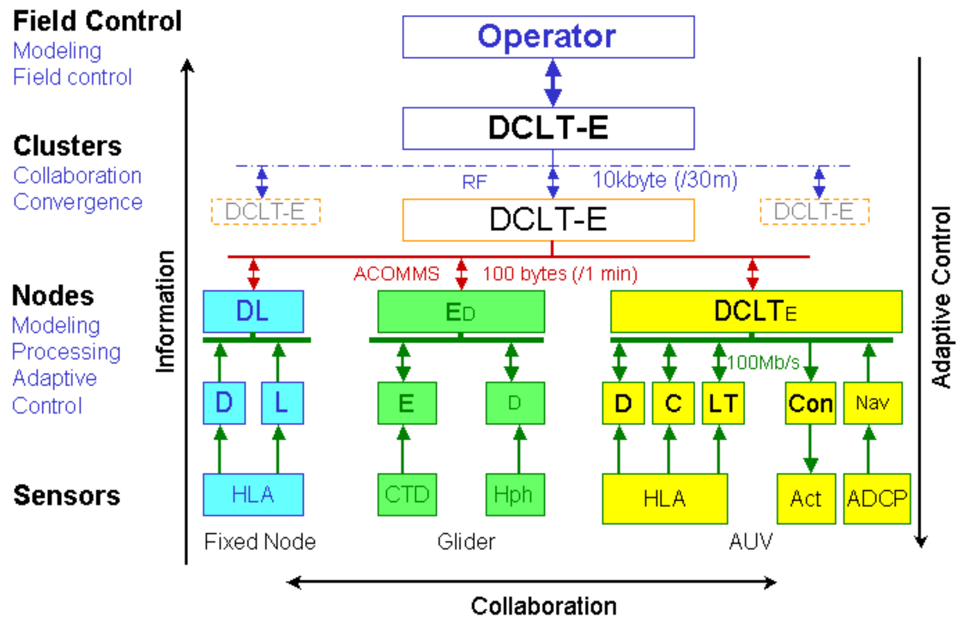

MOOS-IvP
Nested Autonomy Prototype for Distributed Undersea Sensing

User's Guide

H. Schmidt, M.R. Benjamin, A. Balasuriya, D. Battle, T. Schneider, and K. Cockrell

Laboratory for Autonomous Marine Sensing
Department Mechanical Engineering
Massachusetts Institute of Technology, Cambridge MA

January 12, 2009

Abstract

This paper provides an introductory description and User's Guide for the MIT MOOS-IvP Undersea Autonomous Network Architecture, based on the Nested Autonomy paradigm, as implemented and operated on the fleet of Bluefin21 autonomous Underwater vehicles applied by the Laboratry for Autonomous Marine sensing for research and development into distributed acoustic sensing in the ocean.

Contents

I Introduction **10**

1 Overview **10**
1.1 Purpose and Scope of this Document 10
1.2 Distributed Undersea Sensing and Observation 10

II Nested Autonomy Paradigm for Distributed Undersea Sensing **11**

2 Autonomy in Communication-Challenged Environments **11**

3 Nested Autonomy for Ocean Observation Systems **14**

4 Concept of Operations **16**
4.1 Field-level CONOPS 16
4.2 Cluster-level CONOPS 17
4.3 Node-level CONOPS 18

5 Cluster Autonomy System **20**

6 Platform Autonomy System **21**
6.1 The `MOOS-IvP` Autonomy Architecture 21
6.2 The Back-Seat Driver Paradigm 21
6.3 Communication Connectivity 22
6.4 Autonomy States and State-Transitions 24
6.5 On-board, Real-time Signal Processing 26
6.6 `MOOS-IvP` Simulation Environment 27

III MIT-LAMS Nested Autonomy Prototype **28**

7 MOOS-IvP Nested Autonomy Architecture **28**
7.1 Sonar AUV MOOS Community 28

IV IvP-Helm Autonomy for Undersea Sensing **32**

8 Mission Planning and Control **32**

9 Hierarchial State Representation **32**
9.1 MOOS-IvP State Definitions 32
9.2 Mission Terminating States 34
9.3 Deploy States 35
9.3.1 Loiter Sub-States 36

9.3.2 Racetrack Sub-State . . . 37
9.3.3 ZigZag Sub-state . . . 38
9.3.4 Trail Sub-state . . . 39
9.4 Prosecute States . . . 40
9.4.1 Search Sub-State . . . 41
9.4.2 Ambiguous Sub-State . . . 43
9.4.3 Tracking Sub-State . . . 44
9.4.4 Classification Sub-State . . . 45

10 Cluster Autonomy Behaviors **47**
10.1 BHV_Attractor . . . 47
10.1.1 MOOS variables subscribed to by BHV_Attractor: . . . 48
10.1.2 MOOS variables published by BHV_Attractor: . . . 49
10.2 BHV_RubberBand . . . 49
10.2.1 MOOS variables subscribed to by BHV_RubberBand: . . . 50
10.2.2 MOOS variables published by BHV_RubberBand: . . . 50

11 Undersea Adaptive Sensing Behaviors **51**
11.1 BHV_HArrayTurn . . . 51
11.1.1 MOOS variables subscribed to by BHV_HArrayTurn: . . . 51
11.1.2 MOOS variables published by BHV_HArrayTurn: . . . 51
11.2 BHV_HArrayAngle . . . 51
11.2.1 MOOS variables subscribed to by BHV_HArrayAngle: . . . 52
11.2.2 MOOS variables published by BHV_HArrayAngle: . . . 52
11.3 BHV_SmartYoyo . . . 52
11.3.1 MOOS variables subscribed to by BHV_SmartYoyo: . . . 54
11.3.2 MOOS variables published by BHV_SmartYoyo: . . . 54
11.4 BHV_ZigZag . . . 54
11.4.1 MOOS variables subscribed to by BHV_ZigZag: . . . 55
11.4.2 MOOS variables published by BHV_ZigZag: . . . 55
11.5 BHV_RaceTrack . . . 55
11.5.1 MOOS variables subscribed to by BHV_RaceTrack: . . . 56
11.5.2 MOOS variables published by BHV_RaceTrack: . . . 57

V Autonomous Communication, Command and Control **58**

12 MOOS-IvP Communication, Command and Control **58**
12.1 Node Level Command and Control . . . 59
12.2 Cluster Level Command and Control . . . 59
12.3 Field Level Command and Control . . . 61

13 Communication, Command and Control Modules **64**
13.1 pNaFCon . . . 64
13.1.1 Brief Overview . . . 64
13.1.2 Parameters for the pNaFCon Configuration Block . . . 64

13.1.3 MOOS variables subscribed to by pNaFcon: 65
13.1.4 MOOS variables published by pNaFCon: 65
13.2 pMessageSim 67
13.2.1 Brief Overview 67
13.2.2 Parameters for the pMessageSim Configuration Block 67
13.2.3 MOOS variables subscribed to by pMessageSim: 69
13.2.4 MOOS variables published by pMessageSim: 69
13.3 pSearch 70
13.3.1 Brief Overview 70
13.3.2 Parameters for the pSearch Configuration Block 70
13.3.3 MOOS variables subscribed to by pSearch: 70
13.3.4 MOOS variables published by pSearch: 71
13.4 pTrackQuality 72
13.4.1 Brief Overview 72
13.4.2 Parameters for the pTrackQuality Configuration Block 72
13.4.3 MOOS variables subscribed to by pTrackQuality: 73
13.4.4 MOOS variables published by pTrackQuality: 75
13.5 pTargetOpportunity 76
13.5.1 Brief Overview 76
13.5.2 Parameters for the pTargetOpportunity Configuration Block 76
13.5.3 MOOS variables subscribed to by pTargetOpportunity: 77
13.5.4 MOOS variables published by pTargetOpportunity: 77
13.6 pClusterPriority 78
13.6.1 Brief Overview 78
13.6.2 Parameters for the pClusterPriority Configuration Block 78
13.6.3 MOOS variables subscribed to by pClusterPriority: 78
13.6.4 MOOS variables published by pClusterPriority: 79
13.7 pHuxley 80
13.7.1 Brief Overview 80
13.7.2 Parameters for the pHuxley Configuration Block 80
13.7.3 MOOS variables subscribed to by pHuxley: 80
13.7.4 MOOS variables published by pHuxley: 80
13.8 pGeneralCodec 81
13.8.1 Brief Overview 81
13.8.2 Parameters for the pGeneralCodec Configuration Block 82
13.8.3 MOOS variables subscribed to by pGeneralCodec: 82
13.8.4 MOOS variables published by pGeneralCodec: 83
13.8.5 Usage 83
13.8.6 Details 92
13.8.7 Further examples 93
13.8.8 Message XML reference sheet 94
13.8.9 Glossary 96
13.9 pAcommsHandler 97
13.9.1 Brief Overview 97
13.9.2 usage 97

13.9.3 Parameters for the pAcommsHandler Configuration Block 97
13.9.4 MOOS variables subscribed to by pAcommsHandler: 100
13.9.5 MOOS variables subscribed to by pAcommsHandler: 101
13.9.6 details 102
13.10pAcommsPoller 103
13.10.1 Brief Overview 103
13.10.2 Parameters for the pAcommsPoller Configuration Block 103
13.10.3 MOOS variables subscribed to by pAcommspoller: 103
13.10.4 MOOS variables published by pAcommsPoller: 103
13.11iMicroModem 104
13.11.1 Brief Overview 104
13.11.2 Parameters for the pAcommsPoller Configuration Block 104
13.11.3 MOOS variables subscribed to by iMicroModem: 104
13.11.4 MOOS variables published by iMicroModem: 104

VI Real-Time Acoustic Sensing and Processing 105

14 On-Board Acoustic Signal Processing 105
14.1 MOOS-IvP Signal Processing Architecture 105

15 Acoustic Sensing and Processing Modules 107
15.1 pBearingTrack 107
15.1.1 Brief Overview 107
15.1.2 Parameters for the pBearingTrack Configuration Block 107
15.1.3 MOOS variables subscribed to by pBearingTrack: 109
15.1.4 MOOS variables published by pBearingTrack: 109
15.2 p1HTracker 111
15.2.1 Brief Overview 111
15.2.2 Parameters for the p1HTracker Configuration Block 111
15.2.3 MOOS variables subscribed to by p1HTracker: 112
15.2.4 MOOS variables published by p1HTracker: 113
15.3 pMBTracker 114
15.3.1 Brief Overview 114
15.3.2 Configuration MOOS-block 114
15.3.3 MOOS variables subscribed to: 114
15.3.4 MOOS variables published: 114

VII Network Command and Control 115

16 Topside Command and Control 115
16.1 Topside MOOS Community 115
16.2 Launching the Command and Control Topside 116

17 Topside Command and Control Modules **120**
17.1 NaFConSim . . . 120
17.1.1 Brief Overview . . . 120
17.1.2 Parameters for the NaFConSim Configuration Block . . . 120
17.1.3 MOOS variables subscribed to by pNaFConSim: . . . 121
17.1.4 MOOS variables published by NaFConSim: . . . 121
17.2 pHuxley . . . 122
17.2.1 Brief Overview . . . 122
17.2.2 Parameters for the Topside pHuxley Configuration Block . . . 122
17.3 pGeneralCoDec . . . 123
17.3.1 Brief Overview . . . 123
17.4 pAcommsHandler . . . 124
17.4.1 Brief Overview . . . 124
17.4.2 Topside pAcommsHandler Configuration Block . . . 124
17.5 pAcommsPoller . . . 125
17.5.1 Brief Overview . . . 125
17.5.2 Configuration MOOS-block . . . 125
17.5.3 MOOS variables subscribed to by pAcommspoller: . . . 125
17.5.4 MOOS variables published by pAcommsPoller: . . . 125
17.6 iMicroModem . . . 126
17.6.1 Brief Overview . . . 126
17.6.2 Configuration MOOS-block . . . 126
17.6.3 MOOS variables subscribed to by iMicroModem: . . . 126
17.6.4 MOOS variables published by iMicroModem: . . . 126
17.7 uNafconMessageViewer . . . 127
17.7.1 Brief Overview . . . 127
17.7.2 Topside uNafconMessageViewer Configuration Block . . . 127
17.7.3 MOOS variables subscribed to by uNafconMessageViewer: . . . 127
17.7.4 MOOS variables published by uNafconMessageViewer: . . . 127

VIII Sensing Network Simulation Environment **128**

18 MIT Undersea Autonomous Network Simulator **128**
18.1 Sonar-AUV Simulator . . . 129
18.2 Running a Simulation Session . . . 130
18.2.1 pAntler Simulation MOOS Block . . . 130
18.2.2 pHuxley Simulation MOOS Block . . . 134
18.2.3 iModemSim Setup . . . 134
18.2.4 AUV Mission Launch . . . 135
18.2.5 Topside Launch . . . 135

19 Sonar AUV Simulation Modules and Utilities 137
19.1 pTargetSim . . . 137
19.1.1 Brief Overview . . . 137
19.1.2 Parameters for the pTargetSim Configuration Block . . . 137
19.1.3 MOOS variables subscribed to by pTargetSim: . . . 137
19.1.4 MOOS variables published:by pTargetSim . . . 138
19.2 pBearingsSim . . . 139
19.2.1 Brief Overview . . . 139
19.2.2 Parameters for the pBearingsSim Configuration Block . . . 139
19.2.3 MOOS variables subscribed to by pBearingsSim: . . . 139
19.2.4 MOOS variables published by pBearingsSim: . . . 140
19.3 pArraySim . . . 142
19.3.1 Brief Overview . . . 142
19.3.2 Parameters for the pArraySim Configuration Block . . . 142
19.3.3 MOOS variables subscribed to by pArraySim: . . . 144
19.3.4 MOOS variables published by pArraySim: . . . 144
19.4 pMultiTargetSim . . . 145
19.4.1 Brief Overview . . . 145
19.4.2 Parameters for the pMultiTargetSim Configuration Block . . . 145
19.4.3 MOOS variables subscribed to by pMultiTargetSim: . . . 145
19.4.4 MOOS variables published by pMultiTargetSim: . . . 146
19.5 pMultiAcousticSim . . . 147
19.5.1 Brief Overview . . . 147
19.5.2 Parameters for the pMultiAcousticSim Configuration Block . . . 147
19.5.3 MOOS variables subscribed to by pMultiAcousticSim: . . . 148
19.5.4 MOOS variables published by pMultiAcousticSim: . . . 148
19.5.5 pMultiAcousticSim Details . . . 149
19.6 pGPSSim . . . 150
19.6.1 Brief Overview . . . 150
19.6.2 Parameters for the pGPSSim Configuration Block . . . 150
19.6.3 MOOS variables subscribed to by pGPSSim: . . . 150
19.6.4 MOOS variables published by pGPSSimpin: . . . 150
19.7 uCtdSim2 . . . 151
19.7.1 Brief Overview . . . 151
19.7.2 Parameters for the uCtdSim2 Configuration Block . . . 151
19.7.3 MOOS variables subscribed to by uCtdSim2: . . . 151
19.7.4 MOOS variables published by uCtdSim2: . . . 151
19.8 uBathy . . . 152
19.8.1 Brief Overview . . . 152
19.8.2 Parameters for the uBathy Configuration Block . . . 152
19.8.3 MOOS variables subscribed to by uBathy: . . . 152
19.8.4 MOOS variables published by uBathy: . . . 152
19.9 Arraysim.m . . . 153
19.9.1 Brief Overview . . . 153
19.9.2 Configuration Files . . . 153

19.9.3 MOOS variables subscribed to: 153
19.9.4 MOOS variables published: 154
19.10SealabMultiSim.m 155
19.10.1 Brief Overview 155
19.10.2 Configuration MOOS-block 155
19.10.3 MOOS variables subscribed to: 156
19.10.4 MOOS variables published: 156
19.11PassiveTgtSim.m 157
19.11.1 Brief Overview 157
19.11.2 Usage 157
19.11.3 Configuration MOOS-block for `PassiveTgtSim` 157
19.11.4 MOOS variables subscribed to: 158
19.11.5 MOOS variables published: 158

A Appendix - pGeneralCodec Configuration Files 161

Part I
Introduction

1 Overview

1.1 Purpose and Scope of this Document

The purpose of this document is to provide the reader with an overview of the Nested Autonomy paradigm for undersea observation system, followed by to a catalog style overview and User's Guide for the modules and utilities used for the MOOS-IvP autonomy system applied for implementing and operating undersea distributed networks for acoustic sensing in the ocean environment.

The scope of discussion includes, for each module, a brief description of the module function, authorship, source for download, rough measure of complexity, and module dependencies. Further, for use by developers of onboard processing modules, for example, the description includes a detailed listing of MOOS variables published by or subscribed to by each module.

¡¡¡¡¡¡¡ .mine ======= ¿¿¿¿¿¿¿ .r621

1.2 Distributed Undersea Sensing and Observation

Underwater acoustic sensing and surveillance is undergoing a dramatic paradigm shift from platform-centric, human-controlled sensing, processing and interpretation, toward distributed sensing concepts using networks of autonomous underwater vehicles. Being dependent on acoustic communication with a channel capacity many orders of magnitude smaller than the air- and land-based equivalents, the operation of such new undersea surveillance systems require a much higher level of autonomous, distributed data processing and control that land- and air-based equivalents. *Nested Autonomy* is a new command and control paradigm, inherently suited for the layered communication infrastructure provided by the low-bandwidth underwater acoustic communication and the intermittent RF connectivity. Implemented using the open-source MOOS-IvP behavior-based, autonomous command and control architecture, it provides the fully integrated sensing, modeling an control that allows each platform to autonomously detect, classify, localize and track an episodic event in the ocean, without depending on any operator command and control. The prosecution of an event, such as the detection and tracking of a sub-sea vulcanic plume may be initiated by the operators or fully autonomously by an onboard detector. The event information collected by each node in the network is reported back to the operators by transmitting an event report, using a dedicated command and control language. Collaborative processing and control is exploited when the communication channel allows, e.g. for collaborative tracking of an acoustic source, such as a marine mammal.

Part II

Nested Autonomy Paradigm for Distributed Undersea Sensing

2 Autonomy in Communication-Challenged Environments

The primary motivation for designing a distributed command and control architecture for a undersea monitoring and observation is to achieve the ability to deploy a fleet of autonomous mobile marine platforms over a wide area of the ocean environment and over a long period of time with little or no human supervision. Concerns over effective coverage, communication range and safe operation of the platforms are all primary motivations of an effective form of autonomous control. The long duration and unpredictable nature of the environment require the vehicles to adapt their missions and behave autonomously as events unfold. Conversely, practical concerns of marine operations over large areas require an element of operator predictability over the course of time. These two characteristics can be at odds with each other in practice, but can be tempered by effective periodic communication through a network of fixed and mobile nodes co-deployed in a coordinated manner designed to balance individual platform and network objectives.

The connectivity with and between the submerged assets of such networks is almost entirely dependent on underwater acoustic communication, except for rare and time-limited surfacing. Consequently, the undersea network nodes must operate with a communication infrastructure with severely limited bandwidth. Thus current underwater communication technology can robustly provide a point-to-point channel capacity in shallow water of less than a few hundred byte-km/minute, close to ten orders of magnitude smaller than modern electromagnetic communication protocols used for land- and air-based distributed, net-centric systems. Equally critical is the high latency and short communication windows inherently associated with communication between the human operator and the submerged assets, more severe than that experienced in interplanetary space exploration. Thus, operational constraints for some applications prohibit the existence of permanent surface assets which can provide a high-speed communication link with the operators. The connection of the operator to such systems is instead restricted a gateway vehicle, such as an underwater glider, which occasionally surface for a limited time and quickly relay short messages received acoustically from the submerged network nodes, and receive command and control commands which will subsequently be transmitted via the acoustic channel to the other nodes. The latencies using such a gateway vehicle on the continental shelf will typically be of order 10-30 minutes.

The drastically reduced channel capacity of the undersea systems obviously has a dramatic operational effect. Thus, a typical acoustic sensing system will generate data at a rate of order Mbyte/second, which the acoustic communication capacity will be totally inadequate for transmitting. Therefore, in contrast to the air and land-based equivalents, the data processing cannot be performed centrally but must be largely distributed to the individual nodes. Similarly, real-time 'tethered' control of the underwater assets is made impossible by the latencies imposed by the use of occasionally surfacing *gateway* nodes. Consequently, real-time command and control decisions must be performed locally on the nodes, in turn requiring that not only the data processing, but also the analysis and interpretation - traditionally performed by human operators - must be performed locally on the nodes. This requires fully *integrated sensing, modeling and control*, a much higher

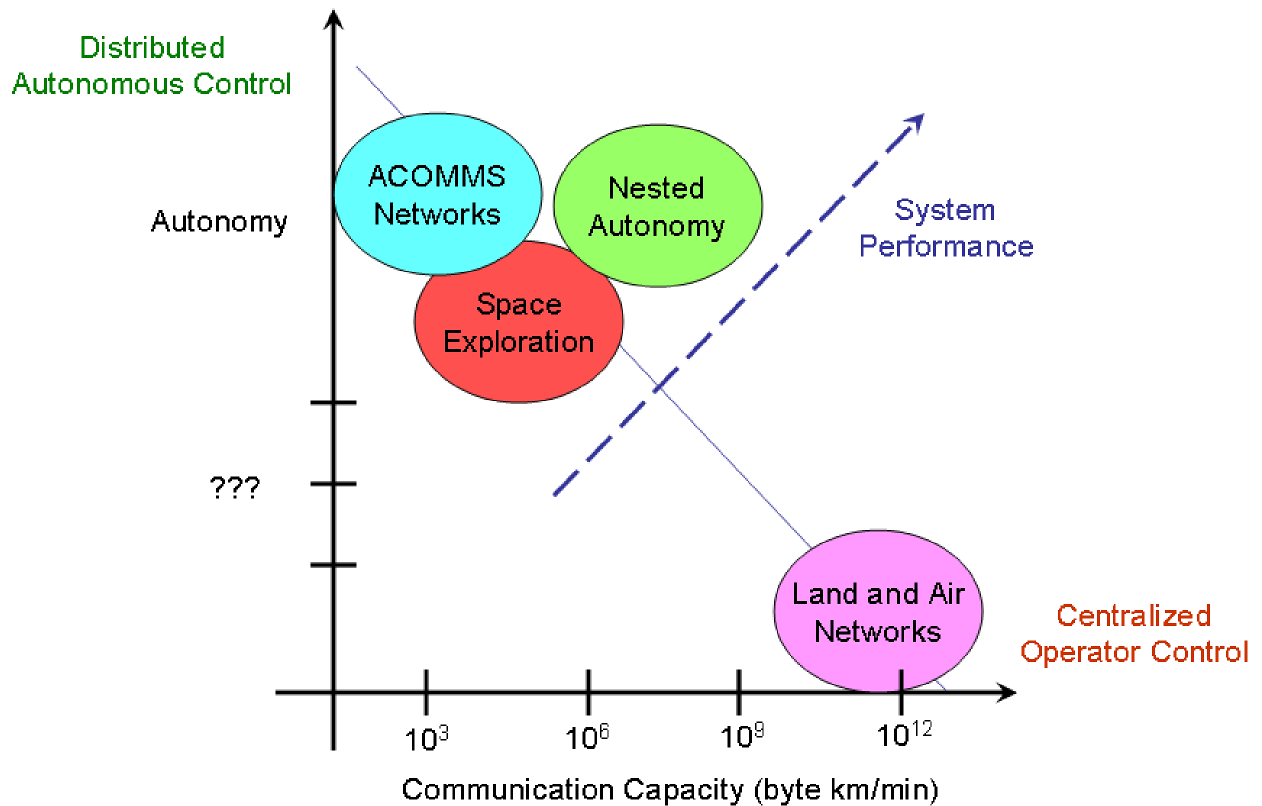

Figure 1: Performance trade-off between communication channel capacity and autonomy for net-centric sensing and observation systems

level of *Intelligent Autonomy* than required in most current applications of autonomous underwater vehicles, where the data collection and the control have been handled independently.

On the other hand, the higher degree of autonomy provides the opportunity of significantly enhancing the overall system performance by enabling the *adaptive* control of the mobile nodes to take optimal advantage of the local environmental and tactical situation. This performance trade-off between the network communication capacity and the autonomy is illustrated in Fig. 1. As illustrated schematically, the increased use of in-node intelligence or autonomy may compensate for the reduction in performance associated with the limited undersea communication channel capacity and latency. Note that the horizontal axis, representing the communication channel capacity, is labeled, with ten orders of magnitude separating the air-and land-based networks on one side and the underwater systems on the other. The vertical axis, however, is not quantified. The reason is that very little data - if any - is available in regards to quantifying the performance improvement associated with increased autonomy, and this is in fact one of the most important scientific issues associated with the development of the new distributed sensing and surveillance paradigm.

For ocean monitoring and observation systems, the main mission objective for the network is the detection, classification, and tracking of episodic, often unpredictable - events. Such events include chemical plumes from undersea volcanoes or man-made systems, biological phenomena such as algae blooms. Another important application of undersea sensing systems is the detection and tracking marine mammals and man-made sources of sound in the presence of and ambient noise.

Without the possibility of transmitting large amounts of data back to the operators, the on-board autonomy for most such applications must be capable of fully autonomously complete the mission objective of adequately sampling and characterizing the event, without any human intervention and assistance.

In addition to autonomously *adapting* to the environmental and tactical situation, the individual nodes may take advantage of *collaboration* with other nodes, again without requiring the human operator in the loop. Thus, a *cluster* of network nodes within - at least occasional - acoustic communication range with each other may fuse it's own data collected for the event with those obtained by and broadcast by other network nodes in the vicinity. For example, two AUVs with acoustic arrays may each track a marine mammal and collaboratively create an accurate tracking solution by triangulation [1]. For this purpose the nodes will collect relevant, intermittent information about the event, such as the current bearing to an acoustic source, in a *Contact Report*, which is broadcast to the rest of the network throughout the prosecution of the event. Once the mission objectives are achieved or the autonomy decides that no more information about the event can be extracted from the data, it will broadcast an *Event Report* containing a compressed set of data characterizing the event. In addition to be fused with data from the rest of the network by the field control operators, the *Even Report* may be used by other nodes to initiate a prosecution of the event. Again such a *hand-off* may be performed fully autonomously without any intervention by the remote operator, and therefore not exposed to the communication latency. Such *Collaborative* processing and control provides yet another level of autonomy - *Cluster Autonomy* - with the potential of enhancing overall system performance. The local node autonomy and the cluster autonomy are to two principal levels of autonomy in the *Nested Autonomy* paradigm. The nesting is completed by a lower-level autonomy responsible for the actuator control and navigation on each autonomous vehicle, interfaced to the higher-level sensor-adaptive control through the "backseat-driver", and any field level autonomy adapting the field to the larger scale environmental and tactical picture.

Under support of several ONR programs over the last decade MIT has been developing a *Nested, Autonomous Communication, Command and Control* (`NAC`^{3}) architecture with fully integrated acoustic sensing, modeling and control within each autonomous underwater vehicle, clusters of assets, and the entire network. The command and control architecture is highly portable and adaptable to the available acoustic communication infrastructure. Thus, the current architecture applies the Compact and Control Language (CCL) developed by Woods Hole Oceanographic institution for their Micromodem, based on 32 byte messages, with a current capacity of one point-to-point message every 20 seconds, approximately.

Implemented using `MOOS-IvP` , the MIT `NAC`^{3} enables fully autonomous adaptation of the mobile network nodes to the environmental and tactical picture, collaborative target event by multiple platforms, and safe and efficient operation in uncharted environments, without the need for reprogramming. Thus, once deployed the entire network is operated using only the CCL messages for communication between nodes and field control and for changing mission objectives and platform states. For example, the `NAC`^{3} allows the operators to effectively control the autonomous mission behaviors to be launched by a simple set of *Deploy* and *Prosecute* commands. In the `MOOS-IvP` framework, the state transitions and associated vehicle behaviors initiated by the CCL commands are controlled using high-level 'behavior algebra' in one, objective-unique mission file.

In addition to a series of field experiments the development of the new Nested Autonomy `NAC`^{3} for undersea acoustic sensing is supported by a comprehensive simulation environment with high-fidelity hydrodynamic and acoustic simulation for AUVs with hydrophone arrays, which allows

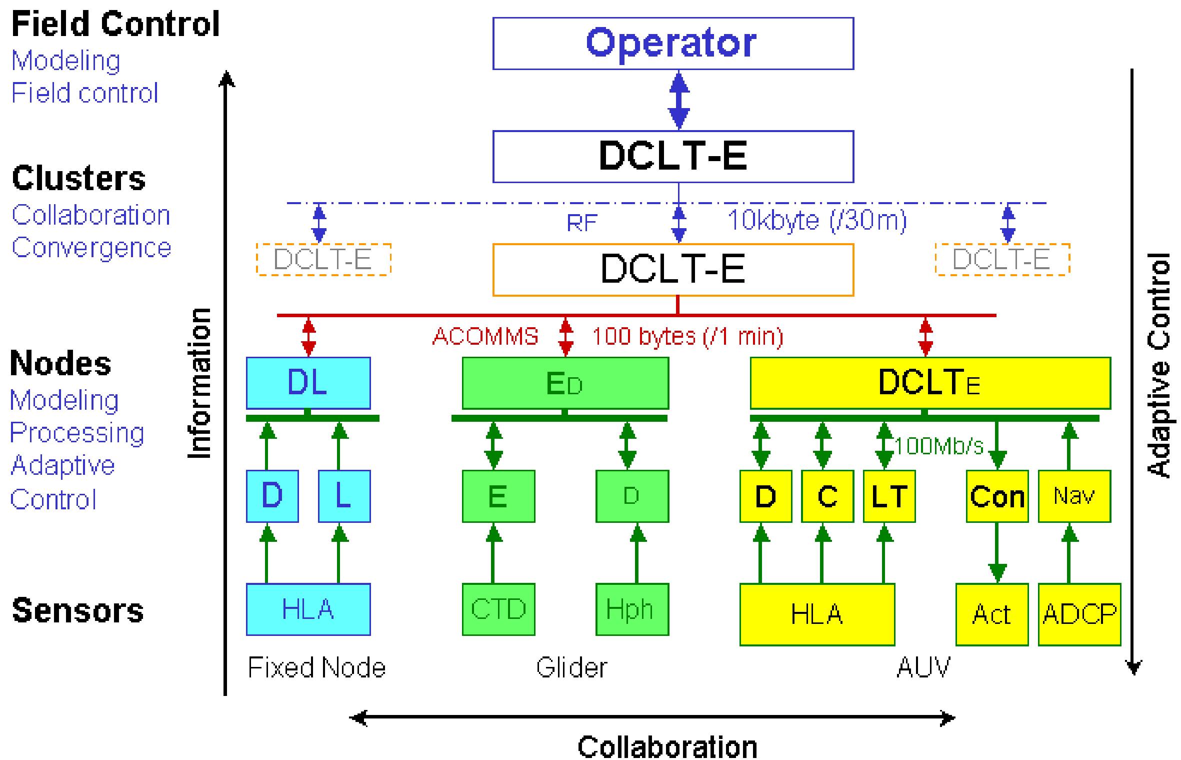

Figure 2: Nested Autonomy. The field operator is communicating with clusters of autonomous nodes through gateway assets occasionally surfacing for transmitting target reports and receiving network commands, e.g. through satellite radio communication, yielding high bandwidth, but latency of 10-30 minutes.. The nodes in the cluster communicate acoustically at low bandwidth but low latency. The Node and Cluster Autonomy are designed accordingly.

comprehensive testing of the behavior sequence, and the overall system performance. The command and control architecture has initially been developed for underwater acoustic sensing and mapping, but is currently being adapted adapted to much more general oceanographic environmental monitoring and observation systems, e.g. for command and control of mobile assets in the NSF ORION ocean observatory framework.

3 Nested Autonomy for Ocean Observation Systems

As illustrated in Fig. 1, the objective of the *Nested Autonomy* is to enhance the system performance beyond what is achievable by local autonomy only. This possibility is associated with the inherent layering of the communication infrastructure, with the underwater network connectivity being provided by low-bandwidth acoustic communication (ACOMMS), and the above-surface networking being provided by high-bandwidth, but latent, radio frequency (RF) communication through a

regularly surfacing gateway node. On-board each node, the computer bus and ethernet networking provides very high bandwidth communication between the sensing, modeling and control processes.

Figure 2 shows this layered structure schematically. The *vertical* connectivity allows information to pass from sensors to the on-board processing and on to the control, or from each node to the field operator, and thus forms the basis for the autonomous *Adaptive Control* which is a key to the capability compensating for the smaller sensor apertures of the distributed nodes. Similarly, the *horizontal* connectivity forms the basis for *Collaboration* between sensors on a node (sensor fusion) or between nodes (collaborative processing and control).

The three layers of horizontal communication have vastly different bandwidths, ranging from 100 byte/min for the inter-node ACOMMS to 100 Mbyte/sec for the on-board systems. Equally important, the layers of the vertical connectivity differ significantly in latency and intermittency, ranging from virtually instantaneous connectivity of the on-board sensors and control processes to latencies of 10-30 minutes for information flowing to and from the field control operators. This, in turn, has critical implication to the time scales of the adaptivity and collaborative sensing and control. Thus, adaptive control of the network assets with the operator in-the-loop is at best possible on hourly to daily basis, allowing the field operator to make tactical deployment decisions for the network assets based on e.g. environmental forecasts and reports of interfering shipping distributions, etc. Shorter time scale adaptivity, such as autonomously reacting to episodic environmental events or a node tracking a marine mammal acoustically must clearly be performed without operator intervention. On the other hand, the operator can still play a role in cuing forward assets in the path of the dynamic phenomenon, using the limited communication capacity, taking advantage of his own operational experience and intuition. Therefore, just as well as a centralized control paradigm is infeasible for such systems, it is unlikely that a concept of operations based entirely on nodal autonomy be optimal.Instead, some combination will likely be optimal, but in view of the severe latency of the *vertical* communication channels, the *Nested Autonomy* CONOPS described in the following is heavily tilted towards autonomy, as illustrated in Fig. 1.

Another important effect of the layered, hybrid communication infrastructure is the significant incentive it provides for *clustering* assets. Thus, for example, it is more efficient to incorporate all relevant sensor types on all sensing nodes, rather than distributing the various sensing roles among different nodes, which would require communication through the slow acoustic communication channel. Similarly, the collaboration among nodes benefits significantly by deploying the fixed and mobile nodes in clusters, within each of whichthere is high probability of communication connectivity, eliminating the negative effect of the latency of the RF communication layer for communication between nodes. Thus, if the underwater assets are clustered in groups with - at least occasional - inter-node communication connectivity, such that the individual nodes may pick up event reports from other nodes, they can fuse this information with their own sensor processing to achieve an improved event tracking solution, for example. On the other hand, the environmentally induced intermittency of the underwater acoustic communication channel is well established, and it is therefore necessary that the cluster CONOPS be designed in such a manner that they are not crucially dependent on 'hand-shake' communication between the nodes. In other words, the CONOPS must be based on the assumption that each individual node may take advantage of any target information it receives, but capable of completing the mission objective in the total absence of communication connectivity. This requirement is a key to the operational robustness and inherently makes the CONOPS *autonomy-centric* in contrast to the *net-centric* CONOPS of the land- and air-based equivalents. The CONOPS of the different layers in the *Nested Autonomy*

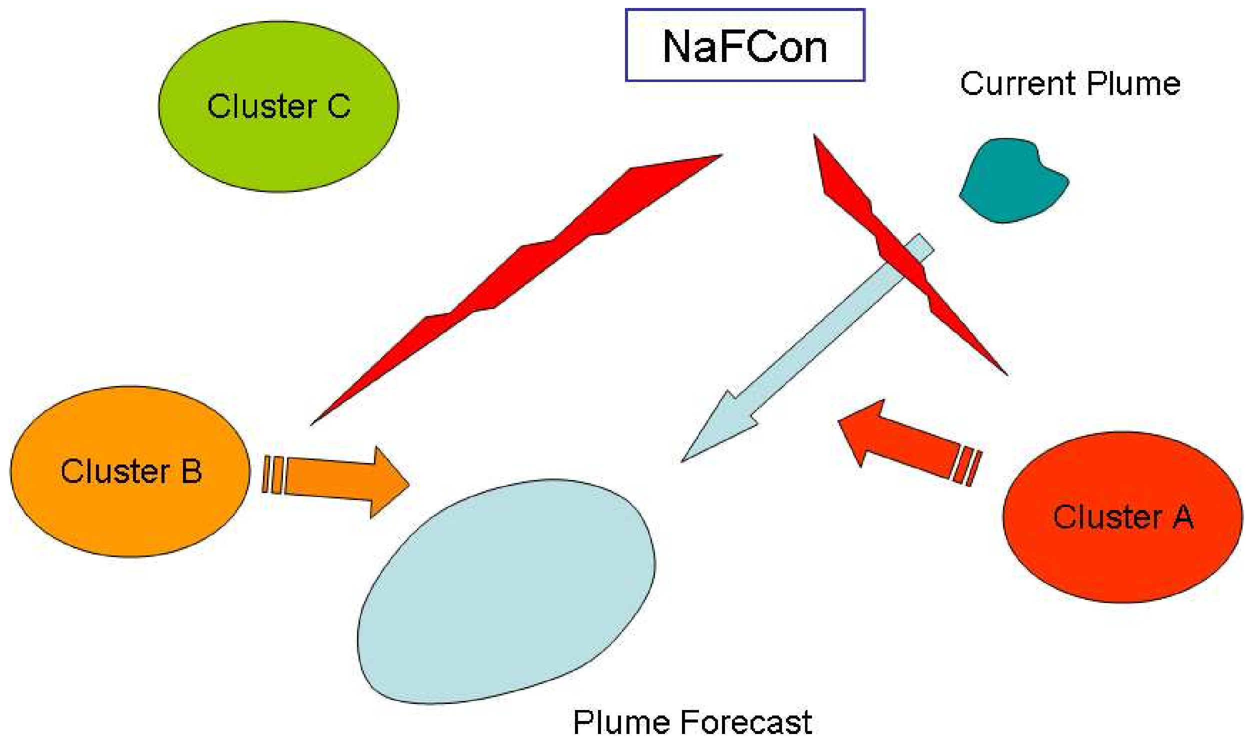

Figure 3: Field-level concept of operations. NaFCon field control is dispatching clusters to autonomously prosecute a chemical plume with a forecast path and expansion. Cluster A i instructed to initiate the prosecute immediately since it is closest to the projected path. The 'downstream' Cluster B is alerted to be ready for action, while field control decides not to activate cluster C, which is not in the path of the plume.

paradigm are described in the following section.

4 Concept of Operations

4.1 Field-level CONOPS

The layered and clustered NAC^3 illustrated in Fig. 2 naturally leads to a nested or layered concept of operations (CONOPS), which, as mentioned earlier, provides some optimal mixture of distributed autonomy and centralized control. Figure 3 shows a possible field-level concept of operations (CONOPS) for an oceanographic observation system for capturing an episodic event, such as a chemical plume released by an undersea volcanic event. The target area is populated by a number of clusters, each with a number of mobile assets such as AUVs and gliders.

One of the adaptive responsibilities of the Network and Field Control *NaFCon* is to deploy the finite number of clusters in a pattern which is optimal for the current environmental situation and with the highest probability for capturing the episodic event of interest. The time scales for deployment and re-deployment are are inherently long, at least of order hours, and more likely days, and is therefore highly dependent of reliable environmental and situational forecasts, often requiring a significant modeling and data assimilation infrastructure.

Once deployed, it is assumed that each cluster is capable of autonomously *Detecting, Classifying,*

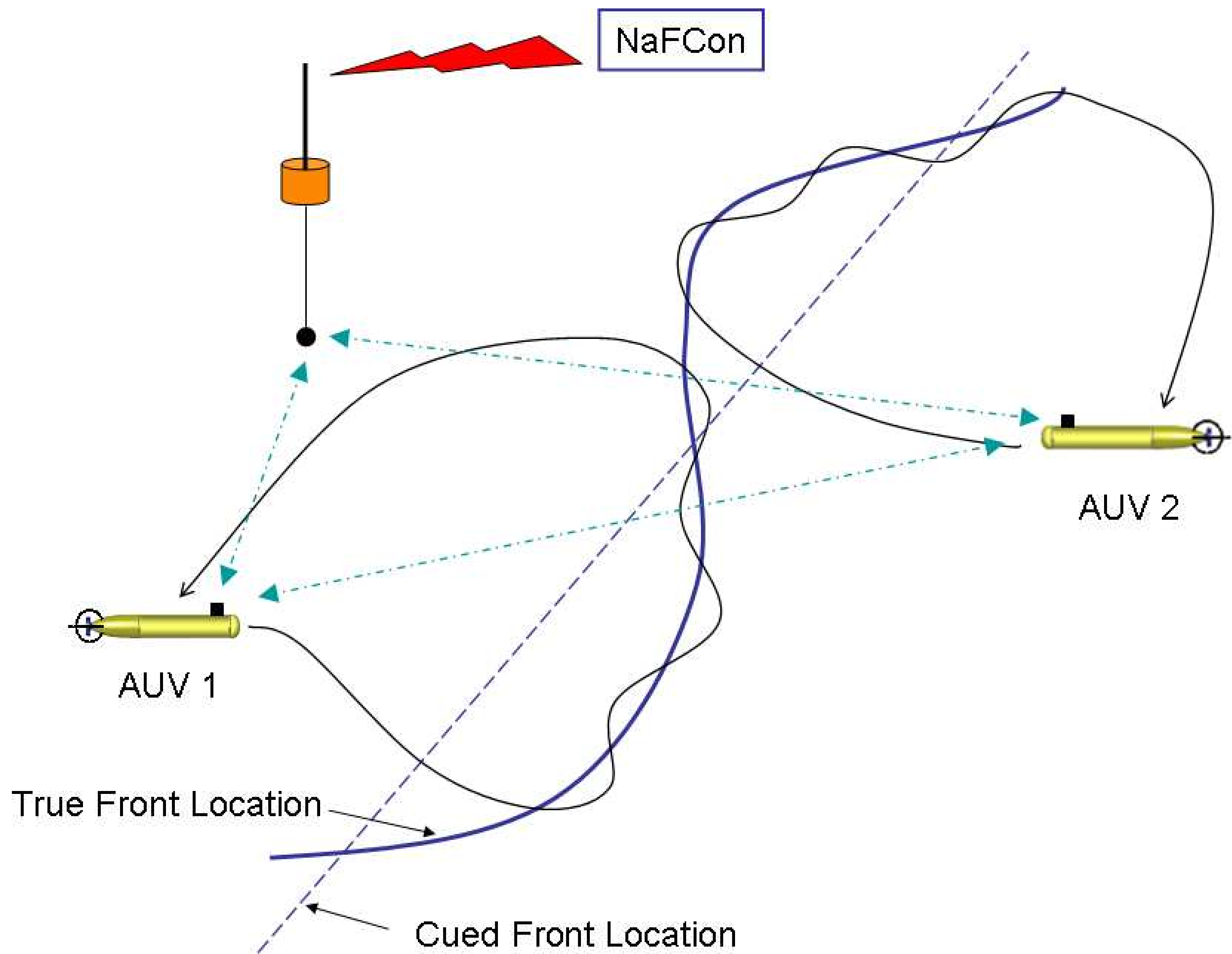

Figure 4: Concept of operations for cluster of AUVs with oceanographic sensors and gateway buoys for communication with NaFCon. The position and heading of a front is cued to the vehicles via the gateway buoy and they initiate a Prosecute mission, autonomously detecting the front and subsequently mapping it by zig-zagging across the frontal boundary. the two AUVs coordinate the survey to increase coverage and avoid overlap..

Localizing and Tracking the episodic event of interest. The event *Prosecution* may be either cued by *NaFCon* through a surface communication gateway, or performed fully autonomously. Once a tracking solution and the nature of the source determined, the result of the prosecution will be reported back to *NaFCon* in the form of an *Event Report*. The human operators may then cue other clusters in the projected path of the event with whatever information is available and packagable into the Compact Control Language (CCL) format suitable for transmission through the network.

The final crucial role of *NaFCon* is the fusing of the *Event Reports* from the various clusters in the path of the event, gradually building up a more and more complete event track and description.

4.2 Cluster-level CONOPS

Depending on the available assets, you can envision a wide spectrum of cluster compositions, including gliders and propelled AUVs with chemical, biological and acoustic sensors.

Figure 4 schematically shows how such cluster assets may be applied in response to an event cuing message from *NaFCon*. The message identifies a front with a location and heading indicated by the dashed line. After the message is received by a surface gateway buoy, it is broadcast using the acoustic modems. Nearby nodes, e.g. a dormant, drifting or bottomed AUV, which picks up the message, will initiate a *Prosecute* behavior sequence, in this case the detection and subsequent mapping and tracking of a frontal boundary. Depending on the level of autonomy authorized by *NaFCon*, the AUV may decide not to pursue the target event if there is little probability it will come within detection range.

If two or more nodes are prosecuting the event, each node may fuse the event information from the other nodes to produce a more accurate event characterization, and to optimize the coverage or resolution. Thus, in Fig. 4 the two vehicles coordinate their survey in order not to overlap and increase coverage. Another example of collaborative control is a node which did not receive the original prosecute command, but which, following the receipt of an *Event Report* from a prosecuting node will determine whether the target event is likely to come within range, and then autonomously initiate a prosecute sequence. All *Event Reports* generated by the prosecuting nodes are then collected by the communication gateway and transmitted back to *NaFCon* via RF communication.

4.3 Node-level CONOPS

A suite of node-level CONOPS have been developed for both single node and collaborative detection and tracking of a variety of episodic events, such as the adaptive mapping of a front or a thermocline, and for tracking an acoustic source, such as a marine mammal or a man-made source of sound. Most of these operational CONOPS have been demonstrated in a sequence of field deployments, the most recent being the Oct. 2008 GLINT'08 experiment at Pianosa Island, Italy, carried out in collaboration with NURC. As an example, Fig. 11.2 shows the core adaptive *Prosecute* sequence developed for the BF21 AUVs equipped with hydrophone arrays, applied for tracking an acoustic source. The numbers identifies the sequence of events involved in acoustic source prosecution sequence.

1. The AUV in this case is deployed in a low-power, continuous, hexagonal loiter pattern, awaiting commands for further action from *NaFCon*

2. A *Prosecute* message is received from *NaFCon*, with an estimated source location, speed and heading.

3. In response to the cuing message the AUV initiates a *Search* sequence, which in its simplest form closes range to the estimated track of the acoustic source.

4. Once the source signal is detected, the AUV autonomously initiates a turn maneuver which allows it to break the left-right ambiguity inherent to linear arrays.

5. Once the left-right ambiguity is resolved by the signal processing algorithm in response to the turn maneuver, the AUV will start broadcasting *Contact reports* containing the absolute bearing to the source, for use by any collaborating nodes and *NaFCon*, The AUV will then initiate the *Tracking* state, where it maneuvers to optimize the on-board *Bearing Tracker*, notably by maintaining the source at broadside, while at the same time maneuvering to

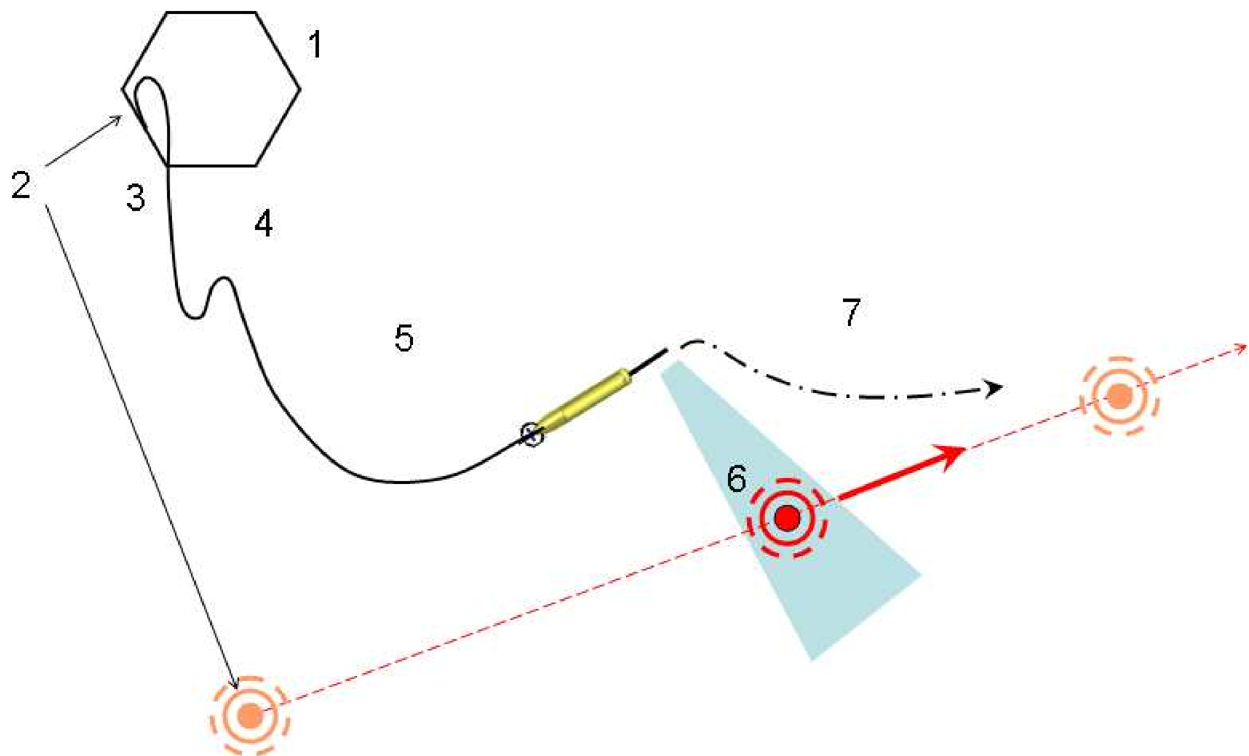

Figure 5: Concept of operations for single AUV with acoustic sensors autonomously Detecting, Localizing and Tracking a moving acoustic source. 1. AUV deployed in hex loiter. 2. AUV issued PROSECUTE command by *NaFCon*. 3. AUV entering Search state, closing range fore improving detection. 4.After detecting the source signal, the AUV enters maneuver for breaking L/R ambiguity. 5. Tracking state with AUV closing range but retaining target close to broadside. 6. Track solution achieved. 7. Classification state for source spectral characterization.

optimally produce a *Source Track*, i.e. a time history of the source motion, including speed and heading. In this *Tracking* state. the AUV may also fuse its own tracking analysis with *Contact reports* from other nodes in the cluster, allowing for more accurate track solution.

6. Once a satisfactory *Source Track* has been determined, the AUV will broadcast a *Track Report* for the for use by *NaFCon* and other nodes for fusion with equivalent information from other sensing assets to produce a more complete and accurate *Global* situational description..

The node CONOPS described here for the tracking of an acoustic source are directly mapped onto any other episodic event in the ocean environment, e.g. the tracking of a plume, where the collaborative, adaptive cluster autonomy is even more important by providing simultaneously the resolution and coverage required for accurately localizing, classifying and tracking the event. Thus, the network must first Detect and localize the plume, and then adaptively track its boundaries, which obviously requires the vehicles to collaborate to cover the expanding spatial extent of the plume.

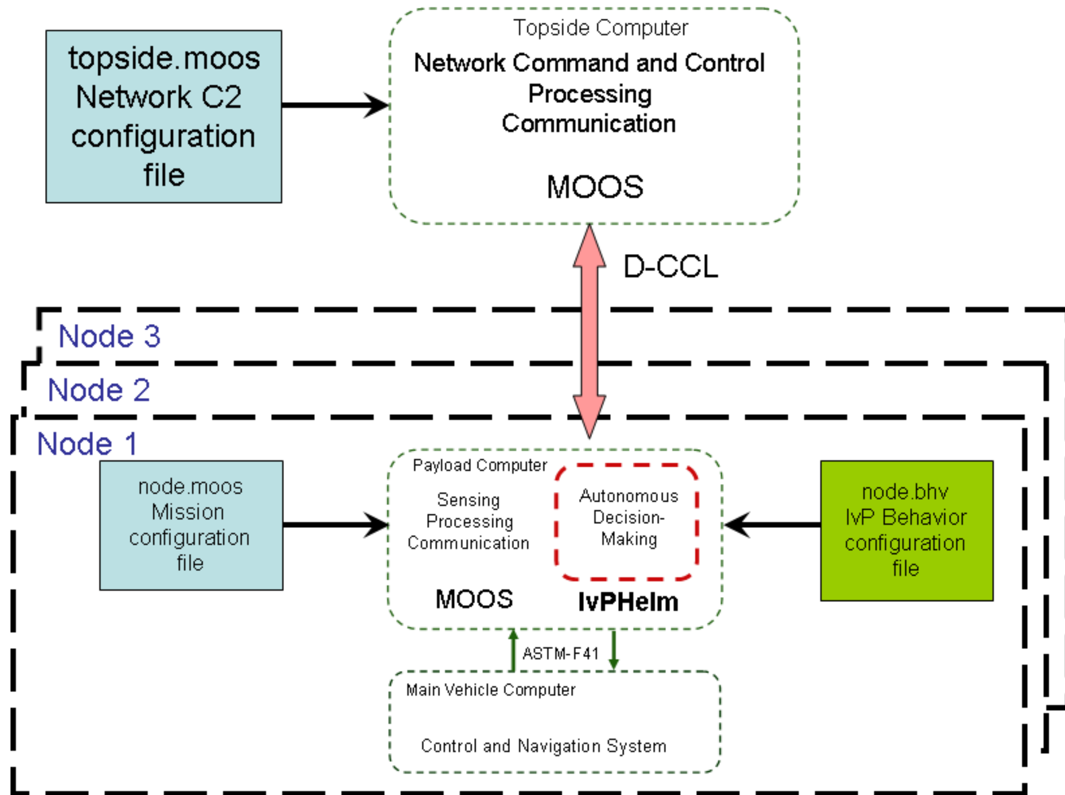

Figure 6: Network architecture for MIT prototype of nested autonomy cluster. An arbitrary number of mobile and fixed nodes, all operated using the `MOOS-IvP` autonomy system architecture and backseat-driver paradigm, is connected via an underwater acoustic communication network, using a dynamically configured, generalized version of the CCL message protocol. The topside Command and Control is implemented using the same `MOOS-IvP` software suite, including the acoustic communication stack, situational displays, and GUIs for issuing commands to the network.

5 Cluster Autonomy System

MIT LAMS has over the last few years developed a stable prototype of a nested autonomy cluster prototype, with the architecture shown schematically in Fig. 6.

An arbitrary number of mobile and fixed nodes, all operated using the `MOOS-IvP` autonomy system architecture and backseat-driver paradigm. Each node autonomy system is configured for its dedicated role in the network cluster through the MOOS configuration file `node.moos` and the `IvP` configuration file `node.bhv`. The process and behavior blocks of these configuration files are described in detail in the following chapters.

The underwater nodes in the network are connected via an underwater acoustic communication network, using a dynamically configured, generalized `D-CCL` version of the CCL message protocol developed for the WHOI MicroModem, for which a dedicated set of MOOS.processes have been developed and extensively tested.

The topside Command and Control is implemented as a MOOS community, including the acoustic communication stack. Also, the topside community incorporates a variety of situational displays, and GUIs for issuing *Deploy* and *Prosecute* commands to the network nodes .

6 Platform Autonomy System

The core of the nested autonomy paradigm is the autonomous, integrated sensing, modeling and control command and control framework on each individual platform. In combination with the collaborative cluster autonomy , the integrated node autonomy enables the adaptivity which may compensate for the reduced physical sensor apertures of the unmanned underwater vehicles.

6.1 The `MOOS-IvP` Autonomy Architecture

The *Nested Autonomy* paradigm for distributed undersea sensing inherently involves reaction to situations and events that are deterministically unpredictable. Thus, the autonomy architecture cannot be based on the availability of a world model that can form the basis for the autonomous decision making. Instead, it requires a capability of fully autonomously adapting to the environmental and tactical situation associated with the phenomenon is is intended to measure. As such it forms a clear example of the type of robotic system for which the Interval Programming (IvP) model for multi-objective behavior coordination was intended and developed. Thus, for example, an underwater vehicle tasked with detecting and tracking an acoustic source is faced with several, often conflicting objectives. It will likely have been assigned a station point, from which it should not move too far, while at the same time having to get close to the source to develop a reliable tracking solution. Also, depending on its sensing capability it may have a preferred heading for achieving tracking resolution. Also, if other vehicles in the vicinity is already tracking the target event, it may not be desirable for it to pursue the same source aggressively, but instead preserve power for future sensing tasks. `MOOS-IvP` provides exactly the flexibility and inherent multi-objective capability for implementing such high-level autonomy with adaptive and collaborative capabilities.

The MOOS community used for the platforms in the MIT-LAMS implementation of the *Nested Autonomy* concept of operations is shown in Fig. 7. As in all `MOOS` communities, the `MOOSDB` process is the core of the `MOOS` architecture and handles all communication between the processes using a publish-and-subscribe architecture. The various `MOOS` processes include all necessary control functions as well as sensing and processing modules, with the `MOOSDB` providing the unified interface standard that enables the fully autonomous integration of sensing, modeling and processing, and control. `MOOS` ensures a process executes its “Iterate” method at a specified frequency and handles new mail on each iteration in a publish and subscribe manner. The `IvP Helm` runs as the MOOS process `pHelmIvP`. A detailed description of the role and configuration of the platform community is giving in the following.

6.2 The Back-Seat Driver Paradigm

To allow the `MOOS-IvP` network control to be applied on a variety of fixed and moving nodes with different control software, a “back-seat driver” paradigm was adopted and integrated with the `MOOS-IvP` control software infrastructure. The idea is that all high-level control, including the adaptation to measured and estimated parameters, is performed in a payload computer (`PLC`) running `MOOS-IvP` . The `PLC` will also handle all communication with the network, either through a radio link while surfaced, or an acoustic modem when submerged. All lower level control, and basic navigation and platform safety tasks are handled by the native vehicle control software running in the main vehicle computer (`MVC`), i.e. Huxley in the case of the Bluefin AUVs. The communication between the `PLC` and the `MVC` is performed over an ethernet socket,operated by a `MOOS` process

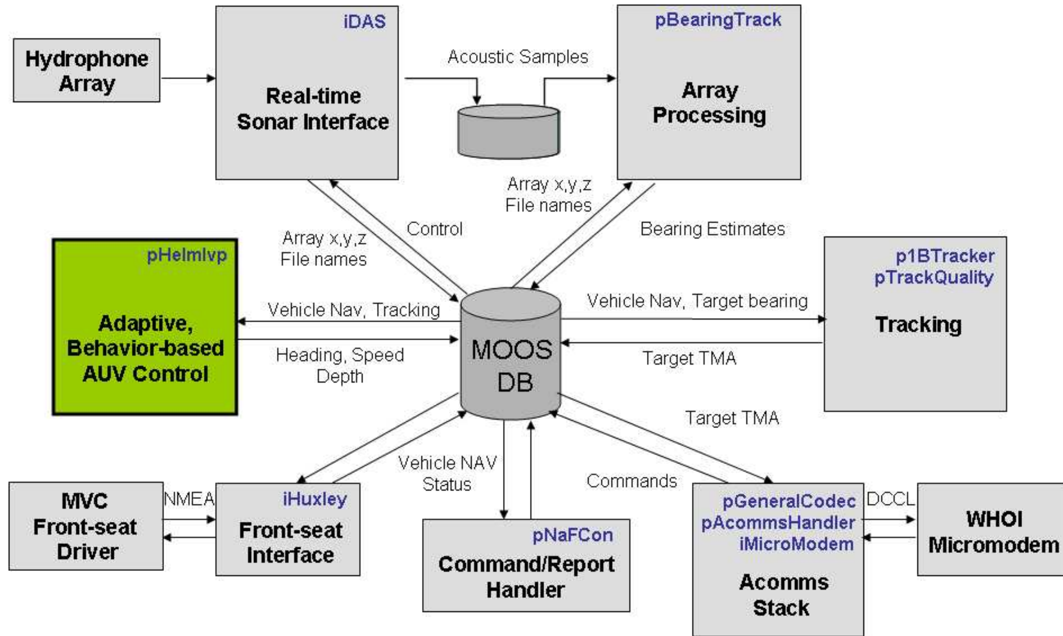

Figure 7: Unicorn MOOS Community with pHelmIvPbehavior-based autonomy module highlighted in green. All communication between processes is handled through the MOOSDB, using a mail 'publish' and 'subscribe' convention

pHuxley running on the PLC, as illustrated in Fig. 8. The commands passed to the MVC are simply continuous updates of desired heading, speed and depth, which the MVC then translates to desired rudder, thrust and elevator signals to the tail cone. The MVC will return current navigation and state data, which pHuxley will then publish in the MOOSDB . In the current BF'21 vehicle configuration the PLC incorporates a Linux-based CPU for general-purpose computational tasks, such as the multi-objective helm, as well as a specialized 8 GFlop vector processor to support heavy signal processing loads such as spectrum analysis and broad-band beamforming. In this architecture, the PLC appears to the MVC as a simple NMEA device, and vice versa. (NMEA is an industry standard for communication between marine devices.)

The basic safety tasks performed by the MVC include mission abort due to bottom altitude limit violations, and lack of commands from the PLC in a certain specified time, as well as an overall mission timeout. Higher level safety tasks such as exceeding the specified operating radius, and individual behavior timeouts or failures, are handled by the PLC.

6.3 Communication Connectivity

Although the concept of operation described above is *autonomy centric*, the communication infrastructure provides a crucial backbone of a distributed sensing and observation network. Each network node, including the sensor platforms and the communication gateways, are equipped with acoustic modems, the WHOI Micromodems. These modems implement an adaptive decision feedback equalizer with integrated Doppler and error-correction to afford 80-bps frequency hopped-FSK mode communications, and in newer implementations 5400-bps PSK mode communication. The modem provides the user with the tools necessary to create a simple time-division, multiple-access

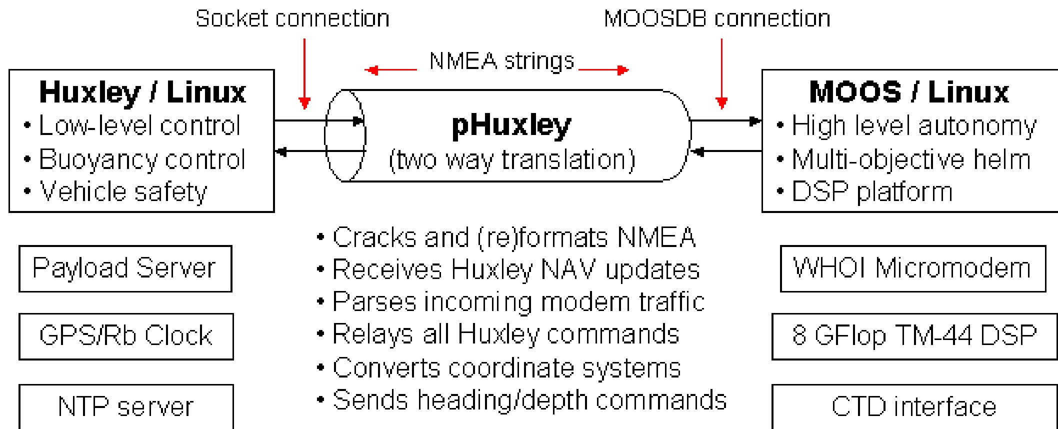

Figure 8: Schematic representation of the 'back-seat driver' paradigm, where higher level adaptive control and network communication is handled by MOOS-IvP on the payload computer, while lower level control, navigation and vehicle safety is handled by the main vehicle computer using the native control software.

(TDMA) network for master-slave polled systems, or a random-access peer-to-peer network. Each communication transaction includes a short network packet called a cycle-initialization. This specifies the source, destination and data rate of the packet to follow. A Compact Control Language (CCL) has been developed developed by Woods Hole Oceanographic Institution. In the prototype network the CCL defines a set of 32 byte messages, which are used exclusively for issuing commands to the nodes through the acoustic modem network, and for carrying sensor messages to other nodes and back to field control, including sensor status, contact and event tracking reports. Using a centralized polling scheme this communication infrastructure provides for 32 byte messages for FSK, and 2048 byte messages for PSK, being transmitted to or from a node once every minute, approximately, depending on the number of nodes in the cluster. The MOOS-IvP AC^3 has been designed within this tight constraint. However, it is entirely flexible and may be straightforwardly adapted to future, higher bandwidth communication infrastructures.

The interfacing of the Micromodems with the MOOS-IvP command and control infrastructure is rather complex to allow the flexibility required for the wide spectrum of mission types and objectives anticipated for an operational surveillance network.Two primary types of messages are used by the *Network and Field Control* (*NaFCon*) to provide centralized operator control; The *Deploy* message instructs the AUV as to where it should operate for efficient use of the field, The message includes latitude, longitude, and depth of deployment, and includes an operation radius within which it may perform autonomous., adaptive and collaboration search and mapping of episodic events. As described in the following section, the *Deploy* message can specify a suite of deployment mission types, depending on the mission objectives. A cluster or individual nodes may be cued by *NaFCon* to autonomously prosecute a target event detected and tracked by another cluster using a *Prosecute* message. Following the receipt of the *Prosecute* message, depending on the prescribed level of autonomy, the vehicle may either immediately initiate a search pattern for detecting the target event, or it may autonomously determine that the event is predicted to get within detection range before initiating the prosecution, and otherwise remain in the deployment pattern until being cued by another *Prosecute Command* from *NaFCon*, or an *Event Report* from another node in the same cluster indicates anh interception opportunity.

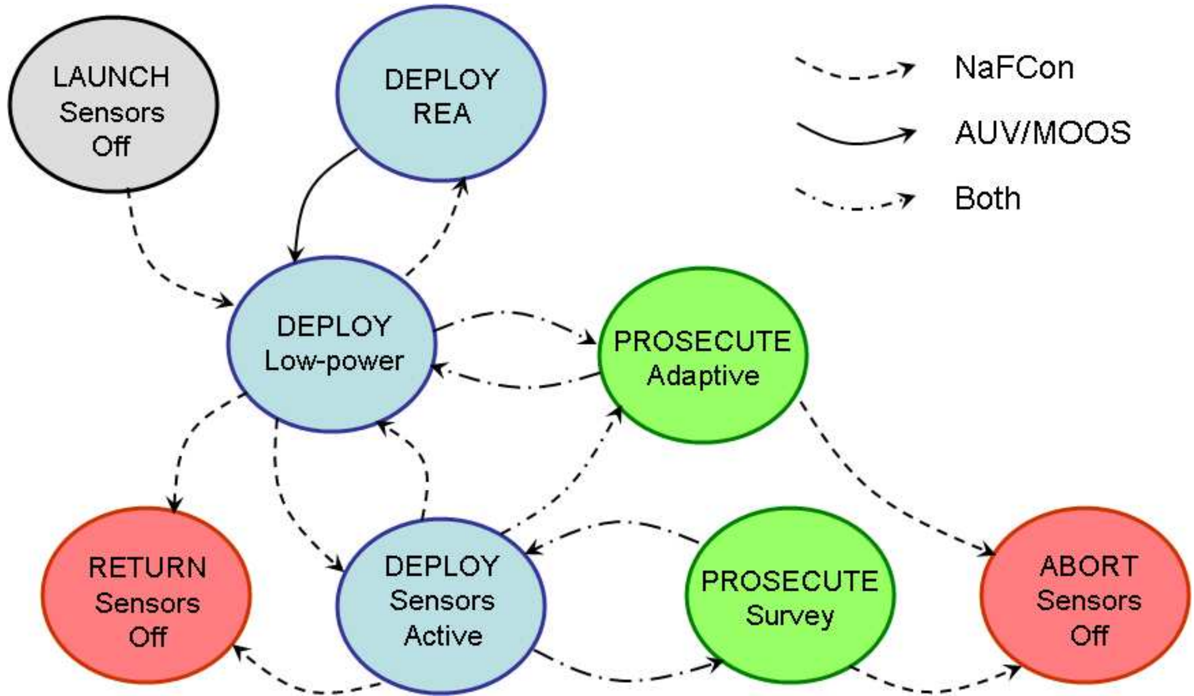

Figure 9: MIT BF21 vehicle states and sub-states, and state transitions for undersea acoustic source tracking. Each state is transitioned into with the receipt of a NAFCON message of the same name, with the sub-state determined by the mission type. The commands may be issued by field control or locally by the `MOOS-IvP` autonomy. Each state is comprised of one or more vehicle behaviors responsible for achieving the objectives of that state.

Similarly, the messages sent back to *NaFCon* in the acoustic source tracking prototype include a status report, including navigation data, battery state, etc., a contact report including a target bearing, and a track report providing a complete tracking solution, obviously constrained to the event information that can be packed into the CCL message.

6.4 Autonomy States and State-Transitions

The vehicle mission specification in a `pHelmIvP`autonomy system is comprised of a set of *states* for the nodes, *behaviors* operating in those states, and episodic, environmental or tactical *events* to which the node must respond. Such events include *Event Reports* issued by a collaborating fixed or mobile node, or target information contained in a CCL *Prosecute Command* from *NaFCon*.

The states that have been defined for vehicles in an acoustic source tracking network are shown in Fig. 9. Each nodal state may have several distinct *sub-states*, each characterized by a set of *behaviors*, which based on the available navigation data, sensor input, and on-board processing results, generate objective functions for speed, depth and heading. The `pHelmIvP`process uses its multi-objective optimization algorithm to convert these objective functions into the `MOOS` variables `DESIRED_SPEED` , `DESIRED_HEADING` , and `DESIRED_DEPTH` , which are then passed on to the lower-level 'front-seat' driver control by the `pHuxley` `MOOS` process.

The current behavior suite supports a total of 8 states, as shown Fig. 9, 3 of which are so-called *Deploy* states, and two are *Prosecute* states:

Surface State: The system initially begins in this state waiting for a *Deploy* message. The vehicle also autonomously returns to the *Surface* state message when it reaches either its return or abort waypoints.
Deploy State: Entered on receipt of the *Deploy Command*, either received from *NaFCon*, or issued immediately internally once `MOOS` is handed over the mission control, depending on the mission file settings. The vehicle transits to its deployment location and depending on the mission type enters one of the *Deploy States*, shown schematically in Fig. 9.

- *Low-power Deploy (middle, left):* This is the base-line, power-saving *Deploy* mode . For a vehicle with active buoyancy control this state will deploy the vehicle in a zero-speed drift at constant depth, while for a regular AUV it is typically defined as a low-speed hexagonal loiter pattern, waiting for a re-deployment or prosecute message.

- *Active Sensing Deploy (lower left):* Here the vehicle enters a hexagonal loiter at a speed and depth that is optimal for the sensing and processing, with the acquisition and on-board signal processing active.

- *REA deploy (upper, center):* This *Rapid Environmental Assessment* state is currently commanding the vehicle to perform a round-trip track-line at a certain heading while performing an adaptive vertical yo-yo pattern for mapping the thermocline. As for all other states, the actual behaviors of this state are defined in the mission file and easily changed.

Prosecute State: In the *Prosecute States* the vehicle will actively perform an event search, localization ant tracking sequence with various levels of autonomous adaptive behavior. The *Prosecute* state is terminated by a *Deploy Command* issued autonomously if the allowed duration or operation radius is exceeded, or the tracking solution is deemed complete by the on-board processing. The *Deploy* location will be the current position or the assigned station point, depending on the mission configuration. Alternatively, *NaFCon* may issue the *Deploy* command.

- *Non-adaptive Prosecute (lower, center):* This mission is a classical, non-adaptive survey pattern, such as a lawn-mower or Zig-Zag pattern, for example, depending on the settings in the behavior configuration file. Upon receipt of the *Prosecute Command*, the vehicle enters its survey pattern with an average heading parallel to the cued target heading. In this non-adaptive mode the breaking of left-right ambiguity and track generation is made possible by the pre-determined turns, but no adaptive behaviors are enabled. This state has no sub-states, since it remains in the survey pattern until the termination by the one of the state manager processes, typically `pTrackQuality`.

- *Adaptive Prosecute (middle, right):* This is the core adaptive prosecute state, for the acoustic source tracking problem executing the mission sequence shown in Fig. 11.2. Depending on the level of autonomy authorized in the mission file, this state may be entered by a *NaFCon* - issued *Prosecute Command*, or it may be autonomously entered by the on-board control on a vehicle in the *Active Sensing Deploy* state, in response to an *Event Report* transmitted by another vehicle. The adaptive prosecute state has five sub-states for executing the nodal CONOPS:

  - *Search:* Currently this pre-detection state will command the vehicle on a heading that closes the range to the cued target track. If the vehicle gets within a predetermined

range of the estimated target position without detecting the source, it initiates a local loiter maneuver waiting for an acoustic detection.

– *Ambiguous:* this state is entered once the source is detected and commands the vehicle to perform a sequence of turn maneuvers aimed at breaking left-right ambiguity for a linear hydrophone array.

– *Tracking:* This is the core acoustic source tracking state, which is autonomously entered once the left-right ambiguity is resolved. The onboard processing will first stabilize the bearing tracker, after which it will initialize the tracker, and the `pHelmIvP`autonomy will adaptively keep the source bearing near broadside, while maintaining some attraction to the source. This behavior is continued until one of the following conditions are met:

    1. Contact reports for the same target received from other nodes, which will transition the vehicle to the collaborative sub-state.
    2. A stable single-bearing track solution is achieved
    3. The bearing rate decreases below a threshold percentage of the maximum bearing rate achieved during the current tracking operation.
    4. The deployment operation radius exceeded.
    5. The vehicle is getting withing a preset minimum distance from the mission operations boundary .

  Except for the first condition, the vehicle will, dependent on the configuration defined in the mission file, transition to the *Classify* prosecute sub-state or the default *Low-Power Deploy* state at the current position.

– *Collaborative Tracking:* In this sub-state the vehicle will fuse its own tracking solution with those obtained from neighbor nodes tracking the same acoustic source. Depending on the authorized level of autonomy, this state in addition to the data fusion may involve collaborative behaviors, optimizing the collaborative tracking performance.

– *Classifying:* This is the main classification sub-state of the acoustic source prosecution, entered once a stable tracking solution has been achieved. In the current implementation the vehicle is sampling the acoustic field over a selected depth interval using a depth-YoYo behavior, with a constant heading close to end-fire, to enable measurement of the vertical, angular spectrum of the acoustic field..

Abort State: The vehicle will transit toward its abort waypoint. When the waypoint is reached, a goto-surface message will be generated and the vehicle will return to the *Surface* state.
Return State: The vehicle will transit toward its return waypoint. When the waypoint is reached, a surfacing message is generated and the vehicle will enter the *Surface State*.

Each state is comprised of one or more vehicle behaviors responsible for achieving the objectives of that state. These behaviors are described later...

6.5 On-board, Real-time Signal Processing

A key to the autonomous, adaptive sampling of chemical, biological or acoustic fields in the ocean is an efficient on-board implementation of a data analysis package that allows for real-time feed-back to the platform control. MOOS-IvP provides a very effective infrastructure for achieving this due to its modular structure and well defined communication infrastructure. For example, for the sonar

payloads for the MIT AUVs, the principal processing module is the `pBearingTrack`. This generic beamformer and bearing tracker is applied to a wide suite of acoustic arrays operated on the AUVs. It comprises a conventional beamformer, a detection algorithm and a stabilized bearing tracker. A detailed description of these real-time acoustic processing algorithms is given elsewhere, [2], and only the functionality relevant to the autonomy shall be described here. `pBearingTrack` subscribes to a `MOOS` variable stating the availability of a new file with acoustic data. It reads the data file, performs a fourier transform and the beamformer generates a beam-time record (BTR) for the selected frequency band, which is subsequently published to the `MOOSDB` . A detection algorithm then determines the presence of a signal of interest, and sets a flag in the `MOOSDB` .. This flag is used in the mission file to identify the behaviors active in each of the sub-states, and hence controls the sub-state transitions, . Upon detection the node enters the *Ambiguous* sub-state, where `pBearingTrack` executes a algorithm that resolves the left-right ambiguity of a linear hydrophone array, executed in synchronization with a set of turn behaviors executed by the `IvP Helm` , as described below. Following the ambiguity resolution, the processor enters the *Tracking* sub-state, where the stabilized bearing tracker is used to estimate the current bearing to the source, which is continuously published to the `MOOSDB` for use of behaviors that optimizes the generation of an on board Target Motion Analysis (TMA), which is generated by a separate tracking module. A separate, real-time beamformer is activated in the *Classification* sub-state, for estimating the relative significance of shallow and steep propagation angles, which may be used as an indicator of whether the source is submerged or not.

6.6 `MOOS-IvP` Simulation Environment

The modular `MOOS-IvP` architecture directly lends itself to the establishment of a high-fidelity simulation environment. Thus, the actual mission files, and all the `MOOS` processes to be used for the underwater vehicle may be operated unchanged on the simulation computer, except those few processes that interact with the outside world. In the `MOOS` community shown in Fig. 7 the latter include the process `pHuxley` communicating with the main vehicle computer. In the simulator `pHuxley` is instead set up to communicate with a platform simulator process emulating the dynamic response of the vehicle to the helm commands for desired speed, heading and depth. Similarly, in the simulator the modem used for underwater acoustic communication is either replaced by an emulator using the actual modem hardware, or a software simulator of the modem network. The MIT `MOOS-IvP` simulator includes such a simulator, emulating realistic underwater communication obstacles such as propagation latency and intermittency, and message collisions. Finally, the simulator includes processes which replace the external sensors such as a CTD or an acoustic array. Thus, for example the MIT high-fidelity simulator incorporates a process for emulating CTD data collection by interpolating in a virtual ocean created by the HOPS ocean circulation model for the environment of interest. The data collected by an acoustic array are similarly emulated at the level of the the individual sensors, allowing for complete simulation of the processing chain and its feedback to and from the platform control. This feature is crucial to the development and testing of adaptive sampling behaviors.

Part III

MIT-LAMS Nested Autonomy Prototype

7 MOOS-IvP Nested Autonomy Architecture

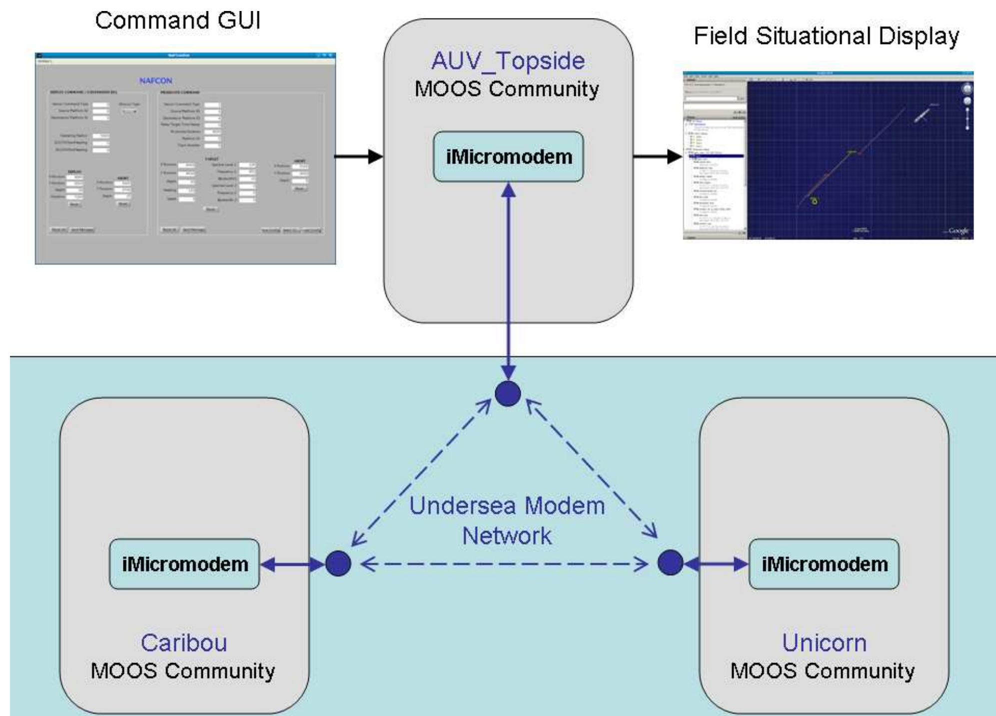

Figure 10: Schematic representation of MIT MOOS-IvP Undersea Autonomous Network Architecture. The topside MOOS Community is connected to all MOOS communities on the submerged or surfaced nodes through an acoustic modem network. The network, including any number of AUV and ASC communities, is communicating using the WHOI Micromodem through the `iMicroModem` MOOS utility.

The network communication, command and control architecture developed and operated by the MIT Laboratory for Autonomous Marine Sensing (LAMS) is shown in Fig. 10. The topside command and control MOOS community `AUV_Topside`, uses GUIs and the situational displays to control and command the network of fixed and mobile platforms, each operated by a `MOOS` community, through dedicated commands using the Compact Control Language (CCL) protocols for the WHOI Micromodem.

7.1 Sonar AUV MOOS Community

The MIT Laboratory for Autonomous Marine Sensing (LAMS) operates two Bluefin21 AUVs for research into adaptive and collaborative, autonomous acoustic sensing in the ocean. The vehicles, called Unicorn, and Caribou or Macrura depending on the configuration, can be carrying a variety of customised acoustic source/receiver payloads. In configurations for seabed object detection, classification and localization the two vehicles each have a 16-element fixed hydrophone array mounted

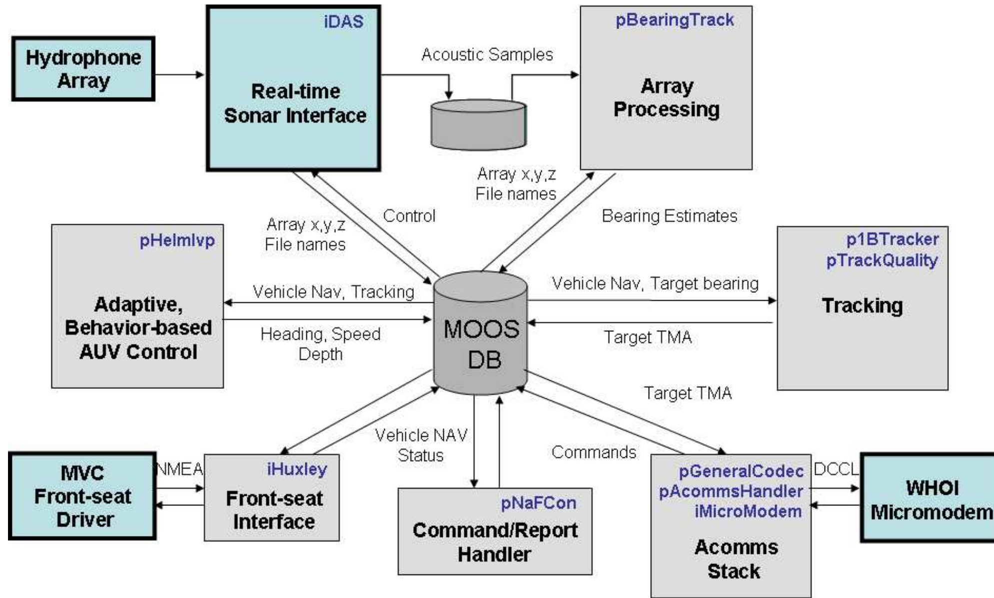

Figure 11: MIT MOOS-IvP Sonar-AUV Community: Principal MOOS-IvP modules shaded in grey, incorporating the IvP-Helm, the handlers af commands and reports communicated with field control, and onboard signal processing. The three principal interfaces to the 'outside world' are shaded in blue: 1. The main vehicle computer (MVC) containing the 'front-seat driver' which handles the lower level AUV navigation and control. 2. The WHOI Micromodem which handles all sub-surface network communication. 3. The sonar, consisting of a receiving hydrophone array, for active systems a source, and a control and data acquisition system (iDAS).

in the front section, and an active source for insionifying the seabed. For passive and multistatic acoustic research below 1 kHz, Unicorn is integrated with a 32-element towed hydrophone array.

In either case, the vehicles are operating with MOOS-IvP handling the higher-level intelligent autonomy, implemented on a computer stack in the payload section, integrated with the acoustic source control and the data acquisition system. The MOOS-IvP configuration is shown schematically in Fig. 11. The MOOS-IvP Autonomy System is connected to the main vehicle computer using the 'back-seat driver' paradigm, consistent with the proposed ASTM F41 standard for AUVControl. The MOOS interface to the front-seat driver is platform dependent, with several existing in the source tree, for example `iHuxley` for BF21 and `iOEX` for the Ocean Explorer. Other connections from the payload include the WHOI Micromodem, which forms the backbone of the undersea conmmunication network. The modem communication is handled by the MOOS communication stack, consisting of a generic message coder-decoder `pGeneralCodec`, and the `pAcommsHandler`process handling message queueing to and from the modem driver module `iMicroModem`. The main purpose of the sonar payload is obviously to operate the acoustic sources and record and process data from the hydrophone array. The interface is provided by a dedicated data acquisition MOOS module `iDAS` (Digital Acquisition System), which interacts with the signal conditioning for each array element. `iDAS` subscribes for control parameters and commands from the MOOSDB, and publishes array element locations and a filename, every time a new snippet of data is available. The high-bandwidth nature of the raw acoustic data makes it inconvenient to publish these directly in the MOOSDB. Instead they are written to a file, which also serves as permanent recording, and only the file location and name is published, for other processes to access, such as the signal processing

module, which for acoustic arrays will typically derive a bearing estimate for a source or a scatterer and publish the result in the MOOSDB for other processing such as tracking.

In addition to these three main connections, there is also a process `iCTD` controlling and receiving data from the CTD sensor which is standard for the vehicle. Also, some of the payloads have their own GPS antenna for accurate timing, and here the MOOS interfacing is handled by the `iGPS` process.

¡¡¡¡¡¡¡ .mine

=======

Part IV

IvP-Helm Autonomy for Undersea Sensing

8 Mission Planning and Control

The typical mission for an AUV performing an adaptive mission involving the capture of an episodic event, such as the detection and tracking of a marine mammal, is governed by more than 30 behaviors, with each sub-state activating a dedicated suite of behaviors. All active behavior produce objective functions for desired speed, depth and heading, which the `pHelmIvP`process then combines into an optimal compromise using its interval programming, multi-objective optimization.

The helm, upon startup, reads a behavior configuration file with sets of state defining conditions and mission parameters. An important component is to allow behaviors to be adaptive not only to environment events, but also to periodic direction from field control. This means a mission file is not a static script with a start and completion, but more aptly described as a state space with an initial state and conditions for migrating between states based on events; mission-control, environment or otherwise.

9 Hierarchial State Representation

An important step in the design of a `MOOS-IvP` autonomy system is to translate the state definitions and the transitions shown in Fig. 9 into a behavior configuration file. The newly added capability of `MOOS-IvP` to allow a Hierarchial State Machine (HSM) representation facilitates this process significantly, and making behavior conflicts more managable. The HSM representation of the current state definitions of the MIT sonar AUVs is shown in Fig. 12. In this form the states are shown in a hierarchial manner, with the states shown as 'parents' and 'children'. Note that the state digram in this form does not indicate the state transitions. This is due to the fact that in the LAMS architecture, transitions are allowed between all combination states, and controlled by either the autonomy or the operators.

9.1 MOOS-IvP State Definitions

The following shows the statements in the behavior file `Unicorn_HSM` that defines the uppermost states in Fig. 12.

Listing 9.1 - MOOS-IvP State Definitions in behavior file `Unicorn_HSM.bhv`

```
 1    initialize  DEPLOY_STATE = FALSE
 2    initialize  GOTO_SURFACE = FALSE
 3    initialize  PROSECUTE_STATE = FALSE
 4 // In mission or all stop on surface waiting for new deploy command
 5
 6 set MODE = ALLSTOP {
 7     (DEPLOY_STATE = FALSE) and (PROSECUTE_STATE = FALSE)
 8     } MISSION
 9 // Diving (active mission) or surfacing (mission done)
10 set MODE = SURFACING {
11     MODE = MISSION
12     GOTO_SURFACE = TRUE
13     } DIVING
```

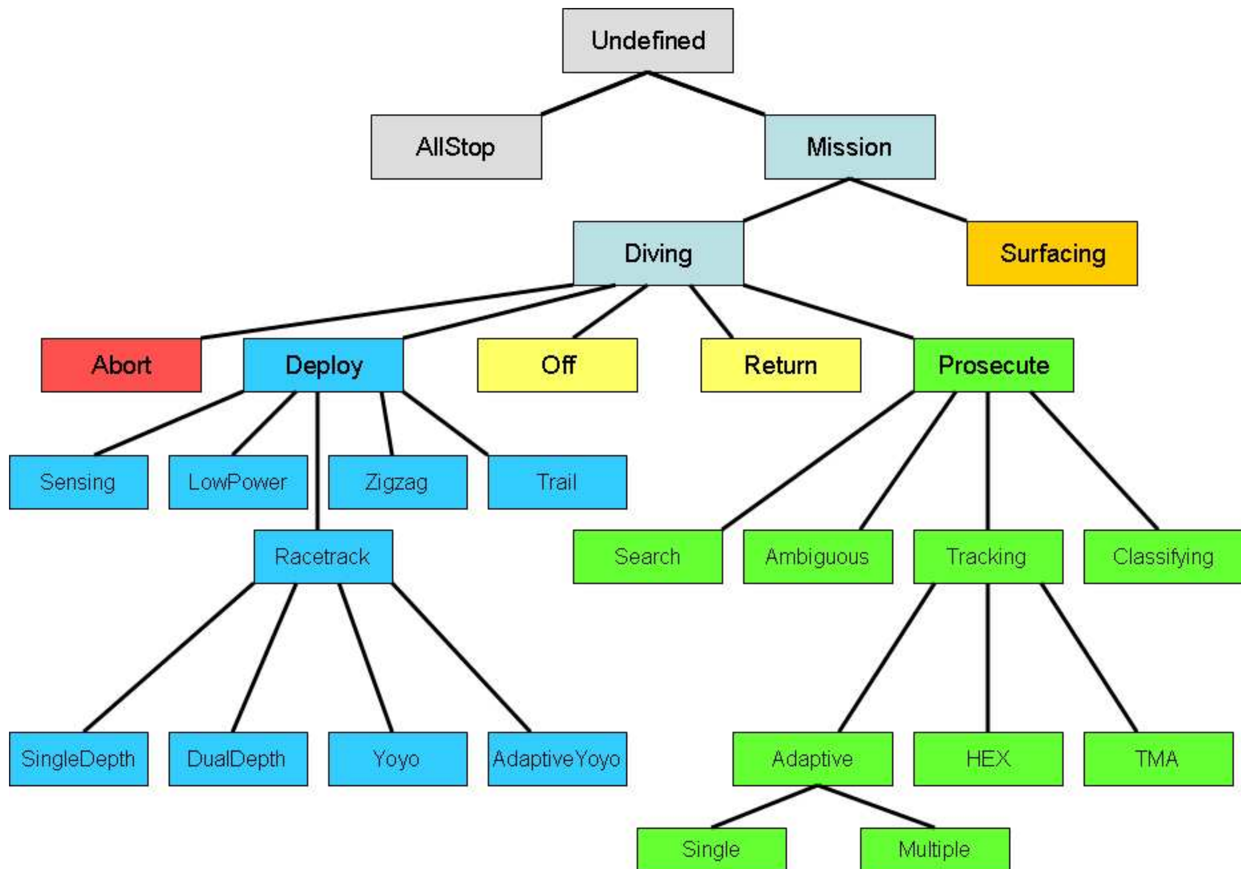

Figure 12: Hierarchial State Machine representation of current MIT-LAMS sonar AUV autonomy system.

```
14 // Deploy, Off, Return, Prosecute or Abort
15 set MODE = DEPLOY {
16      MODE = MISSION:DIVING
17      (DEPLOY_STATE = DEPLOY) and (PROSECUTE_STATE = FALSE)
18      }
19 set MODE = OFF {
20      MODE = MISSION:DIVING
21      (DEPLOY_STATE = OFF) and (PROSECUTE_STATE = FALSE)
22      }
23 set MODE = RETURN {
24      MODE = MISSION:DIVING
25      (DEPLOY_STATE = RETURN) and (PROSECUTE_STATE = FALSE)
26      }
27 set MODE = ABORT {
28      MODE = MISSION:DIVING
29      (DEPLOY_STATE = ABORT) or (PROSECUTE_STATE = ABORT)
30      }
31 set MODE = PROSECUTE {
32      MODE = MISSION:DIVING
33      (PROSECUTE_STATE = PROSECUTE) and (DEPLOY_STATE = FALSE)
34      }
```

Lines 1-3 initialize the state variables controlling the state definitions at mission start. The first state definition block in lines 6-8 decides whether the vehicle is inactive, state `MODE=ALLSTOP`, on the

surface, which is the initial state with the initializations in lines 1-3, or it is in a mission, controlled by the state variables DEPLOY_STATE and PROSECUTE_STATE.

The next blocks defines whether the vehicle is actively diving, MODE=MISSION:DIVING or it is in a surfacing mode, MODE=MISSION:SURFACING, controlled by the state variable GOTO_SURFACE.

If in vehicle is in the active diving state, the combination of the values of the state variables DEPLOY_STATE and PROSECUTE_STATE define a set of mission classes. Three of these are mission terminating states, RETURN, OFF, and ABORT, while the DEPLOY and PROSECUTE states are active mission states, each with a number of sub-states as shown in Fig. 12, and described below.

9.2 Mission Terminating States

The three mission terminating states are very similar, taking the vehicle to a termination point at depth, and then surfacing. As an example, the following shows the configuration for the behaviors which become active in the RETURN state. Note that for each state there must be an active behavior producing objective functions for both speed, heading and depth.

Listing 9.2 - Behavior configurations for RETURN state

```
 1 Behavior = BHV_Waypoint
 2 {
 3    name      = return_to_waypoint
 4    speed     = 1.5
 5    radius    = 50.0
 6    nm_radius = 100.0
 7    updates   = SENSOR_RETURN
 8    duration  = no-time-limit
 9    perpetual = true
10    condition = (MODE == RETURN)
11    endflag   = GOTO_SURFACE,TRUE
12 }
13
14 Behavior = BHV_ConstantDepth
15 {
16    name      = return_depth
17    updates   = SENSOR_DEPTH_RETURN
18    duration  = no-time-limit
19    condition = (MODE == RETURN)
20 }
```

The following features of the behavior configuration should be noted;

- The condition for activating these behaviors uses the '==' comparison feature of the helm, which implies that the behaviors will be active will be active in any RETURN state, irrespective of it's place in the tree. In the present case there are no duplicate state names, but if not used, the entire tree sequence MISSION:DIVING:RETURN would have to be specified in the condition.
- The return location is pecified dynamically through the updates feature, linked to the MOOS variable SENSOR_RETURN, which will contain a string, gebnerated by the pNaFCon process, such as: points=1234,5432. Similarly the depth is specified dynamically through the MOOS variable SENSOR_DEPTH_RETURN.
- Once the return location is reachede, the vehicle is transitioned into the MISSION:SURFACING state, by the endflag-specified action of setting GOTO_SURFACE=TRUE.

The `MODE=OFF` state is currently identical to the `RETURN` state. Both the `RETURN` and `OFF` states are activated via commands from the operators, for deliberate mission termination. In contrast the `ABORT` state is acticvated autonomously by the autonomy system violating established system limits, such as mission duration, exceeding the opration radius, etc. The amission abort is not intantaneous, but will transition the vehicle to an abort point specified by the operator in the *Deploy Command* or *Prosecute Command*.

Once the termination point is reached, and the `SURFACING` state is entered, the surfacing behavior is controlled by the following behavior set:

Listing 9.3 - Behavior configurations for SURFACING state

```
 1 Behavior = BHV_GoToDepth
 2 {
 3    name          = surface_depth
 4    pwt           = 100
 5    depth         = 2.5
 6    arrival_delta = 1
 7    perpetual  = true
 8    condition = (MODE == SURFACING)
 9 // reset to ALLOFF mode, ready for new commands
10    endflag       = DEPLOY_STATE = FALSE
11    endflag       = PROSECUTE_STATE = FALSE
12    endflag       = GOTO_SURFACE = FALSE
13 }
14
15 Behavior = BHV_ConstantSpeed
16 {
17    name          = surface_speed
18    pwt           = 100
19    speed         = 1.5
20    duration = no-time-limit
21    condition     = (MODE == SURFACING)
22 }
```

Note here, that when the vehicle reaches the surfacing depth, the `endflag` feature is used to reset the state variables to the initial values, transitioning the vehicle into the `ALLSTOP` state, and the vehicle will stop propulsion and drift to the surface.

9.3 Deploy States

The various *Deploy* modes of the nested autonomy concept of operations involve relatively simple behaviors, such as putting the vehicle into a hexagonal loiter or racetrack survey patterns. The *Deploy* substates are defined using the following constructs in the behavior configuration file:

Listing 9.4 - MOOS-IvP Deploy State Definitions in behavior file `Unicorn_HSM.bhv`

```
 1 // Deploy States
 2 set MODE = SENSING {
 3     MODE = MISSION:DIVING:DEPLOY
 4     (DEPLOY_MISSION = 0) and (TRAIL_MACRURA = FALSE)
 5     }
 6
 7 set MODE = LOWPOWER {
 8     MODE = MISSION:DIVING:DEPLOY
 9     DEPLOY_MISSION = 1
10     }
11
12 set MODE = RACETRACK {
```

```
13     MODE = MISSION:DIVING:DEPLOY
14     (DEPLOY_MISSION = 4) or (DEPLOY_MISSION = 5)
15     }
16
17 set MODE = ZIGZAG {
18     MODE = MISSION:DIVING:DEPLOY
19     DEPLOY_MISSION = 6
20     }
21
22 set MODE = TRAIL {
23     MODE = MISSION:DIVING:DEPLOY
24     (DEPLOY_MISSION = 0) and (TRAIL_MACRURA = TRUE)
25     }
```

9.3.1 Loiter Sub-States

The SENSING and LOWPOWER states are currently both configured to place the vehicle in a heaxagonal loiter pattern, centered at the commanded location. The principal difference between the two states is that the SENSING has the sensors and onboard processing active, while the LOWPOWER state has them turned off for power saving. In the standard mission configurations, the vehicle is initiially placed in the LOWPOWER state using the pMessageSim module once the backseat driver autonomy is handed control. This is particularly important for AUV configurations with the modem on top, making acoustic commanding impossible at the surface.

The following lists the current configuration block for the LOWPOWER state:

Listing 9.5 - Behavior configurations for LOWPOWER Deploy state

```
 1 //Basic deploy - go to loiter around deploy location
 2 Behavior = BHV_Loiter
 3 {
 4   name      = deploy_loiter
 5   speed     = 1.5
 6   radius    = 50.0
 7   nm_radius = 100.0
 8   updates   = SENSOR_DEPLOY
 9   duration  = no-time-limit
10   clockwise = TRUE
11   acquire_dist = 40
12   condition = MODE == LOWPOWER
13 }
14
15 Behavior = BHV_ConstantDepth
16 {
17    name      = deploy_depth
18    updates   = SENSOR_DEPTH_DEPLOY
19    duration  = no-time-limit
20    condition = MODE == LOWPOWER
21 }
```

The loiter parameters are specified using the updates construct, using the MOOS variable SENSOR_DEPLOY which will contain the location, radius and number of sides in the polygon, f.ex. polygon=radial:2000,3000,300,6, with depth specified through the variable SENSOR_DEPTH_DEPLOY constraining a string such as depth=25.0. Note also that all behaviors have duration=no-time-limit, remaining active as long as the mission lasts, so that each state can be entered and exited an arbitrary number of times.

9.3.2 Racetrack Sub-State

The racetrack substate places the vehicle in a racetrack survey pattern, typically used to create long trackline surveys. The width of the racetrack, i.e. the distance between the outgoing and incoming tracklines is specified in the behavior configuration block. In contrast to all other current *Deploy States* the racetrack state has several substates, depending on the depth behavior.

`SINGLEDEPTH`: Both outgoing and ingoing tracklines at same depth, specified in `SENSOR_DEPTH_DEPLOY`.

`DUALDEPTH`: Outgoing and ingoing tracklines at different, fixed depth, specified directly in configuration block.

`YOYO`: Depth Yoyo.

`ADAPTIVEYOYO`: This is a special environmental sampling behavior, where the yoyo pattern is daptively adjusted to sample the oceanograsphy in the vicinity of a thermocline. Used in conjunction with the `pThermoTrack` process, which processes the sampled oceanographic data and detects when the thermocline has been passed and then reverses the depth change.

The racetrack sub-states are defined as follows in the behavior file:

Listing 9.6 - MOOS-IvP State Definitions in behavior file `Unicorn_HSM.bhv`

```
 1 // Racetrack depth modes
 2 set MODE = SINGLEDEPTH {
 3     MODE = MISSION:DIVING:DEPLOY:RACETRACK
 4     (DEPLOY_MISSION = 4) and (RACETRACK_ALT_DEPTH = FALSE)
 5     }
 6
 7 set MODE = DUALDEPTH {
 8     MODE = MISSION:DIVING:DEPLOY:RACETRACK
 9     (DEPLOY_MISSION = 4) and (RACETRACK_ALT_DEPTH = TRUE)
10     }
11
12 set MODE = YOYO {
13     MODE = MISSION:DIVING:DEPLOY:RACETRACK
14     DEPLOY_MISSION = 5
15     }
16
17 set MODE = ADAPTIVEYOYO {
18     MODE = MISSION:DIVING:DEPLOY:RACETRACK
19     DEPLOY_MISSION = 7
20 }
```

The following shows the behavior configurations for the `YOYO` state:

Listing 9.7 - Behavior configurations for YOYO Deploy state

```
 1 Behavior = BHV_RaceTrack
 2 {
 3   name      = deploy_racetrack
 4   track_length = 2000
 5   track_width  = 0
 6   distance_tolerance  = 15
 7   approach_angle = 60
 8   heading   = 270
 9   speed     = 1.5
10   points = 500.0, 200.0
```

```
11   updates = SENSOR_RACETRACK
12   duration  = no-time-limit
13   condition = (MODE == RACETRACK)
14 }
15 Behavior = BHV_SmartYoyo
16 {
17   name      = environmental_yoyo
18   updates = SENSOR_YOYO
19   min_depth = 5
20   max_depth = 60
21   min_altitude = 10
22   period = 300
23   period_segments = 200
24   depth_function = sinusoidal
25   duration  = no-time-limit
26   condition = (MODE == YOYO)
27   condition = (ON_TRACKLINE = TRUE)
28 }
29 Behavior = BHV_ConstantDepth
30 {
31   name      = yoyo_approach_depth
32   updates   = SENSOR_DEPTH_DEPLOY
33   duration  = no-time-limit
34   condition = (MODE == YOYO)
35   condition = (ON_TRACKLINE = FALSE)
36 }
```

The following features should be noted:

- The `BHV_RaceTrack` configuration block is common for all racetrack substates.
- Even though fixed values are given for several parameters, the racetrack geometry is defined primarily through the use of the `pHelmIvPupdates` feature. Thus, the starting point of the racetrack, the trackline length, and the heading is specified in the MOOS variable `SENSOR_RACETRACK`, containg a string such as `points=2000,3000#heading=270#track_length=2000`. The separation of the tracklines is in this case fixed to `track_width=0`, corresponding to the outgoing and returning tracklines being on top of each other. However, it could be included in the `updates` variable, obviously. The reason this is not done currently, is that the topside command console and the messsage set do not currently support the commanding of this variable.
- The vertical yoyo behavior `BHV_SmartYoyo`, described in detail in sec. **??**, is similarly controlled through updates via the MOOS variable `SENSOR_YOYO`, containing a string such as `min_depth=10#max_depth=80`.
- The vertical yoyo behavior is only activated while on the trackline, flagged in the MOOS-variable `ON_TRACKLINE`. When approaching the trackline, and during the end-turns a constant depth is instead commanded through the standard depth behavior BHV_ConstantDepth.

9.3.3 ZigZag Sub-state

This behavior executes a horizontal zigzag survey at constant depth for oceanographic sampling, e.g. for mapping a frontal feature. It is non-adaptive, and the operator must specify the starting pont, the average heading and the length of the survey in the heading direction. The following shows a listing of the behavior configurations for this state.

Listing 9.8 - Behavior configurations for ZIGZAG Deploy state

```
 1 Behavior = BHV_Waypoint
 2 {
 3   name      = zigzag_mission
 4   speed     = 1.5
 5   radius    = 50.0
 6   nm_radius = 100.0
 7   updates   = ZIGZAG_STATUS
 8   perpetual = true
 9   duration  = no-time-limit
10   condition = (MODE == ZIGZAG)
11   endflag   = GOTO_SURFACE,TRUE
12 }
13
14 Behavior = BHV_ConstantDepth
15 {
16   name      = zigzag_depth
17   updates   = SENSOR_DEPTH_DEPLOY
18   duration  = no-time-limit
19   condition = (MODE == ZIGZAG)
20 }
```

The zigzag pattern is generated by the `pNaFCon` process in response to the corresponding *Deploy Command*, with the geometry specified in the MOOS-variable `ZIGZAG_STATUS`, containing a parameter string such as `points=zigzag:5000,9000,50,1000,300,150`, with the numbers represnting the starting x- and y-coordinates, the heading, the length of the survey, the period, and the amplitude.

The depth is specified as for the other deploy states using the update MOOS-variable `SENSOR_DEPTH_DEPLOY`.

Note that at the end of the survey, the vehicle will surface in this state. If one desires the vehicle to return to the the `LOWPOWER` loiter state, f.ex., line 11 should simply be replaced by `endflag = DEPLOY_MISSION = 1`.

9.3.4 Trail Sub-state

This is a new autonomy state that was first demonstrated in the GLINT'08 experiment. In this state, the vehicle will perform synchronized swimming with another vehicle. This is achieved by constantly extrapolating the location and speed of the other vehicle, using the *Status Reports* received over the acoustic modem. If communication is lost to the other vehicle for a duration exceeding a specified time, the vehicle will enter a local loiter pattern until communication is re-established, and the pursuit can be resumed.

The following shows the behavior configurations for this state.

Listing 9.9 - Behavior configurations for TRAIL Deploy state

```
 1 Behavior = BHV_Trail
 2 {
 3   name = trail_macrura
 4   contact = MACRURA
 5   trail_range = 300
 6   trail_angle = 315
 7   trail_angle_type = absolute
 8   radius = 10
 9   nm_radius = 100
10   max_range = 1000
11   decay = 60,180
12
13   condition = (MODE == TRAIL)
```

```
14   condition = (MACRURA_NAV_SPEED >= 1.0)
15   runflag   = TRAILING = TRUE
16   idleflag   = TRAILING = FALSE
17   activeflag = ACTIVE_TRAIL = TRUE
18   inactiveflag = ACTIVE_TRAIL = FALSE
19 }
20
21 Behavior = BHV_Loiter
22 {
23   name      = trail_idle_loiter
24   pwt       = 100
25   speed     = 1.4
26   radius    = 50.0
27   nm_radius = 100.0
28   updates   = SENSOR_DEPLOY
29   center_activate = true
30   duration  = no-time-limit
31   clockwise = FALSE
32   acquire_dist = 40
33   condition = (MODE == TRAIL)
34   condition = (ACTIVE_TRAIL = FALSE)
35 }
36
37 Behavior = BHV_ConstantDepth
38 {
39   name      = trail_depth
40   updates   = SENSOR_DEPTH_DEPLOY
41   duration  = no-time-limit
42   condition = (MODE == TRAIL)
43 }
```

The following features should be noted:

- This state uses a newly added set of flags to switch from the trailing mode to an idle loiter state in cases where communication is lost, or the trailed vehicle is moving below a certain speed, e.g. when it surfaces for GPS. The `activeflag` and `inactiveflag` parameters are used to specify MOOS-variable settings indicating whether the behavior is generating an objective function or not. Thus, when the conditions keep the trail behavior form generating an objective function, it will flag this by setting `ACTIVE_TRAIL=FALSE` in the MOOSDB, and the idle loiter behavior will become active until the flag is reversed, and the pursuit will be resumed.
- The trail in this case is set to an absolute bearing from the contact.
- Since the current command set does not allow for specifying thye trail distance and bearing angle, these must be set to fixed values in the behavior file.

9.4 Prosecute States

A major challence is the execution of the autonomous *Prosecute* mission, involving all the components of the completely autonomous strategy for the capturing and characterizing the event in question. The *Prosecute* substates are defined using the following constructs in the behavior configuration file:

Listing 9.10 - MOOS-IvP Prosecute State Definitions in behavior file `Unicorn_HSM.bhv`

```
 1 // Prosecute Missions
 2 set MODE = SEARCH {
 3     MODE = MISSION:DIVING:PROSECUTE
 4     TRACKING = NO_TRACK
 5     }
 6
 7 set MODE = AMBIGUOUS {
 8     MODE = MISSION:DIVING:PROSECUTE
 9     TRACKING = AMBIGUOUS
10     }
11
12 set MODE = TRACKING {
13     MODE = MISSION:DIVING:PROSECUTE
14     TRACKING = TRACKING
15     }
16
17 set MODE = CLASSIFYING {
18     MODE = MISSION:DIVING:PROSECUTE
19     TRACKING = CLASSIFYING
20     }
```

9.4.1 Search Sub-State

The initial *Search* prosecute sub-state is entered upon receipt of a *Prosecute Command* from *NaF-Con.* Entering this state, a suite of behaviors optimizing the propability of detection of the event are activated, synchronized with the modules processing the sensor data. Irrespective of the nature of the event, theSearch state involves two behaviors that provide a trade-off between closing range to the cued event location and requirements for station-keeping. By dynamically assigning priorities to these behaviors, the autonomoy system may allow the cluster to automatically dispatch the vehicles closest to the event to be more actively pursuing the event by using available information about the positions of other assets in the cluster.

The `BHV_Attractor` behavior, described in Sec. 10.1, uses the MOOS-IvP *CPA engine* to attract the vehicle towards the Closest Point of Approach (CPA) for the projected event track. Inside a mininimum predicted range, the priority is set to zero, basically turning off the attractor behavior. This feature is used in connection with remote sensing missions using optical or acoustical sensors.

The objective of keeping the vehicle in the vicinity of its assigned station point is handled by the IvP behavior `BHV_RubberBand`, described in detail in Sec. 10.2. This behavior is a 'soft' version of the standard station-keeping IvP behavior, pulling the vehicle back towards its assigned station point.once it exceeds a range set in the configuration file. Outside this range, the priority is linear with distance, similar to the elasticity of a rubber band, hence the name.

The priority of these two competing behaviors can either be fixed or dynamically determined by a command and control process, such as `pClusterPriority`, for example, based on the relative distances of the cluster assets to the target event. By setting the relative valus of the RubberBand and Attractor behaviors, the system designer may control the relative importance of prosecuting an event and maintain stationkeeping. In a cluster configuration with several vehicles, these priorities may be set dynamically using the general IvP `UPDATES` parameter. Thus, to avoid all vehicles in a cluster chasing the same target, the priorities may be set dynamiclly to favorize the assets closest to the projected target track, with 'down-stream' assets not abandoning the target but instead gaining priority with time in case the closer assets miss the detection opportunity. The balancing of these priorities is the *Cluster Autonomy*, the next level of the *Nested autonomy.* Also, while the concept has been developed for the tracking of single target event, this is no inherent limitation.

The handling of multiple targets depends on the capabilty of the signal processing module and the availability of a process handling the *Contact Management*, the subject of future developments.

Depending on the mission objectives, other behaviors may be added. For example, for improving detection of an acoustic source in an anisotropic noise field, the noise interferrence may be minimized using certain array orientations.

The standard behavior configuration block for the SEARCH state is

Listing 9.11 - Behavior configurations for Prosecute SEARCH state

```
 1 Behavior = BHV_Attractor
 2 {
 3   name      = search_attract
 4   updates   = MY_TARGET
 5   priority  = 100
 6   duration  = no-time-limit
 7   condition = (MODE == SEARCH)
 8   condition = IN_RANGE,FALSE
 9   contact = TGT_1
10   dist_priority_interval = 1000,2000
11   strength = 1.0
12   decay = 1800,3600
13   patience = 50
14 }
15 Behavior = BHV_RubberBand
16 {
17   name      = search_rubberband
18   updates   = DEPLOY_STATION
19   priority  = 100
20   duration  = no-time-limit
21   condition = (MODE == SEARCH)
22   condition = IN_RANGE = FALSE
23   inner_radius = 500
24   outer_radius = 500
25   outer_speed  = 1.5
26   stiffness = 0.5
27 }
28 Behavior = BHV_ConstantDepth
29 {
30   name      = search_depth
31   updates   = SENSOR_DEPTH_DEPLOY
32   duration  = no-time-limit
33   condition = (MODE == SEARCH)
34 }
35
36 Behavior = BHV_Loiter
37 {
38   name      = search_loiter
39   speed     = 1.5
40   radius    = 50.0
41   nm_radius = 100.0
42   updates   = PROSECUTE_LOITER
43   duration  = no-time-limit
44   clockwise = TRUE
45   acquire_dist = 40
46   condition = (MODE == SEARCH)
47   condition = IN_RANGE,TRUE
48 }
```

The following features should ne noted

- The MOOS-variables MY_TARGET is used to dynamically assign a specific target ID to the attractor behavior, as well as the desired priority, f.ex. contact=TGT_6#priority=156.

- The DEPLOY_STATION is used are used to define the currently assigned deploy station.
- The IN_RANGE flag indicates whether the vehicle is inside a specified range of the expected target track, without yet having made a detection. If IN_RANGE=TRUE the vehicle will go into a waiting pattern, here a loiter pattern, until the event is detected, or the prosecute mission is timed out. The loiter may of course be replaced by any other behavior which improves the detection probability. Thus, for detecting a chemical plume or an upwelling event, a traditional ocean search behavior, such as an expanding spiral survey may be applied.

9.4.2 Ambiguous Sub-State

Once an event is detected, the node enters the *Ambiguous* sub-state, where the onboard signal processing in corncert with dedicated maneuvering will attemt to break any sensing ambiguities.

Examples of sensing ambiguities include the direction of a detected ocean front, which must be determined to autonomously decide on a mapping survey strategy, and the left/right ambiguity of the bearing from an acoustic array to a source.

For the source tracking problem using a linear hydrophone array, the onboaard processor, e.g. pBearingTrack, will determine the relation between a forced array turn and the associated change in source bearing. A behavior controlling the change in array heading is *BHV_HArrayTurn*.

If the sensing is unambiguous, this state is simply never entered by the onboard processing, and the system is transitioned directly into the TRACKING or CLASSIFYING states.

The following shows the behavior configurations for the acoustic source tracking problem

Listing 9.12 - Behavior configurations for Prosecute AMBIGUOUS state

```
 1 Behavior = BHV_ConstantDepth
 2 {
 3   name      = ambiguous_depth
 4   updates   = SENSOR_DEPTH_DEPLOY
 5   duration  = no-time-limit
 6   condition = (MODE == AMBIGUOUS)
 7 }
 8
 9 Behavior = BHV_HArrayTurn
10 {
11   name = ambiguous_arrayturn
12   turn_angle = 120
13   pwt  = 700
14   condition = (MODE == AMBIGUOUS)
15 }
16
17 Behavior = BHV_ConstantSpeed
18 {
19   name      = ambiguous_speed
20   speed     = 1.5
21   duration  = no-time-limit
22   condition = (MODE == AMBIGUOUS)
23 }
```

Note that as in all other states, there must be behaviors generating objective functions for both speed, heading and depth. Since no specific depth is currently set for the prosecute, the desired depth is simply inherited from the previous deploy command, through the updates variable SENSOR_DEPTH_DEPLOY.

9.4.3 Tracking Sub-State

Once the ambiguity is resolved by the on-board signal processing, it sets the sub-state flag to TRACKING, andthe helm will enter the *Tracking* sub-state, activating behaviors which maneuvers the vehicle to optimize the tracking of the target event..

There are currently three *Tracking Sub-states* defined for the undersea sensing autonomy system. In addition to the principal, adaptive event prosecution, the behavior set also allows the nodes to perform more conventional surveys in response to the *Prosecute Command*, such as a horizontal zig-zag along a frontal bounday.

`ADAPTIVE` This is the principal, adaptive state for optimally track an acoustic source, or a target for active sonar applications. The behaviors in this state defines a platform maneuvering which weighs the various mission oblectives and constraints against a configuration which optimizes tracking performance. The active behaviors in this state include the *BHV_Attractor* and *BHV_RubberBand* behaviors also used in the *Search* sub-state, but not necessarily with the same relative priority. Thus, once the acoustic source is detected and the ambiguity resolved, the vehicle should likely be allowed to increase the priority of the attractor behavior and reduce the importance of the station keeping. Also, the safety behaviors, such as collision avaoidance behaviors, obviously remain active. All of these behaviors will be competing with the principal tracking behavior, *BHV_HArrayAngle*, with the relative significance controlled by the assigned priority weights. If authorized in the configuration, an adaptive sub-mode, `MULTIPLE` will allow the vehicle to fuse source bearing information from collaborating nodes for generating an improved track solution. If no *Contact Reports* are received from other nodes, the system will instead be in the `SINGLE` state, where it will maneuver to optimize a single vehicle track solution. To optimize collaborative tracking dedicated behaviors may be activated in the `MULTIPLE` state.

`HEX` In this non-adaptive state the vehicle will enter a standard hexagonal loiter patter, with sensing and processing active.

`TMA` This state will execute the classical *Target Motion Analysis* zigzag survey with a heading commanded from the topside command and control. In this non-adaptive mode the node will use the `IvP Helm` behavior BHV_ZigZag to perform the survey, in a direction defined by a heading stated in a dedicated `MOOS` variable. In this non-adaptive mode this heading must be defined by the operator, e.g. based on oceanographic forecasts or intuition.

These tracking sub-states are defined as follows in the behavior file:

Listing 9.13 - Tracking Substate Definitions in behavior file `Unicorn_HSM.bhv`

```
 1 // Adaptive Prosecute states
 2 set MODE = ADAPTIVE {
 3     MODE =  MISSION:DIVING:PROSECUTE:TRACKING
 4     PROSECUTE_MISSION <= 2
 5     }
 6 set MODE = MULTIPLE {
 7     MODE =  MISSION:DIVING:PROSECUTE:TRACKING:ADAPTIVE
 8     (COLLABORATION_MODE = COLLABORATING) and (CLOSE_RANGE = TRUE)
 9     } SINGLE
10 // Survey prosecutes
11 set MODE = TMA {
12     MODE = MISSION:DIVING:PROSECUTE:TRACKING
```

```
13     PROSECUTE_MISSION = 3
14 }
15 set MODE = HEX {
16     MODE = MISSION:DIVING:PROSECUTE:TRACKING
17     PROSECUTE_MISSION = 4
18     }
```

The behavior configuration for the single vehicle adaptive source tracking is as follows

Listing 9.14 - Behavior configurations for the Adaptive Tracking state

```
 1 Behavior = BHV_Attractor
 2 {
 3   name      = track_attract
 4   updates   = MY_TARGET
 5   priority  = 100
 6   duration  = no-time-limit
 7   condition = (MODE == ADAPTIVE)
 8   contact = TGT_1
 9   dist_priority_interval = 500,2000
10   strength = 1.0
11   decay = 1800,3600
12 }
13 Behavior = BHV_ConstantDepth
14 {
15   name      = dcl_tracking_depth
16   updates   = SENSOR_DEPTH_DEPLOY
17   duration  = no-time-limit
18   condition = (MODE == TRACKING)
19 }
20 Behavior = BHV_ConstantSpeed
21 {
22   name      = track_speed
23   speed     = 1.5
24   duration  = no-time-limit
25   condition = (MODE == ADAPTIVE)
26 }
27 Behavior = BHV_HArrayAngle
28 {
29   name = track_array_angle
30   pwt  = 100
31   width = 60
32   desired_angle = 90
33   condition = (MODE == ADAPTIVE)
34 }
```

. Note the following features

- In this case the station-keeping behavior `BHV_RubberBand` has been disactivated.
- The priority distance interval has been changed compared to the settings in the `SEARCH` state to allow the vehicle to be attracted closer to the target for optimal tracking.
- The depth behavior is common to all tracking states (`MODE=TRACKING`).

9.4.4 Classification Sub-State

For the acoustic source tracking prototype, the behaviors active in this state attempt to optimize the estimation of the spectral composition of the acoustic field, an important classification clue. In the protototype, a dedicated behavior, `BHV_SmartYoyo` is used to provide a robust average estimate

of the angular spectrum over depth. This state is entered following the tracking state, after a satisfactory tracking solution has been achieved. The `BHV_Attractor` behavior is then used to force the vehicle to constantly point directly towards the estimated source position to maintain a linear hydrophone array with the source on endfire, such that all angular variability in the array response is associated with the vertical angular variability of the source signal. The acoustic array response is averaged over depth to provide a robust spectral estimate, using the depth-adaptive `BHV_SmartYoyo` behavior.

The current behavior configuration for the `CLASSIFYING` state is as follows

Listing 9.15 - Behavior configurations for the Smarting state

```
 1 Behavior = BHV_ConstantSpeed
 2 {
 3   name      = classify_speed
 4   speed     = 1.5
 5   duration  = no-time-limit
 6   condition = (MODE == CLASSIFYING)
 7 }
 8 Behavior = BHV_Attractor
 9 {
10   name      = classify_attract
11   updates   = MY_TARGET
12   priority  = 100
13   duration  = no-time-limit
14   condition = (MODE == CLASSIFYING)
15   contact = TGT_1
16   dist_priority_interval = 200,1000
17   strength = 1.0
18   decay = 1800,3600
19 }
20 Behavior = BHV_SmartYoyo
21 {
22   name      = classify_yoyo
23   min_depth = 10
24   max_depth = 60
25   min_altitude = 5
26   period = 1200
27   period_segments = 10
28   depth_function = linear
29   duration  = no-time-limit
30   condition = (MODE == CLASSIFYING)
31 }
```

The following features should be noted':

- The vertical yoyo in this case has a long period, 1200 m, with only 10 discrete depth changes. This is done to maintain the array at constant depth for a distance long enough for achieving a robust average of the vertical angular distribution at that depth.
- As for all states, there is no specified time limit. Instead, the state managing processes are performing the state transitions based on the data processing or presetdurations. Thus, the classification state is entered and terminated by the `pTrackQuality` process, as described elsewhere.

.

10 Cluster Autonomy Behaviors

This set of behaviors is used for distributed control of a cluster of vehicles which e.g. in response to a prosecute command have to autonomously search for, detect, localize and track an episodic event, such as an approaching target or oceanograhic feature. They are typically used together with the pClusterPriority MOOS process, which manages the priorities of these behaviors depending on the relative distances to the feature. This toolset enables the implemntation of the 'soccer control' paradigm, where all decision making is performed locally on each vehicle based on its own situational awareness and strategies controlled through the base priorities and the condition set. This set of behaviors is typically used for implemeting cluster control paradigms mimicking team sports not dependent on inter-player communication, such as soccer, hockey and basketball, where each player is making discisions based almost entirely on situational awareness, strategy and experience, rather than made in response to communication from captain, coach etc. Thus,, this behavior set was developed for undersea surveillance systems with severely limited communication capacity.

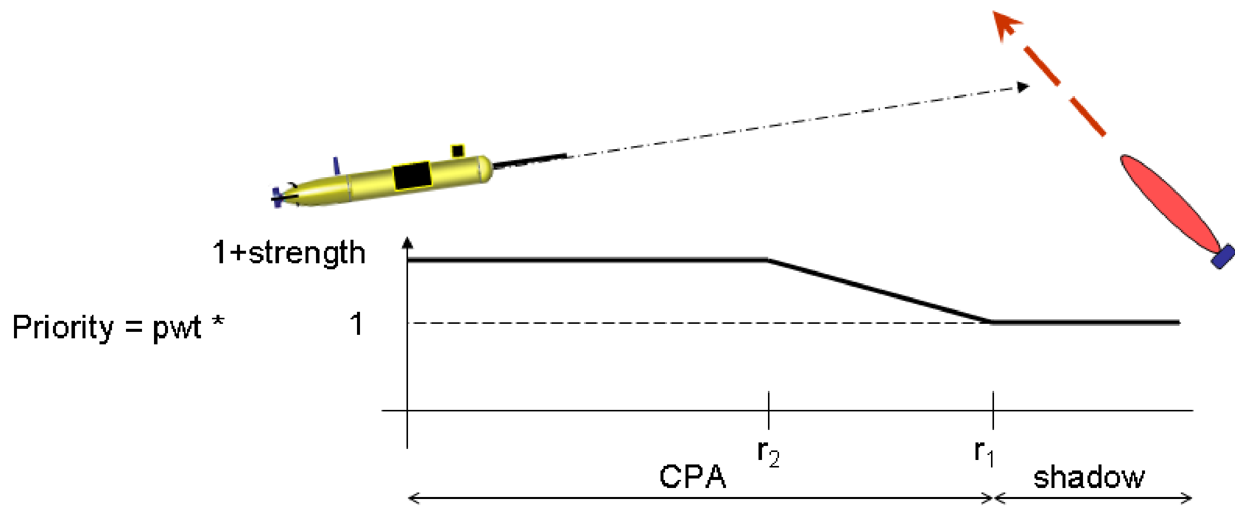

Figure 13: Range-dependent priority of BHV_Attractor.

.

10.1 BHV_Attractor

This behavior is similar BHV_CutRange, using the CPA engine to attract the vehicle towards the target location. The main difference is that this behavior has a user-defined range-dependence of the priority. It is used together with the BHV_RubberBand behavior and the pClusterPriority process to dynamically manage the attraction of multiple vehicles towards one or more targets. If the distance to the target is larger than a specified `GIVEUP_RANGE` the target will be ignored and the behavior will not produce an objective function.

`CONTACT:` Name of the vehicle or target which should be approached.

`STRENGTH:` Parameter controlling the strength of the attaction, affecting the range-dependence of the beahvior priority.

PRIORITY_INTERVAL: This parameter takes two arguments separated by a comma. Defines the range interval where the behavior transitions from "cutting range" to the contact ($r > r_2$), and simply "shadowing" the contact($r < r_1$). The default is $r_1 = 5$ and $r_2 = 15$. The assocoated priority transition is controlled by the STRENGTH parameter as shown in Fig. 13.

GIVEUP_RANGE: The distance (in meters) that the contact must be within for the behavior to be active and produce an objective function. The default giveup_range value is zero, making the behavior active active regardless of the distance to the contact.

PATIENCE: This parameter controls the 'patience' of the vehicle in chasing the target. The default is 0, which forces the vehicle to immidiately head for the contact, even if the contact is approaching and cutting range on its own. This parameter is useful when poaer conservation is critical, and when the vehicle is required to stay close to a station point (zone defense)

DECAY: This parameter takes two arguments separated by a comma. The first argument is the decay start time (in seconds), and the second is the decay end time (also in seconds). The behavior extrapolates the contact position based on the last known position, heading and speed. The speed of the contact begins to *decay* based on the time since the last contact update. This is a safeguard against perpetually trailing a vehicle that ceases to provide a contact report. The default is 5 and 10 seconds respectively. For undersea networks with acoustic communication only, 60-120 seconds would be a reasonable choice.

Listing 10.1 - An example BHV_Attractor configuration.

```
0  Behavior = BHV_Attractor
1  {
2    name         = bhv_trail
3    pwt          = 100
4    contact      = delta
5    strength     = 0.5
6    priority_interval = 100,200
7    patience     = 50
8    decay        = 20,60
9  }
```

10.1.1 MOOS variables subscribed to by BHV_Attractor:

.

MOOS variable	Type	Description	Published by
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
NAV_SPEED	D	Current vehicle speed in m/s	pHuxley
NAV_HEADING	D	Current ownship heading in degrees	pHuxley
'contact'_NAV_X	D	UTM x-coordinate of contact	pTransponderAIS
'contact'_NAV_Y	D	UTM x-coordinate of contact	pTransponderAIS
'contact',_NAV_HEADING	D	Current contact heading in degrees.	pTransponderAIS
'contact'_NAV_SPEED	D	Current contact speed in m/s.	pTransponderAIS
'contact'_NAV_UTC	D	UTC time of latest contact report	pTransponderAIS

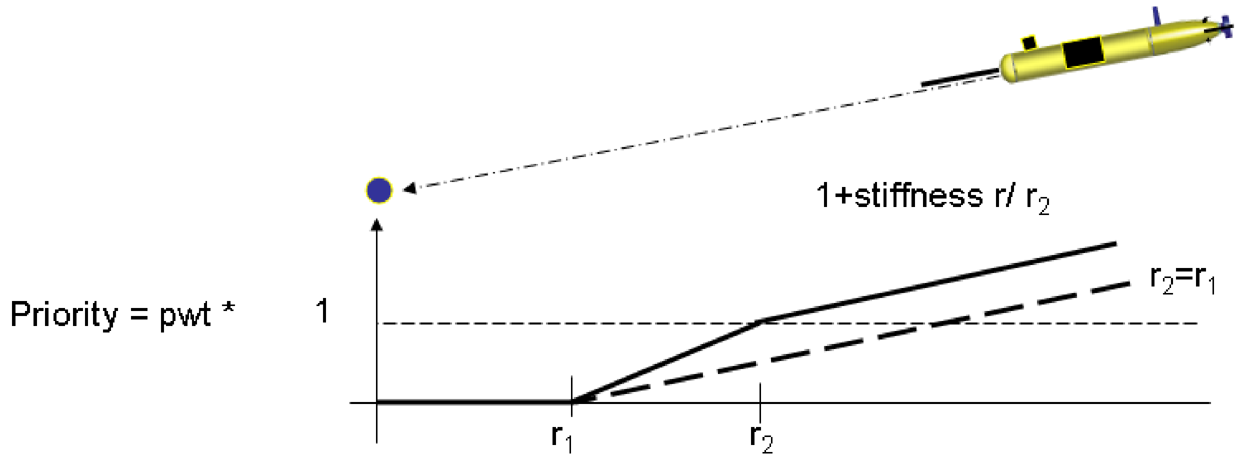

Figure 14: Range-dependent priority of BHV_RubberBand.

.

10.1.2 MOOS variables published by BHV_Attractor:

.

MOOS variable	Type	Description	Format
ATTRACTOR_RELEVANCE	D	Computed relevance in % of objective function. Published at integer precision	50
. ATTRACTOR_RANGE_TO_CONTACT	D	Range to attracting target in meters. Published at integare precision	1500

10.2 BHV_RubberBand

This behavior is very similar to BHV_StationKeep, but has a range-dependent priority, such that the restoring force back towards the station point is increasing with distance similarly to a rubber band attached to the station point. It keep the vehicle at a given lat/lon or x,y position by varying the speed to the station point as a linear function of its distance to the point. The parameters allow one to choose the two distances between which the speed varies linearly between 0 at the inner radius and the value of the parameter `OUTER_SPEED` at the outer radius. Also, the priority is increasing linearly betwen the two radii, and outside the outer radius, the priority is increasing linearly with a rate specified by the parameter `STRENGTH`, as illustrated in Figure 14. The following parameters are defined for this behavior:

`STATION_PT`: The x,y position of the point upon which to station keep. It is a comma separated pair of numerical values.

`INNER_RADIUS`: The radius from the station point within which the behavior takes no action to affect the vehicle position.

`OUTER_RADIUS`: The radius from the station point within which the behavior does take action to affect the vehicle position. It produces an objective function as if the station point were a waypoint. The desired speed varies linear between the `INNER_RADIUS` and the `OUTER_RADIUS`, approaching zero at the inner radius and the speed given by `OUTER_SPEED` at the outer radius.

`OUTER_SPEED`: The desired speed varies linear between the `INNER_RADIUS` and the `OUTER_RADIUS`, approaching zero at the inner radius and the speed given by `OUTER_SPEED` at the outer radius.

`STRENGTH`: If the vehicle is outside the outer radius, the priority will increase linearly with distance by this rate.

Listing 10.2 - An example BHV_RubberBand configuration.

```
 0  Behavior = BHV_RubberBand
 1  {
 2    name         = bhv_rubber_band
 3    station_pt   = 50,75
 4    inner_radius = 3
 5    outer_radius = 15
 6    outer_speed  = 2.2
 7    extra_speed  = 2.2
 8    condition    = ON_STATION = true
 9    condition    = RETURN = false
10  }
```

10.2.1 MOOS variables subscribed to by BHV_RubberBand:

.

MOOS variable	Type	Description	Published by
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley

10.2.2 MOOS variables published by BHV_RubberBand:

.

MOOS variable	Type	Description	Format
STATION_CIRCLE	$	Status message	1000,2000,10,bobby
DIST_TO_STATION	D	Distance to station point in meters. Published at integer precision	1200

11 Undersea Adaptive Sensing Behaviors

11.1 BHV_HArrayTurn

Used for breaking left/right ambiguity inherent to linear hydrophone arrays, this behavior will perform a sequence of clockwize and anti-clockwize turns until the signal processing module has declared that the bearing ambiguity is resolved. If this is not happening before the heading has changed by a specified amount, the behavior will switch to turning up to the same angle in the opposite direction, and so on until the ambiguity is resolved.

The configuration parameters for `BHV_HArrayTurn` are as follows.

`TURN_ANGLE:` The angle the array will be turned for breaking ambiguity before reversing direction.

Listing 11.1 - Example BHV_HArrayTurn configuration

```
1 Behavior = BHV_HArrayTurn
2 {
3   name = ambiguous_arrayturn
4   turn_angle = 120
5   pwt  = 700
6   condition = (MODE == AMBIGUOUS)
7 }
```

11.1.1 MOOS variables subscribed to by BHV_HArrayTurn:

.

MOOS variable	Type	Description	Published by
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
AEL_HEADING	D	Current mean heading of hydrophone array	pAEL
TRACK_STAT	$	Tracking state vector	p1HTracker

11.1.2 MOOS variables published by BHV_HArrayTurn:

.

None.

11.2 BHV_HArrayAngle

The bearing resolution of a linear acoustic array is best at broadside, and to optimize the bearing track solution produced by the signal processing module `pBearingTrack`, it is desirable to keep the acoustic source near broadside, as illustrated in Fig. . This behavior maintains controls the desired heading to keep the array heading, published in the `MOOS` variable `AEL_HEADING` , near broadside. This also aids the creation of a *Source Track* solution because it in general will create a curved path of the array. Because this behavior competes against the other acttive behaviors, notably the attractor behavior, the resulting array angle will be tilted towards forward endfire, but this in effect is also beneficial to the tracking since the closer the vicinity of the target, the more curvature of the vehicle path will result.

The configuration parameters for `BHV_HArrayAngle` are as follows.

DESIRED_ANGLE: The desired angle in dgrees between the hydrophone array and the estimated source direction as reported in the MOOS variable BEARING_STAT. Forward endfire corresponds to 0°, and backward endfire to 180°. The behavior creates a symmetric objective function, and the rusulting heading depends on which of the two optimal heading is closest to the current heading, and other active behaviors, such as BHV_Attractor, for example.

Listing 11.2 - Example BHV_HArrayAngle configuration

```
1 Behavior = BHV_HArrayAngle
2 {
3   name = track_array_angle
4   pwt  = 100
5   width = 60
6   desired_angle = 90
7   condition = (MODE == ADAPTIVE)
8 }
```

11.2.1 MOOS variables subscribed to by BHV_HArrayAngle:

.

MOOS variable	Type	Description	Published by
AEL_HEADING	D	Current mean heading of hydrophone array	pAEL
BEARING_STAT	$	Bearing tracker status vector	pBearingTrack pBearingsSim

11.2.2 MOOS variables published by BHV_HArrayAngle:

.

None.

11.3 BHV_SmartYoyo

This behavior adapts a vertical yoyo with specified horizontal period to the current water depth, as illustrated in Figure 15. By covering the entire water column in the survey, this behavior is also extremely useful for collecting oceanographic information. It provides for accurate control of the depth changes. Thus, the vertical survey pattern can be either linear or sinosoidal, with a controlled period and a specified number of discrete depth intervals. It will attempt to maintain a constant depth in each segment, which is useful for averaging a sensing parameter at each selected depth. In the current MIT-LAMS autonomy system, this behavior is used in the environmental sampling YOYO state using large number of segments and a relatively short range period, and in the CLASSIFY state with few discrete depths and a longer range period.

The configuration parameters for the BHV_SmartYoyo behavior are as follows.

MIN_DEPTH: Upper turning depth in meters.

MAX_DEPTH: Lower turning depth in meters.

MIN_ALTITUDE: Minimum altitude in meters. Depth yoyo will be consistently scaled to yield a lower turning depth which does not get closer to the bottom than this value.

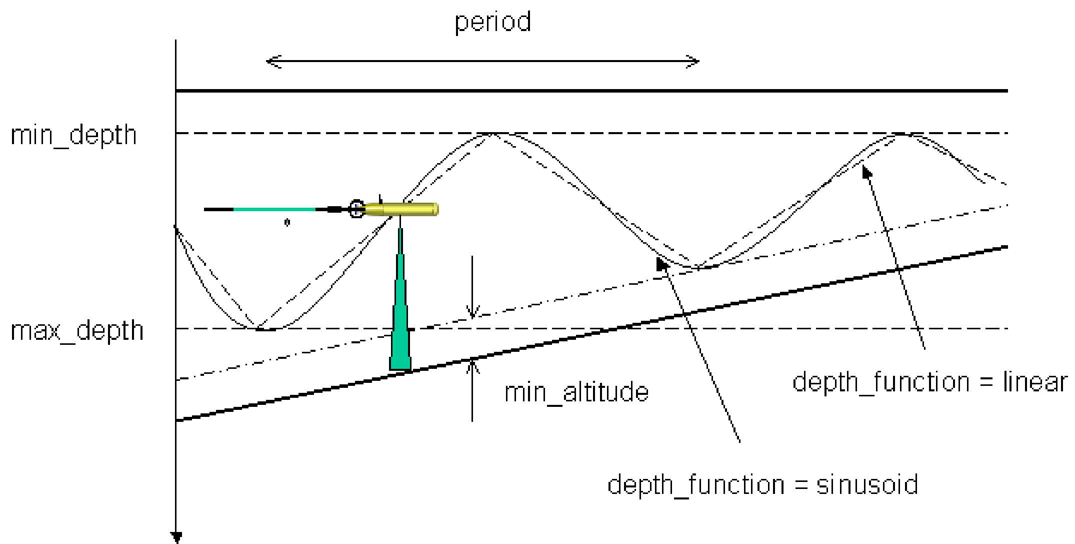

Figure 15: BHV_SmartYoyo. This behavior adapts a vertical yoyo with specific horizontal period in distance traveled to the current water depth. May be combined with arbitrary horizontal behaviors, such as loiter and zigzag.

`PERIOD`: Horizontal length of one full period of depth yoyo in meters.

`PERIOD_SEGMENTS`: Number of depth segments within a yoyo period. The desired depth will be held constant withing each segment.

`DEPTH_FUNCTION`: Shape of depth function. Choices are `linear` and `sinusoid`. Note that the actual depth profile will depend on the number of `period_segments` as well.

Listing 11.3 - Example BHV_SmartYoyo configuration

```
1 Behavior = BHV_SmartYoyo
2 {
3   name      = classify_yoyo
4   min_depth = 10
5   max_depth = 60
6   min_altitude = 5
7   period = 1200
8   period_segments = 10
```

```
 9   depth_function = linear
10   duration  = no-time-limit
11   condition = (MODE == CLASSIFYING)
12 }
```

11.3.1 MOOS variables subscribed to by BHV_SmartYoyo:

.

MOOS variable	Type	Description	Published by
NAV_DEPTH	D	Current vehicle depth in meters	pHuxley
NAV_SPEED	D	Current vehicle speed in m/s	pHuxley
NAV_ALTITUDE	D	Current altitude above bottom in meters	pHuxley
IN_TRACK	$	Logical indicating whether vehicle is on desired survey track (`TRUE`), our ourside, e.g. turning (`FALSE`). Yoyo is only active if `TRUE`, otherwize depth will be held constant.	BHV_RaceTrack

11.3.2 MOOS variables published by BHV_SmartYoyo:

.

None.

11.4 BHV_ZigZag

This behavior creates a horizontal zigzag survey, initiated at the position the behavior becomes active. It is defined by a mean heading, period, and amplitude, as depicted in Fig.16. The functional shape of the survey can be either a sawtooth pattern (linear) or a sinusoid. The horizontal survey pattern may be combined with any depth behavior. .

The configuration parameters for the `BHV_ZigZag` behavior are as follows.

`PERIOD:` Horizontal period in meters.

`AMPLITUDE:` Amplitude of zigzag i meters.

`HEADING:` Heading in degrees of zigzag baseline.

`ZIGZAG_FUNCTION:` Shape of zigzag function. Choices are `linear` and `sinusoid`.

Listing 11.4 - Example BHV_ZigZag configuration

```
 1 Behavior = BHV_ZigZag
 2 {
 3   name      = dlt_zigzag_survey
 4   speed     = 1.5
 5   updates   = PROSECUTE_HEADING
 6   period    = 600
 7   amplitude = 150
 8   heading   = 135
 9   zigzag_function = linear
10   duration  = no-time-limit
11   condition = (MODE == TMA)
12 }
```

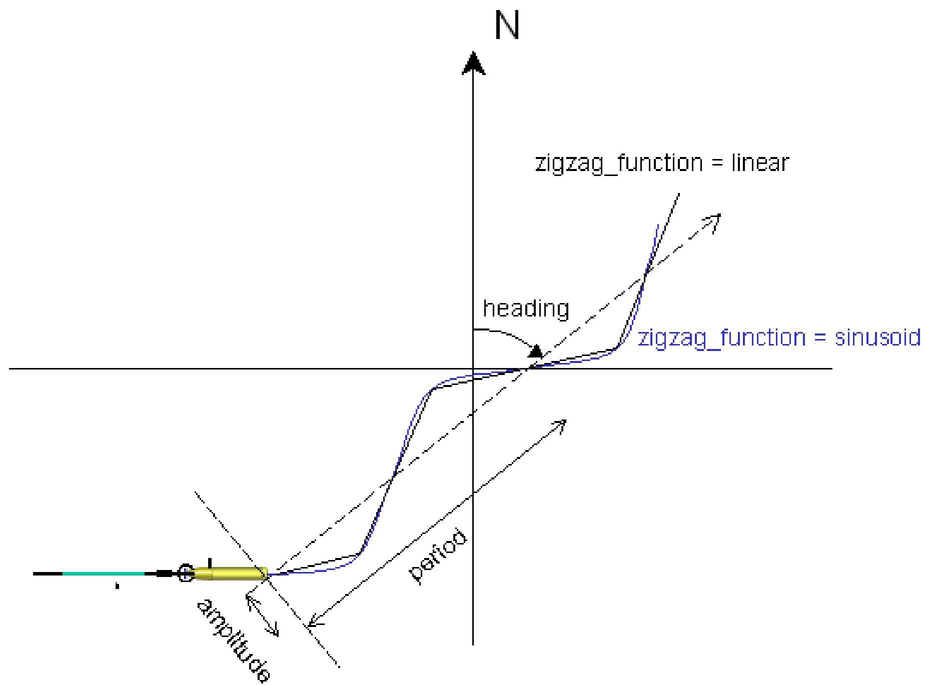

Figure 16: BHV_ZigZag: This behavior produces a horizontal zigzag survey along a specified mean heading and speed. May be combined with arbitrary vertical behaviors, such as constant depth or yoyo.

11.4.1 MOOS variables subscribed to by BHV_ZigZag:

.

MOOS variable	Type	Description	Published by
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley

11.4.2 MOOS variables published by BHV_ZigZag:

.

None.

11.5 BHV_RaceTrack

Although any horizontal survey pattern can be generated using standard behaviors, such as `BHV_Waypoint`, a dedicated racetrack behavior has been designed that emphasises a well defined trackline and a

controlled trackline approach. Also, this behavior will publish a flag indicating whwther the vehicle is on or outside the desired trackline, which is used for limiting the periodic surfacings for GPS to happen only during turns at the end of the tracklines. It will generate a survey composed of two parallel tracklines, separated by a specified distance, which may be set to zero for the two tracklines being identical.

The configuration parameters for the `BHV_RaceTrack` behavior are as follows.

`TRACK_LENGTH`: Length of the outgoing and returning tracklines in meters.

`TRACK_WIDTH`: Separation of the outgoing and returning tracklines in meters.

`DISTANCE_TOLERANCE`: Allowed deviation from the trackline in meters, beyond which the trackline will be approached at the specified approach angle. Inside this distance the approach angle will be exponentially decreasing towards the trackline. .

`APPROACH_ANGLE`: Approach angle in degrees towards the trackline, with zero being parallel to the trackline, applied outside the tolerance distance. Should typically be set to 45-60 degrees to produce effective trackline following.

`HEADING`: Absolute heading in degrees of trackline.

`SPEED`: Vehicle speed in m/s.

`TRACK_LENGTH`: Length of the outgoing and returning tracklines in meters.

`POINTS`: UTM x- and y-coordinates of racetrack starting location, comma separated.

Listing 11.5 - Example BHV_RaceTrack configuration

```
 1 Behavior = BHV_RaceTrack
 2 {
 3   name      = deploy_racetrack
 4   track_length = 2000
 5   track_width  = 0
 6   distance_tolerance  = 15
 7   approach_angle = 60
 8   heading   = 270
 9   speed     = 1.5
10   points = 500.0, 200.0
11   updates = SENSOR_RACETRACK
12   duration  = no-time-limit
13   condition = (MODE == RACETRACK)
14 }
```

11.5.1 MOOS variables subscribed to by BHV_RaceTrack:

.

MOOS variable	Type	Description	Published by
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley

11.5.2 MOOS variables published by BHV_RaceTrack:

.

MOOS variable	Type	Description	Format
IN_TRACK	$	Logical indicating whether vehicle is inside the survey track range(`TRUE`), our ourside, e.g. turning (`FALSE`).	`TRUE` `FALSE`
ON_TRACKLINE	$	Logical indicating whether vehicle is considered on the trackline, i.e. less than `distance_tolerance` from the ideal trackline (`TRUE`), our ourside, e.g. approaching trackline (`FALSE`).	`TRUE` `FALSE`

¿¿¿¿¿¿¿ .r639

Part V
Autonomous Communication, Command and Control

12 MOOS-IvP Communication, Command and Control

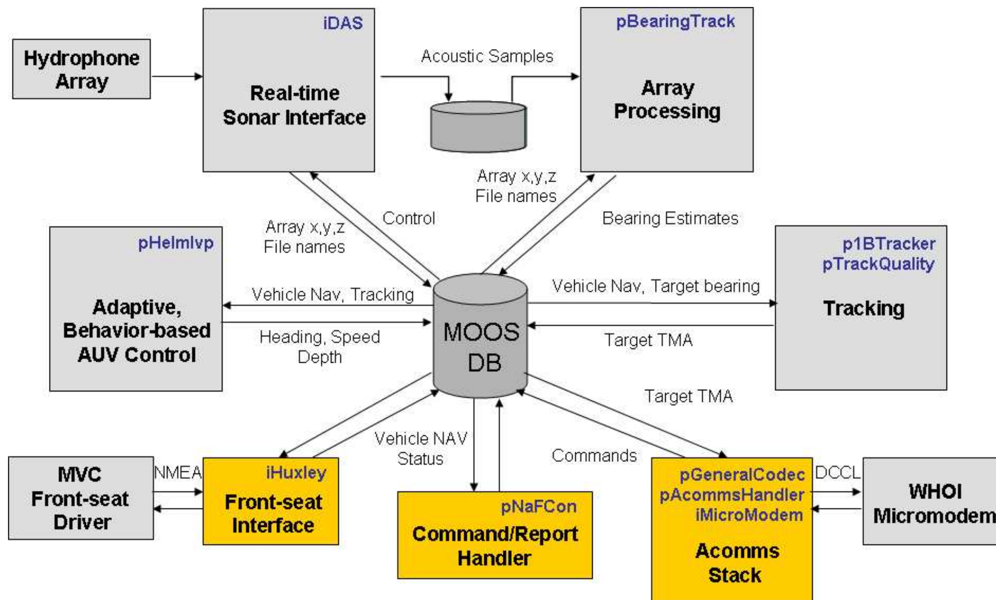

Figure 17: MOOS-IvP community for MIT sonar AUVs, with the autonomous communication, command and control modules highlighted in gold.

The Autonomous Communication, Command and Control components of the MOOS-IvP autonomy system handles the communication between the payload autonomy system, including the autonomous Helm, and the outside world.

The communication with the main vehicle computer "front-seat driver" is handled by the `pHuxley` process, responsible for passing desired speed, heading and depth as desired by the Helm to the main vehicle computer, which is then responsible for translating these commands into actuator actions. The MVC responds back with navigation information, which is then published in the `MOOSDB` for use by other modules, including the autonomy helm `pHelmIvP`.

The connection with the topside field control operators is performed using the acoustic modems, with the so-called *Communication Stack* handling the coding/decoding (CoDec)message scheduling, and modem transmission and reception.

Both of these interfaces are controlled by the AC^3 handler process, `pNaFCon`, which is responsible for translating command messages from the topside operators into state variables for the inboard autonomy system, and for translating the current vehicle states into status reports for broadcasting

to the network. Similarly, the state of the vehicle is translated into commands for the "front-seat driver", such as disabling and enabling the "back-seat driver" control.

A number of other processes are integrated into the prototype autonomy system for managing the state transitions in accordance with the nested autonomy CONOPS.

A list of autonomous communication, command and control processing modules is given in Table 1. A more detailed description is given in the following.

12.1 Node Level Command and Control

In the *Nested Autonomy* paradigm the node level Command and Control is highly tilted towards autonomy, with the vehicle actions being defined by the current vehicle state and the associated behavior set, and the sensor input. The possibility of the cluster and the field control to alter the node behavior is limited to simple, higher level commands, the only role of which is to transition the vehicle to another, allowed state. Thus, for example, if the *NaFCon* operators want to alter the vehicle behavior during a prosecute mission, their only option is to transition the vehicle to another allowed state, defined in the configuration file. The detailed actions such as heading, speed and depth of the node are not directly controllable by the operators. For example, the operators have the power to override an active prosecute mission by transmitting a *Deploy Command* or switching the node to an alternative *Prosecute State*.

Similarly, the state of the node may be altered by other nodes in the cluster only if allowed in the mission configuration on the node. For example, in the MIT Nested Autonomy Prototype, the process `pTargetOpportunity` will react to incoming *Track Reports* by issuing itself a *Prosecute Command* if the target is predicted to come within an allowed perimeter.

12.2 Cluster Level Command and Control

Depending on the CONOPS, the control of the Cluster may be performed either centrally from *NaFCon*, by a 'leader node' in the cluster controlling the states of the 'cluster team', or fully distributed. A team sports analogy of the centralized control is American football, with its reliance on real-time communication infrastructure. In contrast, team sports such as basketball and soccer rely on each individual player making decisions based on his own situational awareness, strategy and tactics.

In the MIT *Nested Autonomy Prototype* the fully distributed option has been chosen, in large part because of the high level of autonomy made available by the `IvP Helm` , and the latent and intermittent undersea communication infrastructure.

An example of the distributed cluster control is the *Synchronized Swimming* activated in the `TRAIL` sub-state. Here the trailing vehicle will use the `BHV_Trail` behavior to maintain a fixed relative or absolute bearing, and range to a collaborating node, simply by monitoring the status reports it broadcasts. Because of the latency and intermittency, the extrapolation feature of the trail behavior is crucial to the performance. Another important feature is a limit on the validity of the messages received from the other vehicle. Thus, if the collaborating node has not been heard from for a specified time, the extrapolation of its position will be ignored, and the trailing node will enter a local loiter behavior until a fresh status report from the other vehicle is received, and the trailing resumed.

Another distributed cluster control feature of the prototype is the collaborative search and tracking behaviors activated in response to a *Prosecute Command*. Again these behaviors are car-

#	Module Name	Module Description	Author	Size
1	**pNaFCon**	Manages translation of commands from field control to autonomy state, and conversely translates autonomy state to status reports. Manages "back-seat driver" control *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Balasuriya	
2	**pMessageSim**	Used to make initial deploy of vehicle in cases where acoustic communication is not applicable on surface. Optionally makes prosecute command as well. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Eickstedt	
3	**pSearch**	State transition manager for acoustic source tracking CONOPS. Responsible for managing the transition into the *Prosecute State* in response to a *Prosecute Command*. Re-deploys if target not interceptable. *Libraries: MOOS, MOOSGen, MOOSUtility*	Eickstedt	
4	**pTrackQuality**	State transition manager for acoustic source tracking CONOPS. Responsible for terminating the *Prosecute State*. Also issues redeploy before vehicle reaches operational limits for all states. *Libraries: MOOS, MOOSGen, MOOSUtility*	Schmidt	
5	**pTargetOpportunity**	State transition manager for acoustic source tracking CONOPS. Transitions autonomously from *Deploy* to *Prosecute State*. Monitors incoming *Track reprorts* and issues *Prosecute Command* to ownship if target can be intercepted *Libraries: MOOS, MOOSGen, MOOSUtility*	Schmidt	
6	**pClusterPriority**	This process weights the priority to track a target (using BHV_Attractor) by how close the vehicle is to the target relative to all the collaborating nodes. Closer vehicles are given higher weights, thus leading to a "zone defense" style of multi-vehicle autonomy. The last piece to this "zone defense" scheme is the BHV_RubberBand, which attracts vehicles back to their initial deploy location which increasing strength with distance. *Libraries: tes_util, mbutil, MOOS, MOOSGen, MOOSUtility*	Schneider	
7	**pHuxley**	Does something *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Balasuriya	
8	**pGeneralCodec**	Highly extensible encoder / decoder of messages from human readable integers, booleans, strings, or floats to hexidecimal strings suitable for sending by the WHOI MicroModem. *Libraries:tes_util, MOOS, MOOSGen, MOOSUtility*	Schneider	
9	**pAcommsHandler**	Manages queues for transmission of acoustic messages. Different message queues can be given priorities that increase with time since the last message was sent from that queue. *Libraries: tes_util, MOOS, MOOSGen, MOOSUtility*	Schneider	
10	**pAcommsPoller**	Manages and schedules polling of other nodes on the modem network. *Libraries: tes_util, MOOS, MOOSGen, MOOSUtility*	Schneider	
11	**iMicroModem**	Driver process for WHOI micromodems. *Libraries: tes_util, MOOS, MOOSGen, MOOSUtility*	Grund	
		Total unique lines of code		
		Total aggregate lines of code		

Table 1: Autonomous Communication, Command and Control Modules for MOOS-ivP onboard autonomy system.

ried out without any need for two-way *handshake* communication with the other cluster nodes. The paradigm philosophy is that any useful information from other nodes is fused into the situational awareness, but the individual nodes are not dependent on any outside information for pursuing the mission objective. A characteristic example is the dynamic, distributed assignment of priorities of the cluster behaviors `BHV_Attractor` and `BHV_RubberBand`, with the first attempting to attract the node towards the cued target position and the other trying to maintain the node near the assigned station point. In the sports analogy, the command and control process `pClusterPriority` implements an autonomous *zone defense* strategy by dynamically assigning priorities based on the known position of the other cluster assets and the cued target position. Without requiring any explicit coordination and communication with the other nodes, each node will therefore on its own assign priority to the two competing behaviors. As a result, the cluster node closest to the target will most aggressively initiate pursuit. On the other hand, the other nodes are not sitting idle, but pursuing the target with some finite priority, thus ensuring that they are in a position to more actively initiate the search in case the closest node fails to detect the target or never reacted because it did not receive the cuing message. Again, the soccer analogy is obvious, with all defenders moving in position to react if a forward from the other team is passing by the first line of defense. The actual behaviors associated with this distributed 'zone defense' is never commanded directly, but built into the relative priorities of the competing behaviors allocated by `pClusterPriority`. Based on a large number of virtual and field experiments a range-exponential priority allocation appears optimal.

In addition to the cluster command and control components of the on-board processing also adopts the distributed decision paradigm. For example, the fusion of bearing tracks received from other platforms are applied on collaborating nodes if deemed useful, e.g. for producing a geo-referenced target track (`pMBTracker`), but the nodes are not dependent on it, and will continue to attempt a single node tracking sequence (`p1HTracker`) if it does not receive useful information from other nodes.

12.3 Field Level Command and Control

Although a large amount of autonomous command and control is delegated to the nodes the field level operators obviously have the ultimate control authority with the power to change the roles of the various clusters and nodes under its command. However, in the *Nested Autonomy* paradigm this authority can only be exercised through simple commands that trigger a state transition on the network nodes. The ultimate action associated with the issued commands is entirely defined in the `MOOS-IvP` configurations on the nodes.

The commands for the network nodes are encoded into *Compact Control Language* (CCL) messages which are then transmitted via the acoustic modem network infrastructure. In the *PLUSNet* prototype a limited message set was hard coded into a couple of dedicated CCL messages. This message set was very restrictive, allowing only a handful of state transitions. Thus, to enable a certain *Deploy* or *Prosecute* mode in was necessary to change the configuration files before the vehicle was deployed. To allow an arbitrarily large state space to be activated without recovery and re-configuration, an new message handler, coder and decoder stack was developed in 2008 by Toby Schneider at MIT. Most of the stack was tested successfully in the August 2008 GLINT'08 experiment. The new software stack was completed in the fall of 2008 with the development of the completely generic coder-decoder (CoDec) `pGeneralCodec`. it uses a dedicated CCL message,

assigned number 32, with a highly flexible message format that allows for optimal compression and content, specified in a configuration file which may be changed to optimally fit the specific geographical region and mission suite. The configuration files for the MIT *Nested Autonomy Prototype* are descried in Appendix A.

In an actual operating system the command and control CCL messages may be generated either by the operators directly, or by a topside command and control autonomy system, e.g. linked to an environmental forecasting framework. In the MIT prototype the commands are issued by the operator, typically the Chief Scientist or a trained pilot using the topside MATLAB GUI, `NaFConSim.m`. In combination with the new CCL software stack, it allows the operators to a activate any of the autonomy states without reconfiguration.

The `NaFConSim` GUI is currently configured for issuing *Deploy Commands* with 7 allowed *Mission Types*:

0 Transitions the node to the `SENSING` state, in the current configuration a hexagonal loiter with the on-board sensing and processing active.

1 Transition to `LOWPOWER` state. In the prototype a hexagonal loiter around the deploy location.

2 Transition to `OFF` state. Currently the vehicle transits to the deploy location and surfaces.

3 Transition to `RETURN` state. In the prototype identical to the `OFF` state.

4 Transition to `RACETRACK` state. In prototype configuration the vehicle will run a zero-width racetrack at the specified heading and length, starting at the deploy location.

5 Transition to `YOYO` state. In prototype the vehicle will run the same racetrack as for mission 4, but executing a vertical YoYo survey using the `BHV_SmartYoyo` behavior.

6 Transition to `ZIGZAG` state.

Similarly, the GUI is configured for selecting 4 *Prosecute Mission Types*. All the mission types initially transitions to the `SEARCH` state, followed by the `AMBIGUOUS` state once the cued target event is detected. Once the ambiguities are resolved. the node will transition into one of the `TRACKING` sub-states, depending on the mission type,

1. Transition to `ADAPTIVE` tracking state, where the vehicle will adaptively maneuver to optimize the tracking performance.

2. Same as 1, but once tracking is deemed satisfactory by the `pTrackQuality` process, it will enter the `CLASSIFYING` state, which in the prototype will maintain a heading pointing towards the computed target track.

3. Transition to `TMA` state, where the vehicle in the prototype will enter a zig-zag survey along a specified heading.

4. Transition to a hexagonal loiter similar to the one used in the `SENSING` *Deploy Mission*.

As mentioned earlier, the number of *Deploy* and *Prosecute* commands the communications stack can handle is unlimited. The current suite is dictated only by the capabilities built into the GUI. Thus for example, the `TRAIL` state used for *Synchronized Swimming* can currently only be activated by changing the configuration, e.g. by replacing one of the other deployment modes. This and other useful transitions will be enabled in the next release of the topside command GUI.

13 Communication, Command and Control Modules

13.1 pNaFCon

13.1.1 Brief Overview

pNaFCon is the MOOS process that provides the connectivity between the network communication infrastructure and the autonomy states of the vehicle. As such it is the principal link between the communication infrastructure and the pHelmIvP behavior-based autonomy. Thus, it subscribes to the command messages from the topside and status and contact reports from other nodes and performs the associated state transitions by posting a set of state variables. Thus for example, in response to a topside Deploy Command the MOOS variables DEPLOY_STATE is published by pNaFCon to place the vehicle in its Deploy state, with DEPLOY_MISSION defining the deploy sub-states, such as a low-power drift, a loiter pattern, or an environmental sampling mission with a YoYo in depth along a horizontal racetrack. Similarly, pNaFCon publishes the variables PROSECUTE_STATE and PROSECUTE_MISSION to transition the vehicle autonomy into a particular Prosecute sub-state, such as adaptive target tracking or a traditional TMA ZigZag survey, for example.

NaFCon also is responsible for generating Status and Contact reports, depending on the state of the vehicle. Thus, for example it will respond to a target bearing being published by the real-time data processing by creating a contact Report message, which will then be queued, scheduled and transmitted bythe acoustic communication stack.

13.1.2 Parameters for the pNaFCon Configuration Block

The following configuration parameters are defined for pNaFCon.

ProsecuteMission: Default rosecute mission: 1: Adaptive DLT; 2: Adaptive DLT with subsequent depth-discremination clasification. This parameter used to be necessary since the pFramer CoDec used until recently did not allow more than a single type of Prosecute mission to be commanded via the acoustic link. When using the new generic CoDec pGeneralCoDec this parameter is obsolete and will be eliminated in future versions.

etc etc

Below is an example configuration block for pNaFCon,

Listing 13.1 - An example pNaFCon *configuration block .*

```
 1 /////////// Global Variables ////////////////
 2 LatOrigin = 36.87
 3 LongOrigin = -122.42
 4 MagneticOffset = 0.0
 5 ////////Configuration Block for pNaFCon ///////////////
 6 ProcessConfig = pNaFCon
 7 {
 8   AppTick   = 4
 9   CommsTick = 4
10   ProsecuteMission = 1
11   DestinationPlatformId = 3
12   CollaboratingPlatformId = 4
13   TrackId = 31
14   Orbit_radius =300
15   Orbit_points = 6
```

```
16   ZigZag_Period = 300
17   ZigZag_Amplitude = 150
18   Report_Delay = 2;
19 }
```

The `LatOrigin` and `LonOrigin` parameters on lines 1-2 are not specific to `pNaFCon`, but are specified at the global level in a MOOS file. They are needed to allow the conversion between local x-y coordinates and latitude-longitude coordinates. They are mandatory parameters and `pNaFCon` will refuse to launch if they are lacking - but they are mandatory for other common MOOS applications as well.

13.1.3 MOOS variables subscribed to by pNaFcon:

.

MOOS variable	Type	Description	Published by
NAV_X	D	Local UTM x-coordinate for ownship	pHuxley
NAV_Y	D	Local UTM y-coordinate for ownship	pHuxley
NAFCON_MESSAGES	S	Incoming acomm messages	

13.1.4 MOOS variables published by pNaFCon:

.

MOOS variable	Type	Description	Format
DEPLOY_STATE	$	Defines DEPLOY state.. “FALSE”: Vehicle not in DEPLOY state “OFF”: Vehicle is idleing “DEPLOY”: Vehicle in DEPLOY state “ABORT”: Vehicle transiting to abort point	
MICROMODEM_COMMAND	$	Polling a Collaborating node	
POLL_REQUEST	D	Poll request flag	
COLLABORATOR_ID	$	Collaborating id from moos file	
SINCE_POLLED	D	Time since last poll	
BEARING_STAT	D	Initialize BEARING_STAT = 0	
REMOTE_COLLABORATION	D	State of the collaborating node ”OFF”: No collaboration ”SYNCH”: In synch ”COLLABORATING”: In collaborating mode	
COMMUNITY_STAT	$	Status of the collaborating vehicle	
BEARING_STAT2	$	Contact info. of the collaborating vehicle	
TRACK_STAT2	$	Track info. of the collaborating vehicle	
GOTO_SURFACE	$	Goto surface flag ”TRUE” or ”FALSE”	
DEPLOY_STATION	$	Station keeping position	
TARGET_ID	$	Track number for prosecuted target	5
TARGET_STATE	$	Cued target location and source characteristics for use by target simulators	tgt_x=2000, tgt_y=1525, ... tgt_id=5
TGT_id_NAV_X	$	Cued UTM x coordinate for id’th target	
TGT_id_NAV_Y	$	Cued UTM y coordinate for id’th target	
TGT_id_NAV_HEADING	$	Cued Heading of id’th target	
TGT_id_NAV_SPEED	$	Cued speed id’th targets	
TGT_id_NAV_UTC	$	UTC time of cued target info	
MY_TARGET	$	Current target of interest	TGT_5
CURRENT_TARGET	$	Current target id	
PLUSNET_MESSAGES	$	Status and contact messages to be broadcast by modem	

13.2 pMessageSim

13.2.1 Brief Overview

This MOOS module provides an automated procedure for launching a vehicle into an initial *Deploy* state at some specified delay following launch. It does so by simply posting a `NAFCON_MESSAGES` in the MOOSDB in the same way as `pNaFCon`. It is particularly useful in cases where the conditions do not support the launching via the acoustic modem. In fact it was originally written by D. Eickstedt at sea in the MB06 trial in Montery Bay, where the communication stack was not yet robust enough to provide reliable cammanding of the vehicles. `pMessageSim` also provides the optional launching of a delayed *Prosecute* command to the vehicle itself. After the acomms stack has become further developed, this option is rarely used.

13.2.2 Parameters for the pMessageSim Configuration Block

The following configuration parameters are defined for `pNaFCon`.

`message_type:` *Deploy* mission type. Action depends on settings in behavior configuration file. Current mission suite is

- 0 Continuous DCL hex-loiter deployment: Sensors and processing turned on
- 1 Low-power deployment.
- 2 Off. Go to surface
- 3 Return to station and surface
- 4 Environmental mission. Racetrack
- 5 Evironmental mission. Adaptive vertical yoyo. Horizontal racetrack
- 6 Environmental mission. Horizontal zigzag, constant depth.

`trigger_depth:` Depth at which delay timer for deploy message is triggered. If negative triggering will occur at start-up..

`deploy_delay:` Delay in seconds of deploy message. If not specified, default is 30 seconds.

`prosecute_delay:` Delay in seconds of prosecute message. In most cases set to value larger than mission time to disable autonmatic prosecute message.

`deploy_x:` UTM x-coordinate of deployment point.

`deploy_y:` UTM y-coordinate of deployment point.

`deploy_depth:` Depth of deployment.

`dest_addr:` Destination platform ID. Here Ownship vehicle ID.

`abort_x:` UTM x-coordinate of abort point. When aborting the vehicle will transit to this point at depth and then surface..

abort_y: UTM y-coordinate of abort point.

abort_depth: Depth at which vehicle will transit to abort point.

op_radius: Operations radius allowed for vehicle relative to current deploy location. If deploy command issued to a location outside this range, the state manager process, currently pTrackQuality, will issue a local deploy.

deploy_duration: Duration in seconds of deploy state. If exceeded without redeploy, vehicle will enter ABORT state and transit to abort location and surface.

target_x: UTM x-coordinate of target to be prosecuted after PROSECUTE_DELAY.

target_y: UTM y-coordinate of target to be prosecuted.

target_depth: Depth of target to be prosecuted.

target_heading: Heading of target to be prosecuted.

target_speed: Speed of target to be prosecuted.

target_id: Identification number allocated to target to be prosecuted.

prosecute_duration: Duration in seconds of prosecute state. If exceeded without redeploy, vehicle will enter ABORT state and transit to abort location and surface.

yoyo_distance: Length of racetrack legs for *Deploy Mission* 4 and 5, and length of zigzag projection on average heading for *Deploy Mission* 6.

yoyo_heading: Heading of racetrack legs for *Deploy Mission* 4 and 5, and average heading for *Deploy Mission* 6.

yoyo_upper: Upper turning depth of vertical yoyo.

yoyo_lower: Lower turning depth of vertical yoyo.

Below is an example configuration block forpMessageSim,

Listing 13.2 - An example pMessageSim *configuration block .*

```
1 /////////// Global Variables ////////////////
2 LatOrigin = 36.87
3 LongOrigin = -122.42
4 MagneticOffset = 0.0
5 ////////Configuration Block for pMessageSim //////////////
6 ProcessConfig = pMessageSim
7 {
8  AppTick    = 4
9  CommsTick  = 4
10
11  message_type = 1
12  trigger_depth = -1
```

```
13  prosecute_delay = 60000
14  deploy_x = 5000 // 500 A6 racetrack
15  deploy_y = 9000 // 2100 A6 racetrack
16  deploy_depth = 25.0
17  dest_addr = 3
18  abort_x = 4500
19  abort_y = 8500
20  abort_depth = 6.0
21  op_radius = 5000
22  deploy_duration = 7200
23  target_x = 0
24  target_y = 1000
25  target_depth = 20.0
26  target_heading = 90.0
27  target_speed = 2.0
28  target_id = 5
29  prosecute_duration = 1800
30  yoyo_distance = 4000
31  yoyo_heading = 135
32  yoyo_upper = 5
33  yoyo_lower = 30
34 }
```

The `LatOrigin` and `LonOrigin` parameters on lines 1-2 are not specific to `pMessageSim`, but are specified at the global level in a MOOS file. They are needed to allow the conversion between local x-y coordinates and latitude-longitude coordinates. They are mandatory parameters and `pMessageSim` will refuse to launch if they are lacking - but they are mandatory for other common MOOS applications as well.

13.2.3 MOOS variables subscribed to by pMessageSim:

.

MOOS variable	Type	Description	Published by
NAV_DEPTH	D	Current ownship depth. Used fro triggering message	pHuxley

13.2.4 MOOS variables published by pMessageSim:

.

MOOS variable	Type	Description	Format
NAFCON_MESSAGES	$	Deploy or prosecute message for ownship.	

13.3 pSearch

13.3.1 Brief Overview

pSearch is a transition manager for the transition from *Deploy State* to *Prosecute State* in response to a *Prosecute Command* issued either by *NaFCon* , pMessageSim or the target watchdog process pTargetOpportunity described later.

Once a NAFCON_MESSAGES command is reived from the MOOSDB, pSearch will first detemine whether the target is interceptable within the specified distance. If not, it will return the vehicle to its earlier *Deploy State*. If the target is interceptable it will continuously monitor the target and ownship motion, checking for continued interceptability. If the minimum distance to the expected target position is reached with detection, *pSearch* will enter a local loiter pattern iwth the sensors remaing active for possible detection of the target event.

13.3.2 Parameters for the pSearch Configuration Block

min_range: Range from predicted target position at which vehicle will enter waiting loiter pattern until possibly detecting the target event.

orbit_radius: Radius applied in waiting loiter hexagon awaiting possible event detection

giveup_range: Target range in meters beyond which the prosecute will not be pursued. Vehicle will remain in or return to *Deploy State*

Below is an example configuration block forpSearch,

Listing 13.3 - An example pSearch *configuration block .*

```
 1 /////////// Global Variables ////////////////
 2 LatOrigin = 36.87
 3 LongOrigin = -122.42
 4 MagneticOffset = 0.0
 5 ////////Configuration Block for pSearch ///////////////
 6 ProcessConfig = pSearch
 7 {
 8   AppTick   = 4
 9   CommsTick = 4
10
11   min_range = 500
12   orbit_radius = 300
13   giveup_range = 2500
14 }
```

13.3.3 MOOS variables subscribed to by pSearch:

.

MOOS variable	Type	Description	Published by
VEHICLE_ID	D	Node number for ownship.	pNaFCon
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
PROSECUTE_STATE	$	Prosecute state	pNaFCon
TARGET_STATE	$	Assumed target state vector from *Prosecute Command*	pNaFCon
SENSOR_DEPLOY	$	Current *Deploy* location	pNaFCon
SENSOR_DEPTH_DEPLOY	$	Current *Deploy* depth	pNaFCon
DEPLOY_RADIUS	D	Maximum operating radius in meters	pNaFCon
ABORT_LAT	D	Abort location latitude in decimal degrees	pNaFCon
ABORT_LONG	D	Abort location longitude in decimal degrees	pNaFCon
ABORT_TIME	D	UTC Abort time	pNaFCon
SENSOR_DEPTH_ABORT	D	Depth ofr transit to abort location	pNaFCon
TRACKING	$	Tracking state. Indicates whether target has been detected or not.. "NO_TRACK": No detection yet "AMBIGUOUS": Detection, but ambiguous bearing "TRACKING": Un-ambiguous bearing tracking "CLASSIFYING": Classifying sub-state	pBearingTrack

13.3.4 MOOS variables published by pSearch:

.

MOOS variable	Type	Description	Format
NAFCON_MESSAGES	$	Deploy for ownship published if the intercept conditions for a reported target are satisfied.	
. TRACK_STAT	$	Comma separated track state vector. Initializes to state=0. If target not detected the target state variable is set to state=4.	node=3,state=0,x=...
IN_RANGE	$	Flag indicating whether vehicle is within minimum range to expected target position.	TRUE FALSE

13.4 pTrackQuality

13.4.1 Brief Overview

This is the principal autonomus state transition managing process responsible for terminating the *Prosecute State* for acoustic source tracking missions. It is monitoring a suite of MOOS variables and performs state transitions accordingly. It's main objective is to terminate a source tracking *Prosecute Mission*, and return the vehicle to its default *Deploy State*. The transition is triggered either by the tracking solution with satisfactory uncertainty being achieved, or the maximum number of *Track Reports* is reached.

`pTrackQuality` is also monitoring the distance to the boundaries of the operations box and the distance from the station point, and will issue a redeploy if too close, to avoid the *Surfacing* triggered by the helm if the boundary is exceeded. This feature is active in both the *Deploy* and *Prosecute* states.

13.4.2 Parameters for the pTrackQuality Configuration Block

`bearing_rate_threshold:` If positive sets the threshold for terminating source tracking. When bearing rate decreases below the set percentage of the max imum bearing rate. Can produce undesired terminations if set too high. Rarely used.

`track_limit:` Maximum duration of *Tracking Sub-State* in seconds. When exceeded, vehicle will be re-deployed.

`min_opbox_distance:` Minimum allowed distance to operations box boundary. Triggers re-deploy when reached to avoid 'hard' abort by Helm.

`max_classify_time:` Maximum duration of *Classify Sub-State* of source tracking prosecute in seconds. When exceeded, vehicle will be re-deployed.

`track_sigma_single:` Standard deviation threshold for single vehicle track solution in meters. `TRACK_REPORT` will be issued when computed standard deviation is less than this value.

`buffer_length_single:` Number of single vehicle track solutions used in computing track uncertainty.

`track_sigma_multiple:` Standard deviation threshold for multi-vehicle, collaborative track solution in meters. `TRACK_REPORT` will be issued when computed standard deviation is less than this value.

`buffer_length_multiple:` Number of multi-vehicle, collaborative track solutions used in computing track uncertainty.

`track_report_time:` Duration of time period where *Track Reports* are issued. When exceeded, vehicle will transition to *Deploy State*.

`deploy_location:` Location of re-deploy. Can be ither 'station' or 'present', for return to station point or preset position, respectively.

Below is an example configuration block forpTrackQuality,

Listing 13.4 - An example pTrackQuality *configuration block .*

```
 1 /////////// Global Variables ////////////////
 2 LatOrigin = 36.87
 3 LongOrigin = -122.42
 4 MagneticOffset = 0.0
 5 ////////Configuration Block for pTrackQuality ///////////////
 6 ProcessConfig = pTrackQuality
 7 {
 8   AppTick = 1
 9   CommsTick = 1

10   bearing_rate_threshold = 0  // percent of max achieved during tracking
11   track_limit = 600           // Tracking duration in seconds
12   min_opbox_distance = 500    // min distance to opreg perimeter
13   max_classify_time  = 600    // max Classify time for PROSECUTE_MISSION = 2
14   track_sigma_single = 100
15   buffer_length_single = 10
16   track_sigma_multiple = 100
17   buffer_length_multiple = 3
18   track_report_time = 180
19   deploy_location = station   // Options: station or present
20 }
```

The LatOrigin and LonOrigin parameters on lines 1-2 are not specific to pMessageSim, but are specified at the global level in a MOOS file. They are needed to allow the conversion between local x-y coordinates and latitude-longitude coordinates. They are mandatory parameters and pMessageSim will refuse to launch if they are lacking - but they are mandatory for other common MOOS applications as well.

13.4.3 MOOS variables subscribed to by pTrackQuality:

.

MOOS variable	Type	Description	Published by
VEHICLE_ID	D	Node number for ownship.	pNaFCon
TGT_#_NAV_X	D	UTM x-coordinate of target. Initially from *Prosecute Command*. Updated by tracking solution	pNaFCon p1HTracker pMBTracker
TGT_#_NAV_Y	D	UTM x-coordinate of target. Initially from *Prosecute Command*. Updated by tracking solution	pNaFCon p1HTracker pMBTracker
TGT_#_NAV_UTC	D	UTC time of target location	p1HTracker pMBTracker
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
SENSOR_DEPLOY	$	Current *Deploy* location	pNaFCon
SENSOR_DEPTH_DEPLOY	$	Current *Deploy* depth	pNaFCon
PROSECUTE_STATE	$	Prosecute state	pNaFCon
PROSECUTE_MISSION	$	Prosecute mission type	pNaFCon
BEARING_STAT	$	Comma-separated bearing state variables: vehID,state,bearing,x,y,beamno,sigma,utc	pBearingTrack
TRACK_STAT	$	Tracking state vector	p1HTracker
TRACKING	$	Tracking state. "NO_TRACK": No detection yet "AMBIGUOUS": Detection, but ambiguous bearing "TRACKING": Un-ambiguous bearing tracking "CLASSIFYING": Classifying sub-state	pBearingTrack
TRACKING_MODE	$	Tracking mode: "SINGLE": Single vehicle tracking "MULTIPLE": Multi-vehicle cross-bearing tracking	pMBTRacker
CLOSE_RANGE	$	Reset flag for geo trackers	pSearch
CLASS_DONE	$	Flag set to `TRUE` when classification substate is completed.	pTrackQuality
. DEPLOY_RADIUS	D	Maximum operating radius in meters	pNaFCon
OPREG_TRAJECTORY_PERIM_DIST	D	Current distance to operations box in meters	pHelmIvP
ABORT_LAT	D	Abort location latitude in decimal degrees	pNaFCon
ABORT_LONG	D	Abort location longitude in decimal degrees	pNaFCon
ABORT_TIME	D	UTC Abort time	pNaFCon
SENSOR_DEPTH_ABORT	D	Depth ofr transit to abort location	pNaFCon

13.4.4 MOOS variables published by pTrackQuality:

.

MOOS variable	Type	Description	Format
NAFCON_MESSAGES	$	Deploy for ownship published if any of the prosecute termination conditions are satisfied.	
. TRACK_REPORT	$	*Track Report* issued when uncertainty satisfactory. Translated into modem message by `pNaFCon`	
TRACKING	$	Tracking substate flag. Changed to "CLASSIFYING" if `PROSECUTE_MISSION = 2` and tracking is satisfactory	
CLASS_DONE	$	Flag set to `TRUE` when classification substate is completed.	
.			

13.5 pTargetOpportunity

13.5.1 Brief Overview

`pTargetOpportunity` is a transition manager that autonomously transitions from *Deploy State* to *Prosecute State* if a *Track Report* is received from a collaborating vehicle for an event, such as an acoustic source or a plume, is heading in a direction that allows ownship to intercept. It currently represents the highest level of autonomous decisionmaking on the vehicle.

When a *Track report* is received from the collaborting vehicle the closest point of approach is calculated, and the prosecute command will be issued only if the CPA is les than a maximum range specified in the configuration file.

There are currently one of two algorithms apllied, dependent on the compiler settings. The ultimate one uses the standard MOOS-IvP CPA engine, but this algorithm needs debugging. The other uses a simple minimum distance computation ignoring the estimated target speed. At this point both set of configurationj parameters are specified.

13.5.2 Parameters for the pTargetOpportunity Configuration Block

`CollaboratorNodes:` Comm-separated list of vehicle ID for trusted collaborator nodes. Only *Track Reports* from nodes on the list will be acted upon.

`OpportunityMission:` *Prosecute Mission* to be applied:

1. Adaptive tracking mission
2. Adaptive tracking with subsequent classification sub-state
3. Zig-Zag mission at estimated target heading
4. Loiter deploy pattern ith tracking.

`OpportunityRadius:` Minimum target distance, below which a *Prosecute Command* is issued to ownship.

`VehicleMaxSpeed:` Maximum estimated speed of target to be intercepted

`ThresholdCPA:` Threshold range for CPA below which a prosecute sequence will be initiated

`ThresholdTOL:` Tolerance of the CPA determination.

Below is an example configuration block for`pTargetOpportunity`,

Listing 13.5 - An example `pTargetOpportunity` *configuration block .*

```
1 /////////// Global Variables ////////////////
2 LatOrigin = 36.87
3 LongOrigin = -122.42
4 MagneticOffset = 0.0
5 ////////Configuration Block for pTargetOpportunity //////////////
6 ProcessConfig = pTargetOpportunity
7 {
8   AppTick = 4
9   CommsTick = 4
```

```
10
11   CollaboratorNodes = 0,4
12   OpportunityMission = 1
13   OpportunityRadius = 2000
14   VehicleMaxSpeed = 2.0
15   ThresholdCPA = 1000
16   ThresholdTOL = 20
17 }
```

The `LatOrigin` and `LonOrigin` parameters on lines 1-2 are not specific to `pMessageSim`, but are specified at the global level in a MOOS file. They are needed to allow the conversion between local x-y coordinates and latitude-longitude coordinates. They are mandatory parameters and `pMessageSim` will refuse to launch if they are lacking - but they are mandatory for other common MOOS applications as well.

13.5.3 MOOS variables subscribed to by pTargetOpportunity:

.

MOOS variable	Type	Description	Published by
VEHICLE_ID	D	Node number for ownship.	pNaFCon
COLLABORATOR_ID	D	Node number for collaborating vehicle	pNaFCon
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
NAV_DEPTH	D	Current ownship depth in meters	pHuxley
DEPLOY_STATE	$	Deploy state	pNaFCon
PROSECUTE_STATE	$	Prosecute state	pNaFCon
TRACK_STAT2	$	Tracking state vector received from collaborating vehicle	pNaFCon
DEPLOY_RADIUS	D	Maximum operating radius in meters	pNaFCon
OPREG_TRAJECTORY_PERIM_DIST	D	Current distance to operations box in meters	pHelmIvP
ABORT_LAT	D	Abort location latitude in decimal degrees	pNaFCon
ABORT_LONG	D	Abort location longitude in decimal degrees	pNaFCon
ABORT_TIME	D	UTC Abort time	pNaFCon
SENSOR_DEPTH_ABORT	D	Depth ofr transit to abort location	pNaFCon

13.5.4 MOOS variables published by pTargetOpportunity:

.

MOOS variable	Type	Description	Format
NAFCON_MESSAGES	$	Deploy for ownship published if the intercept conditions for a reported target are satisfied.	
.			

13.6 pClusterPriority

13.6.1 Brief Overview

This MOOS module does a darn good job at doing what it is supposed to do

13.6.2 Parameters for the pClusterPriority Configuration Block

The following configuration parameters are defined for `pClusterPriority`.

`verbose`: Level of diagnostic verbosity during runtime. Choises are `verbose`, `terse`, and `quiet`.

`target`: Name of target prosecuted by the cluster. Can be fixed, e.g. `TGT_6` or `BOBBY`. If specified to 'target_updates' the target identification is dynamically allocated. The MOOS variable containing the dynamical target ID is specified in the configuration parameter `target_updates`.

`target_updates`: Identifies the MOOS variable containiing the ID for the target currently being prosecuted. If not specified default is `TARGET_ID`.

`behavior_updates`: Identifies the MOOS variable used for publishing the behavior updates, e.g. for use by `BHV_Attractor`. If not specified default is `MY_TARGET`.

`ownship`: Node ID for ownship, case insensitive, e.g. `UNICORN`.

`friend`: Node ID for collaborating cluster node, case insensitive, e.g. `CARIBOU`. Must be specified for each cllaborating node.

`pwt`: Base priority weight for attracting behavior, e.g. `BHV_Attractor`.

`max_delay_friends`: Maximum allowed delay in seconds for status reports recveived from collaborating nodes. If exceeded the node will be ignored in computing attractor priorities.

Below is an example configuration block for `pClusterpriority`,

Listing 13.6 - Sample `pClusterPriority` *configuration block .*

```
 1 ProcessConfig = pClusterPriority
 2 {
 3    AppTick    = 4
 4    CommsTick  = 4
 5    verbosity = verbose
 6    target = target_updates
 7    target_updates = TARGET_ID
 8    behavior_updates = MY_TARGET
 8    ownship = unicorn
 9    friend = macrura
10    friend = OEX
11    pwt =  100
12    max_delay_friends= 60
13 }
```

13.6.3 MOOS variables subscribed to by pClusterPriority:

.

13.6.4 MOOS variables published by pClusterPriority:

.

13.7 pHuxley

13.7.1 Brief Overview

`pHuxley` is the process which interface with the Main Vehicle Computer (MVC-Huxley) of the Bluefin AUV. During start up pHuxley requests the data streams required from the MVC. Once the back-seat driver is turned on (i.e. when the front-seat driver (MVC) is ready to receive commands from the back-seat) a flag is set and pHuxley starts sending desired speed, heading and depth values to the MVC. pHuxley controls the hand-offs between the front-seat and the back-seat.

13.7.2 Parameters for the pHuxley Configuration Block

13.7.3 MOOS variables subscribed to by pHuxley:

.

13.7.4 MOOS variables published by pHuxley:

.

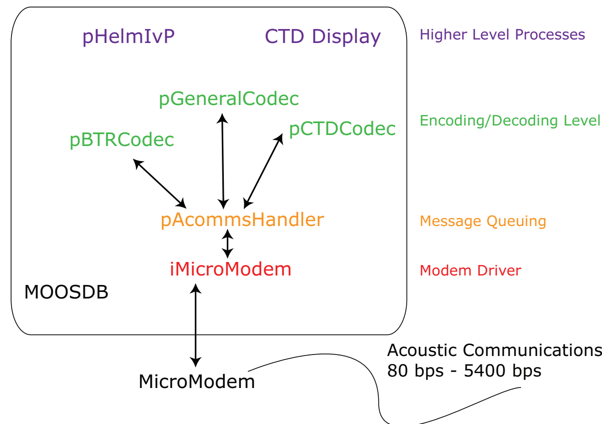

Figure 18: schematic of current acoustic modem stack

13.8 pGeneralCodec

13.8.1 Brief Overview

Any terms in *italics* are defined the Glossary (section 13.8.9).

Problem Communication between multiple autonomous vehicles is often subject to severe throughput limitations, especially in the ocean environment, where highly rate-limited acoustic channels are often the only option. In order to make the best use of this available bandwidth, messages need to be compacted to a minimal size before sending.

Background Figure 18 illustrates the current status of the acoustic communication software "stack" from the viewpoint of this laboratory. pGeneralCodec fits in the level of message encoding/decoding, as suggested by the name.

Solution pGeneralCodec is an process that proposes to be a practical first-pass solution to the difficult problem posed here. It reduces the data required to be sent by:

- Predefined messages: the user must specify a message structure what specifies what fields the message contains and how large each field should be (in an intuitive fashion that pGeneralCodec turns into bits). Both the sender and receiver have preshared knowledge of the message structure. From this knowledge, no meta information about the message (beyond an identifier) needs to be sent, simply the data.
- Custom field sizes: message fields are defined with custom tolerances (ranges and precisions) that are tighter than those given by the IEEE standards for floating point and integer numbers. For example, if a field needs to hold an integer that will never range outside [0, 1000] that field in the message will only be 10 bits long (ceil($\log_2 1001$)).

Furthermore, pGeneralCodec has powerful parsing abilities for both encoding and decoding, including the ability to perform certain geodesic conversions (latitude, longitude $\leftrightarrow$ UTM x,y) and lookups (*modem id* $\leftrightarrow$ vehicle name) on data.

13.8.2 Parameters for the pGeneralCodec Configuration Block

Example .moos file The moos file is simple since the bulk of the configuration is stored in separate XML files (see section 13.8.5 for the configuration of these files):

```
//------------------------------------------------------------------
// pGeneralCodec configuration  block

ProcessConfig = pGeneralCodec
{
  // available to all moos processes.
  AppTick    = 4
  CommsTick  = 4

  // available to all tes moos processes

  //verbose, terse, quiet
  verbosity = verbose

  message_file = example1.xml
  message_file = example2.xml
  message_file = bobs_special_command_set.xml

}
```

Filling out the .moos file

- `verbosity`: choose verbose for full text terminal output, terse for symbolic heartbeat output, and quiet for no terminal output.
- `message_file`: path to an XML file containing a message set of one or messages.

13.8.3 MOOS variables subscribed to by pGeneralCodec:

All the variables subscribed to by pGeneralCodec are configured within the message XML files. See section 13.8.5 and beyond for details on filling out and interpreting these XML files. Thus, in the following table instead of MOOS variables, XML tags from the message XML file are listed. For each `<message></message>` defined, a subscription takes place for the MOOS variable specified at run time within the tag. For example, the .moos file includes a file `message_file = example1.XML`, and `example1.XML` contains the tag `<incoming_hex_moos_var>STATUS_HEX_30B</incoming_hex_moos_var>`. Thus, `STATUS_HEX_30B` is subscribed for.

XML tag specifying a MOOS variable	Type	Description	Published by
<incoming_hex_moos_var/>	$	Hexadecimal string to decode (probably from MicroModem).	pAcommsHandler (or MOOSBlink)
<destination_moos_var key="destkey"/>	$ or D	Contains the modem_id to send this message from. Can either be double (ex: 3) or string (ex: ...,destkey=3,...)	pNaFCon (PLUSNET_MESSAGES) or others
<trigger_moos_var/>	$ or D	A publish here triggers the creation of this message. The contents may contain message parts or not.	pNaFCon (PLUSNET_MESSAGES) or others
<moos_var key="somekey"/>	$ or D	Data for a given *message_var*. Can either be double (ex: 3.234) or string (ex: "bob" or "3.234") or keyed string (ex: ...,somekey=3.234,...).	pNaFCon (PLUSNET_MESSAGES) or others

13.8.4 MOOS variables published by pGeneralCodec:

Similarly to the subscriptions, the publishes done by pGeneralCodec are all defined in the message XML files. Again, instead of MOOS variables, the table below indicates the XML tags for which one can define the publishes.

XML tag specifying a MOOS variable	Type	Description	Format
<outgoing_hex_moos_var/>	$	Encoded hexadecimal string (likely to be subscribed for by pAcommsHandler).	if the destination is NOT broadcast (*modem_id=0*: "Dest=*modem_id*, HexData=*2N character hex string where N is bytes defined by <size></size>*", else "*2N character hex string*"
<publish> <moos_var type="string"/> </publish>	$	(string) Message created from decoded hexadecimal string.	Defined by <format></format>
<publish> <moos_var type="double"/> </publish>	D	(double) Message created from decoded hexadecimal string.	Defined by <format></format>

13.8.5 Usage

Compilation pGeneralCodec depends on the boost string library and the xerces-c XML parsing library in addition to the libraries included in MOOS and moos-ivp-local.

- boost: reference *http://www.boost.org/* or look for your distribution's boost developer package (libboost-dev in debian/ubuntu).
- xerces-c: reference *http://xerces.apache.org/xerces-c/* or look for your distribution's xerces-c developer package (libxerces28-dev in debian/ubuntu as of this writing).

Example message XML file Also see section 13.8.7 for further examples. Let's call this file example1.xml:

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<message_set>
  <message>
    <name>GoToCommand</name>
    <outgoing_hex_moos_var>OUT_GOTO_HEX</outgoing_hex_moos_var>
    <incoming_hex_moos_var>IN_GOTO_HEX</incoming_hex_moos_var>
    <destination_moos_var key="Destination">OUTGOING_COMMAND</destination_moos_var>
    <trigger>publish</trigger>
    <trigger_moos_var mandatory_content="CommandType=GoTo">OUTGOING_COMMAND</trigger_moos_var>
    <size>30</size>
    <layout>
      <static>
        <name>type</name>
        <value>goto</value>
      </static>
      <int>
        <name>goto_x</name>
        <max>10000</max>
        <min>0</min>
      </int>
      <int>
        <name>goto_y</name>
        <max>10000</max>
        <min>0</min>
      </int>
      <bool>
        <name>lights_on</name>
      </bool>
      <string>
        <moos_var>SPECIAL_INSTRUCTIONS</moos_var>
        <name>new_instructions</name>
        <max_length>10</max_length>
      </string>
      <float>
        <name>goto_speed</name>
        <max>3</max>
        <min>0</min>
        <precision>2</precision>
      </float>
    </layout>
    <publish>
      <moos_var>INCOMING_COMMAND</moos_var>
      <all />
    </publish>
    <publish>
      <moos_var>SPECIAL_INSTRUCTIONS</moos_var>
      <format>special_instructions=%1%,lights_on=%2%</format>
      <message_var>new_instructions</message_var>
```

```
      <message_var>lights_on</message_var>
    </publish>
  </message>
  <message>
    <name>VehicleStatus</name>
    <trigger>time</trigger>
    <trigger_time>30</trigger_time>
    <outgoing_hex_moos_var>OUT_STATUS_HEX</outgoing_hex_moos_var>
    <incoming_hex_moos_var>IN_STATUS_HEX</incoming_hex_moos_var>
    <size>30</size>
    <layout>
      <float>
        <name>nav_x</name>
        <moos_var>NAV_X</moos_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <float>
        <name>nav_y</name>
        <moos_var>NAV_Y</moos_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <enum>
        <name>health</name>
        <moos_var>VEHICLE_HEALTH</moos_var>
        <value>good</value>
        <value>low_battery</value>
        <value>abort</value>
      </enum>
    </layout>
    <publish>
      <moos_var>STATUS_SUMMARY</moos_var>
      <all />
    </publish>
  </message>
</message_set>
```

Filling out the message XML file All the messages defined in the XML message files included in the .moos file define the entirety of the available message set. First, a brief background on XML (e**X**tensible **M**arkup **L**anguage). XML files contain tags (like `<name></name>`) that are considered "metadata" and define both the structure of the following data and the contents. Order of the tags does not matter for a given level unless explicitly specified. Text data resides both in the tags (like `<name>bob</name>` or as attributes of the tag (such as `<name id="1245"></name>`). XML files can be edited with any text editor. For more information on XML consult any number of books on the subject or browse the great internet. It is a very widely used format for storing data that can be both read by both people and computers.

Allowed tags Let us first give a description of the allowed tags and then we will go into a description of how to go about making a command file.

- `<?xml version="1.0" encoding="ASCII" standalone="yes"?>`: specifies the file is XML and which schema to use. When schema checking is enabled, this line will change some. All you need to know is that this must be the first line of every message XML file.
- `<message_set></message_set>`: the root element. All XML files must have a single root element. Since we are define a set of messages (one or more per file), this is a logical choice of name for the root element. [mandatory, one allowed].
- `<message></message>`: defines the start of a message. [mandatory, one or more allowed].
  - `<name></name>`: a human readable name for the message. This is not used internally at this point in time. [mandatory, one allowed]
  - `<trigger></trigger>`: how the message is created. Currently this field must take the value "publish" (meaning a message is created on a publish event to a certain moos variable) or "time" (a message is created on a certain time interval). [mandatory, one allowed]
  - `<trigger_moos_var></trigger_moos_var>`: used if `<trigger>publish</trigger>`, this field gives the moos variable that pGeneralCodec should look for publishes to in order to trigger the creation of this message [mandatory iff `<trigger>publish</trigger>`]. optional attribute `mandatory_content` specifies a string that must be a substring of the contents of the trigger variable in order to trigger the creation of a message. For example, if you wanted to create a certain message every time `COMMAND` contained the string `CommandType=GoTo...` but no other time, you would specify `mandatory_content="CommandType=Go` within this tag.
  - `<trigger_time></trigger_time>`: used if `<trigger>time</trigger>`, this field gives the time interval pGeneralCodec should create this message. For example, a value of `<trigger_time>10</trigger_time>` would mean a message was created every ten seconds. [mandatory iff `<trigger>time</trigger>`].
  - `<size></size>`: the size of the message in bytes. There are eight bits (binary digits) to a byte. Use $N - 2$ here for messages passed through pAcommsHandler where N is the desired micromodem frame size (N =32, 64, or 256 depending on the rate). If the `<layout></layout>` of the message exceeds this size, pGeneralCodec will exit on startup with information about sizes, from which you can remove or reduce the size of certain *message_var*s.
  - `<outgoing_hex_moos_var></outgoing_hex_moos_var>`: where to publish the encoded message (as a hexadecimal string). [mandatory, one allowed].
  - `<incoming_hex_moos_var></incoming_hex_moos_var>`: where to look for messages (hex string) to decode. [mandatory, one allowed].
  - `<destination_moos_var></destination_moos_var>`: moos variable to find where this message should be sent. Specify attribute "key=" to specify a substring to look for within the value of this moos variable. For example, if `COMMAND` contained the string

Destination=3 and you want this message sent to modem id 3, then you should set key=Destination to properly parse that string. [optional: default is 0 (broadcast), one allowed].

– <layout></layout>: defines the message structure itself (what fields [the message variables or *message_var*s] the message contains and how they are to be encoded). [mandatory, one allowed].

  * <static></static>: a *message_var* that is not actually sent with the message but can be used to include in received messages (*publishes*). [optional, one or more allowed].
    · <name></name>: the name of this *message_var*. [mandatory, one allowed].
    · <value></value>: the value of this static variable. [mandatory, one allowed].
  * <bool></bool>: a boolean (true or false) *message_var* [optional, one or more allowed].
    · <name></name>
    · <moos_var></moos_var>: the moos variable from which to pull the value of this field. [optional if <trigger>publish</trigger>: default is trigger_moos_var; mandatory if <trigger>time</trigger>, one allowed].
  * <int></int>: an integer *message_var* [optional, one or more allowed].
    · <name></name>
    · <moos_var></moos_var>
    · <max></max>: the maximum value this field can take. [mandatory, one allowed].
    · <min></min>: the minimum value this field can take. [mandatory, one allowed].
  * <float></float>: a floating point *message_var* [optional, one or more allowed].
    · <name></name>
    · <moos_var></moos_var>
    · <max></max>
    · <min></min>
    · <precision></precision>: an integer that specifies the number of decimal digits to preserve. Negatives are allowed. For example, <precision>2</precision> rounds 1042.1234 to 1042.12; <precision>-1</precision> rounds 1042.1234 to 1.04e3. [mandatory, one allowed].
  * <string></string>: an ASCII string *message_var* [optional, one or more allowed].
    · <name></name>
    · <moos_var></moos_var>
    · <max_length></max_length>: the length of the string value in this field. Longer strings are truncated. <max_length>4</max_length> means "ABCDEFG" is sent as "ABCD". [mandatory, one allowed].
  * <enum></enum>: an enumeration *message_var* [optional, one or more allowed].
    · <name></name>
    · <moos_var></moos_var>
    · <value></value>: a possible value (string) the enum can take. Any number of values can be specified. [mandatory, one or more allowed].

- `<publish></publish>`: defines a single output value upon receipt of a message. Any number of publishes containing any subset of the *message_var*s can be specified. [mandatory, one or more allowed].
  - `<moos_var></moos_var>`: the name of the moos variable to publish to. If desired, a format string is allowed here as well (e.g. `%1%_NAV_X` will fill `%1%` with the first *message_var*). See the `<format></format>` tag description for more info. [mandatory, one allowed].
  - `<format></format>`: a string conforming to the format string syntax of the boost::format [1] library. This field will specify the format of the string published to the moos variable defined in `<moos_var></moos_var>`. At its simplest it is a string of incrementing numbers surrounded by %%. Or, instead, you may also use a printf style string, using %d for int *message_var*, %lf for floats, and %s for strings, bools and enums. [optional: default is `name1=%1%,name2=%2%,name3=%3%`, where `name1` is the name of the first `<message_var></message_var>` field to follow, `name2` is the second, etc. exception: default is `%1%` if only a single `<message_var></message_var>` defined. one allowed].
  - `<message_var></message_var>`: the name (`<name></name>` above) of a *message_var* contained in this message (i.e. an `<int></int>`, `<bool></bool>`, etc.) the values of these fields upon receipt of a message will be used to populate the format string and the result will be published to `<moos_var></moos_var>`. [mandatory unless `<all />` used, one or more allowed].
  - `<all />`: equivalent to `<message_var></message_var>` for all the *message_var*s in the message. This is a shortcut when you want to publish all the data in a human readable string. [optional, one allowed].

Designing a publish triggered message We will look at two scenarios and detail how to design a proper message file for each scenario. We will reference the example file given in section 13.8.5 for both scenarios.

Scenario: you want to command an surface craft to move to a new location:

1. Identify the data: location (x (`goto_x`) and y (`goto_y`) on a local grid). you also want to specify a speed (`goto_speed`) at which it should transit, whether it should have lights (`lights_on`) on or not, and finally a string (`special_instructions`) with possible special instructions. All these data will come in to a moos variable `OUTGOING_COMMAND` on a string like:

```
OUTGOING_COMMAND: Destination=3,CommandType=GoTo,goto_x=351,goto_y=294,
                  lights_on=true,special_instructions=make_toast,goto_speed=2.3
```

2. Type the data (i.e. is it an int, a float, a string?) and give the ranges and precisions needed:

   - `goto_x`: integer (in meters) (`int`) that will operate on a (positive valued) local grid not to exceed 10 km in either dimension.
   - `goto_y`: same as `goto_x`.

[1] see the syntax of the **format-string** at *http://www.boost.org/doc/libs/1_37_0/libs/format/doc/format.html#syntax*

- goto_speed: speed in m/s. the vehicle cannot exceed 3 m/s and does not go backwards. we would like to give precise speeds to the hundredths place. thus, we need a float ranging from 0 to 3 with precision 2.
- lights_on: simply a flag (boolean value) whether to have our lights on or off. thus, we need a bool *message_var*.
- special_instructions: We want a field that can hold any string of characters, but we know it will not exceed ten characters. thus, we need a string *message_var*.

3. Putting all this together, we can define the <layout></layout> portion of the first message defined in section 13.8.5. We do not need any <moos_var></moos_var> tags within the *message_vars* since all the data are contained in the contents of the trigger variable message (OUTGOING_COMMAND). That is, when we leave out the <moos_var></moos_var>, pGeneralCodec will insert <moos_var>OUTGOING_COMMAND</moos_var>, which is exactly what we want. For example, taking one of the *message_vars*:

```
<int>
  <name>goto_x</name>
  <max>10000</max>
  <min>0</min>
</int>
```

is exactly the same as saying

```
<int>
  <name>goto_x</name>
  <moos_var>OUTGOING_COMMAND</moos_var>
  <max>10000</max>
  <min>0</min>
</int>
```

4. Now we can fill out the rest of the tags on the <message></message> level:

- <name>GoToCommand</name>: just a name so we can identify this message quickly when reading through the XML.
- <outgoing_hex_moos_var>OUT_GOTO_HEX</outgoing_hex_moos_var>: we will publish our hex strings here for consumption by the low level modem stack processes (probably pAcommsHandler send=OUT_GOTO_HEX...). the publishes will look a bit like:

```
OUT_GOTO_HEX: Dest=3,HexData=a9385bc098109830a9385bc0981098a9385bc098109830a9385bc0981098
```

- <incoming_hex_moos_var>IN_GOTO_HEX</incoming_hex_moos_var>: where we expect to receive incoming messages to decode (probably from pAcommsHandler receive=IN_GOTO_HEX...) messages should be pure hex with the community set to the sending *modem id*. An example for the received message on *modem id*=3 if the sending node (say a topside command computer) is *modem id*=1:

```
IN_GOTO_HEX: a9385bc098109830a9385bc0981098a9385bc098109830a9385bc0981098
{Community=1}
```

- <destination_moos_var key="Destination">OUTGOING_COMMAND</destination_moos_var>: we want to pull the destination *modem id* from the same string as the data (moos variable OUTGOING_COMMAND and it is found in a key called "Destination".

- <trigger>publish</trigger>: we are creating this message on a **publish** (to OUTGOING_COMMAND).
- <trigger_moos_var mandatory_content="CommandType=GoTo">OUTGOING_COMMAND</trigger_mo
  OUTGOING_COMMAND is the trigger variable and it must contain the substring CommandType=GoTo. That is, other commands might be published here (e.g. CommandType=Loiter, CommandType=Track) and we do not define the message structure of those here (this particular <message></message> is only for a GoTo message). other messages can be created to encode/decode these other command types.
- <size>30</size>: we want this message to fit in a WHOI micromodem FSK frame (32 bytes) and thus we have 30 bytes to work with (pAcommsHandler needs 2 bytes of header).

5. Finally, we fill out the <publish></publish> section which indicates where (i.e. what moos variables) and how (what format and which part(s) of the message) pGeneralCodec should publish decoded messages upon receipt of hex from other vehicles. Each <publish></publish> indicates a separate action that is taken upon receipt of a message. As many <publish></publish> sections as desired may be included for a given message. So, for our example message, we want to replicate the original string (a common practice):

```
INCOMING_COMMAND: CommandType=GoTo,goto_x=351,goto_y=294,
                  lights_on=true,special_instructions=make_toast,goto_speed=2.3
```

to do this we fill out a publish <all />. This is the simplest form of the <publish></publish> section:

```
<publish>
  <moos_var>INCOMING_COMMAND</moos_var>
  <all />
</publish>
```

this says to take every *message_var* and make a "key=value" comma-delimited string from it. the above <publish></publish> block is a shortcut for a much longer form:

```
<publish>
  <moos_var>INCOMING_COMMAND</moos_var>
  <format>type=goto,goto_x=%1%,goto_y=%2%,lights_on=%3%,
  special_instructions=%4%,goto_speed=%5%</format>
  <message_var>goto_x</message_var>
  <message_var>goto_y</message_var>
  <message_var>lights_on</message_var>
  <message_var>special_instructions</message_var>
  <message_var>goto_speed</message_var>
</publish>
```

these two blocks are functionally identical.

We may want to also publish the special_instructions to another moos variable, so that:

```
SPECIAL_INSTRUCTIONS: special_instructions=make_toast,lights_on=true
```

we can do this with another publish block:

```
<publish>
  <moos_var>SPECIAL_INSTRUCTIONS</moos_var>
  <format>special_instructions=%1%,lights_on=%2%</format>
  <message_var>new_instructions</message_var>
  <message_var>lights_on</message_var>
</publish>
```

in this case the <format></format> block is necessary because the default would be
<format>new_instructions=%1%,lights_on=%2%</format> not
<format>special_instructions=%1%,lights_on=%2%</format>.

Those are the basics to designing a **publish** triggering message.

Designing a time triggered message Scenario: we need a status message that grabs data from various moos variables and publishes them (encoded) on a time interval. We will not go into as much detail here, but rather highlight the changes from the previous scenario.

- you will notice

  ```
  <trigger>time</trigger>
  <trigger_time>30</trigger_time>
  ```

  instead of

  ```
  <trigger>publish</trigger>
  <trigger_moos_var mandatory_content="CommandType=GoTo">OUTGOING_COMMAND</trigger_moos_var>
  ```

  this indicates that a message should be made on a time interval (given by <trigger_time></trigger_tim
  which is every 30 seconds here), rather than on a publish to some moos variable.

- you will notice that all the *message_var*s have a <moos_var></moos_var> tag, which was omitted in the previous example since we were taking data from the trigger variable. Obviously, there is no trigger variable now so we must specify a location for the data to come from (in the moos db). The newest available value will be used when the message needs to be made. This means there is no guarantee that the data is fresh. Thus, you should use moos variables that are often updated for a <trigger>time</trigger> message. If this is not the case, a <trigger>publish</trigger> message (see previous scenario) may be a better choice.

- the format of the value read from the <moos_var></moos_var> can have several options. First, if the *message_var* is of a numeric type (<int></int>, <float></float>, <bool></bool>) and the <moos_var></moos_var> is a double, the value of the double is used as is. If the *message_var* is a string, two options are available. First, the pGeneralCodec looks for a substring of the form:

  ```
  name=value
  ```

  within the string and picks out the value to send for the message. If there is no such name= substring, the entire string is converted to the appropriate form. An example: we have a <float></float> called <name>my_float</name> that has a tag
  <moos_var>SOME_FLOAT_VARIABLE</moos_var>:

– if

```
(double)SOME_FLOAT_VARIABLE: 3.56
```

then 3.56 is sent.

– if instead

```
(string)SOME_FLOAT_VARIABLE: "my_float=3.56"
```

then 3.56 is still sent.

– if instead

```
(string)SOME_FLOAT_VARIABLE: "3.56"
```

again, 3.56 is sent.

– Finally, if some other string like

```
(string)SOME_FLOAT_VARIABLE: "blah=3.56"
```

then `blah=3.56` is converted (using streams) to a float, which will probably be zero or something else undesired. In other words, this case is not what you want, whereas the above three are fine.

13.8.6 Details

- the code is split into the following classes:
  1. `CMOOSApp` extended class `CpGeneralCodec` in `pGeneralCodec.cpp`. This is the moos app that you run.
  2. `Message` in `message.cpp`. This class defines everything you would want to do with a given message (`SENSOR_DEPLOY`, etc). then i instantiate as many Messages as needed (for every `<message></message>` tag in the XML file), read in the parameters with the Xerces XML parser, and call Message.encode() to create a message, or Message.decode() to decode it.
     (a) `MessageVar` in `message_var.cpp`. This class defines a *message_var* (i.e. a piece of the message (int, bool, etc.)) and any actions to be done to it. an instantiation is made for all the *message_vars* in each message. (e.g. `<int></int>`, `<string></string>`, etc.)
     (b) `Publish` in `publish.cpp`. This defines a given publish action upon receipt of a message. One instantiation for each `<publish></publish>` given.
  3. `MessageParser` in `message_xml_parser.cpp`. Reads in an XML file uses xerces.
  4. `MessageContentHandler` in `message_xml_callbacks.cpp`. Contains callbacks which populate the Message with data from the XML file.
- We may want to know the actual layout of the binary/hex message. Let us explain it with an example; for the first example message in `example1.xml` given in section 13.8.5, if we run pGeneralCodec we get information about that message:

```
        type (static):
                value: {goto}
                size [bits]: [0]
        goto_x (int):
                source: {OUTGOING_COMMAND}
                [min, max] = [0,10000]
                size [bits]: [14]
        goto_y (int):
                source: {OUTGOING_COMMAND}
                [min, max] = [0,10000]
                size [bits]: [14]
        lights_on (bool):
                source: {OUTGOING_COMMAND}
                size [bits]: [1]
        new_instructions (string):
                source: {SPECIAL_INSTRUCTIONS}
                max_length: {10}
                size [bits]: [80]
        goto_speed (float):
                source: {OUTGOING_COMMAND}
                [min, max] = [0,3]
                precision: {2}
                size [bits]: [9]
```

the calculated sizes are used to pack the message like so ({#} equals size of field in bits), where left to right is the same as reading the hex string from left to right:

```
[[0 {122}][goto_x {14}][goto_y {14}][lights_on {1}][new_instructions {80}][goto_speed {9}]]
```

where [0 {122}] means zero fill the front of the message to the full size (30 bytes = 240 bits minus 118 for other fields = 122). Byte boundaries are dissolved and encoded as a string "ABCDEF..." where the most significant byte (MSB, or leftmost 8 bits) is 0xAB, second MSB is 0xCD, etc. Encoding and decoding are done by functions available in binary.h in libtes_util.a.

13.8.7 Further examples

- I currently store our working message files in moos-ivp-local/data/acomms. look for .xml files in this directory for further examples.
- Probably the simplest message you can make (for a single string MOOS variable that gets truncated at 30 chars and sent to broadcast):

```
<?XML version="1.0" encoding="ASCII" standalone="yes"?>
<message_set>
  <message>
    <name>SimpleStringSender</name>
    <outgoing_hex_moos_var>OUT_STRING_HEX</outgoing_hex_moos_var>
    <incoming_hex_moos_var>IN_STRING_HEX</incoming_hex_moos_var>
    <trigger>publish</trigger>
```

```
    <trigger_moos_var>MY_STRING</trigger_moos_var>
    <size>30</size>
    <layout>
      <string>
        <name>my_string</name>
        <max_length>30</max_length>
      </string>
    <publish>
      <moos_var>INCOMING_COMMAND</moos_var>
      <all />
    </publish>
  </message>
</message_set>
```

13.8.8 Message XML reference sheet

This is a quick reference of all the allowed tags with some comments about their use:

```
<?xml version="1.0" encoding="UTF-8"?> <!-- required XML declaration -->
<message_set> <!-- start the message_set. only one per file -->
  <message><!-- start the message. as many as desired per file -->
    <name></name> <!-- overall message name, only for human use -->

    <destination_moos_var key=""></destination_moos_var> <!-- if included, here is where you
                                                               find the destination of the message.
                                                               otherwise, use broadcast (0) -->
    <outgoing_hex_moos_var></outgoing_hex_moos_var> <!-- publish encoded hex string here -->
    <incoming_hex_moos_var></incoming_hex_moos_var> <!-- look for incoming hex string here -->

    <trigger>publish</trigger><!-- make a message (encode) when a write to a variable is noticed ...
    <trigger_moos_var mandatory_content=""></trigger_moos_var>
         <!-- ... specifically a write to THIS variable,
          which must contain ‘mandatory_content‘ within its string -->

    <!-- OR -->
    <trigger>time</trigger><!-- make a message on some regular time interval ... -->
    <trigger_time></trigger_time><!-- ...defined (in seconds) here -->

    <size>30</size>    <!-- enforce this message size (bytes) -->

    <!-- encoding -->
    <layout> <!-- start defining the message layout -->
      <static algorithm=""> <!-- a variable that is not actually sent, its value is given here
                                 ’algorithm’ specifies one or more functions to be performed on
                                 the data before encoding-->
        <name></name> <!-- name of this message variable (message_var) -->
        <value></value><!-- value of this static -->
      </static>
      <int algorithm=""><!-- stores signed or unsigned integers up to size long (typically 32) bits
        <moos_var key=""></moos_var> <!-- what moos variable to pull the value
                                          from when encoding. if omitted, use trigger_var
```

```
                                  ’key’ indicates the substring to look for in a key=value
                                  pair in the contents of this moos variable-->
    <name></name>
    <max></max><!-- maximum value this field can take. larger values are fixed to this limit -->
    <min></min><!-- minimum value this field can take. smaller values are fixed to this limit --
  </int>
  <float algorithm=""><!-- stores float variables up to size of the system long (typically 32 bi
    <moos_var key=""></moos_var> <!-- if omitted, use trigger_var -->
    <name></name>
    <max></max>
    <min></min>
    <precision></precision><!-- number of decimal places to keep -->
  </float>
  <bool algorithm=""> <!-- true or false -->
    <moos_var key=""></moos_var> <!-- if omitted, use trigger_var -->
    <name></name>
  </bool>
  <enum algorithm=""> <!-- a set of predefined strings -->
    <moos_var key=""></moos_var> <!-- if omitted, use trigger_var -->
    <name></name>
    <value></value> <!-- define the first enum string here -->
    <value></value> <!-- and the second here ... -->
    <value></value> <!-- and so on as long as you want ... -->
  </enum>
  <string algorithm="">
    <moos_var></moos_var> <!-- if omitted, use trigger_var -->
    <name></name>
    <max_length></max_length> <!-- truncate string to this number of chars -->
  </string>
</layout>

<!-- decoding -->
<publish> <!-- simple -->
  <moos_var type=""></moos_var> <!-- where to publish to. type="string" (default) or "double" --
  <format></format> <!-- format string. use boost (’%1%’’, etc) or printf(’%d’, etc).
                    if omitted, use ’name=%1%,name=%2% etc for all message_var specified.
                    exception, a single message_var is specified, ’%1%’ is used for the format s
                    (that is, the key is not published) -->
  <message_var algorithm=""></message_var> <!-- name of the message_var to insert first.
                                              algorithm (optional) lets you perform
                                              conversions on the data before inputting to
                                              format string-->
  <message_var algorithm=""></message_var> <!-- name of the second message_var to insert
                                              ... and so on -->
</publish>

<publish> <!-- shortcut for all -->
  <moos_var></moos_var>
  <all /> <!-- equivalent to <message_var></message_var>
               entries for all message_var fields above -->
</publish>
```

```
  </message>
</message_set>
```

13.8.9 Glossary

- *message_var*: the term for a field within a message. a *message_var* can be of several types: int, bool, double, float, string, or enum. the *message_var*s are defined within the `<layout></layout>` part of the message.
- *modem id*: the number given to each whoi micromodem (this is like a variable MAC address) that defines senders and receivers. modem ids must be unique for each network and are configured using `$CCCFG,SRC,#`, where `#` is the modem id (integer from 0 to 127?).
- *publish*: the term for a given action to be performed upon receipt of a message. this term is used since this action will involve a moos publish to some variable (after some string parsing/formatting).
- *XML*: extensible markup language: a specification for defining a custom markup language, which is a set of annotations given to text to indicate structure. here we use XML to indicate the structure of a "message" and its subsequent breakup ("publishes", during decoding)
  - *tag*: the name given to the "metadata" in the XML file. for example: the tag `<message></message>` indicates the start and end of a message (this is "metadata"), but nothing about what it contains (which is"data")
  - *attributes*: data contained within a tag. for example, in `<int algorithm="to_upper"></int>`, **algorithm** is an attribute of the tag **int**.
  - *CDATA*: **C**haracter **DATA**, or the data contained within a tag or an attribute. for example, in `<name>nav_x</name>`, **nav_x** is CDATA.

13.9 pAcommsHandler

13.9.1 Brief Overview

problem acoustic communications (in our case, with the WHOI micromodem) are highly limited in throughput. thus, it is unreasonable to expect "total throughput" of all communications data. furthermore, even if total throughput is achievable over time, certain messages (e.g. vehicle status) have a lower tolerance for delay than others (e.g. CTD sample data). reference *http://acomms.whoi.edu/umodem/documentation.html* for more information on the WHOI micromodem.

pAcommsHandler roughly performs the same functions of pRouter but generalized to handle any number of message queues and extended to give more control over queue parameters. figure 18 gives a rough view of the current acoustic communications stack from the viewpoint of the operations carried out by the Laboratory for autonomous marine Sensing.

solution pAcommsHandler:

- maintains an arbitrary number of message queues (each tied to a different MOOS variable) for hexadecimal data strings (for the WHOI micromodem driver process, currently iMicroModem, to consume upon request)
- allows configuration of the queue priorities and dynamic growth of the priority over the time since the last sent message
- allows management of WHOI CCL message types as well as internal MOOS routing.

13.9.2 usage

compilation pAcommsHandler depends on the boost string library in addition to the libraries included in MOOS and moos-ivp-local. reference *http://www.boost.org/* or look for your distributions boost developer package (libboost-dev in debian/ubuntu).

13.9.3 Parameters for the pAcommsHandler Configuration Block

example moos file here is an example .moos file from the GLINT08 experiment in pianosa, italy. this particular file was used by the BF21 AUV 'Unicorn':

```
ProcessConfig = pAcommsHandler
{
  // available to all moos processes.
  AppTick    = 4
  CommsTick  = 4

  // available to all tes moos processes

  // verbose, terse, quiet
  verbosity = verbose
```

```
  // all case insensitive

  modem_id = 3

  // information about iMicroModem
  micromodem_command_var = MICROMODEM_COMMAND
  micromodem_data_var = MICROMODEM_DATA

  // send, send_CCL =
      // VarName
      //                          VariableID
      //                              [Ack]
      //                                  [BlackoutTime]
      //                                      [MaxQueue]
      //                                           [NewestFirst]
      //                                               [Priority]
      //                                                    [PriorityTimeConstant]

  send = OUT_CTD_HEX_30B,           1,  0,  0,  100, 1,  0.3, 120
  send = OUT_CTD_HEX_62B,           2,  0,  0,  100, 1,  0.3, 120
  send = OUT_CTD_HEX_254B,          3,  0,  0,  100, 1,  0.2, 120
  send = OUT_BTR_HEX_254B,          4,  0,  0,  100, 1,  0.5, 120

      //                          CCLIdentifierByte
  send_CCL = OUT_DEPLOY_HEX_32B,    1a, 1,  0,  1,   1,  10,  120
  send_CCL = OUT_PROSECUTE_HEX_32B, 1a, 1,  0,  1,   1,  10,  120

  send_CCL = OUT_STATUS_HEX_32B,    1b, 0,  30, 1,   1,  1,   120
  send_CCL = OUT_CONTACT_HEX_32B,   1b, 0,  0,  1,   1,  2,   120
  send_CCL = OUT_TRACK_HEX_32B,     1b, 0,  0,  1,   1,  4,   120

  // receive = VarName, VariableID
  receive = IN_CTD_HEX_30B, 1
  receive = IN_CTD_HEX_62B, 2
  receive = IN_CTD_HEX_254B, 3
  receive = IN_BTR_HEX_254B, 4

  // receive_CCL = VarName, CCLIdentifierByte
  receive_CCL = IN_PLUS_COMMANDS_HEX_32B, 1a
  receive_CCL = IN_PLUS_MESSAGES_HEX_32B, 1b
}
```

filling out the .moos file

general parameters

- `verbosity`: choose verbose for full text terminal output, terse for symbolic heartbeat output, and quiet for no terminal output.
- `modem_id`: integer that specifies the modem id of this current vehicle / community. this must match the micromodem SRC configuration parameter (send the modem $CCCFQ,SRC to check). for the remainder of the document, "modem ID" refers to the value $CCCFG,SRC,modem_id

iMicroModem parameters

- `micromodem_command_var`: the variable that iMicroModem is configured to receive commands on. this is the value in the iMicroModem .moos block `VarNamePrefix` concatenated to "_COMMAND". for example if `VarNamePrefix=MM`, then set `micromodem_command_var=MM_COMMAND`
- `micromodem_data_var`: the variable that iMicroModem is configured to send data on. this is the value in the iMicroModem .moos block `VarNamePrefix` concatenated to "_DATA". for example if `VarNamePrefix=MM`, then set `micromodem_command_var=MM_DATA`

outgoing queue parameters

- `send`: configure a queue for messages to be sent from this community. the send parameter is a comma delimited string of two mandatory parameters and up to six optional parameters (in order specified here). to skip an optional field and use the default value, simply use a blank ",":
  - `VarName`: name of the moos variable to subscribe to for messages to add to this queue. publishes here should be pure hexadecimal or a key=value string specified later in section 13.9.4.
  - `VariableID`: a user specified integer from 0-15 that is a tag for the VarName. thus, each `VarName` must have a unique `VariableID`. only the `VariableID` is sent with the message (to save space), not the full `VarName` of the queue. thus, this must match the `VariableID` on the receiving vehicle's `receive` configuration in the incoming parameters section below. for example, if i have `send=SOME_OUT_HEX, 1` on the sending vehicle, the receiving vehicles must have a field `receive=SOME_IN_HEX, 1`. all messages with ID 1 will be put in `SOME_IN_HEX`. clearly, if unique mapping is desired on the receiving end, unique `VariableID`s must be used on the sending end.
  - `Ack`: boolean flag (1=true, 0=false) whether to request an acoustic acknowledgment on all sent messages from this field. if omitted, default of 0 (false, no ack) is used.
  - `BlackoutTime`: time in seconds after sending a message from this queue for which no more messages will be sent. use this field to stop an always full queue from hogging the channel. if omitted, default of 0 (no blackout) is used.
  - `MaxQueue`: number of messages allowed in the queue before discarding messages. if `NewestFirst` is set to true, the oldest message in the queue is discarded to make room for the new message. otherwise, any new messages are disregarded until the space in the queue opens up.
  - `NewestFirst`: boolean flag (1=true=FILO, 0=false=FIFO) whether to send newest messages in the queue first (FILO) or not (FIFO).

– `Priority`: base priority value for this message queue. priorities are calculated on a request for data by the modem (to send a message). the queue with the highest priority (and isn't in blackout) is chosen. the actual priority (P) is calculated by

$$P = P_{base} \exp\left[(t - t_{last})/\tau)\right]$$

where P_{base} is the value set here, t is the current time (in seconds), t_{last} is the time of the last send from this queue, and τ is the `PriorityTimeConstant`. essentially, a message with low `PriorityTimeConstant` will become effective quickly again after a sent message (the exponential grows faster). a higher base priority will also increase the effectiveness of the queue.

- `send_CCL`: configure a queue for a WHOI CCL message type (do not use the moos `VariableID`). the queues mentioned above (`send=`) use the CCL identifier byte 0x20 and then the second byte includes the `VariableID` leaving $N - 2$ bytes for the user where $N = 32, 64,$ or 256 depending on the modem rate. this queue `send_CCL` does not use the second byte, thus making it useful for handling the static CCL types defined by WHOI. all parameters in the `send_CCL` list are the same as those for the `send` list with the exception of the second parameter. here it is the `CCLIdentifierByte` in hexadecimal.

incoming parameters

- `receive`: specifies a mapping between an incoming `VariableID` and a moos variable to place the incoming message. the sender's modem id is stored as the community name for the received message.
- `receive_CCL`: same as `receive` except for WHOI CCL types. rather than a `VariableID` you specify a `CCLIdentifierByte` that is the first byte of the CCL message.

13.9.4 MOOS variables subscribed to by pAcommsHandler:

All variables are configurable in the .moos file as described in section 13.9.3. For example, if `send = OUT_CTD_HEX_30B, ...` is set in the .moos file, then `OUT_CTD_HEX_30B` is the variable actually subscribed.

.moos file key specifying a MOOS variable	Type	Description	Published by
micromodem_data_var	$	micromodem data line (data requests also come on this variable)	iMicroModem
send=varname,...	$	varname contains hexadecimal string to queue (and eventually send).	Various Codecs (pGeneralCodec, pCTDCodec, etc.)

input (to pAcommsHandler) formats pAcommsHandler accepts several formats for the strings placed in the various `VarName` moos variables (outgoing message queues). the simplest is just a hexadecimal string of the appropriate length ($N - 2$ bytes where $N = 32, 64,$ or 256 depending on the modem rate). this message will be queued to send to broadcast (modem id 0[2]). an example:

[2] the WHOI micromodem does not treat certain modem IDs specially (such as zero). all data are reported to the control computer, regardless of whether that machine is the intended recipient. however, the communications structure

```
OUT_CTD_HEX_30B: 6850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
```

would queue a CTD message to be sent to broadcast (modem ID 0), given the above configuration file.

pAcommsHandler also accepts a string of key=value, comma delimited fields allowing for more flexibility. currently, the only fields supported are `HexData=` (required, of course) and `Dest=` for the intended destination modem_id.

```
OUT_CTD_HEX_30B: Dest=3,HexData=6850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
```

would queue a CTD message to be sent to id 3.

13.9.5 MOOS variables subscribed to by pAcommsHandler:

Similarly to the subscriptions, the publishes done by pAcommsHandler are all defined in the .moos file. Again, instead of MOOS variables, the table below indicates the .moos file keys for which one can define the publishes.

.moos file key specifying a MOOS variable	Type	Description	Format
micromodem_command_var	$	commands to iMicroModem	see below
receive=varname,...	$	varname contains a hexadecimal string from the micromodem to route to this queue.	see below

output (from pAcommsHandler) formats Upon receipt of a message from the micromodem driver, the first byte of the message is checked against the moos CCL type (0x20) and any of the additional `receive_CCL CCLIdentifierByte` given in the .moos file. if none of these match, the message is disregarded. if the message matches the moos CCL type (0x20), the second byte is examined for the `VariableID`. if the `VariableID` matches one of the `receive` fields in the .moos file, the hexadecimal string is stripped of its first two bytes and placed in the correspoding `VarName` moos variable. the community name is set to the sender's variable ID [3]. for example, the modem passes to iMicroModem:[4]

```
$CARXD,3,0,0,1,20016850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
```

iMicroModem [5] passes this message to pAcommsHandler, which strips the two 0x2001 bytes, and places the message in the appropriate moos variable, which for the example .moos file here is `IN_CTD_HEX_30B`:

```
IN_CTD_HEX_30B: 6850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
{Community=3}
```

defined by pAcommsHandler and other *tes* processes treat modem ID 0 as broadcast (much like xxx.xxx.xxx.255 is used for broadcast on TCP/IP networks). this means that incoming messages are read if they are to our modem ID or to modem ID 0.

[3]use CMOOSMsg::GetCommunity() to extract the community

[4]see Micro-Modem Mainboard Software Interface Guide at http://acomms.whoi.edu/umodem/documentation.html for details on these messages

[5]it is my hope to replace this process shortly so minimal details are given on iMicroModem in this documentation

13.9.6 details

here we go. rather than a mess more prose, i will resort to another bulleted list of things you may want to know. this program is a bit fluid at the moment so expect additions and changes that this document may or may not keep pace with. when in doubt an email is (almost!) always promptly responded to.

- more details on what we're doing here: pAcommsHandler takes all the configured queues and maintains a stack of messages for each queue. when it is prompted by data by iMicroModem, it has a priority "contest" between the queues. the queue with the current highest priority (as determined by the `Priority` and `PriorityTimeConstant` fields) is selected. the next message in that queue is then provided to iMicroModem to send. for modem messages with multiple frames per packet, each frame is a separate contest. thus a single packet may contain frames from different queues (e.g. a rate 5 PSK packet has eight 256 byte frames. frame 1 might grab a STATUS message since that has the current highest queue. then frame 2 may grab a BTR message and frames 3-8 are filled up with CTD messages (e.g. STATUS is in blackout, BTR queue is empty)).

- i mentioned this above but it's worth going over again. we choose modem id 0 as broadcast. thus messages with the destination field = 0 will always be read by all nodes and reported to the appropriate moos variable. otherwise, we ignore messages unless they correspond to our modem id. so if you send a message to modem id 10, pAcommsHandler for modem ids $1 \rightarrow 9$, $11 \rightarrow N$ will ignore that. this is not the default behavior of the modem, which always reports data, regardless of the sender's ID.

- if you do not wish for dynamic growth of the priorities, simply set the `PriorityTimeConstant` to a very large value such as $1e300$. then the priorities grow as

$$P = P_{base} \exp\left[(t - t_{last})/1e300\right] \simeq P_{base} \exp(0) = P_{base}$$

- for messages with `Ack=1` (acknowledge requested), the last message continues to be re-sent (that is, it is not popped from the message queue) until the ACK is received from the modem (thus blocking the sending of other messages). perhaps i will add max retries at some point soon. messages with `Ack=0` are popped and discarded when they are sent (no retries).

- this isn't as pretty as could be, but if you need to know the destination of the next message to be sent (i.e. the next message in the queue with the present highest priority), you can send a message to moos variable `POLLER_DEST_REQUEST` (contents of the message doesn't matter. you will get a reply (double) in `POLLER_DEST_COMMAND` (e.g. the next message should go to ID 5):

  ```
  POLLER_DEST_COMMAND: 5
  ```

  as the names imply, this is currently used by pAcommsPoller.

- if all else fails, read the terminal output. it's not so bad.

13.10 pAcommsPoller

13.10.1 Brief Overview

This MOOS module does a darn good job at doing what it is supposed to do

13.10.2 Parameters for the pAcommsPoller Configuration Block

13.10.3 MOOS variables subscribed to by pAcommspoller:

.

13.10.4 MOOS variables published by pAcommsPoller:

.

13.11 iMicroModem

13.11.1 Brief Overview

This MOOS module does a darn good job at doing what it is supposed to do

13.11.2 Parameters for the pAcommsPoller Configuration Block

13.11.3 MOOS variables subscribed to by iMicroModem:

.

13.11.4 MOOS variables published by iMicroModem:

.

Part VI
Real-Time Acoustic Sensing and Processing

14 On-Board Acoustic Signal Processing

14.1 MOOS-IvP Signal Processing Architecture

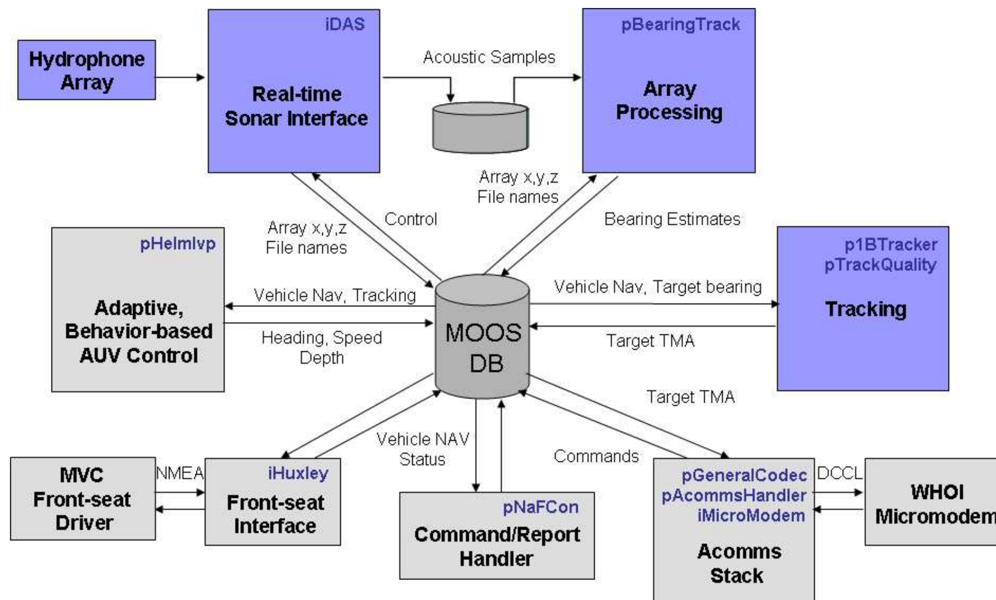

Figure 19: MOOS-IvP sonar AUV environment with inboard acoustic sensing and processing highlighted. Principal MOOS-IvP modules shaded in grey, incorporating the IvP-Helm, the handlers of commands and reports communicated with field control, and inboard signal processing. The acoustic modules consist of a receiving hydrophone array, for active systems a source, and a control and data acquisition system (iDAS), and array processing and target tracking modules.

The MOOS modules interfacing the acoustic data recorded on the array to the autonomy system are highlighted in green in Fig. 19. They include the `iDAS` data acquisition module, which write the high-volume digitized data to disk and publishes pointers to the data in the `MOOSDB`. The core on-board processing module is in the prototype passive acoustics MIT system represented by the module `pBearingTrack` which performs. It is responsible for filtering and beamforming the data to produce a Beam-Time Record (BTR). As the next step it includes a simple maximum beam detector, which identifies the loudest beam in each data section processed. The time history of the max imum beams is then continuously filtered by a stabilized bearing tracker, which publishes the result in a bearing state variable in the `MOOSDB`, containing UTC time, current auv position, and the bearing estimate with a calculated uncertainty.

Integrate of more advanced signal processing is done simply by replacing pBearingTrack with a more sophisticated processor, using the same definition of the interface to the `MOOSDB`, i.e. the same suite of published and subscribed variables. Obviously the interface can be modified and expanded as well, adding new parameters containing information available in the system which may be useful

#	Module Name	Module Description	Author	Size
1	**pBearingTrack**	Generic array processor module with conventional beamformer, detector and stabilized bearing tracker	Schmidt Poulsen Eickstedt	
2	**pCBF**	Generic array processor module with conventional beamformer. Defines `MOOS` interface for future advanced beamformers	Schmidt Poulsen	
3	**pBTracker**	Detector and stabilized bearing tracker for single, high-SNR acoustic source. Defines `MOOS` interface for future multi-target trackers.	Schmidt Poulsen Eickstedt	
4	**pTrackMonitor**	Interfaces the signal processing modules for beamforming and beaaring tracking to the `MOOS-IvP` state variables, e.g for maneuvering to enhance processing performance or breaking ambiguities.	Schmidt	
5	**p1HTracker**	Single bearing geo-tracker *Libraries: anrp_util, MOOS, MOOSGen, MOOSUtility*	Eickstedt Schmidt	
		Total unique lines of code		
		Total aggregate lines of code		

Table 2: Real-Time Acoustic array processing modules

to future advanced processing.

The next link in the on board processing chain in the MIT prototype is the process geographical coordinate tracker for the target, to produce a Target Motion Analysis (TMA). this process is performed by two MOOS modules, `p1HTracker` and `pTrackQuality`. `p1HTracker`subscribes to the bearing state published in the MOOSDB, and fuses it with the AUV's own navigation information to produce an estimate of the geographical position, heading and speed of the target, using either a least square or a Kalman filter approach. Once the number of tracking solutions exceeds a threshold set in the configuration file, a tracking state variable i published to the MOOSDB, again together with the ownship navigation and time information.

The second tracking module, `pTrackQuality` monitors the tracking solutions and once they are stable fall within a specified uncertainty for the location, heading and speed, a valid track state variable is published. This stable track estimate serves several purposes. First of, when published in the `MOOSDB` it will be translated into a *Track Report* by the autonomous command and control module `pNaFCon` and broadcast to the network via the acoustic communication stack. Also, the stable track solution will replace the expected track position communicated via the *Prosecute Command* which initiated the tracking, allowing the `IvP-Helm` to navigate the AUV to optimally achieve additional information or improved tracking solutions.

`pTrackQuality` is also responsible for terminating the prosecute mission that initiated it. Thus it will autonomously generate a *Deploy Command* if one or more criteria are satisfied, such as a specified number stable track solutions generated, the AUV approaching its operational radius or the border of its operations box. Depending on the configuration the redeploy will either be performed locally at the current position or at the previously specified deploy position.

A complete list of real-time acoustic array processing modules is given in Table 2. A more detailed description is given in the following chapter.

15 Acoustic Sensing and Processing Modules

15.1 pBearingTrack

15.1.1 Brief Overview

This MOOS module is the core onboard acoustic signal processing module. Whenever a new datafile is posted on the data disk, and flagged in the MOOSDB, `pBearingTrack` reads the file, and performs filtering and subsequent beamforming to create a Beam-Time Record (BTR) of the received acoustic intensity vs. bearing angle relative to the array. In the quiescent state (TRACKING = NO_TRACK) it runs a detector algorithm, which, if exceeding the detection threshold, identifies the beam with maximum power. This state is excuted in concert with the `pSearch` process, which is setting the control parameters for the search behaviors. Following a detection the processor will attemt to break the left-right ambiguity inherent to linear hydrophone arrays. It does so in concert with a set of HelmIvP behaviors which in this state (TRACKING = AMBIGUOUS) will execute a series of coordinated turns, which allow the alsorithm to break the ambiguity. Once this is achieved, `pBearingTrack` changes its state to (TRACKING = TRACKING) and uses a stabilized bearing tracker to produce a stable bearing estimate, which is then posted in the MOOSDB for use by other processes, such as the geo-tracker `p1HTracker`, and `pNaFCon` for generation of contact reports to the field control operators.

15.1.2 Parameters for the pBearingTrack Configuration Block

The acoustic processing parameters are defined in the configuration (.moos) file. The following configuration parameters are defined for `pBearingTrack`.

`def_array`: Array type, options are VSA, DURIP, SINGLE or DUAL. The first two are towed arrays, the last two are nose-mounted arrays on the MIT BF21 vehicles.

`def_sampling`: Samplinmg strategy in beam space. Options are `LINEAR` and `ARCCOSINE` for selecting linear sampling in wavenumber or beam angle, respectively.

`def_numchannels`: Total number of array channels contained in data files

`def_spacing`: Array element spacing of processed sub-array in m.

`def_numsamples`: Number of samples per file.

`def_fftlength`: Size of FFT applied for time-frequency fourier transforms.

`def_beams`: Number of beams from forward to backwards endfire directions.

`def_overlap`: Overlap in % of fourier transform windows for processing.

`def_integrate`: Integration time window length in seconds.

`def_soundspeed`: Sound spee in m/s used for calculating steeering vectors.

`def_centerfreq`: Center frequency in Hz for array processing

def_bandwidth: Frequency bandwidth in Hz for array processing

def_samplerate: Sampling rate in Hz.

def_goodchannels: Comma-separated list of channels used for processing. Note that channel numbers start with number 1.

def_freq_selection: Flag identifying frequency selection mode. 0 indicates selection as defined by the variable def_centerfreq and def_bandwidth. 1 for using a bin list definition in def_freq_bins.

def_freq_bins: Lisyt of frequency bins included in the processing, with 0 Hz being bin number 0. Any number of comma separated bin intervals may be specified. The bin intervals are indicated using the format first_bin:last_bin, as illustrated by examples in Listing ??.

Below is an example configuration block for pBearingTrack, here configured for the MIT DURIP towed array.

Listing 15.1 - An example pBearingTrack *configuration block (MIT DURIP towed array).*

```
 0 //--------- pBearingTrack Configuration block -----------------
 1 ProcessConfig = pBearingTrack
 2 {
 3   AppTick =   2
 4   CommsTick = 5
 5 // Array type:
 6   def_array        = DURIP
 7   def_sampling     = LINEAR
 8   def_numchannels  = 32
 9   def_spacing      = 0.75
10   def_numsamples   = 8000
11   def_fftlength    = 1024      // dF = 3.9063 Hz
12   def_beams        = 47
13   def_overlap      = 0.50
14   def_integrate    = 2.0
15   def_soundspeed   = 1495
16   def_centerfreq   = 900
17   def_bandwidth    = 200
18   def_samplerate   = 4000
19 // 500 Hz aperture
20 //  def_goodchannels = 1,2,3,4,5,6,8,10,12,14,16,18,20,22,24,26,28,29,30,31,32
21 // 1000 Hz aperture
22   def_goodchannels = 6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28
23   def_alpha        = 0.075
24   def_dep_angle    = 0.0
25   def_det_thresh   = 20.0
26   def_lrbeams      = 7
27   def_Alpha_Head   = 0.075
28   def_Turn_Rate_Threshold       = 0.1
29   def_Beam_Changed_Threshold    = 11
30   def_Max_Ubias_Iter            = 25
31   def_Abs_Uerror_Threshold      = 1.0
32   def_Theta_Min                 = 30.0
33   def_Detection_Count_Threshold = 5
34   def_filt_len = 1  //moving average filter length for vehicle heading and pitch
35   def_freq_selection = 0  //0=use center freq and BW. 1=use bin list
36 //  def_freq_bins = 51:129     // 200-500 Hz. BB
37 //  def_freq_bins = 51:53,76:78,102:104  //Tones 200, 300, 400  Hz
38   def_freq_bins = 205:258     // 800-100 Hz BB
```

```
39   def_Detector_type = 2
40   def_Detector_num_bins    = 15
41   def_Detector_Noise_First_Bin  = 10
42   def_Noise_Depth    = 5
43   def_Detector_Alpha    = 0.9
44 }
```

15.1.3 MOOS variables subscribed to by pBearingTrack:

.

MOOS variable	Type	Description	Published by
VEHICLE_ID	D	Node number for ownship.	pNaFCon
VSA_DIR	$	Directory where data files will be posted by DAS	iDAS
VSA_FRAME	D	Frame number for last posted data file	iDAS
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
NAV_DEPTH	D	Ownship depth	pHuxley
AEL_HEADING	D	Average true heading of array in degrees	pAEL
AEL_PITCH	D	Average pitch of array in radians	pAEL
AEL_X_OFFSET	D	Average x-offset of array relative to ownship	pAEL
AEL_Y_OFFSET	D	Average y-offset of array relative to ownship	pAEL
DEPLOY_STATE	$	Deploy state	pNaFCon
DEPLOY_MISSION	$	Deploy mission type	pNaFCon
PROSECUTE_STATE	$	Prosecute state	pNaFCon
PROSECUTE_MISSION	$	Prosecute mission type	pNaFCon
PREVIEW_CHAN	D	Channel to be previewed	-
PREVIEW_LENGTH	D	Number of samples to be previewed	-

15.1.4 MOOS variables published by pBearingTrack:

.

MOOS variable	Type	Description	Format
TRACKING	$	Tracking state. "NO_TRACK": No detection yet "AMBIGUOUS": Detection, but ambiguous bearing "TRACKING": Un-ambiguous bearing tracking	
BEARING_STAT	$	Comma-separated bearing state variables: node,state,bearing,x,y,beamno,sigma,utc	node=3,state=2, bearing=241, xp=1000,yp=2000, beamno=0, sigma=0.5, time=122..
BTR_DATA	$	Comma-separated list of beam powers	
PREVIEW_DATA	$	Comma separated preview data	

The principal output of `pBearingTrack` is the MOOS variable BEARING_STAT, which containes the current bearing state variables:

State variable	Description
node	Ownship VEHICLE_ID, e.g. 3 for Unicorn
state	Tracking state: 0: no detect, 1: ambiguous bearing; 2: Unambiguous bearing track.
bearing	Absolute, true bearing to target in degrees.
xp	Current UTM x-coordinate of array center
yp	Current UTM y-coordinate of array center
beamno	Beam number for detected beam
sigma	Bearing estimate uncertainty
time	UTC time.

15.2 p1HTracker

15.2.1 Brief Overview

This module attempts to determine a geophaphical tracking solution using the stabilized bearing track produced by `pBearingTrack`, combined with the associated record of the sensing platform motion. The track solution assumes the source is moving at constant speed and heading, and the track solution depends on the curvature of the sensing platform path. The curvature is achieved by the behaviors, described later, responsible for the platform motion in the *Tracking* sub-state, most notably the behavior `BHV_Attractor` which forces the platform towards the target, and the `BHV_ArrayAngle`, which attempts to keep the source broadside to the hydrophone array. The combination of the two, properly weighed using the behavior priorities, provides a robust target track in most cases.

15.2.2 Parameters for the p1HTracker Configuration Block

`range_max`: Maximum range accepted for track solution. Tracker will be reset if exceeded.

`speed_max`: Maximum speed accepted for track solution. Tracker will be reset if exceeded.

`range_guess`: Initial guess for target range.`p1HTracker` will search for the current target location along a radial at the current bearing, use this range as initial guess until a good track solution ghs been achieved, in which case the new guess is extrapolated from the previous track solution..

`init_meas`: Number of initial bearing measurements before tracking is initiated. Serves to stabilize the platform and array.

`min_measurements`: Minimum number of bearing estimates applied for the tracker. Thus a tracking solution will not be produced until `INIT_MEAS` + `MIN_MEASUREMENTS` bearing measurements have been received.

`Z_length`: Number of bearing measurements used for tracking solution in steady state.

`sampling_theta`: Minimum bearing change for accepting bearing measurement. Ensures only bearings carrying new track information are accepted.

Below is an example configuration block for`pSearch`,

Listing 15.2 - An example `p1HTracker` *configuration block .*

```
 1 ////////Configuration Block for p1HTracker ///////////////
 2 ProcessConfig = p1HTracker
 3 {
 4    AppTick   = 2
 5    CommsTick = 4
 6
 7    range_max = 5000
 8    speed_max = 10
 9    range_guess = 1200
10    init_meas = 12
11    min_measurements = 30
12    Z_length         = 120
13    sampling_theta   = 0.6 //degrees
14 }
```

15.2.3 MOOS variables subscribed to by p1HTracker:

.

MOOS variable	Type	Description	Published by
VEHICLE_ID	D	Node number for ownship.	pNaFCon
BEARING_STAT	$	Comma-separated bearing state variables: vehID,state,bearing,x,y,beamno,sigma,utc	pBearingTrack
DEPLOY_STATE	$	Prosecute state	pNaFCon
PROSECUTE_STATE	$	Prosecute state	pNaFCon
PROSECUTE_RANGE	D	Estimated target range. Used for computing required bearing rate	pNaFCon
NAV_SPEED	D	Ownship speed. Used for computing required bearing rate	pHuxley
TRACKING	$	Tracking state. Indicates whether target has been detected or not.. "NO_TRACK": No detection yet "AMBIGUOUS": Detection, but ambiguous bearing "TRACKING": Un-ambiguous bearing tracking "CLASSIFYING": Classifying sub-state	pBearingTrack
TRACKING_MODE	$	Tracking mode: "SINGLE": Single vehicle tracking "MULTIPLE": Multi-vehicle cross-bearing tracking	pMBTRacker
CLOSE_RANGE	$	Reset flag for geo trackers	pSearch
TARGET_ID	D	ID for target being prosecuted	pNaFCon
TGT_#_NAV_X	D	UTM x-coordinate for prosecuted target	pTransponderAIS
TGT_#_NAV_Y	D	UTM y-coordinate for prosecuted target	pTransponderAIS
TGT_#_NAV_UTC	D	UTC time for prosecuted target info	pTransponderAIS
TGT_#_NAV_HEADING	D	Heading fof prosecuted target	pTransponderAIS
TGT_#_NAV_SPEED	D	Speed of prosecuted target	pTransponderAIS

Notes: The tracker subscribes to the target state variables `TGT_#_X` etc. which are initially published by `pNaFCon` when receiving the *Prosecute Command*. They are used for generating an initial guess for use by the tracker. Once an acceptable tracking solution is achieved, the corresponding target state variables will be published by `p1HTracker` using the same MOOS variables. The state variables will also be published by `pTransponderAIS` when an incoming *Track Report* is received from a collaborating cluster node for the same target ID. The user should be aware that in cases where the cued target track is inaccurate, the publishing of these tracking solutions will affect any target-adaptive behaviors.

15.2.4 MOOS variables published by p1HTracker:

.

MOOS variable	Type	Description	Format
TRACK_STAT	$	Tracking state vector. node=,state=,x=,y=,heading=,speed=,time=	node=3,state=2,x=...
TGT_#_NAV_X	D	UTM x-coordinate for target track	
TGT_#_NAV_Y	D	UTM y-coordinate for target track	
TGT_#_NAV_UTC	D	UTC time for target track	
TGT_#_NAV_HEADING	D	Estimated heading for target	
TGT_#_NAV_SPEED	D	Estimated speed of target	
CLOSE_RANGE	$	Reset flag for geo trackers	TRUE FALSE
NY_TARGET	$	Current target, used as update for `BHV_Attractor`	contact=TGT_5

The principal product is the tracking state `TRACK_STAT`, with the second field being the state variable:

1. No tracking, Bearing ambigous
2. Tracking, valid solution

15.3 pMBTracker

15.3.1 Brief Overview

This MOOS module does a darn good job at doing what it is supposed to do

15.3.2 Configuration MOOS-block

15.3.3 MOOS variables subscribed to:

.

15.3.4 MOOS variables published:

.

Part VII
Network Command and Control

16 Topside Command and Control

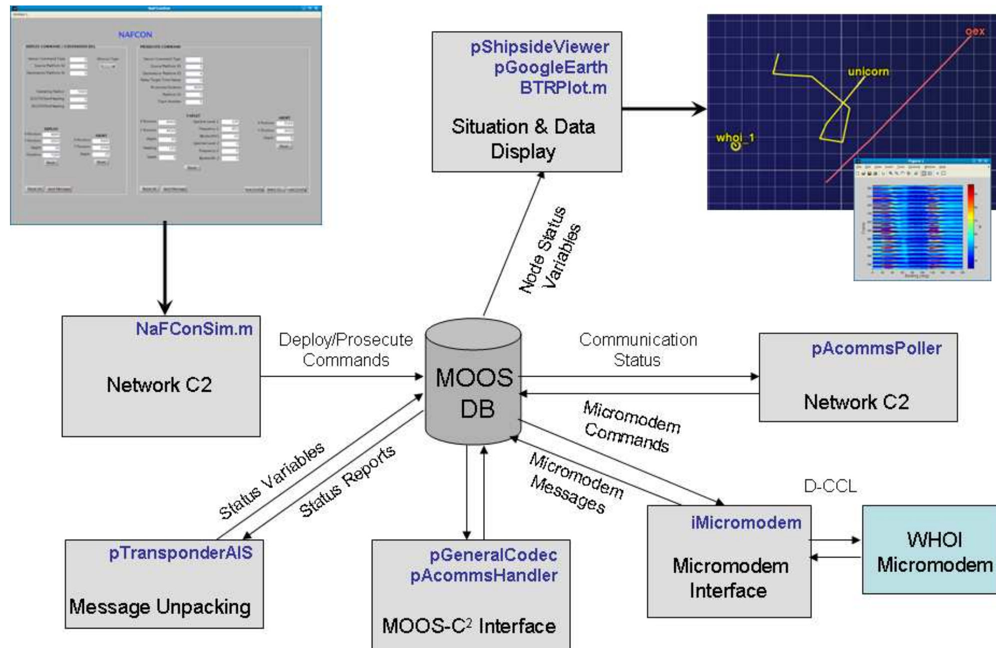

Figure 20: MOOS-IvP community for MIT autonomous network command and control topside.

16.1 Topside MOOS Community

The topside MOOS community provides the interface middle-ware between the field command and control operators and the network communication infrastructure. The topside MOOS community is shown schematically in Fig. 20.

The topside community is similar to the vehicle communities, except for the lack of an autonomy helm. Thus the communication stack is identical to that applied on the vehicles, including the modules `pGeneralCodec`, `pAcommsHandler`, `pAcommsPoller`, and `iMicroModem`.

Since the topside, lacking the helm, only has one state, the state transition manager module `pNaFCon` is not in general used on the topside.It is only used if the topside is broadcasting its own status reports, which may be desirable if operated from a ship, where the navigation information may be needed for the collision avoidance behaviors of the underwater assets.

The principal interpreter of the incoming messages containing node status and contact information is the standard `pTransponderAIS` process, configured for accepting incoming messages published in the MOOS variable `NAFCON_MESSAGES`. It parses the message into status variables that are compatible with the topside situational display driver, `pShipSideViewer`, which will display the node information geographically as shown in Fig. 21.

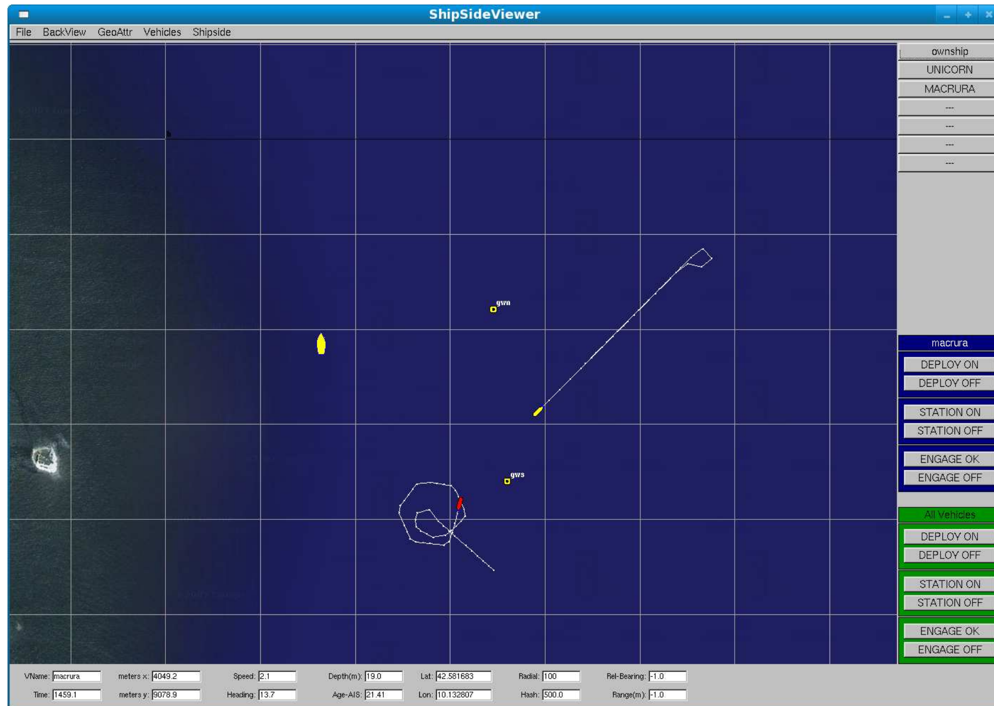

Figure 21: pShipSideViewer topside situational display

Another useful process running on the topside is `pNafconMessageViewer` which shows the complete content of the status and contact reports arriving from each node in the network, with an example from GLINT'08 in Fig. 22.

Finally, dedicated graphical tools are available for displaying measured and processed data received from the nodes, such as BTR and CTD records.

The topside command console is driven by the MATLAB script `NaFConSim.m`, the GUI of which is shown in Fig. 23. The name is an acronym for *Network and Field Control Simulator* because it originates as an MIT simulation of a comprehensive network control console developed by Penn State for the PLUSNet program. The GUI has two sections, one for issuing *Deploy Commands*, the other for issuing *Prosecute Commands*, with each allowing for a number of optional sub-states or mission types. Thus, a deploy may be a hexagonal loiter pattern or a racetrack. Also, deploy missions include environmental sampling missions involving vertical YoYo-s or horizontal zig-zag surveys. Similarly, prosecute missions may be selected to be target-adaptive or simple classical TMA survey patterns.

A complete list of the MOOS topside modules and utilities is given in Table **??**. A more detailed description is given in the following chapter.

16.2 Launching the Command and Control Topside

The topside situational display and communication infrastructure, a MOOS community AUV_Topside, Port 9123, is launched using the commands:

```
> cd ~/moos-ivp-local/missions/AUV_Topside/
```

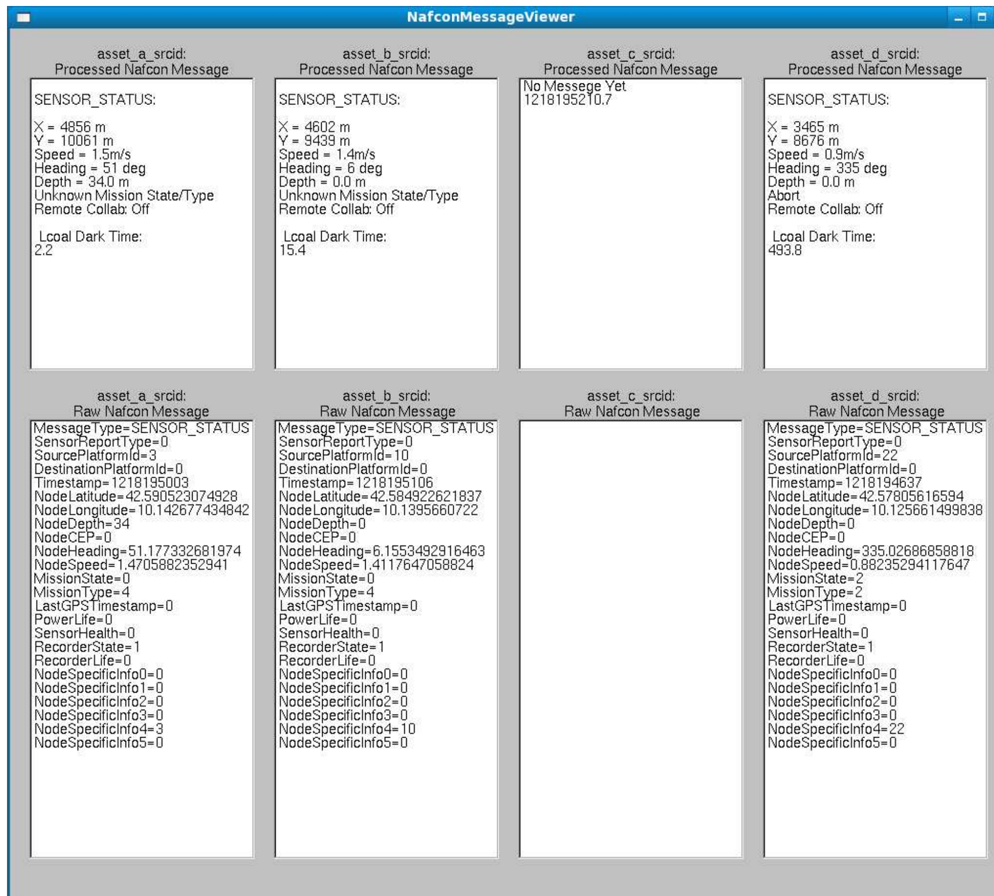

Figure 22: `pNafconMessageViewer` display shows the status reports in completeness, received from the network nodes.

#	Module Name	Module Description	Author	Size
1	**NaFConSim.m**	MATLAB GUI for generating commnds to nodes in the autonomous undersea network	Dumortier	
2	**pHuxley**	Front-seat driver interface. Required for publishing of UTM Datum only. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Balasuriya	
3	**pShipsideViewer**	Situational display *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Benjamin	
4	**uNafconMessageViewer**	Process for displaying messages received from active network nodes. *Libraries: MOOS, MOOSGen, MOOSUtility*	Cockrell	
5	**pGeneralCodec**	Generic module for Coding and Decoding of CCL modem messages *Libraries: MOOS, MOOSGen, MOOSUtility*	Schneider	
6	**pAcommsHandler**	Schedules acomms. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Schneider	
7	**pAcommsPoller**	Manages and schedules polling of other nodes on the modem network. Usually run on topside only, but may be activated on mobile node for relaying. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Schneider	
8	**iMicroModem**	Driver process for WHOI micromodems *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Grund	
		Total unique lines of code		
		Total aggregate lines of code		

Table 3: MOOS modules for Topside Command and Control of Undersea Network

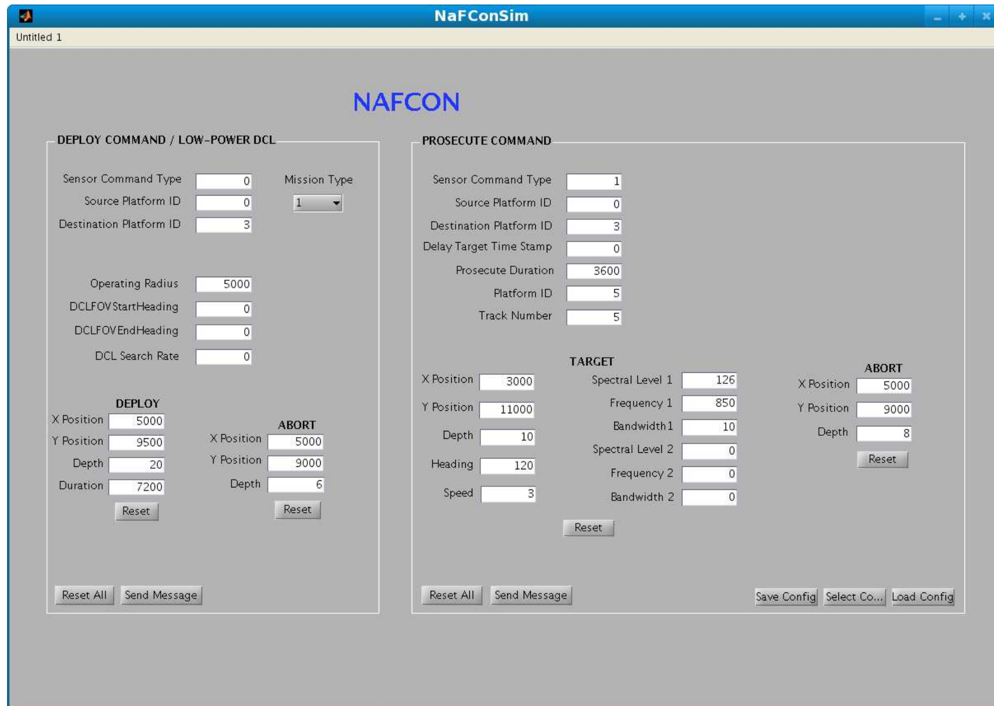

Figure 23: Network and Field Control (NaFCon) GUI for issuing Deploy and Prosecute commands to network nodes.

```
> pAntler AUV_Topside_base.moos &
> pAntler AUV_Topside_CommsStack.moos &
```

This will bring up the situational display using **pShipSideViewer**, with an example shown in Fig. 28.

The topside command and control GUI, shown in Fig. 23 is typically launched on a separate desktop using the commands:

```
> cd ~/moos-ivp-local/src/matlab/NaFConSimTop
> matlab
  >> NaFConSim
```

The GUI will publish the commands to the topside MOOSDB in the topside MOOS community AUV_Topside on Port 9123, and will be broadcast to the virtual network by the modem communication stack.

17 Topside Command and Control Modules

17.1 NaFConSim

17.1.1 Brief Overview

`NaFConSim.m` is a MATLAB GUI which serves as the principal command and control console for the undersea network.

17.1.2 Parameters for the NaFConSim Configuration Block

The following configuration parameters are defined for `NaFConSim`.

`SerialComms`: The `NAFCONSIM` GUI may be configured for either communicating through a serial port to a MOOS community on another computer, or directly to the MOOSDB. The latter is the normal operation, with `SERIALCOMMS = false`.

`MOOSComms`: Must be set to `MOOSComms = true` for communication directly to the MOOSDB. Must always be the negated of `SerialComms`.

`Port`: Serial port used for communication. Only significant for `SERIALCOMMS = true`.

`BaudRate`: Baudrate used for communication via serial port. Only significant for `SERIALCOMMS = true`.

`Verbose`: Verbosity. Only true for testing and dbugging..

`Streaming`: Serial protocol.Only significant for `SERIALCOMMS = true`.

`SERIAL_TIMEOUT`: Timeout for serial communication in seconds .Only significant for `SerialComms = true`.

`SUBSCRIBE`: Identifies MOOS variables to be subscribed to by `NaFConSim`..

Below is an example configuration block for `NaFConSim`,

Listing 17.1 - An example for `pNaFConSim` *configuration block .*

```
 1 /////////// Global Variables ////////////////
 2 ServerHost = localhost
 3 ServerPort = 9123
 4 ////////Configuration Block for pNaFConSim ///////////////
 5 ProcessConfig = NaFConSim
 6 {
 7   AppTick     = 10
 8   CommsTick   = 10
 9   Port     = COM6
10   BaudRate    = 4800
11   Verbose     = false
12   Streaming   = false
13   MOOSComms = true
14   SerialComms = false
15   SERIAL_TIMEOUT = 10.0
16   SUBSCRIBE   = DB_TIME @ 0
17   SUBSCRIBE   = LAT_ORIGIN @ 0
18   SUBSCRIBE   = LONG_ORIGIN @ 0
19 }
```

17.1.3 MOOS variables subscribed to by pNaFConSim:

.

MOOS variable	Type	Description	Published by
DB_TIME	D	UTC time for time stamping of commands	MOOSDB
LAT_ORIGIN	D	Latitude of local UTM Datum	pHuxley
LONG_ORIGIN	D	Longitude of local UTM Datum	pHuxley

17.1.4 MOOS variables published by NaFConSim:

.

MOOS variable	Type	Description	Format
PLUSNET_MESSAGE	$	Human-readable deploy or prosecute message to be coded, scheduled and transmitted by the communication stack.	

17.2 pHuxley

17.2.1 Brief Overview

This module or one of the other front-seat driver interface modules the solely responsible for publishing the local UTM Datum to the MOOSDB, and must therefore be part of the topside community. Since there is no frontseat driver on the topside, `pHuxley` should be operated in the simulation mode. The `pHuxley` process is decribed in detail in Section 13.7, and we shall here only describe the configuration particular to the topside use.

17.2.2 Parameters for the Topside pHuxley Configuration Block

Below is an example configuration block for the topside `pHuxley` process,

Listing 17.2 - An example of a topside `pHuxLey` *configuration block .*

```
 1 ProcessConfig = pHuxley
 2 {
 3    AppTick   = 4
 4    CommsTick = 4
 5
 6    VarNamePrefix = GATEWAY
 7 // Huxley IP address & port
 8    HuxleyHost = localhost
 9    HuxleyPort = 29500
10    HuxleySim = true
11    Verbose = true
12 }
```

17.3 pGeneralCoDec

17.3.1 Brief Overview

This MOOS module is the generic Codec uased for in- and outgoing acomms messages, used consistently throughout the network. It is described in detail in Section 13.8. The configuration for the coding and decoding of the messages, defined in xml-files, should be set identically on all platforms.

17.4 pAcommsHandler

17.4.1 Brief Overview

This MOOS module is the crucial queueing handler for in- and outgoing acomms messages, used consistently throughout the network. It is described in detail in Section 13.9. We shall here only give an example of a configuration block that includes items unique to the topside, such as the queueing andscheduling of *Deploy* and *Prosecute Commands* to be transmitted to the mobile network nodes.

17.4.2 Topside pAcommsHandler Configuration Block

Below is an example configuration block for the topside `pAcommshandler` process, Note that this confguration assumes that the message coding and decoding is performed using the generic CoDec `pGeneralCodec`. If used together with the dedicated CCL CoDec `pFramer`, the configuration should use the `send_CCL` and `receive_CCL` variables instead of the generic ones, as decribed in Section 13.9.

Listing 17.3 - An example of a topside `pAcommsHandler` *configuration block .*

```
 1 ProcessConfig = pAcommsHandler
 2 {
 3    AppTick    = 4
 4    CommsTick  = 4
 5    verbosity = verbose
 6 // all case insensitive
 7    modem_id = 0
 8
 9 // information about iMicroModem
10    micromodem_command_var = MICROMODEM_COMMAND
11    micromodem_data_var = MICROMODEM_DATA
12 // Commands
13 // send = VarName,
14                       //     VariableID
15                       //        [Ack]
16                       //           [BlackoutTime]
17                       //              [MaxQueue]
18                       //                  [NewestFirst]
19                       //                     [Priority]
20                       //                         [Priority Time Constant]
21    send = OUT_DEPLOY_HEX_30B,    10, 1,  0,  1,   1,  10,  120
22    send = OUT_PROSECUTE_HEX_30B, 11, 1,  0,  1,   1,  10,  120
23 // Reports
24 // receive = VarName, VariableID
25    receive = IN_STATUS_HEX_30B,  12
26    receive = IN_CONTACT_HEX_30B, 13
27    receive = IN_TRACK_HEX_30B,   14
28    receive = IN_CTD_HEX_254B,     3
29    receive = IN_BTR_HEX_254B,     4
30 }
```

17.5 pAcommsPoller

17.5.1 Brief Overview

This MOOS module does a darn good job at doing what it is supposed to do

17.5.2 Configuration MOOS-block

17.5.3 MOOS variables subscribed to by pAcommspoller:

.

17.5.4 MOOS variables published by pAcommsPoller:

.

17.6 iMicroModem

17.6.1 Brief Overview

This MOOS module does a darn good job at doing what it is supposed to do

17.6.2 Configuration MOOS-block

17.6.3 MOOS variables subscribed to by iMicroModem:

.

17.6.4 MOOS variables published by iMicroModem:

.

17.7 uNafconMessageViewer

17.7.1 Brief Overview

This MOOS module displays summarized versions of NaFCon messages, along with the entire original message.

17.7.2 Topside uNafconMessageViewer Configuration Block

uNafconMessageViewer can display messages from up to 4 sources at once. Each source is denoted by its source's identification number, which is in the Nafcon message.

Below is an example configuration block for `uNafconMessageViewer`.

Listing 17.4 - An example of `uNafconMessageViewer` *configuration block .*

```
1 {
2  AppTick   = 5
3  CommsTick = 25
4
5  asset_a_srcid = 1
6  asset_b_srcid = 2
7  asset_c_srcid = 3
8  asset_d_srcid = 4
9 }
```

17.7.3 MOOS variables subscribed to by uNafconMessageViewer:

.

MOOS variable	Type	Description	Published by
NAFCON_MESSAGES	$	Nafcon messages from communications stack	pFramer
LAT_ORIGIN	D	Latitude of local UTM Datum	pHuxley
LONG_ORIGIN	D	Longitude of local UTM Datum	pHuxley

17.7.4 MOOS variables published by uNafconMessageViewer:

None

Part VIII
Sensing Network Simulation Environment

18 MIT Undersea Autonomous Network Simulator

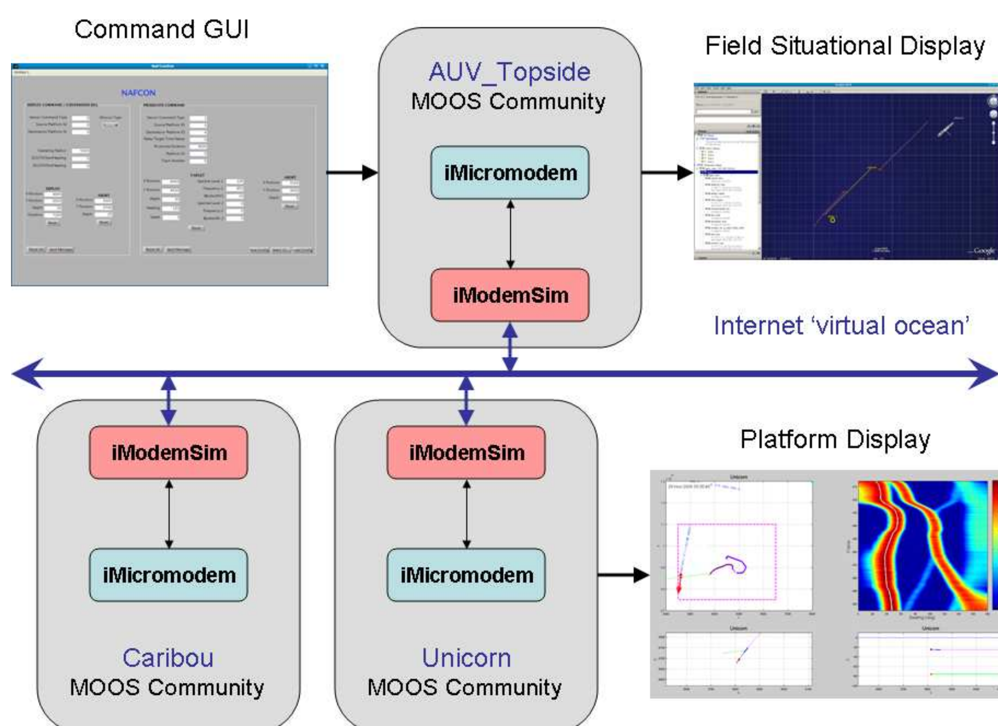

Figure 24: Architecture of MOOS-IvP Undersea Autonomous Network Simulator. The topside MOOS Community, identical to that applied in the field, is connected to all MOOS communities on the local network, including any number of AUV and ASC communities, communicating using the WHOI Micromodem through the `iMicroModem` MOOS utility.

The cornerstone of the MIT multi-node undersea network simulator is the MOOS utility `iModemSim`, which simulates an undersea micromodem network used in the field as shown in Fig. 10 by transparently connecting all MOOS-communities on the local area network, or the same computer, running this process. The utility incorporates physically realistic message transmission uncertainty, intermittency and latency, as well as message collision, and therefore provides a high-fidelity 'virtual ocean' environment for realistic simulation of an undersea network. The network simulation architecture is shown in Fig. 24.

The topside command and control MOOS community `AUV_Topside` is operated unchanged from that used in field deployments, except for the physical serial port to the WHOI micromodem gateway being replaced by a virtual serial port to the `iModemSim` process. Thus, the topside command and control GUIs and the situational and nodal status displays are transparently representing an actual operational environment.

Similarly, the MOOS Communities for an arbitrary number of underwater vehicles, or surface

craft are running on one or more desktop or laptop computers - or on the actual vehicle payload - connected to the 'virtual ocean' by `iModemSim`. Each 'virtual vehicle' are commanded from the topside using CCL modem commands with realistic throughput statistics, and will transmit status reports via the virtual modems to the network when polled to do so. The architecture allows for all virtual nodes to share environmental and situational information. Thus, the environment allows for fully realistic simulation of adaptive and collaborative autonomy by the network assets.

In addition to the field-level display tools applied by the topside community, the simulator incorporates several graphical tools for real-time situational display of nodal information, including results of processed sensor data, such as the inboard processing of acoustic data collected by hydrophone arrays on sonar AUVs, as described in the following.

18.1 Sonar-AUV Simulator

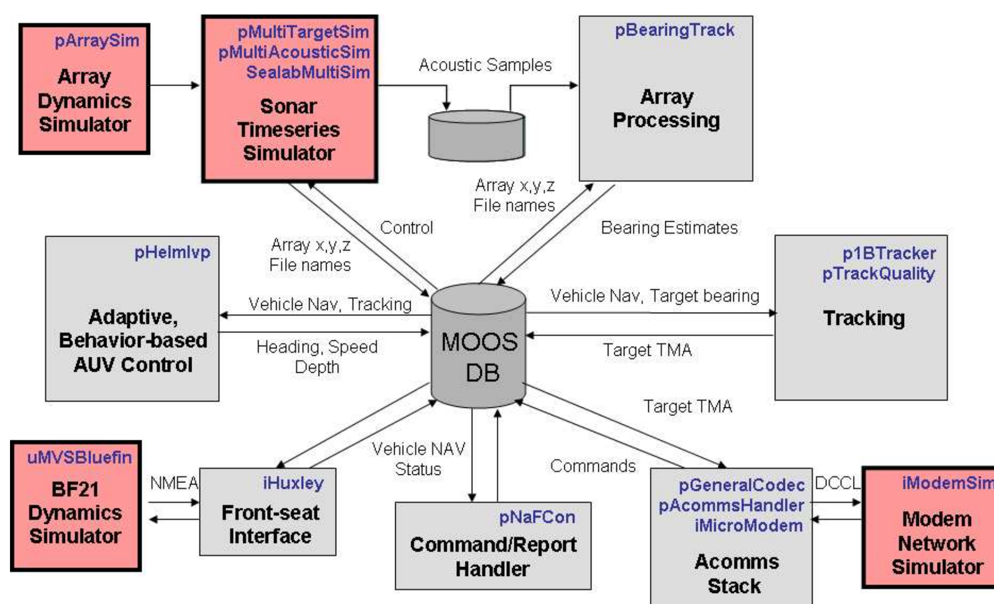

Figure 25: MOOS-IvP Sonar-AUV Simulation Environment: The simulation environment is identical to the AUV payload MOOS environment, except for the 'outside world' interfaces, which are replaced by a set of simulation tools which use the same interface definitions as the actual vehicle hard- and software, making it transparent to the MOOS=-IvP environment whether it is operating in the vehicle or a stand-alone computer. .

The power of the MOOS-IvP simulation environment is that all processes can be operated unchanged, as long as the hardware and software handling the interface between actuators and sensors and the MOOSDB are replaced by dedicated simulation modules publishing and subscribing to the same MOOS variables as the on-board interfaces. Thus, a full simulation capability has been established developing a set of payload connection simulators, which can replace the actual hardware and software transparently to all other MOOS processes connected to the MOOSDB, as illustrated schematically in Fig. 25, where the blue-shaded inboard systems in Fig. 11 are replaced by their red-shaded simulator equivalents. Also, the simulation modules may be used in any combination with the real system components. Thus, the CTD simulator has been used on a vehicle during scientific missions with a broken CTD unit, due to the fact that the BF21 front-seat driver requires

CTD data for operating. Also, the target simulator has been applied in the field in cases of a malfunctioning acoustic array.

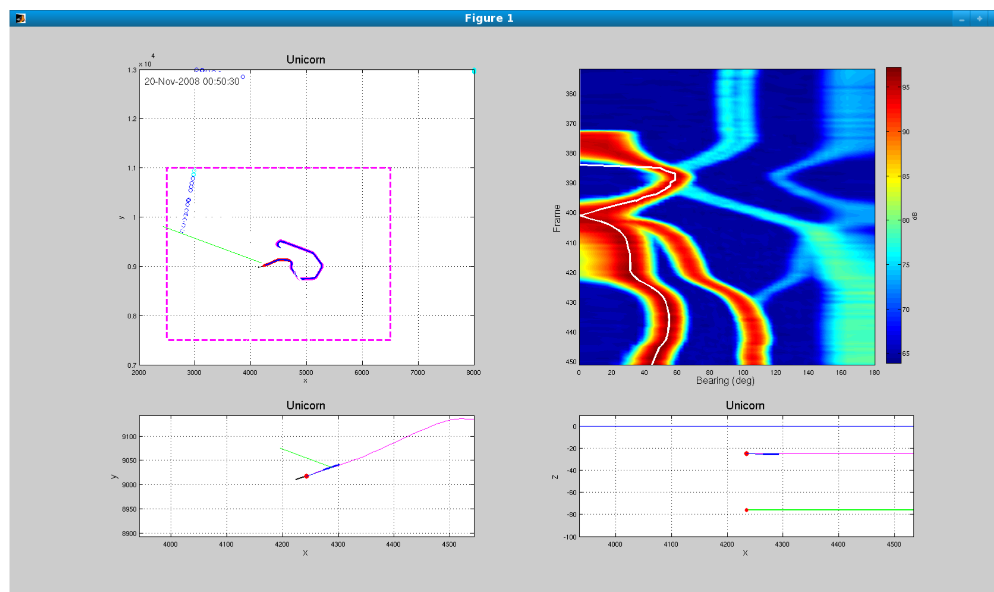

Figure 26: `small_uVis.m`: Real-time display of acoustic sensing mission. The upper left frame shows the auv/array and target history for a sensing mission simulation. The two lower frames show zooms of the auv/array geometry in the horizontal and vertical, while the upper right shows the computed Beam-Time Record (BTR) achieved from the simulated time series, with the real-time stabilized bearing-track estimate indicated by the white curve..

In addition to the system simulator modules, the source tree incorporates several graphical tools for use with the simulator, in addition to the standard topside displays described elsewhere. Thus, the MATLAB script `small_uVis.m` generates a real-time display of auv/array dynamics, and real-time acoustic processing results, as shown in Fig. 26.

Another MATLAB module, `SourceSim.m` activates a GUI, shown in Fig. 27 for adding new acoustic targets to be simulated by `pMultiTargetSim`. Each new target is identified by an initial position, speed and heading, and its acoustic properties, and added to the list of active targets maintained by `pMultiTargetSim`, when activated with 'Apply'..

A complete list of the simulation modules are given in Tables 4 and 5. A more detailed description is given in the following chapter.

18.2 Running a Simulation Session

When running a simulation session it is recommended to assign a desktop to each AUV or ASC, and two for the topside: one for the situational display, one for the command GUIs. The first step is to prepare the mission files for use with the simulator.

18.2.1 pAntler Simulation MOOS Block

The adaptation of a mission file to the simulation environment is straightforward, involving only a few changes and additions. Thus, apart for defining platform dependent file paths etc., the actual

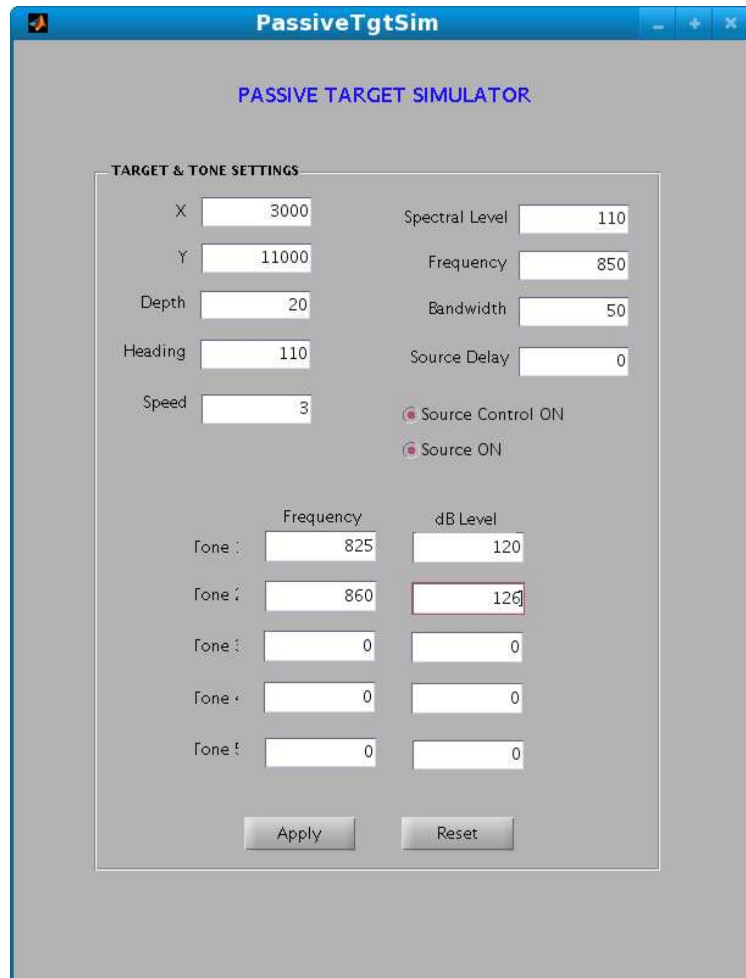

Figure 27: `PassiveTgtSim.m`: GUI for activating acoustic sources to be dynamically simulated by `pMultiTargetSim`.

#	Module Name	Module Description	Author	Size
1	**uMVSBluefin**	Dynamic simulator for Bluefin21 AUV	Battle	
2	**pArraySim**	Towed array simulator used together with uMVSBluefin for coupled dynamics simulation. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Schmidt	4,803
3	**iModemSim**	Simules a modem network. Communicates with iModemSim in all MOOS communities on local network. Handles propagation latency and transmission loss, and collisions. *Libraries: anrp_util, MOOS, MOOSGen, MOOSUtility*	Patrikalakis	1,474 **24,655**
4	**pTargetSim**	Single target simulator used in conjunction with the low-fidelity target bearing simulator **pBearingSim** *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Cockrell	
5	**pMultiTargetSim**	Dynamically simulates an arbitrary number of targets for the multi-source acoustic simulators. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Schmidt	
6	**pBearingSim**	Low-fidelity, single target bearing estimator replacing the onboard acoustic array processing module **pBearingTrack**. Highly efficient and useful for multi-auv collaborative mission simulations, and for onboard bearing simulation in case of hydrophone array malfunctioning.. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Eickstedt	
7	**pMultiAcousticSim**	Medium-fidelity acoustic simulator generating timeseries received on array simulated by **pArraySim**, incorporating ambient noise and signals from targets simulated by **pMultiTargetSim**. Uses simple white-noise model, and target signals simulated as plane waves, but with cylindrical spreading loss and bottom loss included. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Cockrell Schmidt	
8	**pGPSSim**	Simulates received GPS information when vehicle is surfaced. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Schmidt	
9	**uCTDSim2**	Simulates CTD data by interpolating in HOPS-generated virtual ocean or real ocean database. Can be used in field with malfunctioning CTD unit. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Lum	
10	**uBathy**	Simulates bathymetry data by interpolating in bathymetry table. Used for simulating altimeter data. *Libraries: mbutil, MOOS, MOOSGen, MOOSUtility*	Lum	
		Total unique lines of code		
		Total aggregate lines of code		

Table 4: MIT AUV simulation environment. C++ simulation Modules.

#	Module Name	Module Description	Author	Size
1	**ArraySim.m**	Matlab version of array dynamics simulator	Dumortier	
2	**SealabMultiSim.m**	High–fidelity acoustic simulator generating timeseries received on array simulated by **pArraySim**, incorporating ambient noise and signals from targets simulated by **pMultiTargetSim**. Uses SEALAB synthetic sonar environment for envuironmental acoustic modeling. *Libraries:*	Schmidt	
3	**PassiveTgtSim.m**	GUI for adding acoustic sources simulated by **pMultiTarget-Sim**. The source numbering is sequential and transparent to the user. *Libraries:*	Dumortier Petillo	
4	**small_uVis.m**	Graphical display of AUV mission, including array dynamics, and processed acoustic data. Applied on each simulated platform. May also be used in conjunction with **uPlayback** for replay of actual at-sea missions. *Libraries:*	Schmidt	
Total unique lines of code				
Total aggregate lines of code				

Table 5: MIT AUV simulation environment - MATLAB modules.

real-time mission file is to be modified in the `pAntler` block, simply adding the start-up of the relevant simulation utilities, and in the configuration block for the back-seat driver interface. The following shows a typical `pAntler` configuration block for an acoustic sensing mission, with two groups of simulation modules added. The first concerns the basic vehicle sensor and activator operation, including the auv dynamics simulator, the CTD and GPS sensor simulations, etc. The second group are payload sensing simulators, involving the dynamics of the array, and the simulation of the relevant environmental acoustics. In the example given here, the efficient medium-fidelity acoustic simulator is used. In cases where the MATLAB-based high-fidelity simulator is applied, the `SealabMultiSim.m` is executed after the pAntler start-up.

Listing 1 - An example of `pAntler` *configuration block for acoustic sensing simulation.*

```
 0 //-----------------------------------------------------------------
 1 // Antler configuration  block
 2  ProcessConfig = ANTLER
 3  {
 4     MSBetweenLaunches = 500
 5 //Core MOOS processes
 6     Run = MOOSDB             @ NewConsole = false
 7     Run = pEchoVar           @ NewConsole = true
 8     Run = pHuxley            @ NewConsole = true
 9 // AUV Simulator processes
10     Run = uMVS_Bluefin       @ NewConsole = false
11     Run = pMarinePID         @ NewConsole = true
12     Run = iModemSim          @ NewConsole = true
13     Run = pGPSSim            @ NewConsole = true
14     Run = uCtdSim2           @ NewConsole = true
15     Run = uBathy             @ NewConsole = true
16 // Array Dynamics and Acoustic simulators
17     Run = pMultiTargetSim    @ NewConsole = true
```

```
18     Run = pArraySim         @ NewConsole = true
19     Run = pMultiAcousticSim @ NewConsole = true
20 // Communication Processes
21     Run = iMicroModem       @ NewConsole = true
22     Run = pAcommsHandler    @ NewConsole = true
23     Run = pGeneralCodec     @ NewConsole = true
24     Run = pNaFCon   @ NewConsole = true
25     Run = pTransponderAIS   @ NewConsole = true
26 // Payload Processes
27     Run = iDAS              @ NewConsole = true
28     Run = pAEL              @ NewConsole = true
29     Run = pSearch           @ NewConsole = true
30     Run = pBearingTrack     @ NewConsole = true
31     Run = p1HTracker        @ NewConsole = true
32     Run = pTrackQuality     @ NewConsole = true
33 // Autonomous Control
34     Run = pClusterPriority  @ NewConsole = true
35     Run = pHelmIvP          @ NewConsole = true
36 // Logging
37     Run = pLogger           @ NewConsole = false
38 // Initial DEPLOY
39     Run = pMessageSim       @ NewConsole = true
40 // Watchdog
41     Run = uProcessWatch     @ NewConsole = true
42 // Display and Monitoring
43     Run = uMS               @ NewConsole = false
44 }
```

18.2.2 pHuxley Simulation MOOS Block

In addition to the dedicated simulation modules, the back-seat driver interface, for the Bluefin vehicles `pHuxley` must be configured for simulation, allowing it to interact with the vehicle dynamics simulator, rather than the actual vehicle 'front-seat driver'.

Listing 2 - An example of `pHuxley` *configuration block for AUV mission simulation.*

```
 0 //-------------------------------------------------------------
 1 ProcessConfig = pHuxley
 2 {
 3   AppTick   = 4
 4   CommsTick = 4
 5   VarNamePrefix = UNICORN
 6 // Huxley IP address & port
 7   HuxleyHost = localhost
 8   HuxleyPort = 29500
 9   HuxleySim = true
10   Verbose = true
11 }
```

18.2.3 iModemSim Setup

The first step is to set up a set of virtual serial ports, one for each 'virtual modem' for the 'virtual ocean' modem simulator `iModemSim`. This has to be done as *superuser*:

```
> su
Password: xxxxxxxx
> serial_loopback /dev/ttyLOOPA1 /dev/ttyLOOPA2 &
> serial_loopback /dev/ttyLOOPB1 /dev/ttyLOOPB2 &
> serial_loopback /dev/ttyLOOPC1 /dev/ttyLOOPC2 &
```

..
..

18.2.4 AUV Mission Launch

The next step is launching the AUV mission, as specified in in the mission file, here *Unicorn_DURIP_sim.moos*, including a simulation pAntler block as described above:

```
> cd ~/moos-ivp-local/missions/AUVs/Unicorn_sim
> pAntler  Unicorn_DURIP_sim.moos &
```

To start the node display, use

```
> cd ~/moos-ivp-local/src/matlab/Macrura_uVis/
> matlab -nojvm < pianosa_9000.m
```

Note that this is the version using port 9000, and the operation box used in GLINT'08 at Pianosa Island, Italy. For other AUVs simulated on the same computer, a different port must be used, and the operation box obviously will be changed.

18.2.5 Topside Launch

The topside situational display and communication infrastructure, a MOOS community AUV_Topside, Port 9123, is launched using the commands:

```
> cd ~/moos-ivp-local/missions/AUV_Topside/
> pAntler AUV_Topside_base.moos &
> pAntler AUV_Topside_CommsSim.moos &
```

This will bring up the situational display using pShipSideViewer, with an example shown in Fig. 28.

The topside command and control GUI, shown in Fig. 29 is typically launched on a separate desktop using the commands:

```
> cd ~/moos-ivp-local/src/matlab/NaFConSimTop
> matlab
  >> NaFConSim
```

The GUI will publish the commands to the topside MOOSDB in the topside MOOS community AUV_Topside on Port 9123, and will be broadcast to the virtual network by the modem communication stack.

The final piece of the simulator is to launch the GUI activating acoustic sources for simulating targets and interferers. This is performed using the commands:

```
> cd ~/moos-ivp-local/src/matlab/PassiveTgtSim/
> matlab
  >> PassiveTgtSim
```

bringing up the source simulator GUI shown in Fig. 27. The PassiveTgtSim process connects to the topside MOOS community, and the target parameters are broadcast to all nodes in the virtual network via a special modem message.

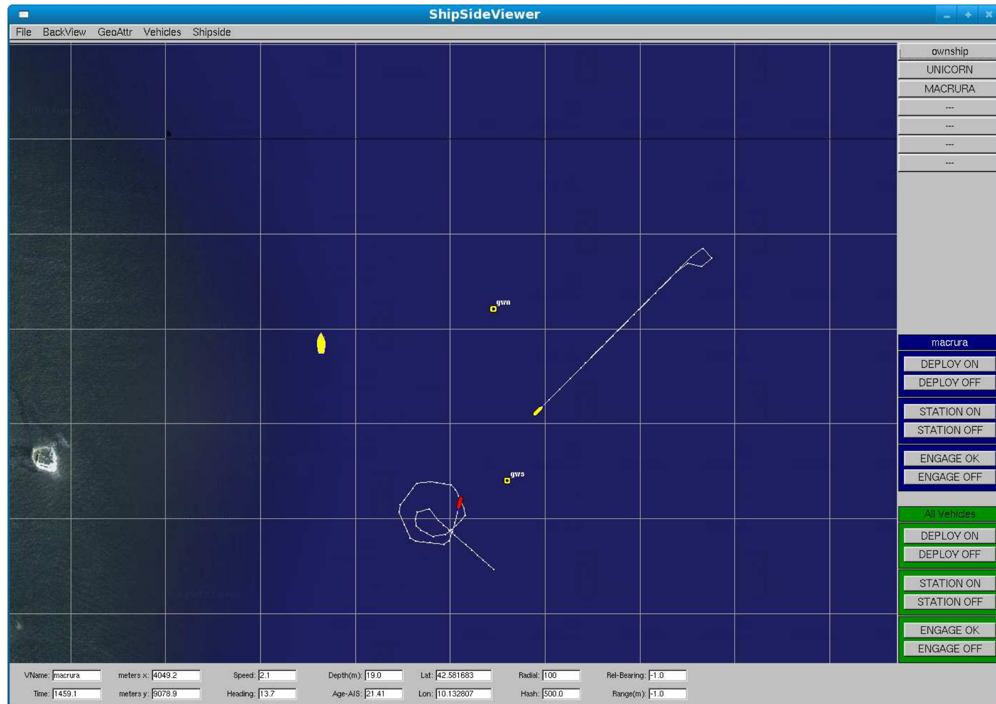

Figure 28: pShipSideViewer situational display

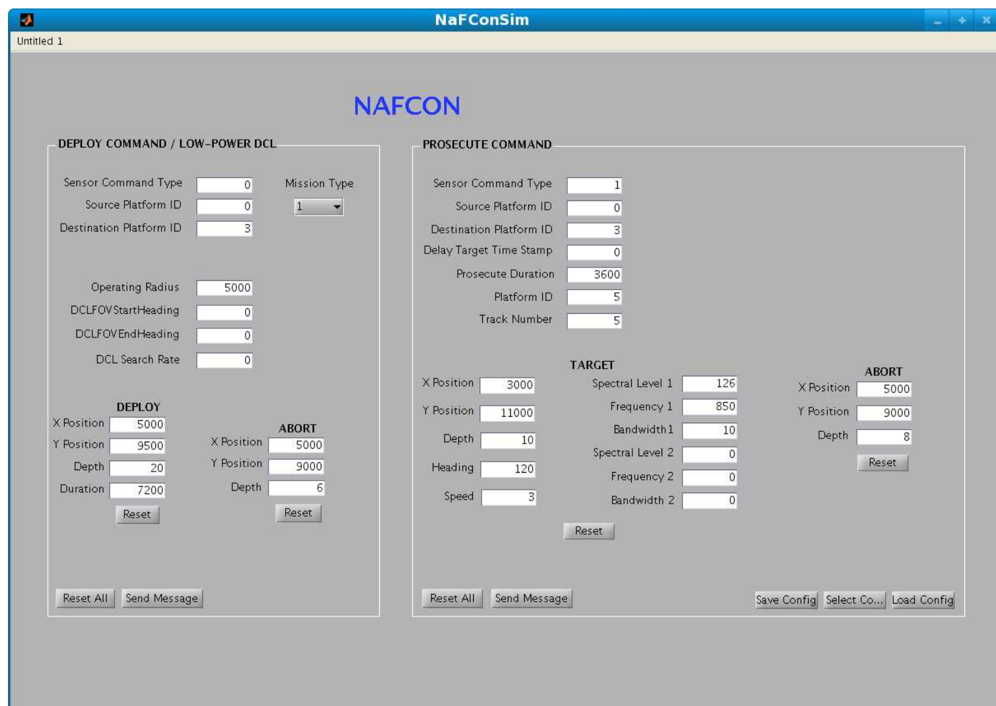

Figure 29: Network and Field Control (NaFCon) GUI for issuing Deploy and Prosecute commands to network nodes.

19 Sonar AUV Simulation Modules and Utilities

19.1 pTargetSim

19.1.1 Brief Overview

This MOOS module simulates a single target moving along a straight-line path. It was the precursor to `pMultiTargetSim` and solely is used together with the low-fidelity target bearing simulator `pBearingsSim`. Thus, it cannot be used with any of the acoustic timeseries simulators, where `pMultiTargetSim` should be used.

19.1.2 Parameters for the pTargetSim Configuration Block

The following configuration parameters are defined for `pTargetSim`.

`rangeCrit:` Critical range beyond which target will be ignored.

Below is an example configuration block for the `pTargetSim`.

Listing 19.1 - An example `pTargetSim` *configuration block.*

```
 1  LatOrigin  = 42.3584
 2  LongOrigin = -71.08745
 3
 4 // ------------ pTargetSim Configuration block ----------
 5 ProcessConfig = pTargetSim
 6 {
 7   AppTick = 4
 8   CommsTick = 4
 9
10   rangeCrit     = 5000 // maximum source range in meters
11 }
```

The `LatOrigin` and `LonOrigin` parameters on lines 1-2 are not specific to `pTargetSim`, but are specified at the global level in a MOOS file. They are needed to allow the conversion between local x-y coordinates and latitude-longitude coordinates. They are mandatory parameters and `pTargetSim` will refuse to launch if they are lacking - but they are mandatory for other common MOOS applications as well.

19.1.3 MOOS variables subscribed to by pTargetSim:

.

MOOS variable	Type	Description	Published by
TARGET_CONTROL	$	"ON": Sources controlled by PassiveTgtSim.m, preample = T_ "OFF": Sources controlled by Prosecute message, preample = TARGET_	PassiveTgtSim.m
T_STATE TARGET_STATE	$	Comma-separated list of source parameters: x,y,depth,heading,speed,freq,bw,spl,delay,utc,number	PassiveTgtSim.m pNaFCon
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
DEPLOY_STATE	$	Deploy state	pNaFCon
DEPLOY_MISSION	$	Deploy mission type	pNaFCon
PROSECUTE_STATE	$	Prosecute state	pNaFCon
PROSECUTE_MISSION	$	Prosecute mission type	pNaFCon

19.1.4 MOOS variables published:by pTargetSim

.

MOOS variable	Type	Description	Format
TRACK_CONTROL	$	Target on/off flag. Set to ON in prosecute state and if deploy mission is 0.	"ON" / "OFF"
TARGET_RESET	$	Set to TRUE when new TARGET_STATE received	"TRUE" / "FALSE"
SIM_TARGET_XPOS	D	Current UTM x-coordinate of target.	2134.5
SIM_TARGET_YPOS	D	Current UTM y-coordinate of target.	5431.3

19.2 pBearingsSim

19.2.1 Brief Overview

This MOOS module provides a bearing estimate with gaussian noise for a single target simulated by pTargetSim. This is a low-fidelity bearing estimator which does not perform any sprocessing of array data, but simply computes the current bearing to the target and publishes the bearing state variable in the same format as the on-board signal processing modules. On the other hand, it is highly efficient and therefore well suited for developing adaptive and collaborative behaviors not sensitive to realistic environmental acoustic uncertainty.

19.2.2 Parameters for the pBearingsSim Configuration Block

The following configuration parameters are defined for pBearingsSim.

detection_range: Detection range for target in meters. If target is outside the tracking state is set to TRACKING = NO_TRACK. When targets enter the detection range the state is changed to TRACKING = AMBIGUOUS, and when the left-right ambiguity is resolved, to TRACKING = TRACKING.

sigma: Standard deviation of bearing estimate. used to make the bearing estimate more realisic in regard to array processing uncertainty.

Below is an example configuration block for the pBearingsSim.

Listing 19.2 - An example pBearingsSim *configuration block.*

```
 1  LatOrigin  = 42.3584
 2  LongOrigin = -71.08745
 3
 4 // ------------ pBearingsSim Configuration block ----------
 5 ProcessConfig = pBearingsSim
 6 {
 7   AppTick = 2
 8   CommsTick = 5
 9
10   detection_range = 1500 // Detection range in meters
11   sigma           = 1.0  // Bearing standard deviation in degrees
12 }
```

The LatOrigin and LonOrigin parameters on lines 1-2 are not specific to pBearingsSim, but are specified at the global level in a MOOS file. They are needed to allow the conversion between local x-y coordinates and latitude-longitude coordinates. They are mandatory parameters and pBearingsSim will refuse to launch if they are lacking - but they are mandatory for other common MOOS applications as well.

19.2.3 MOOS variables subscribed to by pBearingsSim:

.

MOOS variable	Type	Description	Published by
VEHICLE_ID	D	Node number for ownship.	pNaFCon
TRACK_CONTROL	$	Target on/off flag. Set to ON in prosecute state and if deploy mission is 0.	"ON" / "OFF"
TARGET_RESET	$	Set to TRUE when new TARGET_STATE received	"TRUE" / "FALSE"
SIM_TARGET_XPOS	D	Current UTM x-coordinate of target.	2134.5
SIM_TARGET_YPOS	D	Current UTM y-coordinate of target.	5431.3
NAV_X	D	Ownship UTM x-coordinate	pHuxley
NAV_Y	D	Ownship UTM y-coordinate	pHuxley
NAV_HEADING	D	Ownship true heading in degrees	pHuxley

19.2.4 MOOS variables published by pBearingsSim:

.

MOOS variable	Type	Description	Format
TRACKING	$	Tracking state. "NO_TRACK": No detection yet "AMBIGUOUS": Detection, but ambiguous bearing "TRACKING": Un-ambiguous bearing tracking	
"CLASSIFYING": Classifying sub-state			
BEARING_STAT	$	Comma-separated bearing state variables: vehID,state,bearing,x,y,beamno,sigma,utc	3,2,241,1000,2000,0,0.5,122..
IN_RANGE	$	Reset flag for geo trackers	"FALSE"
CLOSE_RANGE	$	Reset flag for geo trackers	"FALSE"

The principal output of `pBearingsSim` is the MOOS variable BEARING_STAT, which containes the current bearing state variables:

State variable	Description
vehID	Ownship VEHICLE_ID, e.g. 3 for Unicorn
state	Tracking state: 0: no detect, 1: ambiguous bearing; 2: Unambiguous bearing track.
bearing	Absolute, true bearing to target in degrees.
x	Current UTM x-coordinate of ownship
y	Current UTM y-coordinate of ownship
beamno	Beam bumber. Set to 0.
sigma	Bearing uncertainty
utc	UTC time.

19.3 pArraySim

19.3.1 Brief Overview

This MOOS process models the dynamics of a multi-sectored towed array in response to the motion of the tow-point. In addition to computing and publishing in the MOOSDB the hydrophone positions, it also publishes the tension on the tow-point, a parameter which is then subscribed to by the model for the vehicle dynamics. This connectivity results in a high-fidelity modeling of the coupled platform/array system.

19.3.2 Parameters for the pArraySim Configuration Block

The cable configuration and hydrophone separations are defined in the configuration (.moos) file. The following configuration parameters are defined for `pArraySim`.

`cond_check`: Flag for checking the condition number for Jacobian computed for the array dynamics updates. Should be set to 1 to avoid unstable solutions. If condition number exceeds 10^6 the array will be re-initialized to steady-state solution with current AUV heading. This can happen in cases of sharp turns and depth changes. Remedied by increasing AppTick.

`start_speed`: Minimum speed for which array dynamics model will be initiated.

`creat_dir_frame`: Flag identifying whether to write the frame number variables `VSA_DIR` and `VSA_FRAME` to the frame file. If set to zero this step will be done by the acoustic simulation module, which is the normal situation.

`write_out_files`: Flag identifying whether to write the non-acoustic data to the file identified by `VSA_DIR` and `VSA_FRAME`. Under normal circumstances it should be set to 0, since the acoustic simulators perform this step.

`filter_length`: Number of time samples used to achieve an average value of the towing speed and the cable tension, which provides a stable cable solution.

`array_type`: Array type. 1: NUWC VSA array, 2: MIT DURIP array

`number_sectors`: Number of array sections, each of which has homogeneous properties as defined in the `SECTOR_#` parameters.

`sector_#`: Comma-separated list of 7 properties characterizing array sector number #.

- Nsub Number of subdevisions applied in the numerical model for this sector.
- Len length of sector in meters.
- Wgt Weight of cable in N/m.
- Dia Diameter of cable in m.
- Td Tangential drag coefficient.(along cable)
- Nd Normal drag coefficient (perpendicular to cable)

E Elastic modulus for array section in N/m^2

`tow_body`: Flag indicating if towbody exists (1) or not (0).

`towbody_parameters`: Comma-separated list of 3 properties the tow body

Wgt Weight of tow body in N

Dcoef Drag coefficient for towbody

Area Cross-sectional area of towbody.

`array_section_number`: cable section containing acoustic array

`first_phone_offset`: Off set in meters of the first hydrophone in array section.

`phone_spacing_groups`: Number of array element spacing groups. For group number # the number of spacings is specified by the variable `NUM_SPACINGS_GROUP_#`, and the spacing by `SPACING_GROUP_#`

`spacing_group_#`: Array element spacing for group number #.

`num_spacings_group_#`: Number of hydrophone spacing intervals for group number #. Note tha the total number of spacings in the array must equal the total number of array elemnts minus one.

Below is an example configuration block for `pArraySim`, here configured for the MIT DURIP towed array.

Listing 19.3 - An example `pArraySim` *configuration block (MIT DURIP towed array).*

```
 0 //--------- pArraySim Configuration block -----------------
 1 ProcessConfig = pArraySim
 2 {
 3   AppTick =   2
 4   CommsTick = 5
 5
 6   cond_check = 1
 7   start_speed  =   0.5
 8   create_dir_frame = 0
 9   write_out_files  = 0
10   filter_length   = 40
11   array_type = 2
12   number_sectors = 3
13 //          Nsub, Len       Wgt        dia       td     nd,     E
14   sector_1 = 20,   20.0,    .0000,     .0095,   .001,   0.1,  1.6e9
15   sector_2 = 30,   36.0,    .0000,     .035,    .0024,  0.5,  1.6e9
16   sector_3 = 20,   20.0,    .0000,     .0095,   .001,   0.1,  1.6e9
17   tow_body = 0
18 //                     Wgt    Dcoef    Area
19   towbody_parameters = 0.0,   0.0,     0.0
20   array_section_number = 2
21   first_phone_offset   = 3.0
22   phone_spacing_groups = 3
23   spacing_group_1      = 1.5
24   num_spacings_group_1 = 5
25   spacing_group_2      = 0.75
26   num_spacings_group_2 = 22
27   spacing_group_3      = 1.5
28   num_spacings_group_3 = 4
29 }
```

19.3.3 MOOS variables subscribed to by pArraySim:

.

MOOS variable	Type	Description	Published by
VSA_DIR	$	Directory containing files with acoustic and non-acoustic array data	iVSA
VSA_FRAME	D	Frame number for acoustic and non-acoustic data files in VSA_DIR	iVSA
VEHICLE_ID	D	Node number for ownship. Used for selecting dynamic parameters	pNaFCon
TOW_POS_X	D	UTM x-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Y	D	UTM y-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Z	D	UTM z-coordinate of array cable tow point.	uMVS_Bluefin
TOW_VEL_X	D	x-velocity in m/s of array cable tow point.	uMVS_Bluefin
TOW_VEL_Y	D	y-velocity in m/s of array cable tow point.	uMVS_Bluefin
TOW_VEL_Z	D	z-velocity in m/s of array cable tow point	uMVS_Bluefin

19.3.4 MOOS variables published by pArraySim:

.

MOOS variable	Type	Description	Format
CABLE_TENSION	D	Tension of tow cable in N at tow point	
ARRAY_X	$	x-position of hydrophones in m relative to tow-point	17.23,18.33, ...
ARRAY_Y	$	y-position of hydrophones in m relative to tow-point	-6.13,-6.33, ...
ARRAY_Z	$	z-position of hydrophones in m relative to tow-point	0.01,0.02, ...
ARRAY_PITCH	$	pitch in degrees of array elements	0.01,0.00, ...
ARRAY_HEADING	$	true heading in degrees of array elements	301.01,300.99, ...
CABLE_SHAPE_X	$	x-position of cable elements in m relative to tow-point	0.43,0.84, ...
CABLE_SHAPE_Y	$	y-position of cable elements in m relative to tow-point	-0.10,-0.21, ...
CABLE_SHAPE_Z	$	z-position of cable elements in m relative to tow-point	0.00,0.00, ...
'auv'_AEL_HEADING	D	average true heading in degrees of acoustic array section. 'auv' is defined by VEHICLE_ID, e.g UNICORN.	
'auv'_AEL_PITCH	D	average pitch in radians of acoustic array section	

19.4 pMultiTargetSim

19.4.1 Brief Overview

This MOOS module simulates an arbitrary number of acoustic sources moving along straight-line paths. The sources are activated one at a time using the MATLAB GUI `PassiveTgtSim.m`. `pMultiTargetSim` will then keep a record of all sources within a critical distance from the vehicle, specified in the MOOS-block, and generate the MOOS variables needed by the acoustic timeseries simulators `pMultiAcousticSim` and `SealabMultiSim.m`. Note that the initiation of targets may be performed using the `PassiveTgtSim.m` GUI, or through an issue Prosecute command, dependent on the value of the MOOS variable TARGET_CONTROL. In general `PassiveTgtSim.m` will be used, but the control by the Prosecute command is useful for distributed virtual experiments, where missions may be commanded from a control center in another part of the world. Work is in progress on centralizing `PassiveTgtSim.m`, using a dedicated, broadcast modem message to activate sources simultaneously on all vehicles involved in such virtual experiments, which will make the Prosecute 'hack' to become obsolete.

19.4.2 Parameters for the pMultiTargetSim Configuration Block

The following configuration parameters are defined for `pMultiTargetSim`.

`rangeCrit`: Critical range beyond which target will be ignored.

Below is an example configuration block for the `pMultiTargetSim`.

Listing 19.4 - An example `pMultiTargetSim` *configuration block.*

```
 1  LatOrigin  = 42.3584
 2  LongOrigin = -71.08745
 3
 4 // ------------ pMultiTargetSim Configuration block ----------
 5 ProcessConfig = pMultiTargetSim
 6 {
 7   AppTick = 4
 8   CommsTick = 4
 9
10   rangeCrit     = 5000 // maximum source range in meters
11 }
```

The `LatOrigin` and `LonOrigin` parameters on lines 1-2 are not specific to `pMultiTargetSim`, but are specified at the global level in a MOOS file. They are needed to allow the conversion between local x-y coordinates and latitude-longitude coordinates. They are mandatory parameters and `pMultiTargetSim` will refuse to launch if they are lacking - but they are mandatory for other common MOOS applications as well.

19.4.3 MOOS variables subscribed to by pMultiTargetSim:

.

MOOS variable	Type	Description	Published by
TARGET_CONTROL	$	"ON": Sources controlled by PassiveTgtSim.m, preample = T_ "OFF": Sources controlled by Prosecute message, preample = TARGET_	PassiveTgtSim.m
T_STATE TARGET_STATE	$	Comma-separated list of source parameters: x,y,depth,heading,speed,freq,bw,spl,delay, tn1_frew,tn1_dbl,...,tn5_freq,tn5_dbl,time,number	PassiveTgtSim.m pGeneralCodec pNaFCon

19.4.4 MOOS variables published by pMultiTargetSim:

.

MOOS variable	Type	Description	Format
TARGET_SIM_LIST	$	Comma-separated list of source numbers within critical distance	1,2,3,6
TGT_#_SIM_STATE	$	Comma-separated list of source parameters for source number # #,on/off,utc,x,y,speed,heading,depth, spl,freq,bw,delay,tn1_freq,tn2_dbl,..	2,1,122...,3000,2100,..,0,850,120

19.5 pMultiAcousticSim

19.5.1 Brief Overview

This medium-fidelity acoustic timeseries simulator uses a plane wave model of the acoustic propagation from the source to the array, but incorporates cylindrical spreading and bottom loss. Highly efficient when high-fidelity acoustics is not needed.

19.5.2 Parameters for the pMultiAcousticSim Configuration Block

The following configuration parameters are defined for `pMultiAcousticSim`.

`water_depth`: Local water depth used for defining transition from spherical to cylindrical sprading loss.

`water_c`: Sound speed in water (m/s).

`bottom_loss`: Estimate of average transmission loss due to bottom interaction. Specified in dB/km.

`noise_level`: Noise level in dB.

`array_type`: Array type, defining file format. Options are DURIP, VSA, SINGLE, or DOUBLE.

`num_array_elements`: Number of hydrophone elements in array.

`sampling_frequency`: Sampling frequency for synthesized time series in Hz.

`number_samples`: Number of samples per file.

`hydrophone_gain`: Conversion factor to dB rel. 1 Pa

`base_dir_name`: Path for directory to contain acoustic and non-acoustic data files.

`acoustic_filename_prefix`: Prefix for acoustic data file names

`acoustic_filename_suffix`: Extension for acoustic data file names

`nonacoustic_filename_prefix`: Prefix for non-acoustic (sensor geometries) data file names

`nonacoustic_filename_suffix`: Extension for non-acoustic (sensor geometries) data file names

Below is an example configuration block for the `pMultiAcousticSim`.

Listing 19.5 - An example `pMultiAcousticSim` *configuration block.*

```
1 // ------------ pMultiAcousticSim Configuration block ----------
2 ProcessConfig = pMultiAcousticSim
3 {
4   AppTick =   12
5   CommsTick = 5
6 // Source Parameters
7 // Environmental Parameters
```

```
 8    water_depth = 100            // used for spherical spreading (m)
 9    water_c = 1500               // Speed of sound in water (m/s)
10    bottom_loss = 5.0            // Bottom loss estimate dB/km
11    noise_level = 60             // dB re 1 uPa
12 // Array and sampling parameters
13    array_type = DURIP           // DURIP or VSA
14    num_array_elements = 32      // number of elements in the array
15    sampling_frequency = 4000    // Hz
16    num_samples = 8000           // number of samples per file
17    hydrophone_gain = 1000000000 // conversion from Pascals to float
18 // Output file location and name
19 // absolute path including initial and final forward slash.
20    base_dir_name = /home/henrik/DURIP/
21 // The output filename will be:
22 // [filename_prefix][frame_number][filename_suffix]
23    acoustic_filename_prefix = ACO
24    acoustic_filename_suffix = .DAT
25    nonacoustic_filename_prefix = NAS
26    nonacoustic_filename_suffix = .DAT
27 }
```

19.5.3 MOOS variables subscribed to by pMultiAcousticSim:

.

MOOS variable	Type	Description	Published by
TOW_POS_X	D	UTM x-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Y	D	UTM y-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Z	D	UTM z-coordinate of array cable tow point.	uMVS_Bluefin
ARRAY_X	$	x-position of hydrophones in m relative to tow-point	pArraySim
ARRAY_Y	$	y-position of hydrophones in m relative to tow-point	pArraySim
ARRAY_Z	$	z-position of hydrophones in m relative to tow-point	pArraySim
ARRAY_PITCH	$	pitch in radians of array elements	pArraySim
ARRAY_HEADING	$	true heading in degrees of array elements	pArraySim
TARGET_SIM_LIST	$	Comma-separated list of source numbers within critical distance	pMultiTargetSim
TGT_#_SIM_STATE	$	Comma-separated list of source parameters for source number # #,on/off,utc,x,y,speed,heading,depth,spl,freq,bw	pMultiTargetSim

19.5.4 MOOS variables published by pMultiAcousticSim:

.

MOOS variable	Type	Description	Format
TARGET_XPOS	D	UTM x position for current target for plotting by `small_uVis.m`	
TARGET_YPOS	D	UTM x position for current target for plotting by `small_uVis.m`	

19.5.5 pMultiAcousticSim Details

pMultiAcousticSim models acoustic sources that have a flat spectrum in addition to sinusoidal tones. pMultiAcousticSim takes into account spreading loss (spherical to the waveguide depth, cylindrical thereafter), as well as bottom loss. The waves impinging upon the hydrophones are cylindrical waves originating from the sources' respective locations. Because the modeled waves are cylindrical, the depth of each hydrophone is irrelevant.

pMultiAcousticSim has the following shortcomings:

- Some approximations are made to simplify the math. These approximations will make the transmission loss inaccurate for distances less than one waveguide depth.
- The synthesized time series is discontinuous across files.
- For files generated for the VSA, the time stamp in the acoustic file is zero.
- pMultiAcousticSim will write a file if it has been longer than `number_samples` divided by `sampling_frequency` since the last file. It will not try to make up for "lost time" because this is not desirable in all cases (i.e. when the hydrophone or source position(s) stop being published to the MOOSDB, and then start again). If the processing chain relies on a file being published at very precise intervals, pMultiAcousticSim will have to be modified.

19.6 pGPSSim

19.6.1 Brief Overview

This module is used for simulating GPS fixes during surfacing. Used e.g. in connection with BHV_PeriodicSurface behavior.

19.6.2 Parameters for the pGPSSim Configuration Block

The following configuration parameters are defined for `pMultiTargetSim`.

`gps_depth`: Depth above which GPS is assumed to be active.

`gps_interval`: Time between GPS fixes in seconds

`min_gps_time`: Minimum time spent on surface for GPS fixes in seconds

Below is an example configuration block for the `pMultiTargetSim`.

Listing 19.6 - An example `pMultiTargetSim` *configuration block.*

```
1 ProcessConfig = pGPSSim
2 {
3    AppTick = 1
4    CommsTick = 1
5     gps_depth = 0.5        // Depth at which GPS becomes active
6     gps_interval = 2       // Interval between gps fixes in seconds
7     min_gps_time = 10      // Minimum surface time for achieving GPS
8 }
```

19.6.3 MOOS variables subscribed to by pGPSSim:

.

MOOS variable	Type	Description	Published by
NAV_X	D	UTM x-coordinate of vehicle.	pHuxley
NAV_Y	D	UTM y-coordinate of vehicle.	pHuxley
NAV_DEPTH	D	Vehicle depth in meter.	pHuxley

19.6.4 MOOS variables published by pGPSSimpin:

.

MOOS variable	Type	Description	Format
SECS_TO_DIVE	D	Time in seconds remaining before diving	
GPS_UPDATE_RECEIVED	$	GPS 'measurement'	Timestamp=122.., Latitude=42.12345, Longitude=10.98765

19.7 uCtdSim2

19.7.1 Brief Overview

This module is used for simulating CTD data using HOPS-generated virtual ocean or real ocean database.

19.7.2 Parameters for the uCtdSim2 Configuration Block

The following configuration parameters are defined for `uCtdSim2`.

`Mods_Bin_File`: Name of binary file containing CTD database.

Below is an example configuration block for the `uCtdSim2`.

Listing 19.7 - An example `uCtdSim2` *configuration block.*

```
1 ProcessConfig = uCtdSim2
2 {
3    AppTick   = 5
4    CommsTick = 5
5
6    Mods_Bin_File = ../../../data/may07_ctdsim2.bin
7 }
```

19.7.3 MOOS variables subscribed to by uCtdSim2:

.

MOOS variable	Type	Description	Published by
NAV_LONG	D	Longitude of vehicle in degrees.minutes.	pHuxley
NAV_LAT	D	Latitude of vehicle in degrees.minutes.	pHuxley
NAV_DEPTH	D	Depth of vehicle in meter.	pHuxley

19.7.4 MOOS variables published by uCtdSim2:

.

MOOS variable	Type	Description	Format
CTD1	$	Message concatenating published and subscribed fields	
CTD_SOUND_VELOCITY	D	Sound velocity in m/s given logitude, latitude and depth of vehicle	
CTD_TEMPERATURE	D	Temperature in degree celsius given logitude, latitude and depth of vehicle	
CTD_SALINITY	D	Salinity in PSU given logitude, latitude and depth of vehicle	

19.8 uBathy

19.8.1 Brief Overview

This module is used for simulating bathymetry data using bathymetry table.

19.8.2 Parameters for the uBathy Configuration Block

The following configuration parameters are defined for `uBathy`.

`Bathy_Bin_File`: Name of binary file containing bathymetry table.

Below is an example configuration block for the `uBathy`.

Listing 19.8 - An example `uBathy` *configuration block.*

```
1 ProcessConfig = uBathy
2 {
3    AppTick   = 5
4    CommsTick = 5
5
6    Bathy_Bin_File = /home/raylum/project-plusnet/data/monterey.xyz.bin
7 }
```

19.8.3 MOOS variables subscribed to by uBathy:

.

MOOS variable	Type	Description	Published by
NAV_X	D	x position of vehicle in meter.	pHuxley
NAV_Y	D	y position of vehicle in meter.	pHuxley
NAV_Z	D	z position of vehicle in meter.	pHuxley

19.8.4 MOOS variables published by uBathy:

.

MOOS variable	Type	Description	Format
BATHY	$	Message concatenating published and subscribed fields	
BATHY_Z	D	Depth of water column given x and y positions of vehicle	

19.9 Arraysim.m

19.9.1 Brief Overview

This is the MATLAB version of the array dynamics simulator. The functionality of the two are identical, and they use identical MOOSDB interface definitions.

19.9.2 Configuration Files

The array information is defined in the configuration file config.txt: For description of the significance of each parameter, see the description for the equivalents for `pArraySim`

Listing 19.9 - An example `pArraySim` *configuration block (MIT DURIP towed array).*

```
1  moos_file        Array.moos
2  moos_name        ArraySim
3  cond_check 1
4  start_speed      0.5
5  create_dir_frame 0
6  write_out_files  0
7  filter_length    40
```

and a cable definition file, which for the MIT DURIP array is cable_durip.dat. Again the significance of the parameters is identical to the `pArraySim` equivalents.

```
0  Parameters for DURIP array  with 3 sections (tow cable+acoustic section+drogue)
1   3
2  20  20.0      .0000      .0095   .001      0.2      1.6e9
3  30  36.0      .0000      .035    .0024     0.3      1.6e9
4  20   20.0      .0000      .0095   .001      0.2      1.6e9
5  0.0 0.0 0.0
```

The remaining parameters, such as those defining hydrophone spacing etc. are specified internally, based on the MOOS variable VEHICLE_ID, which for DURIP should be 3 (Unicorn). This in contrast to the C++ version `pArraySim` which is entirely generic, with all variables specified in the configuration file.

19.9.3 MOOS variables subscribed to:

.

MOOS variable	Type	Description	Published by
VSA_DIR	$	Directory containing files with acoustic and non-acoustic array data	iVSA
VSA_FRAME	D	Frame number for acoustic and non-acoustic data files in VSA_DIR	iVSA
VEHICLE_ID	D	Node number for ownship. Used for selecting dynamic parameters	pNaFCon
TOW_POS_X	D	UTM x-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Y	D	UTM y-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Z	D	UTM z-coordinate of array cable tow point.	uMVS_Bluefin
TOW_VEL_X	D	x-velocity in m/s of array cable tow point.	uMVS_Bluefin
TOW_VEL_Y	D	y-velocity in m/s of array cable tow point.	uMVS_Bluefin
TOW_VEL_Z	D	z-velocity in m/s of array cable tow point	uMVS_Bluefin

19.9.4 MOOS variables published:

.

MOOS variable	Type	Description	Format
CABLE_TENSION	D	Tension of tow cable in N at tow point	
ARRAY_X	$	x-position of hydrophones in m relative to tow-point	17.23,18.33, ...
ARRAY_Y	$	y-position of hydrophones in m relative to tow-point	-6.13,-6.33, ...
ARRAY_Z	$	z-position of hydrophones in m relative to tow-point	0.01,0.02, ...
ARRAY_PITCH	$	pitch in radians of array elements	0.01,0.00, ...
ARRAY_HEADING	$	true heading in degrees of array elements	301.01,300.99, ...
CABLE_SHAPE_X	$	x-position of cable elements in m relative to tow-point	0.43,0.84, ...
CABLE_SHAPE_Y	$	y-position of cable elements in m relative to tow-point	-0.10,-0.21, ...
CABLE_SHAPE_Z	$	z-position of cable elements in m relative to tow-point	0.00,0.00, ...
'auv'_AEL_HEADING	D	average true heading in degrees of acoustic array section. The preample 'auv' is defined by VEHICLE_ID, e.g UNICORN.	
'auv'_AEL_PITCH	D	average pitch in radians of acoustic array section	

19.10 SealabMultiSim.m

19.10.1 Brief Overview

This is the MATLAB-based interface between the MOODDB and the Sealab sonor simulation environment. Sealab is a commercial product (VASA Associates Inc.), which uses state of the art lecacy propagation models and a native 3-D coupled mode model to produce high-fideklity element-level timeseries simulation for arbitrary 3D arrays and source configurations in a complex ocean model. It incorporates ocean wave-guide acoustic effects including mode coupling, doppler spread etc. It is sufficiently efficient for producing high-fidelity simuklated data in real time on a single platform.

19.10.2 Configuration MOOS-block

```
//.moos for SealabMultiTarget
ProcessConfig = SealabMultiTarget
{
    AppTick     = 10
    CommsTick   = 10
    Port        = COM6
    BaudRate    = 4800
    Verbose     = true
    Streaming   = false
    MOOSComms   = true          // Publish output to MOOSDB
    SerialComms = false         // Send output over serial line.
    SERIAL_TIMEOUT = 10.0
    SUBSCRIBE = VEHICLE_ID @ 0
    SUBSCRIBE = DB_TIME @ 0
    SUBSCRIBE = NAV_SPEED @ 0
    SUBSCRIBE = TOW_POS_X @ 0
    SUBSCRIBE = TOW_POS_Y @ 0
    SUBSCRIBE = TOW_POS_Z @ 0
    SUBSCRIBE = TOW_VEL_X @ 0
    SUBSCRIBE = TOW_VEL_Y @ 0
    SUBSCRIBE = TOW_VEL_Z @ 0
    SUBSCRIBE = ARRAY_X @ 0
    SUBSCRIBE = ARRAY_Y @ 0
    SUBSCRIBE = ARRAY_Z @ 0
    SUBSCRIBE = ARRAY_PITCH @ 0
    SUBSCRIBE = ARRAY_HEADING @ 0
    SUBSCRIBE = TARGET_SIM_LIST @ 0
    SUBSCRIBE = VSA_DIR @ 0
    SUBSCRIBE = VSA_FRAME @ 0
}
```

19.10.3 MOOS variables subscribed to:

.

MOOS variable	Type	Description	Published by
VEHICLE_ID	D	Node number for ownship. Used for selecting dynamic parameters	pNaFCon
TOW_POS_X	D	UTM x-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Y	D	UTM y-coordinate of array cable tow point.	uMVS_Bluefin
TOW_POS_Z	D	UTM z-coordinate of array cable tow point.	uMVS_Bluefin
TOW_VEL_X	D	x-velocity of array cable tow point. For doppler.	uMVS_Bluefin
TOW_VEL_Y	D	y-velocity of array cable tow point. For doppler.	uMVS_Bluefin
TOW_VEL_Z	D	z-velocity of array cable tow point. For doppler.	uMVS_Bluefin
ARRAY_X	$	x-position of hydrophones in m relative to tow-point	pArraySim
ARRAY_Y	$	y-position of hydrophones in m relative to tow-point	pArraySim
ARRAY_Z	$	z-position of hydrophones in m relative to tow-point	pArraySim
ARRAY_PITCH	$	pitch in radians of array elements	pArraySim
ARRAY_HEADING	$	true heading in degrees of array elements	pArraySim
VSA_DIR	$	Directory containing files with acoustic and non-acoustic array data	iVSA
VSA_FRAME	D	Frame number for acoustic and non-acoustic data files in VSA_DIR	iVSA
TARGET_SIM_LIST	$	Comma-separated list of source numbers within critical distance	pMultiTargetSim
TGT_#_SIM_STATE	$	Comma-separated list of source parameters for source number # #,on/off, utc, x, y, speed, heading, depth, spl, freq, bw	pMultiTargetSim

19.10.4 MOOS variables published:

.

MOOS variable	Type	Description	Format
TARGET_XPOS	D	UTM x position for current target for plotting by `small_uVis.m`	
TARGET_YPOS	D	UTM x position for current target for plotting by `small_uVis.m`	

19.11 PassiveTgtSim.m

19.11.1 Brief Overview

This MATLAB module uses the GUI shown in Fig. 27 for activating acoustic sources for the simulation environment. Thus, it is used for interactively activating targets to be simulated by the target dynamics simulator `pMultiTargetSim`.

19.11.2 Usage

The dynamic target simulator as well as the various-level fidelity acoustic simulators are always excuted on each sensor node. However, the activation of the acoustic target and interferrer sourecs may occur either locally in a simulated node `MOOS` community, or centrally, e.g in the topside moos community, The latter is important for syncronizing sources simulated on multiple nodes for testing collaborative tracking processing and behaviors. The centralized target control is made possible by defining the interface between the GUI and the target dynamics simulators compatibleusing a format which is compatible ith the generic messages Codec `pGeneralCodec`, such that the targets may be activated on all platforms in a synchronized manner via a madem transmission echoing the initial target status to all nodes. In addition to be used in entirely virtual experiments, this architecture allows for activating acoustic target simulators on actul, submerged nodes, e.g. in cases where equipment failure has made the sensing arrays unusable, thus allowing testing of collaborativ tracking behaviors even in such cases.

The echoing of the target initialization to all network nodes is activated by including the `XML` file `nafcon_targetsim.xml` in the configuration for `pGeneralCodec`, as described in Appendix A.

Note that the `MOOS` variable `TGT_STATE_OUT` is assumed to contain the target initialization message on the local node, while the `MOOS` variable receiving the message on the receiving nodes is `TGT_STATE_IN`. Thus, it will be necessary to echo these variables from and into the standard `MOOS` variable `T_STATE` published by `PassiveTgtSim`, using `pEchoVar`. In addition, the `MOOS` variable `TARGET_CONTROL` is echoed to all other platforms to activate the source control through `PassiveTgtSim`. Thus, it is important to at least once broadcast a target initialization before issuing a *Prosecute Command*, which woulk otherize activate the nodal source simulators.

19.11.3 Configuration MOOS-block for `PassiveTgtSim`

Listing 19.10 - An example `PassiveTgtSim` *configuration block.*

```
 1 ProcessConfig = PassiveTgtSim
 2 {
 3       AppTick     = 10
 4       CommsTick   = 10
 5       Port     = COM6
 6       BaudRate    = 4800
 7       Verbose     = false
 8       Streaming   = false
 9       MOOSComms = true           // true: use MOOSDB.
10       SerialComms = false        // true: use serial communication
11       SERIAL_TIMEOUT = 10.0
12       SUBSCRIBE = DB_TIME @ 0
13 }
```

19.11.4 MOOS variables subscribed to:

.

MOOS variable	Type	Description	Published by
DB_TIME	D	MOOS time	MOOSDB

19.11.5 MOOS variables published:

.

MOOS variable	Type	Description	Format
TARGET_CONTROL	$	"ON": Sources controlled by PassiveTgtSim.m, preample = T_ "OFF": Sources controlled by Prosecute message, preample = TARGET_ . The control flag will be echoed to all platforms when used together with `pGeneralCodec`	
T_SOURCE	$	"ON": Source on "OFF": Source off	
T_STATE	$	Comma-separated list of source parameters with identifiers: position,heading,speed,freq,bw,spl,delay,tones,utc,number	tgt_x=2000, tgt_y=3000, tgt_depth=15, tgt_hdg=90, tgt_speed=4, tgt_freq=850, tgt_bw=100, tgt_spl=135, tgt_delay=0, tn1_freq=800, tn1_dbl=120, ... tn5_freq=0, tn5_dbl=0, tgt_utc=122345.., tgt_num=2

Notes: The parameters listed in the MOOS variable `T_STATE` are

`tgt_x`: UTM x-coordinate of acoustic source starting point

`tgt_y`: UTM y-coordinate of acoustic source starting point

`tgt_depth`: Depth of acoustic source

`tgt_hdg`: Heading of acoustic source in degrees

`tgt_speed`: Speed of acoustic source in m/s.

`tgt_freq`: Center frequency of broadband acoustic source

`tgt_bw`: Bandwidth in Hz of broadband acoustic source

`tgt_spl`: Spectral level of broadband source

`tgt_delay`: Delay in seconds of source transmission

`tn#_freq`: Frequency in Hz of tone number #. GUI allows for up to 5 tones which will be superimposed to broadband source signature.

`tn#_dbl`: Source level in dB of tone number #. GUI allows for up to 5 tones which will be superimposed to broadband source signature.

`tgt_utc`: UTC time of source starting location

`tgt_num`: ID number assigned to acoustic source. Will be automatically incremented for each new source activation.

References

[1] D.P. Eickstedt, M.R. Benjamin, J.P. Ianniello, H. Schmidt, and J.J. Leonard. Adaptive Tracking of Underwater Targets with Autonomous Sensor Networks. *JUA (USN)*, 56:465–495, 2006.

[2] Andrew Poulsen, Donald Eickstedt, and Henrik Schmidt. Stabilized Target Bearing Estimation and Tracking for AUVs with Towed Hydrophone Arrays. *JUA (USN)*, ?:????, 2008.

A Appendix - pGeneralCodec Configuration Files

The message set developed and applied in the ONR PLUSnet program used a dedicated set of CCL messages for transmitting *Deploy* and *Prosecute Commnds*, and *Status, Contact* and *Track Reports*. The CCL message decoding was performed using the MOOS process `pFramer`. In the fall of 2008, Toby Schneder at MIT developed a new, totally generic message coder-decoder, `pGeneralCodec`, replacing the highly restrictive `pFramer`, using a special Dynamic CCL (D-CCL) message with the official CCL type 32. The message coding/decoding scheme is specified at run-time in a configuration file using the xml macro format, thus allowing for a much more flexible message set, easily modified and expanded without re-coding. For backward compatibility, the PLUSNet message format has been retained using the new Codec, specfied in two `xml` configuration files, referenced in the `MOOS` configuration block for `pGeneralCodec`, shown in Listing A.1.

Listing A.1 - MIT-LAMS prototype `pGeneralCodec` *configuration block .*

```
 1 ProcessConfig = pGeneralCodec
 2 {
 3 AppTick    = 4
 4 CommsTick  = 4
 5
 6   verbosity = verbose
 7
 8 // located in moos-ivp-local/data/acomms/
 9   message_file = ../../../data/acomms/nafcon_command.xml
10   message_file = ../../../data/acomms/nafcon_report.xml
11 // henriks attempt at creating message for TARGET_STATE for simulator
12   message_file = ../../../data/acomms/nafcon_targetsim.xml
13 }
```

The message files contain the coding/decoding scheme in xml, as described in Section 13.8. For convenience the configurations for commands and reports are specified in separate files, `nafcon_commands.xml` and `nafcon_reports.xml`. The structure of the two files is shown in Listings A.2 and A.3.

Listing A.2 pGeneralCodex Configuration file `nafcon_commands.xml` *for Sensor Commands*

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<!-- contains nafcon (plus) commands: deploy, prosecute -->
<message_set>
   ...
   DEPLOY Command XML block
   ...
   PROSECUTE Command XML block
   ...
</message_set>
```

Listing A.3 pGeneralCodex Configuration file `nafcon_reports.xml` *for Sensor Reports*

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<!-- contains nafcon (plus) reports: status, contact and track -->
<message_set>
   ...
   STATUS Report XML block
   ...
   CONTACT Report XML block
   ...
   TRACK Report XML block
   ...
</message_set>
```

The last message file in the configuration block, `nafcon_targetsim` is a message specifically designed for simulating acoustic sources synchronously on all nodes. This message is used in conjunction with the target GUI `PassiveTgtSim` in the topside MOOS community for transmitting the target parameters to all nodes via the real or virtual modem network. The current XML configuration file `nafcon_targetsim` is shown in Listing A.9.

Listing A.4 pGeneralCodex XML Configuration Block for Sensor Deploy Command

```
...
<message>
  <name>SENSOR_DEPLOY</name>
  <destination_moos_var key="DestinationPlatformId">
    PLUSNET_MESSAGES
  </destination_moos_var>
  <outgoing_hex_moos_var>
    OUT_DEPLOY_HEX_30B
  </outgoing_hex_moos_var>
  <incoming_hex_moos_var>
    IN_DEPLOY_HEX_30B
  </incoming_hex_moos_var>
  <trigger>publish</trigger> <!-- publish to moos var -->
  <trigger_moos_var mandatory_content=
    "MessageType=SENSOR_DEPLOY">
    PLUSNET_MESSAGES</trigger_moos_var>
  <size>30</size>
  <layout>
    <static>
      <name>MessageType</name>
      <value>SENSOR_DEPLOY</value>
    </static>
    <static>
      <name>SensorCommandType</name>
      <value>0</value>
    </static>
    <int>
      <name>SourcePlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>DestinationPlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>Timestamp</name>
      <max>2000000000</max>
      <min>1000000000</min>
    </int>
    <float>
      <name>DeployLatitude</name>
      <precision>5</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>DeployLongitude</name>
      <precision>5</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>DeployDepth</name>
      <max>255</max>
      <min>0</min>
    </int>
    <int>
      <name>DeployDuration</name>
      <max>63</max>
      <min>1</min>
    </int>
    <float>
      <name>AbortLatitude</name>
      <precision>4</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>AbortLongitude</name>
      <precision>4</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>AbortDepth</name>
      <max>511</max>
      <min>0</min>
    </int>
    <int>
      <name>MissionType</name>
      <max>7</max>
      <min>0</min>
    </int>
    <int>
      <name>OperatingRadius</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <float algorithm="angle_0_360">
      <name>DCLFOVStartHeading</name>
      <max>360</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float algorithm="angle_0_360">
      <name>DCLFOVEndHeading</name>
      <max>360</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>DCLSearchRate</name>
      <max>255</max>
      <min>0</min>
      <precision>1</precision>
    </float>
  </layout>
  <!-- decoding -->
  <publish>
    <moos_var>NAFCON_MESSAGES</moos_var>
    <all />
  </publish>
</message>
...
```

Listing A.5 pGeneralCodec XML Configuration Block for Prosecute Command

```
...
<message>
  <name>SENSOR_PROSECUTE</name>
  <outgoing_hex_moos_var>
    OUT_PROSECUTE_HEX_30B
  </outgoing_hex_moos_var>
  <incoming_hex_moos_var>
    IN_PROSECUTE_HEX_30B
  </incoming_hex_moos_var>
  <destination_moos_var key="DestinationPlatformId">
    PLUSNET_MESSAGES
  </destination_moos_var>
  <trigger>publish</trigger> <!-- publish to moos var -->
  <trigger_moos_var
    mandatory_content="MessageType=SENSOR_PROSECUTE">
    PLUSNET_MESSAGES
  </trigger_moos_var>
  <size>30</size>
  <layout>
    <static>
      <name>MessageType</name>
      <value>SENSOR_PROSECUTE</value>
    </static>
    <int>
      <name>SensorCommandType</name>
      <min>1</min>
      <max>4</max>
    </int>
    <int>
      <name>SourcePlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>DestinationPlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>TargetTimestamp</name>
      <max>2000000000</max>
      <min>1000000000</min>
    </int>
    <int>
      <name>PlatformID</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackNumber</name>
      <max>255</max>
      <min>0</min>
    </int>
    <float>
      <name>TargetLatitude</name>
      <precision>5</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>TargetLongitude</name>
      <precision>5</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>TargetDepth</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <float algorithm="angle_0_360">
      <name>TargetHeading</name>
      <max>360</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>TargetSpeed</name>
      <max>12.8</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <int>
      <name>TargetSpectralLevel1</name>
      <max>127</max>
      <min>0</min>
    </int>
    <int>
      <name>TargetFrequency1</name>
      <max>4095</max>
      <min>0</min>
    </int>
    <int>
      <name>TargetBandwidth1</name>
      <max>20</max>
      <min>0</min>
    </int>
    <int>
      <name>TargetSpectralLevel2</name>
      <max>127</max>
      <min>0</min>
    </int>
    <int>
      <name>TargetFrequency2</name>
      <max>4095</max>
      <min>0</min>
    </int>
    <int>
      <name>TargetBandwidth2</name>
      <max>20</max>
      <min>0</min>
    </int>
    <int>
      <name>ProsecuteDuration</name>
      <max>63</max>
      <min>1</min>
    </int>
    <float>
      <name>AbortLatitude</name>
      <precision>4</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
```

```
      <name>AbortLongitude</name>
      <precision>4</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>AbortDepth</name>
      <max>127</max>
      <min>0</min>
    </int>
  </layout>
  <!-- decoding -->
  <publish>
    <moos_var>NAFCON_MESSAGES</moos_var>
    <all />
  </publish>
</message>
...
```

Listing A.6 pGeneralCodex XML Configuration Block for Status Reports

```
...
<message>
  <name>SENSOR_STATUS</name>
  <outgoing_hex_moos_var>
    OUT_STATUS_HEX_30B
  </outgoing_hex_moos_var>
  <incoming_hex_moos_var>
    IN_STATUS_HEX_30B
  </incoming_hex_moos_var>
  <trigger>publish</trigger> <!-- publish to moos var -->
  <trigger_moos_var
        mandatory_content="MessageType=SENSOR_STATUS">
        PLUSNET_MESSAGES</trigger_moos_var>
  <size>30</size>
  <layout>
    <static>
      <name>MessageType</name>
      <value>SENSOR_STATUS</value>
    </static>
    <static>
      <name>SensorReportType</name>
      <value>0</value>
    </static>
    <int>
      <name>SourcePlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>DestinationPlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>Timestamp</name>
      <max>2000000000</max>
      <min>1000000000</min>
    </int>
    <float>
      <name>NodeLatitude</name>
      <precision>5</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>NodeLongitude</name>
      <precision>5</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>NodeDepth</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <int>
      <name>NodeCEP</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <float algorithm="angle_0_360">
      <name>NodeHeading</name>
      <max>360</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>NodeSpeed</name>
      <max>12.8</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <int>
      <name>MissionState</name>
      <max>7</max>
      <min>0</min>
    </int>
    <int>
      <name>MissionType</name>
      <max>7</max>
      <min>0</min>
    </int>
    <int>
      <name>LastGPSTimestamp</name>
      <max>2000000000</max>
      <min>1000000000</min>
    </int>
    <int>
      <name>PowerLife</name>
      <max>1023</max>
      <min>1</min>
    </int>
    <int>
      <name>SensorHealth</name>
      <max>15</max>
      <min>0</min>
    </int>
    <int>
      <name>RecorderState</name>
      <max>1</max>
      <min>0</min>
    </int>
    <int>
      <name>RecorderLife</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>NodeSpecificInfo0</name>
      <max>255</max>
      <min>0</min>
    </int>
    <int>
      <name>NodeSpecificInfo1</name>
      <max>255</max>
      <min>0</min>
    </int>
    <int>
      <name>NodeSpecificInfo2</name>
      <max>255</max>
      <min>0</min>
    </int>
    <int>
      <name>NodeSpecificInfo3</name>
```

```
    <max>255</max>
    <min>0</min>
  </int>
  <int>
    <name>NodeSpecificInfo4</name>
    <max>255</max>
    <min>0</min>
  </int>
  <int>
    <name>NodeSpecificInfo5</name>
    <max>255</max>
    <min>0</min>
  </int>
</layout>
<!-- decoding -->
<publish>
  <moos_var>NAFCON_MESSAGES</moos_var>
  <all />
</publish>
<!-- start pTransponderAIS replacement -->
<publish>
  <moos_var>AIS_REPORT</moos_var>
  <format>
    NAME=%1%,TYPE=%2%,UTC_TIME=%3$.0lf,
    X=%4%,Y=%5%,LAT=%6$lf,LON=%7$lf,
    SPD=%8%,HDG=%9%,DEPTH=%10%
  </format>
  <message_var algorithm="modem_id2name,to_lower">
    SourcePlatformId</message_var>
  <message_var algorithm="modem_id2type,to_lower">
    SourcePlatformId</message_var>
  <message_var>Timestamp</message_var>
  <message_var algorithm="lon2utm_x:NodeLatitude">
    NodeLongitude</message_var>
  <message_var algorithm="lat2utm_y:NodeLongitude">
    NodeLatitude</message_var>
  <message_var>NodeLatitude</message_var>
  <message_var>NodeLongitude</message_var>
  <message_var>NodeSpeed</message_var>
  <message_var>NodeHeading</message_var>
  <message_var>NodeDepth</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_UTC
  </moos_var>
  <format>%2$lf</format>
  <message_var algorithm="modem_id2name,to_upper">
     SourcePlatformId</message_var>
  <message_var>Timestamp</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_X
  </moos_var>
  <format>%2%</format>
  <message_var algorithm="modem_id2name,to_upper">
     SourcePlatformId</message_var>
  <message_var algorithm="lon2utm_x:NodeLatitude">
     NodeLongitude</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_Y
  </moos_var>
  <format>%2%</format>
  <message_var algorithm="modem_id2name,to_upper">
     SourcePlatformId</message_var>
  <message_var algorithm="lat2utm_y:NodeLongitude">
     NodeLatitude</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_LAT
  </moos_var>
  <format>%2%</format>
  <message_var algorithm="modem_id2name,to_upper">
    SourcePlatformId</message_var>
  <message_var>NodeLatitude</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_LONG
  </moos_var>
  <format>%2$lf</format>
  <message_var algorithm="modem_id2name,to_upper">
     SourcePlatformId</message_var>
  <message_var>NodeLongitude</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_SPEED
  </moos_var>
  <format>%2$lf</format>
  <message_var algorithm="modem_id2name,to_upper">
     SourcePlatformId</message_var>
  <message_var>NodeSpeed</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_HEADING
  </moos_var>
  <format>%2%</format>
  <message_var algorithm="modem_id2name,to_upper">
     SourcePlatformId</message_var>
  <message_var>NodeHeading</message_var>
</publish>
<publish>
  <moos_var type="double">
    %1%_NAV_DEPTH
  </moos_var>
  <format>%2%</format>
  <message_var algorithm="modem_id2name,to_upper">
    SourcePlatformId</message_var>
  <message_var>NodeDepth</message_var>
</publish>
<!-- end pTransponderAIS replacement -->
</message>
...
```

Listing A.7 pGeneralCodex XML Configuration Block for Contact Report

```
...
<message>
  <name>SENSOR_CONTACT</name>
  <outgoing_hex_moos_var>
    OUT_CONTACT_HEX_30B
  </outgoing_hex_moos_var>
  <incoming_hex_moos_var>
    IN_CONTACT_HEX_30B
  </incoming_hex_moos_var>
  <trigger>publish</trigger>
  <trigger_moos_var
    mandatory_content="MessageType=SENSOR_CONTACT">
    PLUSNET_MESSAGES
  </trigger_moos_var>
  <size>30</size>
  <layout>
    <static>
      <name>MessageType</name>
      <value>SENSOR_CONTACT</value>
    </static>
    <static>
      <name>SensorReportType</name>
      <value>1</value>
    </static>
    <int>
      <name>SourcePlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>DestinationPlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactTimestamp</name>
      <max>2000000000</max>
      <min>1000000000</min>
    </int>
    <float algorithm="angle_0_360">
      <name>SensorHeading</name>
      <max>360</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>SensorPitch</name>
      <max>90</max>
      <min>-90</min>
      <precision>0</precision>
    </float>
    <float>
      <name>SensorRoll</name>
      <max>180</max>
      <min>-180</min>
      <precision>0</precision>
    </float>
    <float>
      <name>SensorLatitude</name>
      <precision>5</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>SensorLongitude</name>
      <precision>5</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>SensorDepth</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <int>
      <name>SensorCEP</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <float>
      <name>ContactBearing</name>
      <max>360</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>ContactBearingUncertainty</name>
      <max>10</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>ContactBearingRate</name>
      <max>10</max>
      <min>-10</min>
      <precision>1</precision>
    </float>
    <float>
      <name>ContactBearingRateUncertainty</name>
      <max>3.1</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>ContactElevation</name>
      <max>90</max>
      <min>-90</min>
      <precision>1</precision>
    </float>
    <float>
      <name>ContactElevationUncertainty</name>
      <max>10</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <int>
      <name>ContactSpectralLevel1</name>
      <max>127</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactFrequency1</name>
      <max>4095</max>
      <min>0</min>
```

```
    </int>
    <int>
      <name>ContactBandwidth1</name>
      <max>20</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactSpectralLevel2</name>
      <max>127</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactFrequency2</name>
      <max>4095</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactBandwidth2</name>
      <max>20</max>
      <min>0</min>
    </int>
  </layout>
  <!-- decoding -->
  <publish>
    <moos_var>NAFCON_MESSAGES</moos_var>
    <all />
  </publish>
  <!-- start pTransponderAIS replacement -->
  <publish>
    <moos_var>AIS_REPORT</moos_var>
    <format>
      NAME=%1%,TYPE=%2%,UTC_TIME=%3$.0lf,
      X=%4%,Y=%5%,LAT=%6$lf,LON=%7$lf,
      SPD=,HDG=%8%,DEPTH=%9%
    </format>
    <message_var algorithm="modem_id2name,to_lower">
      SourcePlatformId</message_var>
    <message_var algorithm="modem_id2type,to_lower">
      SourcePlatformId</message_var>
    <message_var>ContactTimestamp</message_var>
    <message_var algorithm="lon2utm_x:SensorLatitude">
      SensorLongitude</message_var>
    <message_var algorithm="lat2utm_y:SensorLongitude">
      SensorLatitude</message_var>
    <message_var>SensorLatitude</message_var>
    <message_var>SensorLongitude</message_var>
    <message_var>SensorHeading</message_var>
    <message_var>SensorDepth</message_var>
  </publish>
  <publish>
    <moos_var type="double">
      %1%_NAV_UTC
    </moos_var>
    <format>%2$lf</format>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>ContactTimestamp</message_var>
  </publish>
  <publish>
    <moos_var type="double">
      %1%_NAV_X
    </moos_var>
    <format>%2%</format>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var algorithm="lon2utm_x:SensorLatitude">
      SensorLongitude</message_var>
  </publish>
  <publish>
    <moos_var type="double">
      %1%_NAV_Y
    </moos_var>
    <format>%2%</format>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var algorithm="lat2utm_y:SensorLongitude">
      SensorLatitude</message_var>
  </publish>
  <publish>
    <moos_var type="double">
      %1%_NAV_LAT
    </moos_var>
    <format>%2%</format>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>SensorLatitude</message_var>
  </publish>
  <publish>
    <moos_var type="double">
      %1%_NAV_LONG
    </moos_var>
    <format>%2$lf</format>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>SensorLongitude</message_var>
  </publish>
  <publish>
    <moos_var type="double">
      %1%_NAV_HEADING
    </moos_var>
    <format>%2%</format>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>SensorHeading</message_var>
  </publish>
  <publish>
    <moos_var type="double">
      %1%_NAV_DEPTH
    </moos_var>
    <format>%2%</format>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>SensorDepth</message_var>
  </publish>
  <!-- end pTransponderAIS replacement -->
</message>
...
```

Listing A.8 pGeneralCodex XML Configuration Block for Track Report

```
...
<message>
  <name>SENSOR_TRACK</name>
  <outgoing_hex_moos_var>
    OUT_TRACK_HEX_30B
  </outgoing_hex_moos_var>
  <incoming_hex_moos_var>
    IN_TRACK_HEX_30B
  </incoming_hex_moos_var>
  <trigger>publish</trigger>
  <trigger_moos_var
    mandatory_content="MessageType=SENSOR_TRACK">
    PLUSNET_MESSAGES
  </trigger_moos_var>
  <size>30</size>
  <layout>
    <static>
      <name>MessageType</name>
      <value>SENSOR_TRACK</value>
    </static>
    <static>
      <name>SensorReportType</name>
      <value>2</value>
    </static>
    <int>
      <name>SourcePlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>DestinationPlatformId</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackTimestamp</name>
      <max>2000000000</max>
      <min>1000000000</min>
    </int>
    <int>
      <name>PlatformID</name>
      <max>31</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackNumber</name>
      <max>255</max>
      <min>0</min>
    </int>
    <float>
      <name>TrackLatitude</name>
      <precision>5</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>TrackLongitude</name>
      <precision>5</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>TrackCEP</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackDepth</name>
      <max>1023</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackDepthUncertainty</name>
      <max>63</max>
      <min>0</min>
    </int>
    <float algorithm="angle_0_360">
      <name>TrackHeading</name>
      <max>360</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>TrackHeadingUncertainty</name>
      <max>10</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>TrackSpeed</name>
      <max>12.8</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <float>
      <name>TrackSpeedUncertainty</name>
      <max>3</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <int>
      <name>DepthClassification</name>
      <max>7</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackClassification</name>
      <max>7</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackSpectralLevel1</name>
      <max>127</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackFrequency1</name>
      <max>4095</max>
      <min>0</min>
    </int>
    <int>
      <name>TrackBandwidth1</name>
      <max>20</max>
      <min>0</min>
```

```
  </int>
  <int>
    <name>TrackSpectralLevel2</name>
    <max>127</max>
    <min>0</min>
  </int>
  <int>
    <name>TrackFrequency2</name>
    <max>4095</max>
    <min>0</min>
  </int>
  <int>
    <name>TrackBandwidth2</name>
    <max>20</max>
    <min>0</min>
  </int>
</layout>
<!-- decoding -->
<publish>
  <moos_var>NAFCON_MESSAGES</moos_var>
  <all />
</publish>
<!-- start pTransponderAIS replacement -->
<publish>
  <moos_var>AIS_REPORT</moos_var>
  <format>
    NAME=TRK_%1%_%2%,TYPE=track,UTC_TIME=%3$.0lf,
    X=%4%,Y=%5%,LAT=%6$lf,LON=%7$lf,
    SPD=%8%,HDG=%9%,DEPTH=%10%</format>
  <message_var>TrackNumber</message_var>
  <message_var algorithm="modem_id2name,to_lower">
    SourcePlatformId</message_var>
  <message_var>TrackTimestamp</message_var>
  <message_var algorithm="lon2utm_x:TrackLatitude">
    TrackLongitude</message_var>
  <message_var algorithm="lat2utm_y:TrackLongitude">
    TrackLatitude</message_var>
  <message_var>TrackLatitude</message_var>
  <message_var>TrackLongitude</message_var>
  <message_var>TrackSpeed</message_var>
  <message_var>TrackHeading</message_var>
  <message_var>TrackDepth</message_var>
</publish>
<publish>
  <moos_var type="double">
    TRK_%1%_%2%_NAV_UTC
  </moos_var>
  <format>%3$lf</format>
  <message_var>TrackNumber</message_var>
  <message_var algorithm="modem_id2name,to_upper">
    SourcePlatformId</message_var>
  <message_var>TrackTimestamp</message_var>
</publish>
<publish>
  <moos_var type="double">
    TRK_%1%_%2%_NAV_X
  </moos_var>
  <format>%3%</format>
  <message_var>TrackNumber</message_var>
  <message_var algorithm="modem_id2name,to_upper">
    SourcePlatformId</message_var>
  <message_var algorithm="lon2utm_x:TrackLatitude">
    TrackLongitude</message_var>
</publish>
  <publish>
    <moos_var type="double">TRK_%1%_%2%_NAV_Y</moos_var>
    <format>%3%</format>
    <message_var>TrackNumber</message_var>
    <message_var algorithm="modem_id2name,to_upper">
     SourcePlatformId</message_var>
    <message_var algorithm="lat2utm_y:TrackLongitude">
            TrackLatitude</message_var>
  </publish>
  <publish>
    <moos_var type="double">TRK_%1%_%2%_NAV_LAT</moos_var>
    <format>%3$lf</format>
    <message_var>TrackNumber</message_var>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>TrackLatitude</message_var>
  </publish>
  <publish>
    <moos_var type="double">TRK_%1%_%2%_NAV_LONG</moos_var>
    <format>%3$lf</format>
    <message_var>TrackNumber</message_var>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>TrackLongitude</message_var>
  </publish>
  <publish>
    <moos_var type="double">TRK_%1%_%2%_NAV_SPEED</moos_var>
    <format>%3%</format>
    <message_var>TrackNumber</message_var>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>TrackSpeed</message_var>
  </publish>
  <publish>
    <moos_var type="double">TRK_%1%_%2%_NAV_HEADING</moos_var>
    <format>%3%</format>
    <message_var>TrackNumber</message_var>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>TrackHeading</message_var>
  </publish>
  <publish>
    <moos_var type="double">TRK_%1%_%2%_NAV_DEPTH</moos_var>
    <format>%3%</format>
    <message_var>TrackNumber</message_var>
    <message_var algorithm="modem_id2name,to_upper">
      SourcePlatformId</message_var>
    <message_var>TrackDepth</message_var>
  </publish>
  <!-- end pTransponderAIS replacement -->
</message>
...
```

Listing A.9 pGeneralCodex XML Configuration file `nafcon_targetsim.xml` *for transmitting simulated target state to all network nodes synchronously.*

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<!-- codec for broadcasting target state for simulators
 running on nodes -->
<message_set>
  <message>
    <name>SIM_TARGET</name>
    <outgoing_hex_moos_var>
      OUT_SIM_TGT_HEX_30B
    </outgoing_hex_moos_var>
    <incoming_hex_moos_var>
      IN_SIM_TGT_HEX_30B
    </incoming_hex_moos_var>
    <trigger>publish</trigger> <!-- publish to moos var -->
    <trigger_moos_var mandatory_content="tgt_delay=">
      TGT_STATE_OUT
    </trigger_moos_var>
    <size>30</size>
    <layout>
      <int>
        <name>tgt_x</name>
        <precision>1</precision>
        <max>100000</max>
        <min>-100000</min>
      </int>
      <int>
        <name>tgt_y</name>
        <precision>1</precision>
        <max>100000</max>
        <min>-100000</min>
      </int>
      <float>
        <name>tgt_depth</name>
        <precision>1</precision>
        <max>1000</max>
        <min>0</min>
      </float>
      <int>
        <name>tgt_hdg</name>
        <max>359</max>
        <min>0</min>
      </int>
      <float>
        <name>tgt_speed</name>
        <precision>1</precision>
        <max>30</max>
        <min>0</min>
      </float>
      <int>
        <name>tgt_freq</name>
        <max>5000</max>
        <min>0</min>
      </int>
      <int>
        <name>tgt_bw</name>
        <max>1000</max>
        <min>0</min>
      </int>
      <int>
        <name>tgt_spl</name>
        <max>200</max>
        <min>0</min>
      </int>
      <int>
        <name>tgt_delay</name>
        <max>5000</max>
        <min>0</min>
      </int>
      <int>
        <name>tn1_freq</name>
         <max>1000</max>
        <min>0</min>
      </int>
      <int>
        <name>tn1_dbl</name>
        <max>200</max>
        <min>0</min>
      </int>
      <int>
        <name>tn2_freq</name>
        <max>1000</max>
        <min>0</min>
      </int>
      <int>
        <name>tn2_dbl</name>
        <max>200</max>
        <min>0</min>
      </int>
      <int>
        <name>tn3_freq</name>
        <max>1000</max>
        <min>0</min>
      </int>
      <int>
        <name>tn3_dbl</name>
        <max>200</max>
        <min>0</min>
      </int>
      <int>
        <name>tn4_freq</name>
        <max>1000</max>
        <min>0</min>
      </int>
      <int>
        <name>tn4_dbl</name>
        <max>200</max>
        <min>0</min>
      </int>
      <int>
        <name>tn5_freq</name>
        <max>1000</max>
        <min>0</min>
      </int>
      <int>
        <name>tn5_dbl</name>
        <max>200</max>
        <min>0</min>
      </int>
      <int>
        <name>time</name>
        <max>2000000000</max>
```

```
        <min>1000000000</min>
      </int>
      <int>
        <name>src_num</name>
        <max>31</max>
        <min>0</min>
      </int>
    </layout>
    <!-- decoding -->
    <publish>
      <moos_var>TGT_STATE_IN</moos_var>
      <all />
    </publish>
    <publish>
      <moos_var>TARGET_CONTROL</moos_var>
      <format>ON</format>
    </publish>
  </message>
</message_set>
```