Nested Autonomy for Unmanned Marine Vehicles with MOOS-IvP

Michael R. Benjamin
Naval Undersea Warfare Center, Newport, Rhode Island 02841, and Department of Mechanical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139
e-mail: michael.r.benjamin@navy.mil
Henrik Schmidt
Department of Mechanical Engineering, Laboratory for Autonomous Marine Sensing Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139
e-mail: henrik@mit.edu
Paul M. Newman
Department of Engineering Science, University of Oxford, Parks Road, Oxford OX1 3PJ, United Kingdom
e-mail: pnewman@robots.ox.ac.uk
John J. Leonard
Department of Mechanical Engineering, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139
e-mail: jleonard@mit.edu

Received 15 January 2010; accepted 16 August 2010

This document describes the MOOS-IvP autonomy software for unmanned marine vehicles and its use in large-scale ocean sensing systems. MOOS-IvP is composed of two open-source software projects funded by the Office of Naval Research. MOOS provides a core autonomy middleware capability, and the MOOS project additionally provides a set of ubiquitous infrastructure utilities. The IvP Helm is the primary component of an additional set of capabilities implemented to form a full marine autonomy suite known as MOOS-IvP. This software and architecture are platform and mission agnostic and allow for a scalable nesting of unmanned vehicle nodes to form large-scale, long-endurance ocean sensing systems composed of heterogeneous platform types with varying degrees of communications connectivity, bandwidth, and latency. Published 2010 Wiley Periodicals, Inc.*

1. INTRODUCTION

The growing desire for autonomy in unmanned marine systems is driven by several trends, including increased complexity in mission objectives and duration, increased capability in onboard sensor processing and computing power, and an increase in the number of users and owners of unmanned vehicles. The MOOS-IvP project is an open-source project designed and developed in this context. It is an implementation of an autonomous helm and substantial support applications that aims to provide a capable autonomy system out of the box. It also has an architecture, software policy, documentation, and support network that allows this newer generation of scientists, with newer vehicles and mission ambitions, to be nimble to build innovative autonomy algorithms to augment an existing set of capabilities. This paper describes the MOOS-IvP autonomy architecture and software structure and describes how groups of vehicles, each with different sensors, processing power, and communications capabilities, may be combined to form a nested autonomy architecture with identical core autonomy software running on each platform.

MOOS-IvP is composed of two distinct open-source software projects. The Mission Oriented Operating Suite (MOOS) is a product of the Mobile Robotics Group at the University of Oxford and provides core middleware capabilities in a publish–subscribe architecture, as well as several applications ubiquitous in unmanned marine robotic and land robotic applications using MOOS. Additional MOOS applications, including the IvP Helm, are available in the MOOS-IvP project. IvP stands for Interval Programming and refers to the multiobjective optimization method used by the IvP Helm for arbitrating between competing behaviors in its behavior-based architecture.

The MOOS-IvP software is available on the web via anonymous read-only access (Benjamin, Schmidt, & Leonard, n.d.). It consists of more than 120,000 lines of C++, comprising about 30 distinct applications and more than a dozen vehicle behaviors. It represents about 20 work years of effort or more from individual contributors. Autonomy configurations and missions in this environment have been tested in several thousands of hours of simulation and several hundred hours of in-water experiments, on platforms including the Bluefin 21-in. unmanned underwater

 Published 2010 Wiley Periodicals, Inc. *This article is a US Government work and, as such, is in the public domain of the United States of America.
View this article online at wileyonlinelibrary.com • DOI: 10.1002/rob.20370

vehicle (UUV), the Hydroid REMUS-100 and REMUS-600 UUVs, the Ocean Server Iver2 UUV, the Ocean Explorer 21-in. UUV, autonomous kayaks from Robotic Marine Systems and SARA, Inc., and two larger unmanned surface vehicles (USVs) from the NATO Underwater Research Center in La Spezia, Italy.

1.1. Trends in Unmanned Marine Vehicles Relating to Autonomy

The algorithms and software described in this paper have their genesis in UUVs. Unlike unmanned sea-surface, ground, and aerial vehicles, underwater vehicles cannot be remotely controlled; they *must* make decisions autonomously due to the low bandwidth in acoustic communications. Remote control, or teleoperation, in land, air, or surface vehicles may be viewed as a means to allow conservative, risk-averse operation with respect to the degree of autonomy afforded to the vehicle. In underwater vehicles, similar conservative tendencies are realized by scripting the vehicle missions to be as predictable as possible. Missions typical of early-model UUVs were composed of a preplanned set of waypoints accompanied by depth and perhaps speed parameters. The onboard sensors merely collected data that were analyzed after the vehicle was recovered from the water.

Advances in sensor technologies include greater capabilities at lower cost, lower size, and lower power consumption. The same is true for the onboard computing components needed to process sensor data. Increasingly underwater vehicles are able to see, hear, and localize objects and other vehicles in their environment and quickly analyze an array of qualities in water samples taken while underway. Likewise, the available mission duration at depth has grown longer due to improvements in inertial navigation systems (INSs), which have become cheaper, smaller, and more accurate, and due to improvements in platform battery life. Each of these trends has contributed to making a UUV owner less satisfied with simply collecting the data and analyzing the results in a postmission analysis phase. The information and analysis are available in stride, in situ: Why not act on that information in stride to the advantage of the mission objectives? Enter adaptive autonomy.

The chart in Figure 1 conveys a rough timeline and relationship between the evolution of UUV autonomy capabilities and the evolution of other critical UUV technologies. The notion of *adaptive* in adaptive autonomy is a sliding scale and refers to the ability to allow increasing amounts of sensor information to affect in-stride autonomy decisions. On one end of the scale, even a vehicle that deterministically follows a set of waypoints may be adapting its heading decisions based on an INS or global positioning system (GPS) sensor. However, sensors that are capable of perceiving qualities about the vehicle's environment, including water quality, bottom type, artifacts, and other moving vehicles, are able to alter the flow of autonomy decisions in a much more profound manner.

The notion of *collaboration* in collaborative autonomy may be viewed as a sliding scale as well. At one end of the spectrum are vehicles deployed alongside each other, executing a mission independently but each contributing to a joint mission. In this case, the collaboration occurs in the predeployment mission planning process. When at least

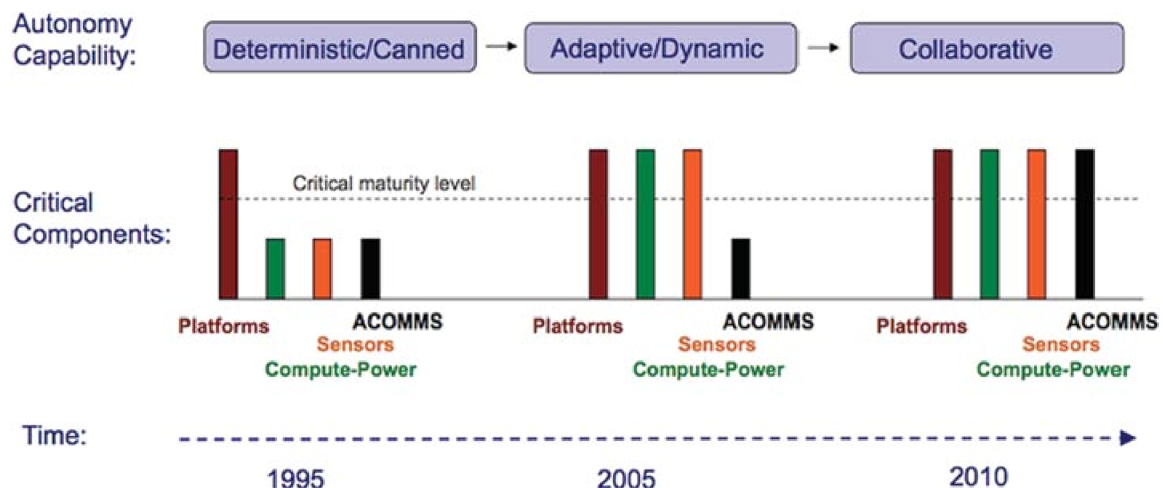

Figure 1. UUV technologies and autonomy: A rough timeline and relationship between UUV autonomy and other critical UUV technologies. Critical components include (a) the platform itself in terms of reliability, cost, and endurance; (b) onboard computing power and sensor processing; (c) onboard sensors in terms of resolution, size, and cost; and (d) ACOMMS. Each of these maturing technology trends affects what is possible and desired from the onboard autonomy system. The corresponding trend in autonomy is from deterministic vehicles acting independently toward adaptive vehicles acting in collaboration.

periodic communication between deployed vehicles is feasible, a whole different kind of collaboration is possible, especially when each vehicle is able to adapt components of its mission to both its sensed environment *and* incoming communications from other vehicles. Advances in underwater acoustic communications (ACOMMS) in terms of reliability, range, flexibility in defining message sets, and bandwidth have enabled the development of adaptive, collaborative autonomy. This trend also occurs in the context of declining cost and size of commercially available UUVs, making it possible for even medium-sized organizations to own and operate several vehicles.

The MOOS-IvP autonomy architecture has been developed and refined in this context of migration to adaptive, collaborative autonomy. Mission structure is defined less in terms of a sequence of tasks, but rather as a set of autonomy modes with conditions, events, and field commands defining the transitions between modes. The modes correlate to a set of one or more active behaviors, in which each behavior may be its own substantial autonomy subcomponent. An autonomy system that includes the ability to adapt its mission to the environment, other collaborating vehicles, and periodic messages from within a field-control hierarchy will inevitably need to balance competing objectives in a way that reflects a singular mission focus. This paper also discusses how multiobjective optimization is used at the behavior coordination level in the helm to achieve this design objective.

1.2. The Backseat Driver Design Philosophy

The main idea in the backseat driver paradigm is the separation between *vehicle control* and *vehicle autonomy*. The vehicle control system runs on a platform's main vehicle computer, and the autonomy system runs on a separate payload computer. This separation is also referred to as the *mission controller–vehicle controller* interface. A primary benefit is the decoupling of the platform autonomy system from the actual vehicle hardware. The vehicle manufacturer provides a navigation and control system capable of streaming vehicle position and trajectory information to the payload computer and accepting a stream of autonomy decisions such as heading, speed, and depth in return. Exactly how the vehicle navigates and implements control is largely unspecified to the autonomy system running in the payload. The relationship is depicted in Figure 2.

The autonomy system on the payload computer consists of a set of distinct processes communicating through a publish–subscribe database called the MOOSDB (Mission Oriented Operating Suite—Database). One such process is an interface to the main vehicle computer, and another key process is the IvP Helm implementing the behavior-based autonomy system. The MOOS community is referred to as the "larger autonomy" system, or the "autonomy system as a whole" because MOOS itself is middleware, and actual autonomous decision making, sensor processing, contact management, etc., are implemented as individual MOOS processes.

1.3. The Publish–Subscribe Middleware Design Philosophy and MOOS

MOOS provides a middleware capability based on the publish–subscribe architecture and protocol. Each process communicates with the others through a single database process in a star topology (Figure 3). The interface of a

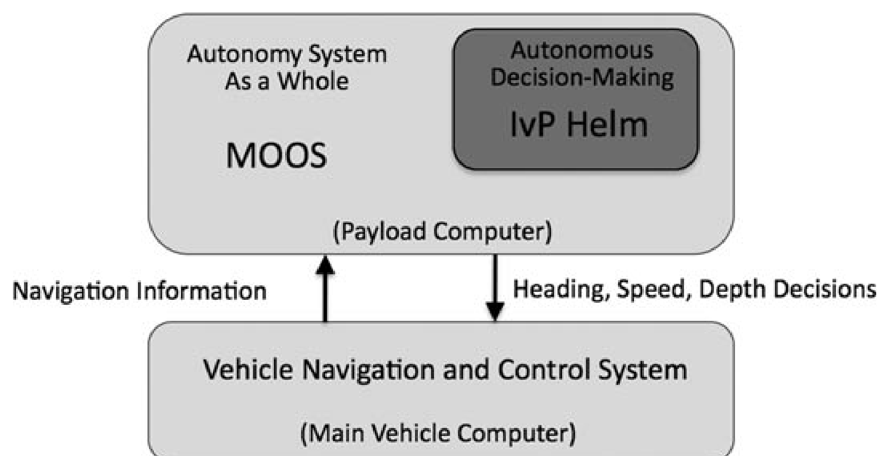

Figure 2. The backseat driver paradigm: The key idea is the separation of vehicle autonomy from vehicle control. The autonomy system provides heading, speed, and depth commands to the vehicle control system. The vehicle control system executes the control and passes navigation information, e.g., position, heading, and speed, to the autonomy system. The backseat paradigm is agnostic regarding how the autonomy system implemented, but in this figure the MOOS-IvP autonomy architecture is depicted.

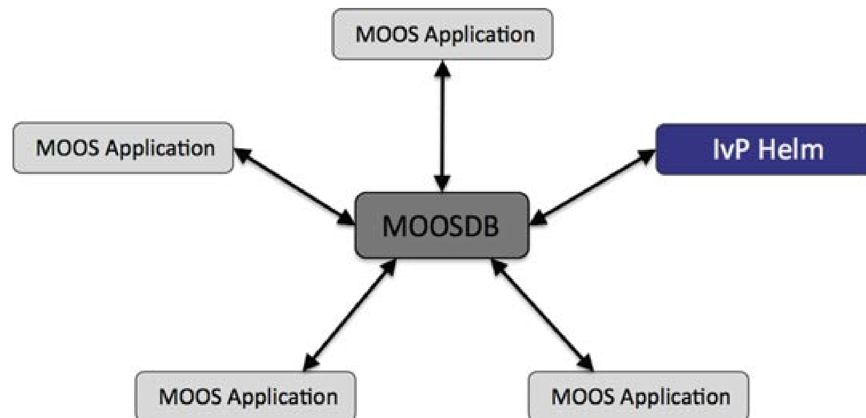

Figure 3. A MOOS community is a collection of MOOS applications typically running on a single machine, each with a separate process ID. Each process communicates through a single MOOS database process (the MOOSDB) in a publish–subscribe manner. Each process may be executing its inner loop at a frequency independent of the others and set by the user. Processes may all be run on the same computer or distributed across a network.

particular process is described by what messages it produces (publications) and what messages it consumes (subscriptions). Each message is a simple variable-value pair in which the values are limited to either string or numerical values such as (`STATE, "DEPLOY"`) or (`NAV_SPEED, 2.2`). Limiting the message type reduces the compile dependencies between modules and facilitates debugging because all messages are human readable.

The key idea with respect to facilitating code reuse is that applications are largely independent, defined only by their interface, and any application is easily replaceable with an improved version with a matching interface. Because MOOS Core and many common applications are publicly available along with source code under an open-source GNU Public License (GPL), a user may develop an improved module by altering existing source code and introduce a new version under a different name. The term MOOS Core refers to (a) the MOOSDB application and (b) the MOOS application superclass that each individual MOOS application inherits from to allow connectivity to a running MOOSDB. Holding the MOOS Core part of the codebase constant between MOOS developers enables the plug-and-play nature of applications.

1.4. The Behavior-Based Control Design Philosophy and IvP Helm

The IvP Helm runs as a single MOOS application and uses a behavior-based architecture for implementing autonomy. Behaviors are distinct software modules that can be described as self-contained mini-expert systems dedicated to a particular aspect of overall vehicle autonomy. The helm implementation and each behavior implementation expose an interface for configuration by the user for a particular set of missions. This configuration often contains particulars such as a certain set of waypoints, search area, and vehicle speed. It also contains a specification of mission modes that determine which behaviors are active under which situations and how states are transitioned. When multiple behaviors are active and competing for influence of the vehicle, the IvP solver is used to reconcile the behaviors (Figure 4).

The solver performs this coordination by soliciting an objective function, i.e., utility function, from each behavior defined over the vehicle decision space, e.g., possible settings for heading, speed, and depth. In the IvP Helm, the objective functions are of a certain type—piecewise linearly defined—and are called IvP functions. The solver algorithms exploit this construct to find a rapid solution to the optimization problem composed of the weighted sum of contributing functions.

The concept of a behavior-based architecture is often attributed to Brooks (1986). Since then, various solutions to the issue of action selection, i.e., the issue of coordinating competing behaviors, have been put forth and implemented in physical systems. The simplest approach is to prioritize behaviors in such a way that the highest-priority behavior locks out all others as in the subsumption architecture in Brooks (1986). Another approach is referred to as the potential fields, or vector summation approach [see Arkin (1987) and Khatib (1985)], which considers the average action between multiple behaviors to be a reasonable compromise. These action-selection approaches have been used with reasonable effectiveness on a variety of platforms, including indoor robots, e.g., Arkin (1987), Arkin, Carter, and Mackenzie (1993), Pirjanian (1998), and Riekki (1999); land vehicles, e.g., Rosenblatt (1997); and marine vehicles, e.g., Bennet and Leonard (2000), Carreras, Batlle, and Ridao (2000), Kumar and Stover (2001), Rosenblatt, Williams, and Durrant-Whyte (2002), and Williams,

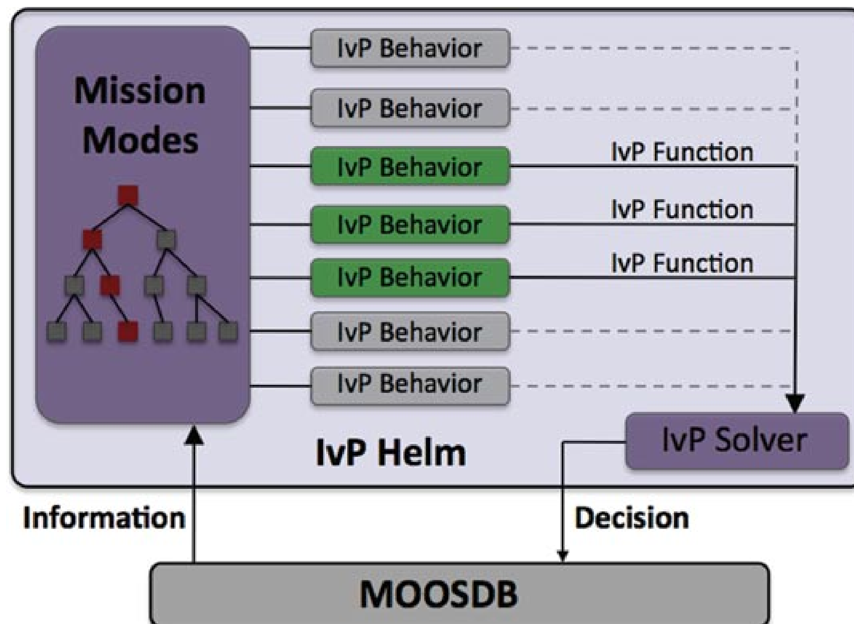

Figure 4. The IvP Helm. The helm is a single MOOS application running as the process pHelmIvP. It is a behavior-based architecture in which the primary output of a behavior on each iteration is an IvP objective function. The IvP solver performs multiobjective optimization on the set of functions to find the single best vehicle action, which is then published to the MOOSDB. The functions are built and the set is solved on *each* iteration of the helm—typically one to four times per second. Only a subset of behaviors is active at any given time, depending on the vehicle situation and the state space configuration provided by the user.

Newman, Dissanayake, Rosenblatt, and Durrant-Whyte (2000). However, action selection via the identification of a single highest-priority behavior and via vector summation have well-known shortcomings later described in Pirjanian (1998), Riekki (1999), and Rosenblatt (1997) in which the authors advocated for the use of multiobjective optimization as a more suitable, although more computationally expensive, method for action selection. The IvP model is a method for implementing multiobjective function-based action selection that is computationally viable in the IvP Helm implementation.

1.5. The Nested Autonomy Paradigm

For large-scale subsurface/surface ocean monitoring and observation systems (100+ km^2), no single unmanned platform has the ability in terms of sensing, endurance, and communications to achieve large-scale, long-endurance (several days to several weeks) system objectives. Even if multiple platforms are applied to the problem, effectiveness may be substantially diminished if limited to a single platform *type*. The *nested autonomy* paradigm, depicted in Figure 5, is an approach to implementing a system of unmanned platforms for large-scale autonomous sensing applications. It is based in part on the objective of making seamless use of heterogeneous platform types using a uniform platform-independent autonomy architecture. It also assumes that the platforms will have differing communications bandwidth, connectivity, and latency.

The *vertical* connectivity allows information to pass from sensors to the onboard sensor processing and autonomy modules or from each node to other nodes in the cluster or up to the field operator and thus forms the basis for the autonomous *adaptive control* that is a key to the capability in 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 bytes/min for the internode acoustic modem communications (ACOMMS) to 100 Mbytes/s for the onboard systems. Equally important, the layers of the vertical connectivity differ significantly in latency and intermittency, ranging from virtually instantaneous connectivity of the onboard sensors and control processes to latencies of 10–30 min for information flowing to and from the field control operators. This, in turn, has critical implications for the timescales 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 a hourly to

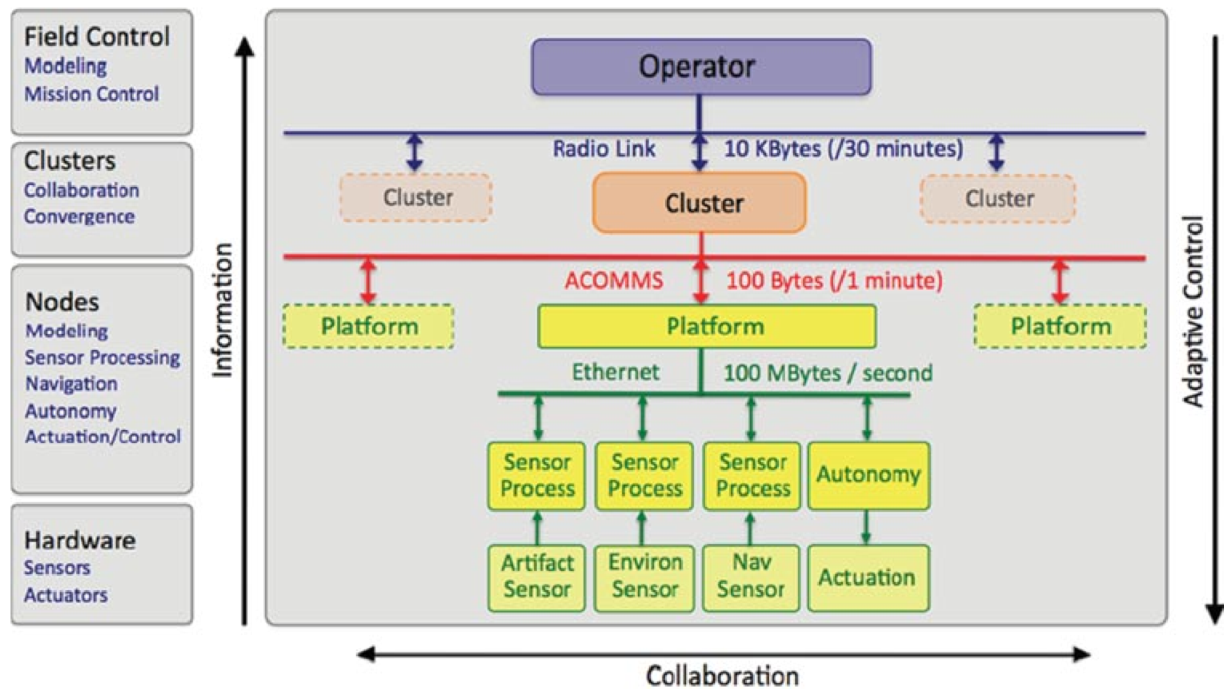

Figure 5. The nested autonomy paradigm: Field control operators receive intermittent information from field nodes as connectivity and bandwidth allow. Elements of clusters may serve a heterogeneous role as a gateway communications agent. Likewise, nodes receive intermittent commands and cues from field operators. Node autonomy compensates for and complements the sporadic connectivity to field control and other nodes in a cluster or network of clusters.

a 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. Shorter timescale 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, as much as a centralized control paradigm is infeasible for such systems, it is also unlikely that a concept of operations based entirely on nodal autonomy is optimal. Instead, some combination will likely be optimal, but in view of the severe latency of the *vertical* communication channels, the *nested autonomy* concept of operations described is heavily tilted toward autonomy.

The MOOS-IvP autonomy implementation discussed in this paper is situated primarily at the node level in the nested autonomy structure depicted in Figure 5. However, aspects of the MOOS-IvP architecture are relevant to the larger picture as well. A key enabling factor of the nested autonomy paradigm is the platform independence of the node-level autonomy system. The backseat driver design permits the decoupling of the vehicle platform from the autonomy system to achieve platform independence. The MOOS middleware architecture and the IvP Helm behavior-based architecture also contribute to platform independence by allowing an autonomy system to be composed of modules that are swappable across platform types. Furthermore, collaborative and nested autonomy between nodes is facilitated by the simple modal interface to the onboard autonomy missions to control behavior activations.

2. A VERY BRIEF OVERVIEW OF MOOS

MOOS is often described as autonomy "middleware," which it can be argued is shorthand for the glue that connects a collection of applications in which the "real" work is going on. MOOS does indeed connect a collection of applications, of which the IvP Helm is one. However, each application inherits a generic MOOS interface whose implementation provides a powerful, easy-to-use means of communicating with other applications and controlling the relative frequency at which the application executes its primary set of functions. Owing to its combination of ease of use, general extendability, and reliability, it has been used in the classroom by students with no prior experience, as well as on many extended field exercises with substantial robotic resources at stake. To frame the later discussion of the IvP Helm, the basic issues regarding

MOOS applications are introduced here. For further information on MOOS, see Newman (2003).

2.1. Interprocess Communication with Publish/Subscribe

MOOS has a star-like topology as depicted in Figure 3. Each application within a MOOS community (a MOOSApp) has a connection to a single MOOS database (called MOOSDB) that lies at the heart of the software suite. All communication happens via this central server application. The network has the following properties:

- No peer-to-peer communication.
- All communication between the client and server is instigated by the client, i.e., the MOOSDB never makes a unsolicited attempt to contact a MOOSApp.
- Each client has a unique name.
- A given client need have no knowledge of which other clients exist.
- A client has no way of transmitting data to a given client—data can be sent only to the MOOSDB.
- The star network can be distributed over any number of machines running any combination of supported operating systems.

This centralized topology is obviously vulnerable to bottlenecking at the server regardless of how well written the server is. However, the advantages of such a design are perhaps greater than its disadvantages. First the network remains simple regardless of the number of participating clients. The server has complete knowledge of all active connections and can take responsibility for the allocation of communication resources. The clients operate independently with interconnections. This prevents rogue clients (badly written or hung) from directly interfering with other clients.

2.2. Message Content

The communications API in MOOS allows data to be transmitted between the MOOSDB and a client. The meaning of the data is dependent on the role of the client. However, the form of the data is constrained by MOOS. Somewhat unusually, MOOS allows for data to be sent only in string or double form. Data are packed into messages (CMOOSMsg class) that contain other salient information shown in Table I.

The fact that data are commonly sent in string format is often seen as a strange and inefficient aspect of MOOS. For example, the string `"Type=EST,Name=AUV,Pos=[3x1]3.4,6.3,-0.23"` might describe the position estimate of a vehicle called "AUV" as a 3×1 column vector. Typically string data in MOOS are a concatenation of comma-separated "name = value" pairs. It is true that using custom binary data formats does decrease the number of bytes sent. However, binary data are unreadable to humans and require structure declarations to decode them and header file dependencies are to be avoided where possible. The communications efficiency argument is not as compelling as one may initially think. The CPU cost invoked in sending a TCP/IP packet is largely independent of size up to about 1,000 bytes. So it is as costly to send 2 bytes as it is 1,000. In this light there is basically no penalty in using strings. There is, however, an additional cost incurred in parsing string data that is far in excess of that incurred when simply casting binary data. Irrespective of this, experience has shown that the benefits of using strings far outweigh the difficulties. In particular,

Table I. The contents of a MOOS message.

Variable	Meaning
Name	The name of the data
String value	Data in string format
Double value	Numeric double float data
Source	Name of client that sent the data to the MOOSDB
Auxiliary	Supplemental message information, e.g., IvP behavior source
Time	Time at which the data were written
Data type	Type of data (`STRING` or `DOUBLE`)
Message type	Type of message (usually `NOTIFICATION`)
Source community	The community to which the source process belongs

- Strings are human readable.
- All data become the same type.
- Logging files are human readable (they can be compressed for storage).
- Replaying a log file is a case of simply reading strings from a file and "throwing" them back at the MOOSDB in time order.
- The contents and internal order of strings transmitted by an application can be changed without the need to recompile consumers (subscribers to those data)—users simply would not understand new data fields, but they would not crash.

Of course, scalar data need not be transmitted in string format—for example, the depth of a subsea vehicle. In this case the data would be sent while setting the data type to `"MOOS_DOUBLE"` and writing the numeric value in the double data field of the message.

2.3. Mail Handling—Publish/Subscribe—in MOOS

Each MOOS application is a client having a connection to the MOOSDB. This connection is made on the client side, and the client manages a private thread that coordinates the communication with the MOOSDB. This thread completely hides the intricacies and timings of the communications

from the rest of the application and provides a small, well-defined set of methods to handle data transfer. By having this thread automatically available to each MOOS application, the application can do the following:

1. Publish data—issue a notification on named data.
2. Register for notifications on named data.
3. Collect notifications on named data—reading mail.

Publishing data: Data are published as a pair—a variable and value—that constitutes the heart of a MOOS message describe in Table I. The client invokes the `Notify(VarName, VarValue)` command where appropriate in the client code. The above command is implemented for both string values and double values, and the rest of the fields described in Table I are filled in automatically. Each notification results in another entry in the client's "outbox," which is emptied the next time the MOOSDB accepts an incoming call from the client.

Registering for notifications: Assume that a list of names of data published has been provided by the author of a particular MOOS application. For example, an application that interfaces to a GPS sensor may publish data called `GPS_X` and `GPS_Y`. A different application may register its interest in these data by subscribing or registering for them. An application can register for notifications using a single method `Register` specifying both the name of the data and the maximum rate at which the client would like to be informed that the data have been changed. The latter parameter is specified in terms of the minimum possible time between notifications for a named variable. For example, setting it to zero would result in the client receiving each and every change notification issued on that variable.

Reading mail: A client can inquire at any time whether it has received any new notifications from the MOOSDB by invoking the `Fetch` method. The function fills in a list of notification messages with the fields given in Table I. Note that a single call to `Fetch` may result in being presented with several notifications corresponding to the same named data. This implies that several changes were made to the data since the last client–server conversation. However, the time difference between these similar messages will never be less than that specified in the `Register` function described above. In typical applications the `Fetch` command is called on the client's behalf just prior to the `Iterate` method, and the messages are handled in the user overloaded `OnNewMail` method. These methods are described next.

2.4. Overloaded Functions in MOOS Applications

MOOS provides a base class called CMOOSApp, which simplifies the writing of a new MOOS application as a derived subclass. Beneath the hood of the CMOOSApp class

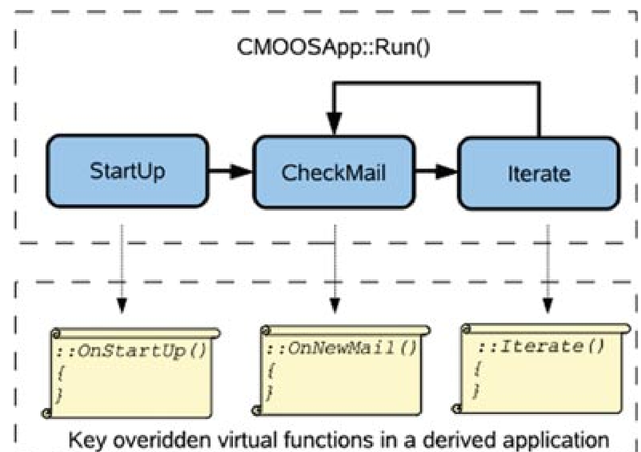

Figure 6. Key virtual functions of the MOOS application base class: The flow of execution once `Run()` has been called on a class derived from CMOOSApp. The scrolls indicate where users of the functionality of CMOOSApp will be writing new code that implements whatever it is that is wanted from the new applications.

is a loop that repetitively calls a function called `Iterate()`, which by default does nothing. One of the jobs as a writer of a new MOOS-enabled application is to flesh this function out with the code that makes the application do what we want. Behind the scenes this overall loop in CMOOSApp is also checking to see whether new data have been delivered to the application. If they have, another virtual function, `OnNewMail()`, is called. This is the function within which code is written to process the newly delivered data.

The roles of the three virtual functions in Figure 6 are discussed below. The `pHelmIvP` application does indeed inherit from CMOOSApp and overload these three functions. The base class contains other virtual functions (`OnConnectToServer()` and `OnDisconnectFromServer()`) not discussed here but discussed in Newman (2003).

The Iterate() method: By overriding the `CMOOSApp::Iterate()` function in a new derived class, the author creates a function from which the work that the application is tasked with doing can be orchestrated. In the `pHelmIvP` application, this method will consider the next-best vehicle decision, typically in the form of deciding values for the vehicle heading, speed, and depth. One may configure the rate at which the `Iterate()` method is called by invoking the `SetAppFreq()` method or by specifying the `AppTick` parameter in the mission file. Note that the requested frequency specifies the maximum frequency at which `Iterate()` will be called—it does not guarantee that it will be called at the requested rate. For example, if you write code in `Iterate()` that takes 1 s to complete, there is no way that this method can be called at more than 1 Hz. If

you want to call `Iterate()` as fast as is possible, simply request a frequency of zero—but you may want to reconsider why you need such a greedy application.

The OnNewMail() method: Just before `Iterate()` is called, the CMOOSApp base class determines whether new mail is present, i.e., whether some other process has posted data for which the client has previously registered, as described above. If new mail is waiting, the CMOOSApp base class calls the `OnNewMail()` virtual function, typically overloaded by the application. The mail arrives in the form of a list of CMOOSMsg objects (see Table I). The programmer is free to iterate over this collection, examining who sent the data, what they pertains to, and how old they are, and whether they are string or numerical data and to act on or process the data accordingly.

The OnStartup() method: This function is called just before the application enters into its own forever-loop depicted in Figure 6. This is the application that implements the application's initialization code and in particular reads configuration parameters (including those that modify the default behavior of the CMOOSApp base class) from a file.

2.5. MOOS Applications in the Public Domain

Below are very brief descriptions of MOOS applications in the public domain. This is not a complete list. It does not include applications outside MIT, Oxford, and the Naval Undersea Warfare Center (NUWC), and it is not a complete list of applications from those organizations. For a more in-depth tour of MOOS applications, see Benjamin, Newman, Schmidt, and Leonard (2009).

pAntler: A tool for launching a collection of MOOS processes given a mission file. See Newman (n.d., 2003).

pMOOSBridge: A tool that allows messages to pass between MOOS communities and allows for the renaming of messages as they are shuffled between communities. See Newman (n.d., 2003).

pLogger: A logger for recording the activities of a MOOS session. It can be configured to record a fraction or all publications of any number of MOOS variables. See Newman (n.d.).

pScheduler: A simple tool for generating and responding to messages sent to the MOOSDB by processes in a MOOS community. See Newman (n.d.).

uMS: A graphical user interface (GUI)-based MOOS scope for monitoring one or more MOOSDBs. See Newman (n.d.).

uPlayback: A cross-platform GUI application that can load in log files and replay them into a MOOS community as though the originators of the data were really running and issuing notifications. See Newman (n.d.).

iMatlab: An application that allows Matlab to join a MOOS community—even if only for listening in and rendering sensor data. It allows connection to the MOOSDB and access to local serial ports. See Newman (n.d.).

iRemote: A terminal-based tool for remote control of a platform running MOOS. It can be configured to associate a predefined variable-value poke with any unmapped key on the keyboard. See Newman (n.d.).

uMVS: A multivehicle autonomous underwater vehicle (AUV) simulator, capable of simulating any number of vehicles and acoustic ranging between them and acoustic transponders. It uses a six-degree-of-freedom vehicle model replete with vehicle dynamics, center of buoyancy, center of gravity geometry, and velocity-dependent drag. The acoustic simulation may also simulate acoustic packets propagating as spherical shells through the water column. See Newman (n.d.).

pHelmIvP: The IvP Helm, and primary focus of this document.

pNodeReporter: Garners vehicle navigation information such as position, speed, heading, and depth, along with other information such as its operation mode and platform type, and publishes a summary variable, consumed by viewer applications such as `pMarineViewer`, and as input to other vehicles participating in cooperative tasks. See Benjamin (2010).

uXMS: A terminal-based tool for monitoring the contents of a MOOSDB process, or history of given variable within a MOOSDB. See Benjamin (2010) and Benjamin et al. (n.d.).

uHelmScope: A terminal-based tool for showing information about a running instance of the IvP Helm. It also embeds a general-purpose scoping utility similar to uXMS. See Benjamin (2010) and Benjamin et al. (n.d.).

uPokeDB: A command-line tool for poking MOOS variable-value pairs and scoping on the before and after values of the poked variable(s) before exiting. See Benjamin (2010) and Benjamin et al. (n.d.).

pMarineViewer: A GUI-based tool primarily used for rending the paths of vehicles in two-dimensional (2D) space on a Geo display but also can be configured to poke the DB with variable-value pairs connected to buttons on the display. See Benjamin (2010) and Benjamin et al. (n.d.).

pEchoVar: A lightweight process that runs without user interaction for "echoing" specified variable-value pairs posted with a follow-on post having different variable name. See Benjamin (2010).

iMarineSim: A very simple single-vehicle simulator that updates vehicle state based on present actuator values. Runs locally in the MOOS community associated with the simulated vehicle, so, unlike uMVS, there is one iMarineSim process running per each vehicle.

pMarinePID: An application providing simple proportional–integral–derivative (PID) control for vehicle speed-thrust, heading-rudder, and depth-pitch. It is useful on platforms where there is no front-seat control capability, or when operating in simulation. It is typically run at 20–40 Hz.

pBasicContactMgr: A simple manager of vehicle contacts, capable of generating of user-configured range-dependent alerts, via posting of variable-value pairs to the MOOSDB. See Benjamin (2010).

uFunctionVis: A application for live rendering of objective functions produced by the IvP Helm behaviors. See Benjamin et al. (n.d.).

uProcessWatch: An application for monitoring the presence (connection) of a set of MOOS processes to a running MOOSDB. Status is summarized by a single published variable. See Benjamin (2010).

uTermCommand: A terminal-based tool for poking the DB with predefined variable-value pairs. The user can configure the tool to associate aliases to quickly poke the DB. See Benjamin (2010) and Benjamin et al. (n.d.).

uTimerScript: A MOOS application that will poke the MOOSDB with predefined variable-value pairs in a script that may repeat. Not unlike pScheduler, but it can do some additional things such as jump forward or pause in the script based on MOOS notifications. It may also schedule its events to occur at a random point in a fixed time interval. See Benjamin (2010) and Benjamin et al. (n.d.).

alogclip: A command-line tool for clipping a log file based on a start and an end time.

aloggrep: A command-line tool for filtering a log file keying on one or more MOOS variables or sources to keep.

aloghelm: A command-line postmission analysis tool for generating IvP Helm–related reports from a given log file.

alogrm: A command-line tool for filtering a log file keying on one or more MOOS variables or sources to remove.

alogscan: A command-line tool for generating statistical reports of a log file.

alogview A command-line postmission analysis tool for rendering vehicle position trajectories and time series data from a set of alog files.

A number of MOOS applications have been written to handle communications via acoustic modems (Schneider & Schmidt, 2010a; 2010b), including dynamically configurable message sets and encryption:

pAcommsHandler: A complete acoustic communications solution composed of (1) compact encoding using the Dynamic Compact Control Language (D-CCL), (2) time-varying priority buffering, (3) medium access control, and (4) support for the Woods Hole Oceanographic Institute (WHOI) Micro-Modem firmware.

iCommander: A GUI for composing DCCL messages (typically commands) for sending through an acoustic modem via pAcommsHandler.

pGeneralCodec: A stand-alone MOOS interface to DCCL for those users not needing the entire package offered by pAcommsHandler.

pCTDCodec: A special-purpose compressional encoder for conductivity–temperature–depth (CTD) data. This encoder uses delta-differencing encoding to make very small messages suitable for transmission through an acoustic modem.

pBTRCodec: An encoder for handling a beam-time record (BTR) from an acoustic array.

pREMUSCodec: A MOOS encoder/decoder for a subset of the WHOI/REMUS Compact Control Language (CCL) message set.

iMOOS2SQL: An interface between MOOS and the Google Earth interface for ocean vehicles (GEOV) utility.

3. IvP HELM AUTONOMY

3.1. Overview and Background

An autonomous helm is primarily an engine for decision making. The IvP Helm uses a behavior-based architecture to organize its decision making and is distinctive in the manner in which it resolves competition between behaviors, by performing multiobjective optimization on their collective output using a mathematical programming model called interval programming. Here the IvP Helm architecture is described along with the means for configuring it given a set of behaviors and a set of mission objectives.

3.1.1. The Influence of Brooks, Stallman, and Dantzig on the IvP Helm

The notion of a behavior-based architecture for implementing autonomy on a robot or unmanned vehicle is most often attributed to Rodney Brooks's subsumption architecture (Brooks, 1986). A key principle at the heart of Brooks's architecture, and arguably the primary reason that its appeal has endured, is the notion that autonomy systems can be built *incrementally*. Notably, Brooks's original publication predated the arrival of open-source software and the Free Software Foundation founded by Richard Stallman. Open-source software is not a prerequisite for building autonomy systems incrementally, but it has the capability of greatly accelerating that objective. The development of complex autonomy systems stands to significantly benefit if the set of developers at the table is large and diverse, even more so if they can be from different organizations with perhaps even the loosest of overlap in interest regarding how to use the collective end product.

As discussed in Section 1.4, a key issue in behavior-based autonomy has been the issue of action selection, and the IvP Helm is distinct in this regard with the use of multiobjective optimization and interval programming. The algorithm behind interval programming, as well as the term itself, was motivated by the mathematical programming model, linear programming, developed by George Dantzig (1948). The key idea in linear programming is the choice of the particular mathematical construct that comprises an instance of a linear programming problem—it has enough expressive flexibility to represent a huge class of practical problems, *and* the constructs can be effectively exploited by the simplex method to converge quickly even on very large problem instances. The constructs used in interval programming to represent behavior output (piecewise linear functions) were likewise chosen to have enough expressive flexibility to handle any current and future behavior and due to the opportunity to develop solution algorithms that exploit the piecewise linear constructs.

3.1.2. Traditional and Nontraditional Aspects of the IvP Behavior–Based Helm

The IvP Helm indeed takes its motivation from early notions of the behavior-based architecture but is also quite different in many regards. The notion of behavior independence to temper the growth of complexity in progressively larger systems is still a principle closely followed in the IvP Helm. Behaviors may certainly influence one another from one iteration to the next. However, within a single iteration, the output generated by a single behavior is not affected at all by what is generated by other behaviors in the same iteration. The only interbehavior "communication" realized within an iteration comes when the IvP solver reconciles the output of multiple behaviors. The independence of behaviors not only helps a single developer manage the growth of complexity, but it also limits the dependency between developers. A behavior author need not worry that a change in the implementation of another behavior by another author requires subsequent recoding of one's own behavior(s).

Certain aspects of behaviors in the IvP Helm may also be a departure from some notions traditionally associated (fairly or not) with behavior-based architectures:

- Behaviors have state. IvP behaviors are instances of a class with a fairly simple interface to the helm. Inside they may be arbitrarily complex, keep histories of observed sensor data, and may contain algorithms that could be considered "reactive" or "plan-based."
- Behaviors influence each other between iterations. The primary output of behaviors is their objective function, ranking the utility of candidate actions. IvP behaviors may also generate variable-value posts to the MOOSDB observable by behaviors on the next helm iteration. In this way they can explicitly influence other behaviors by triggering or suppressing their activation or even affecting the parameter configuration of other behaviors.
- Behaviors may accept externally generated plans. The input to a behavior can be anything represented by a

MOOS variable and perhaps generated by other MOOS processes outside the helm. It is allowable to have one or more planning engines running on the vehicle generating output consumed by one or more behaviors.

- Several instances of the same behavior are allowed. Behaviors generally accept a set of configuration parameters that allow them to be configured for quite different tasks or roles in the same helm and mission. Different waypoint behaviors, for example, can be configured for different components of a transit mission. Or different collision avoidance behaviors can be instantiated for different contacts.
- Behaviors can be run in a configurable sequence. The `condition` and `endflag` parameters defined for all behaviors allow for a sequence of behaviors to be readily configured into a larger mission plan.
- Behaviors rate actions over a coupled decision space. IvP functions generated by behaviors are defined over the Cartesian product of the set of vehicle decision variables. Objective functions for certain behaviors may be adequately expressed only over such a decision space. See, for example, the function produced by the AvoidCollision behavior later in this paper. This is distinct from the decoupled decision-making style proposed in Pirjanian (1998) and Rosenblatt (1997)—early advocates of multiobjective optimization in behavior-based action selection. There is indeed a computational cost associated with this approach. This is mitigated by the use of the interval programming (IvP) model and IvPBuild Toolbox for representing, building, and solving multiobjective optimization problems over functions with multidimension domains.

The autonomy in play on a vehicle during a particular mission is the product of two distinct efforts: (1) the development of vehicle behaviors and their algorithms and (2) mission planning via the configuration of behaviors and mode declarations. The former involves the writing of new source code, and the latter involves the editing of mission behavior files.

3.2. Inside the IvP Helm: A Look at the Helm Iterate Loop

Like other MOOS applications, the IvP Helm implements an `Iterate()` loop within which the basic function of the helm is executed. Components of the `Iterate()` loop, with respect to the behavior-based architecture, are described in this section. The basic flow, in five steps, is depicted in Figure 7. Descriptions of the five components follow.

Step 1—Reading mail and populating the information buffer: The first step of a helm iteration occurs outside the `Iterate()` loop. As depicted in Figure 6, a MOOS application will read its mail by executing its `OnNewMail()` function just prior to executing its `Iterate()` loop if there is any mail in its in-box. The helm parses mail to maintain its own information buffer, which is also a mapping of variables to values. This is done primarily for simplicity—to ensure that each behavior is acting on the same world state as represented by the information buffer. Each behavior has a pointer to the buffer and is able to query the current value of any variable in the buffer or get a list of variable-value changes since the previous iteration.

Step 2—Evaluation of mode declarations: Once the information buffer is updated with all incoming mail, the helm evaluates any mode declarations specified in the behavior file. Mode declarations are discussed in Section 3.3. In short, a mode is represented by a string variable that is reset on each iteration based on the evaluation of a set of logic expressions involving other variables in the buffer. The variable representing the mode declaration is then available to the behavior on the current iteration when it, for example, evaluates its `condition` parameters. A condition for behavior participating in the current iteration could therefore read something like `condition = (MODE==SURVEYING)`. The exact value of the variable `MODE` is set during this step of the `Iterate()` loop.

Step 3—Behavior participation: In the third step much of the work of the helm is realized by giving each behavior a chance to participate. Each behavior is queried sequentially—the helm contains no separate threads in this regard. The order in which behaviors are queried does not affect the output. This step contains two distinct parts for each behavior: (1) determination of whether the behavior will participate and (2) production of output if it is indeed participating on this iteration. Each behavior may produce two types of information as Figure 7 indicates. The first is an objective function (or "utility" function) in the form of an IvP function. The second kind of behavior output is a list of variable-value pairs to be posted by the helm to the MOOSDB at the end of the `Iterate()` loop. A behavior may produce both kinds of information, neither, or one or the other, on any given iteration.

Step 4—Behavior reconciliation: In the fourth step depicted in Figure 7, the IvP functions are collected by the IvP solver to produce a single decision over the helm's decision space. Each function is an IvP function—an objective function that maps each element of the helm's decision space to a utility value. In this case the functions are of a particular form—piecewise linearly defined. That is, each piece is an *interval* of the decision space with an associated linear function. Each function also has an associated weight, and the solver performs multiobjective optimization over the weighted sum of functions (in effect a single objective optimization at that point). The output is a single

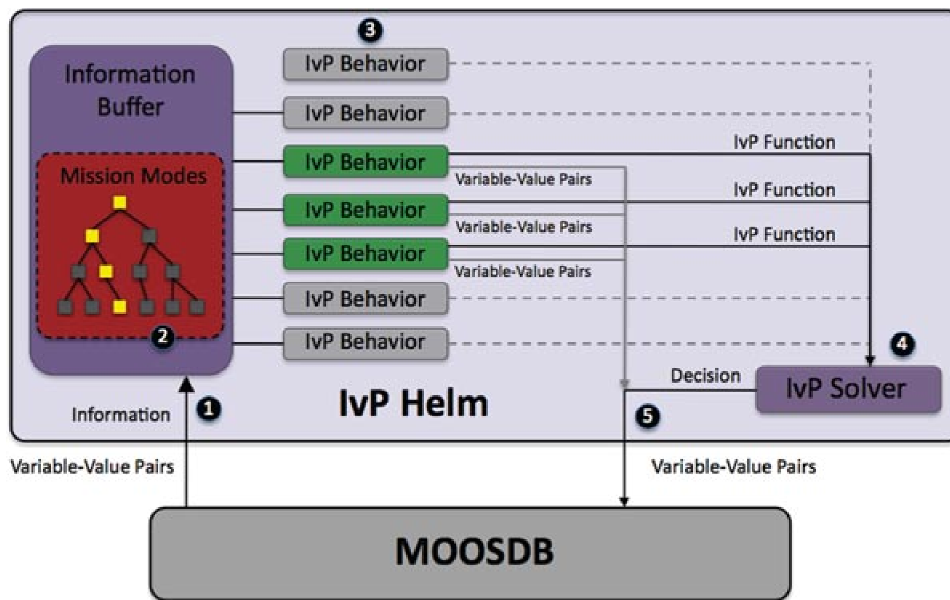

Figure 7. The pHelmIvP Iterate Loop: (1) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors. (2) If there were any mode declarations in the mission behavior file, they are evaluated at this step. (3) Each behavior is queried for its contribution and may produce an IvP function and a list of variable-value pairs to be posted to the MOOSDB at the end of the iteration. (4) The objective functions are resolved to produce an action, expressible as a set of variable-value pairs. (5) All variable-value pairs are published to the MOOSDB for other MOOS processes to consume.

optimal point in the decision space. For each decision variable, the helm produces another variable-value pair, such as `DESIRED_SPEED = 2.4` for publication to the MOOSDB.

Step 5—Publishing the results to the MOOSDB: In the last step, the helm simply publishes all variable-value pairs to the MOOSDB, some of which were produced directly by the behaviors and some of which were generated as output from the IvP solver. The helm employs a duplication filter that may prevent successive variable-value pairs from being posted if the value does not change. This filter is applied only to the variable-value pairs generated directly from the behaviors and not the variable-value pairs generated by the IvP solver that represent a decision in the helm's domain. For example, even if the decision about a vehicle's depth, represented by the variable `DESIRED_DEPTH`, produced by the helm were unchanged for 5 min of operation, it would be published on each iteration of the helm. To do otherwise could give the impression to consumers of the variable that the variable is "stale," which could trigger an unwanted override of the helm out of a concern for safety.

3.3. Hierarchical Mode Declarations

Hierarchical mode declarations (HMDs) are an optional feature of the IvP Helm for organizing the behavior activations according to declared mission modes. Modes and submodes can be declared, in line with a mission planner's own concept of mission evolution, and behaviors can be associated with the declared modes. In more complex missions, it can facilitate mission planning (in terms of less time and better detection of human errors), and it can facilitate the understanding of exactly what is happening in the helm—during the mission execution and in postanalysis.

3.3.1. The Evolution of Autonomy Modes

A trend of unmanned vehicle usage can be characterized as being increasingly less of the shorter, scripted variety and to be increasingly more of the longer, adaptive mission variety. A typical mission in our own lab 5 years ago would contain a certain set of tasks, typically waypoints and ultimately a rendezvous point for recovering the vehicle. Data acquired during deployment were off-loaded and analyzed later in the laboratory. What has changed? The simultaneous maturation of acoustic communications, onboard sensor processing, and longer vehicle battery life has dramatically changed the nature of mission configurations. The vehicle is expected both to adapt to the phenomena it senses and processes onboard and to adapt its operation given field-control commands received via acoustic,

radio, or satellite communications. Multivehicle collaborative missions are also increasingly viable due to lower vehicle costs and mature ACOMMS capabilities. In such cases a vehicle is adapting not only to sensed phenomena and field commands but also to information from collaborating vehicles.

Our missions have evolved from having a finite set of fixed tasks to be composed instead of a set of modes, an initial mode when launched, an understanding of what brings us from one mode to another, and what behaviors are in play in each mode. Modes may be entered and exited any number of times, in exact sequences unknown at launch time, depending on what the vehicle senses and how they are commanded in the field.

3.3.2. Syntax of Hierarchical Mode Declarations

An example is provided showing of the use of HMDs with an example simple mission. This mission is referred to as the Bravo mission and can be found alongside the Alpha mission in the set of example missions distributed with the MOOS-IvP public domain software. The modes are explicitly declared in the Bravo behavior file to form the hierarchy shown in Figure 8.

The hierarchy in Figure 8 is formed by the mode declaration constructs on the left-hand side, taken as an excerpt from the `bravo.bhv` file. After the mode declarations are read when the helm is initially launched, the hierarchy remains static thereafter. The hierarchy is associated with a particular MOOS variable, in this case the variable `MODE`. Although the hierarchy remains static, the mode is reevaluated at the outset of each helm iteration based on the conditions associated with nodes in the hierarchy. The mode evaluation is represented as a string in the variable `MODE`. As shown in Figure 8, the variable is the concatenation of the names of all the nodes. The mode evaluation begins sequentially through each of the blocks. At the outset the value of the variable `MODE` is reset to the empty string. After the first block in Figure 8, `MODE` will be set to either `"Active"` or `"Inactive."` When the second block is evaluated, the condition `"MODE=Active"` is evaluate based on how `MODE` was set in the first block. For this reason, mode declarations of children need to be listed after the declarations of parents in the behavior file.

Once the mode is evaluated, at the outset of the helm iteration, it is available for use in the run conditions of the behaviors (described below) via a string-matching relation that matches when one side matches exactly one of the components in the other side's colon-separated list of strings. Thus `"Active" == "Active:Returning,"` and `"Returning" == "Active:Returning."` This is to allow a behavior to be easily associated with an internal node regardless of its children. For example, if a collision-avoidance behavior were to be added to the Bravo mission, it could be associated with the `"Active"` mode rather than explicitly naming all the submodes of the `"Active"` mode.

3.4. Behavior Participation in the IvP Helm

The primary work of the helm comes when the behaviors participate at each round of the helm `Iterate()`

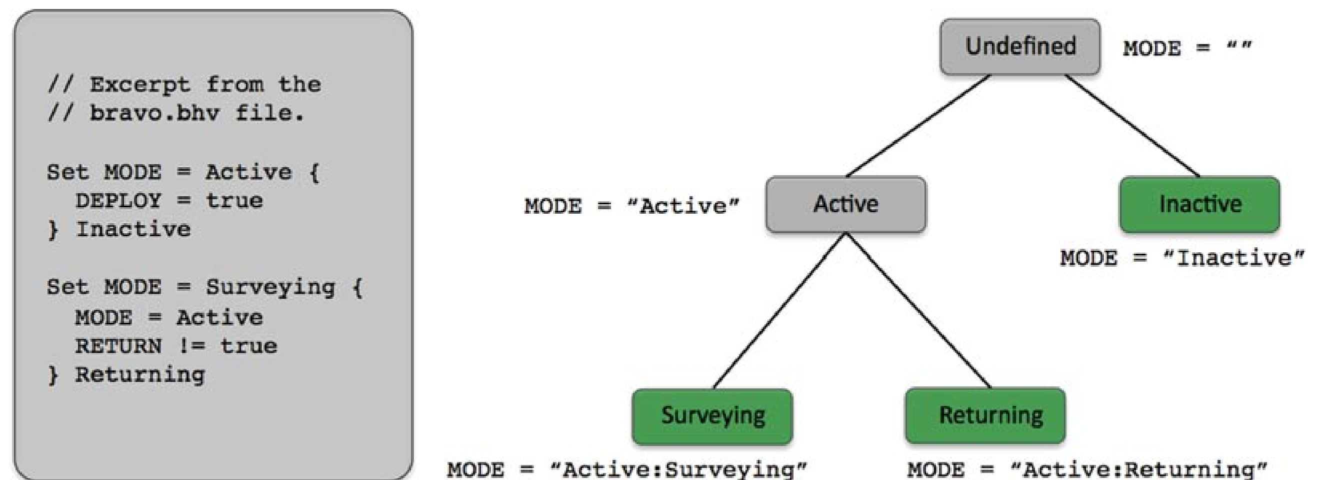

Figure 8. Hierarchical modes for the Bravo mission: The vehicle will always be in one of the modes represented by a leaf node. A behavior may be associated with any node in the tree. If a behavior is associated with an internal node, it is also associated with all its children.

loop, by producing IvP functions, posting variable-value pairs to MOOS, or both. As depicted in Figure 7, once the mode has been reevaluated taking into consideration newly received mail, it is time for the relevant behaviors to participate.

3.4.1. Behavior Conditions

On any single iteration a behavior may participate by generating an objective function to influence the helm's output over its decision space. Not all behaviors participate in this regard, and the primary criterion for participation is whether it has met each of its "run conditions." These are the conditions laid out in the behavior file of the form

```
condition = <logic-expression>.
```

Conditions are built from simple relational expressions, the comparison of MOOS variables to specified literal values, or the comparison of MOOS variables to one another. Conditions may also involve Boolean logic combinations of relation expressions. A behavior may base its conditions on any MOOS variable, such as

```
condition = (DEPLOY=true) and (STATION != true).
```

A run condition may also be expressed in terms of a helm mode, such as

```
condition = (MODE == LOITERING).
```

All MOOS variables involved in run condition expressions are automatically subscribed for by the helm to the MOOSDB.

3.4.2. Behavior Run States

On any given helm iteration a behavior may be in one of four states depicted in Figure 9:

- Idle: A behavior is `idle` if it is not `complete` and it has not met its run conditions as described in Section 3.4.1. The helm will invoke an idle behavior's `onIdleState()` function.
- Running: A behavior is `running` if it has met its run conditions and it is not `complete`. The helm will invoke a running behavior's `onRunState()` function, thereby giving the behavior an opportunity to contribute an objective function.
- Active: A behavior is `active` if it is running and it did indeed produce an objective function when prompted. There are a number of reasons that a running behavior may not be active. For example, a collision avoidance behavior may opt to not produce an objective function when the contact is sufficiently far away.
- Complete: A behavior is `complete` when the behavior itself determines that it is complete. It is up to the behavior author to implement this, and some behaviors may never complete. The function `setComplete()` is defined generally at the behavior superclass level, for calling by a behavior author. This provides some standard steps to be taken upon completion, such as posting of `endflags`, described in Section 3.4.3. Once a behavior is in the `complete` state, it remains in that state permanently. All behaviors have a `DURATION` parameter defined to allow the behavior to be configured to time out if desired. When a time-out occurs, the behavior state will be set to `complete`.

3.4.3. Behavior Flags and Behavior Messages

Behaviors may produce a set of messages, i.e., variable-value pairs, on any given iteration (see Figure 7). These messages can be critical for coordinating behaviors with each other and to other MOOS processes. They can also be invaluable for monitoring and debugging behaviors configured for particular missions. Behaviors do not post messages to the MOOSDB; they request the helm to post messages on their behalf. The helm collects these requests and publishes them to the MOOSDB at the end of the `Iterate()` loop.

There is a standard method, configurable in the behavior file, for posting messages based on the run state of the behavior. These are referred to as behavior flags, and there are five types: (1) `endflag`, (2) `idleflag`, (3) `runflag`, (4) `activeflag`, and (5) `inactiveflag`. The variable-value pairs representing each flag are set in the behavior file for the corresponding behavior:

- `endflag`: An `endflag` is posted once when or if the behavior enters the `complete` state. The variable-value pair representing the endflag is given in the `endflag` parameter in the behavior file. Multiple endflags may be configured for a behavior. By default, when a behavior is completed and has posted its endflag, it does not participate further. Its instance is destroyed and removed

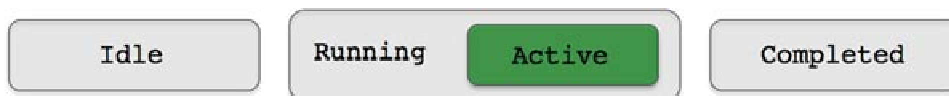

Figure 9. Behavior states: A behavior may be in one of these four states at any given iteration of the helm `Iterate()` loop. The state is determined by examination of MOOS variables stored locally in the helm's information buffer.

from the helm. A behavior may, however, be configured to survive completion by setting the `perpetual` parameter to be `true`.

- `idleflag`: An `idleflag` is posted on each iteration of the helm when the behavior is determined to be in the `idle` state. The variable-value pair representing the idleflag is given in the `idleflag` parameter in the behavior file. Multiple idleflags may be configured for a behavior.
- `runflag`: A `runflag` is posted on each iteration of the helm when the behavior is determined to be in the `running` state, regardless of whether it is further determined to be active or not. A `runflag` is posted exactly when an `idleflag` is not. The variable-value pair representing the runflag is given in the `runflag` parameter in the behavior file. Multiple runflags may be configured for a behavior.
- `activeflag`: An `activeflag` is posted on each iteration of the helm when the behavior is determined to be in the `active` state. The variable-value pair representing the activeflag is given in the `activeflag` parameter in the behavior file. Multiple activeflags may be configured for a behavior.
- `inactiveflag`: An `inactiveflag` is posted on each iteration of the helm when the behavior is determined to be not in the `active` state. The variable-value pair representing the inactiveflag is given in the `inactiveflag` parameter in the behavior file. Multiple inactiveflags may be configured for a behavior.

A `runflag` is meant to "complement" an `idleflag`, by posting exactly when the other one does not; similarly with the `inactiveflag` and `activeflag`. The situation is shown in Figure 10.

Behavior authors may implement their behaviors to post other messages as they see fit. For example, the waypoint behavior publishes a status variable, `WPT_STATUS`, with a status message similar to `"vname=alpha,index=0,dist=124,eta=62,"` indicating the name of the vehicle, the index of the next point in the list of waypoints, the distance to that waypoint, and the estimated time of arrival.

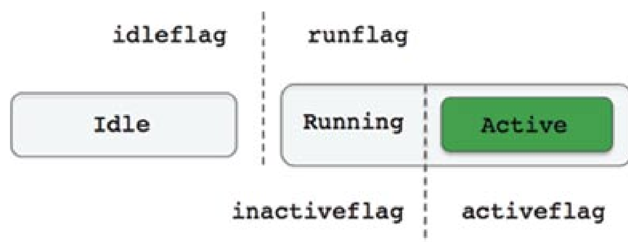

Figure 10. Behavior flags: The four behavior flags `idleflag`, `runflag`, `activeflag`, and `inactiveflag` are posted depending on the behavior state and can be considered complementary in the manner indicated.

3.5. Behavior Reconciliation in the IvP Helm—Multiobjective Optimization

A unique aspect of the IvP Helm is the manner in which it reconciles the output of behaviors when they are competing for influence of the helm decision.

3.5.1. IvP Functions

IvP functions are produced by behaviors to influence the decision produced by the helm on the current iteration (see Figure 7). The decision is typically composed of the desired heading, speed, and depth, but the helm decision space could be composed of any arbitrary configuration. Some points about IvP functions are as follows:

- IvP functions are piecewise linearly defined. Each piece is defined by an interval over some subset of the decision space, and there is a linear function associated with each piece (see Figure 11).
- IvP functions are an *approximation* of an underlying function. The linear function for a single piece is the best linear approximation of the underlying function for the portion of the domain covered by that piece.
- IvP domains are discrete with an upper and lower bound for each variable, so an IvP function *may* achieve zero error in approximating an underlying function by associating a piece with each point in the domain. Behaviors seldom need to do so in practice, however.
- The Ivp function construct and IvP solver are generalizable to N dimensions.
- The pieces in IvP functions need not be uniform size or shape. More pieces can be dedicated to parts of the domain that are harder to approximate with linear functions.
- IvP functions need only be defined over a subset of the domain. Behaviors are not affected if the helm is configured for additional variables that a behavior may not care about. It is allowable, for example, to have a behavior that produces IvP functions solely over the vehicle depth, even though the helm may be producing decisions over heading, speed, and depth.

IvP functions are produced by behaviors using the IvP Build Toolbox—a set of tools for creating IvP functions based on any underlying function defined over an IvP domain. Many, if not all, of the behaviors in this document make use of this toolbox, and authors of new behaviors have this at their disposal. A primary component of writing a new behavior is the development of the "underlying function," the function approximated by an IvP function with the help of the toolbox. The underlying function is a correlation between all candidate helm decisions, e.g., heading, speed, and depth choices, to a utility value from the perspective of what the behavior is trying to achieve.

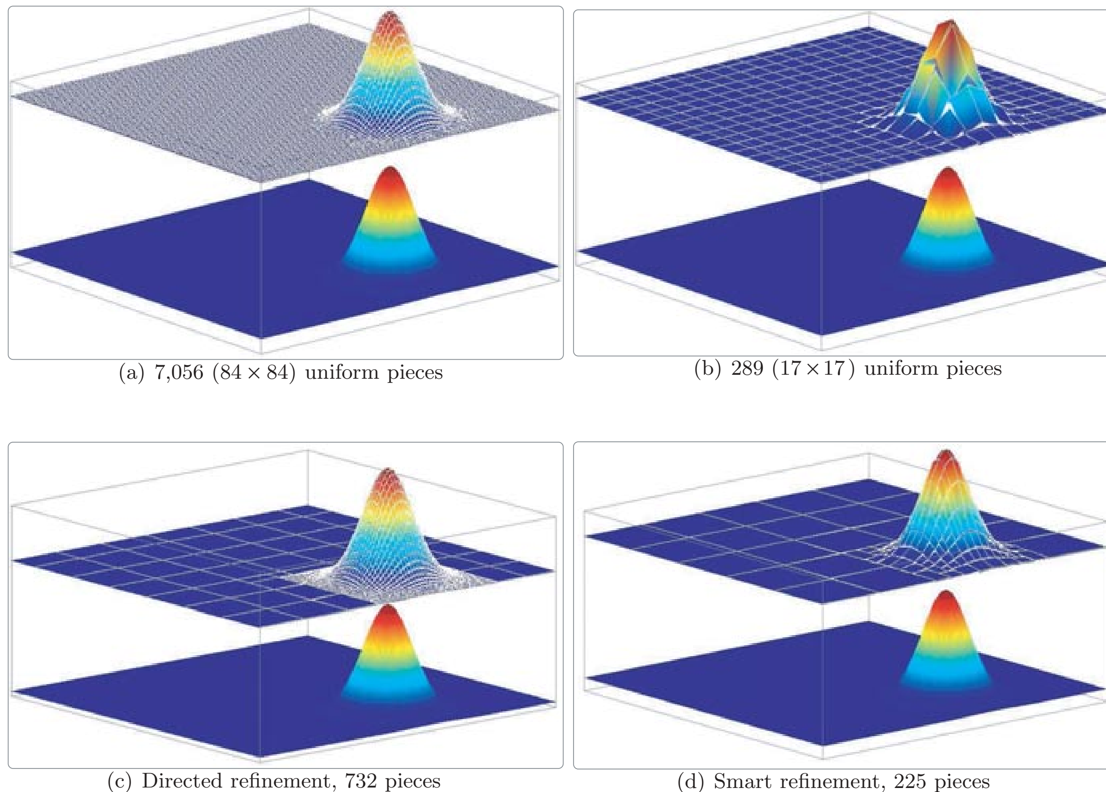

Figure 11. A rendering of four different IvP functions approximating the same underlying function: The function in (a) uses a uniform distribution of 7,056 pieces. The function in (b) uses a uniform distribution of 289 pieces. The function in (c) was created by first building a uniform distribution of 49 pieces and then focusing the refinement on a subdomain of the function. This is called directed refinement in the IvP Build Toolbox. The function in (d) was created by first building a uniform function of 25 pieces and repeatedly refining the function based on which pieces were noted to have a poor fit to the underlying function. This is termed smart refinement in the IvP Build Toolbox.

3.5.2. The IvP Build Toolbox

The IvP toolbox is a set of tools (a C++ library) for building IvP functions. It is typically utilized by behavior authors in a sequence of library calls within a behavior's (C++) implementation. There are two sets of tools—the *reflector* tools for building IvP functions in N dimensions and the *ZAIC* tools for building IvP functions in one dimension as a special case. The reflector tools work by making available a function to be approximated by an IvP function. The tools simply need this function for sampling. Consider the Gaussian function rendered in Figure 12.

The x and y variables, each with a range of $[-250, 250]$, are discrete, taking on integer values. The domain therefore contains $501^2 = 251{,}001$ points or possible decisions. The IvP Build Toolbox can generate an IvP function approximating this function over this domain by using a uniform piece size, as rendered in Figures 11(a) and 11(b). The difference in these two figures is only the size of the piece. More pieces [Figure 11(a)] results in a more accurate approximation of the underlying function but takes longer to generate and creates further work for the IvP solver when the functions are combined. IvP functions need not use uniformly sized pieces.

By using the *directed refinement* option in the IvP Build Toolbox, an initially uniform IvP function can be further refined with more pieces over a subdomain directed by the caller, with smaller uniform pieces of the caller's choosing. This is rendered in Figure 11(c). Using this tool requires the caller to have some idea where, in the subdomain, further refinement is needed or desired. Often a behavior author indeed has this insight. For example, if one of the domain variables is vehicle heading, it may be good to have a fine refinement in the neighborhood of heading values close to the vehicle's current heading.

In other situations, insight into where further refinement is needed may not be available to the caller. In these cases, using the *smart refinement* option of the IvP Build

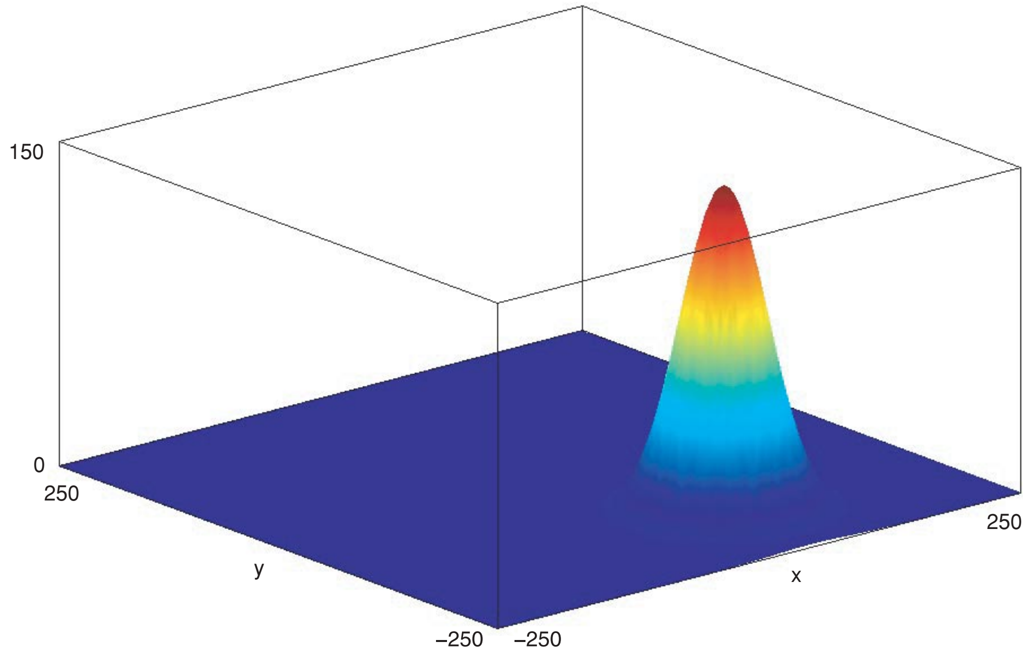

Figure 12. A rendering of the function $f(x, y) = Ae^{-\{(x=x_0)^2+(y=y_0)^2/2\sigma^2\}}$, where A = `range` = 150, σ = `sigma` = 32.4, x_0 = `xcent` = 50, and y_0 = `ycent` = −150. The domain here for x and y ranges from −250 to 250.

Toolbox, an initially uniform IvP function may be further refined by asking the toolbox to automatically "grade" the pieces as they are being created. The grading is in terms of how accurate the linear fit is between the piece's linear function and the underlying function over the subdomain for that piece. A priority queue is maintained based on the grades, and pieces for which poor fits are noted are automatically refined further, up to a maximum piece limit chosen by the caller. This is rendered in Figure 11(d).

The reflector tools work similarly in N dimensions and on multimodal functions. The only requirement for using the reflector tool is to provide it with access to the underlying function. Because the tool repetitively samples this function, a central challenge to the user of the toolbox is to develop a fast implementation of the function. In terms of the time consumed in generating IvP functions with the reflector tool, the sampling of the underlying function is typically the longest part of the process.

3.5.3. The IvP Solver and Behavior Priority Weights

The IvP solver collects a set of weighted IvP functions produced by each of the behaviors and finds a point in the decision space that optimizes the weighted combination. If each IvP objective function is represented by $f_i(\boldsymbol{x})$, and the weight of each function is given by w_i, the solution to a problem with k functions is given by

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x}} \sum_{i=0}^{k-1} w_i f_i(\boldsymbol{x}).$$

The algorithm is described in detail in Benjamin (2004) but is summarized in the following few points.

- *The search tree*: The structure of the search algorithm is branch and bound. The search tree is composed of an IvP function at each layer, and the nodes at each layer are composed of the individual pieces from the function at that layer. A leaf node represents a single piece from each function. A node in the tree is realizable if the piece from that node and its ancestors intersect, i.e., share common points in the decision space.
- *Global optimality*: Each point in the decision space is in exactly one piece in each IvP function and is thus in exactly one leaf node of the search tree. If the search tree is expanded fully, or pruned properly (only when the pruned out subtree does not contain the optimal solution), then the search is guaranteed to produce the globally optimal solution. The search algorithm employed by the IvP solver does indeed start with a fully expanded tree and utilizes proper pruning to guarantee global optimality. The algorithm does allow for a parameter for guaranteed limited back-off from the global

optimality—a quicker solution with a guarantee of being within a fixed percent of global optima. This option is not exposed in the configuration of the IvP Helm, which always finds the global optimum because this stage of computation is very fast in practice.

- *Initial solution*: A key factor of an effective branch-and-bound algorithm is seeding the search with a decent initial solution. In the IvP Helm, the initial solution used is the solution (typically heading, speed, depth) generated on the previous helm iteration. In practice this appears to provide a speedup by about a factor of two.

In cases in which there is a "tie" between optimal decisions, the solution generated by the solver is nondeterministic. When the solver is used in the helm, the nondeterminism is mitigated somewhat by the fact that the solution is seeded with the output of the previous helm iteration as discussed above. In other words, all things being equal, the helm will default to producing a decision that matches its previous decision.

The setting of function priority weights occurs in one of three manners. First, many behaviors are designed with a *policy* for setting their own weights. For example, in a collision avoidance behavior the weight varies between zero and a maximum weight depending on the relative position of the two vehicles. Second, weights are influenced by initial behavior configuration parameters regarding the priority weight policy. These are set during a premission planning phase. Finally, weights may be affected at run time via the dynamic reconfiguration of behavior parameters to values different from those set at the time of mission launch. Such reconfiguration may be the result of a field-control message received from a remote operator or another platform or the result of another onboard process outside the helm. In practice, users often make good use of simulation tools to confirm that parameter configurations and behavior weight-setting policies are in line with their expectations for the mission.

4. IvP HELM BEHAVIORS

Helm behaviors derive part of their function from inheriting properties from a base class behavior implementation, and their unique capabilities are an extension of the base capability. The uniform base capability allows for mission configurations to be constructed in a simple predictable manner. Here we discuss (a) the base capabilities of IvP behaviors, (b) how behaviors are handled generally by the helm in each iteration, (c) the hooks for creating a new third-party behavior, (d) an overview of standard behaviors that are distributed with the helm in the open domain, and (e) a more detailed look at a few representative behaviors.

4.1. Brief Overview

Behaviors are implemented as C++ classes with the helm having one or more instances at run time, each with a unique descriptor. The properties and implemented functions of a particular behavior are partly derived from the `IvPBehavior` superclass, shown in Figure 13. The is-a relationship of a derived class provides a form of code reuse as well as a common interface for constructing mission files with behaviors.

The `IvPBehavior` class provides three virtual functions that are typically overloaded in a particular behavior implementation:

- The `setParam()` function: Parameter-value pairs are handled to configure a behavior's unique properties distinct from its superclass.

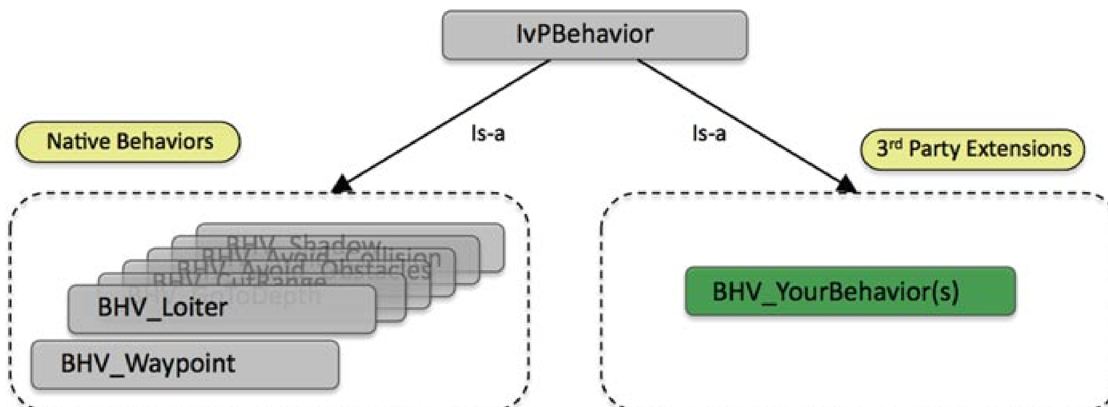

Figure 13. Behavior inheritance: Behaviors are derived from the `IvPBehavior` superclass. The native behaviors are the behaviors distributed with the helm. New behaviors also need to be a subclass of the IvPBehavior class to work with the helm. Certain virtual functions invoked by the helm may be optionally but typically overloaded in all new behaviors. Other private functions may be invoked within a behavior function as a way of facilitating common tasks involved in implementing a behavior.

- The `onRunState()` function: The primary function of a behavior implementation, performed when the behavior has met its conditions for running, with the output being an objective function and a possibly empty set of variable-value pairs for posting to the MOOSDB.
- The `onIdleState()` function: What the behavior does when it has not met its run conditions. It may involve updating internal state history, generation of variable-value pairs for posting to the MOOSDB, or absolutely nothing at all.

This section discusses the properties of the `IvPBehavior` superclass that an author of a third-party behavior needs to be aware of in implementing new behaviors. It is also relevant material for users of the native behaviors as it details general properties.

4.2. Parameters Common to All IvP Behaviors

A behavior has a standard set of parameters defined at the `IvPBehavior` level as well as unique parameters defined at the subclass level. By configuring a behavior during mission planning, the setting of parameters is the primary venue for affecting the overall autonomy behavior in a vehicle. Parameters are set in the behavior file but can also be dynamically altered once the mission has commenced. A parameter is set with a single line of the form

```
parameter = value.
```

In this section, the parameters defined at the superclass level and available to all behaviors are discussed. Each new behavior typically augments these parameters with new parameters unique to the behavior.

4.2.1. A Summary of the Full Set of General Behavior Parameters

The following parameters are defined for all behaviors at the superclass level.

`NAME`: The name of the behavior must be unique between all behavior instances.

`PRIORITY`: The priority weight of the produced objective function. The default value is 100. A behavior may also be implemented to determine its own priority weight, depending on information about the world.

`DURATION`: The time in seconds that the behavior will remain running before declaring completion. If no duration value is provided, the behavior will never time out. The clock starts ticking once the behavior satisfies its run conditions (becoming nonidle) the first time. *Should the behavior switch between running and idle states, the clock keeps ticking even during the idle periods.*

`DURATION_STATUS`: If the `DURATION` parameter is set, the remaining duration time, in seconds, can be posted by naming a `DURATION_STATUS` variable. This variable will be updated and posted only when the behavior is in the running state.

`DURATION_RESET`: This parameter takes a variable-pair such as `MY_RESET=true`. If the `DURATION` parameter is set, the duration clock is reset when the variable is posted to the MOOSDB with the specified value. Each time such a post is noted, the duration clock is reset.

`POST_MAPPING`: This parameter takes a comma-separated pair such as `WPT_STAT, WAYPT_STATUS` in which the left-hand value is a variable normally posted by the behavior and the right-hand value is an alternative variable name to be used. There is no error checking to ensure that the left-hand value names a variable actually posted by the behavior. Transitive relationships are not respected. For example, if the two remappings are declared, `FOO,BAR` and `BAR,CAR`, `FOO` will be posted as `BAR`, not `CAR`. To disable the normal posting of a variable `FOO`, use `POST_MAPPING = FOO,SILENT`.

`DURATION_IDLE_DECAY`: If this parameter is `false` the duration clock is paused when the vehicle is in the "idle" state. The default value is `true`.

`CONDITION`: This parameter specifies a condition that must be met for the behavior to be active. Conditions are checked for each behavior at the beginning of each control loop iteration. Conditions are based on current MOOS variables, such as `STATE = normal` or $(K \leq 4)$. More than one condition may be provided, as a convenience, treated collectively as a single conjunctive condition. The helm automatically subscribes for any condition variables.

`RUNFLAG`: This parameter specifies a variable and a value to be posted when the behavior has met all its conditions for being in the *running* state. It is an equal-separated pair such as `TRANSITING`=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors.

`IDLEFLAG`: This parameter specifies a variable and a value to be posted when the behavior is in the *idle* state. It is an equal-separated pair such as `WAITING`=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors.

`ACTIVEFlAG`: This parameter specifies a variable and a value to be posted when the behavior is in the *active* state. It is an equal-separated pair such as `TRANSITING`=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors.

`INACTIVEFlAG`: This parameter specifies a variable and a value to be posted when the behavior is *not* in the *active* state. It is an equal-separated pair such as `OUT_OF_RANGE`=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors.

`ENDFLAG`: This parameter specifies a variable and a value to be posted when the behavior has set the `completed`

state variable to be true. The circumstances causing completion are unique to the individual behavior. However, if the behavior has a `DURATION` specified, the `completed` flag is set to true when the duration is exceeded. The value of this parameter is an equal-separated pair such as `ARRIVED_HOME`=true. Once the completed flag is set to true for a behavior, it remains inactive thereafter, regardless of future events, barring a complete helm restart.

`UPDATES`: This parameter specifies a variable from which updates to behavior configuration parameters are read from after the behavior has been initially instantiated and configured at the helm startup time. Any parameter and value pair that would have been legal at startup time is legal at run time. This is one of the primary hooks to the helm for mission control—the other being the behavior conditions described above.

`NOSTARVE`: The `NOSTARVE` parameter allows a behavior to assert a maximum staleness for one or more MOOS variables, i.e., the time since the variable was last updated. The syntax for this parameter is a comma-separated list `"variable, ..., variable, value,"` where the last component in the list is the time value given in seconds.

`PERPETUAL`: Setting the `perpetual` parameter to `true` allows the behavior to continue to run even after it has completed and posted its end flags. The parameter value is not case sensitive, and the only two legal values are `true` and `false`.

4.2.2. Altering Behavior Parameters Dynamically with the UPDATES Parameter

The parameters of a behavior can be made to allow dynamic modifications—after the helm has been launched and is executing the initial mission in the behavior file. The modifications come in a single MOOS variable specified by the parameter `UPDATES`. For example, consider the simple waypoint behavior configuration below in Listing 1. The return point is the (0,0) point in local coordinates, and return speed is 2.0 m/s. When the conditions are met, this is what will be executed.

Listing 1—An example behavior configuration using the `UPDATES` parameter:

```
 0  Behavior = BHV_Waypoint
 1  {
 2    name      = WAYPT_RETURN
 3    priority  = 100
 4    speed     = 2.0
 5    radius    = 8.0
 6    points    = 0,0
 7    UPDATES   = RETURN_UPDATES
 8    condition = RETURN = true
 9    condition = DEPLOY = true
10  }
```

If, during the course of events, a different return point or speed is desired, this behavior can be altered dynamically by writing to the variable specified by the `UPDATES` parameter, in this case the variable `RETURN_UPDATES` (line 7 in Listing 1). The syntax for this variable is of the form

```
parameter = value,   # ... #, parameter = value.
```

White space is ignored. The # character is treated as special for parsing the line into separate parameter-value pairs. It cannot be part of a parameter component or value component. For example, the return point and speed for this behavior could be altered by any other MOOS process that writes to the MOOS variable:

```
RETURN_UPDATES = points = (50,50) # speed = 1.5.
```

Each parameter-value pair is passed to the same parameter setting routines used by the behavior on initialization. The only difference is that an erroneous parameter-value pair will simply be ignored as opposed to halting the helm as done on startup. If a faulty parameter-value pair is encountered, a warning will be written to the variable `BHV_WARNING`. For example,

```
BHV_WARNING = "Faulty update for behavior:\
  WAYPT_RETURN. Bad parameter(s): speed."
```

Note that a check for parameter updates is made at the outset of the helm iteration loop for a behavior with the call `checkUpdates()`. Any updates received by the helm on the current iteration will be applied prior to behavior execution and in effect for the current iteration.

4.3. Behavior Functions Invoked by the Helm

The `IvPBehavior` superclass implements a number of functions invoked by the helm on each iteration. Two of these functions are overloadable as described previously—the `onRunState()` and `onIdleState()` functions. The basic flow of calls to a behavior from the helm is shown in Figure 14. They are discussed in more detail later in the section, but the idea is to execute certain behavior functions based on the *activity state*, which may be one of the four states depicted. An *idle* behavior is one that has not met its conditions for running. A *completed* behavior is one that has reached its objectives or exceeded its duration. A *running* behavior is one that has not yet completed and has met its run conditions but may still opt not to produce any output. An *active* behavior is one that is running and is producing output in the form of an objective function.

The types of functions defined at the superclass level fall into one of the three categories below, only the first two of which are shown in Figure 14:

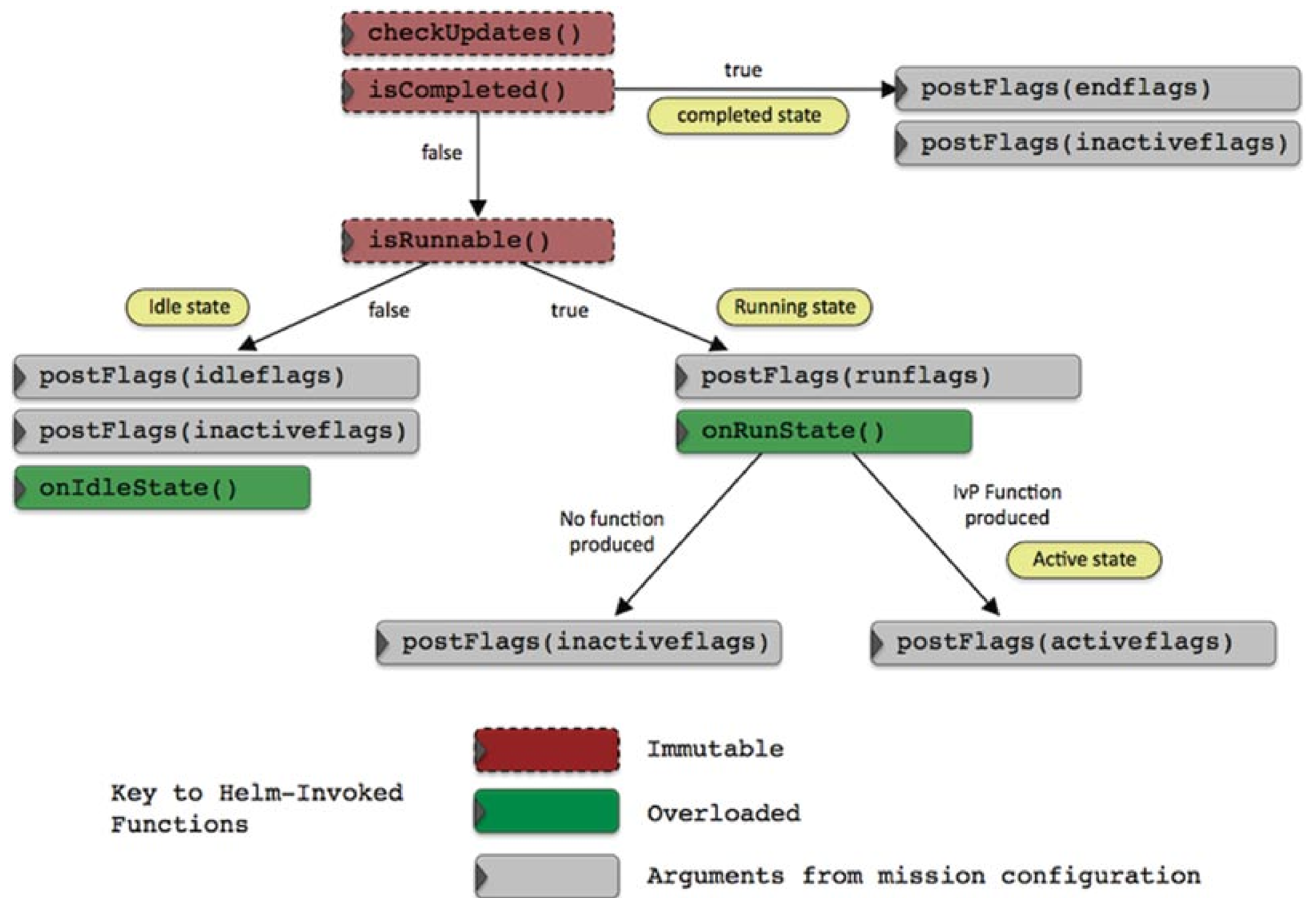

Figure 14. Behavior function calls by the helm: The helm invokes a sequence of functions on each behavior on each iteration of the helm. The sequence of calls is dependent on what the behavior returns and reflects the behavior's activity state. Certain functions are immutable and cannot be overloaded by a behavior author. Two key functions, `onRunState()` and `onIdleState()`, can indeed be overloaded as the usual hook for an author to provide the implementation of a behavior. The `postFlags` function is also immutable, but the parameters (flags) are provided in the helm configuration (`*.bhv`) file.

- Helm-invoked immutable functions—functions invoked by the helm on each iteration that the author of a new behavior may not reimplement.
- Helm-invoked overloadable functions—functions invoked by the helm that an author of a new behavior typically reimplements or overloads.
- User-invoked functions—functions invoked within a behavior implementation.

The user-invoked functions are utilities for common operations typically invoked within the implementation of the `onRunState()` and `onIdleState()` functions written by the behavior author. Their discussion is beyond the scope of this document, but descriptions may be found in Benjamin, Newman, Schmidt, and Leonard (2010).

4.3.1. Helm-Invoked Immutable Functions

These functions, implemented in the `IvPBehavior` superclass, are called by the helm but are *not* defined as virtual functions, which means that attempts to overload them in a new behavior implementation will be ignored. See Figure 14 regarding the sequence of these function calls.

`void checkUpdates()`: This function is called first on each iteration to handle requested dynamic changes in the behavior configuration. This needs to be the very first function applied to a behavior on the helm iteration so any requested changes to the behavior parameters may be applied on the present iteration. See Section 4.2.2 for more on dynamic behavior configuration with the `UPDATES` parameter.

`bool isComplete()`: This function simply returns a Boolean indicating whether the behavior was put into the *complete* state during a prior iteration.

`bool isRunnable()`: Determines whether a behavior is in the *running* state. Within this function call four things are checked: (a) if the duration is set, the duration time remaining is checked for timeout, (b) variables that are monitored for staleness are checked against, (c) the run conditions must be met, and (d) the behavior's decision domain (IvP domain) is a proper subset of the helm's configured IvP domain.

`void postFlags(string flag_type)`: This function will post flags depending on whether the value of `flag_type` is set to `"idleflags,"` `"runflags,"` `"activeflags,"` `"inactiveflags,"` or `"endflags."` Although this function is immutable, not overloadable by subclass implementations, its effect is indeed mutable because the flags are specified in the mission configuration `*.bhv` file.

4.3.2. Helm-Invoked Overloaded Functions

Helm-invoked overloaded functions are invoked by the helm but are defined as virtual functions so that a behavior author may overload them. Typically the bulk of writing a new behavior resides in implementing these functions.

`IvPFunction* onRunState()`: The `onRunState()` function is called by the helm when the behavior is deemed to be in the *running* state (Figure 14). The bulk of the work in implementing a new behavior is in this function implementation.

`void onIdleState()`: This function is called by the helm when the behavior is deemed to be in the *idle* state (Figure 14). Many behaviors are implemented with this function left undefined, but it is a useful hook to have in certain situations. For example, the behavior may post information to the MOOSDB about issues it is monitoring while in the idle state, or it could post information to the MOOSDB that may contribute to the behavior entering the running state on the next iteration.

`bool setParam(string, string)`: This function is called by the helm when the behavior is first instantiated with the set of parameter and parameter values provided in the behavior file. It is also called by the helm within the `checkUpdates()` function to apply parameter updates dynamically.

4.4. IvP Helm Behaviors in the Public Domain

Below is a brief description of 16 commonly used behaviors in the IvP Helm for the marine vehicle domain. This is followed by a longer description of a few behaviors chosen for illustration.

The AvoidCollision behavior: The AvoidCollision behavior will maneuver the vehicle to avoid a collision with a given contact. The generated objective function is based on the calculated closest-point-of-approach (CPA) for a given contact and candidate maneuvers. The safe-distance tolerance and policy for priority based on range is provided in the mission configuration.

The AvoidObstacles behavior: The AvoidObstacles behavior will maneuver the vehicle to avoid obstacles as known locations, expressed as one or more convex polygons. The safe-distance tolerance and policy for priority based on range is provided in the mission configuration.

The ConstantDepth behavior: This behavior will drive the vehicle at a specified depth. It merely expresses a preference for a particular depth, or range of depths, in terms of an IvP objective function. If other active behaviors also have a depth preference, coordination/compromise will take place through the multiobjective optimization process.

The ConstantSpeed behavior: This behavior will drive the vehicle at a specified speed. It merely expresses a preference for a particular speed, or range of speeds, in terms of an IvP objective function.

The ConstantHeading behavior: This behavior will drive the vehicle at a specified heading. It merely expresses a preference for a particular heading, or range of headings, in terms of an IvP objective function.

The CutRange behavior: The cut-range behavior will maneuver the vehicle to reduce the range between itself and a given contact. The generated objective function is based either on the calculated CPA for a given contact and candidate maneuver or purely on a greedy approach of heading toward the contact's present position. The policies may be combined (weighted) by the user in mission configuration. This behavior also has the ability to extrapolate the other vehicle's position from prior reports when contact reports are received intermittently.

The GoToDepth behavior: This behavior will drive the vehicle to a sequence of specified depths and duration at each depth. The duration is specified in seconds and reflects the time at depth after the vehicle has first achieved that depth, where achieving depth is defined by the user-specified tolerance perameter. If the current depth is within the tolerance, this depth is considered to have been achieved. The

behavior also stores the depth from the prior behavior iteration, and if the target depth is between the prior depth and current depth, the depth is considered to be achieved regardless of tolerance setting.

The Loiter behavior: A behavior for transiting to and repeatedly traversing a set of waypoints. A similar effect can be achieved with the waypoint behavior, but this behavior assumes a set of waypoints forming a convex polygon. It also robustly handles dynamic exit and reentry modes when or if the vehicle diverges from the loiter region due to external events. It is dynamically reconfigurable to allow a mission control module to repeatedly reassign the vehicle to different loiter regions by using a single persistent instance of the behavior.

The MemoryTurnLimit behavior: The objective of the MemoryTurnLimit behavior is to avoid turns that may cause the vehicle to cross back on its own path and risk damage to towed equipment. Its configuration is determined by two parameters combined to set a vehicle turn radius pseudo-limit. This behavior is described more fully in Section 4.5.2.

The OpRegion behavior: This behavior provides four different types of safety functionality—(a) a boundary box given by a convex polygon in the x–y or lat–lon plane, (b) an overall time-out, (c) a depth limit, and (d) an altitude limit. The behavior does not produce an objective function to influence the vehicle to avoid violating these safety constraints. It merely monitors the constraints and may post an error resulting in the posting of an all-stop command.

The PeriodicSpeed behavior: This behavior will periodically influence the speed of the vehicle while remaining neutral at other times. The timing is specified by a given period in which the influence is on and a period specifying when the influence is off. The motivation was to provide an ability to periodically reduce self-noise to allow for a window of acoustic communications.

The PeriodicSurface behavior: This behavior will periodically influence the depth and speed of the vehicle while remaining neutral at other times. The purpose is to bring the vehicle to the surface periodically to achieve some event specified by the user, typically the receipt of a GPS fix. Once this event is achieved, the behavior resets its internal clock to a given period length and will remain idle until a clock time-out occurs.

The Shadow behavior: The Shadow behavior will mimic the observed heading and speed of another given vehicle, regardless of its position relative to the vehicle.

The StationKeep behavior: This behavior is designed to keep the vehicle at a given lat/lon or x, y station-keep 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, the range of linear speeds, and a default transit speed if the vehicle is outside the outer radius.

The Trail behavior: The Trail behavior will attempt to keep the vehicle at a specified position relative, given in terms of range and bearing, to another specified vehicle. It may serve the purpose of formation keeping. It has the ability to extrapolate the other vehicle's position from prior reports when contact reports are received intermittently.

The Waypoint behavior: The Waypoint behavior is used for transiting to a set of specified waypoint in the x–y plane. The primary parameter is the set of waypoints. Other key parameters are the inner and outer radius around each waypoint that determine what it means to have met the conditions for moving on to the next waypoint. This behavior is discussed further in Section 4.5.1.

4.5. A Closer Look at Four Behaviors

4.5.1. The Waypoint Behavior

The Waypoint behavior is used for transiting to a set of specified waypoints in the x–y plane. The primary parameter is the set of waypoints. Other key parameters are the inner and outer radius around each waypoint that determine what it means to have met the conditions for moving on to the next waypoint. The basic idea is shown in Figure 15.

The behavior may also be configured to perform a degree of track-line following, that is, steering the vehicle not necessarily toward the next waypoint but to a point on the line between the previous and next waypoints. This is to ensure that the vehicle stays closer to this line in the face of external forces such as wind or current. The behavior may also be set to "repeat" the set of waypoints indefinitely or a fixed number of times. The waypoints may be specified either directly at start-up or supplied dynamically during the operation of the vehicle. There are also a number of accepted geometry patterns that may be given in lieu of specific waypoints, such as polygons and lawnmower pattern.

The Waypoint Behavior Configuration Parameters

The configuration parameters and variables published collectively define the interface for the behavior. The following are the parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 4.2:

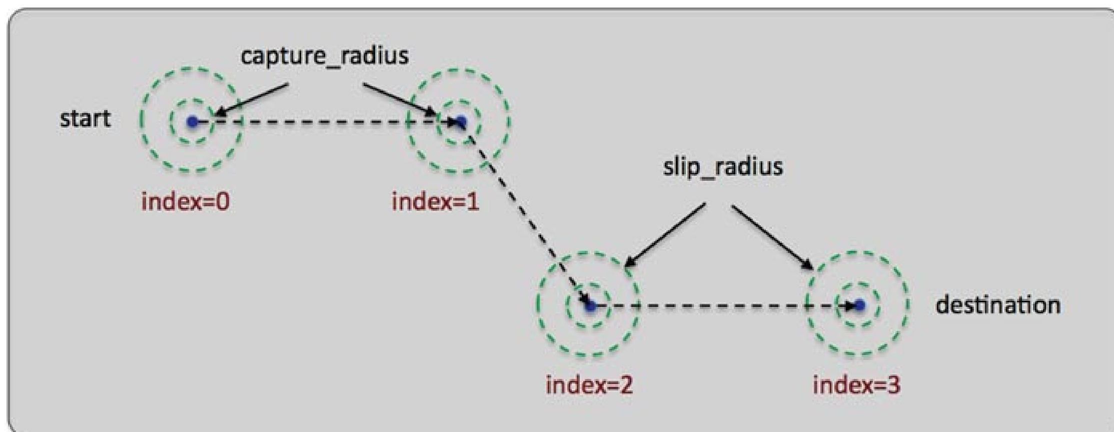

Figure 15. The Waypoint behavior: The Waypoint behavior's basic purpose is to traverse a set of waypoints. A capture radius is specified to define what it means to have achieved a waypoint, and a slip radius is specified to define what it means to be "close enough" should progress toward the waypoint be noted to degrade.

- `POINTS:` A colon-separated list of x, y pairs given as points in 2D space, in meters.
- `POINT:` A single x, y pair given as a point in 2D space, in meters.
- `POLYGON:` An alias for `POINTS`.
- `SPEED:` The desired speed (meters per second) at which the vehicle travels through the points.
- `CAPTURE_RADIUS:` The radius tolerance, in meters, for satisfying the arrival at a waypoint.
- `RADIUS:` An alias for `CAPTURE_RADIUS`.
- `SLIP_RADIUS:` An "outer" capture radius. Arrival declared when the vehicle is in this range and the distance to the next waypoint begins to increase.
- `ORDER:` The order in which waypoints are traversed—`"normal"` or `"reverse."`
- `LEAD:` If this parameter is set, track-line following between waypoints is enabled.
- `LEAD_DAMPER:` Distance from trackline within which the `lead` distance is stretched out.
- `REPEAT:` The number of *extra* times traversed through the waypoints. Or `"forever."`
- `CYCLEFLAG:` MOOS variable-value pairs posted at end of each cycle through waypoints.
- `POST_SUFFIX:` A suffix tagged onto the `WPT_STATUS`, `WPT_INDEX`, and `CYCLE_INDEX` variables.
- `VISUAL_HINTS:` Hints for visual properties in variables posted intended for rendering.

Variables Published by the Waypoint Behavior

The MOOS variables below will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a different variable name using the `post_mapping` configuration parameter, defined for all behaviors.

- `WPT_STAT:` A comma-separated string showing the status in hitting the list of points.
- `WPT_INDEX:` The index of the current waypoint. First point has index 0.
- `CYCLE_INDEX:` The number of times the full set of points has been traversed, if repeating.
- `VIEW_POINT:` A visual cue for indicating the waypoint being currently heading toward.
- `VIEW_POINT:` A visual cue for indicating the steering point, if the `lead` parameter is used.
- `VIEW_SEGLIST:` A visual cue for rendering the full set of waypoints.

The following are some examples:

```
    WPT_STAT = vname=alpha,behavior-name=waypt
               _survey,index=1,hits=1/1,cycles
               =0,dist=30,eta=15,
   WPT_INDEX = 3,
 CYCLE_INDEX = 1,
  VIEW_POINT = active,true:label,alpha's track-
               point:label_color,0,0.5,0:
               type,track_point:source,
               alphawaypt_survey:vertex_color,
               1,0,0:60,-152.57,0,
```

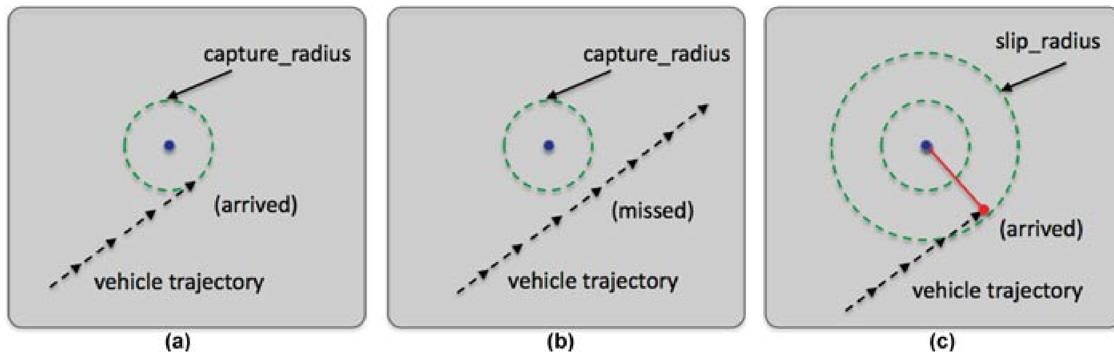

Figure 16. The capture radius and slip radius: (a) a successful waypoint arrival by achieving proximity less than the capture radius, (b) a missed waypoint likely resulting in the vehicle looping back to try again, and (c) a missed waypoint but arrival declared anyway when the distance to the waypoint begins to increase and the vehicle is within the slip radius.

```
VIEW_SEGLIST = label,alpha_waypt_survey:vertex_
               color,1,1,0:edge_size,1.0:
               vertex_size,2.0:60,-40:60,-160:
               150,-160:180,-100:150,-40.
```

Specifying Waypoints—the `Points`, `Order`, and `Repeat` Parameters

The waypoints may be specified explicitly as a colon-separated list of comma-separate pairs or implicitly using a geometric description. The order of the parameters may also be reversed with the `order` parameter. An example specification follows:

```
points    = 60,-40:60,-160:150,-160:180,
            -100:150,-40,
order     = reverse  // default is "normal,"
repeat    = 3        // default is 0.
```

A waypoint behavior with this specification will traverse the five points in reverse order, (150, −40 first), four times (one initial cycle repeated three times) before completing. If there is a syntactic error in this specification at helm start-up, an output error will be generated and the helm will not continue to launch. If the syntactic error is passed as part of a dynamic update (see Section 4.2.2), the change in waypoints will be ignored and a warning posted to the `BHV_WARNING` variable. The behavior can be set to repeat its waypoints indefinitely by setting `repeat="forever."`

The `capture_radius` and `slip_radius` Parameters

The `capture_radius` parameter specifies the distance to a given waypoint where the vehicle must be before it is considered to have arrived at or achieved that waypoint. It is the inner radius around the points in Figure 15. The slip radius parameter specifies an alternative criterion for achieving a waypoint.

As the vehicle progresses toward a waypoint, the sequence of measured distances to the waypoint decreases monotonically. The sequence becomes nonmonotonic when it hits its waypoint or when there is a near miss of the waypoint capture radius. The `slip_radius` is a capture radius distance within which a detection of increasing distances to the waypoint is interpreted as a waypoint arrival. This distance would have to be larger than the capture radius to have any effect. As a rule of thumb, a distance of twice the capture radius is practical. The idea is illustrated in Figure 16. The behavior keeps a running tally of hits achieved with the capture radius and those achieved with the slip radius. These tallies are reported in a status message.

Track-Line Following Using the `lead` Parameter

By default the waypoint behavior will output a preference for the heading that is directly toward the next waypoint. By setting the `lead` parameter, the behavior will instead output a preference for the heading that keeps the vehicle closer to the track line, or the line between the previous waypoint and the waypoint currently being driven to. See Figure 17.

The distance specified by the `lead` parameter is based on the perpendicular intersection point on the track line. This is the point that would make a perpendicular line to the track line if the other point determining the perpendicular line were the current position of the vehicle. The distance specified by the `lead` parameter is the distance from the perpendicular intersection point toward the next waypoint and defines an imaginary point on the track line. The behavior outputs a heading preference based on this imaginary steering point. If the lead distance is greater than the distance to the next waypoint along the track line, the imaginary steering point is simply the next waypoint.

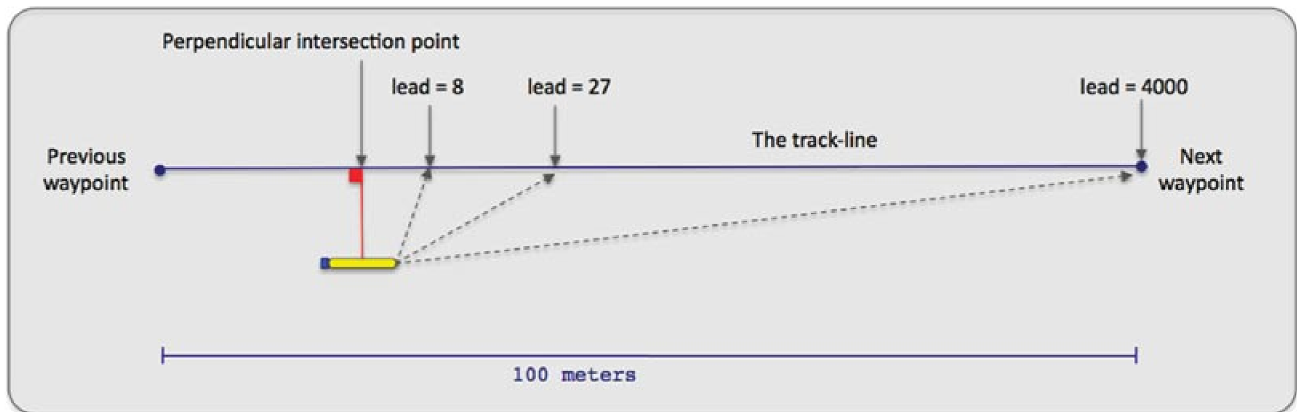

Figure 17. The track-line mode: When in track-line mode, the vehicle steers toward a point on the track line rather than simply toward the next waypoint. The steering point is determined by the `lead` parameter. This is the distance from the perpendicular intersection point toward the next waypoint.

If the `lead` parameter is enabled, it may be optionally used in conjuction with the `lead_damper` parameter. This parameter expresses a distance from the track line in meters. When the vehicle is within this distance, the value of the `lead` parameter is stretched out toward the next waypoint to soften, or dampen, the approach to the track line and reduce overshooting the track line.

The Objective Function Produced by the Waypoint Behavior

The Waypoint behavior produces a new objective function, at each iteration, over the variables `speed` and `course/heading`. The behavior can be configured to generate this objective function in one of two forms, either by coupling two independent one-variable functions or by generating a single coupled function directly. The function rendered in Figure 18 is built in the first manner.

4.5.2. The MemoryTurnLimit Behavior

The objective of the MemoryTurnLimit behavior is to avoid turns that may cause the vehicle to cross back on its own path and risk damage to any towed equipment. Its configuration is determined by the two parameters described below, which combine to set a vehicle turn radius limit.

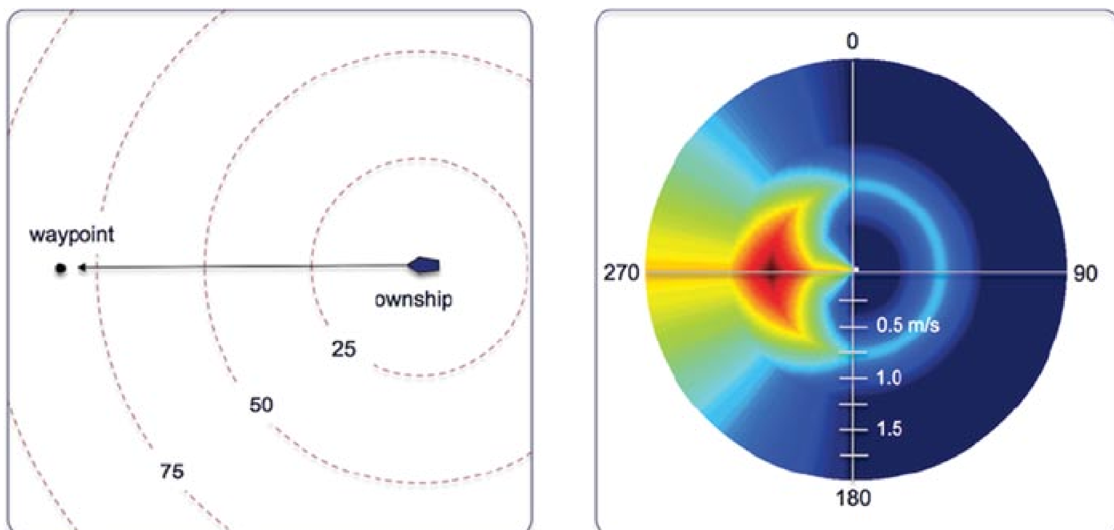

Figure 18. A Waypoint objective function: The objective function produced by the Waypoint behavior is defined over possible heading and speed values. Depicted here is an objective function favoring maneuvers to a waypoint 270 deg from the current vehicle position and favoring speeds closer to the midrange of capable vehicle speeds. Higher speeds are represented farther radially out from the center.

However, it is not strictly described by a limited turn radius; it stores a time-stamped history of recent recorded headings and maintains a *heading average* and forms its objective function on a range deviation from that average. This behavior merely expresses a preference for a particular heading. If other behaviors also have a heading preference, coordination/compromise will take place through the multiobjective optimization process. The following parameters are defined for this behavior:

`MEMORY_TIME`: The duration of time for which the heading history is maintained and heading average calculated.

`TURN_RANGE`: The range of heading values deviating from the current heading average outside of which the behavior reflects sharp penalty in its objective function.

The heading history is maintained locally in the behavior by storing the currently observed heading and keeping a queue of n recent headings within the `MEMORY_TIME` threshold. The heading average calculation below handles the issue of angle wrap in a set of n headings $h_0 \ldots h_{n-1}$, where each heading is in the range [0, 359]:

$$\text{heading_avg} = \text{atan2}(s, c) \cdot 180/\pi, \tag{1}$$

where s and c are given by

$$s = \sum_{k=0}^{n-1} \sin[h_k \pi/180], \quad c = \sum_{k=0}^{n-1} \cos[h_k \pi/180].$$

The vehicle turn radius r is not explicitly a parameter of the behavior but is given by

$$r = v/[(u/180)\pi],$$

where v is the vehicle speed and u is the turn rate given by $u =$ `TURN_RANGE/MEMORY_TIME`. The same turn radius is possible with different pairs of values for `TURN_RANGE` and `MEMORY_TIME`. However, larger values of `TURN_RANGE` allow sharper initial turns but temper the turn rate after the initial sharper turn has been achieved.

The objective function produced by this behavior looks effectively like a constraint on the value of the heading. Some typical functions are rendered from simulation in Figure 19. This figure depicts the MemoryTurnLimit IvP function alongside the Waypoint IvP function as the vehicle traverses a set of waypoints, where the final waypoint (waypoint #3 in the figure) represents nearly a 180-deg turn with respect to the prior waypoint. The first frame in the figure depicts the vehicle trajectory *without* the influence of the MemoryTurnLimit behavior and shows a very sharp turn between waypoints #2 and #3.

The MemoryTurnLimit behavior, shown in the last five frames of Figure 19, is used to avoid the sharp turn in the first frame. This behavior was configured with `TURN_RANGE=45` and `MEMORY_TIME=20`. The turn between waypoints #1 and #2 is not affected by the MemoryTurnLimit behavior because the new desired heading (180 deg) is within the tolerance of the heading history recorded up to the turning point. The same cannot be said when the vehicle reaches waypoint #2 and begins to traverse to waypoint #3. The recent heading history as it arrives at waypoint #2 reflects the time spent traversing from waypoint #1 and #2. The heading average given by Eq. (1) at this point (time = 98) is 180 deg, and the IvP function produced by the MemoryTurnLimit behavior restricts the vehicle heading to be 180 ± 45 deg. As the vehicle continues its turn toward waypoint #3, the heading average of the MemoryTurnLimit behavior and its IvP function evolve to eventually allow a desired heading that is consistent with the optimal point of the waypoint IvP function as shown in the last frame at time = 170. The resulting turn between waypoints #2 and #3 is much wider than that shown in the first frame. Discussion of this behavior used in field experiments with a UUV towing a sensor array can be found in Benjamin, Battle, Eickstedt, Schmidt, and Balasuriya (2007).

4.5.3. The AvoidCollision Behavior

The AvoidCollision behavior will produce IvP objective functions designed to avoid collisions (and near collisions) with another specified vehicle. The IvP functions produced by this behavior are defined over the domain of posssible heading and speed choices. The utility assigned to a point in this domain (a heading-speed pair) depends in part on the calculated CPA between the candidate maneuver leg and the contact leg formed from the contact's position and trajectory. Figure 20 shows the relationship cpa(θ, v) between CPA and candidate maneuvers (θ, v), where θ = heading and v = speed, for a given relative position between ownship and a given contact vehicle and trajectory. The IvP function generated by the AvoidCollision behavior applies a further user-defined utility function to the CPA calculation for a candidate maneuver, $f(\text{cpa}(\theta, v))$. The form of $f()$ is determined by behavior configuration parameters, described below.

The AvoidCollision Configuration Parameters

The following parameters are defined for this behavior, in addition to the parameters defined for all IvP behaviors discussed earlier. A more detailed description follows.

`COMPLETED_DIST`	Range to contact outside of which the behavior completes and dies.
`MAX_UTIL_CPA_DIST`	Range to contact outside which a considered maneuver will have max utility.

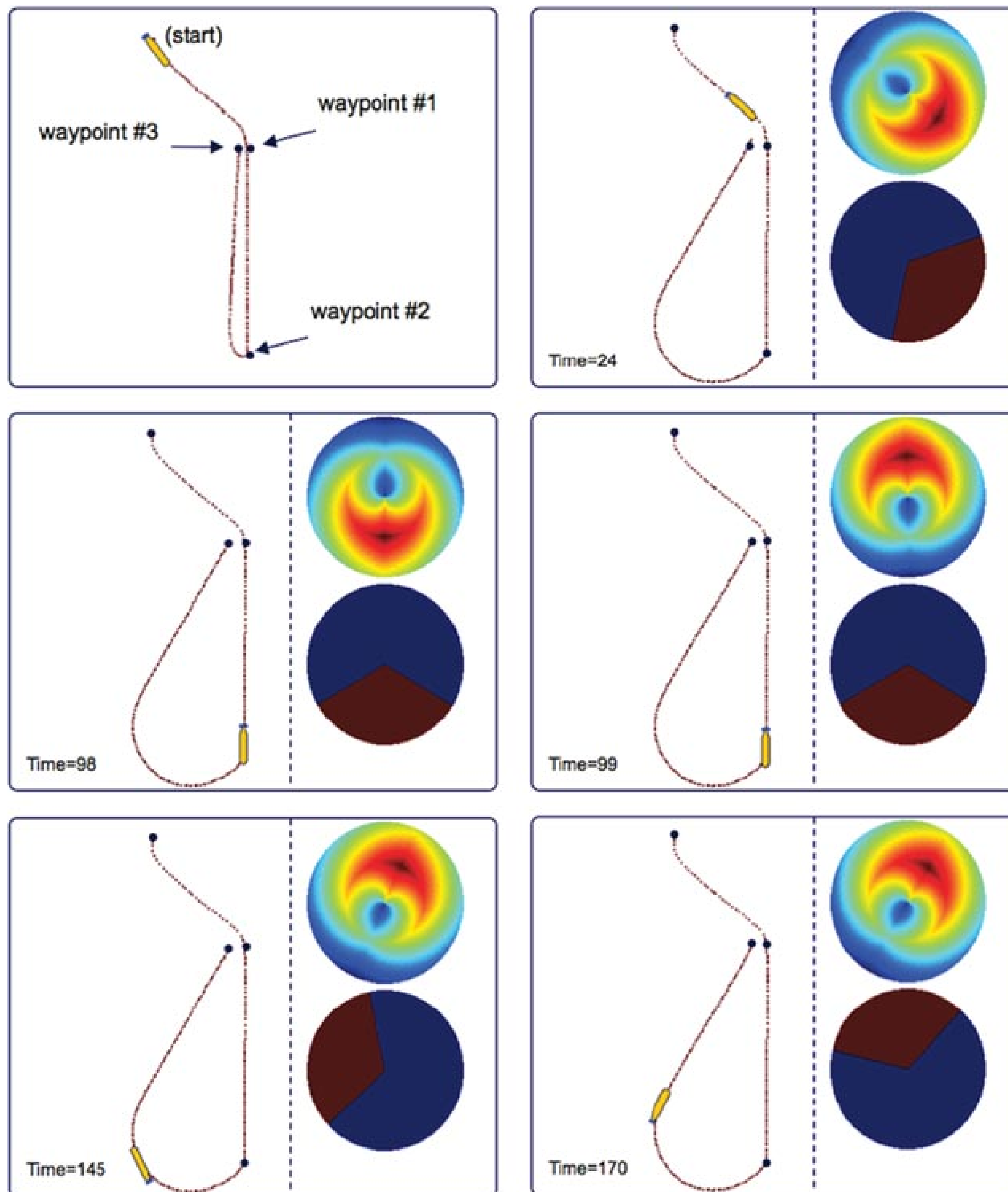

Figure 19. The MemoryTurnLimit and Waypoint behaviors: The MemoryTurnLimit behavior is used to avoid sharp vehicle turns between waypoints as shown, from simulation, in the first, upper-left frame. The MemoryTurnLimit behavior is configured with `TURN_RANGE=45` and `MEMORY_TIME=20`. In each frame, the IvP function of the waypoint behavior is shown in the upper right, and that of the MemoryTurnLimit behavior is in the lower right. At time=24, the vehicle is approaching waypoint #1 and its recent heading is 140 deg. At that point, the range of headings allowed by the MemoryTurnLimit behavior is 140 ± 45, which includes the heading of 180 desired by the Waypoint behavior to reach waypoint #2. When the vehicle reaches waypoint #2, between time=98 and time=99, the desired heading from the waypoint behavior is not in the range of allowed headings from the MemoryTurnLimit behavior. As the vehicle turns, e.g., time=145, the heading history of the MemoryTurnLimit behavior evolves to eventually allow the peak desired heading of the Waypoint behavior, at time=170.

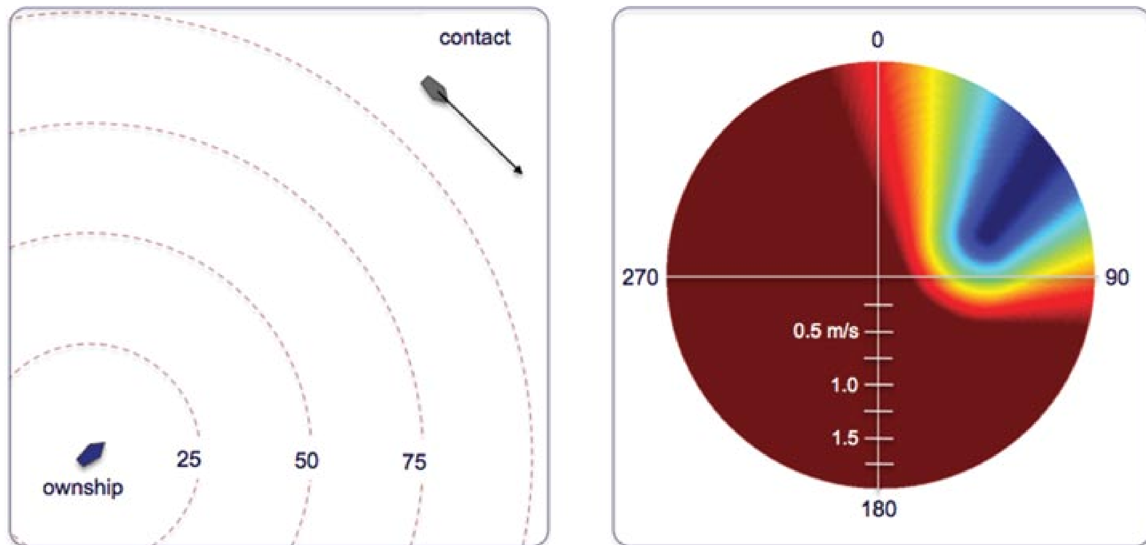

Figure 20. The closest point of approach mapping: The function on the right indicates the relative change in calculated closest point of approach between ownship and contact position and trajectory shown on the left.

`MIN_UTIL_CPA_DIST`	Range to contact within which a considered maneuver will have min utility.
`PWT_INNER_DIST`	Range to contact within which the behavior has maximum priority weight.
`PWT_OUTER_DIST`	Range to contact outside which the behavior has zero priority weight.
`CONTACT`	Name or unique identifier of a contact to be avoided.
`DECAY`	Time interval during which extrapolated position slows to a halt.
`EXTRAPOLATE`	If `true`, contact position is extrapolated from last position and trajectory.
`TIME_ON_LEG`	The time on leg, in seconds, used for calculating closest point of approach.

Spawning and Killing New Behavior Instances Dynamically

The AvoidCollision behavior may be configured as a *template* that spawns a new behavior for each new contact that presents itself. The below two configuration lines would achieve this objective:

```
templating = spawn,
updates    = COLL_AVOID_INFO.
```

Configured in this manner, the following posting to the MOOSDB would suffice to spawn a new instance of the behavior:

```
COLL_AVOID_INFO = name=avd_zulu_002 # contact\
                = zulu_002.
```

The MOOS variable matches the MOOS variable specified in the `UPDATES` configuration parameter. The `"name=avd_zulu_002"` component specifies a unique behavior name and indicates to the helm that a new behavior is to be spawned upon receipt. When the behavior is spawned, all initial behavior parameters supplied in the behavior mission file are applied, and the `"contact=zulu_002"` component is applied as a further configuration parameter. The new AvoidCollision instance will remain with the helm until the contact goes out of the range specified by the `COMPLETED_DIST` parameter. The posting of the MOOS variable that triggers the spawning is done by a contact manager. In this case the contact manager is a separate MOOS application called pBasicContactMgr. Details of this are beyond the scope of this paper.

Configuring the AvoidCollision Behavior Utility Function

The IvP function generated by this behavior is defined over the range of possible heading and speed decisions. Its form is derived in part from the calculation of the CPA calculated for a candidate maneuver leg, of a duration given by the `TIME_ON_LEG` parameter, which is set to 60 s by default. The *utility* of a given CPA value is determined further by a pair of configuration parameters. Distances less than or equal to `MIN_UTIL_CPA_DIST` are given the lowest utility,

equivalent to an actual collision. Distances greater than or equal to MAX_UTIL_CPA_DIST are given the highest utility. These two parameters, and the raw CPA calculations, determine the form of the function. The magnitude, or weight, of the function is determined by the range between the two vehicles and two further configuration parameters. At ranges greater than PWT_OUTER_DIST, the weight is set to zero. At ranges less than PWT_INNER_DIST, the weight is set to 100% of the user-configured priority weight. The weight varies linearly when the range between vehicles falls somewhere between.

Closest Point of Approach Calculations and Caching

The production of IvP functions for this behavior is potentially CPU intensive compared to other behaviors, primarily due to the repeated calculations of CPA values. Recall from Section 3.5.2 that IvP functions built with the reflector tool are built by repeated sampling of the underlying function. This repeated sampling, and the existence of common partial calculations between samples, allows for caching of intermediate results to greatly speed up sampling for this behavior. This is implemented in a C++ class called a CPAEngine in a separate utility library for use in other behaviors that reason about CPA, such as the CutRange and Trail behaviors.

Current ownship position is known and given by (x, y), and the other vehicle's current position and trajectory is given by $(x_b, y_b, \theta_b, v_b)$. To compute the CPA distance for a given $\langle\theta, v, t\rangle$, first the time $t_{\min}$ when the minimum distance between two vehicles occurs is computed. The distance between the two vehicles at the current time can be determined by the Pythagorean theorem. For any given time t (where the current time is $t = 0$), and assuming that the other vehicle stays on a constant trajectory, the distance between the two vehicles for any chosen $\langle\theta, v, t\rangle$ is given by

$$\mathrm{dist}^2(\theta, v, t) = k_2 t^2 + k_1 t + k_0, \tag{2}$$

where

$$\begin{aligned} k_2 &= \cos^2(\theta)v^2 - 2\cos(\theta)v\cos(\theta_b)v_b + \cos^2(\theta_b)v_b^2 \\ &\quad + \sin^2(\theta)v^2 - 2\sin(\theta)v\sin(\theta_b)v_b + \sin^2(\theta_b)v_b^2, \\ k_1 &= 2\cos(\theta)vy - 2\cos(\theta)vy_b - 2y\cos(\theta_b)v_b \\ &\quad + 2\cos(\theta_b)v_b y_b + 2\sin(\theta)vx - 2\sin(\theta)vx_b \\ &\quad - 2x\sin(\theta_b)v_b + 2\sin(\theta_b)v_b x_b, \\ k_0 &= y^2 - 2yy_b + y_b^2 + x^2 - 2xx_b + x_b^2. \end{aligned}$$

The stationary point is obtained by taking the first derivative with respect to t:

$$\mathrm{dist}^2(\theta, v, t)' = 2k_2 t + k_1.$$

Because there is no "maximum" distance, this stationary point always represents the time of the closest point of approach, and therefore $t_{\min} = -k_1/2k_2$. The value of $t_{\min}$ may be in the past, i.e., less than zero, if the two vehicles are currently opening range. On the other hand $t_{\min}$ may occur after t, the time length of the candidate maneuver $\langle\theta, v, t\rangle$. Therefore the value of $t_{\min}$ is clipped by $[0, t]$. Furthermore, $t_{\min} = 0$ in the special case when the two vehicles have the same heading and speed (the only case in which k_2 is zero). The actual CPA value is obtained by substituting $t_{\min}$ back into Eq. (2):

$$\mathrm{cpa}(\theta, v) = \sqrt{k_2 {t_{\min}}^2 + k_1 t_{\min} + k_0}. \tag{3}$$

In the generation of a single IvP function with $\mathrm{cpa}(\theta, v)$ as a component of each sample of the decision space, intermediate values [Eq. (2)] may be cached that have the same values of current vehicle position (x, y) and current position and trajectory of the other vehicle $(x_b, y_b, \theta_b, v_b)$. A further cache, normally of size 360, is typically used for terms involving θ, ownship heading.

The AvoidCollision Behavior in Action

The AvoidCollision behavior is shown in Figure 21 in a simple scenario working with the waypoint behavior in simulation. In Figure 29 later in the paper, the effect of the AvoidCollision behavior in two fielded UUVs is shown.

4.5.4. The StationKeep Behavior

This behavior is designed to keep the vehicle at a given lat/lon or x,y station-keep 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, the range of linear speeds, and a default transit speed if the vehicle is outside the outer radius. See Figure 22.

An alternative to this station-keeping behavior is an active loiter around a very tight polygon with the Loiter behavior. This station-keeping behavior conserves energy and aims to minimize propulsor use. The behavior can be configured to station keep at a preset point or wherever the vehicle happens to be when the behavior transitions into an active state.

The station-keep behavior was initially developed for use on an autonomous kayak. A vehicle's control system, i.e., the front-seat driver described in Section 1.2., may have a native station-keeping mode, in which case the activation of this behavior would be replaced by a message from the backseat autonomy system to invoke the station-keeping mode. It is also worth pointing out that most UUVs are positively buoyant and will simply come to the surface if commanded with a zero speed.

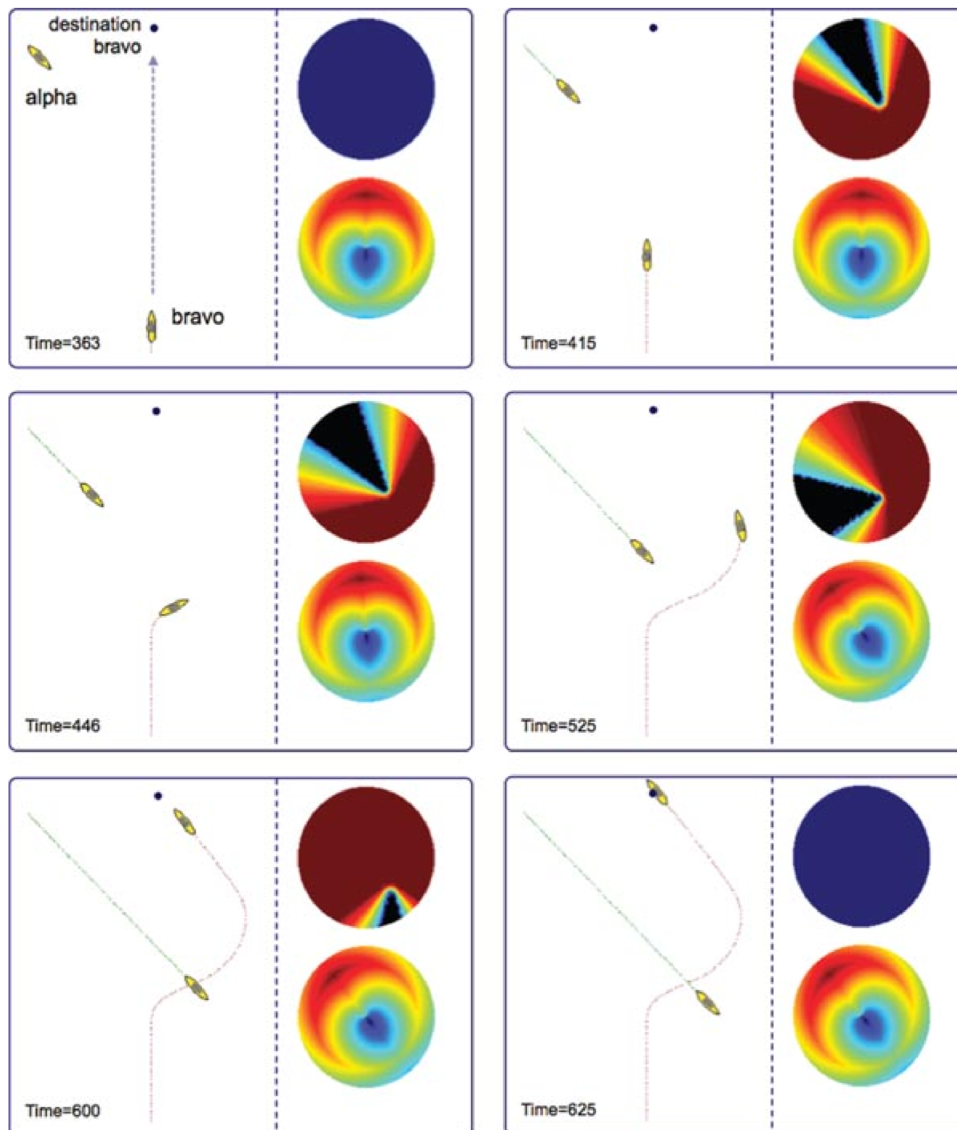

Figure 21. The AvoidCollision and Waypoint behaviors: The bravo vehicle maneuvers to a destination point with the Waypoint behavior. The IvP objective function produced by the waypoint behavior is the lower function shown on the right in each frame. Beginning in the second frame, time=415, the AvoidCollision behavior becomes active and begins to produce IvP objective functions shown in the upper right of each frame. At each point in time, the helm chooses the heading and speed that represent the optimal decision given the pair of IvP functions. At time=625, the AvoidCollision behavior ceases to produce an IvP function due to the range between vehicles.

The StationKeep Behavior Configuration Parameters

The following parameters are defined for this behavior, in addition to the parameters defined for all IvP behaviors discussed earlier. A more detailed description is provided below.

`STATION_PT`: An x,y pair given as a point in local coordinates.

`POINT`: A supported alias for `STATION_POINT`.

`CENTER_ACTIVE`: If `true`, station keep at position upon activation.

`INNER_RADIUS`: Distance to station point within which the preferred speed is zero.

`OUTER_RADIUS`: Distance within which the preferred speed begins to decrease.

`OUTER_SPEED`: Preferred speed at outer radius, decreasing toward inner radius.

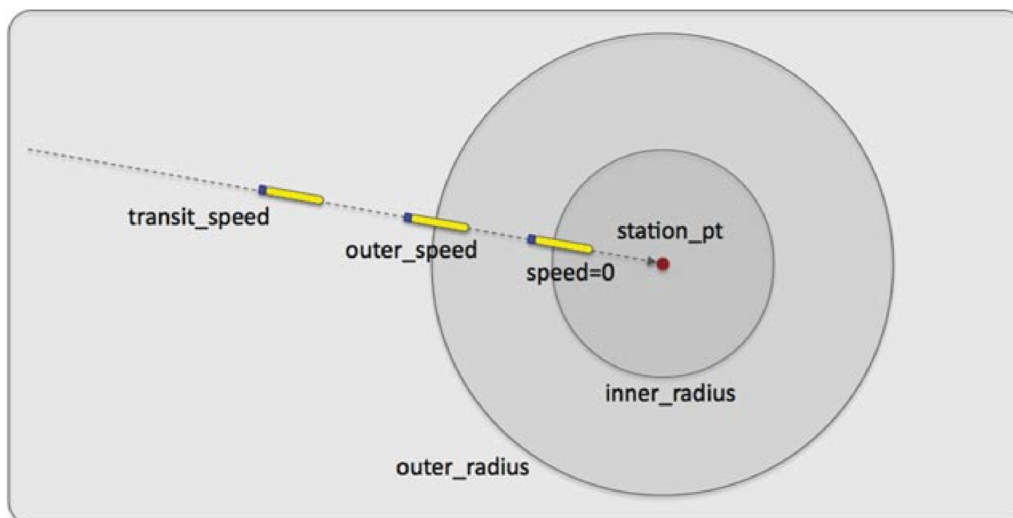

Figure 22. The station-keep behavior parameters: The station-keep behavior can be configured to approach the outer station circle with a given transit speed and will decrease its preference for speed linearly between the outer radius and the inner radius. The preferred speed is zero when the vehicle is at or inside the inner radius.

`SWING_TIME:`	Duration of drift of station circle with vehicle upon activation.
`TRANSIT_SPEED:`	Preferred speed beyond the outer radius.
`EXTRA_SPEED:`	A deprecated alias for `TRANSIT_SPEED`.
`PASSIVE_STATION_RADIUS:`	A radius used for low-power, passive station keeping.
`PASSIVE_STATION_VARIABLE:`	Name of MOOS variable used for conveying passive-station mode.

Setting the Station-Keep Point and Radial-Speed Relationships

The station-keep point is set in one of two ways: either with a prespecified fixed position or with the vehicle's current position when the vehicle transitions into the running state. To set a fixed station-keep position,

```
station_pt = 100,250.
```

To configure the behavior to station keep at the vehicle's current position when it enters the running state,

```
center_active = true.
```

At the outset of station keeping via `center_active`, the vehicle typically is moving at some speed. Despite the fact that station keeping is immediately active and typically results in a desired speed of zero if no other behaviors are active, the vehicle will continue some distance before coming to a near or complete stop in the water, thus "overshooting" the station-keep point. This often means that the station-keep behavior will immediately turn the vehicle around to come back to the station-keep point. This can be countered by setting the behavior's "swing time" parameter, the amount of time after initial center activation that the station-keep point is allowed to drift with the current position of the vehicle before becoming fixed. The format is

```
swing_time = <time-duration>  // default is 0.
```

The `<time-duration>` is given in seconds, and the duration is clipped by the range [0, 60].

If the behavior enters the running state, but center activation is not set to true and no prespecified fixed position is given, the behavior will not produce an objective function. It will remain in the running state but not the active state. (See Section 3.4.2. for more detail on behavior run states.) In this situation, a warning will be posted: `BHV_WARNING="STATION_POINT_NOT_SET."`

The `INNER_RADIUS` and `OUTER_RADIUS` parameters affect the preferred speed of the behavior as it relates to the vehicle's current range to the station point. The preferred speed at the outer radius is given by the parameter `OUTER_SPEED`. The preferred speed decreases linearly to zero as the vehicle approaches the inner radius. The default values for the inner and outer radii are 4 and 15, respectively. If configured with values such that the inner is greater then the outer, this will not trigger an error, but the two radius parameters will be collapsed to the value of the inner radius on the first iteration of the behavior.

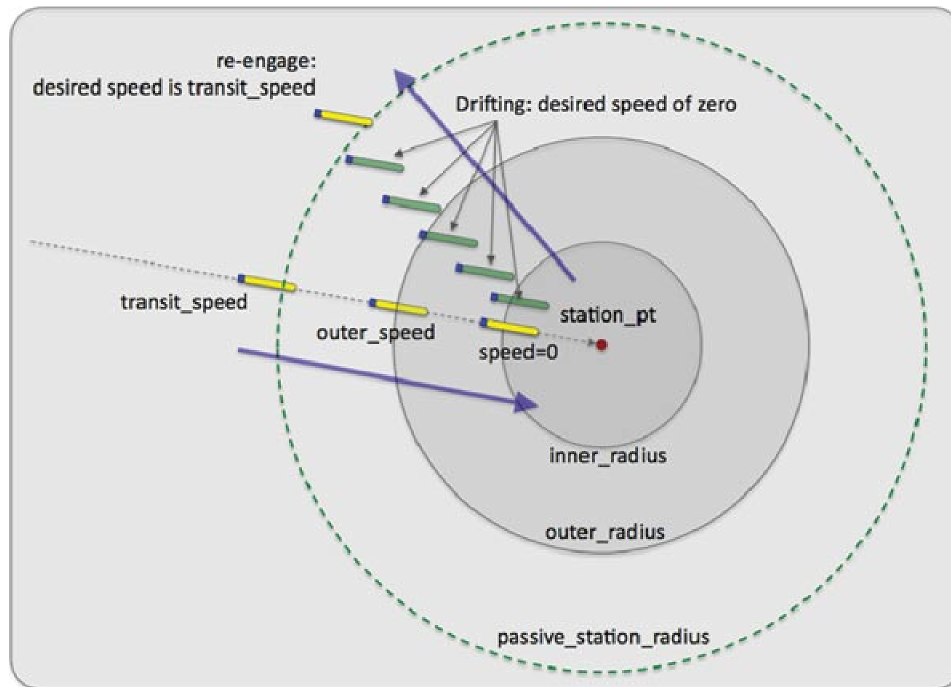

Figure 23. Passive station-keeping: The station-keep behavior can be configured in the "passive" mode. The vehicle will move toward the station point until it reaches the inner_radius or until progress ceases. It will then drift until its distance to the station point is beyond the passive_station_radius. At this point it will reengage to reach the station point and may trigger another behavior to dive.

Passive Low-Energy Station-Keeping Mode

The station-keep behavior can be configured to operate in a "passive" mode. This mode differs from the default mode primarily in the way it acts after it reaches the inner radius, i.e., the point at which the behavior regards the vehicle to be on station and outputs a preferred speed of zero. In the normal mode, the behavior will begin to output a preferred heading and nonzero speed as soon as the vehicle slips beyond the inner radius. In the passive mode, the behavior will let the vehicle drift or otherwise move to a distance specified by the `PASSIVE_STATION_RADIUS` before it resumes outputting a preferred heading and nonzero speed. The idea is illustrated in Figure 23.

This mode was built with UUVs in mind. Most UUVs are deployed having a positive buoyancy (battery dies—vehicle floats to the surface). They need to be moving at some speed to maintain a depth. Furthermore, it may not be safe to assume that a UUV can effectively execute a desired heading when it is operating on the surface. For these reasons, when operating in the passive mode, this behavior will publish a variable indicating whether it is in the mode of drifting or attempting to make progress toward the station point. The status is published in the variable `PSKEEP_MODE`, short for "passive station-keeping mode." This variable will be set to `"SEEKING_STATION"` when outputting a nonzero speed preference and presumably moving toward the station point. The variable will be set to `"HIBERNATING"` otherwise. This opens the option of configuring the helm with the ConstantDepth behavior to work in conjunction with the StationKeep behavior by conditioning the ConstantDepth behavior to be running only when `PSKEEP_MODE="SEEKING_STATION."` The idea is illustrated in Figure 24.

This behavior mode is regarded as "low-power" due to the presumably long periods of drifting before resuming actively seeking the station point. A couple of safeguards are designed to ensure that when the behavior is in the `"STATION_SEEKING"` mode, it does not get hung or stuck in this mode for much longer than intended or needed. How could one become stuck in this mode? Two ways—by either reaching an equilibrium at-speed (and perhaps at-depth) state in which the vehicle is neither progressing toward nor away from the `inner_radius` or repeatedly "missing" the `inner_radius` by heading right past it.

Both cases can be guarded against and detected by monitoring the history of vehicle speed in the direction of the station point. If this speed becomes zero, an equilibrium state is assumed, and if it becomes negative, it is assumed that the vehicle missed the inner radius circle entirely. In short, the StationKeep behavior exits the `"STATION_SEEKING"` mode and enters the `"HIBERNATING"`

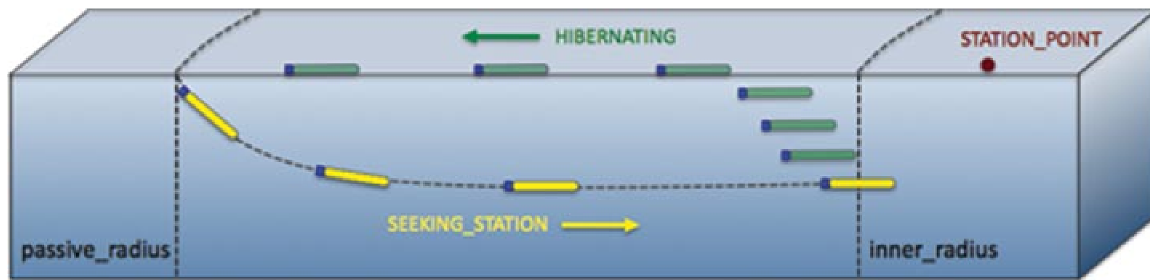

Figure 24. Passive station-keeping with depth coordination: The passive mode can be coordinated with the ConstantDepth behavior to dive each time the StationKeep behavior enters the "SEEKING_STATION" mode. This ensures that a UUV needing to be at depth to have reliable heading control will indeed be at depth when it needs to be.

mode when it detects that the vehicle speed toward the station point reaches zero. To calculate this vehicle speed, a 10-s history of range to the station point is kept by the behavior. A zero speed or "stale-progress" criterion is declared simply if the range to the station point for the most recent measure in the history is not less than the range of 10 s ago in the history list. The behavior will transition into the "HIBERNATING" mode if either the inner-radius or stale-progress criterion is met.

It is also possible that when the StationKeep behavior enters the "SEEKING_STATION" mode from the "HIBERNATING" mode, the vehicle initially begins to open its range to the station point before it begins to close range. This would be expected, for example, if the vehicle were pointed away from the station point when the behavior first entered the "SEEKING_STATION" mode. In this case it is quite possible that the behavior would correctly, but in a way that is unwanted, infer that the stale-progress criterion has been met. For this reason, the stale-progress criterion is not applied until an "initial-progress" criterion is met after entering the "SEEKING_STATION" mode. The same 10-s history is used to detect when the vehicle begins to make initial progress, i.e., closing range, toward the station point.

5. APPLICATION EXAMPLE: MULTISTATIC ACOUSTIC SENSING NETWORKS

5.1. Nested Autonomy for Environmental Acoustic Sensing

Undersea observation, mapping, and monitoring are experiencing a dramatic paradigm shift from platform-centric, human-controlled sensing, processing, and interpretation toward distributed sensing concepts using networks of autonomous platforms. A similar trend in land- and air-based systems has long been underway. The principal reason for the delayed adaptation of the undersea distributed systems is the fact that the capacity and reliability of underwater communication is many orders of magnitude below land- and air-based equivalents. What has made the transition to distributed underwater sensing systems feasible is the possibility of incorporating significant computational capacity on the network nodes and therefore making them more autonomous and less dependent on communication connectivity than conventional shipborne and fixed or cabled sensors. In addition, onboard "intelligence" in the form of behavior-based autonomy enables the autonomous detection, classification, localization, and tracking (DCLT) of episodic undersea events, exploiting adaptive and collaborative sensing behaviors, without direct operator control. The nested autonomy architecture for environmental acoustic sensing shown in Figure 25 is a particular instantiation of the general nested autonomy concept shown in Figure 5.

Although a high level of autonomy allows the network nodes to complete their mission with limited, latent, and intermittent communication, a robust communication system is still required for cueing the sensor nodes, for alerting the network of a detected event, and for collaborative event tracking.

As mentioned earlier, the *nested autonomy* paradigm is specifically developed for heterogeneous networks operating within layered communication capabilities of the undersea environment. Thus, the operator may use high-bandwidth RF communication with buoys and other surface nodes, but the communication link between these and the submerged assets is many orders of magnitude lower in capacity. On the other hand, in many cases the RF connectivity may be occasional when at the surface, and the latency of the operator communication may be minutes or hours, and in such cases the network may exploit local acoustic communication within clusters of nodes for enhancing performance through autonomous collaboration.

In addition, an optimal platform suite will often be highly heterogeneous with large differences in mobility, maneuverability, sensing capability, and communication connectivity. The sensor systems have different constraints on platform mobility and communication capacity, and some network operations require highly coordinated maneuvering of heterogeneous platforms. The MOOS-IvP *payload autonomy* architecture inherently supports collaborative sensing missions by a heterogeneous network. Thus, by concentrating not only the autonomous command and

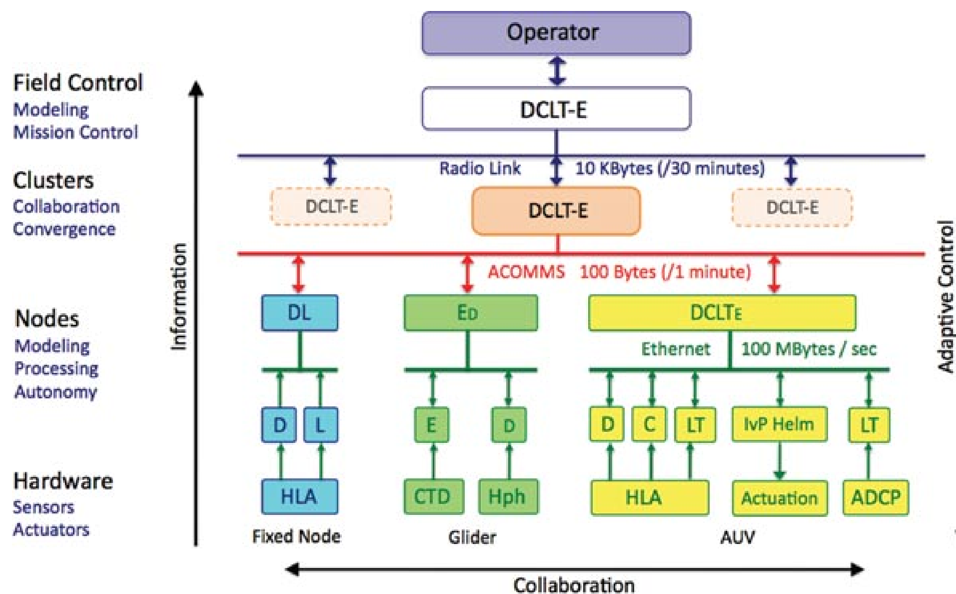

Figure 25. Nested autonomy for environmental acoustic sensing: Nested autonomy architecture for heterogeneous undersea networks for environmental and acoustic sensing, involving gliders for environmental profiling and fixed hydrophone line arrays (HLAs) and AUVs for acoustic sensing. The activity at each level component is denoted by the key D for detection, C for classification, L for localization, T for tracking, and E for environmental sampling or modeling. Larger and smaller script keys denote primary and secondary roles of the activity within a component.

control (C2), but also the communication management in the portable payload architecture, a unified network C2 can be established.

5.2. Communication, Command, and Control Infrastructure

Acoustic communication is the only option for establishing undersea network connectivity. In contrast to air- and land-based equivalents, the extremely limited bandwidth, latency, and intermittency of underwater acoustic communication impose severe requirements to the selectivity of message handling. Thus, contact and track reports for a high-priority event, such as a detected chemical plume from a deep ocean vent, which may indicate an imminent volcanic eruption, must be transmitted to the system operators without delay. On the other hand, reports concerning less important events and platform status reports may be delayed without significant effects. Most message handling systems have only a rigid, hard-coded queuing infrastructure and do not support such advanced priority-based selectivity, hampering the type and amount of information that can be passed between cooperating nodes in the network, severely limiting the level of autonomy that can be supported on the network nodes.

To overcome these limitations, a new MOOS-IvP communication software stack and associated C2 infrastructure has been developed at the MIT Laboratory for Autonomous Marine Sensing Systems (LAMSS). This new unified communication, C2 infrastructure is described in a companion paper (Schneider & Schmidt, 2010b), and only a summary shall be given here. The new MOOS communication software stack provides extremely robust message handling for collaborative autonomous sensing by heterogeneous, undersea autonomous assets, as demonstrated through successful deployment in a handful of major field experiments on autonomous sensing under programs such as the ONR ASAP MURI, GOATS, and SWAMSI programs. Being based on established libraries of message-handling software, the open-source architecture of this new MOOS communication stack lends itself directly to a wide range of military and civilian applications. Thus, it supports an arbitrary message suite and content without the requirement of modifying software. All message coding and decoding information is specified in a mission-unique configuration file written in the standard XML format.

As an example, Figure 26 shows the collaborative, multistatic MCM mission by the Unicorn and Macrura Bluefin-21 AUVs during SWAMSI-09 in Panama City, Florida. The two vehicles are circling a proud cylinder (cp) at a distance of 80 m, maintaining a constant bistatic angle of 60 deg. The collaboration was achieved fully autonomously without any intervention by the operators, with each vehicle adapting its speed based on its current position and the position of the other vehicle extrapolated from the latest status, contact, or track report. Such collaborative maneuvers would not be possible using traditional communication schemes, in which navigation packets must be rigidly

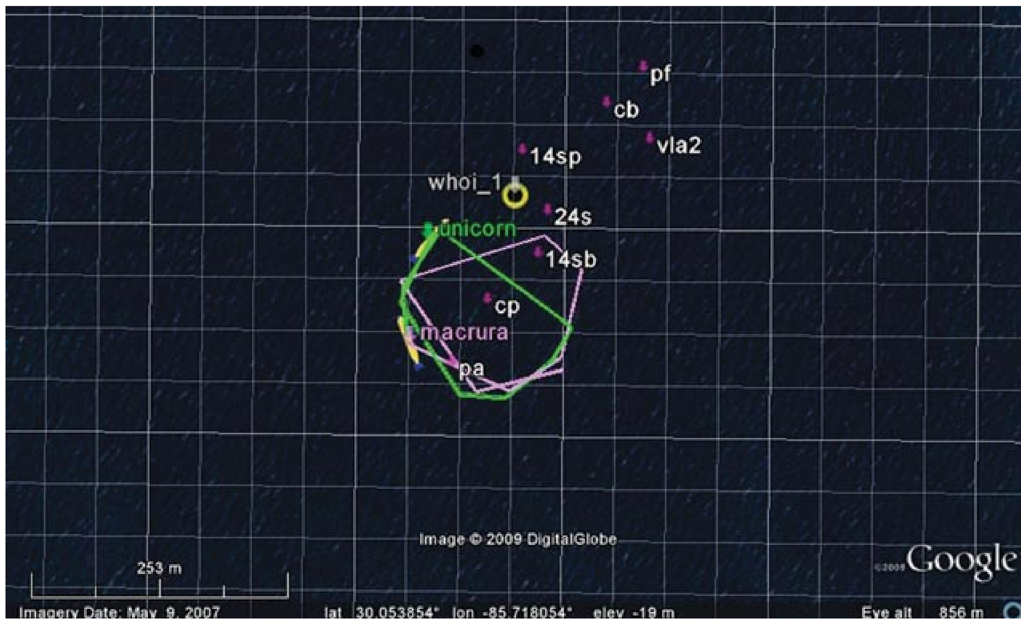

Figure 26. Collaborative autonomy demonstrated in SWAMSI09 using MIT LAMSS communication stack. The two BF21 AUVs Unicorn and Macrura perform synchronized swimming maintaining a constant bistatic angle of 60 deg relative to a proud cylindrical target (cp).

interleaved with messages containing data and C2 sequences. In contrast, the D-CCL coding applied by the MOOS communication stack allows for adequate navigation information to be packed with all other required message content.

A number of project-specific message sets have been defined, including active sonar contact and track reports for active, multistatic acoustic applications, environmental messages for oceanographic missions, and passive sonar contact and track reports for undersea surveillance missions. In addition to the new communication software stack, the system provides a user interface for creating commands for transmission to the network nodes. Using the same XML configuration files as the communication stack, it automatically adapts the GUI to the current message configuration, without any need for code changes. Together with a situational display module based on Google Earth, it allows the operator to command and control any number of nodes in a subsea network from a single topside control console, as demonstrated in several major at-sea deployments. The GLINT'08 and '09 experiments (Generic Littoral Interoperable Network Technology) carried out jointly with the NATO Undersea Research Centre represent the most diverse, interoperable node and sensor suite operated under the LAMSS C2 infrastructure, with four different AUVs, several surface craft, and fixed nodes all operating with the same MOOS-IvP payload autonomy.

5.3. GLINT'08 Experiment

As part of a Joint Research Project (JRP) on undersea sensing network technology (NURC project 4G4), MIT in collaboration with the NATO Underwater Research Center (NURC), the Woods Hole Oceanographic Institute (WHOI), the Naval Undersea Warfare Center, Division Newport (NUWC-NPT), and several Italian organizations carried out GLINT'08, a major field demonstration of a hybrid undersea sensing network near the island of Pianosa, Italy (Figure 27). The experiment had several scientific objectives, relating to both the sensing concepts, communication networking, and distributed, autonomous control. The overall objective was to demonstrate the use of a heterogeneous fleet of acoustic sensing platforms, including bottom-mounted arrays and AUVs, in a coordinated fashion for establishing an autonomous, distributed multistatic acoustic network with environmental adaptation capability. The experiment was a joint effort between NURC, NUWC-NPT, WHOI, and MIT, who all contributed with vehicles and sensors.

NURC deployed a cabled communication infrastructure from a C2 center on Pianosa Island. The installation also included a bottom-mounted vertical array for overall acoustic monitoring. NURC also contributed the two research vessels, the NATO research vessel (NRV) alliance and coastal research vessel (CRV) Leonardo, which served

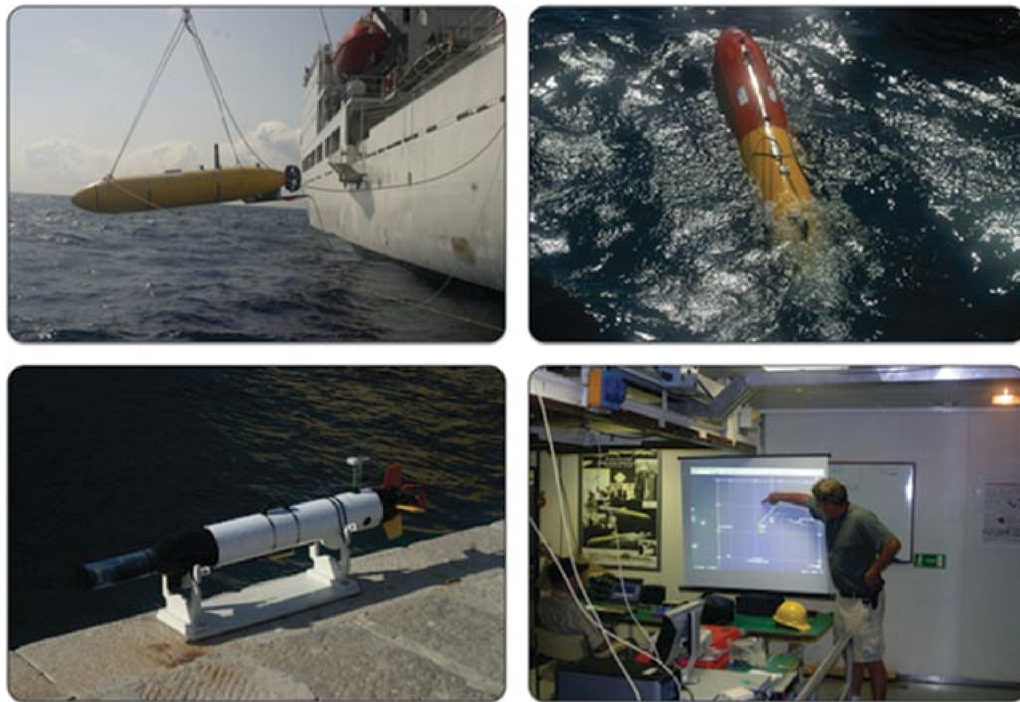

Figure 27. GLINT'08 experiment, Pianosa, Italy, July–August 2008: The upper left frame shows the Unicorn BF21 AUV with towed DURIP array being deployed from NRV Alliance. The upper right frame shows the NURC OEX AUV also being deployed from the NRV Alliance. The bottom left frame shows the NUWC-NPT Ocean Server Iver2 AUV in Pianosa. The bottom right frame shows the C2 center on NRV Alliance with situational display.

as a command and control center, and launch and recovery of the mobile platforms. As part of the collaborative effort, the NURC Ocean Explorer AUV (OEX) was converted to the MOOS-IvP payload autonomy, allowing this vehicle with its towed hydrophone array (SLITA) to execute collaborative maneuvers with the MIT and NUWC-NPT AUVs and commanded via the common C2 infrastructure. The new NURC FOLAGA environmental sampling vehicle was also converted to the MOOS-IvP payload autonomy architecture and used as part of the network.

NUWC-NPT participated with two Iver2 AUVs. These vehicles were primarily used for environmental sampling but also served as part of the multistatic acoustic sampling network, using their integrated, towed hydrophone arrays. WHOI participated with acoustic communication equipment and assisted in establishing the unified C2 infrastructure. The MIT LAMSS participated with one of its Bluefin BF21 vehicles, towing the 32-element DURIP array. In addition, two SCOUT autonomous surface craft were deployed, one for environmental sampling using a CTD winch, the other equipped with a towed acoustic modem used as a mobile communication gateway.

An important objective of GLINT'08 was to demonstrate the communication, command, and control of the hybrid platform suite, using a common communication infrastructure based on the WHOI Micro-Modem and the new D-CCL and a common implementation of the MOOS-IvP payload autonomy in all mobile and fixed assets. The architecture had previously been successfully integrated and demonstrated on the SCOUT kayaks, the Bluefin BF21 AUVs, and several land robots at MIT. In preparation for and during GLINT'08, it was successfully integrated into the NURC OEX AUV and the NUWC-NPT Iver2 AUVs, both deployed in the experiment towing hydrophone arrays for multistatic acoustics. In addition, the architecture was partially integrated into the NURC FOLAGA environmental sampler and two bottom moorings equipped with micromodems for undersea networking. The hybrid network with these assets is shown schematically in Figure 28. In addition to the different autonomous vehicle nodes, virtual, simulated nodes were operated onboard the research vessels, communicating through over-the-side acoustic modems.

The principal scientific objective of GLINT'08 was to collect a comprehensive multistatic active data set using three AUVs with towed hydrophone arrays, which will support the development of robust multistatic active processing approaches suited for operation in the limited computational environment of AUVs. The three vehicles were the NURC OEX with the 48-element SLITA array, the MIT Unicorn BF21 with the 32-element DURIP

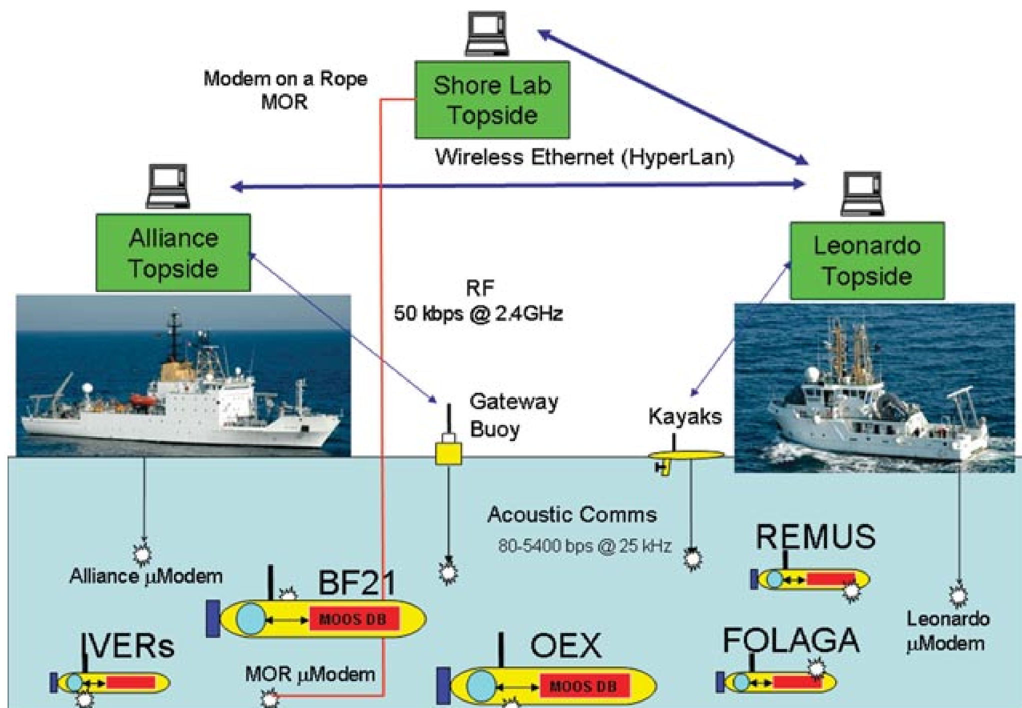

Figure 28. Hybrid undersea acoustic sensing network deployed in GLINT'08 at Pianosa Island, Italy.

array, and the NUWC-NPT Iver2 vehicle towing a 16-element hydrophone array. The two large vehicles, the OEX and Unicorn, had fully integrated MOOS-IvP autonomy systems early in the experiment and were routinely used in coordinated data collection missions. On the last day of the experiment, all three array-towing vehicles were operated together. Also, the MOOS-IvP/D-CCL communication infrastructure allowed several demonstrations to be performed of fully autonomous obstacle and collision avoidance by Unicorn and OEX, as illustrated in Figure 29, showing the topside real-time situational display, which graphically displays all status and contact information transmitted from the vehicles via the undersea communication network.

With the Unicorn BF21 AUV fully integrated with the MOOS-IvP autonomy and D-CCL communication, as developed in past experiments MB06, MINUS07, and PN07, it was operated with fewer personnel than previously required. Significant effort went into streamlining and automating operating procedures so that more effort could be spent focusing on the scientific goals. Thus, once initially deployed, the vehicles are entirely commanded using the D-CCL message set, including redeployments, target prosecute behaviors, and return to base commands.

An additional major accomplishment in GLINT'08 was the development and field demonstration of the new LAMSS D-CCL communication stack with its enhanced report and command structure. Integrated with the MOOS-IvP payload autonomy, it allowed for real-time, interleaved transmission of regular low-bandwidth FSK messages with high-rate PSK coded messages, with up to 2-kbyte messages at 5.4 kbytes/s. The protocol was demonstrated for real-time transmission and display of CTD measurements and array signal processing products such as Beam-Time Records (BTR). It is believed that the real-time topside display of BTR data from an AUV has not previously been achieved in the field. Acoustic communications messages from Unicorn and the other AUVs were assimilated with a heterogeneous mixture of other data sources (AIS, ship's NMEA, etc.) to give a unified situational display available to both the science crew and the ship's captain, as illustrated in Figure 29. The left frame shows the topside display of the network nodes during a mission in which AUV Unicorn is executing a trail behavior, maintaining a fixed relative bearing and range from AUV OEX. The trail behavior is performed solely by forecasting the current OEX position, speed, and heading, based on the most recent status reports received from that vehicle. Such synchronized swimming is crucial to collaborative acoustic sensing, and as demonstrated here, the nested autonomy architecture allows this to happen with neither the operator in the loop nor the requirement of a communication handshake between the two vehicles. If status reports from the OEX go stale, due to communication intermittency, the Unicorn will autonomously enter a loiter pattern awaiting resumption of the communication connectivity. This is an example of the

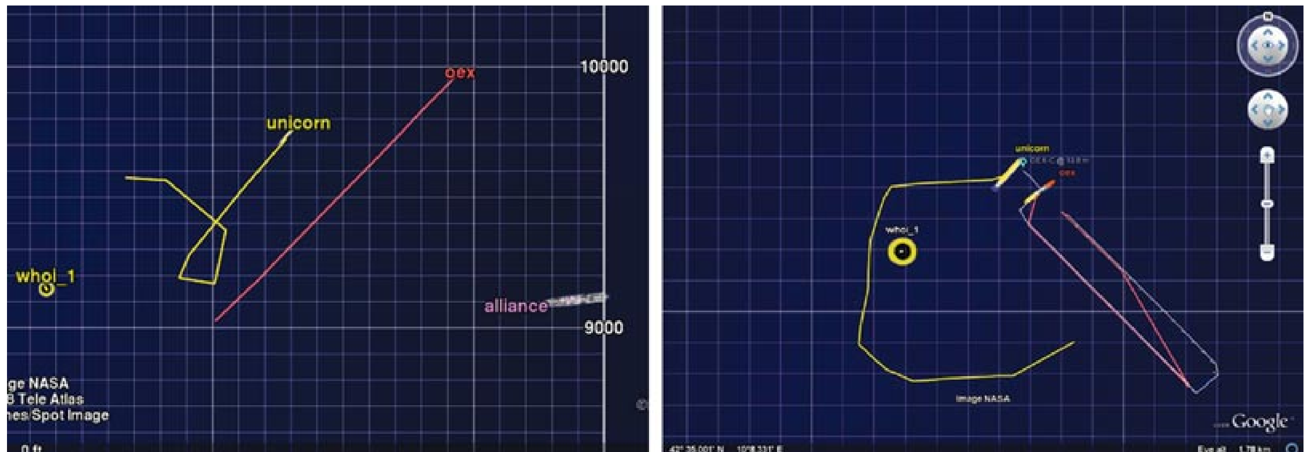

Figure 29. Real-time topside situational display in GLINT'08 command center onboard NRV Alliance. The left frame shows the autonomous trailing by the Unicorn BF21 of the NURC OEX, based solely on predicting the location of the OEX using the received status reports, a behavior first demonstrated in this experiment. The right frame shows the topside rendering of a Unicorn performing its obstacle avoidance and collision avoidance behaviors, with the WHOI Gateway buoy and the OEX AUV, respectively.

robustness of the nested autonomy to communication intermittency. The right frame shows the topside display of the collision avoidance carried out by Unicorn relative to the communication buoy and the OEX, demonstrating the power of the MOOS-IvP autonomy to handle multiobjective missions.

5.4. GLINT'08: Autonomy Results and Lessons Learned

Apart from the comprehensive acoustic data set collected during GLINT'08, an important result was the experience gained in operating a complex heterogeneous network of autonomous platforms with limited and intermittent communication connectivity. The successful deployment and collaborative operation of five to six undersea vehicles with acoustic and nonacoustic sensors were in large part due to the extensive use of a consistent, high-fidelity simulation infrastructure. Thus, before deploying the multiple vehicles together, virtual vehicles were operated on computers onboard the RVs, communicating with the network through over-the-side modems, allowing for minimal-risk testing of advanced collaborative behaviors and strategies.

Another important lesson learned was the criticality of a robust configuration management infrastructure, ensuring the consistency between simulated and real vehicles. Thus, following the experiment a robust configuration management infrastructure was developed by LAMSS, which has subsequently been used in five to six separate field efforts, eliminating many of the sources of vehicle mishaps experienced in earlier experiments and attributed to simple human errors.

Finally, the experiment showed the need for a rigorous state space management system for the autonomous vehicles to eliminate unintended conflicts and mishaps associated with the IvP behavior suite. This experience led to the subsequent addition to the IvP-Helm of a HMD framework for the autonomy mode management. This development was another critical factor in eliminating experimental mishaps in the five to six major field deployments that LAMSS has been involved in. Figure 30 shows the HMD representation of the modes applied by the AUVs operated in GLINT'08.

Once launched, the vehicle is in the Undefined mode. Once the IvP-Helm starts up, the vessel will initially be in the AllStop mode, in which no behaviors are active. Once issued a command, the vehicle will enter the Mission mode. If the command is a deploy command, the vehicle will enter this mode. The active behavior set includes behaviors that are active in all deploy submodes, including behaviors for vehicle safety, such as obstacle and collision avoidance, and some that are specific to a particular submode. Thus, the Loiter mode will place the vehicle in a hexagonal loiter pattern at a specified location at constant speed and depth, where it will stay until it is issued a new command. Alternatively the deployment mode may be a Racetrack with various depth behaviors, including constant depth, a fixed vertical yo-yo, and an adaptive depth behavior used for tracking the thermocline. Another deploy mode, which was first demonstrated in this experiment, was the Trail mode, in which one vehicle is trailing another at a specified bearing and range, using solely a prediction of the current location, heading, and speed based on the previously received status report from the trailed vehicle.

6. SUMMARY

In this paper we have described two paradigms and two architectures and descriptions of their fielded implementations. The *payload autonomy* paradigm includes (a) the

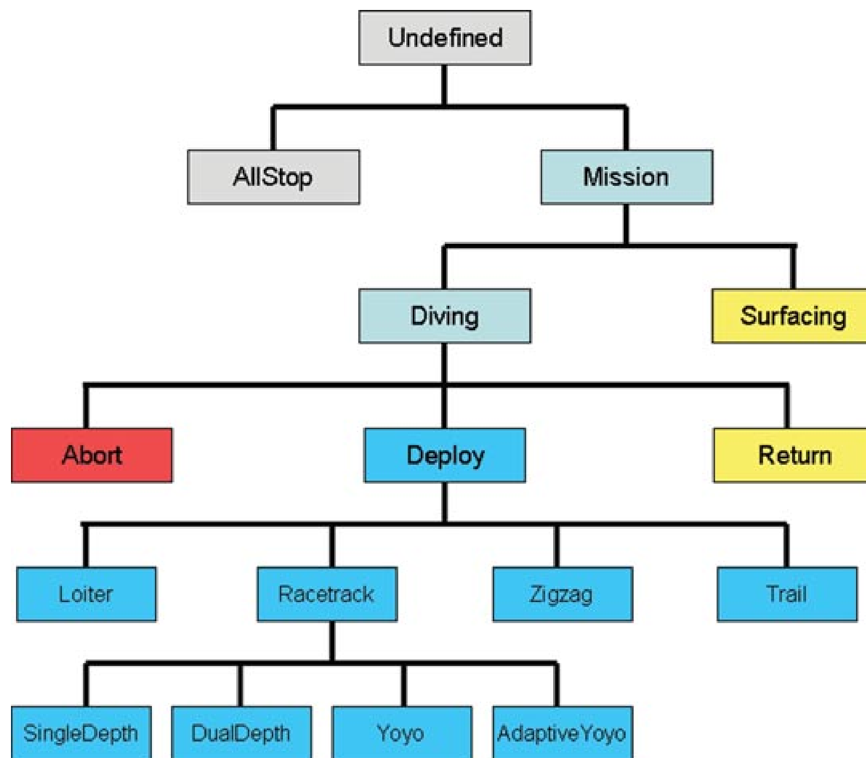

Figure 30. GLINT'08 autonomy HMDs: Each AUV node in the nested autonomy system deployed at GLINT'08 exercises had the shown HMD representation. Each mode mapped directly to a distinct set of IvP Helm behaviors that were running in each mode. The ACOMMS message set configured for this fielded system of AUVs was closely coupled with the declared nodes to facilitate both command and control messages passed down to platforms from field operators to change between autonomy modes, as well as status messages passed up from platforms to facilitate the understanding of the tactical picture to field operators.

separation of autonomy sensing and decision making from the problem of vehicle control and navigation, (b) the MOOS publish–subscribe middleware for allowing independent development of sensing, communication, and autonomous decision-making software, and (c) the behavior-based IvP-Helm for independent development of autonomy modules. The *nested autonomy* paradigm is an approach for implementing a system of unmanned platforms for large-scale, long-endurance, autonomous sensing applications. It exploits the platform-independent payload autonomy paradigm to field a network of heterogeneous nodes to address the limitations of unpredictable, environmentally dependent sensing and low-bandwidth communications in the underwater domain.

In this paper two architectures were described in detail. The MOOS publish–subscribe middleware is both an architecture and a mechanism for interprocess communication and process scheduling but also, as an open-source software project, a collection of substantial applications for sensing, communications, autonomy, debugging, and post-mission analysis. The IvP-Helm is a behavior-based architecture, unique in its use of the IvP model for multiobjective optimization for resolving competing autonomy behaviors. It is also an open-source project that includes many well-tested vehicle behaviors and autonomy tools for creating, debugging, and analyzing autonomy capabilities.

ACKNOWLEDGMENTS

The prototype of MOOS was developed by Paul Newman at MIT under the GOATS'2000 NURC Joint Research Program, with Office of Naval Research (ONR) support from Grant N-00014-97-1-0202 (Program Managers Tom Curtin, Jeff Simmen, Tom Swean, and Randy Jacobson). The development of the nested autonomy concept for environmental acoustic sensing and the MIT component of the GLINT'08 experiment was funded by the ONR under the GOATS program, Grant N-00014-08-1-0013 (Program Manager Ellen Livingston). The development of the unified communication, command, and control infrastructure and the execution of the SWAMSI09 experiment were supported by ONR, Grant N-00014-08-1-0011 (Program Manager Bob Headrick). The IvP-Helm autonomy software and the basic research involved in the interval programming model for

multiobjective optimization were developed under support from ONR Code 311 (Program Managers Don Wagner and Behzad Kamgar-Parsi). Prior prototype development of IvP concepts benefited from the support of the In-house Laboratory Independent Research (ILIR) program at the Naval Undersea Warfare Center in Newport, Rhode Island. The NATO Undersea Research Centre (NURC) has supported the development of the MOOS-IvP nested autonomy concept by conducting seven major field experiments, in which MIT LAMSS has been a partner, including GOATS'2000 and '2002, FAF'2003, FAF'2005, CCLNet'08, GLINT'08, and GLINT'09. Without the world-class seagoing experiment capabilities of NURC, with its state-of-the-art RVs, NRV Alliance and CRV Leonardo, and their outstanding crew, and NURC's excellent engineering and logistics support, the nested autonomy concept and the underlying MOOS-IvP software base would not have reached the level of sophistication and robustness that it has achieved.

REFERENCES

Arkin, R. C. (1987, March). Motor schema based navigation for a mobile robot: An approach to programming by behavior. In Proceedings of the IEEE Conference on Robotics and Automation, Raleigh, NC (pp. 264–271).

Arkin, R. C., Carter, W. M., & Mackenzie, D. C. (1993). Active avoidance: Escape and dodging behaviors for reactive control. International Journal of Pattern Recognition and Artificial Intelligence, 5(1), 175–192.

Benjamin, M., Battle, D., Eickstedt, D., Schmidt, H., & Balasuriya, A. (2007, April). Autonomous control of an unmanned underwater vehicle towing a vector sensor array. In International Conference on Robotics and Automation (ICRA), Rome, Italy.

Benjamin, M., Schmidt, H., & Leonard, J. J. (n.d.). MOOS-IvP. Accessed September 2010 at http://www.moos-ivp.org.

Benjamin, M. R. (2004). The interval programming model for multi-objective decision making (Tech. Rep. AIM-2004-021). Cambridge, MA: Computer Science and Artificial Intelligence Laboratory, MIT.

Benjamin, M. R. (2010). MOOS-IvP autonomy tools users manual (Tech. Rep. MIT-CSAIL-TR-2010-039). Cambridge, MA: Computer Science and Artificial Intelligence Laboratory, MIT.

Benjamin, M. R., Newman, P. M., Schmidt, H., & Leonard, J. J. (2009). A tour of MOOS-IvP autonomy software modules (Tech. Rep. MIT-CSAIL-TR-2009-006). Cambridge, MA: Computer Science and Artificial Intelligence Laboratory, MIT.

Benjamin, M. R., Newman, P. M., Schmidt, H., & Leonard, J. J. (2010). An overview of MOOS-IvP and a users guide to the IvP Helm autonomy software (Tech. Rep. MIT-CSAIL-TR-2010-041). Cambridge, MA: Computer Science and Artificial Intelligence Laboratory, MIT.

Bennet, A. A., & Leonard, J. J. (2000). A behavior-based approach to adaptive feature detection and following with autonomous underwater vehicles. IEEE Journal of Oceanic Engineering, 25(2), 213–226.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. IEEE Journal of Robotics and Automation, 2(1), 14–23.

Carreras, M., Batlle, J., & Ridao, P. (2000, May). Reactive control of an AUV using motor schemas. In International Conference on Quality Control, Automation and Robotics, Cluj Napoca, Romania.

Dantzig, G. B. (1948). Programming in a linear structure. Comptroller, U.S. Air Force.

Khatib, O. (1985, March). Real-time obstacle avoidance for manipulators and mobile robots. In Proceedings of the IEEE International Conference on Robotics and Automation, St. Louis, MO (pp. 500–505).

Kumar, R., & Stover, J. A. (2001). A behavior-based intelligent control architecture with application to coordination of multiple underwater vehicles. IEEE Transactions on Systems, Man, and Cybernetics—Part A: Cybernetics, 30(6), 767–784.

Newman, P. (n.d.). MOOS. Accessed September 2010 at http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php.

Newman, P. M. (2003). MOOS—A mission oriented operating suite (Tech. Rep. OE2003-07). Cambridge, MA: Department of Ocean Engineering, MIT.

Pirjanian, P. (1998). Multiple objective action selection and behavior fusion. P.hD. thesis, Aalborg University.

Riekki, J. (1999). Reactive task execution of a mobile robot. P.hD. thesis, Oulu University.

Rosenblatt, J. K. (1997). DAMN: A distributed architecture for mobile navigation. P.hD. thesis, Carnegie Mellon University, Pittsburgh, PA.

Rosenblatt, J. K., Williams, S. B., & Durrant-Whyte, H. (2002). Behavior-based control for autonomous underwater exploration. International Journal of Information Sciences, 145(1–2), 69–87.

Schneider, T., & Schmidt, H. (2010a, May). The dynamic compact control language: A compact marshalling scheme for acoustic communications. In Proceedings of the IEEE Oceans Conference 2010, Sydney, Australia.

Schneider, T., & Schmidt, H. (2010b). Unified command and control for heterogeneous marine sensing networks. Journal of Field Robotics, 27(6), 876–889.

Williams, S. B., Newman, P., Dissanayake, G., Rosenblatt, J. K., & Durrant-Whyte, H. (2000). A decoupled, distributed AUV control architecture. In Proceedings of 31st International Symposium on Robotics, Montreal, Canada (pp. 246–251).