal Remote Operations Planning Architecture) planner. However, this architecture can only represent what EUROPA can represent. This means it is unable to branch or re-plan. Even these developers, though, admit that their design relies heavily on the software framework that they used to develop their architecture.

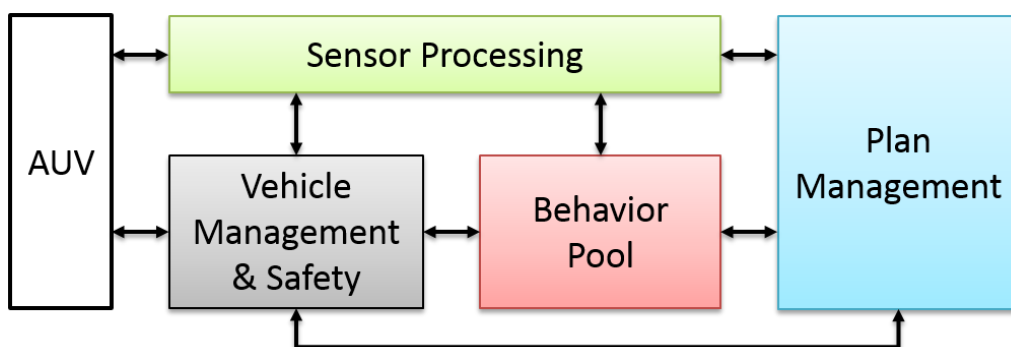


Figure 57 - AUV AVALON Control Architecture

The AUV AVALON (Autonomous Vehicle for Aquatic Learning, Operation, and Navigation) was used during experimentation to detect leaks on underwater pipes and report to a control point. The

team was successful in completing several test missions which proves the viability of their behavior pool. Their composition, however, follows a less modular approach than this project hopes to define.

2.2.2 Nested Autonomy Architecture

Since low bandwidths, latencies, and other uncertainties prevent underwater networks from relying heavily on acoustic communications, the researchers behind [37] developed the Nested Autonomy Architecture. Its goal is to overcome these limitations by making nodes intelligent enough to “take action to the changing environmental conditions without depending much on the communication infra-structure.” The behavior engine of their design is the interval programming based IvP-Helm, a mathematical model for multi-objective optimization. Each objective, or behavior, is a distinct software module dedicated to reaching a certain goal. These objectives, when operating concurrently, compete for control of the vehicle and the winner is chosen by the IvP Solver. Each behavior must provide a utility function which the solver uses, after evaluating heading, speed, and other commands, to determine the most appropriate behavior.

2.2.3 AUV Aided Node Localization

When UASNs are placed without ties to fixed objects they are subject to movement due to high currents and rolling. These changes can be reconciled by periodic surveys performed by AUVs to determine the node’s new location and report it back to the field operators. In [38], two preplanned vehicle trajectories were proposed to facilitate node interrogation and localization with no a priori infrastructure or synchronization. When performing a lattice like trajectory covering the entire known field typical localization was around 90%. The study then evaluated an Archimedean spiral

trajectory where typically only about 60% of the nodes were localized. Obviously the lattice localization method achieved better results but at a cost of more distance travelled. This method may be improved by the injection of reactive behaviors into the subsystem's control framework.

The authors in [39] discuss network layouts deployed in hostile environments via ship or artillery. Naturally, the positions of these nodes are known only to a very coarse bounding box. The authors go on to present methods for determining node location based on trilateration and multilateration techniques. While briefly mentioning the existence of mobile nodes, the paper avoids direct discussion of the level of autonomy involved with the ideal positioning of the mobile asset. The techniques discussed were based on static nodes but should easily be extrapolated for use onboard a mobile node. Furthermore, the experiments were confined to a simulation environment and thus require further field testing.

2.3 Modular Open Systems

Since 1994, the DoDs Open Systems Joint Task Force has been overseeing the implementation of open system specifications and standards on weapon system acquisitions. Previously defined as a Modular Open Systems Approach (MOSA), the Open Systems Architecture (OSA) strategy seeks to ensure that scientists and engineers design for affordable change, employ evolutionary acquisition and spiral development, and develop an integrated roadmap for system design and development [40]. Five major principles underlie its successful realization: Establishing a MOSA enabling environment, employment of modular design, designation of key interfaces, use of open standards for key interfaces, and certifying conformance [41].

2.3.1 Joint Architecture for Unmanned Systems / SAE AS4

The Joint Architecture for Unmanned Systems (JAUS) was conceived by the United States Department of Defense. Since that time it has been accepted as a Society of Automotive Engineers (SAE) International standard known as AS-4. It was an effort to reduce logistical requirements and maximize the interoperability of unmanned assets. It is a service-based communications architecture that abstracts inter-process communications to a structure very similar to current TCP/IP networks and defines an extensible message set used by 'components'. This allows any JAUS compliant systems to communicate and exchange information in a compact format with little knowledge of the other component's structure. For example, a COTS controller capable of sending JAUS command and control messages would be able to drive any JAUS compliant robot to which it was attached, eliminating the need for operator training on multiple operator interface systems. Many of today's proprietary unmanned vehicles require that you also use their customized controller. The result is higher costs logistically, longer lead times, and longer learning curves for individual users. A good analog to promote the open architecture standard is the Universal Serial Bus (USB) standard. Like clockwork, when plugging one into your computer running the Mac, Linux, or Windows operating system, it automatically registers and works properly. JAUS has similar goals. When a new JAUS based vehicle comes online it should register and begin working immediately. This goal has not been fully realized by the DoD and is not trivial to implement. Tools to ease the transition though are well into development. OpenJAUS 4.0, for example, is freely available in C/C++ at <http://www.openjaus.com/>⁴.

⁴ Those wishing to become familiar with the protocol without paying for the most recent documents can find the last standard freely published (2007) on the openJAUS website (Reference Architecture 3.3).

2.3.1.1 JAUS Hierarchy

J AUS breaks down the hierarchy of a system into subsystems, nodes, components, and instances as shown in Figure 58. Keep in mind that, although the J AUS documentation suggests certain configurations of nodes and component it does not require them. The identification and subsequent grouping of software capabilities into nodes is fully in the system engineer’s hands.

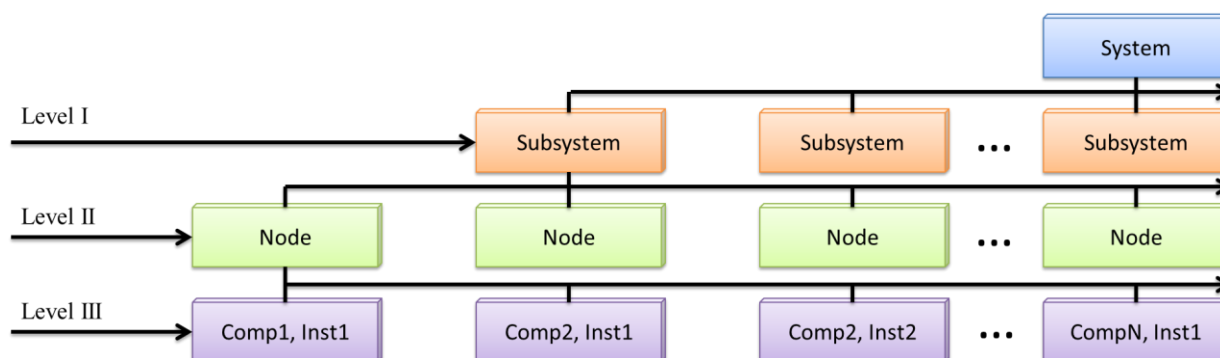


Figure 58 –The structure of a JAUS system. The highest level is system and the lowest level is instance. In between are subsystem, node, and component.

System Level

The system level is the highest layer in the hierarchy and comprises all of the vehicles, controllers, and support payloads chosen for a mission. For example, in a nominal underwater search mission, your system may consist of an Operator Control Unit (OCU), a gateway buoy, and three autonomous underwater vehicles.

Subsystem Level

Subsystems are individual units within the system. They cannot comprise small groupings of vehicles. This would be a separate system. Subsystems are required to execute mobility commands as a single unit and retain a defined center of gravity with respect to any articulated arm

movements. Following the nominal underwater search mission scenario, the subsystem would be an individual unit in the system, such as the AUV or the OCU. Subsystems may provide any number of computing nodes, however, that represent the subsystems functional abilities.

Node Level

A JAUS node represents a distinct processing capability within a subsystem. It must always have a node manager component to route messages to the correct component contained within the node. For example, a mobility controller onboard our AUV would be a node. The components contained therein might be a forward thruster and fin controller.

Component/Instance Level

Components are the lowest level in the hierarchy and provide a unique functionality to the subsystem. For example, a forward looking camera or sidescan sonar might be a component. If multiple cameras or other hardware devices are available the redundant functionality can be duplicated through ‘instances.’

2.3.1.2 JAUS Header

While the software configuration of JAUS components is very flexible, JAUS maintains one absolute requirement: *To achieve the desired level of interoperability between intelligent computing entities, all messages that pass between JAUS defined components shall be JAUS compatible messages* [42]. Figure 59 depicts the JAUS header. This collection of 16 bytes precedes every JAUS messages and provides the system with enough information on how to handle the

message and decode the information. Contained within the various sections are placeholders for origination, destination, message size, and message content.



Figure 59 - Joint Architecture for Unmanned Systems (JAUS) Header that accompanies all JAUS messages

2.3.1.3 Sample JAUS Configurations

A simple vehicle with no autonomy might consist of a system commander and a primitive driver.

System Commander

A system commander coordinates all activity within a given system. It issues commands to the subsystem vehicles and queries them for status updates. These duties may be performed by a human or by a computer.

Primitive Driver

The primitive driver performs all basic driving functions of a vehicle including common operations of engines and lights. It is not transport mode specific but rather it describes movement in six

degrees of freedom and the onboard functionality of the device must translate that into appropriate movement commands.

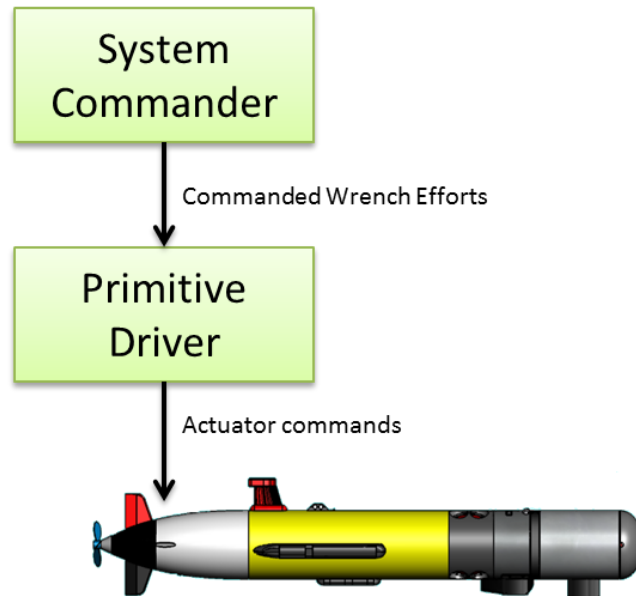


Figure 60 –A simple JAUS configuration comprising only a system commander and primitive driver.

A vehicle in this configuration would offer only simple teleoperational control. With the right sensors and proper autonomy algorithms in place, a reflexive driver could be added to enhance the vehicle's performance.

Reflexive Driver

Normally, the reflexive driver simply passes through a commanded wrench effort. However, if that motion would result in harm to the vehicle or an otherwise unsafe condition, it will modify this wrench effort to maintain safety. For example, if our AUV is fitted with a forward looking sonar meant to report the range to the nearest approaching obstacle the reflexive driver would maintain course unless to avoid an upcoming hazard.

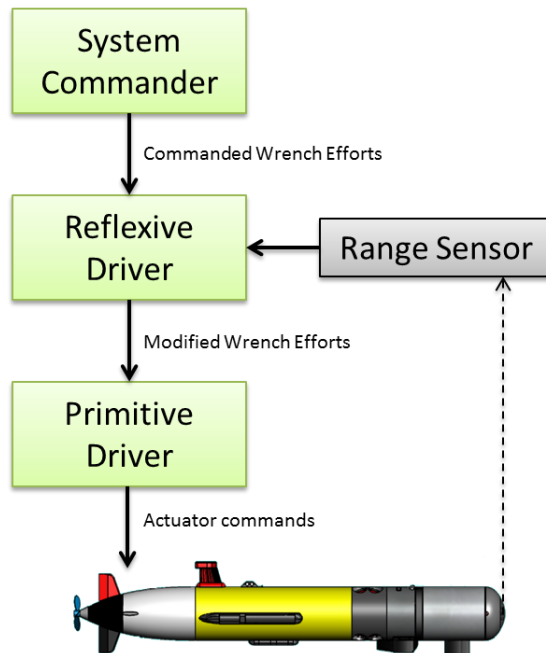


Figure 61 - A simple JAUS configuration comprising a system commander, reflexive driver, and primitive driver, capable of obstacle avoidance.

2.3.2 CARACaS

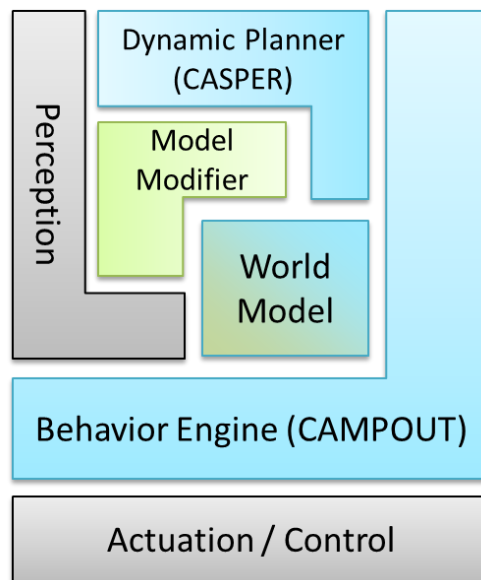


Figure 62 –A block diagram of the major CARACaS system.

The Control Architecture for Robotic Agent Command and Sensing (CARACaS) was developed by NASA's Jet Propulsion Laboratory (JPL). It utilizes the Robust Real-time Reconfigurable

Robotic Software Architecture (R4SA) as its underlying real-time engine. It comprises a dynamic planning engine, known as CASPER, a behavior engine, known as CAMPOUT, a perception engine, and a world model (shown in Figure 62).

2.3.2.1 R4SA

The Robust Real-Time Reconfigurable Robotics Software Architecture (R4SA) was developed by JPL in ANSI C as an adaptable architecture for Mars rovers, terrestrial robots, or even robotic toys [43]. It boasts a modular design able to meet hard real-time aerospace requirements. It comprises 3 layers between the hardware and the system software: The device driver layer, device layer, and application layer (shown in Figure 63).

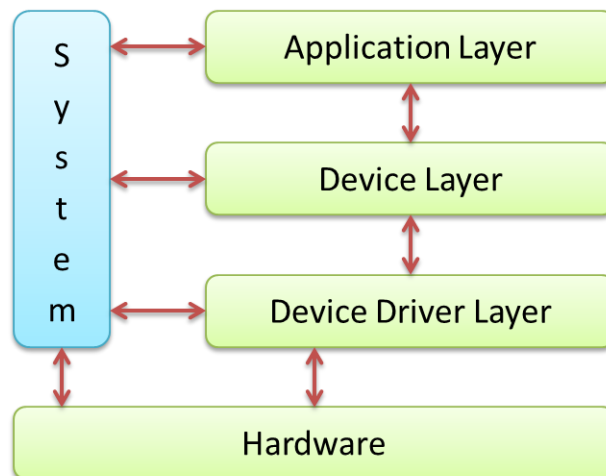


Figure 63 - The R4SA Architecture comprising an Application Layer, a Device Layer, and a Device Driver Layer. The System Layer coordinates the activities of the other layers.

Device Driver Layer

The Device Driver Layer of the R4SA architecture abstracts the functionality of many common robotic associated hardware devices in much the same way that an Operating System abstracts the

functionality of common computer equipment (keyboard, mouse, speakers, etc...). Examples of such are analog-to-digital and digital-to-analog converters. Access to the hardware is provided through ISA or PCI buses. Components are either added or removed by compile time switches in the configuration file.

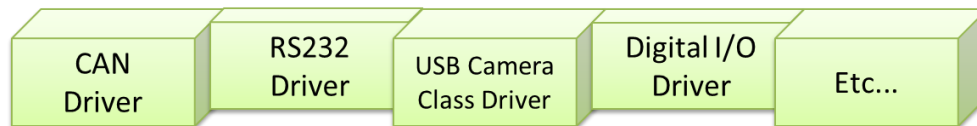


Figure 64 –Some of the key components of the device driver layer of R4SA

Device Layer

The Device Layer abstracts common interactions between the higher level software and the lower level devices. For example, steering commands from high level software are transformed at this level to the necessary motion compensated commands for proportional-integral-derivative (PID) controllers. Transformations are also available for devices such as wheels and arms, odometry and inertial navigation, vision processing, communications, and kinematics for multi-wheel or multi-leg robots. Once again, components may be added or removed by specific compile time switches in the configuration file.

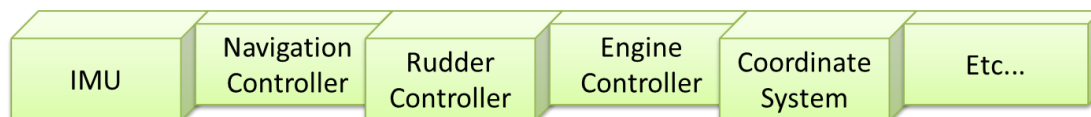


Figure 65 - Some of the key components of the device layer of R4SA

Application Layer

The application layer provides the user with means to execute prescribed behaviors such as obstacle avoidance and path following. The selected behaviors are translated into commands that

are transformed by the aforementioned layers to correctly demonstrate the desired behavior. For example, an obstacle avoidance algorithm housed in the application layer will generate rudder controller commands to be sent to the device layer. As in the previous layers, many of these algorithms are platform specific and will be included based on compile time switches in the configuration file.

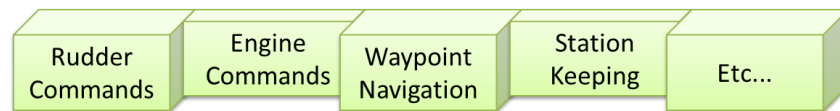


Figure 66 - Some of the key components of the application layer of R4SA

System

The system layer coordinates activities that take place in the other layers to provide a synchronized control environment. It provides a command processor, telemetry display and data logging, a continuous sequence scheduler, a configuration processor, and a system processor.

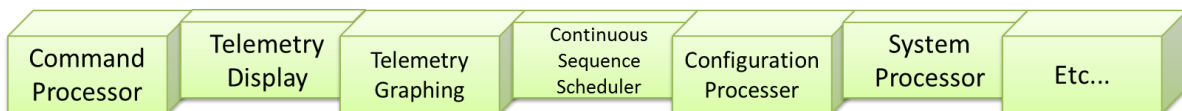


Figure 67 - Some of the key components of the system layer of R4SA

2.3.2.2 CASPER

The Continuous Activity Scheduling Planning Execution and Re-planning (CASPER) portion of CARACaS is the dynamic planner. It continuously monitors objectives and vehicle state to give the most beneficial return of a mission given hard resource constraints. Re-planning may take effect in the event of hardware failures such as an individual battery unit failure.

2.3.2.3 CAMPOUT

The behavior engine of CARACaS is called the Control Architecture for Multi-robot Planetary Outposts (CAMPOUT). It uses Finite State Machines (FSM) to describe desired behaviors. The coordination of these behaviors utilizes a consensus building method based on Multi-Objective Decision Theory (MODT) to form an optimal set of control actions.

CHAPTER 3

DEVELOPING A TEST PLATFORM

The research goal of this project was to determine the validity of using behavior trees as vehicle behavior controllers and, if valid, to describe 3-dimensional, reflective patterns and practices for doing so. The deliverable is a proof of concept using a collection of behavior tree algorithms capable of coordinating the navigation and sensor controls of a simulated machine through a non-trivial, 3-dimensional environment. The environment hosts simulated acoustic nodes that the vehicle attempts to localize. A framework of classes and patterns was also developed. This framework comprises a simulated world-like environment and simulated acoustic nodes, both stationary and mobile. The development of these simulated components follows the DoDs open system guidelines as discussed in section 2.3. All component configuration files are crafted using XML and follow current industry standards if applicable. A graphical representation of the framework is shown in Figure 68, below.

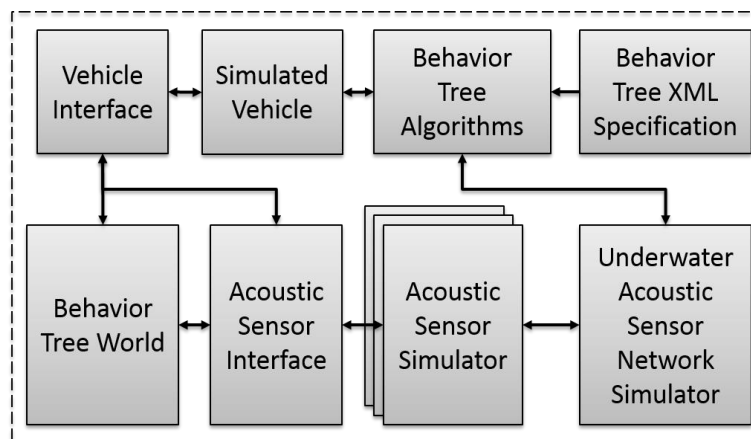


Figure 68 - Graphical representation of proposed software framework

Each of the following sections describe a component of this framework. The title of each section has, in parentheses, the name of the component used in the framework implementation.

3.1 Behavior Tree World (BTWorld)

Providing evidence for the validity of using behavior trees as vehicle behavior controllers requires an ample demonstration in an environment similar to what they would experience in the real world. Thus, a 3-dimensional software model of the world's terrain was developed. The implementation was realized through the use of MonoGame (see section 1.6.1.1), which can further target either Microsoft's DirectX or other platform's OpenGL capability. The terrain itself is generated in code by analyzing a height map in the form of a 32-bit bitmap image. Each pixel of these bitmap images are 4-bytes, representing 8-bits each for the red, green, blue, and alpha (transparency) values. The 'red' value of each RGBA (Red-Green-Blue-Alpha) pixel is used to set the height of the land at that location. For example, the height map of Figure 69 (below, left) generates the terrain shown and colors it in based on how high the red-byte of each pixel is. Lower elevations are colored blue to indicate water while higher elevations are colored white to indicate snow. The heights and colors of individual cells is made available to vehicles and algorithms so that decisions can be made about the validity of certain actions, like traveling through water or up a very steep cliff.

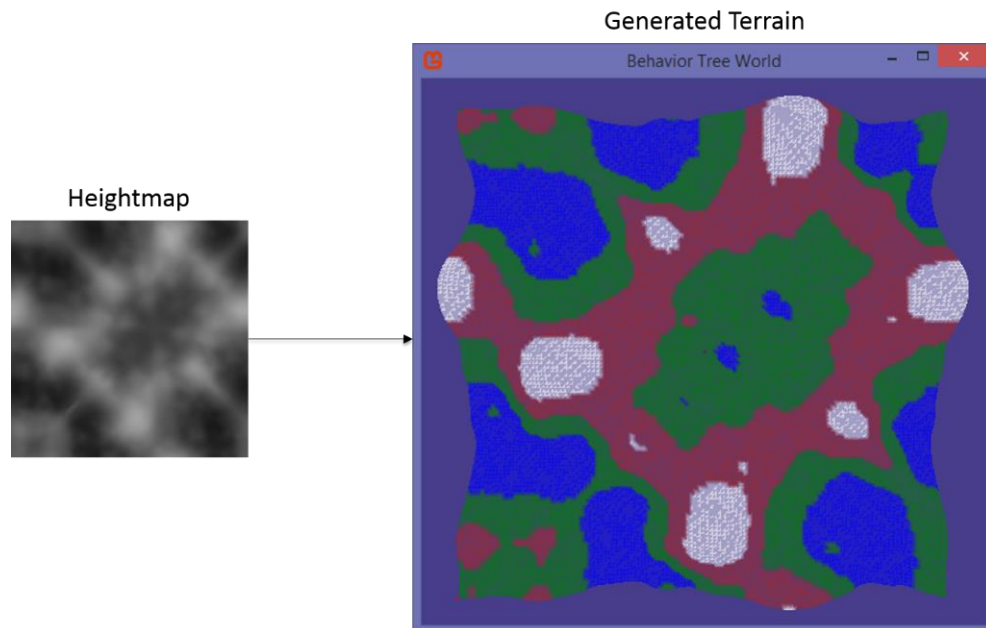


Figure 69 - A sample terrain (right) generated by a bitmap image (height map, left)

The use of height maps is advantageous in open systems because it allows unhindered access to terrain editors using off the shelf image manipulators such as Microsoft Paint, GIMP, or Adobe Photoshop.

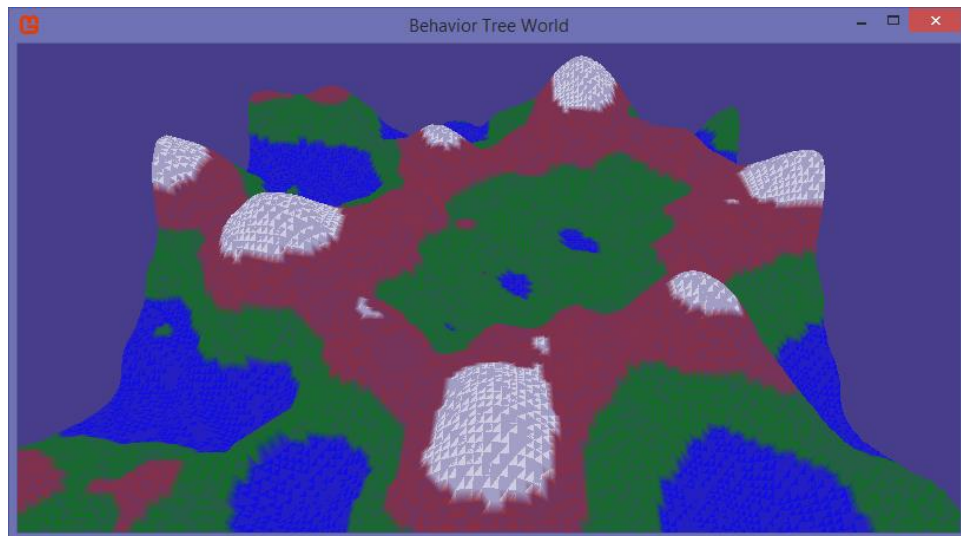


Figure 70 - A close-up look across the horizon of the behavior tree world, showing its 3-dimensional nature.

3.1.1 Helpful Game Semantics

Professionals and hobbyists in the game development field have developed semantics that quickly and concisely convey important information. This section will introduce the reader to some of these semantics for use in understanding the sections that follow.

Camera – In a video game, the camera would be considered the location at which all of the light from images is focused for viewing by the user. A simple analogy would be that of the viewfinder on a digital camera. Think of the game window (window in the operating system sense) as the viewfinder on a camera. As you move the camera, its perspective on world objects changes and you see different sides of things. In BTWorld, as you navigate the terrain, you are in a sense moving a camera around and seeing the world through the camera's viewfinder.

Fog of War – Many games, especially Real-Time Strategy (RTS) games, present a map of unexplored territory with abundant resources that the player has to look for and find. These resources are sometimes hidden on the map until the player sends one of their actors (people or vehicles) to find them. As the actor searches the landscape the blanket of fog is removed. This ‘fog’ is frequently referred to as a Fog-of-War. Updates to the landscape, such as enemy movement or building, are usually hidden also unless the player has an actor with a line of sight to where the activity is taking place.

Ticks – One of the useful properties of game development that has evolved over the years is the use of two main operational cycles: The (1) update cycle & (2) draw cycle. The update cycle iterates through each of the game components causing it to update its state based on current environment variables. The draw cycle draws all of the necessary components onscreen according to their world property and the current camera location. Each completion of the update and draw cycle is called a *tick*. Most games try to achieve 60 ticks per second which fools the human eye into seeing a continuous movement on screen.

World – This particular term invites confusion since it already has a well-defined meaning. Whenever the ‘world’ property of a component is mentioned, the author is describing that particular component’s translation and / or rotation with respect to the X, Y, and Z axis origin. The confusing aspect of this term is that each model within the game has its own world value that differs from all of the other models. So, in effect, there are many different worlds all coexisting within close proximity to each other.

3.1.2 Configuration

When instantiated, the terrain component loads the file `terrain.xml` (Figure 71). The `<Settings>` section of the configuration file is a collection of key=value pairs. Any number of these may be inserted but the *HeightMapFile* and *MapScaleFactor* are the only two that are necessary for the **TerrainBase** class, which is what all proprietary terrain files should inherit from. The **TerrainBase** class stores these pairs into a protected dictionary that any child class can access for necessary functionality. Alternately, the child classes may read in their own xml configuration files. After reading and storing the values in this file, the **TerrainBase** class will retrieve the given heightmap file. It then scans it and creates a 3D landscape. The *MapScaleFactor* value is used to tamp down every given pixel's R value. For more mountainous terrain use a high factor. For flatter terrain, use a low factor.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <Settings>
    <!--This is where key=value pairs will go-->
    <Setting key="HeightMapFile" value="Maps/heightmap08"/>
    <Setting key="MapScaleFactor" value="0.1"/>
  </Settings>
  <Layers>
    <!--Layers define how the terrain should appear on the screen-->
    <Layer Name="Water" Percentage="0.25" Color="Blue" Vary="false"/>
    <Layer Name="Grass" Percentage="0.50" Color="Green" Vary="true" VaryLo="#006700" VaryHi="#329932"/>
    <Layer Name="Dirt" Percentage="0.75" Color="Brown" Vary="true" VaryLo="#9B2A27" VaryHi="#AF3E2D"/>
    <Layer Name="Snow" Percentage="1.00" Color="White" Vary="true" VaryLo="#EBEBEB" VaryHi="#FFFFFF"/>
  </Layers>
</Configuration>
```

Figure 71 - The `terrain.xml` file used to configure the BTWorld terrain.

The `<Layers>` section of the configuration file determines what the landscape will look like rendered onscreen. Each layer requires a *Name*, *Percentage*, and *Color*. The name is simply for reference in code. The percentage attribute determines which pixels belong to which layer. For example, since the water layer is assigned a percentage of 25, any pixels in the lowest 25% of all of the 'R' height values will fall into this layer and be assigned the color blue. If a user wants more

water, they can increase the percentage of pixels that fall into this category (e.g. set `Percentage="0.40"`). Since the grass layer is assigned a percentage of 50, any pixels in the lowest 50% of all 'R' height values *that aren't already accounted for* will fall into this layer and be assigned the color green. Finally, the color attribute assigns a color to the pixel. The *Vary* attribute is optional and defaults to false. It determines whether or not to 'smudge' each color to give the landscape a more realistic look. Since each pixel is determined by an RGBA value, the allowed variance of each byte is given in the respective *VaryHi* and *VaryLo* attributes. These two attributes are given as hexadecimal html-color representations. If a user wanted the water to look like lava instead, all they would have to do is change the name of the first layer to lava and its color to red.

3.1.3 Input

MonoGame was originally developed as a video game creation API so human interfacing is heavily integrated. In BTWorld, the user assumes control of a camera component with 6-axes of freedom. Navigation is handled mostly through mouse input although keyboard input is in place as a secondary controller. When BTWorld is first executed, the camera view into the world is located at a position above the landscape looking straight down. To move forward, the user can either roll the mouse wheel up or press the 'W' button. To look up, down, or side to side, the user can click the left mouse button and then slide it up, down, and side to side. For keyboard control, the user can use A and D to look side to side, R and F to look up and down, and Q and E to strafe side to side. The keys Z and C allow the user to 'roll' the camera clockwise and counter-clockwise.

3.1.4 Navigation

When rendered in wireframe mode, the **BTWorld** terrain takes the appearance of an undirected graph like the one shown in Figure 72. Each vertex is a discrete point at which a model can rest

and the connecting edges are paths that a model can take to travel between them. The vertices are made available to other actors through a class called **CellLandscape**. This class contains a 2-dimensional array of Cells. These cells contain all of the pertinent information regarding each location. Cells must implement the ICell interface shown in Figure 73.

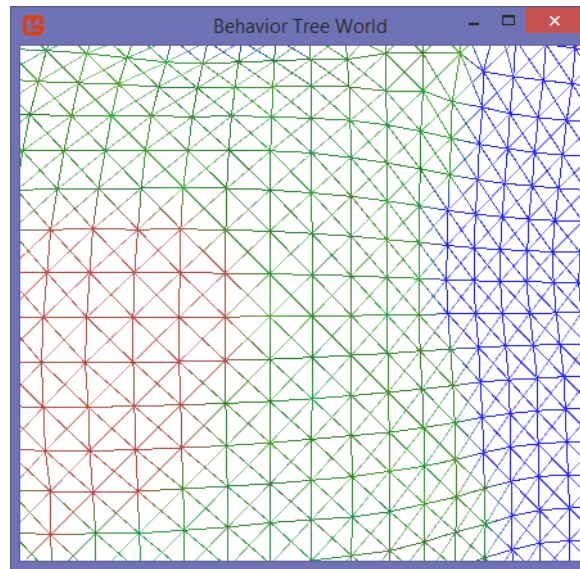


Figure 72 - The wireframe mesh representing traversable paths in the Behavior Tree World

```
public interface ICell
{
    int X { get; set; }
    int Y { get; set; }
    int Index { get; set; }
    bool Visited { get; set; }
    float DistanceSoFar { get; set; }
    ICell CameFrom { get; set; }
    List<ICell> Neighbors { get; }
    Color Color { get; set; }
    void Reset();
    void UndoColor();
    bool IsHidden { get; set; }
    bool IsPassable { get; }
    bool IsPassableFrom(ICell cell);
    VertexPositionColor VertexInfo { get; }
    Vector3 Vertex { get; }
    CellLandscape Landscape { get; set; }
    float DistanceTo(ICell cell);
}
```

Figure 73 - The contents of the ICell interface

3.1.5 Fog of War

The **BTWorld** is optionally covered by a blanket of fog indicating unexplored territory. Accounting for this fog in calculations and uncovering it is up to the developer. In this research paper we uncover the map through the use of the **Traveler** by way of its sight radius. These points will be discussed in Chapter 4, Developing World Actors. The unexplored map areas will be presumed traversable and flat when used in our algorithms calculations. As the map is uncovered, non-traversable land and steep inclines will be accounted for so that more accurate paths will be determined.

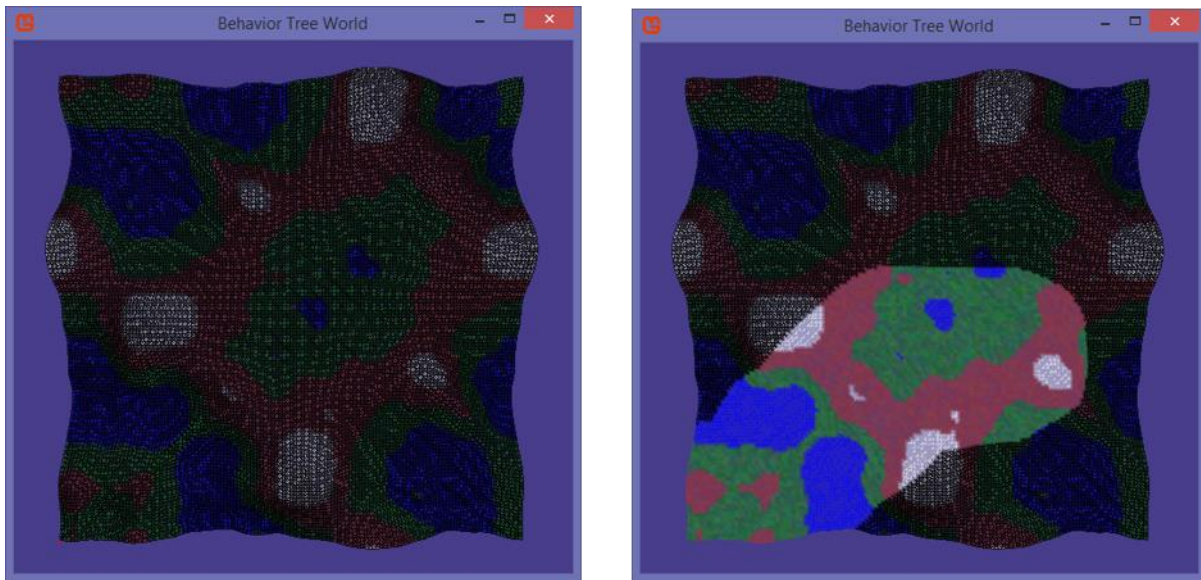


Figure 74 - A representation of the Fog-of-War capability of BTWorld. The terrain in the left image is completely unexplored. The terrain on the right has a wide swath cut through it that shows where the vehicle has been.

CHAPTER 4

DEVELOPING WORLD ACTORS

The Behavior Tree World exists as the foundation of the experiments and represents the real world obstacles that help or hinder progress. The actors representing proxies to achieve our objectives will be a vehicle simulator and one or more static acoustic node simulators with attached modules. The vehicle simulator and acoustic node simulators will be discussed in this chapter. The optional modules are mainly just hooks for future developers to add custom functionality to the design.

4.1 Vehicle Simulator (Traveler)

A simple vehicle simulator has been developed to navigate the **BTWorld** mesh. It has two basic functionalities. When commanded to move, it simply moves in that direction. When queried about its current location, it returns whatever information is available to it. Vehicle dynamics are not the theme of this discussion and have thus been ignored. Decision making was not built into the simulator either. It was deferred until creation of the behavior tree framework itself. Control can be assumed either manually using the keypad or by a collection of behavior tree classes/algorithms that were designed to intelligently navigate through the **BTWorld** mesh. The default implementation of the vehicle has a dictionary of objects (modules) making it capable of housing any future developed capability that conforms to the interface. The three main modules used in this research were a communication module, a driver module, and a locator module. Each of these will be discussed later within the context of their usefulness to the design. Behavior-Tree algorithms designed to account for terrain aspects and make intelligent decisions about navigating it have also been developed. The details of these classes will be discussed in Chapter 7, Behavior Tree Application Results.

4.1.1 Configuration

Configuring the vehicle involves setting several key variables in the Travelers.xml file (Figure 75).

Notice that each traveler has a collection of modules. These modules are all optional and contribute additional functionality to the traveler.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <Travelers>
    <Traveler>
      <Setting key="name" value="Vehicle"/>
      <Setting key="pathHeight" value="1"/>
      <Setting key="color" value="red"/>
      <Setting key="sightRadius" value="25"/>
      <Setting key="enabled" value="true"/>
      <Modules>
        <Module type="Communicator">
          <Setting key="range" value="50"/>
          <Setting key="ID" value="0"/>
        </Module>
        <Module type="BTDriverState"/>
        <Module type="BTLocator">
          <Setting key="delay" value="5000"/>
        </Module>
      </Modules>
    </Traveler>
    <Traveler>
      <Setting key="name" value="TargetComm"/>
      <Setting key="pathHeight" value="1"/>
      <Setting key="color" value="purple"/>
      <Setting key="sightRadius" value="0"/>
      <Setting key="enabled" value="false"/>
      <Modules>
        <Module type="Communicator">
          <Setting key="range" value="50"/>
          <Setting key="ID" value="1"/>
        </Module>
      </Modules>
    </Traveler>
  </Travelers>
</Configuration>
```

Figure 75 - The Travelers.xml file used to dynamically create 'travelers' at runtime in our Behavior Tree World

Just like in previous framework components, the *name* is used just to differentiate modules of the same type. The *pathHeight* variable is used to determine how high above the terrain to draw the vehicle marker. The *color* is used to differentiate between travelers on the terrain while the simulator is running. The *sightRadius* value is used to determine how much of the map has been

‘seen’ by the vehicle. The *enabled* variable is used to determine if this game component’s world property should be updated during each game tick. The enabled property is set to true when the vehicle is selected and false when the vehicle is deselected.

An alternative to indicating all of the required modules in the Travelers.xml file, the modules can be declared in a separate file and noted as a `<File name="filename.xml"/>` element. The loader will then locate the file, gather all of the modules, and add them to the traveler.

4.2 Acoustic Node Simulator (Communicator)

A static sensor simulator has been developed to emulate the functionality of an acoustic communication node. In this framework they will be passive, meaning that they only ‘speak’ when prompted. The term ‘speaking’ is used since the typical frequency employed by acoustic nodes is audible to the human ear, much like a voice, and is used to carry information, like any human language. The node is implemented by a class called **Communicator**.

The communicator’s function is very basic. When provided a message in the *SendMessage* method, it checks the distance to all fellow nodes in the field. If they are within the node’s maximum communication range it calls *RegisterMessage* on that node. It then becomes the responsibility of the receiving node to formulate a response. In this framework, the **Communicator** sends out a delayed response based on the distance between itself and the sender. The delay calculations are currently based on the speed of sound underwater (approximately 1500 meters per second).

4.2.1 Configuration

Configuring the **Communicator** requires only two setting, *range* and *ID*. The *range* value indicates how far the communicator's transmissions will travel and the *ID* is an indicator of who messages are coming from and being sent to. For example, the communicator with id 0 might address messages to the communicator with id 1.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <Modules>
    <Module type="Communicator">
      <Setting key="range" value="50"/>
      <Setting key="ID" value="0"/>
    </Module>
  </Modules>
</Configuration>
```

Figure 76 - The contents of Communicator0.xml

CHAPTER 5

BEHAVIOR TREE FRAMEWORK IMPLEMENTATION

The discussion in this section will build on the information first introduced in section 1.4, Behavior Trees. If a particular point about behavior tree operation is unclear, please reference that section for a more thorough explanation.

When behavior tree algorithm development begins the question necessarily arises: What does it mean to use a Behavior Tree Algorithm? The answer, simply, is that you confine the components of any algorithm you develop to the constructs described in section 1.3 (page 14). These constructs again are sequences, selectors, decorators, actions, and assertions (Figure 77). The reason for such simplicity is the concept of reuse across behavioral spectrums, which was a major goal of this project. Also, by utilizing these atomic blocks, entire behavior trees can be deconstructed and rebuilt without modifying the underlying code using reflective class construction techniques (discussed below in section 6.4)

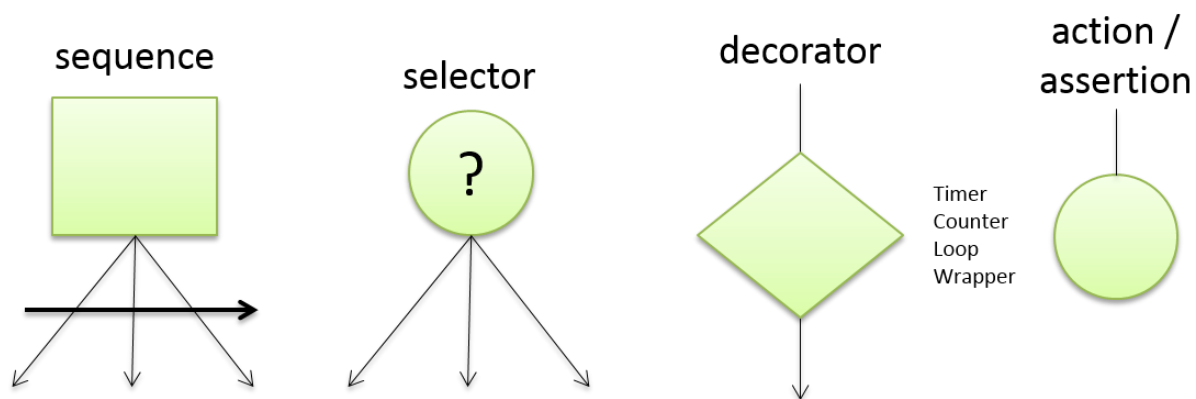


Figure 77 –The fundamental building blocks of a behavior tree: sequence, selector, decorator, action, & assertion. Also known as ‘nodes’

5.1 Fundamental Interfaces

In order for these ‘nodes’ to be completely interchangeable, a series of interfaces were developed (see section 1.7.1 Interface Based Programming). For basic functionality, only three interfaces were deemed necessary. As development matured, generic counterparts were also developed. These will be discussed in a later section. The initial interfaces were IComposite, IDecorate, and INode. Since IComposite and IDecorate also implement INode, they can be used interchangeably anywhere an INode is required.

```
public interface INode
{
    string Name { get; }
    INode Parent { get; set; }
    NodeStatus Tick();
    NodeStatus Status { get; set; }
    void Initialize();
    int Count();
    void FindNode(int number, List<INode> nodes);
    bool RunAsync { get; set; }
    bool HasEntrant { get; }
}
```

Figure 78 - The contents of the interface INode

```
public interface IComposite : INode
{
    List<INode> Children { get; }
    int NumChildren { get; }
    void AddChild(params INode[] node);
    void RemoveChild(INode node);
    INode GetChildAt(int index);
}
```

Figure 79 - The contents of the interface IComposite

```
public interface IDecorate : INode
{
    INode Child { get; set; }
}
```

Figure 80 - The contents of the interface IDecorate

The IComposite interface supplements INode by adding the ability to contain and manage several child nodes. The IDecorate interface supplements INode by adding the ability to contain just one

child. As you will recall from the introduction on behavior trees, composite nodes are defined by how they activate their children.

To make the development of extensions easy, abstract base classes have been created to satisfy the requirements of each of these interfaces. Developers wishing to supplement the current functionality need only subclass one of these abstract classes and implement an *Update* method. A graphical hierarchy is shown in Figure 81.

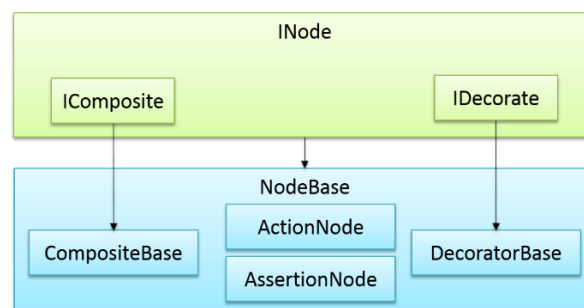


Figure 81 - A graphical representation of the fundamental interfaces and base classes of the behavior tree framework

The **NodeBase** class alone implements all of the necessary functionality of an **INode** except for one abstract method called *Update*. Any time a class defines an abstract method, the compiler requires that any subclasses implement it. This *Update* method, along with the cardinality of children, is what sets each type of behavior tree node apart from the others.

```
protected abstract void Update(); /* subclasses MUST implement this or else a compile time error occurs */
```

The **NodeBase**'s definition of the required *Tick* method (required by **INode**) simply calls this *Update* method which consequently runs the code located inside the sequence, selector, decorator, or action node implementing the **NodeBase** class. The reason behind implementing the *Tick*

method inside of the **NodeBase** class instead of requiring subclasses to do it is that it allows the **NodeBase** to seamlessly handle the necessary asynchronous capability as well.

5.2 Asynchronous Operation

Whenever a node is set to run asynchronously, the **NodeBase** will instead call *UpdateAsync*. This method then calls *Update* but on a separate operating system thread. The node will return a *NodeStatus* of ‘running’ as long as this separate thread is active (see Figure 82).

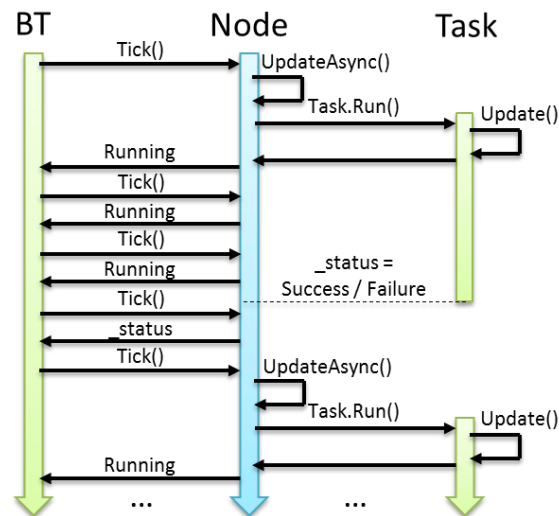


Figure 82 - A sequence diagram showing a behavior tree's operation whenever *RunAsync* is set to true

The C# language has built in support for parallelization of code through the use of Tasks. All nodes will contain a Task that is ‘run’ whenever the Boolean value *RunAsync* is set to true. The return status of the node when running asynchronously will be a reflection of the Task status (see Figure 82). If a task has been created and given the name *_task*, the task’s status can be checked by calling *_task.Status*.

Table 10 – Alignment of TaskStatus enum to NodeStatus enum

Task Status	Node Return Status
TaskStatus.Created	NodeStatus.Running
TaskStatus.Running	NodeStatus.Running
TaskStatus.WaitingForActivation	NodeStatus.Running
TaskStatus.WaitingForChildrenToComplete	NodeStatus.Running
TaskStatus.WaitingToRun	NodeStatus.Running
TaskStatus.Canceled	NodeStatus.Failure
TaskStatus.Faulted	NodeStatus.Failure
TaskStatus.RanToCompletion	// [Success Failure] set by Update method

5.3 Composite Nodes

Sequences and selectors are known as composite nodes because they can contain a collection (List, Queue, Stack, etc) of other nodes as children. They subclass **CompositeBase** which means they also implement the IComposite and INode interfaces. All of the additional required functionality of IComposite nodes, aside from the required *Update* method, is contained here. This greatly simplifies extension since any subclasses will only be required to implement that one method.

```
protected override void Update()
{
    foreach (INode n in Children)
    {
        Status = n.Tick();
        if (Status != NodeStatus.Success) {break;}
    }
    base.Update();
}
```

Figure 83 - The contents of the Sequence Update method

```
protected override void Update()
{
    foreach (INode n in Children)
    {
        Status = n.Tick();
        if (Status != NodeStatus.Failure) {break;}
    }
    base.Update();
}
```

Figure 84 - The contents of the Selector Update method

In their basic form, sequences activate their children one by one, in order, until all of them complete successfully. It then returns a NodeStatus of success. If one of its children fails, it halts the

activation of children and returns a `NodeStatus` of failure (see section 1.4.2). Selectors, on the other hand, activate their children in order until one of them returns success and then it halts. This is similar to the way an if-else statement operates (see section 1.4.3).

5.4 Decorator Nodes

Decorator nodes are simply links in a chain between calling nodes and the nodes below them in the graph. Most of the special functionality required by any extensions to this behavior tree framework will reside here. For example, user defined exception handlers could be placed in a custom decorator. This would simply require a try / catch block to be inserted around the call to its child's *Tick* method with appropriate actions to take for each exception type. As with the other predefined types, an abstract base class called **DecoratorBase** has been developed to satisfy the requirements of `IDecorate`. Any subclass of **DecoratorBase** will only be required to implement the *Update* method. For a more thorough discussion of decorator nodes see section 1.4.4.

5.4.1 While-True Decorators

The need for a special 'while-true' decorator has presented itself in many of the algorithms. This specialty node will tick its child node continually until it receives a `NodeStatus` of failure (see Figure 85).

```
protected override void Update()
{
    do
    {
        Status = Child.Tick();
    } while (Status == NodeStatus.Success);
}
```

Figure 85 - The Update method of a while-true decorator

5.4.2 Exception Decorators

Exception handling is a necessary component of any serious software development enterprise. Therefore, a special decorator designed to handle this aspect of development has been created. The decorator is supplied with one additional method: `AddExceptionHandler(Exception e, Func<NodeStatus> handler);` This method accepts an exception of any type along with a `Func` to handle that exception. This `Func` must return a `NodeStatus` to indicate whether the handler was successfully able to rectify the situation. If not, a `NodeStatus` of failure should be returned. The modification to the *Update* method of this decorator involves surrounding the call to its child's *Tick* method with a try / catch block. The catch block will search its dictionary of handlers to see if there is one specified for whatever exception was thrown. If there is, it will call that handler and set its own status to the result of the handling `Func`.

```
private Dictionary<string, Func<NodeStatus>> _handlers;
protected override void Update()
{
    try
    {
        Status = Child.Tick();
    }
    catch (Exception ex)
    {
        string typeName = ex.GetType().FullName;
        if (_handlers.ContainsKey(typeName) == true)
        {
            Status = _handlers[typeName]();
        }
        else
        {
            Status = NodeStatus.Failure;
        }
    }
}
```

Figure 86 - The contents of the ExceptionDecorator's Update method

5.5 Action / Assertion Nodes

Action and Assertion nodes are helper nodes that relieve the developer from reinventing similar functionality for each software product. They are wrappers for `Func` delegates that return a `NodeStatus` or a `Boolean`, respectively. When functions or class methods match the signature of

the required Func (Func<NodeStatus> or Func<bool>) they can be handed to the class constructor without modification (see Figure 88 - The code behind an AssertionNode). The compiler handles the cast for the developer. This eases the custom code development required for crafting behavior trees. However, when utilizing reflection to assemble them, a more creative approach will be required.

```
public class ActionNode : NodeBase
{
    private Func<NodeStatus> _action;

    public ActionNode(Func<NodeStatus> action)
        : base()
    {
        _action = action;
    }

    public ActionNode(string name, Func<NodeStatus> action)
        : base(name)
    {
        _action = action;
    }

    // Other constructors and methods
    // Update method simply returns the value received from _action
}
```

Figure 87 - The code behind an ActionNode

```
public class AssertionNode : NodeBase
{
    private Func<bool> _assertion;
    public AssertionNode(string name, Func<bool> assertion)
        : base(name)
    {
        _assertion = assertion;
    }
    // other constructors and methods
    protected override void Update()
    {
        if (_assertion() == true)
        {
            Status = NodeStatus.Success;
        }
        else
        {
            Status = NodeStatus.Failure;
        }
    }
}
```

Figure 88 - The code behind an AssertionNode

```
public NodeStatus MyMethod()
{
    return NodeStatus.Success;
}

public void CreateActionNode()
{
    ActionNode actionNode = new ActionNode(MyMethod);
}
```

Figure 89 - The instantiation of an ActionNode using a regular class method

5.6 Generic Counterparts

In order for a behavior tree to share and operate on vehicle state, each node must accept operable values such as native types, classes, or structs. In order to accommodate this, I have developed single parameter, generic counterparts to each of the main components previously described. C# generics, like C++ templates, allow the body of a class or struct to be developed without knowing the exact argument types that it will operate on. The difference is that in C# the classes are not constructed at compile time. The arguments remain unknown until runtime and are compiled then, just-in-time (JIT), which allows for runtime optimizations. The need for comprehensive input arguments is negated by the appropriate use of class composition. For example, the TArgs type argument can be a `Tuple<int, string, double, float, Exception []>` which effectively allows operation on four parameters and the collection of any exceptions that may be thrown. If a 'class' type is substituted for TArgs this will allow peer nodes (nodes with the same ancestor) to communicate with each other. Since class types are by default passed by reference, this ensures that changes made to the class will propagate down the call chain and then bubble up to the parents afterward. Since structs are by default passed by value, the changes made in a particular behavior tree node would be passed down the call chain but would become void as the call stack was unwinding.

```

public interface INode<TArgs> : INode where TArgs : class
{
    TArgs MyArg { get; }
    NodeStatus Tick(BTNodeArgs t);
}

public interface IComposite<TArgs> : IComposite, INode<TArgs> { }

public interface IDecorate<TArgs> : IDecorate, INode<TArgs> { }

```

Figure 90 - Suggested interfaces for a generic behavior tree framework

Notice that the interface in Figure 90 also inherits INode. This means that even if a portion of a behavior tree is made generic, its parent nodes do not need to be. They can still ‘adopt’ these nodes as children and call their *Tick* method. A special decorator node called a **GStateComposer** has been developed to bridge the gap between non-generic and generic sections of code. In this decorator’s constructor, a special Func must be supplied that instantiates the proper argument type for that generic section of the behavior tree (see Figure 91). If no state composing Func is supplied, the default constructor (i.e. parameterless constructor) will be called to create the necessary TArgs object. All generic components of this framework are preceded by a ‘G’ in their name.

```

public GStateComposer(Func<TArgs> stateComposer)

```

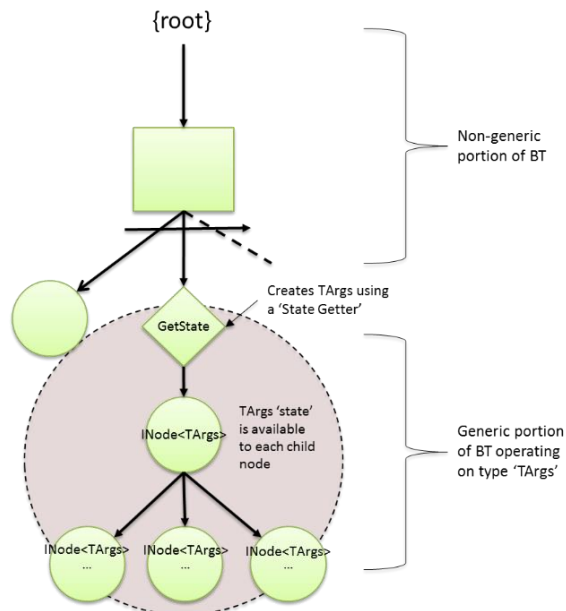


Figure 91 - A Generic Behavior Tree Framework

5.6.1 State Selector Nodes

State Selector nodes will look similar in structure to other composites (i.e. Sequences and selectors). The difference will be that, instead of operating on every child in sequence, it will use the current 'STATE' to determine which child to activate. 'State Machines' are a very popular approach for allowing an object to act differently when presented with the same data over and over again based on the object's current state. The one stipulation to the TArgs argument of this type is that they implement IStateNodeArgs (see Figure 92).

```
public interface IStateNodeArgs
{
    string State { get; }
}
```

Figure 92 - The requirements of the IStateNodeArgs interface

The children of **GStateSelector** composites are stored in a dictionary keyed by a string. When the Update method is called, the child node matching the state is retrieved and its tick method called (see Figure 93).

```
Dictionary<string, INode<TArgs>> KeyedChildren;
void Update(TArgs t)
{
    Status = KeyedChildren[t.State].Tick(t);
}
```

Figure 93 - The Update method implementation of state selector composites.

5.7 Action and Assertion Repositories

Having a common framework and methodology prescribed will increase the ease of reuse across platforms. Up to this point, though, the burden of code writing doesn't seem to be any less than most typical frameworks would require. The following discussion on action and assertion repositories will show that once adopted, this framework will indeed present wide reuse capability and at the same time reduce the authoring requirements of programmers.

5.7.1 Action Repository

As previously stated, action nodes cause the actor to effect a change. These might be commands affecting the vehicle's vector or commands to an acoustic node requesting a query be made. In this framework, action nodes will be represented by the `ActionNode` class (see section 5.5) that accepts an argument of type `Func<NodeStatus>`. The generic variant of `ActionNodes` will accept `Funcs` that take one argument *and* return a `NodeStatus` (e.g. `Func<int, NodeStatus>`). For a more thorough discussion on Actions and Funcs, see section 1.7.2.2 above.

The action repository will be a helper class that facilitates adding `Funcs` to, and retrieving `Funcs` from, a complex collection (a collection within a collection). The outer collection will be a dictionary of sorted dictionaries. It will be keyed by a string representation of the type of arguments that its underlying collection of `Funcs` require as an argument. That is, if a particular `Func` takes an `int` and returns a `NodeStatus` (`Func<int, NodeStatus>`) the key will be "System.Int32." The inner collection will be a sorted dictionary keyed by a string and containing an object as its value. A sample declaration is shown below:

```
Dictionary<string, SortedDictionary<string, object>> _actionRepos;
```

In C#, every class type derives from the 'object' class. By using the underlying object class as the value type in the dictionary we can store any type of `Func` object regardless of what argument type it accepts. When retrieving these objects from the repository there will be a negligible amount of overhead needed to convert them back to their original types. This conversion should only take place once in the algorithm initialization phase, thus negating its impact on performance. See a potential `ActionRepository` implementation in Figure 94.


```

public class ActionRepository
{
    private static Dictionary<string, SortedDictionary<string, object>> _actionRepos =
        new Dictionary<string, SortedDictionary<string, object>>();
    public Func<NodeStatus> GetAction(string name)
    {
        Func<NodeStatus> f = _actionRepos["NodeStatus"][name] as Func<NodeStatus>;
        return f;
    }
    public void RegisterAction(string name, Func<NodeStatus> t)
    {
        if (_actionRepos.ContainsKey("NodeStatus") == false)
        {
            _actionRepos.Add("NodeStatus", new SortedDictionary<string, object>());
        }
        _actionRepos["NodeStatus"][name] = t;
    }
    public void RegisterAction<T>(string name, Func<T, NodeStatus> t)
    {
        string typeName = typeof(T).Name;
        if (_actionRepos.ContainsKey(typeName) == false)
        {
            _actionRepos.Add(typeName, new SortedDictionary<string, object>());
        }
        _actionRepos[typeName][name] = t;
    }
    public Func<T, NodeStatus> GetAction<T>(string name)
    {
        string typeName = typeof(T).Name;
        Func<T, NodeStatus> f = _actionRepos[typeName][name] as Func<T, NodeStatus>;
        return f;
    }
}

```

Figure 94 - A potential implementation of the necessary ActionRepository

To provide access to available repositories and to allow further categorization by users, a static **BTRepositories** class has been developed. The class will simply maintain all added repositories by owner specified category. Its implementation is shown in Figure 95. Since it is implemented as static, it will be available to all users of the framework who reference the behavior tree framework dll in their code.

```

public static class BTRepositories
{
    private static Dictionary<string, ActionRepository> _repos
        = new Dictionary<string, ActionRepository>();

    public static void AddRepository(string category)
    {
        _repos.Add(category, new ActionRepository());
    }
}

```

Figure 95 - A potential implementation of the static BTRepositories class.

```

public static ActionRepository GetRepository(string category)
{
    if (_repos.ContainsKey(category) == false)
    {
        AddRepository(category);
    }
    return _repos[category];
}
}

```

Figure 95 - continued

5.7.2 Assertion Repository

Assertion nodes are similar to action nodes except that they are meant to imply simple Boolean answers to vehicle state or world state queries. These might be queries such as “Is the vehicle capable of traveling in a particular direction” or “Did the modem get a response from another node?” The Boolean response true and false would align with a behavior tree NodeStatus responses of success and failure, respectively (see Table 11).

In the same way as action nodes, assertions may be held in a dictionary keyed by a name. However, since assertions return Boolean values they will necessarily be within a separate domain of dictionaries. Four additional methods have been added to our ActionRepository to accommodate this functionality.

Table 11 – Alignment of Boolean values to NodeStatus enum

Boolean Result	Node Return Status
True	NodeStatus.Success
False	NodeStatus.Failure

```

public void RegisterAssertion(string name, Func<bool> t)
{
    if (_assertionRepos.ContainsKey("Bool") == false)
    {
        _assertionRepos.Add("Bool", new SortedDictionary<string, object>());
    }
    _assertionRepos ["Bool"][name] = t;
}

```

Figure 96 - The additional methods necessary to accommodate assertions in our repository

```

public Func<bool> GetAssertion(string name)
{
    Func<bool> f = _assertionRepos ["Bool"][name] as Func<bool>;
    return f;
}

public void RegisterAssertion<T>(string name, Func<T, bool> t)
{
    string typeName = typeof(T).Name;
    if (_assertionRepos.ContainsKey(typeName) == false)
    {
        _assertionRepos.Add(typeName, new SortedDictionary<string, object>());
    }
    _assertionRepos [typeName][name] = t;
}

public Func<T, bool> GetAssertion<T>(string name)
{
    string typeName = typeof(T).Name;
    Func<T, bool> f = _assertionRepos [typeName][name] as Func<T, bool>;
    return f;
}

```

Figure 96 - continued

CHAPTER 6

COMPOSING BEHAVIOR TREE ALGORITHMS

The first step in developing an extensive framework for wide use among robotic systems is demonstrating the capability to perform the basic, fundamental needs of common algorithms such as searching and waypoint navigation. Therefore, development began by implementing a breadth first search on the Behavior Tree World mesh. For more information on the algorithms described below, see section 1.3. Once correct pseudo code for an algorithm has been developed, a behavior tree solution will naturally present itself. Therefore, for each algorithm we will begin with the proper pseudo code and then provide an acceptable behavior tree that implements it.

6.1 Breadth First Search

The breadth first search is an exhaustive approach that, given a starting vertex and goal vertex on a graph, searches simultaneously in every direction for a path to that goal vertex. The benefit to this approach is that it is guaranteed to find the shortest path. The downside is that many cells with very little possibility of being in the final path are searched. The pseudo code for a breadth first search is found in Figure 97. Cell references refer to implementations of ICell mentioned earlier.

The ‘frontier’ is a queue of cells (`frontier = new Queue<ICell>()`).

```
// Given a starting cell 'start'
// and a goal cell 'goal'
1: Enqueue start in frontier
2: Mark start as visited
3: If goal has been reached, return
4: While frontier is not empty
5:   Get next cell (current)
6:   Get current cell's neighbors (neighbors)
7:   For each cell in neighbors
8:     If neighbor is not visited
9:       Put neighbor in frontier
10:      Mark neighbor came-from to current
11:      Mark neighbor as visited
```

Figure 97 - Pseudo code for a breadth first search algorithm

6.1.1 Decomposing the Pseudo Code

A simple analysis of the pseudo code above will reveal the functionality necessary in a complete behavior tree.

Table 12 - A line by line analysis of the Breadth First Search pseudo code revealing necessary behavior tree nodes

Line	Statement	Node Required
1	Enqueue start in frontier	Action
2	Mark start as visited	Action
3	If goal has been reached, return	Assertion
4	While frontier is not empty	Decorator, Assertion
5	Get next cell (current)	Action
6	Get current cell's neighbors (neighbors)	Action
7	For each cell in neighbors	Composite
8	If neighbor is not visited	Assertion
9	Put neighbor in frontier	Action
10	Mark neighbor came-from to current	Action
11	Mark neighbor as visited	Action

What may not be obvious is that on line 5, the algorithm begins operating on discrete segments of the graph which should be parsed into a state object. This means that we will need a **GStateComposer** and all of its children must also be generic nodes. For this example, I have developed a class called **BTLoopArgs** containing all of the necessary state information for each segment of the search (see Figure 98).

```
public class BTLoopArgs
{
    public ICell Current { get; set; }
    public List<ICell> Neighbors { get; set; }
    public int Counter { get; set; }
    public ICell CurrentNeighbor { get; set; }
}
```

Figure 98 - The contents of BTLoopArgs which contains the necessary values for our breadth first search

The state composing Func required of the **GStateComposer**'s constructor will be authored like the one in Figure 99.

```

Func<BTLoopArgs> composer = new Func<BTLoopArgs>(() =>
{
    BTLoopArgs la = new BTLoopArgs();
    la.Current = _frontier.ElementAt(0).Value;
    la.Neighbors = GetNeighbors(la.Current);
    la.Counter = 0;
    return la;
});
//etc
GStateComposer <BTLoopArgs> gsc = new GStateComposer <BTLoopArgs>(composer);

```

Figure 99 - The GStateComposer's argument that provides the generic section of the behavior tree its operable state

The **WhileTrueDecorator** needed to achieve the functionality of line 4 of Table 12 will contain a sequence that activates the assertions on lines 3 and 4. If these assertions fail, the while loop fails and the **WhileTrueDecorator** will return. This indicates that the Breadth First Search has either (a) succeeded or (b) failed due to running out of nodes to search before the goal was found.

The for-loop required to achieve the function of line 7 will require the cooperation of several nodes. To determine what they are, let's examine the operation of a typical for loop:

```
for (int i = 0; i < someValue; i++) { /* loop body */ }
```

Composing a for-loop requires 3 things (some may be omitted), an initialization (int i = 0), a pre-operation check (i < someValue), and a post-operation (i++). The initialization can be handled in the **GStateComposer**'s state composer function as shown above by setting la.Counter=0. The pre-operation check can be handled by an assertion that makes sure that the current counter is less than the total number of neighbors. The post-operation may be handled by an action that increments la.Counter. Therefore we have identified 5 nodes required to simulate our for loop:

- 1) WhileTrueDecorator (WT)
- 2) Initialization (handled by state composer)
- 3) Sequence (ForSeq)
- 4) Assertion: Pre-operation

5) Action: Post-operation

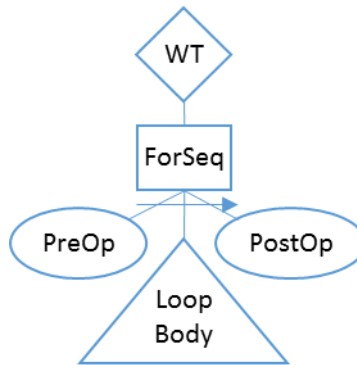


Figure 100 - A behavior tree representation of a for loop

The triangle labeled ‘Loop Body’ is a placeholder for the remaining behavior tree nodes that perform the necessary function of a loop body.

6.1.2 Composing the Breadth First Search Behavior Tree

In section 6.1.1 we decomposed the breadth first search into the necessary behavior tree nodes. Now we carefully put them together in a flow chart layout for easy conversion into code.

Notice in Figure 101 that there are two previously unmentioned nodes on the chart: Succ1 and Succ2. These nodes simply return `NodeStatus.Success` no matter what. This tricks the parent node into continuing when a non-fatal failure has occurred. For example, a `NodeStatus` of failure is expected to occur in the `PreCheck` node when the loop counter exceeds the number of neighbors. This is necessary to short circuit the remaining nodes and relinquish control to the parent. When this happens, `Selector1 (Sel1)` will return the `NodeStatus.Success` from `Succ1` instead, allowing the previous `WhileTrueDecorator (WT1)` to continue searching the frontier. This behavior tree presumes that before ticking the root node, the starting cell has been put into the frontier queue

and that it has already been marked as visited. Similarly, in Sequence2, a NodeStatus of failure is expected when the node has already been visited. When this happens, Selector2 will return the success value from Succ2 so that searching may proceed.

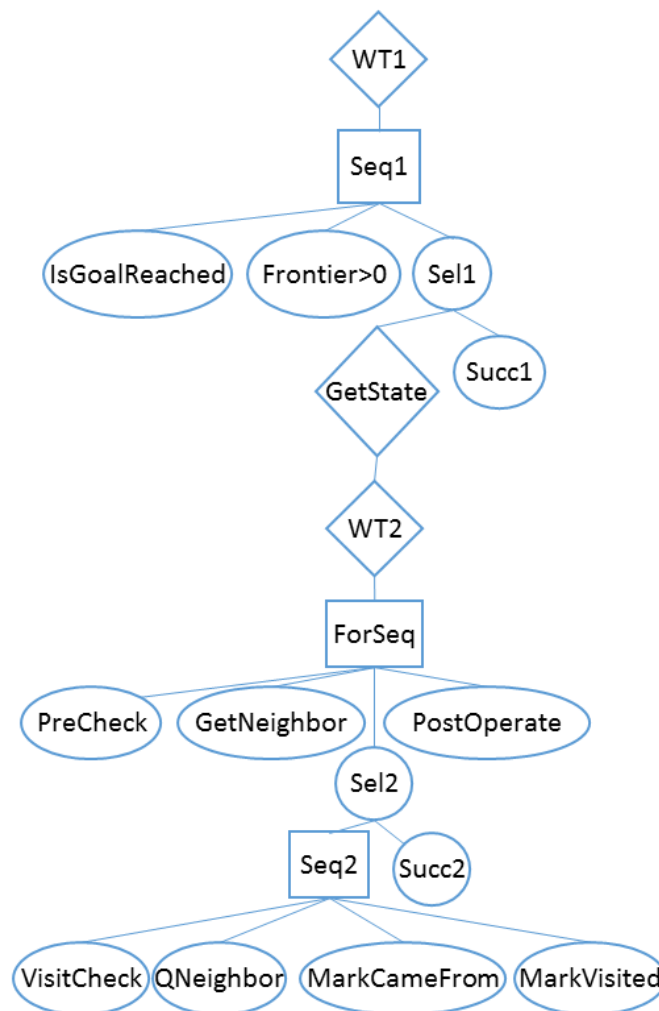


Figure 101 - A behavior tree representation of the Breadth First Search

6.2 Dijkstra

Dijkstra's Algorithm performs very similar to the Breadth First Search except that it acknowledges potential travel costs along the way. While the Breadth First Search will find the shortest Manhattan distance to a goal, it may very well take a traveler directly over the peak of a mountain.

Dijkstra, on the other hand, will lead the traveler along the easiest path which may be around the base of said mountain. One other slight change is that now the frontier is a prioritized list of cells (SortedList<float, ICell>). This enables the algorithm to search along a more optimal path than the breadth first search by prioritizing which cells to check next.

```
// Given a starting cell 'start'
// and a goal cell 'goal'
1: Enqueue start in frontier with cost 0
2: Mark start as visited
3: If goal has been reached, return
4: While frontier is not empty
5:   Get next cell (current)
6:   Get current cell's neighbors (neighbors)
7:   For each cell in neighbors
8:     Calculate travel cost
9:     If neighbor is not visited or cost < currentCost
10:      Apply cost
11:      Put neighbor in frontier sorted by cost
12:      Mark neighbor came-from to current
13:      Mark neighbor as visited
```

Figure 102 - Pseudo code for Dijkstra's algorithm

6.2.1 Decomposing the Pseudo Code

Once again, a line-by-line analysis of the pseudo code above will reveal the functionality necessary in a complete behavior tree.

Table 13 - A line by line analysis of Dijkstra's algorithm pseudo code revealing necessary behavior tree nodes

Line	Statement	Node Required
1	Enqueue start in frontier	Action
2	Mark start as visited	Action
3	If goal has been reached, return	Assertion
4	While frontier is not empty	Decorator, Assertion
5	Get next cell (current)	Action
6	Get current cell's neighbors (neighbors)	Action
7	For each cell in neighbors	Composite
8	Calculate travel cost	Action
9	If neighbor is not visited or cost < currentCost	Selector
9a	Check visited	Assertion
9b	Check cost	Assertion
10	Apply cost	Action
11	Put neighbor in frontier	Action
12	Mark neighbor came-from to current	Action
13	Mark neighbor as visited	Action

These requirements vary only slightly from our previously formed breadth first search behavior tree. We need only one extra composite, two action nodes, and an assertion node. The composite will be Selector3, which will return success if either (a) the node has not been visited or (b) the prioritized list does not already contain a node with the currently calculated cost of travel.

To accommodate the additional functionality of Dijkstra's algorithm, we need to modify our **BTloopArgs** just slightly. There is now a variable to contain the currently calculated cost of travelling to this node (`public double NewCost`). The state composing Func required of the **GStateComposer**'s constructor will also be modified to set the current cost to 0.

6.2.2 Composing Dijkstra's Behavior Tree

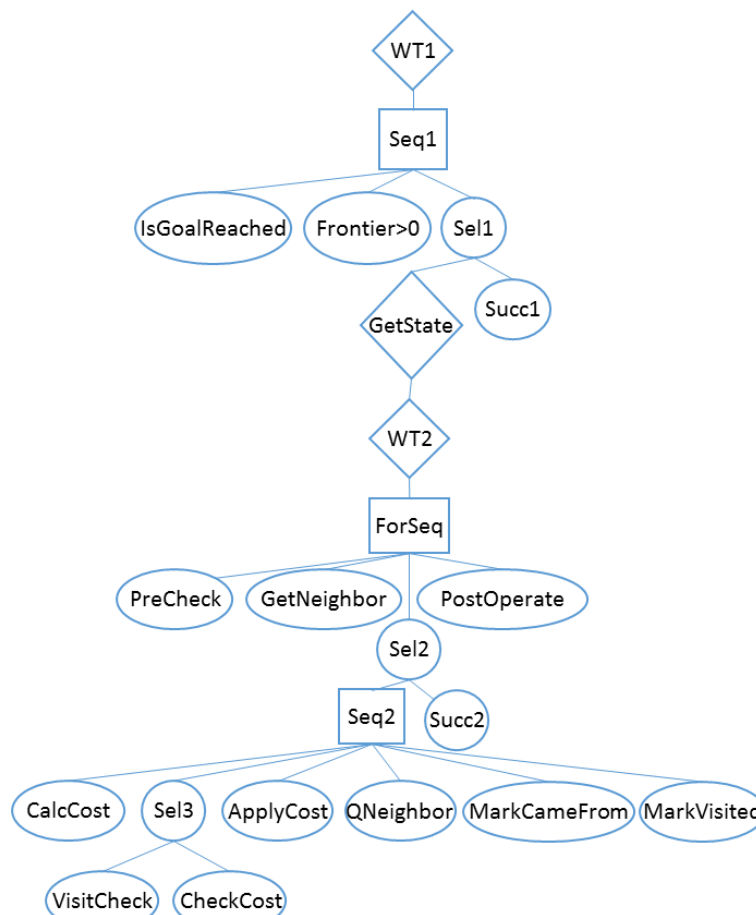


Figure 103 - A behavior tree representation of Dijkstra's algorithm

The composition of Dijkstra's graphical tree follows the same logic used when composing the breadth first search diagram. This feature stands out as important. Once the nodes are identified, a simple intuitive analysis leads to the obvious solution.

6.3 A*

The A* (A-star) algorithm is a popular path finding algorithm that works like Dijkstra's algorithm but with an added heuristic to calculate which cells should have priority in the search. There are any number of heuristics that may be used but a very popular approach is to use the currently active node's distance to the end goal. In this way, the search spreads out unevenly in favor of nodes closest to the goal. Ideally, this will lead the algorithm on a straight line path toward the goal. If obstacles or otherwise blocked nodes come into the path, it will begin searching for ways around that obstacle. In the end it may not provide the absolute shortest path but the number of searched nodes will be much smaller.

```
// Given a starting cell 'start'
// and a goal cell 'goal'
1: Enqueue start in frontier with cost 0
2: Mark start as visited
3: If goal has been reached, return
4: While frontier is not empty
5:   Get next cell (current)
6:   Get current cell's neighbors (neighbors)
7:   For each cell in neighbors
8:     Calculate travel cost
9:     If neighbor is not visited or cost < currentCost
10:      Apply cost
11:      Calculate priority
11:      Put neighbor in frontier sorted by priority
12:      Mark neighbor came-from to current
13:      Mark neighbor as visited
```

Figure 104 - Pseudo code for our implementation of the A* algorithm

6.3.1 Decomposing the Pseudo Code

Our final pseudo code analysis will reveal the necessary nodes required for the A* algorithm's operation. As you can see, we only need to make slight modifications to enable the A* algorithm's functionality. There is one new ActionNode required to calculate the node's priority and the

enqueue neighbor node will now use that priority value instead of the calculated cost value previously used in Dijkstra's algorithm.

The **BTLoopArgs** class must now be fitted with a new variable to contain the search priority of this node. The state composing Func required of the **GStateComposer**'s constructor will be modified to set the first node's priority to 0 as shown in Figure 105.

Table 14 - A line by line analysis of the A* algorithm pseudo code revealing necessary behavior tree nodes

Line	Statement	Node Required
1	Enqueue start in frontier	Action
2	Mark start as visited	Action
3	If goal has been reached, return	Assertion
4	While frontier is not empty	Decorator, Assertion
5	Get next cell (current)	Action
6	Get current cell's neighbors (neighbors)	Action
7	For each cell in neighbors	Composite
8	Calculate travel cost	Action
9	If neighbor is not visited or cost < currentCost	Selector
9a	Check visited	Assertion
9b	Check cost	Assertion
10	Apply cost	Action
11	Calculate priority	Action
12	Put neighbor in frontier sorted by priority	Action
13	Mark neighbor came-from to current	Action
13	Mark neighbor as visited	Action

```
public class BTLoopArgs
{
    public ICell Current { get; set; }
    public List<ICell> Neighbors { get; set; }
    public int Counter { get; set; }
    public ICell CurrentNeighbor { get; set; }
    public double NewCost { get; set; }
    public double Priority { get; set; }
}
```

Figure 105 - The contents of BTLoopArgs which contains the necessary values for implementation of the A* algorithm

```
Func<BTLoopArgs> composer = new Func<BTLoopArgs>(() =>
{
    BTLoopArgs la = new BTLoopArgs();
    la.Current = _frontier.ElementAt(0).Value;
    la.Neighbors = GetNeighbors(la.Current);
    la.Counter = 0;
    la.NewCost = 0;
    la.Priority = 0;
    return la;
});
//etc
GStateComposer <BTLoopArgs> gsc = new GStateComposer <BTLoopArgs>(composer);
```

Figure 106 - The GStateComposer argument that provides the generic section of the behavior tree its operable state

6.3.2 Composing the A* Behavior Tree

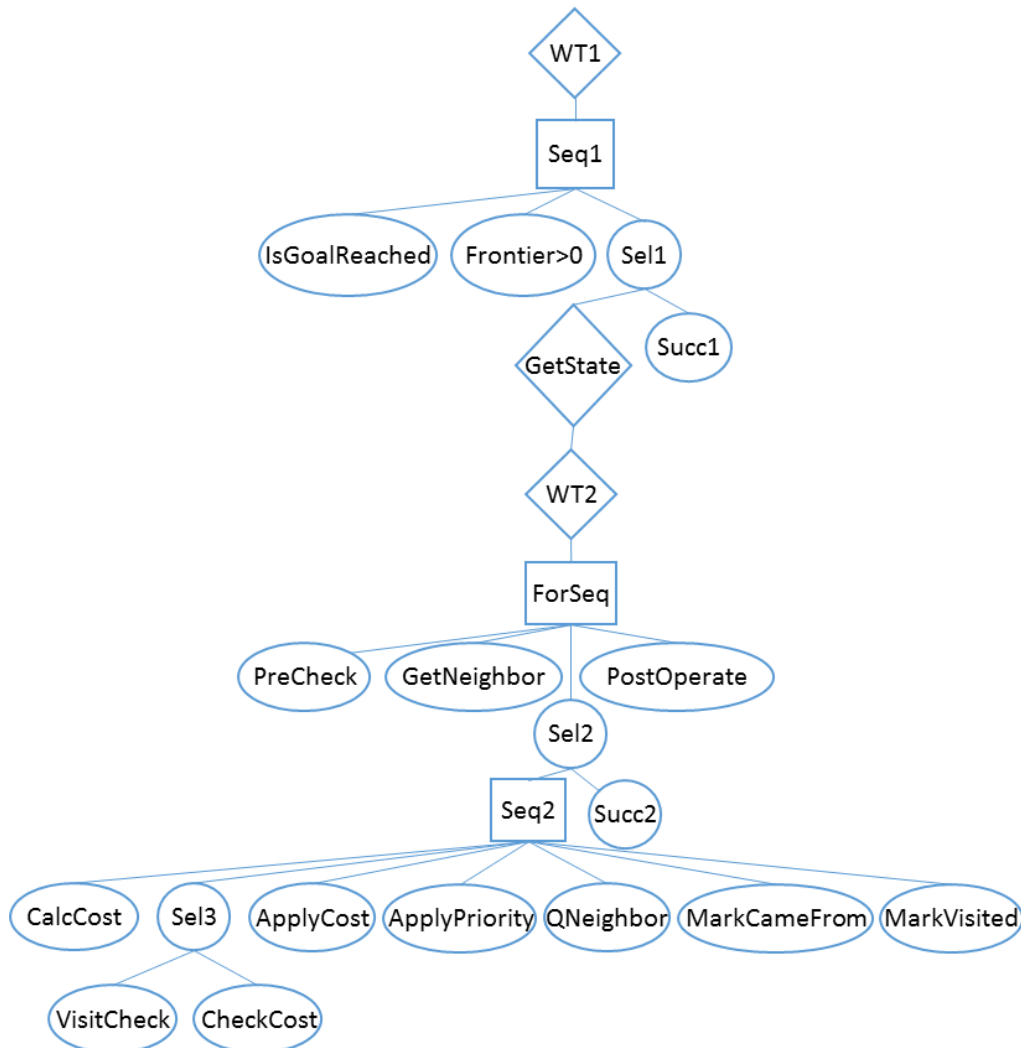


Figure 107 - A behavior tree representation of the A* algorithm

6.4 Reflective Behavior Tree Instantiation

Through the use of reflection (section 1.10), we can create, modify, and load these same behavior trees without making any modifications to the underlying code. An XML based behavior tree loader has been developed that will scan any well-formed XML document that contain behavior tree nodes, instantiate them, and craft the proper parent / child hierarchy.

6.4.1 Reflective Loader

The reflective loader scans a well-formed XML document and instantiates a behavior tree based on its contents. There are three parts to this file: (1) Necessary assemblies, (2) necessary types, and (3) the behavior tree itself. The way it operates is very simple. It first loads all of the specified Assemblies into a collection that is made available to later stages. It then loads all of the specified types by retrieving them from the preloaded assemblies. Finally, it instantiates each of these types in the order specified by the xml file and assigns child and parent status.

An assembly is a reusable, versionable, and self-describing block of a common language runtime application [44]. These will usually be in the form of dynamically linked libraries (dlls). The first section of our well-formed document should be a list of Assembly elements with a *Name* and *File* attribute for all of the dll files containing types used in the behavior tree. The **XmlBTLoader** will load these files as C# Assembly classes and store them in a dictionary sorted by name for later use (Dictionary<string, Assembly> _assemblies). Whenever an assembly element does not contain a *File* attribute, the program will load the ‘entered’ assembly which is the executable file that was run to get the behavior tree program started in the first place.

```
<Assemblies>
  <!--No File attribute loads entry assembly-->
  <Assembly Name="Entered"/>
  <Assembly Name="BTLib" File="JBTFramework.dll"/>
</Assemblies>
```

Figure 108 - A list of assemblies necessary to load all of the types in our behavior tree

The second section of our XML document should be all of the types used in the behavior tree. These types might include Sequence, Selector, Action, GSequence, GSelector, and others. These should match the actual class names used in the authored code. Each Type element should contain at least an *Assembly*, *Name*, and *Namespace* attribute. The assembly attribute represents the key

value we used to store the assemblies that were loaded in phase one into the dictionary. The name attribute should match the name of the class used the code. The namespace attribute should match the entire surrounding namespace of the type. This will allow the loader to search through the loaded assemblies and gather the metadata for that type. Optional attributes include *IsGeneric* and *ArgType*. When loaded types are generic, they must have an argument type to send to the object creator so that it can dynamically create the proper intermediate language for that type. For example, in C++, when you load a template class like a vector, you must tell the compiler what type of objects the vector will operate on (e.g. vector<int>). Similarly, C# generic types require a type declaration so that the compiler knows what type of information it will be working with (e.g. GActionNode<int>). When a node is generic and requires an *ArgType*, the *ArgType* itself must be individually declared prior to using it as a generic node's argument type. As you can see in Figure 109, The **BTLoopArgs** type is loaded before being used in several of the generic types below it (GStateComposer, GSequence, etc).

```
<Types>
  <Type Assembly="Entered" Name="BTLoopArgs" Namespace="Jeremy.BTWorld.Tools"/>
  <!--Default IsGeneric is false-->
  <Type Assembly="BTLib" Name="WhileTrue" Namespace="Jeremy.BTLib.Nodes.Decorators" />
  <Type Assembly="BTLib" Name="Sequence" Namespace="Jeremy.BTLib.Nodes" />
  <Type Assembly="BTLib" Name="ActionNode" Namespace="Jeremy.BTLib.Nodes" />
  <Type Assembly="BTLib" Name="Selector" Namespace="Jeremy.BTLib.Nodes" />
  <Type Assembly="BTLib" Name="GStateComposer" Namespace="Jeremy.BTLib.Nodes.Generic.Decorators"
    IsGeneric="true" ArgType="BTLoopArgs"/>
  <Type Assembly="BTLib" Name="GSequence" Namespace="Jeremy.BTLib.Nodes.Generic"
    IsGeneric="true" ArgType="BTLoopArgs"/>
  <Type Assembly="BTLib" Name="GActionNode" Namespace="Jeremy.BTLib.Nodes.Generic"
    IsGeneric="true" ArgType="BTLoopArgs"/>
  <Type Assembly="BTLib" Name="GLoopDecorator" Namespace="Jeremy.BTLib.Nodes.Generic.Decorators"
    IsGeneric="true" ArgType="BTLoopArgs"/>
  <Type Assembly="BTLib" Name="GSelector" Namespace="Jeremy.BTLib.Nodes.Generic"
    IsGeneric="true" ArgType="BTLoopArgs"/>
</Types>
```

Figure 109 - A list of Types necessary craft our behavior tree

The final section of our XML document will be the behavior tree nodes themselves, nested the same way as they are in the previously designed figures (Figure 101, Figure 103, and Figure 107).

Having these figures on hand when crafting the XML makes it a very simple task. The XML element names should match one of the previously loaded types. Every node has an optional *Name* attribute. Certain types will have required attributes. **ActionNodes** and **GActionNodes** have a required *Action* attribute that defines the name of the Func that they should retrieve from the action repository.

6.4.2 Breadth First Search

The following XML snippet represents the same behavior tree shown in Figure 101. Once instantiated by the reflective loader, it will perform identical to the breadth first search implemented strictly in code. Ironically, the behavior tree operation uses a depth first approach.

```
<Tree>
  <WhileTrue Name="WT1">
    <Sequence Name="Seq1">
      <ActionNode Action="CheckGoalReachedBFSX"/>
      <ActionNode Name="CheckFrontier" Action="Frontier>>0BFSX" />
      <Selector Name="Sel1">
        <GStateComposer Name="GetState" Getter="BTLoopArgsBFSX">
          <GWhileTrue Name="WT2">
            <GSequence Name="ForSeq">
              <GActionNode Action="PreCheck"/>
              <GActionNode Action="UpdateCurrentNeighbor"/>
              <GSelector Name="Sel2">
                <GSequence Name="Seq2">
                  <GActionNode Action="NeighborVisitCheck"/>
                  <GActionNode Action="CanCross"/>
                  <GActionNode Action="EnqueueCurrentNeighborBFSX"/>
                  <GActionNode Action="MarkNeighborVisited"/>
                  <GActionNode Action="MarkCameFrom"/>
                </GSequence>
                <GActionNode Name="Succ2" Action="NodeStatus.Success"/>
              </GSelector>
              <GActionNode Name="PostCheck" Action="IncrementLoopCounter"/>
            </GSequence>
          </GWhileTrue>
        </GStateComposer>
        <ActionNode Name="Succ1" Action="NodeStatus.Success"/>
      </Selector>
    </Sequence>
  </WhileTrue>
</Tree>
```

Figure 110 - The XML representation of a Breadth First Search behavior Tree. This XML code is used by the reflective loader to instantiate the tree.

6.4.3 Dijkstra

The XML snippet in Figure 111 represents the same behavior tree shown in Figure 103. Again, once the reflective loader parses the file and instantiates each of the behavior tree nodes, it will perform identically to its hardcoded counterpart.

```
<Tree>
  <WhileTrue Name="WT1">
    <Sequence Name="Seq1">
      <ActionNode Name="IsGoalReached" Action="CheckGoalReached"/>
      <ActionNode Name="CheckFrontier" Action="Frontier>>0" />
      <Selector Name="Sel1">
        <GStateComposer Name="GetState" Getter="BTLoopArgs">
          <GLoopDecorator Name="LoopDecorator" Init="Init" PreCheck="PreCheck"
            PostOperations="IncrementLoopCounter">
            <GSelector Name="Sel2">
              <GSequence Name="Seq2">
                <GActionNode Action="UpdateCurrentNeighbor"/>
                <GActionNode Action="CanCross"/>
                <GActionNode Action="CalculateCost"/>
                <GSelector Name="Sel3">
                  <GActionNode Action="NeighborVisitCheck"/>
                  <GActionNode Action="CheckCost"/>
                </GSelector>
                <GActionNode Action="MarkNeighborVisited"/>
                <GActionNode Action="MarkCameFrom"/>
                <GActionNode Action="ApplyCost"/>
                <GActionNode Action="EnqueueCurrentNeighborByCost"/>
              </GSequence>
              <GActionNode Action="NodeStatus.Success"/>
            </GSelector>
          </GLoopDecorator>
        </GStateComposer>
        <ActionNode Name="Succ1" Action="NodeStatus.Success"/>
      </Selector>
    </Sequence>
  </WhileTrue>
</Tree>
```

Figure 111 - The XML representation of Dijkstra's algorithm behavior Tree. This XML code is used by the reflective loader to instantiate the tree.

6.4.4 A*

The following XML snippet represents the same behavior tree shown in Figure 107. Once again, and finally, when the reflective loader instantiates each of the behavior tree nodes, its behavior will be identical to its hardcoded counterpart.

```

<Tree>
  <WhileTrue Name="WT1">
    <Sequence Name="Seq1">
      <ActionNode Name="IsGoalReached" Action="CheckGoalReached"/>
      <ActionNode Name="CheckFrontier" Action="Frontier&gt;0" />
      <Selector Name="Sel1">
        <GStateComposer Name="GetState" Getter="BTLoopArgs">
          <GSequence Name="ForSeq">
            <GActionNode Action="CurrentVisitCheck"/>
            <GActionNode Action="MarkCurrentVisited"/>
            <GLoopDecorator Name="LoopDecorator" Init="Init" PreCheck="PreCheck"
              PostOperations="IncrementLoopCounter">
              <GSelector Name="Sel2">
                <GSequence Name="Seq2">
                  <GActionNode Action="UpdateCurrentNeighbor"/>
                  <GActionNode Action="NeighborVisitCheck"/>
                  <GActionNode Action="CanCross"/>
                  <GActionNode Action="CalculateCost"/>
                  <GActionNode Action="ApplyZDifference"/>
                  <GActionNode Action="CheckCost"/>
                  <GActionNode Action="CalculatePriority"/>
                  <GActionNode Action="EnqueueCurrentNeighborByPriority"/>
                  <GActionNode Action="ApplyCost"/>
                  <GActionNode Action="MarkCameFrom"/>
                  <GActionNode Action="MarkNeighborVertex"/>
                </GSequence>
                <GActionNode Action="NodeStatus.Success"/>
              </GSelector>
            </GLoopDecorator>
          </GSequence>
        </GStateComposer>
        <ActionNode Name="Succ1" Action="NodeStatus.Success"/>
      </Selector>
    </Sequence>
  </WhileTrue>
</Tree>

```

Figure 112 - The XML representation of the A* behavior Tree. This XML code is used by the reflective loader to instantiate the tree.

CHAPTER 7

BEHAVIOR TREE APPLICATION RESULTS

The following observable behavior patterns are seen as necessary components to fulfill the objectives of this research project. Each behavioral pattern's usefulness will be examined through a simple scenario.

7.1 Path Planning

Autonomous vehicles are frequently used to explore places where humans would either be in danger or would otherwise be impractical to send. Examples of danger might be radioactive sites or potential bomb locations. An impractical scenario for a human would be to map several square kilometers of underwater terrain. Therefore, most situations dictate some sort of autonomous path planning and following capability.

The current version of the Breadth First Search, Dijkstra, and A* algorithms perform their intended functions on simple connected graphs as they are currently implemented. What they lack though are specialty functions mean to account for **BTWorld** map conditions (water or fog of war) and feedback to human observers. This gives us the opportunity to demonstrate the ease of extension that behavior trees offer. Here we will add the functionality to the Breadth First Search behavior tree to check the map for water and also to mark nodes as we search them.

Table 15 - Additional nodes needed to enable custom functionality on BTWorld

Node	Function	Name
Assertion	Check whether travel to neighbor node is possible	CanCross
Action	Mark current cell that is being considered for shortest path	MarkCurrent
Action	Mark neighbor nodes as they are added to frontier	MarkNeighbor

The first additional step we are going to take in our Behavior Tree is to check if travel to a particular node is possible. When the fog-of-war option is turned on, any nodes underneath the fog will be presumed traversable. Also, the cost to travel from node to node will be calculated as if it was flat land. These calculations will be adjusted once the land is uncovered. We will place this functionality just after the *visitCheck* node. Notice in Figure 113 that if the node is hidden, we automatically return success. Otherwise, an *IsPassable* check is made inside the cell which should account for conditions such as water or steep inclines.

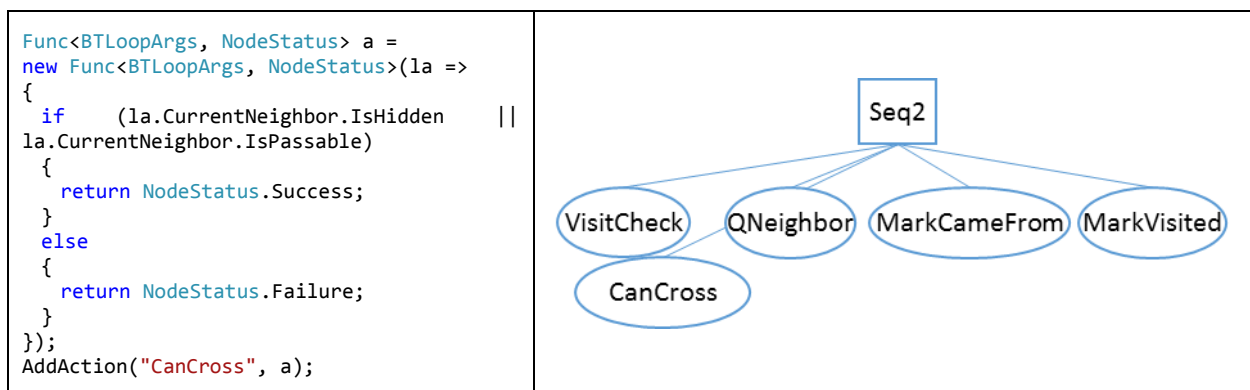


Figure 113 - The CanCross functionality needed for the Breadth First Search to check nodes for non-traversable conditions

The MarkCurrent node is simply going to add a color to the map for each node that we are currently considering for shortest path membership. There are at least two ways to go about handling this functionality. We could either (a) add a sequence underneath the GetState node with the MarkCurrent action as the first child and the already created WhileTrue (WT2) node as the second or (b) we can just add this node coloring function to the state composer that is sent to the GetState node. I have opted for the simpler approach and added this function to the GetState composer (shown in Figure 114)

```

Func<BTLoopArgs> composer = new Func<BTLoopArgs>(() =>
{
    BTLoopArgs la = new BTLoopArgs();
    la.Current = _frontier.Dequeue();
    la.Current.Color = Color.Yellow;
    la.Neighbors = GetNeighbors(la.Current);
    return la;
});
GStateComposer <BTLoopArgs> gsc = new GStateComposer <BTLoopArgs>(composer);

```

Figure 114 - A modified GStateComposer argument that sets the Breadth First Search current node to yellow

The MarkNeighbor node is going to change the color of any nodes that we add to the frontier. Ideally this will be different color than the one used for the currently considered node. This node can be added to Sequence2 just after the QNeighbor node. The code for the node's action and the resulting sequence is shown in Figure 115.

```

Func<BTLoopArgs, NodeStatus> a =
new Func<BTLoopArgs, NodeStatus>(la =>
{
    la.CurrentNeighbor.Color = Color.Purple;
    return NodeStatus.Success;
});
GActionNode<BTLoopArgs> an =
new GActionNode<BTLoopArgs>(a);

```

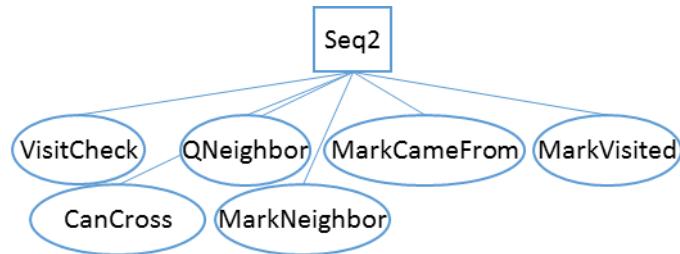


Figure 115 - The MarkNeighbor functionality needed for the Breadth First Search to give color coded feedback to users

With similar intuition, the Dijkstra and A* algorithms can be modified to accommodate this additional functionality. In Figure 117 the nodes were staggered just to conserve horizontal space.

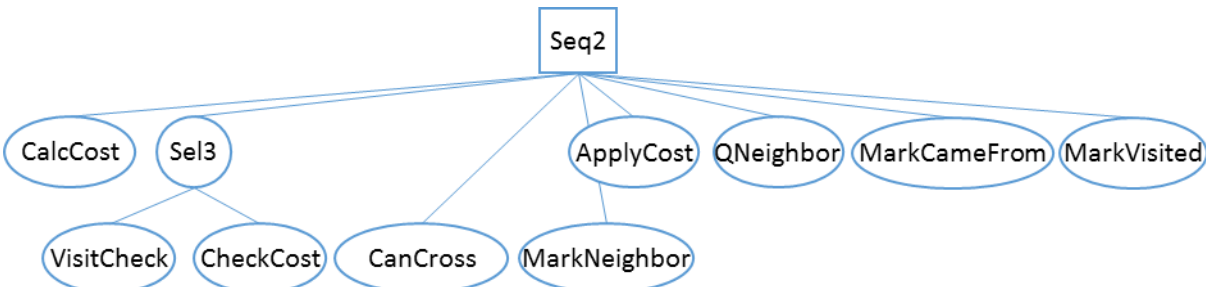


Figure 116 - The modified sequence necessary to add the CanCross and MarkNeighbor functionality to Dijkstra's algorithm

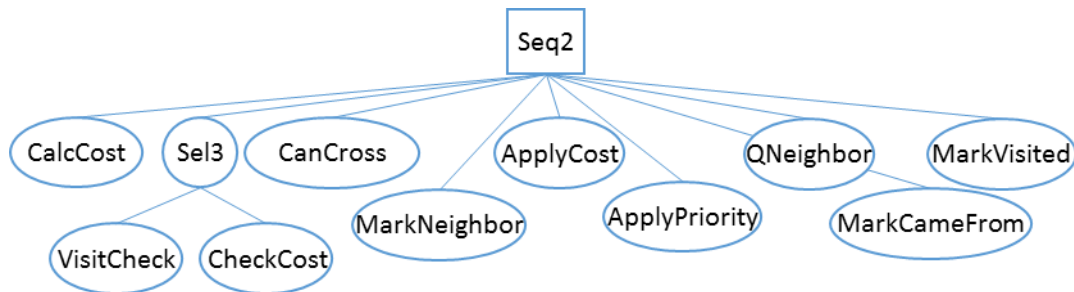


Figure 117 - The modified sequence necessary to add the CanCross and MarkNeighbor functionality to the A* algorithm

With our newly modified behavior trees we are able to demonstrate the path planning behavior trees in our **BTWorld**. In Figure 118 you'll see two different paths resulting from the same A* algorithm implementation. The one on the left represents the shortest path without accounting for water or terrain difficulties. You'll notice that it returns a path directly over the top of a steep mountain. The one on the right accounts for the steep terrain as well as water obstacles and takes the vehicle on a more optimal path around the mountain. Similarly, in Figure 119, the A* algorithm provides a path directly across a lake that is unknown to the vehicle. Once explored, though, it plans around it properly.

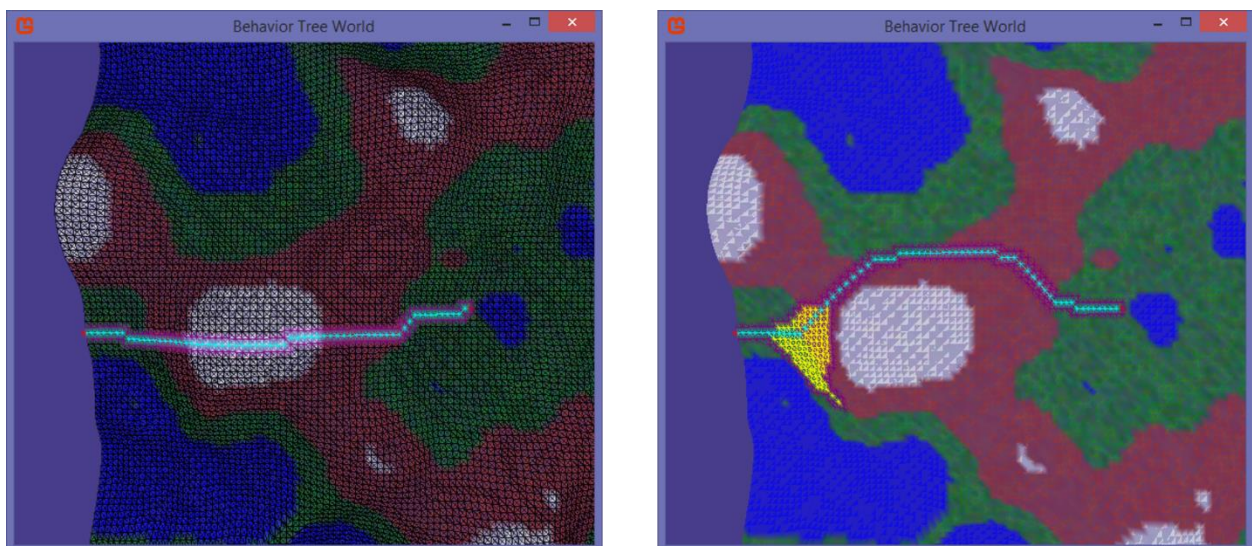


Figure 118 - The A* algorithm performing a search with fog of war enabled (left) and disabled (right)

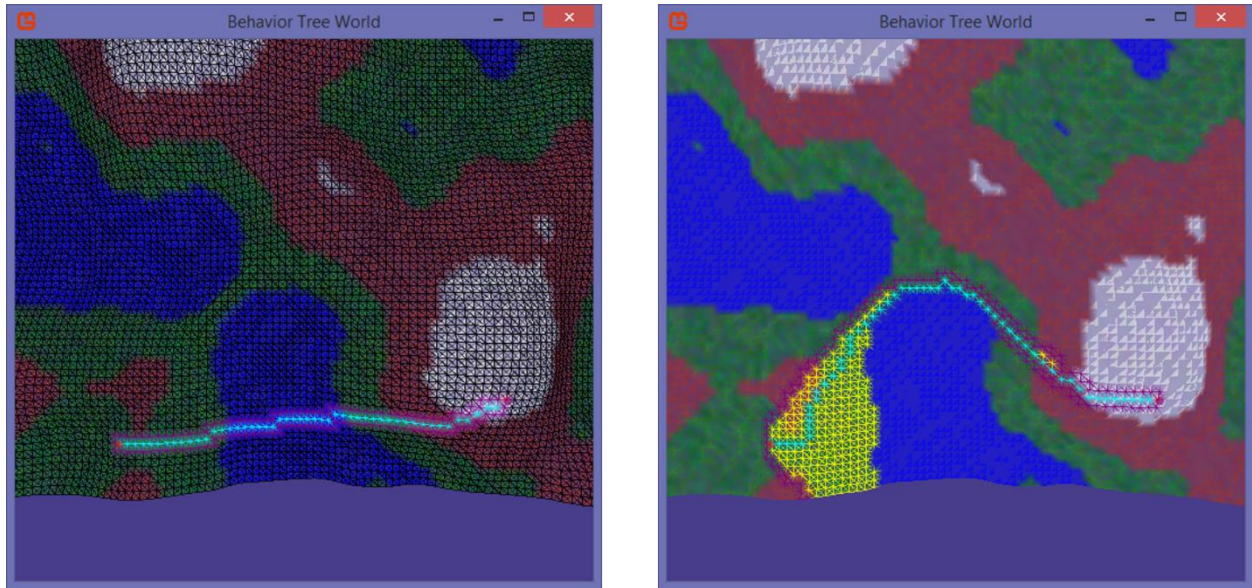


Figure 119 - The A* path planning algorithm providing a path through unexplored water (left). Once explored it plans around it properly (right)

7.2 Acoustic Node Mapping

Underwater vehicles are frequently used to communicate with underwater acoustic nodes. To simulate the mission of searching and finding nodes, communication components have been developed to interact with each other inside of our **BTWorld** (see section 4.2, Acoustic Node Simulator (Communicator)). These communicators simulate underwater acoustic communications by (a) only registering messages with other communicators when they are in range and (b) responding to messages after a timed delay based on underwater acoustic communication speeds. The determination of locations based on received messages is built into a component called **BTLocator**. The **BTLocator** was constructed as a state machine with each state being a single action (see Figure 120). When an instance of a **BTLocator** is created it must be provided with a **Traveler** (section 4.1) from which it gains access to the communicator. The **BTLocator** furthermore controls the current navigational goal of the traveler as well as when the communicator should ping another acoustic node.

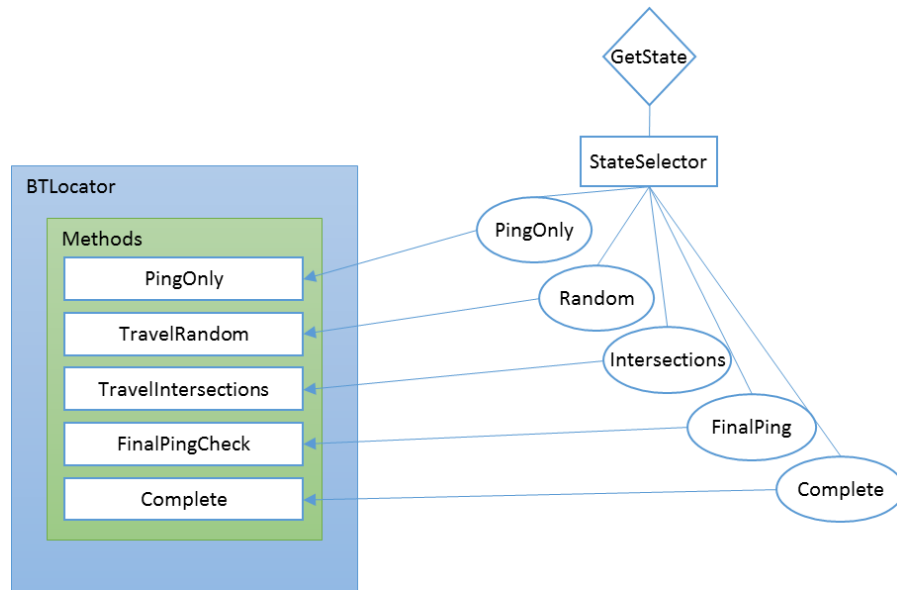


Figure 120 - The makeup of the BTLocator component used to localize nodes

When the user first initiates the locator object, it creates a mow the lawn pattern around the map. As the vehicle travels these predetermined locations it is simultaneously pinging the other acoustic node nearby. Whenever it receives a reply it stores the location from which it pinged as well as the distance measurement received by the other node. In this way, it can build up a series of circles from which to triangulate the position of the other node (see Figure 121 and Table 16). Once communication to another node has been established, the locator breaks out of the mow the lawn pattern and begins travelling randomly inside the bounds of this newly created circle. As time passes, these restrictive bounds are removed to allow greater flexibility for travel. This is to overcome the possibility of an errant message or a subsequent failure to reestablish communications. Otherwise, the vehicle might be forever trapped in a tight travel pattern with no hope of getting another message. When a second communication is received, the vehicle calculates the new circle and finds any intersections between the two. If they are found, it goes into the travel intersections state to determine which one is correct. When a ping is received within a predetermined radius the vehicle registers the node as found and moves into the complete state.

An example of this behavior in **BTWorld** is shown in Figure 122. The red circles are constructed using the center point and radius at which the communication was received from the other node.

The equations necessary to solve for the components of Figure 121 are shown below, in Table 16.

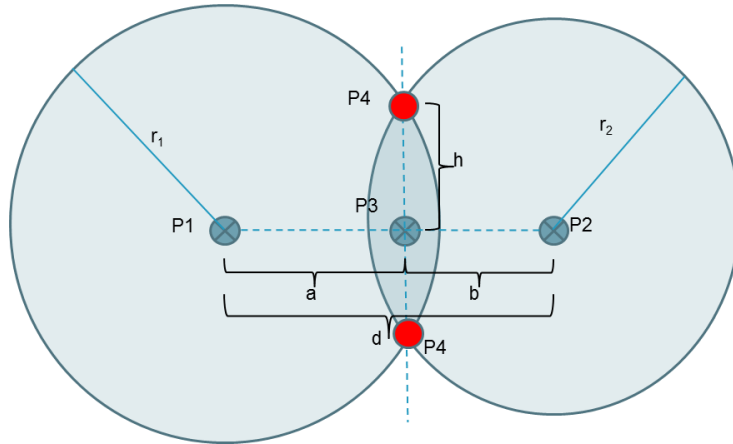


Figure 121 – A graphical depiction of solving for the intersection of two circles.

Table 16 –The descriptions of equations needed to solve for the intersection of two circles.

Equation	Description
$\Delta x = x_2 - x_1$	The difference between the x components of the vehicle's first and second range request.
$\Delta y = y_2 - y_1$	The difference between the y components of the vehicle's first and second range request.
$d = \sqrt{\Delta x^2 + \Delta y^2}$	The straight line distance between the two vehicle locations
$a = \frac{r_1^2 - r_2^2 + d^2}{2d}$	The distance between the first ranging point and the halfway point of the two ranging points.
$h = \sqrt{r_1^2 - a^2}$	The perpendicular offset of the straight line between the two ranging points and the two possible locations of the ranged node.
$P_3 = \frac{a}{d}(\Delta x, \Delta y)$	The point directly halfway between the two ranging locations
$x_4 = x_3 \pm \frac{h}{d}(y_2 - y_1)$	The two possible x coordinates of the ranged node.
$y_4 = y_3 \mp \frac{h}{d}(x_2 - x_1)$	The two possible y coordinates of the ranged node.

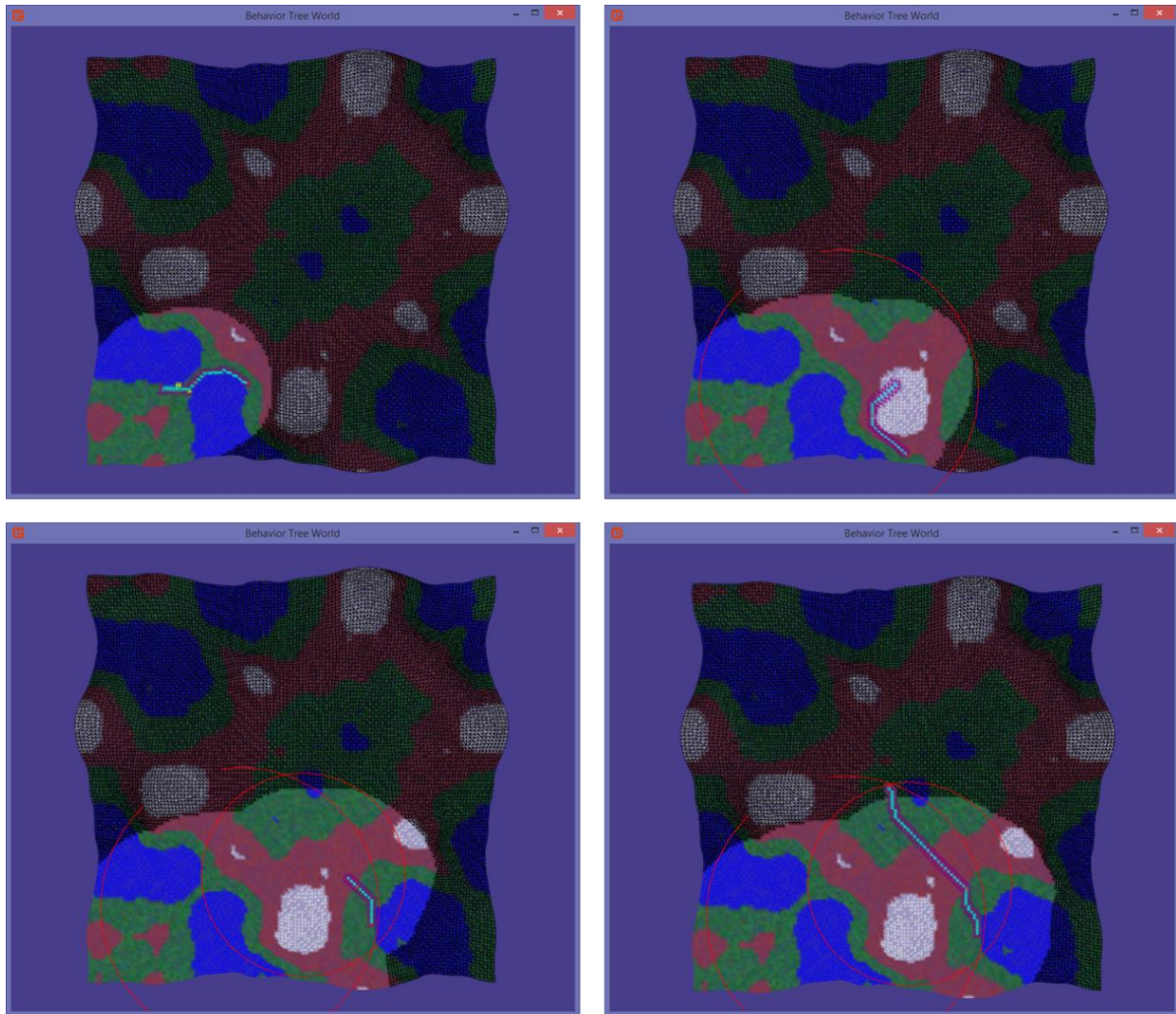


Figure 122 – When observed from top left to bottom right (1) The vehicle begins a mow the lawn pattern around the map (2) The vehicle gets a message from the static node and builds a circle using the center point of the received communication and the distance received. The vehicle begins randomly traveling inside of this circle (3) The vehicle receives a second communication from the static node. It will now calculate intersections and check for the static node at those locations. It initially searches the wrong intersection (4) The vehicle checks the second intersection location where it will find the static node and move into the ‘complete’ state.

7.3 REMUS Simulator Integration

The previous behavior tree applications represent fundamental autonomous vehicle capabilities that must exist before more sophisticated algorithms can be developed. I have taken this one step further to demonstrate the feasibility on actual underwater vehicles. Below you will find a

walkthrough of the steps taken to implement a simple waypoint following behavior tree algorithm on the REMUS simulator.

7.3.1 Step 1: Decompose the Vehicle Control Library

As in the previous algorithms, the first step is to decompose the available methods inside the vehicle control library into the necessary actions and assertions that will achieve the desired output. A cursory examination of the behavior requirements revealed what these necessary actions (!) and assertions (?) were.

The vehicle must travel submerged. A partially submerged propeller contending with constant wave energy is very inefficient. Therefore we need a set depth action. Since this action will be a one-time adjustment, it will be sent to the vehicle before the behavior tree begins its traversal.

- Set depth!

The vehicle's current location must be established to determine bearings to waypoints. As discussed earlier these vehicle status updates will be submitted into a queue until after they have been examined. We need to check this status each time before issuing a navigation command to the REMUS. The algorithm will decide whether a left or right azimuthal adjustment is more optimal.

- Is there a status message available?
- Check location!
- Determine optimal heading adjustment!
- Modify heading to the left!

- Modify heading to the right!

A determination of what to do next will need to be made once the current waypoint is reached. Currently the only actions are to set the next waypoint as the target or mark the mission as complete.

- Has the current waypoint been reached?
- Have all of the mission tasks been completed?
- Is there another waypoint in the queue?
- Set target to next waypoint!

Table 17 lists the necessary actions and assertions. The ‘Index’ column is supplied so that the leaf nodes of a behavior tree can be labeled with an action index rather than full names. The ‘Behavior Output’ column gives a very simple description of what the action accomplishes. The details of how it is accomplished are hidden within the vehicle library itself. Two previously unmentioned nodes have been added: AutoSuccess and MissionComplete. The auto success node simply returns a NodeStatus.Success value indicating that a particular sequence reached the end without error. The mission complete node sends a signal to the root node that all of the mission tasks have been completed.

Table 17 - The underwater vehicle behaviors available for use in the following example

Index	Behavior Name	Behavior Output
1	IsMessageAvailable	Checks the vehicle communication queue for a new status message
2	GetLocation	Checks the location field of the status message
3	ShouldBearLeft	Determines whether a left turn is optimal
4	ShouldBearRight	Determines whether a right turn is optimal
5	BearLeft	Instructs the vehicle to position the rudder for a left turn
6	BearRight	Instructs the vehicle to position the rudder for a right turn
7	IsGoalReached	Checks whether the vehicle has reached its current waypoint target
8	IsEndOfMission	Determines whether all of the tasks have been accomplished
9	PopWaypoint	Sets the vehicle’s waypoint target to the next waypoint
10	AutoSuccess	Simply returns a NodeStatus.Success value to its parent node
11	MissionComplete	Signals the tree root that the mission is complete

7.3.2 Step 2: Construct the Action and Assertion Code

Once each of the necessary actions and assertions are properly coded to perform the desired task, they must be inserted by name into the appropriate repository for consumption later by behavior tree nodes (see Figure 123). Behavior tree action and assertion nodes will be assigned the appropriate Func to execute at runtime by way of hardcoded tree construction, which is shown below in Figure 125, or through the reflective instantiation of an XML tree based on the reflective loader discussed in section 6.4.1.

```
_actionRepo.RegisterAssertion("IsMessageAvailable", IsMessageAvailable);  
_actionRepo.RegisterAction("GetLocation", GetLocation);  
_actionRepo.RegisterAssertion("ShouldBearLeft", ShouldBearLeft);  
_actionRepo.RegisterAssertion("ShouldBearRight", ShouldBearRight);  
_actionRepo.RegisterAction("BearLeft", ModifyHeading);  
_actionRepo.RegisterAction("BearRight", ModifyHeading);  
_actionRepo.RegisterAssertion("IsGoalReached", IsGoalReached);  
_actionRepo.RegisterAssertion("IsEndOfMission", IsEndOfMission);  
_actionRepo.RegisterAction("PopWaypoint", PopWaypoint);  
_actionRepo.RegisterAction("MissionComplete", MissionComplete);
```

Figure 123 - Entering the actions and assertions into the demo repository

7.3.3 Step 3: Sketch the Behavior Tree

With the actions and assertions intact, the structure of the behavior tree can be drawn. Doing so helps guide the developer's decisions regarding sequences and selectors which are the heart of behavior trees. The root node (R) is just a basic decorator that ticks its only child. The decorator underneath, labeled 'WT' is a specialized 'while-true' node (see section 5.4.1) that executes from start to finish until a NodeStatus.Failure is received. Sequences and selectors have been labeled as seq# and sel# respectively so that they may be easily identified in the code. Each leaf node with a number is a reference to the indices in Table 17. For example, the node with a 3 in it calls "ShouldBearLeft." Since this node falls under a sequence, if it returns NodeStatus.Success then the next node is called. This node (5) calls the "BearLeft" action and returns NodeStatus.Success. Following this action, the sequence (seq2) is satisfied and also returns NodeStatus.Success. This

satisfies the selector (sel1) which means that a check for bearing right is skipped. Control continues to bubble up to the parent sequence (seq1) which then passes control to its next child, another selector (sel2). This grouping of nodes determine overall mission status and target waypoint adjustments.

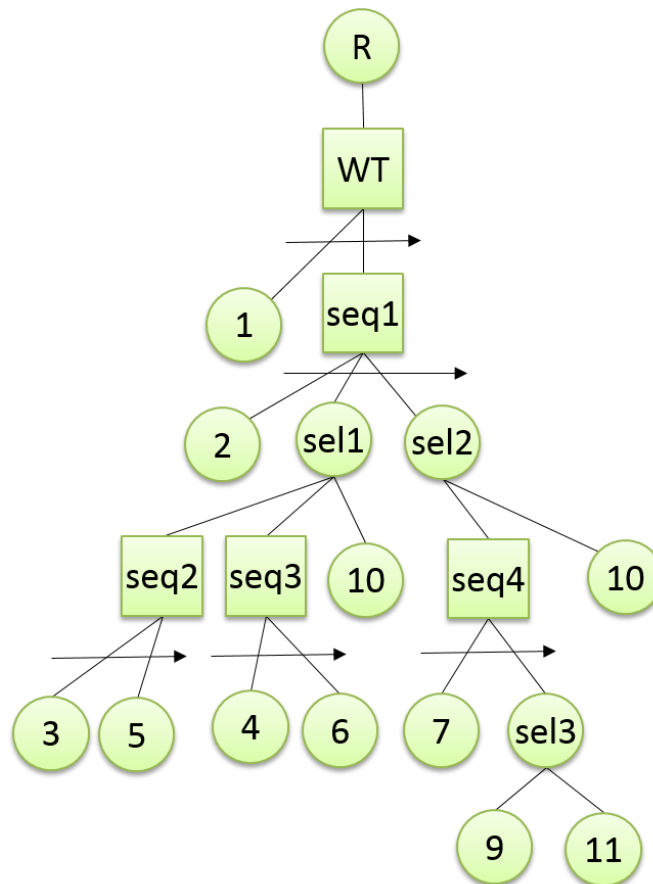


Figure 124 – Behavior tree sketch for the REMUS waypoint following test

7.3.4 Step 4: Building the Tree in Code

Now that all of the necessary actions and assertions have been developed and a preliminary sketch has been drawn we can easily develop the background code to execute the tree. I have opted for a ‘Factory’ approach that crafts the tree and returns the root node to the caller. This factory uses the

action and assertion Funcs that were previously inserted into the repository as arguments for the nodes.

Action and Assertion class constructor signatures:

- `public ActionNode(string name, Func<NodeStatus> action)`
- `public AssertionNode(string name, Func<bool> assertion)`

```
public static INode GetWaypointBT(ActionRepository repo, EventWaitHandle trigger)
{
    // Step 1: create all of the individual nodes
    IDecorate root = new TriggeredRoot("Root", trigger, 150);
    INode isMessageAvailable = new AssertionNode("1", repo.GetAction<Func<bool>>("IsMessageAvailable"));
    INode getLoc = new ActionNode("2", repo.GetAction<Func<NodeStatus>>("GetLocation"));
    INode shouldBearLeft = new AssertionNode("3", repo.GetAction<Func<bool>>("ShouldBearLeft"));
    INode shouldBearRight = new AssertionNode("4", repo.GetAction<Func<bool>>("ShouldBearRight"));
    INode bearLeft = new ActionNode("5", repo.GetAction<Func<NodeStatus>>("BearLeft"));
    INode bearRight = new ActionNode("6", repo.GetAction<Func<NodeStatus>>("BearRight"));
    INode isGoalReached = new AssertionNode("7", repo.GetAction<Func<bool>>("IsGoalReached"));
    INode isEOM = new AssertionNode("8", repo.GetAction<Func<bool>>("IsEndOfMission"));
    INode popWaypoint = new ActionNode("9", repo.GetAction<Func<NodeStatus>>("PopWaypoint"));
    INode successNode = new SuccFailNode("10", NodeStatus.Success);
    INode missionComplete = new ActionNode("11", repo.GetAction<Func<NodeStatus>>("MissionComplete"));
    // Step 2: Create all of the selectors with representative names
    IComposite whileMessage = new WhileTrueSequence("CheckMessageAvailable");
    IComposite seq1 = new Sequence("MainSequence");
    IComposite seq2 = new Sequence("TestTurnLeft");
    IComposite seq3 = new Sequence("TestTurnRight");
    IComposite seq4 = new Sequence("PopWaypointOrExit");
    IComposite sel1 = new Selector("TestTurning");
    IComposite sel2 = new Selector("CheckGoalStatus");
    IComposite sel3 = new Selector("PopOrExit");
    // Populate the root with the while-true node
    root.Child = whileMessage;
    // Populate the while-true node with the message check and sequence 1
    whileMessage.AddChild(isMessageAvailable, seq1);
    // Populate each sequence with the appropriate children
    seq1.AddChild(getLoc, sel1, sel2);
    seq2.AddChild(shouldBearLeft, bearLeft);
    seq3.AddChild(shouldBearRight, bearRight);
    seq4.AddChild(isGoalReached, sel3);
    // Populate all of the selectors with their children
    sel1.AddChild(seq2, seq3, new SuccFailNode("10", NodeStatus.Success));
    sel2.AddChild(seq4, new SuccFailNode("10", NodeStatus.Success));
    sel3.AddChild(popWaypoint, missionComplete);
    // Return the root node - this effectively gives access to the entire tree
    return root;
}
```

Figure 125 - The Behavior Tree Factory returning a waypoint following behavior tree

Notice in the factory's construction that each Action and Assertion node is given a name that aligns with the index in Table 17. As the tree writes its execution history to a file this makes debugging

analysis much easier. For example, in Figure 126 we see one traversal through the behavior tree that indicates a waypoint has been reached and that the target should be updated.

```
WhileTrueSequence[CheckMessageAvailable] updating
  AssertionNode[1] updating
  AssertionNode[1] updated Success
  Sequence[MainSequence] updating
    ActionNode[2] updating
    Heading: 82.5999999046326
    ActionNode[2] updated Success
    Selector[TestTurning] updating
      Sequence[TestTurnLeft] updating
        AssertionNode[3] updating
        AssertionNode[3] updated Success
        ActionNode[5] updating
        Offset is 83.4182046757866
        Delta is -0.818204771154015
        Turning left at -0.73867919909028
        ActionNode[5] updated Success
      Sequence[TestTurnLeft] updated Success
    Selector[TestTurning] updated Success
    Selector[CheckGoalStatus] updating
      Sequence[PopWaypointOrExit] updating
        AssertionNode[7] updating
        Distance to target: 4.84090294887801
        AssertionNode[7] updated Success
      Selector[PopOrExit] updating
        ActionNode[9] updating
        ActionNode[9] updated Success
      Selector[PopOrExit] updated Success
    Sequence[PopWaypointOrExit] updated Success
    Selector[CheckGoalStatus] updated Success
  Sequence[MainSequence] updated Success
```

Figure 126 - WaypointFollowing.txt: The waypoint following demo behavior tree file output

7.3.5 Step 5: Executing the Tree Using the REMUS Simulator

Now that the tree has been constructed we can run a simulation using the REMUS simulator. The project is setup to run inside of a standard windows console. It creates a communication link with the REMUS simulator using a UDP socket. It then waits for the user to press the 'ENTER' key to begin traversing the behavior tree. Four (4) waypoints located in the Gulf of Mexico near Panama City, Florida have been entered as the vehicle target waypoints. When the user presses ENTER, root.Tick() is called and the vehicle will begin trying to reach each one in series. As shown in Figure 127, this small tree successfully commands the vehicle to navigate to each waypoint. The relative ease of implementing the waypoint following behavior suggests that more sophisticated

algorithms are equally achievable. Further development and experimentation will show great potential for adoption.

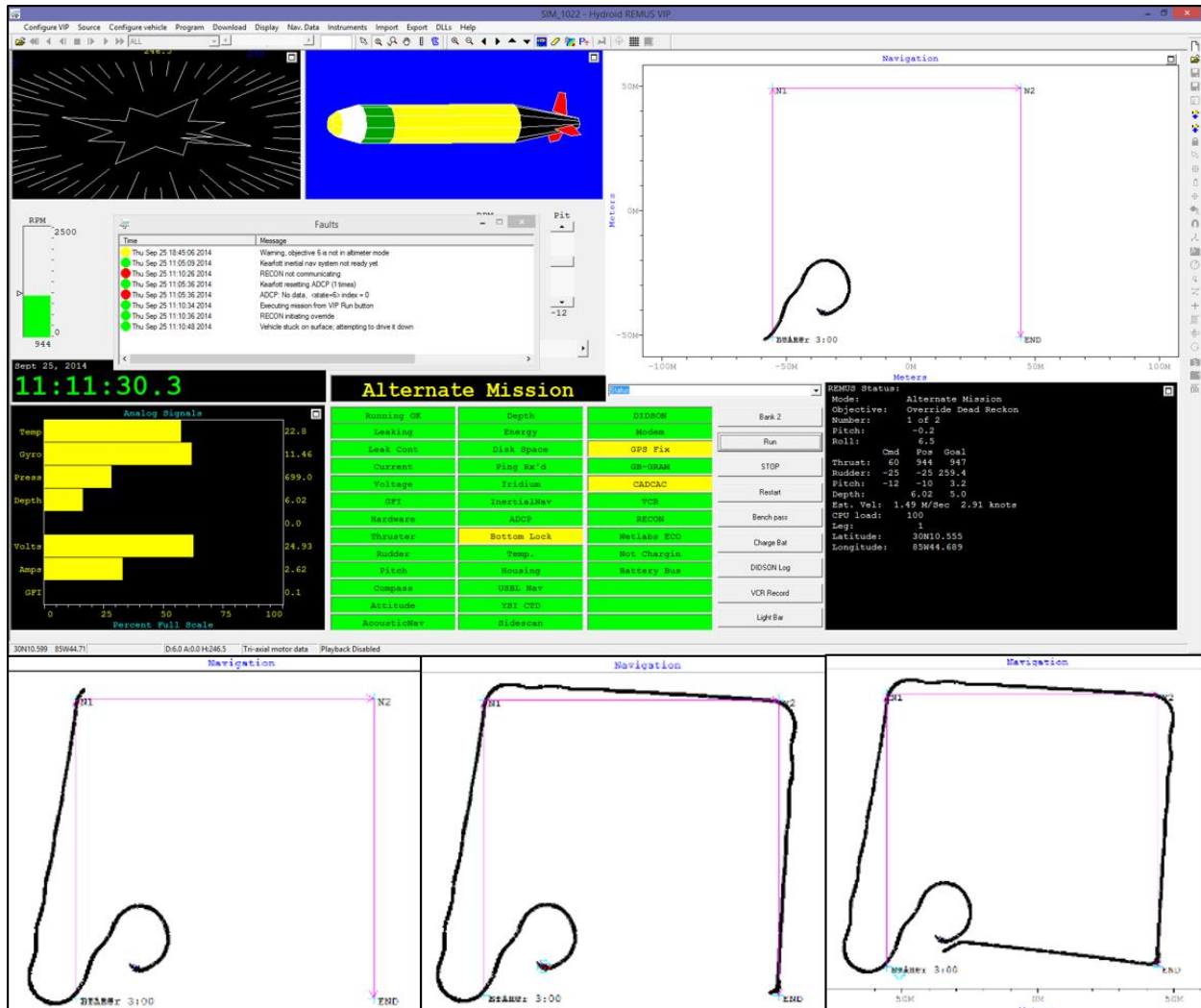


Figure 127 – The vehicle simulator output screen after a successful experiment with the proposed waypoint following behavior tree

7.3.6 Summary

The proposed behavior tree framework was capable of integrating with an existing REMUS vehicle control library with very little effort. The constructed behavior tree successfully navigated

the four waypoints that it was given. A logical next step in this direction would be to integrate behavior trees with the Benthos acoustic modems available onboard.

7.4 Performance Cost

The increased flexibility brought about by utilizing behavior trees comes with some processing overhead. Performance tests were run utilizing three different versions of the A* algorithm presented earlier. The first version is coded inside of a single method making method calls directly to local references to the frontier and cells. The second version is the behavior tree representation shown in Figure 107. The third version is the same behavior tree version but instantiated using the reflective loader and the XML file shown in Figure 112. Each of these algorithms were then used to calculate the distance from cell (0, 0) to every other cell in a 128x128 bitmap terrain image. There were a total of 16,384 calculations for each algorithm.

The following test was run on an Intel® Core™ i7 CPU @ 2.5 GHz with 8 GB of RAM. The operating system was Windows 8.1 x64. As expected, the sequential code version of the algorithm runs the fastest, with the behavior tree versions taking approximately 2.05 times as long. The results of the test are shown in Table 18.

Table 18 - The results of an operational time comparison between a sequential A* algorithm, a hardcoded behavior tree representation, and a reflective instantiation of a behavior tree

Algorithm Name	Iterations	Total Milliseconds	Milliseconds per iteration	Speed Ratio
AStar	16384	17247.1528	1.0527	1.0000
BTAStar	16384	35519.1439	2.1679	2.0594
BTAStarXml	16384	35375.9262	2.1592	2.0511

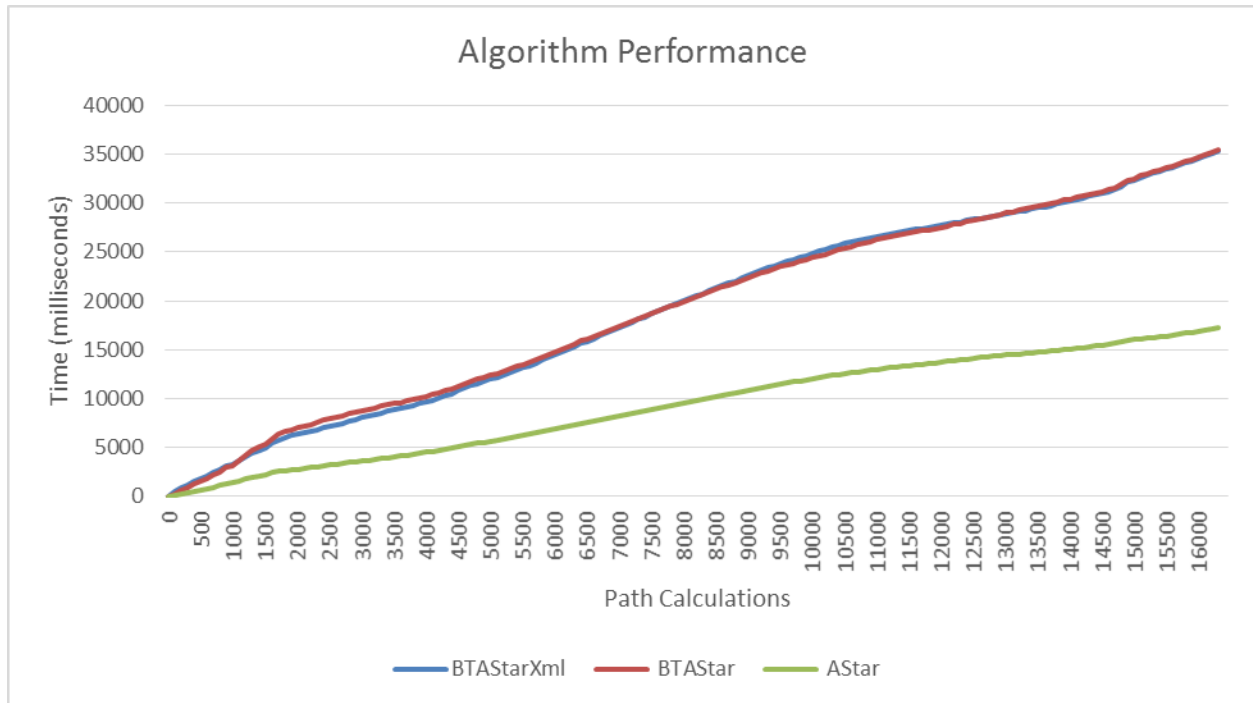


Figure 128 – A chart representing the data shown in Table 18

The slight edge that is shown by BTAStarXml representation in the performance results is interesting. The most probable explanation is pressure from other processes running simultaneous to the experiment. Subsequent tests show similar results with the BTAStar and BTAStarXml versions taking turns having the fastest completion time. The main takeaway from the similarity between these two versions is that the reflective instantiation does not impose any additional operational overhead. It has a single upfront cost taken care of at program start up.

The question of whether an algorithm taking twice as long to run is acceptable arises. The answer to that question is up to the individual leader of a particular software product. However, some cases come to mind where it would and wouldn't be acceptable. In the case of the mobile acoustic node locator, described in section 4.1, a path planning time of around 2 ms would be acceptable. The algorithm only needs to run one time for each waypoint in the set and then potentially twice when investigating the intersections of the constructed circles (used when localizing a node, see

section 7.2). The algorithm replans whenever impassable terrain is encountered as well. On multi-core processors, this delay is mitigated by the fact that the other independent processes can continue to operate without delays from this approach. Situations where it might not be feasible to use a behavior tree would be when dealing with high performance control software that integrates sensory input with aileron motion or propeller thrust commands or high performance computing models that integrate several gigabytes of data as fast as possible.

7.5 Algorithm Representation Potential

In 1966, Bohm and Jacopini related the Church-Turing Thesis to programming languages stating that any programming language capable of representing sequential composition, selection structures, and repetition structures could represent any conceivable algorithm [45]. Since behavior trees contain methods for representing each of these structures, they too can represent any conceivable algorithm.

Sequential Composition - The sequence node naturally represents the sequential ordering of programming statements, operating on each one in the order they are placed in the sequence.

Selection Structure – The selector node portrays the necessary characteristics to satisfy this requirement by operating on nodes in an if-else if manner. Follow Figure 129 as an example. The first sequence will attempt to operate on each of its nodes in order until a failure is received. Since the first node returns failure, this sequence will also return failure to its parent, the selector. The selector will then tick Seq2 in hopes that it will return success. Since the first node returns success, the resulting NodeStatus is dependent on the result of activating Action2.

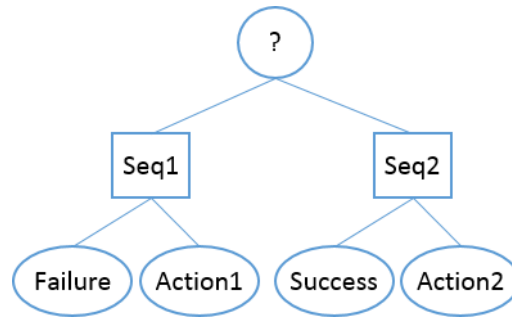


Figure 129 - A graphical representation of an if-else-if structure using behavior tree nodes

Repetition Structure – To address the repetition structure, Figure 100 is reintroduced to the reader as Figure 130. As discussed in section 6.1.1, a standard ‘for-loop’ can be simulated using the generic behavior tree nodes introduced in this paper. The argument type of the nodes shown in the figure need only contain a field representing the iteration variable. The PreOp node will take care of checking for the appropriateness of acting on the loop body (e.g `i < someNumber`) and the PostOp node will handle to pose operation such as incrementing (`i++`) or decrementing (`i--`) the iteration variable. When the iteration variable exceeds the desired count, the PreOp node returns false, causing the while true decorator node to return false.

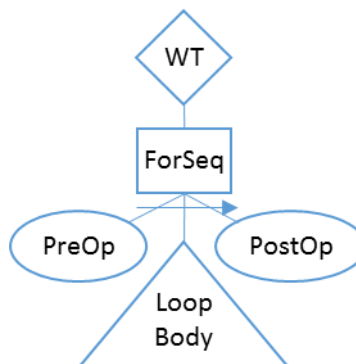


Figure 130 - A behavior tree representation of a for loop

CHAPTER 8

CONCLUSION AND FUTURE RESEARCH POTENTIAL

8.1 Conclusion

Developing a generic framework capable of enabling limitless behavior tree configurations was very challenging. The resulting capability, however, shows the potential for very quick development and instantiation of algorithms that are able to operate across autonomous vehicle platform boundaries. Initial development showed that common path planning and localization algorithms worked well and were very extensible in simulated environments. Later work showed that the transition to an actual REMUS simulator was virtually seamless and presented no new limitation to BT potential. This work demonstrates that the investment made in utilizing a behavior tree approach will reward the investors in the end.

Furthermore, no new limitations in programming flexibility were discovered. Developers who prefer a particular programming styles are free to do so. Also, the behavior tree granularity is completely left up to the designer. For example, one behavior tree node could contain an entire algorithm spanning thousands of lines of code. On the other hand, the behavior tree could be broken down line by line into nodes which give much greater modification flexibility and potential for reuse but come with computational overhead tradeoffs.

8.2 Future Research Potential

8.2.1 Autonomous Vehicle Sensor Integration

This research showed reasonable applications in land based and underwater vehicle autonomy. Future implementations might seek to take advantage of common sensor platforms and offer these

as components to other behavior tree developers. Open systems designers could host these components on open source sites and collaborate with others to refine and optimize the performance. Commercial developers could fork development efforts and invest in high quality, high performance behavior tree libraries for sale to other commercial developers.

8.2.2 Graphic Based Platform Integration

The XML based reflective instantiation of behavior trees discussed in this research enables excellent potential for integration with graphic based design platforms. The industry already offers products like LabVIEW (<http://www.ni.com/labview>), where users can drag and drop software operations onto the screen and connect them with buses to control the flow of data. Having a graphical designer capable of integrating behavior tree functionality would offer limitless applications to every area of industry.

8.2.3 Genetic Algorithm Based Auto-Code Application

The common infrastructure offered by this research offers unlimited reconfiguration capability. Genetic algorithms could be developed that automatically reconfigure code and test it using predefined performance metrics with the end goal of highly optimized code. As capabilities are introduced and inserted into the action and assertion repositories, the GAs would automatically attempt to insert the functionality into the tree, evolving over time to produce proper behavior tree algorithms suited to the application desired.

REFERENCES

- [1] National Oceanic and Atmospheric Administration, "Ocean," 11 October 2012. [Online]. Available: <http://www.noaa.gov/ocean.html>. [Accessed 12 October 2012].
- [2] E. M. Sozer, M. Stojanovic and J. G. Proakis, "Underwater Acoustic Networks," *IEEE Journal of Oceanic Engineering*, vol. 25, no. 1, pp. 72-83, 2000.
- [3] M. Chitre, S. Shahabudeen, L. Freitag and M. Stojanovic, "Recent Advances in Underwater Acoustic Communications and Networking," in *OCEANS*, vol. 2008, IEEE, 2008.
- [4] S. Misra and A. Ghosh, "The Effects of Variable Sound Speed on Localization in Underwater Sensor Networks," in *Australasian Telecommunication networks and Applications Conference (ATNAC) 2011*, Melbourne, 2011.
- [5] Z. Jiang, "Underwater Acoustic Networks - Issues and Solutions," *International Journal of Intelligent Control and Systems*, vol. 13, no. 3, pp. 152-161, 2008.
- [6] LinkQuest Inc., "SoundLink Underwater Acoustic Modems," [Online]. Available: http://www.link-quest.com/html/uwm_hr.pdf. [Accessed 24 January 2013].
- [7] DSPCOMM, "AquaComm: Underwater Wireless Modem," [Online]. Available: http://www.dspcomm.com/products_aquacomm.html. [Accessed 24 January 2013].
- [8] Teledyne Benthos, "Acoustic Modems," Teledyne, 2013. [Online]. Available: <http://www.benthos.com/acoustic-teleonar-modem-product-comparison.asp>. [Accessed 24 January 2013].
- [9] Woods Hole Oceanographic Institution, "Micro-Modem Overview," WHOI, 2013. [Online]. Available: <http://acomms.whoi.edu/umodem/>. [Accessed 24 January 2013].
- [10] M. Stojanovic, "Underwater Acoustic Communications: Design Considerations on the Physical Layer," in *Wireless on Demand Network Systems and Services, 2008. Fifth Annual Conference on*, IEEE, 2008.
- [11] J. Rice, B. Creber, C. Fletcher, P. Baxley, K. Rogers, K. McDonald, D. Rees, M. Wolf, S. Merriam, R. Mehio, J. Proakis, K. Scussel, D. Porta, J. Baker, J. Hardiman and D. Green, "Evolution of Seaweb Underwater Acoustic Networking," *OCEANS 2000 MTS/IEEE Conference and Exhibition*, vol. 3, pp. 2007-2017, 2000.

- [12] I. F. Akyildiz, D. Pompili and T. Melodia, "State-of-the-Art in Protocol Research for Underwater Acoustic Sensor Networks," in *ACM International Workshop on Underwater Networks*, Los Angeles, 2006.
- [13] M. K. Wafta, T. Nsouli, M. Al-Ayache and O. Ayyash, "Reactive Localization in Underwater Wireless Sensor Networks," in *Second International Conference on Computer and Network Technology (ICCNT)*, Bangkok, 2010.
- [14] A. Y. Teymorian, W. Cheng, L. Ma, X. Cheng, X. Lu and Z. Lu, "3D Underwater Sensor Network Localization," *IEEE Transactions on Mobile Computing*, vol. 8, no. 12, pp. 1610-1621, 2009.
- [15] Teledyne Benthos, "Teledyne Benthos Modems Product Catalog 2012 Rev. 32612," [Online]. Available: <http://www.benthos.com/undersea-document-downloads.asp>. [Accessed 12 October 2012].
- [16] J. Partan, J. Kurose and B. N. Levine, "A Survey of Practical Issues in Underwater Networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 11, no. 4, pp. 23-33, 2007.
- [17] Woods Hole Oceanographic Institute, "REMUS," [Online]. Available: www.whoi.edu. [Accessed 13 January 2013].
- [18] Kongsberg Maritime, "Autonomous Underwater Vehicles - REMUS AUVs," [Online]. Available: www.km.kongsberg.com. [Accessed 13 January 2013].
- [19] Bluefin Robotics, "Bluefin-9," Bluefin Products, 2013. [Online]. Available: <http://www.bluefinrobotics.com/products/>. [Accessed 24 January 2013].
- [20] T. C. Austin, R. P. Stokey and K. M. Sharp, "PARADIGM: A Buoy-Based System for AUV Navigation and Tracking," in *OCEANS 2000 MTS/IEEE Conference and Exhibition*, Providence, 2000.
- [21] P. L. Frana and T. J. Misa, "An interview with Edsger W. Dijkstra," *Viewpoints*, vol. 53, no. 8, pp. 41-47, 2010.
- [22] R. Stephens, *Essential Algorithms: A Practical Approach to Computer Algorithms*, Indianapolis: John Wiley & Sons, Inc., 2013.
- [23] ECMA International, *Standard ECMA 335 Common Language Infrastructure*, 2012.
- [24] International Organization for Standardization, "ISO/IEC 23271 Information Technology - Common Language Infrastructure (CLI)," 15 February 2012. [Online]. Available: <http://standards.iso.org>. [Accessed 30 January 2013].

- [25] A. Troelsen, *Pro C# 5.0 and the .NET 4.5 Framework*, New York: Apress, 2012.
- [26] ECMA International, *Standard ECMA-334 C# Language Specification*, 2006.
- [27] Xamarin, "MONO," Xamarin, 2013. [Online]. Available: http://www.mono-project.com/Main_Page. [Accessed 29 January 2013].
- [28] MonoGame Team, "MonoGame About Page," 2014. [Online]. Available: <http://www.monogame.net/about/>. [Accessed 23 February 2015].
- [29] J. Albahari and B. Albahari, *C# 5.0 In a Nutshell*, Sebastopol: O'Reilly, 2012.
- [30] A. Aamodt and E. Plaza, "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches," *AI Communications*, vol. 7, no. 1, pp. 39-59, 1994.
- [31] World Wide Web Consortium, "XML Essentials," W3C, 2010. [Online]. Available: <http://www.w3.org/standards/xml/core>. [Accessed 12 February 2013].
- [32] B. Evjen, K. Sharkey, T. Thangarathinam, M. Kay, A. Vernet and S. Ferguson, *Professional XML*, Indianapolis: Wiley, 2007.
- [33] Y. Hoshino, T. Takagi, U. Di Profio and M. Fujita, "Behavior Description and Control Using Behavior Module for Personal Robot," in *IEEE Conference on Robotics and Automation*, New Orleans, 2004.
- [34] G. Florez-Puga, M. A. Gomez-Martin, P. P. Gomez-Martin, B. Diaz-Agudo and P. A. Gonzalez-Calero, "Query-Enabled Behavior Trees," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 298-308, 2009.
- [35] V. Berenz and K. Suzuki, "Usability Benchmarks of the Targets-Drive-Means Robotic Architecture," in *International Conference on Humanoid Robots*, Osaka, Japan, 2012.
- [36] J. Albiez, S. Joyeux and M. Hildebrandt, "Adaptive AUV Mission Management in Under-Informed Situations," in *Oceans*, Seattle, 2010.
- [37] A. Blasuriya, S. Petillo, H. Schmidt and M. Benjamin, "Behavior-Based Planning and Prosecution Architecture for Autonomous Underwater Vehicles in Ocean Observatories," in *OCEANS*, Sydney, 2010.
- [38] M. Erol, L. F. Vieira and M. Gerla, "AUV-Aided Localization for Underwater Sensor Networks," in *International Conference on Wireless Algorithms, Systems and Applications 2007. WASA 2007*, Chicago, 2007.

- [39] C. H. Yu, K. H. Lee, H. P. Moon, J. W. Choi and Y. B. Seo, "Sensor Localization Algorithms in Underwater Wireless Sensor Networks," in *ICCAS-SICE*, Fukuoka, 2009.
- [40] Office of the Deputy Assistant Secretary of Defense, "Systems Engineering," 12 September 2012. [Online]. Available: http://www.acq.osd.mil/se/initiatives/init_osa.html. [Accessed 15 October 2012].
- [41] Open Systems Joint Task Force, "A Modular Open Systems Approach to Acquisition," September 2004. [Online]. Available: http://www.acq.osd.mil/osjtf/pdf/PMG_04.pdf. [Accessed 23 08 2012].
- [42] JAUS Working Group, "Reference Architecture Specification Volume II, Part 1 v3.3 Architecture Framework," 27 June 2013. [Online]. Available: <http://www.openjaus.com/support/39-documentation/79-jaus-documents>. [Accessed 8 February 2013].
- [43] H. Aghazarian, E. Baumgartner and M. Garrett, "Robust Software Architecture for Robots," *NASA Tech Briefs*, p. 36, 2009.
- [44] Microsoft, "Assembly Class," 2015. [Online]. Available: [https://msdn.microsoft.com/en-us/library/system.reflection.assembly\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.reflection.assembly(v=vs.110).aspx). [Accessed 13 March 2015].
- [45] C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages With Only Two Formation Rules," *Communications of the ACM*, vol. 9, no. 5, pp. 336-371, 1966.
- [46] Lockheed Martin, "Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program," December 2005. [Online]. Available: http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc. [Accessed 25 January 2013].
- [47] S. Prata, C++ Primer Plus, Sixth Edition, New Jersey: Addison-Wesley Professional, 2011.
- [48] NPD Group, "Research Shows \$14.80 Billion Spend on Video Game Content in the U.S. for 2012," 2013. [Online]. Available: <https://www.npd.com/wps/portal/npd/us/news/press-releases/research-shows-14.80-billion-spent-on-video-game-content-in-the-us-for-2012/>. [Accessed 15 February 2013].
- [49] J. Orkin, "12 Tips From The Trenches," in *AI Game Programming Wisdom*, Hingham, Jenifer Niles, 2002, pp. 29-35.

BIOGRAPHICAL SKETCH

Jeremy Hatcher received his Bachelor's degree in electrical engineering from Florida State University in 2007. He was immediately hired at the Naval Surface Warfare Center in Panama City, Florida by the Littoral Warfare Research Facility. For the past 8 years, he has been working with and developing software for a variety of ground and underwater robots. He also served as the lead software engineer on several large projects including the Littoral Combat Ship Mission Module Automation Program and Mission Package Automated Inventory System. He received his Master's degree in computer science, specializing in software engineering, from Florida State University in 2011. That same year he received a Science Math and Research for Transformation (SMART) Scholarship which he used to pursue his PhD, also in computer science, at FSU. Upon completion, Jeremy intends to continue researching and improving the state of underwater autonomy.