An Overview of MOOS-IvP and a Users Guide to the IvP Helm - Release 24.8

Michael R. Benjamin[1], Paul Newman[2]

[1]Department Mechanical Engineering
Massachusetts Institute of Technology, Cambridge MA

[2]Department of Engineering Science
University of Oxford, Oxford England

August, 2024 - Release 24.8

Last Updated: March 23, 2025

Abstract

This document describes the IvP Helm - an Open Source behavior-based autonomy application for unmanned vehicles. IvP is short for interval programming - a technique for representing and solving multi-objective optimizations problems. Behaviors in the IvP Helm are reconciled using multi-objective optimization when in competition with each other for influence of the vehicle. The IvP Helm is written as a MOOS application where MOOS is a set of Open Source publish-subscribe autonomy middleware tools. This document describes the configuration and use of the IvP Helm, provides examples of simple missions and information on how to download and build the software from the MOOS-IvP server at www.moos-ivp.org.

This work is the product of a multi-year collaboration between the Marine Autonomy Lab in the Department of Mechanical Engineering at Massachusetts Institute of Technology in Cambridge Massachusetts, and the Oxford University Mobile Robotics Group.

Points of contact for collaborators:

Dr. Michael R. Benjamin
Department of Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Intitute of Technology
mikerb@csail.mit.edu

Dr. Paul Newman
Department of Engineering Science
University of Oxford
pnewman@robots.ox.ac.uk

Other collaborators have contributed greatly to the development and testing of software and ideas within, notably - Henrik Schmidt, Toby Schneider, Stephanie Kemna, Arjan Vermeij, David Battle, Christian Convey, Chris Gagner.

Sponsorship:

This work is sponsored by Dr. Behzad Kamgar-Parsi and Dr. Don Wagner of the Office of Naval Research (ONR), Code 311. Further support for testing and coursework development sponsored by Battelle, Dr. Robert Carnes.

Contents

1 Overview of the MOOS-IvP Autonomy Project

1.1 Brief Background of MOOS-IvP

MOOS was written by Paul Newman in 2001 to support operations with autonomous marine vehicles in the MIT Ocean Engineering and the MIT Sea Grant programs. At the time Newman was a post-doc working with John Leonard and has since joined the faculty of the Mobile Robotics Group at Oxford University. MOOS continues to be developed and maintained by Newman at Oxford and the most current version can be found at themoos.org. The MOOS software available in the MOOS-IvP project includes a snapshot of the MOOS code distributed from Oxford. The IvP Helm was developed in 2004 for autonomous control on unmanned marine surface craft, and later underwater platforms. It was written by Mike Benjamin as a post-doc working with John Leonard, and as a research scientist for the Naval Undersea Warfare Center in Newport Rhode Island. The IvP Helm is a single MOOS process that uses multi-objective optimization to implement behavior coordination.

Acronyms

MOOS stands for "Mission Oriented Operating Suite" and its original use was for the Bluefin Odyssey III vehicle owned by MIT. IvP stands for "Interval Programming" which is a mathematical programming model for multi-objective optimization. In the IvP model each objective function is a piecewise linear construct where each piece is an *interval* in N-Space. The IvP model and algorithms are included in the IvP Helm software as the method for representing and reconciling the output of helm behaviors. The term interval programming was inspired by the mathematical programming models of linear programming (LP) and integer programming (IP). The pseudo-acronym IvP was chosen simply in this spirit and to avoid acronym clashing.

1.2 Sponsors of MOOS-IvP

Original development of MOOS and IvP were more or less infrastructure by-products of other sponsored research in (mostly marine) robotics. Those sponsors were primarily The Office of Naval Research (ONR), as well as the National Oceanic and Atmospheric Administration (NOAA). MOOS and IvP are currently funded by Code 311 at ONR, Dr. Don Wagner and Dr. Behzad Kamgar-Parsi. Testing and development of course work at MIT is further supported by Battelle, Mr. Mike Mellott. The Battelle sponsorship has been very instrumental in the development of the documentation and online course material. MOOS is additionally supported in the U.K. by EPSRC. Early development of IvP benefited from the support of the In-house Laboratory Independent Research (ILIR) program at the Naval Undersea Warfare Center in Newport RI. The ILIR program is funded by ONR.

1.3 Where to Get Further Information

1.3.1 Websites and Email Lists

There are two web sites - the MOOS web site maintained by Oxford University, and the MOOS-IvP web site maintained by MIT. At the time of this writing they are at the following URLs:

- www.themoos.org

- www.moos-ivp.org

What is the difference in content between the two web sites? As discussed previously, MOOS-IvP, as a set of software, refers to the software maintained and distributed from Oxford *plus* additional MOOS applications including the IvP Helm and library of behaviors. The software bundle released at moos-ivp.org does include the MOOS software from Oxford - usually a particular released version. For the absolute latest in the core MOOS software and documentation on Oxford MOOS modules, the Oxford web site is your source. For the latest on the core IvP Helm, behaviors, and MOOS tools distributed by MIT, the moos-ivp.org web site is the source.

There are two mailing lists open to the public. The first list is for MOOS users, and the second is for MOOS-IvP users. If the topic is related to one of the MOOS modules distributed from the Oxford web site, the proper email list is the "moosusers" mailing list. You can join the "moosusers" mailing list at the following URL:

https://lists.csail.mit.edu/mailman/listinfo/moosusers

For topics related to the IvP Helm or modules distributed on the moos-ivp.org web site that are not part of the Oxford MOOS distribution (see the software page on moos-ivp.org for help in drawing the distinction), the "moosivp" mailing list is appropriate. You can join the "moosivp" mailing list at the following URL:

https://lists.csail.mit.edu/mailman/listinfo/moosivp

1.3.2 Documentation

Documentation on MOOS can be found on the MOOS web site:

www.themoos.org

This includes documentation on the MOOS architecture, programming new MOOS applications as well as documentation on several bread-and-butter applications such as `pAntler`, `pLogger`, `uMS`, `pShare`, `iRemote`, `iMatlab`, `pScheduler` and more. Documentation on the IvP Helm, behaviors and autonomy related MOOS applications not from Oxford can be found on the www.moos-ivp.org web site under the Documentation link.

2 Design Considerations of MOOS-IvP

The primary motivation in the design of MOOS-IvP is to build highly capable autonomous systems. Part of this picture includes doing so at a reduced short and long-term cost and a reduced time line. By "design" we mean both the choice in architectures and algorithms as well as the choice to make key modules for infrastructure, basic autonomy and advanced tools available to the public under an Open Source license. The MOOS-IvP software design is based on three architecture philosophies, (a) the backseat driver paradigm, (b) publish and subscribe autonomy middleware, and (c) behavior based autonomy. The common thread is the ability to separate the development of software for an overall system into distinct modules coordinated by infrastructure software made available to the public domain.

2.1 Public Infrastructure - Layered Capabilities

The central architecture idea of both MOOS and IvP is the separation of overall capability into separate and distinct modules. The unique contributions of MOOS and IvP are the methods used to *coordinate* those modules. A second central idea is the decision to make algorithms and software modules for infrastructure, basic autonomy and advanced tools available to the public under an Open Source license. The idea is pictured in Figure 1. There are three things in this picture - (a) modules that actually perform a function (the wedges), (b) modules that coordinate other modules (the center of the wheel), and (c) standard wrapper software use by each module to allow it to be coordinated (the spokes).

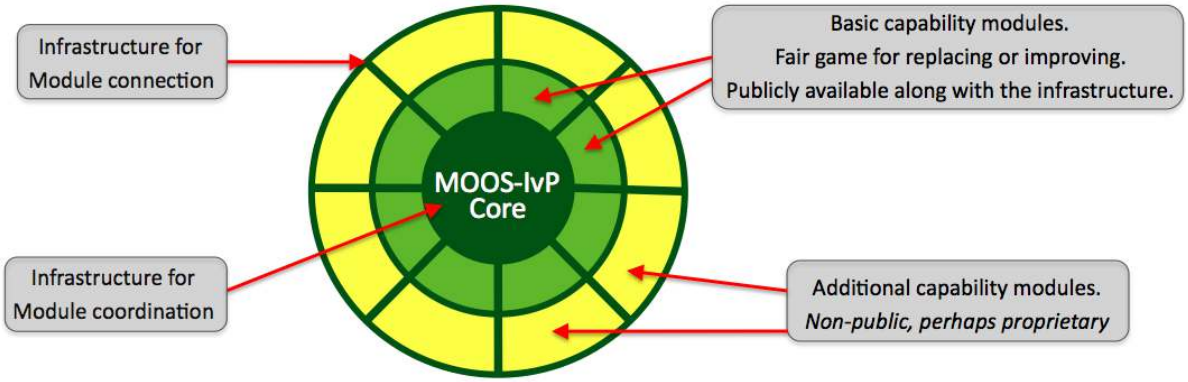

Figure 1: **Public Infrastructure - Layered Capabilities:** The center of the wheel represents MOOS-IvP Core. For MOOS this means the MOOSDB and the message passing and scheduling algorithms. For IvP this means the IvP helm behavior management and the multi-objective optimization solver. The wedges on the wheel represent individual modules - either MOOS processes or IvP behaviors. The spokes of the wheel represent the idea that each module inherits from a superclass to grab functionality key to plugging into the core. Each wedge or module contains a wrapper defined by the superclass that augments the function of the individual module. The darker wedges indicate publicly available modules and the lighter ones are modules added by users to augment the public set to comprise a particular fielded autonomy system.

The darker wedges in Figure 1 represent application modules (not infrastructure) that provide basic functionality and are publicly available. However, they do not hold any special immutable status. They can be replaced with a better version, or, since the source code is available, the code of

the existing module can be changed or augmented to provide a better or different version (hopefully with a different name - see the section on branching below). Later sections provide an overview of about 40 or so particular modules that are currently available. By modules we mean MOOS applications and IvP behaviors and the above comments hold in either case. The yellow wedges in Figure 1 represent the imaginable unimplemented modules or functionality. A particular fielded MOOS-IvP autonomy system typically is comprised of (a) the MOOS-IvP core modules, (b) *some* of the publicly available MOOS applications and IvP behaviors, and (c) additional perhaps non-public MOOS applications and IvP behaviors provided by one or more third party developers.

The objective of the public-infrastructure/layered-capabilities idea is to strike an important balance - the balance between effective code re-use and the need for users to retain privacy regarding how they choose to augment the public codebase with modules of their own to realize a particular autonomy system. The benefits of code re-use are an important motivation in fundamental architecture decisions in both MOOS and IvP. The modules that comprise the public MOOS-IvP codebase described in this document represent over twenty work-years of development effort. Furthermore, certain core components of the codebase have had hundreds if not thousands of hours of usage on a dozen or so fielded platform types in a variety of situations. The issue of code re-use is discussed next.

2.2 Code Re-Use

Code re-use is critical, and starts with the ability to have a system comprised of separate but coordinated modules. They key technical hurdle is to achieve module separation without invoking a substantial hit on performance. In short, MOOS middleware is a way of coordinating separate processes running on a single computer or over several networked computers. IvP is a way of coordinating several autonomy behaviors running within a single MOOS process.

Factors Contributing to Code Re-use:

- *Freedom from proprietary issues.* Software serving as infrastructure shared by all components (MOOS processes and IvP behaviors) are available under an Open Source license. In addition many mature MOOS and IvP modules providing commonly needed capabilities are also publicly available. Proprietary or non-publicly released code may certainly co-exist with non-proprietary public code to comprise a larger autonomy system. Such a system would retain a strategic edge over competitors if desired, but have a subset of components common with other users.

- *Module independence.* Maintaining or augmenting a system comprised of a set of distinct modules can begin to break down if modules are not independent with simple easy-to-augment interfaces. Compile dependencies between modules need to be minimized or eliminated. The maintenance of core software libraries and application code should be decoupled completely from the issues of 3rd party additional code.

- *Simple well-documented interfaces.* The effort required to add modules to the code base should be minimized. Documentation is needed for both (a) using the publicly available applications and libraries, and (b) guiding users in adding their own modules.

- *Freedom to innovate.* The infrastructure does not put undue restrictions on how basic problems

can be solved. The infrastructure remains agnostic to techniques and algorithms used in the modules. No module is sacred and any module may be replaced.

Benefits of Code Re-Use:

- *Diversity of contributors.* Increasingly, an autonomy system contains many components that touch many areas of expertise. This would be true even for a vanilla use of a vehicle, but is compounded when considering the variety of sensors and missions and ways of exploiting sensors in achieving mission objectives. A system that allows for wide code re-use is also a system that allows module contributions from a wide set of developers or experts. This has a substantial impact on the issues mentioned below of lower cost, higher quality and reliability, and reduced development time line.

- *Lower cost.* One immediate benefit of code re-use is the avoidance of repeatedly re-inventing modules. A group can build capabilities incrementally and experts are free to concentrate on their area and develop only the modules that reflect their skill set and interests. Perhaps more important, code re-use gives the systems integrator *choices* in building a complete system from individual modules. Having choices leads to increased leverage in bargaining for favorable licensing terms or even non-proprietary terms for a new module. Favorable licensing terms arranged at the outset can lead to substantially lower long-term costs for future code maintenance or augmentation of software.

- *Higher performance capability.* Code re-use enhances performance capability in two ways. First, since experts are free to be experts without re-inventing the modules outside their expertise and provided by others, their own work is more likely to be more focused and efficient. They are likely to achieve a higher capability for a given a finite investment and given finite performance time. Second, since code re-use gives a systems integrator *choices*, this creates a meritocracy based on optimal performance-cost ratio of candidate software modules. The under-capable, more expensive module is less likely to diminish the overall autonomy capability if an alternative module is developed to offer a competitive choice. Survival of the fittest.

- *Higher performance reliability.* An important part of system reliability is testing. The more testing time and the greater diversity of testing scenarios the better. And of course the more time spent testing on physical vehicles versus simulation the better. By making core components of a codebase public and permitting re-use by a community of users, that community provides back an enormous service by simply using the software and complaining when or if something goes wrong. Certain core components of the MOOS-IvP codebase have had hundreds if not thousands of hours of usage on a dozen or so platform types in a variety of situations. And many more hours in simulation. Testing doesn't replace good coding practice or formal methods for testing and verifying correctness, but it complements those two aspects and is enhanced by code re-use.

- *Reduced development time line.* Code re-use means less code is being re-developed which leads to quicker overall system development. More subtly, since code re-use can provide a systems integrator choices and competition on individual modules, development time can be reduced as a consequent. An integrator may simply accept the module developed the quickest,

or the competition itself may speed up development. If choices and competition result in more favorable license agreements between the integrator and developer, this in itself may streamline agreements for code maintenance and augmentation in the long term. Finally, as discussed above, if code re-use leads to an element of community-driven bug testing, this will also quicken the pace in the evolution toward a mature and reliable autonomy system.

2.3 The Backseat Driver Design Philosophy

The key 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 from the main vehicle computer, and accepting a stream of autonomy decisions such as heading, speed and depth in return from the payload computer. 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.

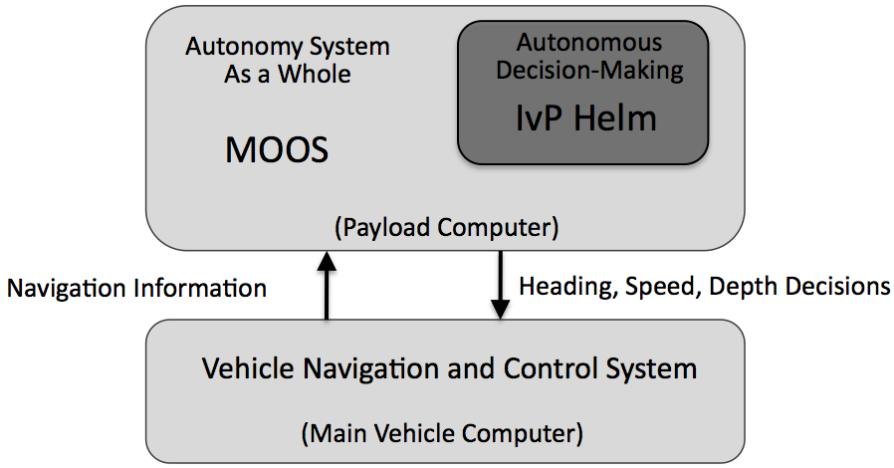

Figure 2: **The backseat driver paradigm**: The key idea is the separation of vehicle autonomy from vehicle control. The autonomy system provides heading, speed and depth commands to the vehicle control system. The vehicle control system executes the control and passes navigation information, e.g., position, 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.

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" since MOOS itself is middleware, and actual autonomous decision making, sensor processing, contact management

etc., are implemented as individual MOOS processes.

2.4 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 each other through a single database process in a star topology (Figure 3). The interface of a particular process is described by what messages it produces (publications) and what messages it consumes (subscriptions). Each message is a simple variable-value pair where the values are typically either string or numerical values such as (`STATE`, `"DEPLOY"`), or (`NAV_SPEED`, `2.2`). MOOS messages may also contain raw binary data for passing images for example.

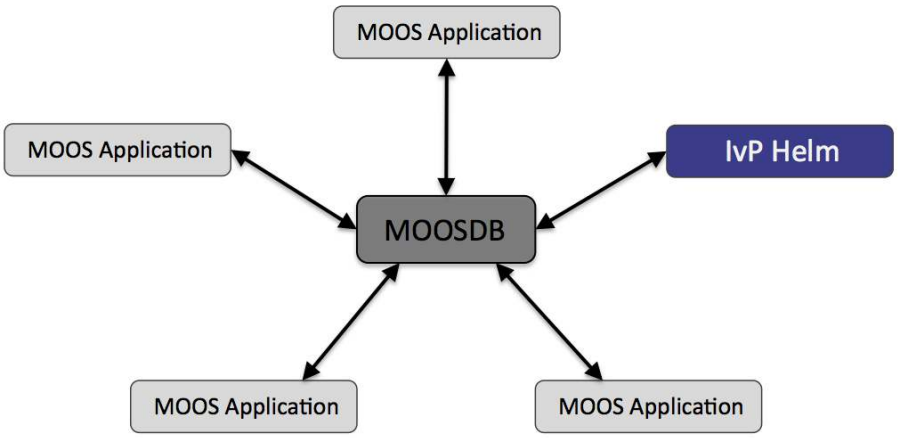

Figure 3: **A MOOS community**: is a collection of MOOS applications typically running on a single machine each with a separate process ID. Each process communicates through a single MOOS database process (the MOOSDB) in a publish-subscribe manner. Each process may be executing its inner-loop at a frequency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

The key idea with respect to facilitating code re-use 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. Since MOOS Core and many common applications are publicly available along with source code under an Open Source GPL license, a user may develop an improved module by altering existing source code and introduce a new version under a different name. With MOOS-IvP 13.2 which includes MOOS V10, the MOOS libraries are distributed under an LGPL license, to allow the development and use of commercial MOOS applications alongside open source applications. The MOOSDB and MOOS applications remain under an GPL license. 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.

2.5 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 exposes 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, vehicle speed, and so on. It also contains a specification of state spaces that determine which behaviors are active under what 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).

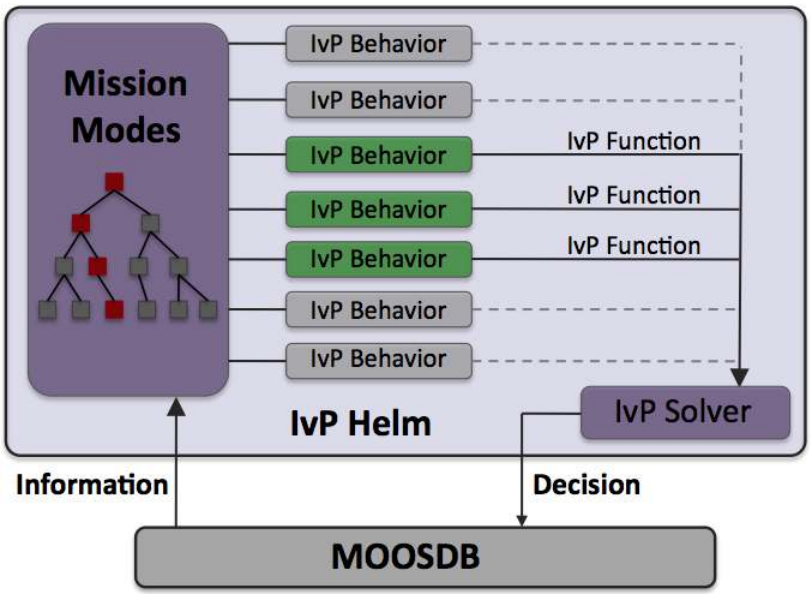

Figure 4: **The IvP Helm**: The helm is a single MOOS application running as the process pHelmIvP. It is a behavior-based architecture where the primary output of a behavior on each iteration is an IvP objective function. The IvP solver performs multi-objective optimization on the set of functions to 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 are active at any given time depending on the vehicle situation, and the state space configuration provided by the user.

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 comprised of the weighted sum of contributing functions.

The concept of a behavior-based architecture is often attributed to [**?**]. 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 a way that the highest priority behavior locks out all others as in the Subsumption Architecture in [**?**]. Another approach is referred to as the potential fields, or vector summation approach

(See [**?**], [**?**]) 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., [**?**], [**?**], [**?**], [**?**], land vehicles, e.g., [**?**], and marine vehicles, e.g., [**?**], [**?**], [**?**], [**?**], [**?**]. However, action-selection via the identification of a single highest priority behavior and via vector summation have well known shortcomings later described in [**?**], [**?**] and [**?**] in which the authors advocated for the use of multi-objective optimization as a more suitable, although more computationally expensive, method for action selection. The IvP model is a method for implementing multi-objective function based action-selection that is computationally viable in the IvP Helm implementation.

3 A First Example with MOOS-IvP - the Alpha Mission

This section describes a simple mission using the helm. It is designed to run in simulation on a single machine. The mission configuraiton files for this example are distributed with the source code. Information on how to find these files and launch this mission are described below in Section 16.1. In this example the vehicle simply traverses a set of pre-defined given waypoints and returns back to the launch position. The user may return the vehicle any time before completing the waypoints, and may subsequently command the vehicle to resume the waypoints at any time. This example touches on the following issues:

- Launching a mission with a given mission (`.moos`) file and behavior (`.bhv`) file.
- Configuration of MOOS processes, including the IvP Helm, with a `.moos` file.
- Configuration of the IvP Helm (mission planning) with a `.bhv` file.
- Implementation of simple command and control with the IvP Helm.
- Interaction between MOOS processes and the helm during normal mission operation.

Here is a bit of what the Alpha mission should look like:

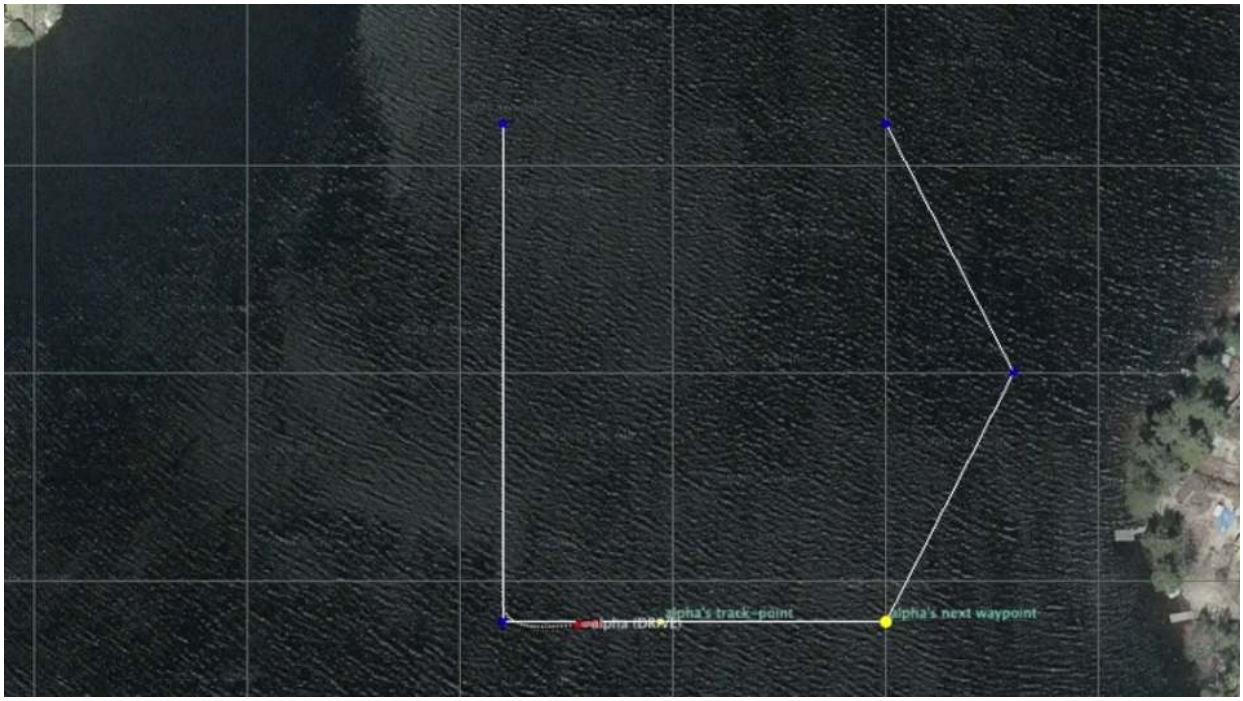

Figure 5: The Alpha mission.

video:(0:19): https://vimeo.com/84549446

3.1 Find and Launch the Alpha Example Mission

The example mission should be in the same directory tree containing the source code. There are two files - a MOOS file, also mission file or `.moos` file, and a behavior file or `.bhv` file:

```
moos-ivp/
   MOOS/
   ivp/
      missions/
         s1_alpha/
            alpha.moos    <---- The MOOS file
            alpha.bhv     <---- The Behavior file
```

To run this mission from a terminal window, simply change directories and launch:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ pAntler alpha.moos
```

After pAntler has launched each process, the pMarineViewer window should be open and look similar to that shown in Figure 59. After clicking the DEPLOY button in the lower right corner the vehicle should start to traverse the shown set of waypoints.

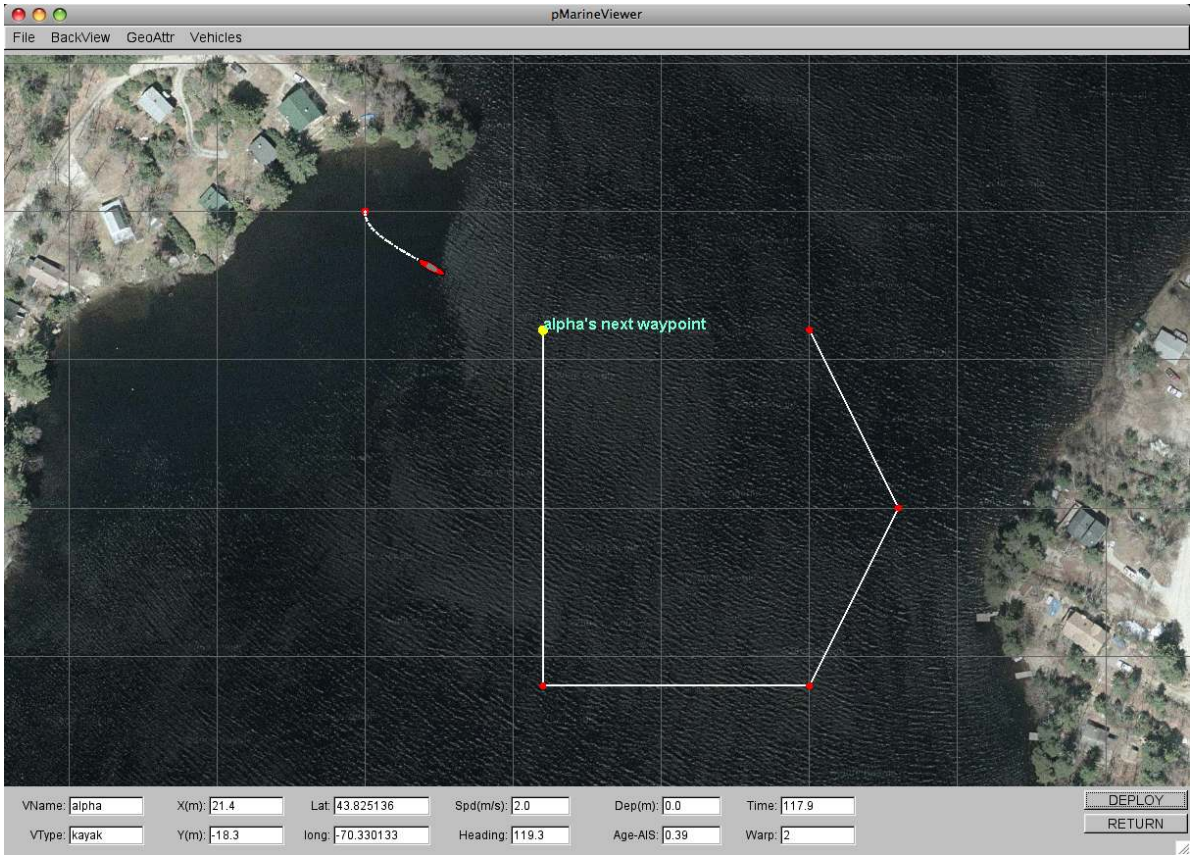

Figure 6: **The Alpha Example Mission - In the Surveying Mode**: A single vehicle is dispatched to traverse a set of waypoints and, upon completion, traverse to the waypoint (0,0) which is the launch point.

This mission will complete on its own with the vehicle returning to the launch point. Alternatively, by hitting the `RETURN` button at any time before the points have been traverse, the vehicle will change course immediately to return to the launch point, as shown in Figure 60. When the vehicle is returning as in the figure, it can be re-deployed by hitting the `DEPLOY` button again.

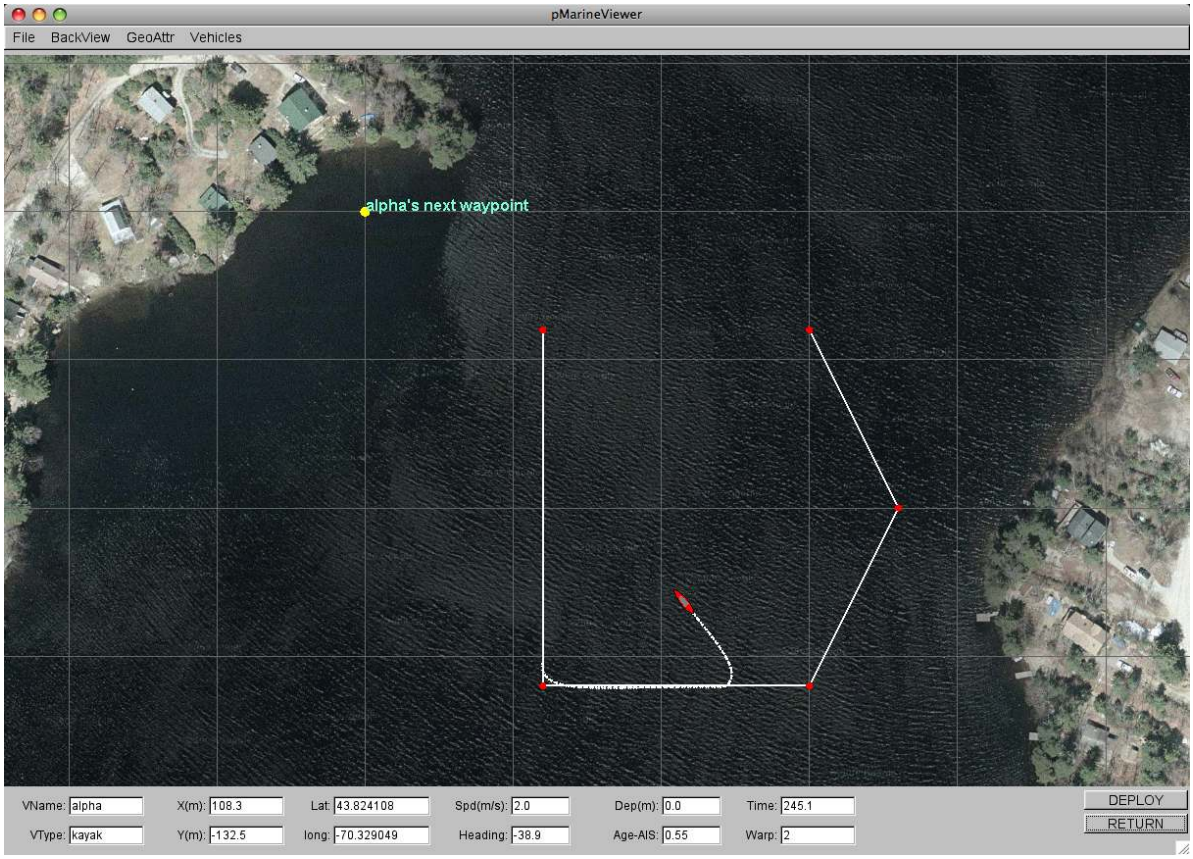

Figure 7: **The Alpha Example Mission - In the Returning Mode**': The vehicle can be commanded to return prior to the completion of its waypoints by the user clicking the `RETURN` button on the viewer.

The vehicle in this example is configured with two basic waypoint behaviors. Their configuration with respect to the points traversed and when each behavior is actively influencing the vehicle, is discussed next.

3.2 A Look at the Behavior File used in the Alpha Example Mission

The mission configuration of the helm behaviors is provided in a *behavior file*, and the complete behavior file for the example mission is shown in Listing 1. Behaviors are configured in blocks of parameter-value pairs - for example lines 6-17 configure the waypoint behavior with the five waypoints shown in the previous two figures. This is discussed in more detail in Section **??**.

Listing 3.1: The behavior file for the Alpha example.

```
1  initialize   DEPLOY = false
2  initialize   RETURN = false
3
4  //------------------------------------------------
5  Behavior = BHV_Waypoint
6  {
7    name      = waypt_survey
8    pwt       = 100
```

```
 9    condition = RETURN = false
10    condition = DEPLOY = true
11    endflag   = RETURN = true
12    perpetual = true
13
14            lead = 8
15     lead_damper = 1
16           speed = 2.0   // meters per second
17          radius = 4.0
18       nm_radius = 10.0
19          points = 60,-40:60,-160:150,-160:180,-100:150,-40
20          repeat = 1
21  }
22
23  //-----------------------------------------------
24  Behavior = BHV_Waypoint
25  {
26    name       = waypt_return
27    pwt        = 100
28    condition  = RETURN = true
29    condition  = DEPLOY = true
30    perpetual  = true
31    endflag    = RETURN = false
32    endflag    = DEPLOY = false
33
34         speed = 2.0
35        radius = 2.0
36     nm_radius = 8.0
37         point = 0,0
38  }
```

The parameters for each behavior are separated into two groups. Parameters such as `name`, `priority`, `condition` and `endflag` are parameters defined generally for all IvP behaviors. Parameters such as `speed`, `radius`, and `points` are defined specifically for the Waypoint behavior. A convention used in `.bhv` files is to group the general behavior parameters separately at the top of the configuration block.

In this mission, the vehicle follows two sets of waypoints in succession by configuring two instances of a basic waypoint behavior. The second waypoint behavior (lines 23-37) contains only a single waypoint representing the vehicle launch point (0,0). It's often convenient to have the vehicle return home when the mission is completed - in this case when the first waypoint behavior has reached its last waypoint. Although it's possible to simply add (0,0) as the last waypoint of the first waypoint behavior, it is useful to keep it separate to facilitate recalling the vehicle pre-maturely at any point after deployment.

Behavior conditions (lines 9-10, 28-29), and endflags (line 110, lines 31-32) are primary tools for coordinating separate behaviors into a particular mission. Behaviors will not participate unless each of its conditions are met. The condtions are based on current values of the MOOS variables involved in the condition. For example, both behaviors will remain idle unless the variable `DEPLOY` is set to true. This variable is set initially to be false by the initialization on line 1, and is toggled by the `DEPLOY` button on the `pMarineViewer` GUI shown in Figures 59 and 60. The `pMarineViewer` MOOS application is one option for a command and control interface to the helm. The MOOS variables in the behavior conditions in Listing 1 do not care which process was responsible for setting the value. Endflags are used by behaviors to post a MOOS variable and value when a behavior has reached a completion. The notion of completion is different for each behavior and some behaviors have no

notion of completion, but in the case of the waypoint behavior, completion is declared when the last waypoint is reached. In this way, behaviors can be configured to run in a sequence, as in this example, where the returning waypoint behavior will have a necessary condition (line 28) met when the surveying behavior posts its endflag on line 11.

3.3 A Closer Look at the MOOS Apps in the Alpha Example Mission

Running the example mission involves five other MOOS applications in addition to the IvP helm. In this section we take a closer look at what those applications do and how they are configured. The full MOOS file, `alpha.moos`, used to run this mission is given in full in the appendix. An overview of the situation is shown in Figure 61.

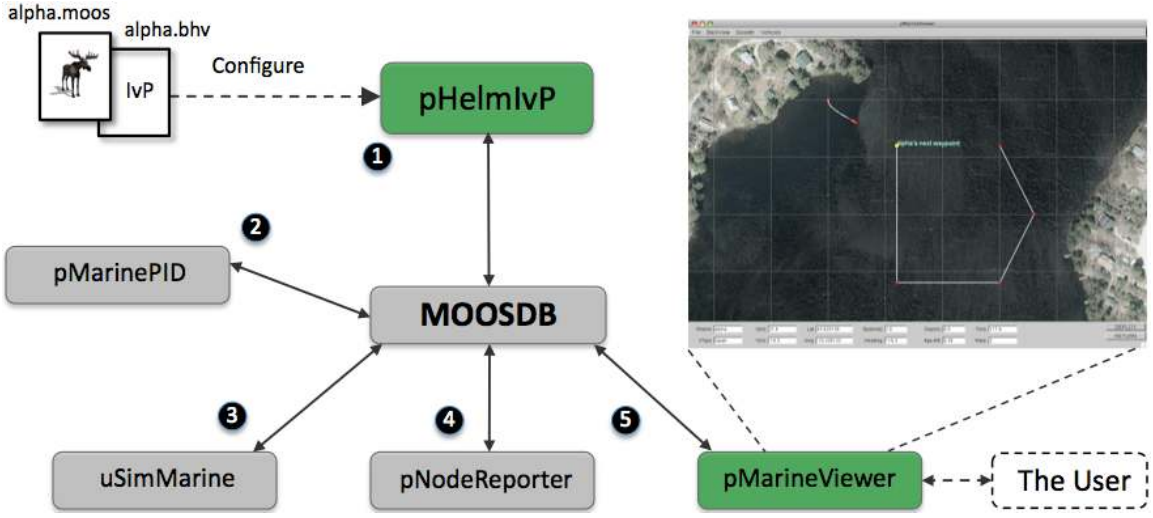

Figure 8: **The MOOS processes in the example "alpha" mission**: In (1) The helm produces a desired heading and speed. In (2) the PID controller subscribes for the desired heading and speed and publishes actuation values. In (3) the simulator grabs the actuator values and the current vehicle pose and publishes a set of MOOS variables representing the new vehicle pose. In (4) all navigation output is wrapped into a single node-report string to be consumed by the helm and the GUI viewer. In (5) the pMarineViewer grabs the node-report and renders a new vehicle posision. The user can interact with the viewer to write limited commmand and control variables to the MOOSDB.

3.3.1 Antler and the Antler Configuration Block

The `pAntler` tool is used to orchestrate the launching of all the MOOS processes participating in this example. From the command line, `pAntler` is run with a single argument the `.moos` file. As it launches processes, it hands each procoess a pointer to this same MOOS file. The Antler configuration block in this example looks like:

Listing 3.2: An example Antler configuration block for the Alpha mission.

```
1  ProcessConfig = ANTLER
2  {
3    MSBetweenLaunches = 200
4
```

```
 5    Run = MOOSDB          @ NewConsole = false
 6    Run = uSimMarine      @ NewConsole = false
 7    Run = pNodeReporter   @ NewConsole = false
 8    Run = pMarinePID      @ NewConsole = false
 9    Run = pMarineViewer   @ NewConsole = false
10    Run = pHelmIvP        @ NewConsole = false
11  }
```

The first parameter on line 2 specifies how much time should be left beteen the launching of each process. Lines 4-9 specify which processes to launch. The MOOSDB is typically launched first. The `NewConsole` switch on each line determines whether a new console window should be opened with each process. Try switching one or more of these to `true` as an experiment.

3.3.2 The pMarinePID Application

The pMarinePID application implements a simple PID controller which produces values suitable for actuator control based on inputs from the helm. In simulation the output is consumed by the vehicle simulator rather than the vehicle actuatiors.

In short: The pMarinePID application typically gets its info from pHelmIvP; produces info consumed by uSimMarine or actuator MOOS processes when not running in simulation.

Subcribes to: DESIRED_HEADING, DESIRED_SPEED.

Publishes to: DESIRED_RUDDER, DESIRED_THRUST.

3.3.3 The uSimMarine Application and Configuration Block

The uSimMarine application is a very simple vehicle simulator that considers the current vehicle pose and actuator commands and produces a new vehicle pose. It can be initialized with a given pose as shown in the configuration block used in this example, shown in Listing 3:

Listing 3.3: An example uSimMarine configuration block for the Alpha mission.

```
 1  ProcessConfig = uSimMarine
 2  {
 3    AppTick  = 10
 4    CommsTick = 10
 5
 6    START_X       = 0
 7    START_Y       = 0
 8    START_SPEED   = 0
 9    START_HEADING = 180
10    PREFIX        = NAV
11  }
```

In short: The uSimMarine application typically gets its info from pMarinePID; produces info consumed by pNodeReporter and itself on the next iteration of uSimMarine.

Subcribes to: DESIRED_RUDDER, DESIRED_THRUST, NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING.

Publishes to: NAV_X, NAV_Y, NAV_HEADING, NAV_SPEED.

3.3.4 The pNodeReporter Application and Configuration Block

An Automated Information System (AIS) is commonplace on many larger marine vessels and is comprised of a transponder and receiver that broadcasts one's own vehicle ID and pose to other nearby vessels equiped with an AIS receiver. It periodically collects all latest pose elements, e.g., latitutude and longitude position and latest measured heading and speed, and wraps it up into a single update to be broadcast. This MOOS process collects pose information by subscribing to the MOOSDB for NAV_X, NAV_Y, NAV_HEADING, NAV_SPEED, and NAV_DEPTH and wraps it up into a single MOOS variable called NODE_REPORT_LOCAL. This variable in turn can be subscribed to another MOOS process connected to an actual serial device acting as an AIS transponder. For our purposes, this variable is also subscribed to by pMarineViewer for rendering a vehicle pose sequence.

In short: The pNodeReporter application typically gets its info from uSimMarine or otherwise on-board navigation systems such as GPS or compass; produces info consumed by pMarineViewer and instances of pHelmIvP running in other vehicles or simulated vehicles.

Subcribes to: NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING.

Publishes to: NODE_REPORT_LOCAL

3.3.5 The pMarineViewer Application and Configuration Block

The pMarineViewer is a MOOS process that subscribes to the MOOS variable NODE_REPORT_LOCAL and NODE_REPORT which contains a vehicle ID, pose and timestamp. It renders the updated vehicle(s) position. It is a multi-threaded process to allow both communication with MOOS and let the user pan and zoom and otherwise interact with the GUI. It is non-essential for vehicle operation, but essential for visually confirming that all is going as planned.

In short: The pMarineViewer application typically gets its info from pNodeReporter and pHelmIvP; produces info consumed by pHelmIvP when configured to have command and control hooks (as in this example).

Subcribes to: NODE_REPORT, NODE_REPORT_LOCAL, VIEW_POINT, VIEW_SEGLIST, VIEW_POLYGON, VIEW_MARKER.

Publishes to: Depends on configuration, but in this example: DEPLOY, RETURN.

4 A Very Brief Overview of MOOS

MOOS is often described as autonomy *middleware* which implies that it is a kind of glue that connects a collection of applications where the real work happens. MOOS does indeed connect a collection of applications, of which the IvP Helm is one. MOOS is cross platform stand-alone and dependency free. It needs no other third-party libraries. 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. Due to its combination of ease-of-use, general extendibility 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.

4.1 Inter-process communication with Publish/Subscribe

MOOS has a star-like topology as depicted in Figure 3. Application within a MOOS community (a MOOSApp) have a connection to a single MOOS Database (called MOOSDB) lying 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.
- Communication between the client and server is initiated by the client, i.e., the MOOSDB never makes a unsolicited attempt to contact a MOOSApp from out of the blue
- Each client has a unique name.
- A given client need have no knowledge of what other clients exist.
- One client does not transmit data to another - it can only be sent to the MOOSDB and from there to other clients. Modern versions of the library sport a sub-one millisecond latency when transporting multi-MB payloads between processes.
- The star network can be distributed over any number of machines running any combination of supported operating systems.
- The communications layer supports clock synchronization across all connected clients and in the same vein, can support ”time acceleration” whereby all connected clients operate in an accelerated time stream - something that is very useful in simulations involving many processes distributed over many machines.
- data can be sent in small bites as ”string” or ”double” packets or in arbitrarily large binary packets.

4.2 Message Content

The communications API in MOOS allows data to be transmitted between the MOOSDB and a client. The meaning of that data is dependent on the role of the client. However the form of that data is not constrained by MOOS although for the sake on convenience MOOS does offer bespoke support for small ”double” and string payloads. (Note that very early versions of MOOS only allowed data to be sent as strings or doubles - but this restriction is now long gone.) Data is packed into messages which contain other salient information shown in Table 1.

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 this data to the MOOSDB
Auxiliary	Supplemental message information, e.g., IvP behavior source
Time	Time at which the data was written
Data Type	Type of data (STRING or DOUBLE or BINARY)
Message Type	Type of Message (usually NOTIFICATION)
Source Community	The community to which the source process belongs

Table 1: The contents of MOOS message.

Often it is convenient to send data in string format for example the string "Type=EST,Name=AUV,Pos=[3x1]3.4,6.3,0. might describe the position estimate of a vehicle called "AUV" as a 3x1 column vector. This is human readable and does not require the sharing and synchronizing of header files to ensure both sender and recipient understand how to interpret data (as is the case with binary data). It is quite common for MOOS applications to communicate with string data in a concatenation of comma separated "name=value" pairs.

- Strings are human readable.
- All data becomes the same type.
- Logging files are human readable (they can be compressed for storage).
- Replaying a log file is simply a case of 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 that data) - users simply would not understand new data fields but they would not crash.

The above are well understood benefits of sending self-explanatory ASCII data. However many applications use data types which do not lend themselves to verbose serialization to strings — think for example about camera image data being generated at 40Hz in full colour. At this point the need to send binary data is clear and of course MOOS supports it transparently (and the application pLogger supports logging and replaying it).

At this point it is up to the user to ensure that the binary data can be interpreted by all clients and that any and all perturbations to the data structures are distributed and compiled into each and every client. It is here that modern serialization tools such as "Google Protocol Buffers" find application. They offer a seamless way to serialize complex data structures into binary streams. Crucially they offer *forward* compatibility – it is possible to update and augment data structures with new fields in the comforting knowledge that all existing apps will still be able to interpret the data - they just won't parse the new additions.

4.3 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 threaded machinery that coordinates the communication with the MOOSDB. This 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. The application can:

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

4.3.1 Publishing Data

Data is published as a pair - a variable and value - that constitute the heart of a MOOS message described in Table 1. The client invokes the `Notify(VarName, VarValue)` command where appropriate in the client code. The above command is implemented both for string, double and binary values, and the rest of the fields described in Table 1 are filled in automatically. Each notification results in another entry in the client's "outbox", which in older versions of MOOS, is emptied the next time the `MOOSDB` accepts an incoming call from the client or in recent versions, is pushed instantaneously to all interested clients.

4.3.2 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, a application that interfaces to a GPS sensor may publish data called `GPS_X` and `GPS_Y`. A different application may register its interest in this data by subscribing or registering for it. 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 has 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. MOOS V10 and later also supports "wildcard" subscriptions. For example a client can register for `"*:*"` to receive all messages from all other clients. Or `"GPS_*:?NAV"` to receive messages beginning with `"GPS_"` from any process with a four letter name ending in `"NAV"`.

4.3.3 Reading Mail

A client can enquire 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 1. 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.

4.4 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 is a loop which 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 if new data has been delivered to the application. If it has, another virtual function, `OnNewMail()`, is called. This is the function within which code is written to process the newly delivered data.

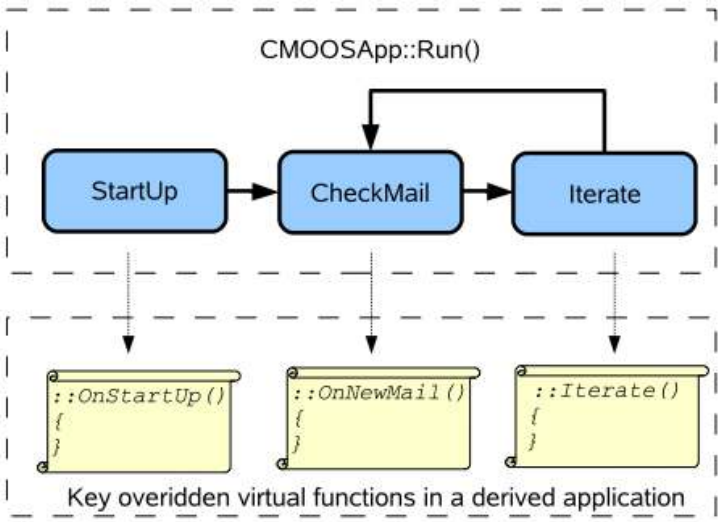

Figure 9: **Key virtual functions of the MOOS application base class**: The flow of execution once `Run()` has been called on a class derived from `CMOOSApp` . The scrolls indicate where users of the functionality of CMOOSApp will be writing new code that implements whatever it is that is wanted from the new applications. Note that it is not the case (as the above may suggest) that mail is polled for - in modern incantations of MOOS it is pushed to a client a synchronously `OnNewMail()` is called as soon as `Iterate()` is not running.

The roles of the three virtual functions in Figure 9 are discussed below. The `pHelmIvP` application does indeed inherit from `CMOOSApp` and overload these functions. The base class contains other virtual functions (`OnConnectToServer()` and `OnDisconnectFromServer()`).

4.4.1 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. The rate at which `Iterate()` is called by the `SetAppFreq()` method or by specifying the `AppTick` parameter in a 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 second to complete there is no way that this method can be called at more than 1Hz. 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.

4.4.2 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 1). The programmer is free to iterate over this collection examining who sent the data, what it pertains to, how old it is, whether or not it is string or numerical data and to act on or process the data accordingly. In recent versions of MOOS it is possible to have `OnNewMail()` called in a directly and rapidly in response to new mail being received by the back-end communications threads. This architecture allows for very rapid response times (sub ms) between a client posting data and it being received and handled by all interested parties.

4.4.3 The OnStartUp() Method

The `OnStartUp()` function is called just before the application enters into its own forever-loop depicted in Figure 9. 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.

4.5 MOOS Mission Configuration Files

Every MOOS process can read configuration parameters from a mission file which by convention has a `.moos` extension. Traditionally MOOS processes share the same mission file to the maximum extent possible. For example, it is customary for there to be one common mission file for all MOOS processes running on a given machine. Every MOOS process has information contained in a configuration block within a `*.moos` file. The block begins with the statement

```
ProcessConfig = ProcessName
```

where `ProcessName` is the unique name the application will use when connecting to the MOOSDB. The configuration block is delimited by braces. Within the braces there is a collection of parameter statements, one per line. Each statement is written as:

```
ParameterName = value
```

where `value` can be any string or numeric value. All applications deriving from `CMOOSApp` inherit several important configuration options. The most important options for `CMOOSApp` derived applications are `CommsTick` and `AppTick`. The former configures how often the communications thread talks to the `MOOSDB` and the latter how often (approximately) `Iterate()` will be called.

Parameters may also be defined at the "global" level, i.e., not in any particular process' configuration block. Three parameters that are mandatory and typically found at the top of all `*.moos` files are:

ServerHost naming the IP address associated with the MOOSDB server being launched with this file, ServerPort naming the port number over which the MOOSDB server is communicating with clients, and Community naming the community comprising the server and clients. An example is shown in lines 1-3 in Listing 4.

4.6 Launching Groups of MOOS Applications with Antler

Antler provides a simple and compact way to start a MOOS mission comprised of several MOOS processes, a.k.a., a MOOS *community*. For example if the desired mission file is `alpha.moos` then executing the following from a terminal shell:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ pAntler alpha.moos
```

will launch the required processes for the mission. It reads from its configuration block (which is declared as ProcessConfig=ANTLER) a list of process names that will constitute the MOOS community. Each process to be launched is specified with a line with the general syntax

where `LaunchConfiguration` is an optional comma-separated list of `parameter=value` pairs which collectively control how the process `procname` (for example pHelmIvP, or pLogger or MOOSDB) is launched. Exactly what parameters can be specified is outside the scope of this discussion. Antler looks through its entire configuration block and launches one process for every line which begins with the `RUN=` left-hand side. When all processes have been launched Antler waits for all of them to exit and then quits itself.

There are many more aspects of Antler not discussed here but can be found in the Antler documentation at the MOOS web site (see Section 1.3). These include hooks for altering the console appearance for each launched process, controlling the search path for specifying how executables are located on the host file system, passing parameters to launched processes, running multiple instances of a particular process, and using Antler to launch multiple distinct communities on a network.

4.7 Scoping and Poking the MOOSDB

An important tool for writing and debugging MOOS applications (and IvP Helm behaviors) is the ability for the user to interact with an active MOOS community and see the current values of particular MOOS variables (scoping the DB) and to alter one or more variables with a desired value (poking the DB). Below are listed tools for scoping and poking respectively. More information on each can be found on the Oxford or MIT web sites, or in in some instances, other parts of this document.

Tools for scoping the MOOSDB:

- uMS - A GUI-based tool written in FLTK and maintained and distributed from the Oxford website.
- uXMS - A terminal-based tool maintained and distributed from the MIT website
- uHelmScope - A terminal-based tool specialized for displaying information about a running instance of the helm, but it also contains a general-purpose scoping utility similar to uXMS. Distributed from the MIT website.

Tools for poking the MOOSDB:

- uMS - The GUI-based tool for scoping, listed above, also provides a means for poking. Distributed from the Oxford website.
- uPokeDB - A light-weight command-line tool for poking one or more variable-value pairs, with the option of scoping on the before and after values of the poked variable before exiting. Distributed from the MIT website.
- pMarineViewer - A GUI-based tool primarily used for rendering the paths of vehicles in 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. Distributed from the MIT website.
- uTimerScript - Allows the user to script a set of pre-configured pokes to a MOOSDB with each entry in the script happening after a speci- fied amount of time. Script may be paused or fast-forwarded. Events may also be configured with random values and happen randomly in a chosen window of time. Distributed from the MIT website.
- uTermCommand - A terminal-based tool for poking the DB with pre-defined variable-value pairs. The user can configure the tool to associate aliases (as short as a single character) to quickly poke the DB. Distributed from the MIT website.
- iRemote - A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associate a pre-defined variable-value poke with any un-mapped key on the keyboard. Distributed from the Oxford website.

The above list is almost certainly not a complete list for scoping and poking a MOOSDB, but it's a decent start.

4.8 A Simple MOOS Application - pXRelay

The bundle of applications distributed from `www.moos-ivp.org` contains a very simple MOOS application called pXRelay. The pXRelay application registers for a single "input" MOOS variable and publishes a single "output" MOOS variable. It makes a single publication on the output variable for each mail message received on the input variable. The value published is simply a counter representing the number of times the variable has been published. By running two (differently named) versions of pXRelay with complementary input/output variables, the two processes will perpetuate some basic publish/subscribe handshaking. This application is distributed primarily as a simple example of a MOOS application that allows for some illustration of the following topics introduced up to this point:

- Finding and launching with pAntler example code distributed with the MOOS-IvP software bundle.

- An example mission configuration file.
- Scoping variables on a running MOOSDB with the uXMS tool.
- Poking the MOOSDB with variable-value pairs using the uPokeDB tool.
- Illustrating the OnStartUp(), OnNewMail(), and Iterate() overloaded functions of the CMOOSApp base class.

Besides touching on these topics, the collection of files in the pXRelay source code sub-directory is not a bad template from which to build your own modules.

4.8.1 Finding and Launching the pXRelay Example

The pXRelay example mission should be in the same directory tree containing the source code. There is a single mission file, xrelay.moos:

```
moos-ivp/
   MOOS/
   ivp/
      missions/
         xrelay/
            xrelay.moos    <---- The MOOS file
```

To run this mission from a terminal window, simply change directories and launch:

```
$ cd moos-ivp/ivp/missions/xrelay
$ pAntler xrelay.moos
```

After pAntler has launched each process, there should be four open terminal windows, one for each pXRelay process, one for uXMS, and one for the MOOSDB itself.

4.8.2 Scoping the pXRelay Example with uXMS

Among the four windows launched in the example, the window to watch is the uXMS window, which should have output similar to the following (minus the line numbers):

Listing 4.1: Example uXMS output after the pXRelay example is launched.

```
0   VarName           (S)ource       (T)ime     (C)ommunity  VarValue
1   ----------------  ----------     ---------  ----------   ----------- (73)
2   APPLES             n/a             n/a       n/a           n/a
3   PEARS              n/a             n/a       n/a           n/a
4   APPLES_ITER_HZ     pXRelay_APPLES 14.93     xrelay       24.93561
5   PEARS_ITER_HZ      pXRelay_PEARS  14.94     xrelay       24.93683
6   APPLES_POST_HZ     n/a             n/a       n/a           n/a
7   PEARS_POST_HZ      n/a             n/a       n/a           n/a
```

Initially the only thing that is changing in this window is the integer at the end of line 1 representing the number of updates written to the terminal. Here uXMS is configured to scope on

the six variables shown in the VarName column. Column 2 shows which process last posted on the variable, column 3 shows when the last posting occurred, column 4 shows the community name from which the post originated, and column 5 shows the current value of the variable. The "n/a" entries indicate that a process has yet to write to the given variable.

There are two pXRelay processes running - one under the alias pXRelay_APPLES publishing the variable APPLES as its output variable, APPLES_ITER_HZ indicating the frequency in which the Iterate() function is executed, and APPLES_POST_HZ indicating the frequency at which the output variable is posted. There is likewise a pXRelay_PEARS process and the corresponding output variables.

4.8.3 Seeding the pXRelay Example with the uPokeDB Tool

Upon launching the pXRelay example, the only variables actively changing are the *_ITER_HZ variables (lines 4-5 in Listing 1) which confirm that the Iterate() loop in each process is indeed being executed. The output for the other variables in Listing 1 reflect the fact that the two processes have not yet begun handshaking. This can be kicked off by poking the APPLES (or PEARS) variable, which is the input variable for pXRelay_PEARS, by typing the following:

```
$ cd moos-ivp/ivp/missions/xrelay
$ uPokeDB xrelay.moos APPLES=1
```

The uPokeDB tool will publish to the MOOSDB the given variable-value pair APPLES=1. It also takes as an argument the mission file, xrelay.moos, to read information on where the MOOSDB is running in terms of machine name and port number. The output should look similar to the following:

Listing 4.2: Example uPokeDB output after poking the MOOSDB with APPLES=1.

```
0  PRIOR to Poking the MOOSDB
1    VarName               (S)ource      (T)ime       VarValue
2    ----------------      ----------    ----------   -------------
3    APPLES
4
5
6  AFTER Poking the MOOSDB
7    VarName               (S)ource      (T)ime       VarValue
8    ----------------      ----------    ----------   -------------
9    APPLES                uPokeDB       40.19        1.00000"
```

The output of uPokeDB first shows the value of the variable prior to the poke, and then the value afterwards. Once the MOOSDB has been poked as above, the pXRelay_PEARS application will receive this mail and, in return, will write to its output variable PEARS, which in turn will be read by pXRelay_APPLES and the two processes will continue thereafter to write and read their input and output variables. This progression can be observed in the uXMS terminal, which may look something like that shown in Listing 3:

Listing 4.3: Example uXMS output after the pXRelay example is seeded.

```
0   VarName           (S)ource         (T)ime    (C)ommunity   VarValue
1   ----------------   ----------      --------  ----------    ----------- (221)
2   APPLES             pXRelay_APPLES  44.78     xrelay        151
3   PEARS              pXRelay_PEARS   44.74     xrelay        151
4   APPLES_ITER_HZ     pXRelay_APPLES  44.7      xrelay        24.90495
```

```
5   PEARS_ITER_HZ      pXRelay_PEARS   44.7      xrelay        24.90427
6   APPLES_POST_HZ     pXRelay_APPLES  44.79     xrelay        8.36411
7   PEARS_POST_HZ      pXRelay_PEARS   44.74     xrelay        8.36406
```

Upon each write to the MOOSDB the value of the variable is incremented by 1, and the integer progression can be monitored in the last column on lines 2-3. The APPLES_POST_HZ and PEARS_POST_HZ variables represent the frequency at which the process makes a post to the MOOSDB. This of course is different than (but bounded above by) the frequency of the Iterate() loop since a post is made within the Iterate() loop only if mail had been received prior to the outset of the loop. In a world with no latency, one might expect the "post" frequency to be exactly half of the "iterate" frequency. We would expect the frequency reported on lines 6-7 to be no greater than 12.5, and in this case values of about 8.4 are observed instead.

4.8.4 The pXRelay Example MOOS Configuration File

The mission file used for the pXRelay example, `xrelay.moos` is discussed here. This file is provided as part of the MOOS-IvP software bundle under the "missions" directory as discussed above in Section 4.8.1. It is discussed here in three parts in Listings 4 through 6 below.

The part of the `xrelay.moos` file provides three mandatory pieces of information needed by the MOOSDB process for launching. The MOOSDB is a server and on line 1 is the IP address for the machine, and line 2 indicates the port number where clients can expect to find the MOOSDB once it has been launched. Since each MOOSDB and the set of connected clients form a MOOS "community", the community name is provided on line 3. Note the `xrelay` community name in the `xrelay.moos` file and the community name in column 4 of the uXMS output in Listing 1 above.

Listing 4.4: The `xrelay.moos` *mission file for the pXRelay example.*

```
1  ServerHost = localhost
2  ServerPort = 9000
3  Community  = xrelay
4
5  //------------------------------------------
6  // Antler configuration  block
7  ProcessConfig = ANTLER
8  {
9    MSBetweenLaunches = 200
10
11   Run = MOOSDB  @ NewConsole = true
12   Run = pXRelay  @ NewConsole = true ~ pXRelay_PEARS
13   Run = pXRelay  @ NewConsole = true ~ pXRelay_APPLES
14   Run = uXMS  @ NewConsole = true
15 }
```

The configuration block in lines 7-15 of `xrelay.moos` is read by the pAntler for launching the processes or clients of the MOOS community. Line 9 specifies how much time, in milliseconds, between the launching of processes. Lines 11-14 name the four MOOS applications launched in this example. On these lines, the component "NewConsole = true" determines whether a new console window will be opened for each process. Try changing them to `false` - only the uXMS window really needs to be open. The others merely provide a visual confirmation that a process has been launched. The "~ pXRelay_PEARS" component of lines 12 and 13 tell pAntler to launch these applications with the given alias. This is required here since each MOOS client needs to have a unique name, and in this example two instances of the pXRelay process are being launched.

In lines 17-39 in Listing 5-B below, the two pXRelay applications are configured. Note that the argument to ProcessConfig on lines 20 and 32 is the alias for pXRelay specified in the Antler configuration block on lines 12 and 13. Each pXRelay process is configured such that its incoming and outgoing MOOS variables complement one another on lines 25-26 and 37-38. Note the AppTick parameter (see Section 4.4.1) is set to 25 in both configuration blocks, and compare with the observed frequency of the Iterate() function reported in the variables APPLES_ITER_HZ and PEARS_ITER_HZ in Listing 1. MOOS has done a pretty faithful job in this example of honoring the requested frequency of the Iterate() loop in each application.

Listing 4.5: The xrelay.moos mission file - configuring the pXRelay processes.

```
17  //------------------------------------------
18  // pXRelay config block
19
20  ProcessConfig = pXRelay_APPLES
21  {
22    AppTick       = 25
23    CommsTick     = 25
24
25    OUTGOING_VAR  = APPLES
26    INCOMING_VAR  = PEARS
27  }
28
29  //------------------------------------------
30  // pXRelay config block
31
32  ProcessConfig = pXRelay_PEARS
33  {
34    AppTick       = 25
35    CommsTick     = 25
36
37    INCOMING_VAR  = APPLES
38    OUTGOING_VAR  = PEARS
39  }
```

In the last portion of the xrelay.moos file, shown in Listing 6-C below, the uXMS process is configured. In this example, uXMS is configured to scope on the six variables specified on lines 54-59 to give the output shown in Listings 1 and 3. By setting the paused parameter on line 49 to false, the output of uXMS is continuously and automatically updated - in this case four times per second due to the rate of 4Hz specified in lines 46-47. The display_* parameters in lines 50-52 ensure that the output in columns 2-4 of the uXMS output is expanded.

Listing 4.6: Configuring uXMS in the pXRelay example.

```
41  //------------------------------------------
42  // uXMS config block
43
44  ProcessConfig = uXMS
45  {
46    AppTick    = 4
47    CommsTick  = 4
48
49    paused            = false
50    display_source    = true
51    display_time      = true
52    display_community = true
53
54    var  = APPLES
```

```
55    var  = PEARS
56    var  = APPLES_ITER_HZ
57    var  = PEARS_ITER_HZ
58    var  = APPLES_POST_HZ
59    var  = PEARS_POST_HZ
60  }
```

4.8.5 Suggestions for Further Things to Try with this Example

- Take a look at the OnStartUp() method in the XRelay.cpp class in the pXRelay module in the software bundle to see how the handling of parameters in the xrelay.moos configuration file are implemented, and the subscription for a MOOS variable.
- Take a look at the OnNewMail() method in the XRelay.cpp class in the pXRelay module in the software bundle to see how incoming mail is parsed and handled.
- Take a look at the Iterate() method in the XRelay.cpp class in the pXRelay module in the software bundle to see an example of a MOOS process that acts upon incoming mail and conditionally posts to the MOOSDB
- Try changing the AppTick parameter in *one* of the pXRelay configuration blocks in the xrelay.moos file, re-start, and note the resulting change in the iteration and post frequencies in the uXMS output.
- Try changing the CommsTick parameter in *one* of the pXRelay configuration blocks in the xrelay.moos file to something much lower than the AppTick parameter, re-start, and note the resulting change in the iteration and post frequencies in the uXMS output.

4.9 MOOS Applications Available to the Public

Below are very brief descriptions of MOOS applications in the public domain. This is by no means a complete list. It does not include applications outside MIT and Oxford, and it is not even a complete list of applications from those organizations.

4.9.1 MOOS Modules from Oxford

- pAntler: A tool for launching a collection of MOOS processes given a mission file.
- pShare: A tool that allows messages to pass between communities and allows for the renaming of messages as they are shuffled between communities.
- pLogger: A logger for recording the activities of a MOOS session. It can be configured to record a fraction of, or all publications of any number of MOOS variables.
- uMS: A GUI-Based MOOS scope for monitoring one or more MOOSDBs.
- uPlayback: An FLTK-based, 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.
- 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.

- iRemote: A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associate a pre-defined variable-value poke with any un-mapped key on the keyboard.

4.9.2 Mission Monitoring Modules

Mission monitoring modules aid the user in either keeping a high-level tab on the mission as it unfolds, or help the user analyze and debug a mission. In release 13.2 this includes two powerful new tools for appcast monitoring, uMAC and uMACView. The pMarineViewer has also been substantially augmented to support appcast viewing.

- pMarineViewer: GUI tool for rendering events in an area of vehicle operation. It repeatedly updates vehicle positions from incoming node reports, and will render several geometric types published from other MOOS apps. The viewer may also post messages to the MOOSDB based on user-configured keyboard or mouse events. See the online documentation for pMarineViewer.
- uHelmScope: A terminal-based (non-GUI) scope onto a running IvP Helm process, and key MOOS variables. It provides behavior summaries, activity states, and recent behavior postings to the MOOSDB. A very useful tool for debugging helm anomalies. See the documenation for uHelmScope.
- uXMS: A terminal-based (non GUI) tool for scoping a MOOSDB Users may precisely configure the set of variables they wish to scope on by naming them explicitly on the command line or in the MOOS configuration block. The variable set may also be configured by naming one or more MOOS proceses on which all variables published by those processes will be scoped. Users may also scope on the *history* of a single variable. See the documentation for uXMS.
- uProcessWatch: This application monitors the presence of MOOS apps on a watch-list. If one or more are noted to be absent, it will be so noted on the MOOS variable PROC_WATCH_SUMMARY. uProcessWatch is appcast-enabled and will produce a succinct table summary of watched processes and the CPU load reported by the processes themselves. The items on the watch list may be named explicitly in the config file or inferred from the Antler block or from list of DB_CLIENTS. An application may be excluded from the watch list if desired. See the documentation for uProcessWatch.
- uMAC: The uMAC application is a utility for Monitoring AppCasts. It is launched and run in a terminal window and will parse appcasts generated within its own MOOS community or those from other MOOS communities bridged or shared to the local MOOSDB. The primary advantage of uMAC versus other appcast monitoring tools is that a user can remotely log into a vehicle via ssh and launch uMAC locally in a terminal. See the documentation for uMAC.
- uMACView: A GUI tool for visually monitoring appcasts. It will parse appcasts generated within its own MOOS community or those from other MOOS communities bridged or shared to the local MOOSDB. Its capability is nearly identical to the appcast viewing capability built into pMarineViewer. It was intended to be an appcast viewer for non-pMarineViewer users. See the documentation for uMACView.

4.9.3 Mission Execution Modules

Mission execution modules participate directly in the proper execution of the mission rather than simply helping to monitor, plan or analyze the mission.

- pNodeReporter: A tool for collecting node information such as present vehicle position, trajectory and type, and posting it in a single report for sharing between vehicles or sending to a shoreside display. See the documentation for pNodeReporter.
- pContactMgrV20: The contact manager deals with other known vehicles in its vicinity. It handles incoming reports perhaps received via a sensor application or over a communications link. Minimally it posts summary reports to the MOOSDB, but may also be configured to post alerts with user-configured content about one or more of the contacts. May be used in conjunction with the helm to spawn contact-related behaviors for collision avoidance, tracking, etc. See the documentation for pContactMgrV20.
- pEchoVar: A tool for subscribing for a variable and re-publishing it under a different name. It also may be used to pull out certain fields in string publications consisting of comma-separated parameter=value pairs, publishing the new string using different parameters. See the documentation for pEchoVar.
- pSearchGrid: An application for storing a history of vehicle positions in a 2D grid defined over a region of operation.

4.9.4 Mission Simulation Modules

Mission simulation modules are used only in simulation. Many of the applications in the uField Toolbox may also be considered simulation modules, but they also have a use case involving simulated sensors on actual physical vehicles. The two modules below are purely for simulated vehicles.

- uSimMarineV22: A simple 3D vehicle simulator that updates vehicle state, position and trajectory, based on the present actuator values and prior vehicle state. Typical usage scenario has a single instance of uSimMarineV22 associated with each simulated vehicle. See the documentation for uSimMarineV22.
- uSimCurrent: A simple application for simulating the effects of water current. Based on local current information from a given file, it repeately reads the vehicle's present position and publishes a drift vector, presumably consumed by uSimMarine.

4.9.5 Modules for Poking the MOOSDB

Poking the MOOSDB is a common and often essential part of mission execution and/or command and control. The pMarineViewer tool also contains several methods for poking the MOOSDB on user command.

- uPokeDB: A command-line tool for poking a MOOSDB with variable-value pairs provided on the command line. It finds the MOOSDB via mission file provided on the command line, or the IP address and port number given on the command line. It will connect to the DB, show the value prior to poking, poke the DB, and wait for mail from the DB to confirm the result of the poke. See the documentation for uPokeDB.
- uTimerScript: Allows the user to script a set of pre-configured pokes to a MOOSDB with each entry in the script happening after a speci- fied amount of time. Script may be paused or fast-forwarded. Events may also be configured with random values and happen randomly in a chosen window of time. See the documentation for uTimerScript.

- uTermCommand: A terminal application for poking the MOOSDB with pre-defined variable-value pairs. A unique key may be associated with each poke. See the documentation for uTermCommand.

4.9.6 The Alog Toolbox

The Alog Toolbox is set of offline tools for analyzing and manipulating alog files produces by the pLogger application distributed with the Oxford MOOS codebase.

- alogscan: A command line tool for reporting the contents of a given MOOS .alog file. See the documentation for alogscan, part of the Alog Toolbox.
- alogclip: A command line tool that will create a new MOOS .alog file from a given .alog file by removing entries outside a given time window. See the documentation for alogclip, part of the Alog Toolbox.
- aloggrep: A command line tool that will create a new MOOS .alog file by retaining only the given MOOS variables or sources from a given .alog file. See the documentation for aloggrep, part of the Alog Toolbox.
- alogrm: A command line tool that will create a new MOOS .alog file by removing the given MOOS variables or sources from a given .alog file. See the documentation for alogrm, part of the Alog Toolbox.
- alogview: A GUI tool for analyzing a vehicle mission by plotting one or more vehicle trajectories on the operation area, while viewing a plot of any of the numerical values in the alog file(s). See the documentation for alogview, part of the Alog Toolbox.

4.9.7 The uField Toolbox

The uField Toolbox contains a number of tools for supporting multi-vehicle missions where each vehicle is connected to a shoreside community. This includes both simulation and real field experiments. It also contains a number of simulated sensors that run on offboard the vehicle on the shoreside.

- pHostInfo: Automatically detect the vehicle's host information including the IP addresses, port being used by the MOOSDB, the port being used by local pShare for UDP listening, and the community name for the local MOOSDB. Post these to facilitate automatic intervehicle communications in especially in multi-vehicle scenarios where the local IP address changes with DHCP.
- uFldNodeBroker: Typically run on a vehicle or simulated vehicle in a multi-vehicle context. Used for making a connection to a shoreside community by sending local information about the vehicle such as the IP address, community name, and port number being used by pShare for incoming UDP messages. Presumably the shoreside community uses this to know where to send outgoing UDP messages to the vehicle. See the documentation for uFldNodeBroker, part of the uField Toolbox.
- uFldShoreBroker: Typically run in a shoreside community. Takes reports from remote vehicles describing how they may be reached. Posts registration requests to shoreside pShare to bridge user-provided list of variables out to vehicles. Upon learning of vehicle JAKE will

create bridges FOO_ALL and FOO_JAKE to JAKE, for all such user-configured variables. See the documentation for uFldShoreBroker, part of the uField Toolbox.

- uFldNodeComms: A shoreside tool for managing communications between vehicles. It has knowledge of all vehicle positions based on incoming node reports. Communications may be limited based on vehicle range, frequency of messages, or size of message. Messages may also be blocked based on a team affiliation. See the documentation for uFldNodeComms, part of the uField Toolbox.
- uFldMessageHandler: A tool for handling incoming messages from other nodes. The message is a string that contains the source and destination of the message as well as the MOOS variable and value. This app simply posts to the local MOOSDB the variable-value pair contents of the message. See the documentation for uFldMessageHandler, part of the uField Toolbox.
- uFldScope: Typically run in a shoreside community. Takes information from user-configured set of incoming reports and parses out key information into a concise table format. Reports may be any report in the form of comma-separated parameter-value pairs.
- uFldPathCheck: Typically run in a shoreside community. Takes node reports from remote vehicles and calculates the current vehicle speed and total distance travelled and posts them in two concise reports. Odometry tallies may be re-set to zero by other apps. See the documentation for uFldPathCheck, part of the uField Toolbox.
- uFldHazardSensor: Typically run in a shoreside community. Configured with a set objects with a given x,y location and classification (hazard or beningn). The sensor simulator receives a series of requests from a remote vehicle. When sensor determines that an object is is within the sensor field of a requesting vehicle, it may or may not return a sensor detection report for the object, and perhaps also a proper classification. The odds of receiving a detection and proper classification depend on the sensor configuration and the user's preference for P_D/P_FA on the prevailing ROC curve.
- uFldHazardMetric: An application for grading incoming hazard reports, presumably generated by users of the uFldHazardSensor after exploring a simulated hazard field.
- uFldHazardMgr: The uFldHazardMgr is a strawman MOOS app for managing hazard sensor information and generation of a hazard report over the course of an autonomous search mission.
- uFldBeaconRangeSensor: Typically run in a shoreside community. Configured with one or more beacons with known beacon locations. Takes range requests from a remote vehicle and returns a range report indicating that vehicle's range to nearby beacons. Range requests may or may not be answered depending on range to beacon. Reports may have noise added and may or may not include beacon ID. See the documentation for uFldBeaconRangeSensor, part of the uField Toolbox.
- uFldContactRangeSensor: Typically run in a shoreside community. Takes reports from remote vehicles, notes their position. Takes a range request from a remote vehicle and returns a range report indicating that vehicle's range to nearby vehicles. Range requests may or may not be answered dependent on inter-vehicle range. Reports may also have noise added to their range values. See the documentation for uFldContactRangeSensor, part of the uField Toolbox.

5 The IvP Helm as a MOOS Application

In this section the helm is discussed in terms of its identity as a MOOS application - its MOOS configuration parameters, its `Iterate()` loop, its output to the console, and its output in terms of publications to other applications running in the MOOS community. The *helm state* and *all-stop status* are also introduced since they are the highest level descriptions regarding helm activity.

5.1 Overview

The IvP Helm is implemented as the MOOS module called `pHelmIvP`. On the surface it is similar to any other MOOS application - it runs as a single process that connects to a running `MOOSDB` process interfacing solely by a publish-subscribe interface, as depicted in Figure 10. It is configured from a behavior file, or `.bhv` file, in addition to the MOOS file used to configure other MOOS applications. The helm primarily publishes a steady stream of information that drives the platform, typically regarding the desired heading, speed or depth. It may also publish information conveying aspects of the autonomy state that may be useful for monitoring, debugging or triggering other algorithms either within the helm or in other MOOS processes. The helm can be configured to generate decisions over virtually any user-defined decision space.

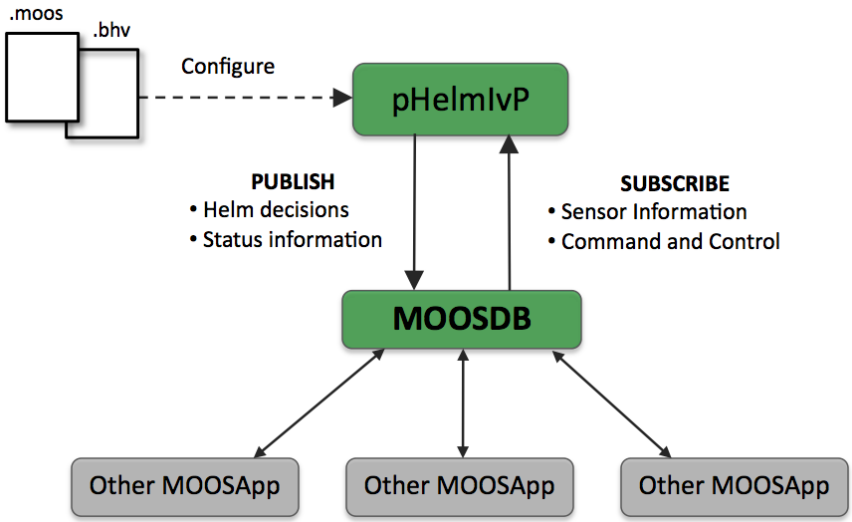

Figure 10: **The `pHelmIvP` MOOS application**: The IvP Helm is implemented as the MOOS application `pHelmIvP`. The helm is configured with two files - the mission file and behavior file. Once launched it connects to the `MOOSDB` along with other MOOS applications performing other functions. Information flowing into the helm include both sensor information and command and control inputs. The helm produces commands for maneuvering the vehicle along with other status information produced by active behaviors.

The helm subscribes for sensor information or any other information it needs to make decisions. This information includes navigation information regarding the platform's current position and trajectory, information regarding the position or state of other vehicles, or environmental information. The information it subscribes for is prescribed by the behaviors themselves, configured in the `.bhv`

file. In addition to sensor information, the helm also receives some level of command and control information. For example, in some marine vehicle configurations, one of the "Other MOOSApp" modules in the figure is a driver for an acoustic modem over which command and control information may be relayed.

The helm has a couple informative high-level state descriptions, the *helm state* and the *all-stop status*, that may be compared to the operation of an automobile. Launching the `pHelmIvP` MOOS process is analagous to turning on the car's engine. Putting the helm in the `DRIVE` mode is like shifting the car from "Park" to "Drive". And the all-stop status refers whether or not the car is breaking to a stop. The analogy is summarized below.

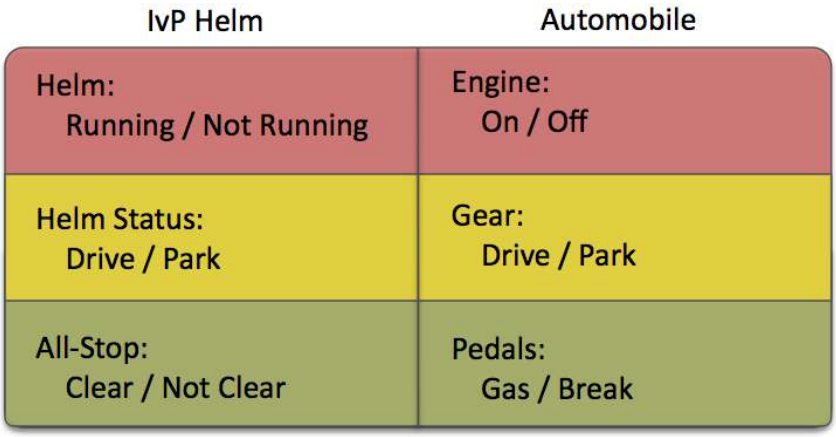

Figure 11: **The Helm/Automobile Analogy**: The helm and its high-level state descriptions, the *helm state* and *all-stop status*, have analogies with the operation of an automobile.

5.2 The Helm State

The highest level interface with the helm is reflected in the *helm state*. The helm state may have one of two values, *park* or *drive* (except when using a *standby helm* described in Section 5.7) In the *drive* mode, the helm is likely in the process of executing a mission. In the *park* mode, the helm is waiting, likely because it is being asked to wait, but also because the helm may have noticed something wrong (generated an all-stop) and subsequently put itself into *park* on its own, awaiting a the chance to return to the *drive* state. In this section we discuss (a) how the helm state is changed, (b) what it is going on in the helm when it is parked, and (c) how the helm state is initialized at start-up. At any point in time after the helm is launched, the helm will post the MOOS variable `IVPHELM_STATE` with either the value `"DRIVE"`, `"PARK"` or `"MALCONFIG"`. This is posted on each iteration and registering for this mail is the manner recommended by which other MOOS applications monitor the helm's heart beat.

5.2.1 Helm State Transitions

The helm state may be transitioned by writing to the MOOS variable `MOOS_MANUAL_OVERIDE`. As Figure 12 depicts, a value of `false`, which is case insensitive, transitions the helm state into `DRIVE`. A value of `true` puts it into the `PARK`. When the helm transitions from `DRIVE` to `PARK` it makes *one*

more publication to the helm decision variables, each with a value zero. This is referred to as the production of an *all-stop posting*, discussed in more detail in a later section.

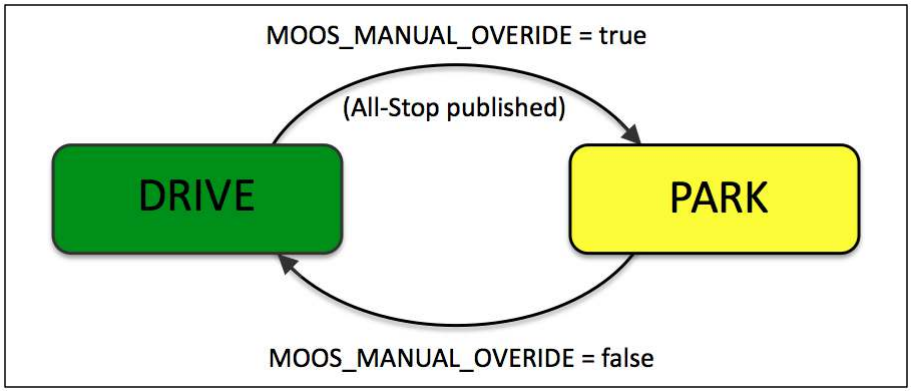

Figure 12: **The Helm State of the IvP Helm**: The helm state has a value of either `PARK` or `DRIVE`, depending on both how the helm is initialized and the mail received by the helm after start-up on the variable `MOOS_MANUAL_OVERIDE`. The helm may also park itself if an all-stop event has been detected.

The variable `MOOS_MANUAL_OVERIDE` contains the mis-spelling of "override". However, it is a variable that has some legacy presence in other MOOS applications such as `iRemote`. To avoid a situation where there is an attempt to override the helm, but the request is ignored because of a (proper) spelling, the helm will also respect transition requests on the properly spelled variable `MOOS_MANUAL_OVERRIDE`. This has the drawback however that these two variables could conceivably have different values in the `MOOSDB`. This is not a problem but could be confusing for someone trying to infer the helm state by opening a scope on the `MOOSDB`, on either the wrong variable or the two disagreeing variables. In this case the helm state would be aligned with the variable with the most recent publication time stamp. In any event, the best way to monitor the helm state is to scope on the MOOS variable `IVPHELM_STATE`, published by the helm itself, or use the `uHelmScope` tool.

The helm may also automatically transition itself from the `DRIVE` to the `PARK` state (but never the other way around), by posting an all-stop event. All-stop events and the helm are discussed separately in Section 5.3. All-stop events are generated by the helm upon finding that one or more possible error conditions have been detected during the normal execution of the helm iteration. If the helm parks due to an all-stop, it may be returned to the `DRIVE` state by another MOOS client posting `MOOS_MANUAL_OVERIDE=false`, but this is no guarantee that the helm wouldn't just park again immediately if the same condition persists that caused the all-stop event.

5.2.2 What Is and Isn't Happening when the Helm is Parked

When the helm state is `PARK`, the MOOS application loop carries on. The `OnNewMail()` continues to be called and new mail is read and dealt with exactly as it would if the helm state were `DRIVE`. The `Iterate()` loop, however, is truncated to virtually a no-op, with the only action being the output of a heartbeat character to the console if the helm is configured to do so (Section 5.5). No behavior code is called whatsoever. The helm iteration counter, a key index in the `uHelmScope` output, is also suspended despite the fact that technically the `Iterate()` loop continues to be called.

5.2.3 Initializing the Helm State at Process Launch Time

The helm, by default, is configured to be initiallly in PARK upon start-up. By setting the parameter start_in_drive=true in the mission file configuration block, the helm will indeed be in the DRIVE upon start-up. This feature was found to have practical use in UUV operations to allow for rebooting of the autonomy computer to automatically launch the helm, in DRIVE and ready to accept field commands. This feature should be used with caution, and it may be phased out in a later software release.

5.3 Helm All-Stop Events and All-Stop Status

An *all-stop event* is something that brings the vehicle to a full-stop, with typically zero-speed, zero-depth commanded. The *all-stop status* is simply a string describing *why* the vehicle is at an all-stop. Sometimes an all-stop indicates a problem, e.g., missing critical sensor information. Sometimes a vehicle may just stop as part of the mission, e.g., coming to the surface for a GPS fix. The all-stop status message can be used to discern the two types of situations. In the automobile analogy, an all-stop is equivalent to hitting the car's breaks with the intent to stop completely. An all-stop event will result in the following:

- Zero-values will be posted for all decision variables, DESIRED_SPEED=0, DESIRED_DEPTH=0, etc.
- The helm will possibly transition into PARK. By default the helm is configured to remain in the DRIVE upon an all-stop event, but if configured instead with park_on_allstop = true, the helm will indeed park upon an all-stop.
- The reason for the all-stop will be posted to MOOS variable IVPHELM_ALLSTOP. The value of this variable will be "clear" if there are no all-stop events that have occurred since the helm has entered the DRIVE state.

The reasons for all-stop may be:

- No behaviors are active. The helm has absolutely no opinion about any of its decision variables. In this case, the following would be posted: IVPHELM_ALLSTOP ="NothingToDo".
- Some behaviors are active, but decisions are missing on one or more mandatory decision variables. In this case, the following would be posted: IVPHELM_ALLSTOP ="MissingDecVars".
- When the vehicle is parked due to manual override, the following would be posted: IVPHELM_ALLSTOP ="ManualOverride".
- One of the behaviors has determined an all-stop is warranted for some reason. For example, a waypoint behavior that cannot determine own-platform's current position would declare an all-stop. In this case, the following would be posted: IVPHELM_ALLSTOP ="BehaviorError".

To gain further insight on an all-stop caused by a behavior error, the nature of the error is expressed in a separate posting to the MOOS BHV_ERROR variable. It's possible that more than one behavior error occurred in the helm iteration where the all-stop event occurred, in which case there would be multiple postings to the BHV_ERROR variable. When the vehicle is in DRIVE and operating free of an all-stop, the all-stop status is reflected by IVPHELM_ALLSTOP ="clear".

5.4 Parameters for the pHelmIvP MOOS Configuration Block

The following configuration parameters are defined for the helm. The parameter names are case insensitive.

Listing 5.1: Configuration Parameters for pHelmIvP.

allow_park:	If false, the helm cannot be put into PARK. This parameter is not mandatory. The default is true. Section 5.4.1.
behaviors:	The name and location of the behavior configuration file. This parameter is not mandatory, but typically it is used. Technically the helm can be launched from the command line and provided the behavior file on the command line. Section 5.4.2.
ivp_behavior_dir:	A directory to look for dynamically loaded behaviors. This parameter is not mandatory, since the directory information may also be handled using a shell environment variable. Section 5.4.3.
community:	Global MOOS parameter. Determines ownship name. This parameter is mandatory, but it is provided outside the helm configuration block and used by other applications. Section 5.4.4.
park_on_allstop:	If true helm will park on all-stop. This parameter is not mandatory and the default is false. Secttion 5.4.1.
domain:	The decision space for the IvP Solver. This parameter is mandatory. Section 5.4.6.
hold_on_app:	A list of MOOS apps to wait for before the helm publishes postings from behaviors' onHelmStart() function calls. Available after Release 17.7.x. Section 5.4.10.
ok_skew:	The tolerance on the age, in seconds, of incoming mail before rejected as being too old. This parameter is not mandatory. Section 5.4.7.
other_override_var:	The parameter names an additional MOOS variable acting as MOOS_MANUAL_OVERIDE. This parameter is not mandatory. Section 5.4.8.
start_in_drive:	Determines whether or not the helm is in override mode at start-up. This parameter is not mandatory. The default is false. Secton 5.4.9.
helm_prefix:	Add a prefix to all the DESIRED_* helm output. For example helm_prefix=FOO_ would result in FOO_DESIRED_SPEED and so on. Introduced after Release 19.8.x
verbose:	Determines verbosity of terminal output - quiet, terse, or verbose. This parameter is not mandatory. The default is verbose. Section 5.4.11.

5.4.1 The allow_park Parameter

Optional. By setting this parameter to false, the helm cannot be manually overriden (parked) once it has been put into DRIVE. This can be dangerous and should be carefully considered, and thus the default is true. This option was implemented based on experiences with launching UUV autonomy missions and preventing an inadvertent park due to a remote login to the vehicle. There was a tendency for some users to use iRemote upon remote login to interact with MOOS, and iRemote

posts MOOS_MANUAL_OVERIDE =true upon launch and connection to the MOOSDB.

5.4.2 The behaviors Parameter

Optional (sort of). The parameter names the behavior file, i.e., *.bhv file, on the local file system from which the helm behaviors are read. More than one file may be specified on separate lines, and the helm will read in all files almost as if they were one single file. This is technically an optional parameter because a behvior file could be provide on the command line. A behavior file must be specified via one means or the other. If a behavior file is specified *both* on the command line and in the pHelmIvP configuration block with this parameter, they will both be used to configure the helm behaviors.

5.4.3 The behavior_dir Parameter

Optional. The parameter names a directory in the local files system where the helm is to look for dynamically loadable behaviors (as opposed to default set built in statically to the helm). Authors augmenting the helm with their own behaviors will need to specify the location of those behaviors with this parameter. More than one line may be provided, each specifying a different directory location.

5.4.4 The community Parameter

This parameter is defined at the "global" level outside of any MOOS process configuration block. The helm reads this parameter and uses its value as the name associated with "ownship". It is a mandatory parameter.

5.4.5 The park_on_allstop Parameter

Optional. By setting this parameter to `true`, the helm will park when an all-stop event occurs. The default setting is `false`.

5.4.6 The domain Parameter

Mandatory. This parameter prescribes the decision space of the helm. It consists of one line per decision variable. Each line contains a colon-separated list of four fields. Field one is the domain variable name, field two is the lower bound value, field three is the higher bound value, and field four is the number of points in the domain. For example domain = speed:0:3:16 indicates a domain variable called "speed", with a lower and upper bound 0 and 3 meters/second respectively. Since there are 16 points, the speed choices are 0, 0.2, 0.4, ..., 2.8, 3.0. The helm requires that a decision be made on all listed variables on each iteration of the control loop. If a variable is used by some behaviors but is not necessarily involved in all decisions, it can be declared as optional. For example domain=speed:0:3:16:optional.

5.4.7 The ok_skew Parameter

Optional. This parameter sets the allowable skew tolerated by the helm for receiving incoming mail messages. If a clock skew is detected greater than this value, the message will be ignored. A check

for skews can be disabled by setting ok_skew=any. The default value is 60 seconds.

5.4.8 The other_override_var Parameter

Optional. This parameter names a MOOS variable the helm will regard as being synonomous with the two default variables accepted for manual override, MOOS_MANUAL_OVERRIDE, and the legacy mispelling of this variable, MOOS_MANUAL_OVERIDE.

5.4.9 The start_in_drive Parameter

Optional. This parameter is set to either true or false. The default is false as the helm normally starts in PARK and needs to receive MOOS mail on the variable MOOS_MANUAL_OVERIDE with the value of this variable set to false. When start_in_drive is set to true, the helm is in the DRIVE state upon start-up. The issue of helm state was discussed in more detail in Section 5.2.

5.4.10 The hold_on_app Parameter

Optional. Available after Release 17.7.x. This parameter names one or more MOOS apps that the helm will monitor in the DB_CLIENTS list. Once it has seen each named app at least once, the helm will release all mail generated by behaviors through their onHelmStart() function.

This may be useful for behaviors that produce configuration information to other support applications. Examples include the pBasicContactMgr and pTaskManager application. These apps need to be told by the behaviors the nature of alerts that may be generated, potentially resulting in spawned behaviors. If the behaviors produce multiple postings of the same MOOS variable, and those postings occur before the intended recipient apps have come on line (connected to the MOOSDB), then they may only receive the last piece of mail. With this utility, the helm will wait until all named apps are connected before posting the onHelmStart() mail.

The parameter may take a comma-separated list of apps, or multiple lines may be provided. The following two styles are equivalent:

```
hold_on_app = pBasicContactMgr, pTaskManager
```

and

```
hold_on_app = pBasicContactMgr
hold_on_app = pTaskManager
```

5.4.11 The verbose Parameter

Optional. This parameter affects how much information is written to the terminal on each iteration of the helm. The possible values are *verbose*, *terse*, or *quiet*. The *verbose* setting will write a brief helm report to the terminal on each iteration. With the *terse* setting minimal output will be produced, a '*' character when not producing helm commands, and a '$' character when active and healthy. With the *quiet* setting, no output at all will be written to the terminal. The default value is *terse*. This setting can be changed after the helm is started by changing the value of HELM_VERBOSE in the MOOSDB.

5.4.12 An Example pHelmIvP MOOS Configuration Block

Below is an example configuration block for the helm.

Listing 5.2: An example `pHelmIvP` *configuration block.*

```
//-------- pHelmIvP configuration block  -------------
ProcessConfig = pHelmIvP
{
  AppTick    = 4   // Defined for all MOOS processes
  CommsTick  = 4   // Defined for all MOOS processes

  domain     = course:0:359:360
  domain     = speed:0:3:16
  domain     = depth:0:500:101

  behaviors  = foobar.bhv
  verbose    = terse
  ok_skew    = ANY

  start_in_drive  = false
  allow_park      = true
  park_on_allstop = false
}
```

The AppTick and CommsTick parameters are defined for all MOOS processes and specify the frequency in which the helm process iterates and communicates with the MOOSDB. The `community` parameter is not included in the configuration block because it is specified at the global level in the mission file.

5.5 Launching the Helm and Output to the Terminal Window

The helm can be launched either directly from the command line, or from within Antler. On the command line the usage is as follows:

```
Usage: pHelmIvP file.moos [file.bhv]...[file.bhv]
       [--help|-h] [--version|-v]

[file.moos] Filename to get MOOS config parameters.
[file.bhv]  Filename to get IvP Helm config paramters.
[-v]        Output version number and exit.
[-h]        Output this usage information and exit.
```

If no behavior file is specified in the `.moos` file then a behavior file must be given on the command line. Multiple behavior files may be provided. Order of the arguments do not matter - command line arguments ending in `.bhv` will be read as behavior files, and those ending with `.moos` as MOOS files. The specification of behavior files may also be split between references in the `.moos` file and the command line. The duplicate specification of a single file will simply be ignored. Typical start-up output to the terminal is shown in Listing 3 below.

Listing 5.3: Example start-up output generated by the pHelmIvP process.

```
0  ************************************************
1  *                                              *
2  *       This is MOOS Client                    *
3  *       c. P Newman 2001                       *
4  *                                              *
5  ************************************************
```

```
 6
 7  ---------------MOOS CONNECT-----------------------
 8    contacting a MOOS server localhost:9000 -  try 00001
 9    Contact Made
10    Handshaking as "pHelmIvP"
11    Handshaking Complete
12    Invoking User OnConnect() callback...ok
13  -------------------------------------------------
14
15  The IvP Helm (pHelmIvP) is starting....
16  Loading behavior dynamic libraries....
17      Loading directory: /Users/mikerb/project-colregs/src/lib_behaviors-colregs
18  Loading behavior dynamic libraries - FINISHED.
19  Number of behavior files: 1
20  Processing Behavior File: bravo.bhv  START
21      Successfully found file: bravo.bhv
22      InitializeBehavior: found static behavior BHV_Loiter
23      InitializeBehavior: found static behavior BHV_Loiter
24      InitializeBehavior: found static behavior BHV_Waypoint
25      InitializeBehavior: found static behavior BHV_Timer
26  Processing Behavior File: bravo.bhv  END
27  pHelmIvP is Running:
28    AppTick   @ 4.0 Hz
29    CommsTick @ 4 Hz
30    Time Warp @ 1.0
31  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

The output in lines 0-13 are standard output generated by a MOOS process launched and successfully connected to a running MOOSDB. Lines 15-30 are start-up output generated unique to the helm and the particular user usage. Behaviors used by the helm are either static or dynamic. Static behaviors are compiled in to the pHelmIvP executable. Dynamic behaviors are brought in at run time via shared libraries compiled separately. The helm looks for an environment variable IVP_BEHAVIOR_DIRS for a colon-separated list of directories to search for shared libraries. If this variable is not set, or if one or more of the directories are not legitimate directories, an error message will indicate so between what is otherwise line 16 and 18 in Listing 3. This kind of error may not actually be problematic if the behaviors specified in the behavior file can all be otherwise successfully found.

For each specified behavior file, the information shown in lines 20-26 is generated to the terminal. For each behavior configuration in a given `.bhv` file, a single line is output as in lines 22-25 indicating that the behavior type is recognized and it is configured properly. A single unrecognized behavior or improper configuration will result in (a) an error message indicating the offending line number and file name, (b) the output of the actual offending line, and (c) immediate disconnection of the process from the MOOSDB and exit. (Tip: If the helm is launched with Antler an error during start-up will result in the closing of the pHelmIvP console window which makes it hard to catch useful error output for debugging. In this case, the helm should just be launched outside of Antler in its own terminal window.)

The output on line 31 of Listing 3, a series of dollar sign characters, indicates for each character, the completion of a single helm iteration - a heartbeat output. This is the output when the verbose parameter is set to the default setting of `terse`. When set to `quiet` no output is generated at all. When set to `verbose`, a short multi-line report is generated for each iteration. An example is shown below in Listing 4:

Listing 5.4: An example helm iteration report generated by an active helm.

```
 1  Iteration: 161  ****************************************
 2  Helm Summary  --------------------------
 3  loiter_a did NOT produce an obj-function
 4  loiter_b produces obj-function - time:0.00 pcs: 9.00000 pwt: 100.00000
 5  waypt_return did NOT produce an obj-function
 6  loiter_timer did NOT produce an obj-function
 7  Number of Objective Functions: 1
 8  DESIRED_SPEED:  2.10
 9  DESIRED_COURSE:  145.00
10  (End) Iteration: 161  ****************************************
```

On each iteration the Helm Summary indicates which behaviors produced objective functions (lines 2-5), and for those that did, it indicates the CPU time needed to generate the function, the number of pieces in the piecewise linear IvP function, and its priority weight. Following this, the decision rendered for current iteration is output with one line per decision variable (lines 7-8). This is a very thin summary of what is going on within the helm and it should be noted that the uHelmScope tool is a much better suited for monitoring helm activity and debugging.

5.6 Publications and Subscriptions for IvP Helm

The IvP Helm, like any MOOS process, can be specified in terms of its interface to the MOOSDB, i.e., what variables it publishes and what variables it subscribes for. It is impossible to provide a complete specification here since the helm is comprised of behaviors, and the means to include any number of third party behaviors. Each behavior is able to post variable-value pairs, published to the MOOSDB by the helm on behalf of the behavior at the end of the iteration. Likewise, each behavior may declare to the helm any number of MOOS variables it would like the helm to register for on its behalf. Barring these variables, published and subscribed for by the helm on behalf of individual behaviors, this section addresses the remaining portion of the helm's publish - subscribe interface.

5.6.1 Variables published by the IvP Helm

Variables published by the IvP Helm are summarized below.

- IVPHELM_SUMMARY: Produced on each iteration of the helm for consumption by the uHelmScope application. It contains information on the current helm iteration regarding the number of IvP functions created, create time, solve time, which behaviors are active, running, idle, and the decision ultimately produced during the iteration. The summary does not include every component in each summary. Components that have not changed in value since the prior summary are dropped from the present summary. This is motivated by the goal to reducing the log file footprint for the helm.
- IVPHELM_STATEVARS: Produced periodically by the helm for consumption by the uHelmScope application. It contains a comma-separated list of MOOS variables involved in preconditions of any behavior, i.e., variables affecting behavior run states.
- IVPHELM_DOMAIN: Produced once by the helm at start-up for consumption by the uHelmScope application. It contains the specification of the IvP Domain in use by the helm.

- IVPHELM_MODESET: Produced once by the helm at start-up for consumption by the uHelmScope application. It contains the specification of the Hierarchical Mode Declarations, if any, in use by the helm.
- IVPHELM_STATE: Written by the helm on each iteration of the pHelmIvP MOOS application, regardless of whether the helm is in the DRIVE state or not. (see Section 5.2). It is either "DRIVE" or "PARK". This is the recommended MOOS variable for regarding as a "heartbeat" indicator of the helm.
- HELM_IPF_COUNT: Produced on each iteration of the helm. It contains the number of IvP functions involved in the solver on the current iteration.
- CREATE_CPU: The CPU time in seconds used in total by all behaviors on the current iteration for constructing IvP functions.
- LOOP_CPU: The CPU time in seconds used by the IvP solver in the current helm iteration.
- BHV_IPF: The helm will publish this variable for each active behavior in the current iteration. It contains a string representation of the IvP function produced by the behavior. It is used for visualization by the uFunctionVis application, and for logging and later playback and analysis.
- PLOGGER_CMD: This variable is published with the below value to ensure that the pLogger application logs the .bhv file along with the other data log files and the .moos file.

```
"COPY_FILE_REQUEST = filename.bhv"
```

- DESIRED_*: Each of the decision variables in the IvPDomain provided in the helm configuration will have a separate posting prefixed by DESIRED_ as in DESIRED_SPEED. One exception is that the variable course will be converted to heading for legacy reasons.
- BHV_WARNING: Although this variable may never be posted, it is the default MOOS variable used when a behavior posts a warning. A warning may be harmless but deserves consideration.
- BHV_ERROR: Although this variable may never be posted, it is the default MOOS variable used when a behavior posts what it considers a fatal error - one that the helm will interpret as a request to generate the equivalent of ALL-STOP.

In addition to the above variables, the helm will post any variable-value pair on behalf of a behavior that makes the request. These include endflags, runflags, idleflag, activeflags and inactiveflags.

5.6.2 Variables Subscribed for by the IvP Helm

Variables subscribed for by the IvP Helm are summarized below:

- MOOS_MANUAL_OVERIDE: When set to true, usually by a third-party application such as iRemote, of from a command-and-control communication, the helm may relinquish control. If the helm was configured with active_start = true, it will not relinquish control (this may be changed).
- HELM_VERBOSE: Affects the console output produced by the helm. Legal values are verbose, terse, or quiet. See Section 5.5.
- HELM_MAP_CLEAR: When received, the helm clears an internal map that is used to surpress repeated duplicate postings. See Section 5.8.

In addition to the above variables, the helm will subscribe for any variable-value pair on behalf of a behavior that makes the request. This includes, but is not limited to, variables involved in the `condition` and `updates` parameters available generally for all behaviors.

5.7 Using a Standby Helm

A *standby helm* refers to the launching of a second helm running alongside another otherwise normally-configured primary helm. This is done to mitigate the risk associated with the possible failure of the primary helm. Both helms are instances of the `pHelmIvP` MOOS application configured with a different mission and/or behaviors. Presumably the standby helm is configured with a simpler, more conservative set of behaviors focused on the safe recovery of a vehicle. Although provisions are generally made during vehicle operations to detect a missing helm heartbeat, in situations such as the operation of a UUV under ice, it is not acceptable to simply have a vehicle halt and come to the surface. An attempt should be made to execute a simpler mission to return the vehicle to a safe location for recovery.

Use of a standby helm is as simple as adding a second configuration block in the `.moos` configuration file. The Kilo example mission demonstrates a mission using the standby helm. Declaring a second helm as a standby helm requires a single additional line of the form

```
STANDBY = N
```

where `N` is the number of seconds a standby helm will tolerate an absent primary helm heartbeat before taking control from away the primary helm. *Once a standby helm take-over is triggered, the take-over is irreversible.*

5.7.1 Two Types of Helm Failure, the Causes, and Detection

What does a helm crash mean, and how might it happen? A crash is result of code that causes the process to quit unexpectedly and without warning. A line as simple as `assert(0);` in the code would be sufficient to replicate a crash, but the cause of a crash could be much more subtle due referencing memory not properly allocated and so on. The other type of helm failure is a helm that *hangs*. This refers to the scenario where the helm enters a piece of code that takes too long or never finishes execution. This could be as trivial as reaching the line `while(1);` somewhere in the code. Either type of failure is serious. Despite the fact that we have never experienced these failures in any field exercise on any vehicle, the sudden disappearance of the helm process should be considered and handled as gracefully as possible.

How is a helm failure detected? The helm produces a heartbeat message on each iteration by posting to the variable `IVPHELM_STATE`. If the helm is configured to iterate four times per second, suspicion of failure could begin anytime after a quarter of a second passes without a posting to this variable. Under typical helm CPU load with common standard behaviors, the quarter second interval should be sufficient to finish the iteration. There are additional factors to consider however. There may be other processes on the machine dominating the CPU, thus challenging the helm to do its work in the expected time. There may be a periodic behavior calculation, such as recalculating a long path of waypoints, that may cause a spike in CPU cycles needed by the behavior. As a rule of thumb, the interval of time with the absence of a heartbeat, should arguably be two seconds or more before

declaring a helm failure. This interval is directly configurable, as the parameter N, in the standby=N configuration line.

5.7.2 Handling a Helm Crash with the Standby Helm

A helm *crash* is the easier of the two failure cases to handle. The process is simply gone and no longer publishes to the MOOSDB. Of course the other MOOS processes, including the standby helm, have no way of knowing simply through their MOOS mailbox whether or not the primary helm is gone or just delayed. In either case the sequence of events is the following:

1. The standby helm detects the absence of heartbeat for more than N seconds.
2. On the very same iteration, the standby helm posts IVPHELM_STATE = DRIVE+ rather than IVPHELM_STATE = STANDBY which it had been posting on every iteration up until now. It also begins posting a desired helm decision on this iteration.

The standby helm has all the same functionality as the primary helm, modulo the behavior and mission configuration. It will be in the DRIVE state only if not manually overridden. If MOOS_MANUAL_OVERRIDE=true when the standby helm takes over, it will be in initially in PARK, not DRIVE. It will post IVPHELM_STATE=PARK+. Note the standby helm will append the '+' character to the helm state string to help other applications and the user discern that the helm state being posted is from a standby helm that has taken control.

The Kilo example mission walks through a simulated helm crash.

5.7.3 Handling a Hung Helm with the Standby Helm

Handling a helm that has *hung* requires a bit more consideration. The tricky part is that quite possibly the hung helm is only *temporarily* hung, and at some point it will become unhung and may operate for a single iteration as if it is still the helm in charge. Two helms, with different missions, both thinking they are in charge! The sequence of events is summarized below:

1. The standby helm detects the absence of heartbeat for more than N seconds.
2. On the very same iteration, the standby helm posts IVPHELM_STATE = DRIVE+ rather than IVPHELM_STATE = STANDBY which it had been posting on every iteration up until now. It also begins posting a desired helm decision on this iteration.
3. Some time later, the original primary helm finishes its iteration and posts a helm decision, e.g., a set of postings to the helm decision variables, DESIRED_HEADING etc.
4. On the next iteration of the original primary helm it notices that another helm has posted a non-standby heartbeat, e.g., a posting to IVPHELM_STATE not equal to "STANDBY".
5. The original primary helm posts an all-stop and IVPHELM_STATE = DISABLED. It is the last time it will post a heartbeat or helm decision.
6. The standby helm continues to be in control posting helm decisions oblivious to the former primary helm's epiphany and temporary influence.

The key issue in this scenario is that the original primary helm does indeed post a helm decision when it becomes un-hung even though the standby helm may have long ago taken over and may be posting a sequence of helm decisions completely at odds with the decision posted by the newly awakened un-hung primary helm.

No assumptions can be made about the MOOS process listening to the sequence of helm decisions. It may be payload interface process, or a native PID controller, or some other process responsible for converting helm decisions to lower level actuator commands. The assumption that *is* made here is that a one-time aberation in the sequence in helm decisions is tolerable. This aberation is not going to result in an actuator breaking due to a sudden change in the command sequence such as high-speed, zero-speed, high-speed.

It's also worth noting that original primary helm, upon awakening and learning it is no longer in control, does indeed post one more helm decision, an all-stop decision (`DESIRED_SPEED = 0`). This is done to ensure that an all-stop decision, that may have been put into effect by the standby helm, is not superceded by the output of the original primary helm's final helm decision made upon wakeup.

5.7.4 Activity of the Standby Helm While Standing By

In the standby state the helm will do nothing in its iterate loop other than post a heartbeat character to the terminal if configured to do so. It will not call on any behaviors to do anything. The helm will however read and process all of its otherwise subscribed for mail. In particular it will monitor for postings to `IVPHELM_STATE`, and will initiate a take-over when it hasn't received such mail for `N` seconds. Note the standby helm also publishes to this variable, `IVPHELM_STATE = STANDBY`, but ignores mail originating from itself.

The helm will also read and process all other mail in its inbox, updating the helm's information buffer. Note that the information buffer also keeps a *history* of postings to a particular variable so that normally a behavior may process multiple postings if multiple postings were made to the `MOOSDB` between helm iterations. This history is cleared by the helm at the end of each helm iteration. When the helm is in the standby state it will also clear the history after each iteration even though the behaviors have not been given the chance to access this history. This is to ensure that the information buffer history doesn't grow without bound while in standby mode. For example, if the standby and primary helm are both configured with a waypoint behavior and the primary helm visits half the points at the point of a helm crash, the standby helm's waypoint behavior would start with the first waypoint.

5.7.5 Activity of the Primary Helm After Take-Over

A primary helm that has been taken over is refered to as a *didsabled* helm. It will post only once `IVPHELM_STATE = DISABLED`. The helm process lives on however doing next to nothing. It does not read its mail, and the iterate loop simply posts a heartbeat character (`'!'`) to the terminal if configured to do so, with the helm configuration parameter `verbose=terse`.

5.8 Automated Filtering of Successive Duplicate Helm Publications

The helm implements a "duplication filter" to drastically reduce the amount of mail posted by the helm on behalf of behaviors. This filter has been noted to reduce the overall log file size seen during

in-water exercises by 60-80%. Reductions at this level noticably facilitate the use of post-mission analysis tools and data archiving. For the most part this filter is operating behind the scenes for the typical helm user. However, knowledge of it is indeed relevant for users wishing to implement their own behaviors, and we discusss it here to explain a bit what is behind the variable HELM_MAP_CLEAR to which the helm subscribes, and listed above in Section 5.6.2.

5.8.1 Motivation for the Duplication Filter

The primary motivation of implementing the duplication filter is to reduce the amount of unnecessary mail posted by the helm on behalf behaviors, and thereby greatly reduce the size of log files and facilitate the post-mission handling of data. By unnecessary we mean successive variable-value pairs that match exactly in both fields. Surely there are cases when a behavior developer may not want this filter, and there are simple ways to bypass the filter for any post. But in most cases, successive duplicate posts are just redundant and unnecessary.

5.8.2 Implementation and Usage of the Duplication Filter

The helm keeps two maps (STL maps in C++), one for string data and one for numerical data:

```
KEY --> StringValue
KEY --> DoubleValue
```

The two maps correspond to the double and string message types in MOOS. The KEY is typically the MOOS variable name. Inside a behavior implementation, the following four functions are available:

```
void postMessage(string varname, string value, string key="");
void postMessage(string varname, double value, string key="");
void postBoolMessage(string varname, bool value, string key="");
void postIntMessage(string varname, double value, string key="");
```

These functions are available in all behavior implementations because they are defined in the IvPBehavior superclass, of which all behaviors are subclasses. Before the helm posts a message to the MOOSDB the filter is applied by a simple check to its map to determine if there is a value match on the given key. If a match is made, the post will not be made to the MOOSDB on the behavior's behalf. The postIntMessage() function is merely a convenience version of the postMessage() function that rounds the variable value to the nearest integer to further reduce posts when combined with the filter. The postBoolMessage() ultimately posts a string value "true" or "false".

The default value of the *key* parameter is the empty string, and in most cases this parameter can be ommitted without disabling the duplication filter. This is because the KEY used by the caller is only part of the key actually used by the duplication filter. The actual key is the concatenation of (a) the behavior name, (b) the variable name, and (c) the key passed by the caller. Thus the default value, the empty string, still results in a decent key being used by the filter. The key is augmented by the behavior name because often there is more than one behavior posting messages on same variable. The optional key parameter is used for two reasons. First, it can be used to further distinguish posts within a behavior on the same variable name. Second, when the key value has the special value "repeatable", then no key is used and the duplication filter is disabled for that variable posting.

Two additional convenience functions are availble:

```
void postRepeatableMessage(string varname, string value);
void postRepeatableMessage(string varname, double value);
```

A posting of `postRepeatableMessage("FOO", 100)` is equivalent to `postMessage("FOO", 100, "repeatable")`.

5.8.3 Clearing the Duplication Filter

Occasionally a user, or another MOOS application in the same community as the helm, may want to "clear" the map used by the helm to implement its duplication filter. This can be done by writing to variable HELM_MAP_CLEAR, with any value. This may be necessary for the following reason. Suppose a GUI application subscribes for the variable VIEW_SEGLIST which contains a list of line segments for rendering. If the viewer application is launched *after* the variable is published, the application will only receive the most recent mail on the variable VIEW_SEGLIST. There may be publications to this variable, made prior to the most recent publication, that are relevant to the GUI application at launch time. Those publications for the variable VIEW_SEGLIST may not be the most recent from the perspective of the MOOSDB, but they may be the most recent from the perspective of a particular behavior in the helm. By clearing the filter, it gives each behavior the chance to once again have all of its variable-value posts made to the MOOSDB. In the pMarineViewer application, a publication to HELM_MAP_CLEAR is made upon start-up. Clearing the filter will only clear the way for the next post for a given variable. It will not result in the publishing to the MOOSDB of the contents of the maps used by the filter.

6 IvP Helm Autonomy

6.1 Overview

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 competing behaviors - it performs multi-objective optimization on their collective output using a mathematical programming model called interval programming. Here the IvP Helm architecture is described and the means for configuring it given a set of behaviors and a set of mission objectives.

6.1.1 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' Subsumption Architecture, [**?**]. A key principle at the heart of Brooks' architecture and arguably the primary reason its appeal has endured, is the notion that autonomy systems can be built *incrementally*. Notably, Brooks' original publication pre-dated the arrival of Open Source software and the Free Software Foundation founded by Richard Stallman. Open Source software is not a pre-requisite 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 2.5, 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 multi-objective 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, [**?**]. 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.

6.1.2 Traditional and Non-traditional 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, as we'll see in discussions in this section. This was also evident in the Alpha example mission in Section 16 where the completion of the Survey behavior triggered the Return behavior. But 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 inter-behavior "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. 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. Due to the `condition` and `endflag` parameters defined for all behaviors, a sequence of behaviors can 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. This is distinct from the de-coupled decision making style proposed in [**?**] and [**?**] - early advocates of multi-objective optimization in behavior-based action selection.

6.1.3 Two Layers of Building Autonomy in the IvP Helm

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, such as the simple example for the Alpha example mission in Listing 1.

6.2 Inside the 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 13. Description of the five components follow.

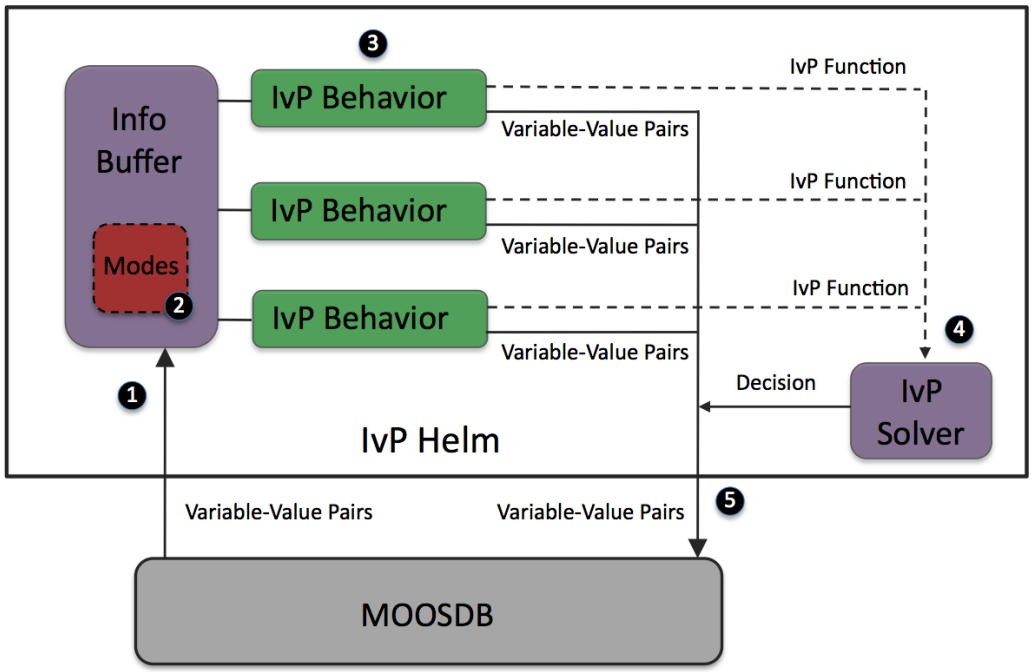

Figure 13: **The pHelmIvP Iterate Loop**: (1) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors, (2) If there were any mode declarations in the mission behavior file they are evaluated at this step. (3) Each behavior is queried for its contribution 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.

6.2.1 Step 1 - Reading Mail and Populating the Info Buffer

The first step of a helm iteration occurs outside the `Iterate()` loop. As depicted in Figure 9, 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 info 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.

6.2.2 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 6.4. 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.

6.2.3 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 is 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 the Figure 13 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.

6.2.4 Step 4 - Behavior Reconciliation

In the fourth step depicted in Figure 13, 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 multi-objective optimization over the weighted sum of functions (in effect a single objective optimization at that point). The output is a single 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.

6.2.5 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 the duplication filter described in Section 5.8, only on 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 minutes 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 concern for safety.

6.3 Mission Behavior Files

The helm is configured for a particular mission primarily through one or more mission behavior files, typically with a `*.bhv` suffix. Behavior files have three types of entries, usually but not necessarily kept in three distinct parts - (1) variable initializations, (2) behavior configurations, and (3) hierarchical mode declarations. These three parts are discussed below. The example `alpha.bhv` file in Listing 1 did not contain hierarchical mode declarations, but does contain examples of variable initializations and behavior configurations.

6.3.1 Variable Initialization Syntax

The syntax for variable initialization is fairly straight-forward:

```
initialize  <variable> = <value>
...
initialize  <variable> = <value>
```

Multiple initializations may be declared on a single line by separating each variable-value pair with a comma. The keyword `initialize` is case insensitive. The `<variable>` is indeed case sensitive since it will be published to the `MOOSDB` and MOOS variables are case sensitive when registered for by a client. The `<value>` may or may not be case sensitive depending on whether or not a client registering for the variable regards the case. Considering again the helm `Iterate()` loop depicted in Figure 13, variable initializations are applied to the helm's information buffer prior to the very first helm iteration, but are posted to the `MOOSDB` at the end of the first helm iteration.

By default, an initialization will overwrite any prior value posted to the `MOOSDB`. There may be situations, however, where the user's desired effect is that the initialization only be applied if no other value has yet been written to the given MOOS variable. The syntax in this case would be:

```
initialize_  <variable> = <value>    // Deferring to prior posts if any
```

By using the "underscore" version of the `initialize` declaration, the helm will first register with the MOOSDB for the given variable, wait an iteration until it has had chance to receive mail from the MOOSDB on that variable, and only initialize the variable if nothing is otherwise known about that variable. (Note to the very discerning reader: Such an initialization also includes both an update to the helm's information buffer and a post to the MOOSDB. Posts to the MOOSDB by the helm, as part of a variable initialization, will indeed show up in the helm's incoming mailbox on the next iteration, but they are tagged in such a way as to be ignored by the helm. This is to ensure that they do not "collide" with posts made by other processes.)

6.3.2 Behavior Configuration Syntax

The bulk of the helm configuration is done with individual behavior parameter blocks which have the following form:

```
Behavior = <behavior-type>
{
  <parameter> = <value>
  ...
  <parameter> = <value>
}
```

The first line is a declaration of the behavior type. The keyword `Behavior` is not case sensitive, but the `<behavior-type>` is. This is followed by an open brace on a separate line. Each subsequent line sets a particular parameter of the behavior to a given value. The behavior configuration concludes with a close brace on a separate line. The issue of case sensitivity for the `<parameter>` and `<value>` entries is a matter determined by the individual behavior implementation.

As a convention (not enforced in any way) general behavior parameters, defined at the IvP Behavior superclass level, are grouped together and listed before parameters that apply to a specific behavior. For example, in the Alpha example in Listing 1, the general behavior parameters are listed on lines 8-12 and 22-25, but the parameters specific to the waypoint behavior, `speed`, `radius`, and `points`, follow in a separate block. Generally it is not mandatory to provide a parameter-value pair for each parameter defined for a behavior, given that meaningful defaults are in place within the behavior implementation. Some parameters are indeed mandatory however. Documentation for the individual behavior should be consulted. Multiple instances of a behavior type are allowed, as in the Alpha example where there are two waypoint behaviors - one for traversing a set of points, and one for returning to a vehicle recovery point. Each behavior should have its own unique value provided in the `name` parameter.

6.3.3 Hierarchical Mode Declaration Syntax

Hierarchical Mode Declarations are covered in depth in Section 6.4, but the syntax is briefly discussed here. A behavior file contains a set of declaration blocks of the form:

```
Set <mode-variable-name> = <mode-value>
{
  <mode-variable-name> = <parent-value>
  <condition>
  . . .
  <condition>
} <else-value>
```

A tree will be formed where each node in the tree is described from the above type of declaration. The keyword `Set` is case insensitive. The `<mode-variable-name>`, `<parent-value>` and `<else-value>` are case sensitive. The `<condition>` entries are treated exactly as with the `condition` parameter for behaviors, see Section **??**.

As indicated in Figure 13, the value of each mode variable is reset at the outset of the `Iterate()` loop, after the information buffer is updated with incoming mail. A mode variable is set by progressing through each declaration block, and determining whether the conditions are met. Thus the ordering of the declaration blocks is significant - the specification of parent should be made prior to that of a child. Examples are further discussion can be found below in Section 6.4.

6.4 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 sub-modes 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 post-analysis.

6.4.1 Background

A trend of unmanned vehicle usage can be characterized as being increasingly less of the shorter, scripted variety to be increasingly more of the longer, adaptive mission variety. A typical mission in our own lab five years ago would contain a certain set of tasks, typically waypoints and ultimately a rendezvous point for recovering the vehicle. Data acquired during deployment was off-loaded and analyzed later in the laboratory. What has changed? The simultaneous maturation of acoustic communications, on-board sensor processing, and longer vehicle battery life has dramatically changed the nature of mission configurations. The vehicle is expected to adapt to both the phenomena it senses and processes on board, as well as adapt its operation given field-control commands received via acoustic, radio or satellite communications. Multi-vehicle collaborative missions are also increasingly viable due to lower vehicle costs and mature acomms capabilities. In such cases a vehicle is not only adapting 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 they sense and how they are commanded in the field.

6.4.2 Behavior Configuration *Without* Hierarchical Mode Declarations

Behaviors can be configured for a mission without the use of hierarchical mode declarations - support for HMDs is a relatively recent addition to the helm. HMDs are a tool for organizing which behaviors are idle or participating in which circumstances. Consider the alpha example mission in Section 16, and the behavior file in Listing 1. By examination of the behavior file, and experimenting a bit with the viewer during simulation, the vehicle apparently is always in one of three modes - (a) idle, (b) surveying the waypoints, or (c) returning to the launch point. This is achieved by the `condition` parameters for the two behaviors. There are only two variables involved in the behavior conditions, `DEPLOY` and `RETURN`. If restricted to Boolean values, the below table confirms the observation that there are only three possible modes.

DEPLOY	RETURN	Mode
true	true	Returning
true	false	Surveying
false	true	Idle
false	false	Idle

Table 3: Possible modes implied by the condition parameters in the alpha mission in Listing 1.

There are a couple drawbacks with this however. First, the modes are to be inferred from the behavior conditions and this is not trivial in missions with larger behavior files. Mapping the behavior conditions to a mode is useful both in mission planning and mission monitoring. In the alpha mission, in order to understand at any given moment what mode the vehicle is in, the two variables need to be monitored, and the above table internalized. The second drawback is the increased likelihood of error, in the form of unintentionally being in two modes at the same time, or being in an undefined mode. For example, line 11 in Listing 1 really should read `RETURN != true`, and not `RETURN = false`. Since there is no Boolean type for MOOS variables, this variable could be set to `"False"` and the condition as it reads on line 11 in Listing 1 would not be satisfied, and the vehicle would be in the idle state, despite the fact that `DEPLOY` may be set to `true`. These problems are alleviated by the use of hierarchical mode declarations.

6.4.3 Syntax of Hierarchical Mode Declarations - The Bravo Mission

An example is provided showing of the use of hierarchical mode declarations by extending the Alpha mission described in Section 16. This example mission is dubbed the "Bravo" mission in the directory `s2_bravo` alongside the Alpha mission `s1_alpha` in the MOOS-IvP distribution (Section 16.1). It is also given fully in Listing 1 on the next page. The *implicit* modes of the Alpha mission, described in Table 3, are explicitly declared in the Bravo behavior file to form the following hierarchy:

```
// Excerpt from the
// bravo.bhv file.

Set MODE = Active {
  DEPLOY = true
} Inactive

Set MODE = Surveying {
  MODE = Active
  RETURN != true
} Returning
```

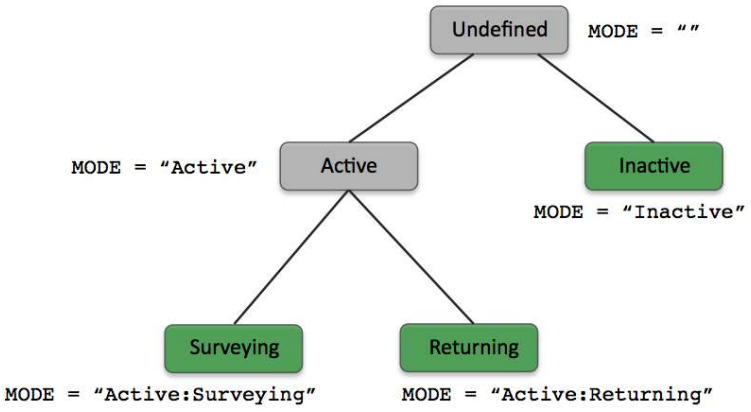

Figure 14: **Hierarchical modes for the Bravo mission**: The vehicle will always be in one of the modes represented by a leaf node. A behavior may be associated with any node in the tree. If a behavior is associated with an internal node, it is also associated with all its children.

The hierarchy in Figure 14 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 re-evaluated 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 14 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 14 `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 conditions of the behaviors, as in lines 20 and 23 in Listing 1. Note the `"=="` relation in lines 18 and 36. This is 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 this mission, it could be associated with the `"Active"` mode rather than explicitly naming all the sub-modes of the `"Active"` mode.

Listing 6.1: The Bravo Mission - Use of Hierarchical Mode Declarations.

```
 1  initialize   DEPLOY = false
 2  initialize   RETURN = false
 3
 4  //------------------- Declaration of Hierarchical Modes
 5  set MODE = ACTIVE {
 6    DEPLOY = true
 7  } INACTIVE
 8
 9  set MODE = SURVEYING {
10    MODE = ACTIVE
11    RETURN != true
12  } RETURNING
13
14  //-----------------------------------------------
15  Behavior = BHV_Waypoint
16  {
17    name      = waypt_survey
18    pwt       = 100
19    condition = MODE == SURVEYING
20    endflag   = RETURN = true
21    perpetual = true
22
23            lead = 8
24     lead_damper = 1
25           speed = 2.0   // meters per second
26          radius = 4.0
27       nm_radius = 10.0
28          points = 60,-40:60,-160:150,-160:180,-100:150,-40
29          repeat = 1
30  }
31
32  //-----------------------------------------------
```

```
33  Behavior = BHV_Waypoint
34  {
35    name       = waypt_return
36    pwt        = 100
37    condition  = MODE == RETURNING
38    perpetual  = true
39    endflag    = RETURN = false
40    endflag    = DEPLOY = false
41
42         speed = 2.0
43        radius = 2.0
44     nm_radius = 8.0
45         point = 0,0
46  }
```

6.4.4 A More Complex Example of Hierarchical Mode Declarations

The Bravo example given above, while having the benefit of being a working example distributed with the codebase, is not complex. In this section a modestly complex, although fictional, hierarchy is provided to highlight some issues with the syntax. The hierarchy with the corresponding mode declarations are shown in Figure 15. The declarations are given in the order of layers of the tree ensuring that parents are declared prior to children. As with the Bravo example in Figure 14, the nodes that represent realizable modes are depicted in the darker (green) color.

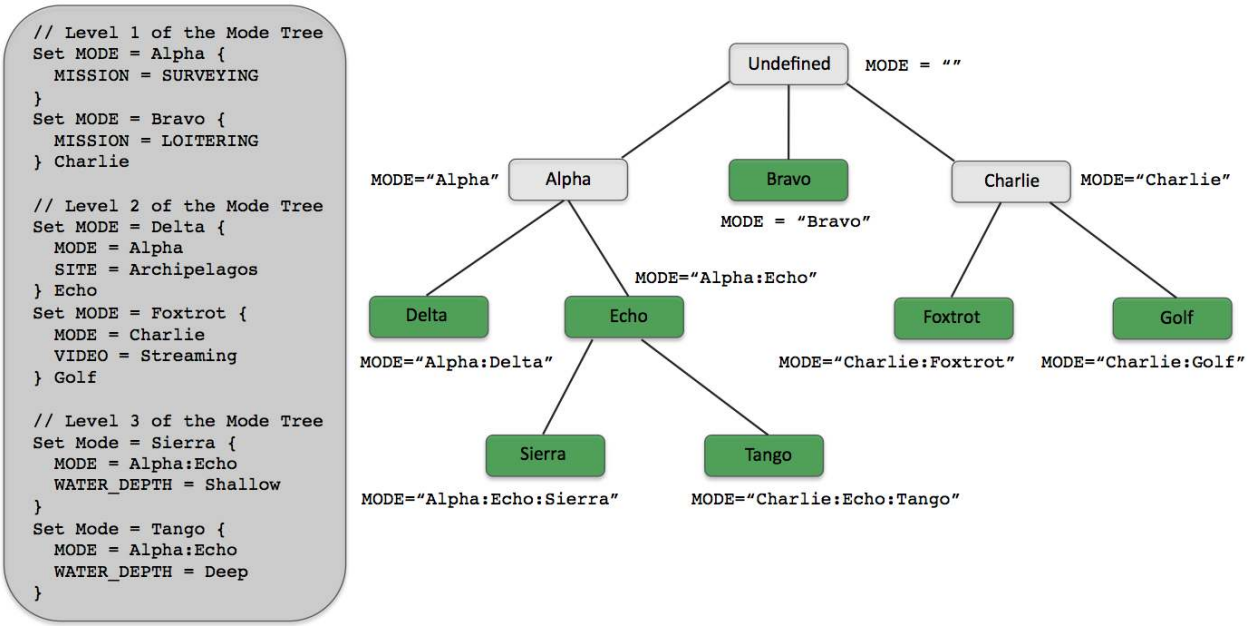

Figure 15: **Example Hierarchical Mode Declaration**: The hierarchy on the right is constructed from the set of mode declarations on the the left (with fictional conditions). Darker nodes represent modes that are realizable through some combination of conditions.

The `"Alpha"` mode for example is not realizable since it has the children `"Delta"` and `"Echo"`, with the latter being set as the `<else-value>` if the conditions of the former at not met. The `"Bravo"`

mode is realizable since it has no children. The "Echo" mode is realizable despite having children because the "Tango" mode is not the <else-value> of the "Sierra" mode declaration. For example, if the following three conditions hold, (a) "MISSION=SURVEYING", (b) "SITE!=Archipelagos", and (c) "WATER_DEPTH=Medium", then the value of the variable MODE would be set to "Alpha:Echo". Finally, note that the condition in the "Sierra" declaration, "MODE=Alpha:Echo", is specified fully, i.e., "MODE=Echo" would not achieve the desired result.

6.4.5 Monitoring the Mission Mode at Run Time

The mission mode can be monitored at run time in a couple ways. First, since the mode variable is posted as a MOOS variable, any MOOS scope tool will work, e.g., uXMS, uMS, uHelmScope. Using uHelmScope, the mission variable can be monitored as part of the basic MOOSDB scoping capability, but it is also displayed as part of the uHelmScope app, and in the AppCasting output of pHelmIvP.

The uHelmScope tool also has a mode in which the entire mode hierarchy may be rendered - solely to provide a visual confirmation that the hierarchy specified with the mode declarations in the behavior file does in fact correspond to what the user intended. Currently there are no tools to automatically render the mode hierarchy in a manner like the right hand side of Figure 15. The uHelmScope output for the example in Figure 15 is shown in listing 2 below.

Listing 6.2: The mode hierarchy output from uHelmScope for the example in Figure 15.

```
1   ModeSet Hierarchy:
2   -----------------------------------------------
3   Alpha
4       Delta
5       Echo
6           Sierra
7           Tango
8   Bravo
9   Charlie
10      Foxtrot
11      Golf
12  -----------------------------------------------
13  CURENT MODE(S): Charlie:Foxtrot
14
15  Hit 'r' to resume outputs, or SPACEBAR for a single update
```

More on this feature of the uHelmScope can be found in the uHelmScope documention, [**?**]. It's worth noting that poking the value of a mode variable will have no effect on the helm operation. The mission mode cannot be commanded directly. The mode variable is reset at the outset of the helm iteration, and the helm doesn't even register for mail on mode variables.

6.5 Behavior Participation in the IvP Helm

The primary work of the helm comes when the behaviors participate and do their thing, at each round of the helm Iterate() loop. As depicted in Figure 13, once the mode has been re-evaluated taking into consideration newly received mail, it is time for the behaviors (well, some at least) to step up and do their thing.

6.5.1 Behavior Run 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 criteria for participation is whether or not it has met each of its "run conditions". These are the conditions laid out in the behavior file of the form:

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

The `<logic-expression>` syntax is described in Appendix 22. 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_KEEP != true)
```

A run condition may also be expressed in terms of a helm mode, as described in the next Section 6.5.2 such as:

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

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

6.5.2 Behavior Run Conditions and Mode Declarations

The use of hierarchical mode declarations potentially simplify the expressions used as run conditions. The conditions in practice could be limited to:

```
condition = <mode-variable> = <mode-value>, or
condition = <mode-variable> == <mode-value>.
```

Conditions were used in this way with the Bravo mission in Listing 1, as an alternative to their usage in the Alpha mission example in Listing 1.

Note the use of the double-equals relation above. This relation is used for matching against the strings used to represent the hierarchical mode. The two strings match if the ordered components of one side are a subset of the ordered components of the other. Components are colon-separated. For example, using the illustrative hierarchy from Figure 15:

```
"Alpha:Echo:Sierra" == "Sierra"
"Alpha:Echo:Sierra" == "Echo:Sierra"
"Alpha:Echo:Sierra" == "Alpha"
          "Sierra" == "Alpha:Echo:Sierra"
  "Charlie:Foxtrot" == "Charlie:Foxtrot"

"Alpha:Echo:Sierra" != "Alpha:Sierra"
```

6.5.3 Behavior Run States

On any given helm iteration a behavior may be in one of four states depicted in Figure 16:

Figure 16: **Behavior States:** A behavior may be in one of these four states at any given iteration of helm `Iterate()` loop. The state is determined by examination of MOOS variables stored locally in the helm's information buffer.

- Idle: A behavior is `idle` if it is not `complete` and it has not met its run conditions as described above in Section **??**. 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 why a running behavior may not be active. For example, a collision avoidance behavior where the object of the behavior is sufficiently far away.
- Complete: A behavior is `complete` when the behavior itself determines it to be 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 some standard steps to be taken upon completion, such as posting of `endflags`, described below in Section 6.5.4. Once a behavior is in the `complete` state, it remains in that state permanently. All behaviors have a `duration` parameter defined to allow it to be configured to time-out if desired. When a time-out occurs the behavior state will be set to `complete`.

6.5.4 Behavior Flags and Behavior Messages

Behaviors may post some number of messages, i.e., variable-value pairs, on any given iteration (see Figure 13). These message can be critical for coordinating behaviors with each other and to other MOOS processes. The can also be invaluable for monitoring and debugging behaviors configured for particular missions. To be more accurate, behaviors don't post messages to the `MOOSDB`, they request the helm to post messages on its behalf. The helm collects these requests and publishes them to the `MOOSDB` at the end of the `Iterate()` loop. It also filters them for successive duplicates as discussed in Section 5.8.

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 several types, `endflag`, `idleflag`, `runflag`, `activeflag`, `inactiveflag`, and `spawnflag`. The variable-value pairs representing each flag are set in the behavior file for the corresponding behavior. See line 11 in Listing 1 for example.

- 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.
- idleflag: An idleflag is posted by the helm when the behavior enters 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 by the helm when the behavior enters the running state from the idle state. 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 by the helm when the behavior enters 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 by the helm when the behavior enters a state that is not 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.
- spawnflag: An spawnflag is posted by the helm when templated behavior is spawned. Examples include the collision and obstacle avoidance behaviors. The variable-value pair representing the spawnflag is given in the spawnflag parameter in the behavior file. Multiple spawnflags 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 17:

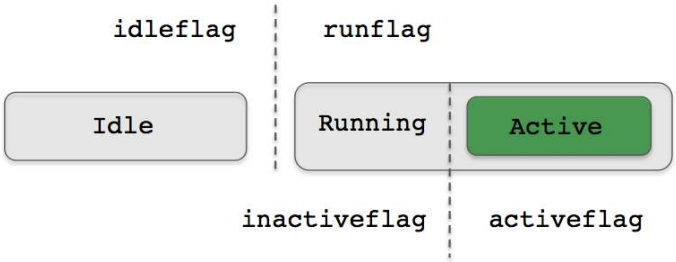

Figure 17: **Behavior Flags:** The four behavior flags idleflag, runflag, activeflag, and inactiveflag are posted depending on the behavior state and can be considered complementary in the manner indicated.

Behavior authors may implement their behaviors to post other messages as they see fit. For example the waypoint behavior used in the Alpha example in Section 16 also published the variable WPT_STAT 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, in seconds. (You might want to re-run the Alpha mission with uXMS scoping on this variable to watch it change as the mission unfolds.)

6.5.5 Monitoring Behavior Run States and Messages During Mission Execution

The run states for each behavior, are wrapped up on each iteration by the helm into a single string and published in the variable IVPHELM_SUMMARY. This variable is subscribed for by the uHelmScope tool and behavior states are parsed from this variable and summarized in the main output of uHelmScope, as in the below excerpt:

```
12  Behaviors Active: ---------- (1)
13    waypt_survey (13.0) (pwt=100.00) (pcs=1227) (cpu=0.01) (upd=0/0)
14  Behaviors Running: --------- (0)
15  Behaviors Idle: ------------ (1)
16    waypt_return (22.8)
17  Behaviors Completed: ------- (0)
```

Behaviors are grouped into the four possible states, with a summary line for each state, e.g., lines 12, 14, 15, 17, containing the number of behaviors in that state in parentheses at the end of the line. Each behavior configured for the helm shows up on a dedicated line in the appropriate group, e.g., lines 13 and 16. In these lines immediately following the behavior name, the number of seconds is displayed in parentheses indicating how long the behavior has been in that state.

6.6 Behavior Reconciliation in the IvP Helm - Multi-Objective Optimization

6.6.1 IvP Functions

IvP functions are produced by behaviors to influence the decision produced by the helm on the current iteration (see Figure 13). The decision is typically comprised of the desired `heading`, `speed`, and `depth` but the helm decision space could be comprised of any arbitrary configuration (see Section 5.4.6). Some points about IvP functions:

- 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 19).
- 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. Behaviors that produce functions solely over vehicle `depth` are perfectly ok.

How are IvP functions built? The IvP Build Toolbox is 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 represents the relationship between a candidate helm decision and the expected utility with respect to the behavior's objectives. The IvP Toolbox is not covered in detail in this document, but an overview is given below.

6.6.2 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 below in Figure 18:

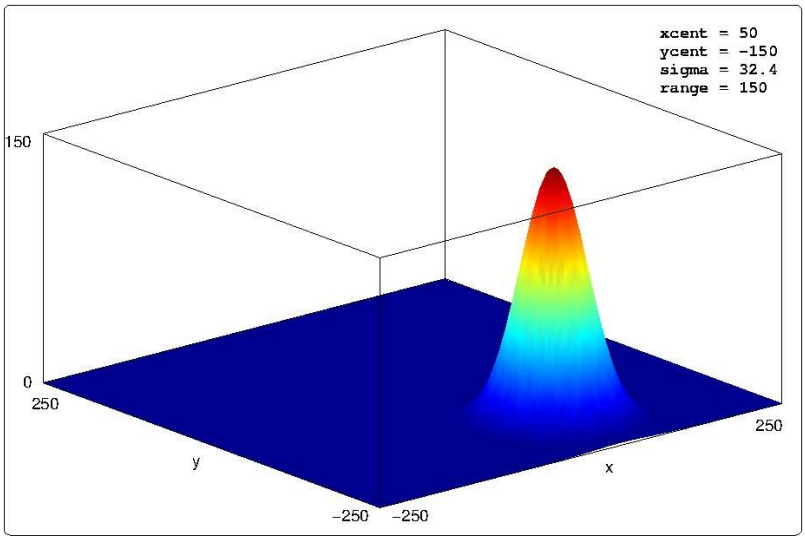

Figure 18: A rendering of the function $f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where $A =$ `range` $= 150$, $\sigma =$ `sigma` $= 32.4$, x_0 $=$ `xcent` $= 50$, $y_0 =$ `ycent` $= -150$. The domain here for x and y ranges from -250 to 250.

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 Figure 19. The only difference in these four piecewise function is the number

and size of the piece. More pieces (Figure 19 (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.

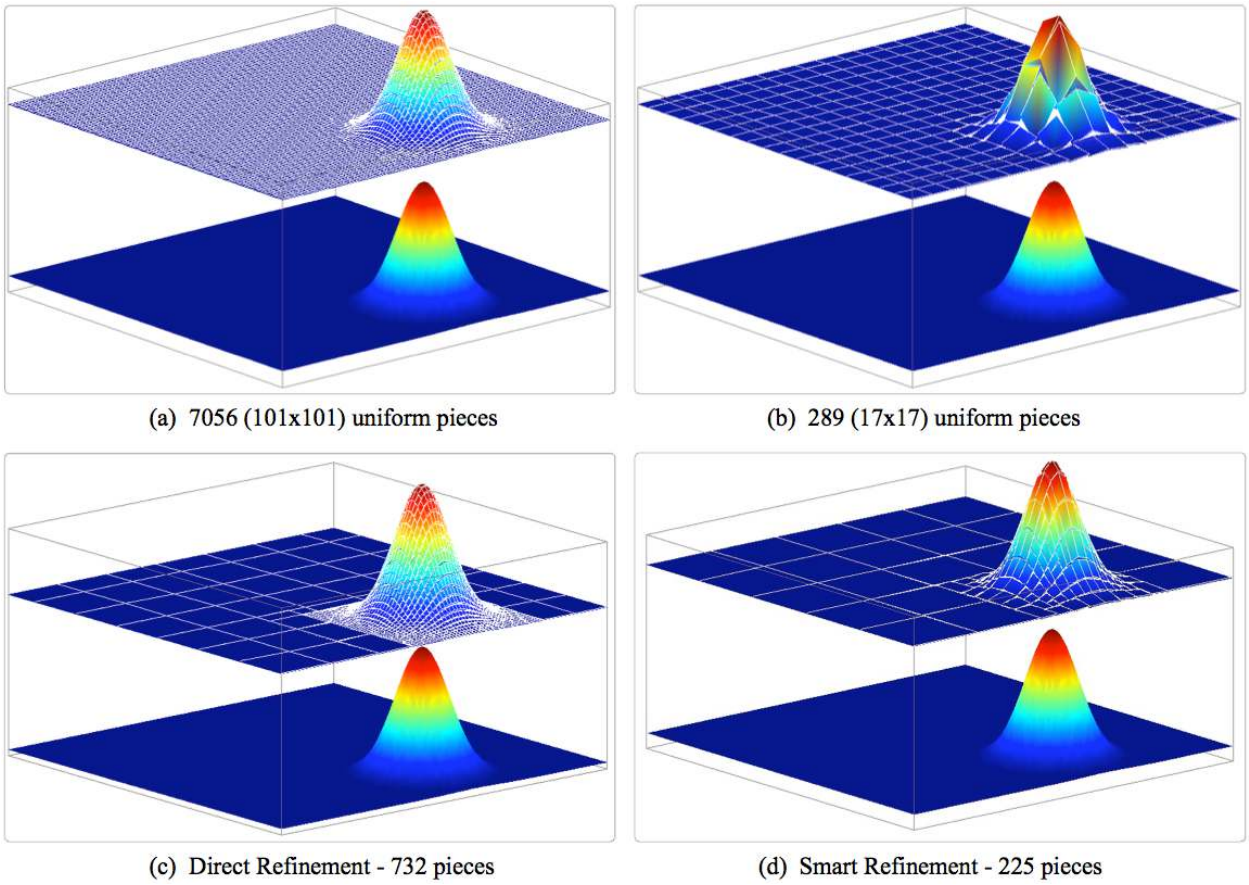

Figure 19: **A rendering of four different IvP functions approximating the same underlying function**: The function in (a) uses a uniform distribution of 7056 pieces. The function in (b) uses a uniform distribution of 1024 pieces. The function in (c) was created by first building a uniform distribution of 49 pieces and then focusing the refinement on a sub-domain 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.

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 sub-domain directed by the caller, with smaller uniform pieces of the caller's choosing. This is described more fully in the documentation for the IvP Build Toolbox. Using this tool requires the caller to have some idea where, in the sub-domain, 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 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 sub-domain for that piece. A priority queue is maintained based on the grades, and pieces where poor fits are noted, are automatically refined further, up to a maximum piece limit chosen by the caller. This is described more fully in the documentation for the IvP Build Toolbox

The Reflector tools work similarly in N dimensions and on multi-modal functions. The only requirement for using the Reflector tool is to provide it with access to the underlying function. Since 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 long pole in the tent.

6.6.3 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 $f_i(\overrightarrow{x})$, and the weight of each function is given by w_i, the solution to a problem with k functions is given by:

$$\vec{x}^* = \underset{\vec{x}}{\text{argmax}} \sum_{i=0}^{k-1} w_i f_i(\vec{x})$$

The algorithm is described in detail in [**?**], 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 comprised of an IvP function at each layer, and the nodes at each layer are comprised 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 sub-tree 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 to the IvP Helm which always finds the global optimum.

- *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. Upon casual observation this appears to provide a speed-up by about a factor of two.

In cases where there is a "tie" between optimal decisions, the solution generated by the solver is non-deterministic. This is mitigated somewhat by the fact that the solution is seeded with the output of the previous iteration as discussed above.

6.6.4 Monitoring the IvP Solver During Mission Execution

The performance of the solver can be monitored with the uHelmScope tool. The output shown below is an excerpt from an example mission. On line 5, the total time needed to solve the multi-objective optimization problem is given in seconds, and the max time need for all recorded loops is given in parentheses. It is zero here since there is only one objective function in this example. On line 6 is the total time for creating the IvP functions in all behaviors, with the max across all iterations in parentheses. On line 7 is the total loop time - the sum of the previous two lines. Active behaviors display useful information regarding the IvP solver. For example, on line 13, the Survey waypoint behavior had a priority weight of 100 and generated 1,227 pieces, taking 0.01 seconds of CPU time to create.

Listing 6.3: Example uHelmScope *output containing information about the IvP solver.*

```
 1  ==============   uHelmScope Report  ============== DRIVE  (17)
 2    Helm Iteration: 66      (hz=0.38)(5)  (hz=0.35)(66)  (hz=0.56)(max)
 3    IvP functions:  1
 4    Mode(s):        Surveying
 5    SolveTime:      0.00    (max=0.00)
 6    CreateTime:     0.02    (max=0.02)
 7    LoopTime:       0.02    (max=0.02)
 8    Halted:         false   (0 warnings)
 9  Helm Decision: [speed,0,4,21] [course,0,359,360]
10    speed = 3.00
11    course = 177.00
12  Behaviors Active: ---------- (1)
13    waypt_survey (13.0) (pwt=100.00) (pcs=1227) (cpu=0.01) (upd=0/0)
14  Behaviors Running: --------- (0)
15  Behaviors Idle: ------------ (1)
16    waypt_return (22.8)
17  Behaviors Completed: ------- (0)
18
```

The solver can be additionally monitored and analyzed through the two MOOS variables LOOP_CPU and CREATE_CPU published on each helm iteration. The former indicates the system wall time for building each IvP function and solving the multi-objective optimization problem, and the latter indicates just the time to create the IvP functions.

7 Properties of Helm Behaviors

The objective of this section is to describe properties common to all IvP Helm behaviors, describe how to overload standard functions for 3rd party behaviors, and to provide a detailed simple example of a behavior.

7.1 Brief Overview

Behaviors are implemented as C++ classes with the helm having one or more instances at runtime, each with a unique descriptor. The properties and implemented functions of a particular behavior are partly derived from the `IvPBehavior` superclass, shown in Figure 20. The is-a relationship of a derived class provides a form of code re-use as well as a common interface for constructing mission files with behaviors.

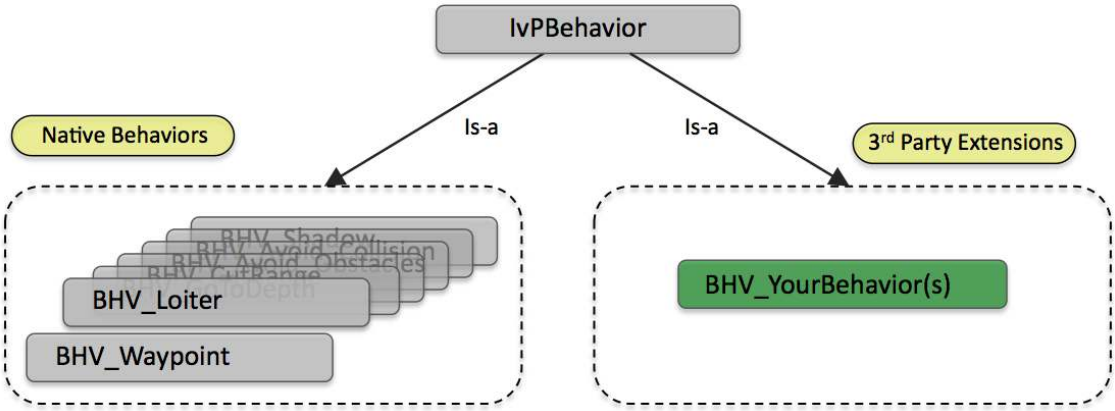

Figure 20: **Behavior inheritance**: Behaviors are derived from the `IvPBehavior` superclass. The native behaviors are the behaviors distributed with the helm. New behaviors also need to be a subclass of the IvPBehavior class to work with the helm. Certain virtual functions invoked by the helm may be 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 `IvPBehavior` class provides five virtual functions which 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.
- The `onRunState()` function: the meat 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.
- The `onIdleToRunState()` function: invoked once by the helm upon transitioning from the

idle to running state (compared to the `onRunState()` function which is invoked on *each* helm iteration where the behavior has met its conditions).

- The `onRunToIdleState()` function: invoked once by the helm upon transitioning from the running to idle state (compared to the `onIdleState()` function which is invoked on *each* helm iteration where the behavior has not met its conditions).

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.

7.2 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
```

The left-hand side, the parameter component, is case insensitive, while the value component is typically case sensitive. This was discussed in depth in Section **??**. In this section, the parameters defined at the superclass level and available to all behaviors are exhaustively listed and discussed. Each behavior typically augments these parameters with new ones unique to the behavior, and in the next section the issue of implementing new parameters by overloading the `setParam()` function is addressed.

7.2.1 A Summary of the Full Set of General Behavior Parameters

The following parameters are defined for all behaviors at the superclass level. They are listed here for reference - certain related aspects are discussed in further detail in other sections.

Listing 7.1: Configuration Parameters Common to All Behaviors.

`activeflag`: This parameter specifies a variable and a value to be posted when the behavior is in the *active* state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was not in the *avtive* 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.

`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 < 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. See Section **??** for more detail on run conditions.

configflag: More info TBD

comms_policy: This parameter governs inter-vehicle messaging for messages originating within a behavior. More info TBD.

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 non-idle) the first time. *Should the behavior switch between running and idle states, the clock keeps ticking even during the idle periods.* See Section 7.2.6 for more detail.

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. See Section 7.2.6 for more detail.

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. See Section 7.2.6 for more detail.

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 update/posted only when the behavior is in the running state. See Section 7.2.6 for more detail.

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 a 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.

idleflag: This parameter specifies a variable and a value to be posted when the behavior is in the *idle* state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was not 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.

inactiveflag: This parameter specifies a variable and a value to be posted when the behavior is *not* in the *active* state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was in the *active* state. It is a equal-separated pair such as OUT_OF_RANGE=true. More than one flag may be specified by the user in the behavior configuration. These can be used to satisfy or block the conditions of other behaviors.

`name`: The name of the behavior - should be unique between all behaviors. Due to the implementation of behavior templating, spawned behavior take on a new name by concatenating on the end of a base name. For this reason all configured behavior names could be regarded as base names. The check for uniqueness includes hypothetical extension. Thus the names `loiter` and `loiter_two` are not regarded as safe since the first could potentially grow into the latter with the contcatenation of `_two`. Logging and output sent to the helm console during operation will organize information by the behavior name.

`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 pair `"variable, ..., variable, value"`, where last component in the list is the time value given in seconds. See Section 7.2.9 for more detail.

`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. See Section 7.2.7 for more detail.

`post_mapping`: This parameter takes a comma-separated pair such as `WPT_STAT, WAYPT_STATUS` where 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 re-mappings 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`.

`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.

`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 only posted if the behavior, on the previous helm iteration, was not 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.

`runxflag`: More info TBD

`spawnflag`: More info TBD

`spawnxflag`: More info TBD

`templating`: The `templating` parameter may be used to turn a behavior specification into a template for spawning new behaviors dynamically after the helm has been launched. Instantiation requests are received via the `updates` parameter described in Section 7.7.

`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 runtime. The syntax for this string is a #-separated list of parameter-value pairs: `"param=value # param=value # ... # param=value"`. This is one of the primary hooks to the helm for mission control - the other being the behavior conditions described above. See Section 7.2.5 for more detail.

7.2.2 The `name` Parameter

The `name` parameter is a mandatory parameter with no default value that assigns a name to the behavior instance. The name must be unique. The behavior name is case sensitive, so having the names `"return_home"` and `"Return_Home"` is acceptable (although not advisable).

Due to the capability of *behavior templating* (Section 7.7) the policy of behavior uniqueness was expanded to include pairs of names where one name matches the beginning component of another. For example `"return"` and `"return_home"` are not allowed. This is due to way template spawning is implemented. The spawning process begins by setting the name of the newly spawned behavior. This is achieved by appending to the *base name* declared for the behavior template. By allowing, for example, `"return"` and `"return_home"`, it leaves open the possiblity that the first behavior may spawn an instance with a name clash if it were spawned with a suffix directive of `"_home"`.

Behavior names have no relation to the behavior class name. For example an instance of the `Waypoint` behavior may be called `"station-keep"` (although not advisable). Furthermore, multiple instances of the same behavior may certainly co-exist so long as each instance has a unique name. If the helm is configured with one or more behaviors with non-unique names, the helm will come up in the `"MALCONFIG"` state. The terminal or appcasting output would look something like:

```
=====================================================================
pHelmIvP alpha                                              1/0(411)
=====================================================================
Configuration Warnings: 1
 [1 of 1]: Duplicate behavior name found: waypt_return

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   !!                                            !!
   !!  The helm is in the MALCONFIG state due to  !!
   !!  unresolved configuration warnings.         !!
   !!                                            !!
   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

7.2.3 The `priority` Parameter

The `priority` parameter is a optional parameter with a default value of 100. It assigns a priority weight to be applied to an objective function produced by the behavior. Many behaviors interpret this priority to establish a range of priorities depending on the further evaluation of the situation by the behavior. For example, the collision avoidance behavior will treat the configured priority to

be the maximum priority (when a contact is close) and will degrade the priority accordingly as the contact range is greater.

The range of acceptable values is from zero to infinity. If a negative value is given for a behavior the helm will come up in the "MALCONFIG" state. The terminal or appcasting output would look something like:

```
=====================================================================
pHelmIvP alpha                                              1/0(21)
=====================================================================
Configuration Warnings: 1
 [1 of 1]: alpha.bhv: Line 61: pwt   = -1

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   !!                                                !!
   !!  The helm is in the MALCONFIG state due to     !!
   !!  unresolved configuration warnings.            !!
   !!                                                !!
   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

7.2.4 The post_mapping Parameter

The post_mapping parameter is a optional parameter with no default value. It allows the user to configure a behavior to change the name of the variables otherwise posted by the behavior. For example, the Waypoint behavior by default publishes a variable WPT_INDEX indicating the index of the next point on its path. If there are multiple waypoint behaviors, or if you just don't like this variable, it may be change for example to SURVEY_INDEX with the following configuration:

```
post_mapping=WPT_INDEX,SURVEY_INDEX
```

7.2.5 Altering 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 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 2. The return point is the (0,0) point in local coordinates, and return speed is 2.0 meters/second. When the conditions are met, this is what will be executed.

Listing 7.2: An example behavior configuration using the updates parameter.

```
 1  Behavior = BHV_Waypoint
 2  {
 3    name      = WAYPT_RETURN
 4    priority  = 100
 5    speed     = 2.0
 6    radius    = 8.0
 7    points    = 0,0
 8    updates   = RETURN_UPDATES
 9    condition = RETURN = true
10    condition = DEPLOY = true
11  }
```

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 8 in Listing 2). The syntax for this variable is of the form:

```
parameter = value # 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 helm iteration loop for a behavior (Figure 20) 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.

7.2.6 Limiting Behavior Duration with the `duration` Parameter

The `duration` parameter specifies a time period in seconds before a behavior times out and permanently enters the completed state (Figure 21). If left unspecified, there is no time limit to the behavior. By default, the duration clock begins ticking as soon as the helm is in drive. The duration clock remains ticking when or if the behavior subsequently enters the idle state. It even remains ticking if the helm temporarily parks. When a timeout occurs, end flags are posted. The behavior can be configured to post the time remaining before a timeout with the `duration_status` parameter. The forms for each are:

```
duration        =  value (positive numerical)
duration_status =  value (variable name)
```

Note that the duration status variable will only be published/updated when the behavior is in the running state. The duration status is rounded to the nearest integer until less than ten seconds remain, after which the time is posted out to two decimal places. The behavior can be configured to have the duration clock pause when it is in the idle state with the following:

```
duration_idle_decay = false   // The default is true
```

Configured in the above manner, a behavior's duration clock will remain paused until it's conditions are met. The behavior may also be configured to allow for the duration clock to be reset upon the writing of a MOOS variable with a particular value. For example:

```
duration_reset = BRAVO_TIMER_RESET=true
```

The behavior checks for and notes that the variable-value pair holds true and the duration clock is then reset to the original duration value. The behavior also marks the time at which the variable-value pair was noted to have held true. Thus there is no need to "un-set" the variable-value pair, e.g., setting `BRAVO_TIMER_RESET=false`, to allow the duration clock to resume its count-down.

7.2.7 The `perpetual` Parameter

When a behavior enters the *completed* state, it by default remains in that state with no chance to change. When the `perpetual` parameter is set to true, a behavior that is declared to be complete does not actually enter the complete state but performs all the other activity normally associated with completion, such as the posting of end flags. See Section 6.5.4 for more detail on posting flags to the `MOOSDB` from the helm. The default value for `perpetual` is false. The form for this parameter is:

```
perpetual = value
```

The value component is case insensitive, and the only legal values are either true or false. A behavior using the `duration` parameter with `perpetual` set to true will post its end flags upon time out, but will reset its clock and begin the count-down once more the next time its run conditions are met, i.e., enters the running state. Typically when a behavior is used in this way, it also posts an endflag that would put itself in the idle state, waiting for an external event.

7.2.8 The `post_mapping` Parameter

This parameter takes a comma-separated pair where 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. For example, the Waypoint behavior normally posts the variable `WPT_STAT` when it is running. This can be changed with:

```
post_mapping = WPT_STAT, WAYPT_STATUS
```

Note 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 re-mappings 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
```

7.2.9 Detection of Stale Variables with the nostarve Parameter

A behavior utilizing a variable generated by a MOOS process outside the helm, may require the variable to be sufficiently up-to-date. The staleness of a variable is the time since it was last written to by any process. The nostarve parameter allows the mission writer to set a staleness threshold. The form for this parameters is:

```
nostarve = variable_1, ..., variable_n, duration
```

The value of this parameter is a comma-separated list such as nostarve = NAV_X, NAV_Y, 5.0. The variable components name MOOS variables and the duration component, the last entry in the list, represents the tolerated staleness in seconds. If staleness is detected, a behavior failure condition is triggered which will trigger the helm to post all-stop values and relinquish to manual control.

7.3 Overloading the setParam() Function in New Behaviors

The setParam() function is a virtual function defined in the IvPBehavior class, with parameters implemented in the superclass (Section 7.2) handled in the superclass version of this function:

```
bool IvPBehavior::setParam(string parameter, string value);
```

The setParam() function should return true if the parameter is recognized and the value is in an acceptable form. In the rare case that a new behavior has no additional parameters, leaving this function undefined in the subclass is appropriate. The example below in Listing 3 gives an example for a fictional behavior BHV_YourBehavior having a single parameter period.

Listing 7.3: Example setParam() *implementation for fictional BHV_YourBehavior.*

```
1  bool BHV_YourBehavior::setParam(string param, string value)
2  {
3    if(param == "period") {
4      double time_value = atof(value.c_str());
5      if((time_value < 0) || (!isNumber(value)))
6        return(false);
7      m_period = time_value;
8      return(true);
9    }
10   return(false);
11 }
```

Since the period parameter refers to a time period, a check is made on line 4 that the value component indeed is not a negative number. The atof() function on line 6, which converts an ASCII string to a floating point value, returns zero when passed a non-numerical string, therefore the isNumber() function is also used to ensure the string represented by value represents a numerical value. The isNumber() function is part of the IvP utility library. A behavior implementation of this function without sufficient syntax or semantic checking simply runs the risk that faulty parameters are not detected at the time of helm launch, or during dynamic updates. Solid checking in this function will reduce debugging headaches down the road.

7.4 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 are shown in Figure 21. These 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.

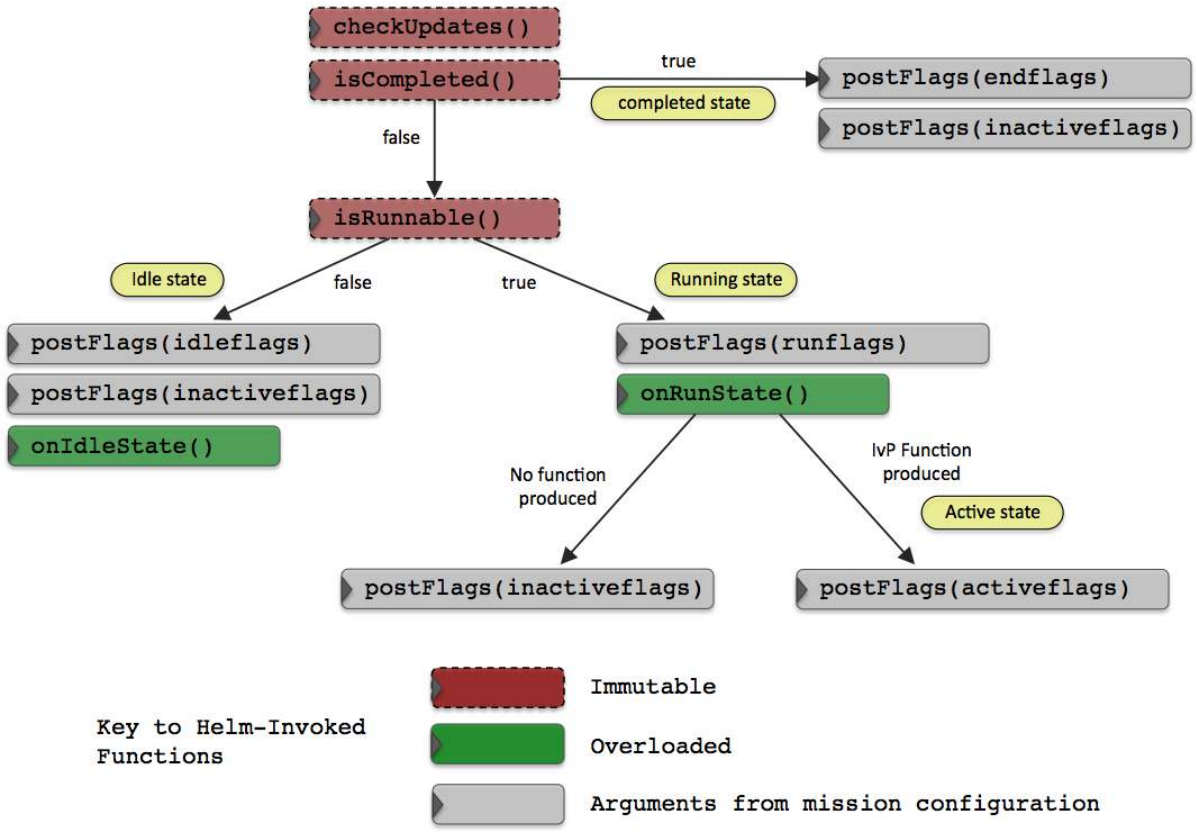

Figure 21: **Behavior function-calls by the helm**: The helm invokes a sequence of functions on each behavior on each iteration of the helm. The sequence of calls is dependent on what the behavior returns, and reflects the behaviors activity state. Certain functions are immutable and can not be overloaded by a behavior author. Two key functions, `onRunState()` and `onIdleState()` can be indeed 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.

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, 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 21:

- Helm-invoked immutable functions - functions invoked by the helm on each iteration that the author of a new behavior may not re-implement.
- Helm-invoked overloadable functions - functions invoked by the helm that an author of a new behavior typically re-implements of 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.

7.4.1 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 21 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 7.2.5 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 if a behavior is in the *running* state or not. 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 (Section 7.2.9). (c) the run conditions must be met. (d) the behavior's decision domain (IvP domain) is a proper subset of the helm's configured IvP domain. See Section **??** for more detail on run conditions.

- `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 since the flags are specified in the mission configuration `*.bhv` file. See Section 6.5.4 for more detail on posting flags to the `MOOSDB` from the helm.

7.4.2 Helm-Invoked Overloaded Functions

These are functions called by the helm. They 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 three functions.

- `IvPFunction* onRunState()`: This function is called by the helm when deemed to be in the *running* state (Figure 21). The bulk of the work in implementing a new behavior is in this function implementation, and is the subject of Section 7.6.

- `void onIdleState()`: This function is called by the helm when deemed to be in the *idle* state (Figure 21). Many behaviors are implemented with this function left undefined, but it is a useful hook to have in many cases.

- `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.

7.5 Local Behavior Utility Functions

The bulk of the work done in implementing a new behavior is in the implementation of the `onIdleState()` and `onRunState()` functions. The utility functions described below are designed to aid in that implementation and are generally "protected" functions, that is callable only from within the code of another function in the behavior, such as the `onRunState()` and `onIdleState()` functions, and not invoked by the helm.

7.5.1 Summary of Implementor-Invoked Utility Functions

The following is summary of utility functions implemented at the `IvPBehavior` superclass level.

- `void setComplete()`: The notion of what it means for a behavior to be "complete" is largely an issue specific to an individual behavior. When or if this state is reached, a call to `setComplete()` can be made and end flags will be posted, and the behavior will be permanently put into the *completed* state unless the `perpetual` parameter is set to true. See Section 6.5.3 for more on behavior run states.

- `void addInfoVars(string var_names)`: The helm will register for variables from the `MOOSDB` on a need-only basis, and a behavior is obligated to inform the helm that certain variables are needed on its behalf. A call to the `addInfoVars()` function can be made from anywhere with a behavior implementation to declare needed variables. This can be one call per variable, or the string argument can be a comma-separated list of variables. The most common point of invoking this function is within a behavior's constructor since needed variables are typically known at the point of instantiation. More on this issue in Section 7.5.3.

- `double getBufferDoubleVal(string varname, bool& result)`: Query the `info_buffer` for the latest (double) value for a given variable named by the string argument. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

- `double getBufferStringVal(string varname, bool& result)`: Query the `info_buffer` for the latest (string) value for a given variable named by the string argument. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

- `double getBufferCurrTime()`: Query the `info_buffer` for the current buffer local time, equivalent to the duration in seconds since the helm was launched. More on this in Section 7.5.2.

- `vector<double> getBufferDoubleVector(string var, bool& result)`: Query the `info_buffer` for all changes to the variable (of type double) named by the string argument, since the last iteration. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

- `vector<string> getBufferStringVector(string var, bool& result)`: Query the `info_buffer` for all changes to the variable (of type string) named by the string argument, since the last iteration. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

- `void postMessage(string varname, string value, string key)`: The helm can post messages (variable-value pairs) to the `MOOSDB` at the end of the helm iteration. Behaviors can request such postings via a call to the `postMessage()` function where the first argument is the variable name, and the second is the variable value. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

- `void postMessage(string varname, double value, string key)`: Same as above except used when the posted variable is of type `double` rather than `string`. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

- `void postBoolMessage(string varname, bool value, string key)`: Same as above, except used when the posted variable is a `bool` rather than `string`. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

- `void postIntMessage(string varname, double value, string key)`: Same as `postMessage(string, double)` above except the numerical output is rounded to the nearest integer. This, combined with the helm's use of the *duplication filter*, can reduce the number of posts to the `MOOSDB`. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

- `void postWMessage(string warning_msg)`: Identical to the `postMessage()` function except the variable name is automatically set to `BHV_WARNING`. Provided as a matter of convenience to the caller and for uniformity in monitoring warnings.

- `void postEMessage(string error_msg)`: Similar to the `postWMessage()` function except the variable name is `BHV_ERROR`. This call is for more serious problems noted by the behavior. It also results in an internal `state_ok` bit being flipped which results in the helm posting all-stop values to the actuators.

- `void postRepeatableMessage(string varname, string value)`: A convenience function. A posting of `postRepeatableMessage("FOO", "bar")` is equivalent to `postMessage("FOO", "bar", "repeatable")`.

- `void postRepeatableMessage(string varname, double value)`: A convenience function. A posting of `postRepeatableMessage("FOO", 100)` is equivalent to `postMessage("FOO", 100, "repeatable")`.

7.5.2 The Information Buffer

Behaviors do not have direct access to the MOOSDB - they don't read mail, and they don't post changes directly, but rather through the helm as an intermediary. The information buffer, or `info_buffer`, is a data structure maintained by the helm to reflect a subset of the information in the MOOSDB and made available to each behavior. This topic is hidden from a user configuring existing behaviors and can be safely skipped, but is an important issue for a behavior author implementing a new behavior. The `info_buffer` is a data structure shared by all behaviors, each behavior having an pointer to a single instance of the `InfoBuffer` class. This data structure is maintained by the helm, primarily by reading mail from the MOOSDB and reflecting the change onto the buffer on each helm iteration, before the helm requests input from each behavior. Each behavior therefore has the exact same snapshot of a subset of the MOOSDB. A behavior author needs to know two things - how to ensure that certain variables show up in the buffer, and how to access that information from within the behavior. These two issues are discussed next.

7.5.3 Requesting the Inclusion of a Variable in the Information Buffer

A variable can be specifically requested for inclusion in the `info_buffer` by invoking the following function:

```
void  IvPBehavior::addInfoVars(string varnames)
```

The string argument is either a single MOOS variable or a comma-separated list of variables. Duplicate requests are simply ignored. Typically such calls are invoked in a behavior's constructor, but may be done dynamically at any point after the helm is running. The helm will simply register with the MOOSDB for the requested variable at the end of the current iteration. Certain variables are registered for automatically on behalf of the behavior. All variables referenced in run conditions will be registered and accessible in the buffer. Variables named in the `updates` and `nostarve` parameters will also be automatically registered.

7.5.4 Accessing Variable Information from the Information Buffer

A variable value can be queried from the buffer with one of the following two function calls, depending on whether the variable is of type double or string.

```
string  IvPBehavior::getBufferStringVal(string varname, bool& result)
double  IvPBehavior::getBufferDoubleVal(string varname, bool& result)
```

The first string argument is the variable name, and the second argument is a reference to a Boolean variable which, upon the function return, will indicate whether the queried variable was found in the buffer. A duration value indicating the elapsed time since the variable was last changed in the buffer can be obtained from the following function call:

```
double  IvPBehavior::getBufferTimeVal(string varname);
```

The string argument is the variable name. The returned value should be exactly zero if this variable was updated by new mail received by the helm at the beginning of the current iteration. If the variable name is not found in the buffer, the return value is -1. The "current" buffer time, equivalent

to the cumulative time in seconds since the helm was launched, can be retrieved with the following function call:

```
double  IvPBehavior::getBufferCurrTime()
```

The buffer time is a local variable of the `info_buffer` data structure. It is updated once at the beginning of the helm `OnNewMail()` loop prior to processing all new updates to the buffer from the MOOS mail stack, or at the beginning of the `Iterate()` loop if no mail is processed on the current iteration. Thus the time-stamp returned by the above call should be exactly the same for successive calls by all behaviors within a helm iteration.

The values returned by `getBufferStringVal()` and `getBufferDoubleVal()` represent the latest value of the variable in the `MOOSDB` at the point in time when the helm began its iteration and processed its mail stack. The value may have changed several times in the `MOOSDB` between iterations, and this information may be of use to a behavior. This is particularly true when a variable is being posted in pieces, or a sequence of delta changes to a data structure. In any event, this information can be recovered with the following two function calls:

```
vector<string>  IvPBehavior::getBufferStringVector(string varname, bool& result)
vector<double>  IvPBehavior::getBufferDoubleVector(string varname, bool& result)
```

They return all values updated to the buffer for a given variable since the last iteration in a vector of strings or doubles respectively. The latest change is located at the highest index of the vector. An empty vector is returned if no changes were received at the outset of the current iteration.

7.6 Overloading the onRunState() and onIdleState() Functions

The `onRunState()` function is declared as a virtual function in the `IvPBehavior` superclass intended to be overloaded by the behavior author to accomplish the primary work of the behavior. The primary behavior output is the objective function. This is what drives the vehicle. The objective function is an instance of the class `IvPFunction`, and a behavior generates an instance and returns a pointer to the object in the following function:

```
IvPFunction* onRunState()
```

This function is called automatically by the helm on the current iteration if the behavior is deemed to be in the *running* state, as depicted in Figure 21. The invocation of `onRunState()` does not necessarily mean an objective function is returned. The behavior may opt not to for whatever reason, in which case it returns a null pointer. However, if it does generate a function, the behavior is said to be in the *active* state. The steps comprising the typical implementation of the `onRunState()` implementation can be summarized as follows:

- Get information from the `info_buffer`, and update any internal behavior state.
- Generate any messages to be posted to the `MOOSDB`.
- Produce an objective function if warranted.
- Return.

The same steps hold for the onIdleState() function except for producing an objective function. The first two steps have been discussed in detail. Accessing the info_buffer was described in Sections 7.5.2 - 7.5.4. The functions for posting messages to the MOOSDB from within a behavior were discussed in Section 7.5.1. Further issues regarding the posting of messages were covered in Section 6.5.4 The remaining issue to discuss is how objective functions are generated. This is covered in the IvPBuild Toolbox documentation.

7.7 Dynamic Behavior Spawning

In certain scenarios it may not be practical or possible to know in advance all the behaviors needed to accomplish mission objectives. For example, if the helm uses a certain kind of behavior to deal with another vehicle in its operation area, for collision avoidance or trailing etc., the identities or number of such vehicles may not be known when the mission planner is configuring the helm's behavior file. One way to circumvent this problem is to design a collision avoidance behavior, for example, to handle all known contacts. However, this has a couple drawbacks. It would entail a degree of multi-objective optimization be implemented within the behavior to produce an objective function that was comprehensive across all contacts. This would likely be much more computationally expensive than simply generating an objective function for each contact. It also may be advantageous to have different types of collision avoidance behaviors for different contact types or collision avoidance protocols. In any event, the helm support for dynamic behavior spawning gives behavior architects and mission planners another potentially powerful option for implementing an autonomy system.

7.7.1 Behavior Specifications Viewed as Templates

The templating parameter may be used to turn an otherwise static behavior specification into a template for spawning new behaviors dynamically after the helm has been launched. Instantiation requests are received via the updates parameter described in Section 7.2.5. Updates received through this variable are normally used to change behavior parameters dynamically, but they can be further used to request the spawning of a new behavior by including the following component:

```
name = <new-behavior-name>
```

If the <new-behavior-name> is not the name of the behavior given in the behavior specification, and if it is not the name of a behavior already presently instantiated by the helm, the helm interprets this as a request to spawn a new behavior, if templating is enabled. Templating is enabled by including the following component in the behavior specification:

```
templating = <templating-mode>
```

The <templating-mode> may be set to either "disallowed" (the default), "clone", or "spawn". In the "clone" mode, the helm will instantiate a behavior immediately upon helm startup. In the "spawn" mode, the helm will not instantiate a behavior until it receives a request to do so via the updates parameter as described above. An example of a behavior configured to allow dynamic spawning is given in Listing **??**, taken from the Berta example mission.

For a behavior configured with templating enabled in the "spawn" mode, the helm will *not* spawn a behavior at the helm startup time. However, internally it will indeed spawn such a behavior, check that it can be found and built as configured, and then destroy it immediately. This means that the behavior configuration found in the .bhv configuration file must not have an invalid configuration. It is preferable to know at helm launch time that a behavior is misconfigured, rather than waiting for the spawning event to occur perhaps hours into a mission and being surprised that a critical behavior, such as collision avoidance, failed to be spawned.

7.7.2 Behavior Completion and Removal from the Helm

All behaviors, whether statically spawned upon helm startup, or dynamically spawned during the mission, are capable of dying and being removed from the helm. Death and removal are part of the consequences of a behavior entering the completed state. Behavior run states were discussed in Section 6.5.3. A completed behavior configured with perpetual=true will not die upon completion. Once a behavior dies, its name is removed from the helm's internal registry of currently-spawned behaviors and a new behavior by the same name may be spawned at a future time.

7.7.3 Example Missions with Dynamic Behavior Spawning

Two example missions are provided that demonstrate the workings of dynamic behavior spawning, The Echo mission in Section 19, and the Berta mission in Section **??**. The Echo mission involves a single vehicle with its helm configured to spawn dynamic behaviors of the type BHV_BearingLine. These behaviors do nothing more than post a viewable line segment to the MOOSDB between ownship and a point in the operation area. The interesting thing about this example is that the mission is configured with an event script (via uTimerScript) to automatically cue the spawning of 5000 behaviors over about one hour. Each behavior has a random duration of less than a minute, so behaviors are spawning and dying quite rapidly with visual confirmation via the viewable line segments.

The second example mission, the Berta mission, involves two vehicles that are loitering near one another. Periodically their loiter assignments are randomly altered (again through uTimerScript). The change in loiter locations repeatedly puts them on an unpredictable and random near-collision course and each vehicle needs to spawn a collision avoidance behavior. The interesting thing about this scenario is that the behavior, the BHV_AvoidCollision behavior, is an actual behavior of common use, unlike the BHV_BearingLine behavior used in the Echo mission. This example also uses the pBasicContactMgr to coordinate the receiving of contact reports with helm behavior spawning.

7.7.4 Examining the Helm's Life Event History

Behavior spawning, and behavior completion and removal from the helm, are two types of *life events* the helm takes note of and posts in the MOOS variable IVPHELM_LIFE_EVENT. A third type of life event occurs when a behavior spawning is aborted due to either a syntax error or a name collision. Monitoring life events at run time is possible by scoping on the variable IVPHELM_LIFE_EVENT with either uXMS or uMS. A better method is available via the uHelmScope application. It automatically registers for the IVPHELM_LIFE_EVENT variable and will generate a formatted report like that shown in Listing 4. In the post-mission analysis phase, the aloghelm application may be used to examine the life event history and will generate the same formatted report from a given alog file.

Listing 7.4: A Life Event History generated with either the uHelmScope or aloghelm utilities.

```
 1        ************************************************
 2        *        Summary of Behavior Life Events         *
 3        ************************************************
 4
 5  Time    Iter  Event  Behavior      Behavior Type       Spawning Seed
 6  ------  ----  -----  ------------  ------------------  ----------------------------
 7   47.84     1  spawn  loiter        BHV_Loiter          helm startup
 8   47.84     1  spawn  waypt_return  BHV_Waypoint        helm startup
 9   47.84     1  spawn  station-keep  BHV_StationKeep     helm startup
10  101.79   218  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
11  161.20   423  death  avd_henry     BHV_AvoidCollision
12  297.07   969  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
13  351.80  1159  death  avd_henry     BHV_AvoidCollision
14  461.37  1599  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
15  516.51  1795  death  avd_henry     BHV_AvoidCollision
16  644.94  2311  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
17  704.31  2517  death  avd_henry     BHV_AvoidCollision
18  730.02  2620  abort                BHV_AvoidCollision  name=avd_henry#foo=bar
19  825.17  3002  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
```

The life event history shown in Listing 4 was taken from the Berta example mission, the "gilda" vehicle, described in Section **??**. The time-stamp reported in column one is the elapsed time between the event and the time of the helm's startup. The first three events, in lines 6-8, reflect the three static behaviors spawned when the helm was launched, in first iteration of the helm. The collision avoidance behavior was spawned each time (lines 9, 11, 13 etc.) the vehicle "henry" came within sufficiently close range. Each time the "henry" vehicle passed and opened range to a sufficient amount, the collision avoidance behavior completed and died (lines 10, 12, 14, etc.). Line 17 shows an example of an aborted spawning. This was brought about purposely by poking the MOOSDB with the Spawning Seed shown for that line. Since the collision avoidance parameter does not have a parameter "foo", the spawning failed.

Accessing the life event history via uHelmScope may be done by launching the scope with the vehicle's mission file, and hitting the 'L' key to toggle into the life event history mode. Or one may launch the scope directly into this mode via:

```
uHelmScope --life targ_gilda.moos}
```

The same summary may also be accessed after mission completion via the log files:

```
aloghelm --life gilda.alog
```

Note that perhaps not all life events will be displayed when using uHelmScope, depending on when it is launched relative to pHelmIvP. When uHelmScope connects to the MOOSDB it will only receive the latest and all following posts to the variable IVPHELM_LIFE_EVENT. If uHelmScope connects after pHelmIvP is launched and put into drive, it may have missed older postings. The initial spawning events do not occur in the helm until the helm enters the DRIVE state. (See Section 5.2 about helm state). In the example in Listing 4, the helm apparently was in the PARK state for about 48 seconds

before it was put into drive and began to execute its first iteration. The full event history should always be accessible via the log file however.

8 Extending MOOS-IvP By Example

8.1 Brief Overview

This section describes an example repository distributed with the MOOS-IvP software bundle at www.moos-ivp.org. This repository merely provides a template with an example MOOS application, IvP Behavior, and example mission. More importantly perhaps is that the CMake build files are provided. A cursory look at these files reveal the hooks to add a new behavior or application. This is meant to provide one easy way to begin extending the MOOS-IvP software capabilities with one's own modules.

8.2 Obtaining and Building the Example Extensions Folder

The example extensions folder is available at the following URL:

```
http://www.moos-ivp.org/software/extensions.html
```

Instructions are provided for downloading the software from an SVN server with anonymous read-only access. After checking out the tree from the SVN server as prescribed at this link, the top level directory should have the following structure:

```
$ cd moos-ivp-extend
$ ls
moos-ivp-extend/
   bin/
   docs/
   missions/
   src/
```

The build instructions are maintained in the README files and are probably more up to date than this document. In short building the software amounts to two steps:

```
$ cd moos-ivp-extend/src/
$ cmake ./
$ make
```

The build depends on the directory `moos-ivp-extend` being in the same directory as `moos-ivp`. If this needs to be different on your system, the file `CMakeLists.txt` in the `src/` directory can be edited. The relevant lines are at the top of the file:

```
GET_FILENAME_COMPONENT(MOOS_BASE_DIR_A    ../../moos-ivp/trunk/MOOS  ABSOLUTE)
GET_FILENAME_COMPONENT(IVP_BASE_DIR_A     ../../moos-ivp/trunk/ivp   ABSOLUTE)
GET_FILENAME_COMPONENT(MOOS_BASE_DIR_B    ../../moos-ivp/MOOS        ABSOLUTE)
GET_FILENAME_COMPONENT(IVP_BASE_DIR_B     ../../moos-ivp/ivp         ABSOLUTE)
```

After building the software there should be a new MOOS application called pXRelayTest in the `bin/` directory, and a new IvP Behavior in the directory `src/lib_behaviors-test/` directory. The new behavior is in the form of a shared object, having the name `libBHV_SimpleWaypoint.so` in Linux, and `libBHV_SimpleWaypoint.dylib` on the Mac OS X platform.

8.3 Using the New MOOS Application

To use the new MOOS application, the directory `moos-ivp-extend/bin/` needs to be added to the user's shell path. This is typically done in the `.cshrc` or `.bashrc` file for tcsh and bash users respectively. To confirm that things are ready to go, use the built-in shell command `which`:

```
$ which pXRelayTest
```

which returns the directory where the executable resides if it is indeed in the shell's path. Otherwise it returns nothing. Don't forget that an edited path doesn't take effect until a new shell is launched or unless the user types `"source .cshrc"`, or `"source .bashrc"`.

The `pXRelayTest` application is the same as the `pXRelay` application distributed with the MOOS-IvP software bundle. It differs only in name for the sake of illustrating the process of building a new application outside the `moos-ivp` tree. This example MOOS application is described in detail in Section 4.8. In that section, an example mission file is described for running two `pXRelay` processes to illustrate their function. A similar mission file is provided in:

```
$ moos-ivp-extend/missions/xrelay/xrelay.moos
```

that launches two processes, `pXRelay` and `pXRelayTest` as a way of confirming that you are running a MOOS application from the extensions build alongside the build of the main moos-ivp repository. Information on how to work through this example is provided in Sections 4.8.2 and 4.8.3.

8.4 Using the New IvP Helm Behavior

To use the new IvP Helm behavior built in the extensions folder, the helm needs to know about it. The helm already contains a number of behaviors compiled in to the `pHelmIvP` executable, but the objective of adding behaviors in the manner outlined here, is to avoid any recompiling of the helm as new behaviors are added. Loosely speaking, there is a one-way dependency between repositories - new behaviors are layered onto the set of behaviors shipped with the helm with no modifications or re-build required of the basic moos-ivp software tree.

Newly built behaviors are compiled in to shared object files, `*.so` in Linux, and `*.dylib` in Mac OS X. The helm references a path variable called `IVP_BEHAVIOR_DIRS` which contains a colon-separated list of all directories containing dynamically loadable behaviors. This variable is a shell environment variable and is typically set in the `.cshrc` or `.bashrc` file for tcsh and bash users respectively. For example, the following lines in the `.bashrc` file for bash users:

```
export IVP_BEHAVIOR_DIRS=/home/bob/moos-ivp-extend/src/lib_behaviors-test
```

A mission file to test this is provided in:

```
moos-ivp-extend/missions/alder/alder.moos
```

The mission is launched with:

```
$ cd moos-ivp-extend/missions/alder/
$ pAntler alder.moos
```

The output produced in the helm terminal window should look like that shown in Listing 1 below, and provides useful feedback on whether the dynamically loadable behavior was loaded properly.

Listing 8.1: Example pHelmIvP terminal output when loading a dynamic behavior.

```
 0  ************************************************
 1  *                                              *
 2  *       This is MOOS Client                    *
 3  *       c. P Newman 2001                       *
 4  *                                              *
 5  ************************************************
 6
 7  ---------------MOOS CONNECT-----------------------
 8    contacting a MOOS server localhost:9000 -  try 00001
 9    Contact Made
10    Handshaking as "pHelmIvP"
11    Handshaking Complete
12    Invoking User OnConnect() callback...ok
13  ------------------------------------------------
14
15  The IvP Helm (pHelmIvP) is starting....
16  Loading behavior dynamic libraries....
17      Loading directory: /Users/mikerb/Research/moos-ivp-extend/src/lib_behaviors-test
18          About to load behavior library: BHV_SimpleWaypoint ... SUCCESS
19  Loading behavior dynamic libraries - FINISHED.
20  Number of behavior files: 1
21  Processing Behavior File: alder.bhv  START
22      Successfully found file: alder.bhv
23      InitializeBehavior: found dynamic behavior BHV_SimpleWaypoint
24      InitializeBehavior: found dynamic behavior BHV_SimpleWaypoint
25  Processing Behavior File: alder.bhv  END
26  mode description:
27  pHelmIvP is Running:
28           AppTick   @ 4.0 Hz
29           CommsTick @ 4 Hz
```

The output prior to line 15 is standard MOOS output for an application connecting to the MOOSDB server. The lines thereafter are specific to the pHelmIvP application. In lines 16-19, the helm indicates that the directories specified in the IVP_BEHAVIOR_DIRS environment variable were found and indicates all dynamic behaviors loaded from those directories, regardless of whether they are used in this mission. Line 20 indicates the number of behavior files (.bhv files) comprising this mission. For each behavior file, output similar to lines 21-26 are generated which reports on the attempts to load individual behavior, noting for each whether they are a static behavior of a dynamically loaded behavior.

When the example is fully launched, the pMarineViewer should appear with a simulated vehicle, and two buttons at the lower right corner. The vehicle can be launched by clicking the ”DEPLOY” button. The dynamically loaded behavior is called BHV_SimpleWaypoint and is described in detail in Section 13.

8.5 Extending the Extensions

To add further MOOS application modules, the simplest way by this example is to create sibling directories to the `pXRelayTest`, and add the corresponding entry to the `CMakeLists.txt` file in the `src/` directory. Further IvP behaviors can be added within the `lib_behaviors-test` directory, or in a separate `lib_*` directory. In the former case, the `CMakeLists.txt` file in the behavior directory needs to be augmented for the new behavior. In the latter case, an extra entry in the `CMakeLists.txt` file in the `src/` directory is required, as well as the addition of another directory in the `IVP_BEHAVIOR_DIRS` variable as described above in Section 8.4.

9 Introduction to the IvPBuild Toolbox

The IvPBuild Toolbox is a set of C++ classes and algorithms for building IvP functions. The primary objective is to provide tools to the implementors of new helm behaviors that are fast and easy to use. In the behavior implementation example in Listing 4, the creation of an IvP function required only about a dozen lines of code using two different methods available in the IvPBuild Toolbox.

9.1 Brief Overview

An instance of the class `IvPFunction` is the primary output of a behavior on each helm iteration, and is comprised of anywhere from a handful to thousands of "pieces" that approximate the utility function of the behavior. An example IvP function approximating a utility function is shown below in Figure 22.

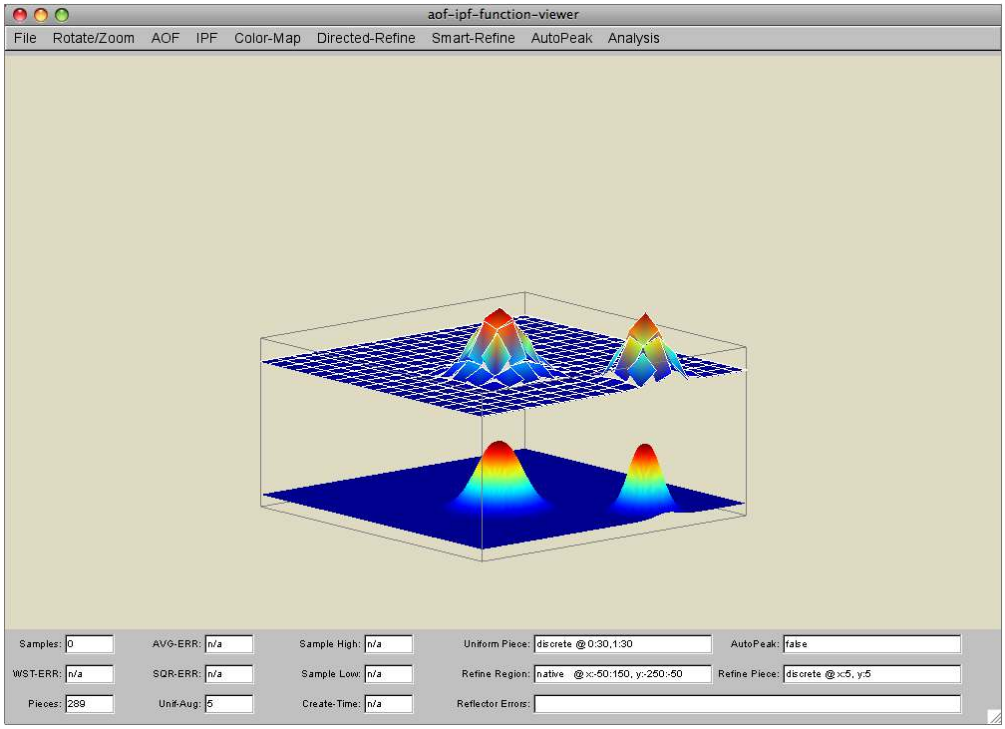

Figure 22: **An IvP function approximating an underlying function**: The `fview` tool is used to render an IvP function with 289 pieces to approximate a given function, shown below the IvP function.

The toolbox contains tools for making simple one-variable objective functions (the "ZAIC" tools) as well as functions over N variables (the "Reflector" tools). The primary contribution of the user (behavior implementor) is to provide the underlying utility function provided to the toolbox. The IvP function approximation is generated automatically given user parameter preferences.

9.1.1 Where to Get the IvPBuild Toolbox

The IvPBuild Toolbox is part of the standard moos-ivp bundle distributed from www.moos-ivp.org. See Section **??**. In the software tree it is entirely contained in the module `lib_ivpbuild`.

9.1.2 What is an Objective Function?

An objective function is a function like any other, a mapping from a domain to a range. In the case where the domain variables correspond to decisions or choices, and the range corresponds to the utility with respect to a particular user objective, the function is often called an "objective" function, or "utility" function.

9.1.3 What is Multi-objective Optimization?

The term multi-objective optimization refers to a situation where there are multiple objective functions defined over the same domain, i.e., decision space, and the ideal goal is to find a point in the decision space that optimizes all functions simultaneously. Rarely is such a mutually agreeable decision available and typically the functions can be said to be "competing". Techniques vary widely on how to handle this. A simple technique would be to rank order the functions and optimize the most important first, and so on. Another technique involves setting a competence threshold for each and choosing from decisions that satisfy a minimum competence for each function.

Many techniques for optimization are predicated on there being a user involved in the decision process who can interactively alter parameters of the problem until an agreeable resolution emerges. In these cases the notion of Pareto optimality, [**?**], often plays a central role. A Pareto optimal solution is one that cannot be improved in regard to one objective unless it comes at the expense of another objective. Typical user-interactive multi-objective optimization techniques involve letting the user explore the Pareto frontier, i.e., those solutions that are all Pareto optimal differing only on the user's value function or relative preference in importance of objectives.

In repeatedly applying multi-objective optimization to the output of behaviors in an on-board *autonomous* decision making system, there is no user involved by definition. There is no exploration of the Pareto frontier since that exploration requires a user. Instead, part of the autonomy process involves also setting the value function, i.e., the relative importance of objectives. In the IvP helm, this value function is reflected by *priority weights* assigned to each function, and the multi-objective optimization problem is reduced to a single objective optimization problem, given k functions and w_i being the weight of the ith function:

$$\vec{x}^* = \underset{\vec{x}}{\text{argmax}} \sum_{i=0}^{k-1} w_i f_i(\vec{x})$$

The properties of IvP multi-objective optimization and solution algorithms were discussed in Section 6.6.3.

9.1.4 What is an IvP Function?

An IvP function is a piecewise linearly defined function where each piece has an upper and lower boundary (or *interval*) on the decision space and linear function defined over the piece. An IvP function is defined over a domain that itself has an upper and lower boundary for each decision variable. Furthermore, the domain is comprised of equally spaced discrete points, and therefore each piece is defined over a finite set of points in the domain. An IvP function is typically an *approximation* of the user's underlying utility function. The fidelity of this approximation can be controlled by the user of the toolbox by deciding how many pieces are used in the approximation. Since the size or extents of each piece may vary within a function, the toolbox methods may also create functions that user smaller pieces where the underlying function is less amenable to local linear approximations.

An IvP function as an instance of the `IvPFunction` class defined as part of the `lib_ivpcore` module included in the basic software bundle distributed from `www.moos-ivp.org`.

9.1.5 Why the IvP Function Construct? A Brief Description of the Solver

The IvP function construct was chosen because it balances three aspects needed for use in the extendable behavior-based autonomy philosophy.

- Flexibility: a piecewise defined function approximation can be formed from any underlying function and thus the behavior author is not compelled to produce objective functions of a restricted form, such as convex or continuous functions. The behavior author is free to innovate. The behavior author typically has insight into the degree of fidelity needed to faithfully reflect the underlying utility function.
- Speed: the IvP function constructs, once produced, can be exploited by solution algorithms to give very fast solutions with a guarantee of global optimality modulo the error introduced in function approximation.
- Accuracy: a piecewise defined function can be highly accurate for a few reasons, (a) by controlling the piece size and distribution the approximation can be made to be as accurate as needed. (b) by being free to approximate any underlying function form, a piecewise function may better reflect a behavior's utility. (c) by allowing for guaranteed globally optimal solutions in the resulting optimization problem, errors of this type are eliminated.

The IvP Solver uses a branch-and-bound method to search through the combination space of pieces, one from each of k contributing function. Since each point in the decision space is contained in exactly one piece in each function, the optimal decision corresponds to a k-tuple of pieces. Thus finding the optimal k-tuple guarantees that the optimal point in the decision space has been found. A leaf node in the tree is simply the "intersection" of pieces from contributing functions. Likewise the intersection of interior functions at a leaf node is simply the sum of the linear functions of each contributing piece. Both the intersection of rectilinear pieces and the sum of linear functions be rapidly and simply computed. For a detailed description of the IvP solver solution algorithms, see [**?**].

9.1.6 Properties of the IvPDomain Class

The domain of an IvP function is an instance of the class `IvPDomain`. It is the same between all functions produced by all behaviors. The domain has a finite set of labeled variables with a lower and upper bound for each variable, and an integer number of evenly spaced points between the bounds. The domain is also referred to as the *decision space*. The 2D base of the cube in Figure 22 represents the domain. A point in the domain is contained in exactly one piece of an IvP function. The domain is built by the helm at the time of launch, and a copy is handed to each behavior in its constructor to ensure uniformity between behaviors. Listing 1 shows an example domain with three variables as it would be specified within the MOOS configuration block for the `pHelmIvP` process. See Listing 2.

Listing 9.1: An IvP domain with three variables, as specified in `pHelmIvP` *configuration.*

```
1  Domain  = course:0:359:360
2  Domain  = speed:0:3:16
3  Domain  = depth:0:500:101
```

Each line augments the initially null domain with a new variable. The first of four arguments is the variable name, e.g., `course`. The second and third arguments indicate the lower and upper bound of the variable. They are integers here, but could be floating point values. The last of four arguments is the number of points in the domain for that variable. This domain would have $(360 * 16 * 101)$ 581,760 distinct possible decisions.

The behavior author, using the IvPBuild Toolbox, only needs to create a black-box function routine able to evaluate any point in the IvP domain with respect to the objectives of that behavior. To use the toolbox, this routine needs to reside within an implementation of a class that subclasses the `AOF` class described in Section 11.2. Although behaviors share a common domain, they can be defined over a different subset of variables as long as the common variables match in extents. For example, a behavior in a helm configured with the domain above could be defined over the following sub-domain:

```
Domain  = depth:0:500:101
```

It could not be defined over:

```
Domain  = depth:0:100:101
```

To facilitate the proper creation of sub-domains, the following function is provided in the build toolbox (in `BuildUtils.h` in `lib_ivpbuild`):

```
IvPDomain  subDomain(IvPDomain original_domain, string variable_names);
```

The first argument is the original domain. The string argument is a comma separated list of variable names to be included in the sub-domain. If a variable is named in the argument that doesn't exist in the original domain, an empty domain is returned. This is detected by checking the size of the domain with a call to `domain.size()`, which returns the number of domain variables. A proper sub-domain of the domain shown in Listing 1, with only the `depth` variable, could be created with the following function call:

```
#include "BuildUtils.h"
 ...
domain = subDomain(original_domain, "depth");
```

It is common for a behavior to declare its sub-domain in its constructor, even if it is expected to be the same as the total helm domain. See for example line 21 in Listing.

A call to `subDomain()` has no effect if it names all of the original variables. By declaring a sub-domain in the behavior's constructor, problems can be avoided later if the overall helm domain is expanded. If the function call above erroneously creates a null domain for the behavior, the helm detects this automatically in the `isRunnable()` function call described in Section 7.4. This will cause an error message to be posted to the `BHV_ERROR` variable and result in the helm posting all-stop values to its actuators.

9.2 Tools Available in the IvPBuild Toolbox

The IvPBuild Toolbox contains a few different sets of tools. (a) The ZAIC tool is used for creating IvP functions with only one decision variable. (b) The basic Reflector tool is used for creating IvP functions over N coupled variables. (c) The advanced Reflector tools are an extension of the basic Reflector tools that allow for piecewise defined functions with non-uniform pieces. (d) The Coupler tool allows a pair of decoupled IvP functions to be converted to a single coupled IvP function. (e) The encoding/decoding tools allow IvP functions to be converted to a string representation and vice versa.

9.2.1 The ZAIC Tools for Functions with One Variable

The ZAIC tools are used for functions defined over a single decision variable. An example is shown in Figure 23 where a fictional behavior may want to keep the vehicle in the so-called "deep sound channel" or "SOFOR (SOund Fixing and Ranging) channel". This is a horizontal layer in the ocean around which the speed of sound is at a minimum and where sound, especially at low frequencies, may travel for thousands of meters with little loss of signal (See [**?**]).

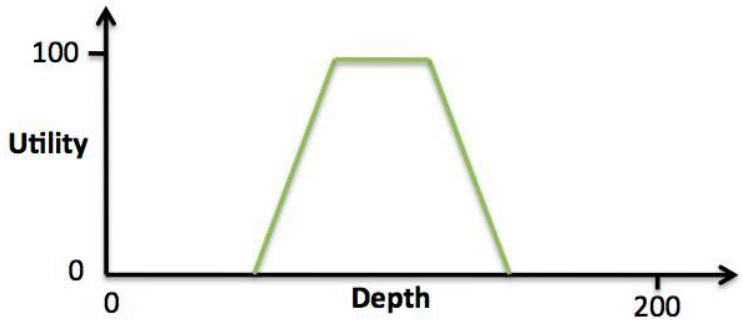

Figure 23: **An objective function with a single decision variable**: This function assigns a maximum utility to depths in a range of roughly 100 ± 20 meters. A linear decrease in utility is associated with depths outside this interval up to another additional 20 meters.

A piecewise defined IvP function can be constructed to represent this function using five pieces. Assuming the "depth" decision space is 0 to 200 meters at one meter increments, the intervals would be: [0,59], [60,79], [80,120], [121,140], [141,200]. The linear function for each piece also needs to be set. This is not terribly difficult, but it is tedious and prone to human error. Instead, the `ZAIC_PEAK` utility is a tool in the ZAIC toolbox used for automating the production of IvP piecewise functions of the form shown in Figure 23. It is described in Section 10.

9.2.2 The Reflector Tool for Functions with Multiple Variables

The Reflector tool is used for creating IvP functions over n decision variables where n is greater than two. The Reflector was used to generate the IvP function rendered in Figure 22. The tools do work for n=1, but one variable functions are typically handled with the ZAIC tools described above. The Reflector produces an IvP function approximation of a given underlying function, where the underlying function is provided to the Reflector in the form of an instance of a class containing the underlying function implementation, and a specification of the IvP domain. The basic use of the Reflector is described in detail in Section 11, but the basic usage boils down to the following:

- Create an instance of the underlying function to be approximated.
- Create an instance of the Reflector, passing it the underlying function.
- Invoke the Reflector with a requested number of pieces.
- Retrieve the new IvP function from the Reflector.

The basic usage of the Reflector involves only the choosing the number of pieces used in the IvP function representation. By choosing only the number, pieces of uniform size will be used in the function. This suffices for most applications, but there are ways to produce a function that is both more accurate and uses less pieces by exploring advanced options and algorithms of the Reflector. These include:

- The *Directed Refinement* Reflector option.
- The *Smart Refinement* Reflector option.
- The *AutoPeak Refinement* Reflector option.

The details of these advanced options are discussed in Section 12. Each of the advanced tools are used after an initial basic uniform function has been generated. The *directed refinement* option allows the user to specify subsets of the domain and use pieces of different sizes for that region only. The *smart refinement* option asks the Reflector to estimate the fit of each piece as it is generated in terms of accuracy in approximating the underlying function and performs further refinement on those pieces that need it the most. The *autopeak refinement* option repeatedly refines the single piece containing the maxima of the underlying function until that point is contained in a piece containing only that point.

9.2.3 The Coupler Tool for Coupling Two Decoupled IvP Functions

Two IvP functions defined over different variables can be combined to form a single IvP function defined over the union of the two sets of variables. The basic usage of the Coupler can be summarized as follows:

- Create the two independent IvP functions.
- Create an instance of the `OF_Coupler` class.
- Pass the two functions to the Coupler.
- Retrieve the new IvP function from the Coupler.

This tool was used in the example SimpleWaypoint behavior of Section 13 in Listing 5 to couple two one-variable IvP functions. When a Coupler is passed the two IvP function pointers, it takes over ownership of the functions, i.e., it deletes the two one-variable functions when the Coupler object is deleted. When a coupled IvP function is extracted from the Coupler, ownship of the IvP function is passed to the caller, i.e., the caller is responsible for deleting the IvP function.

10 The ZAIC Tools for Building One-Variable IvP Functions

The ZAIC tools are part of the IvP Build Toolbox for facilitating the building of IvP functions over a single domain variable. There are three tools - ZAIC_PEAK, ZAIC_HEQ, ZAIC_LEQ, and ZAIC_Vector. To use the tools, in short, one creates a instance of the corresponding class, sets some parameters, and then extracts an IvP function. The tools described in Section 11 for n-variable functions can also be used for building one-variable functions but are perhaps overkill for certain classes of common one-variable functions that motivated the ZAIC tools. The term ZAIC is not an acronym, but merely a play on the word mosaic.

10.1 The ZAIC_PEAK Tool

10.1.1 Brief Overview

The ZAIC_PEAK tool is designed with the objective function shown in Figure 24 in mind. There is a identifiable preferred single decision choice (the summit) with maximum utility, and then a gradual drop in utility as the variable value varies from the preferred choice.

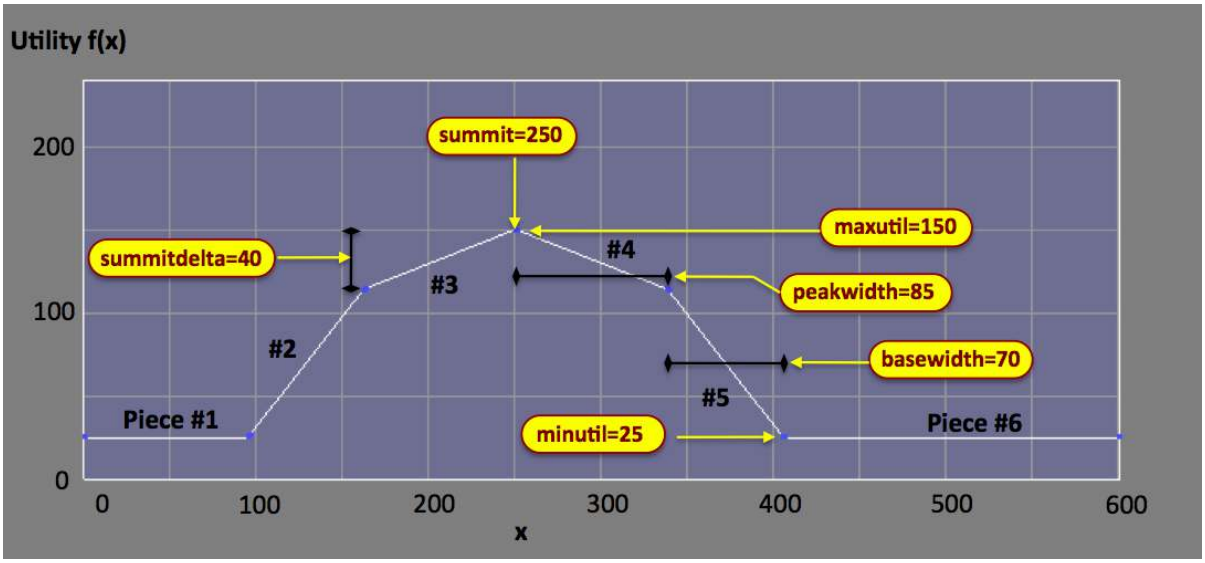

Figure 24: **The ZAIC_PEAK tool**: defines an IvP function over one variable defined by the six parameters shown here. In the case rendered here, the tool would create an IvP function with six pieces. The function rendered was created with summit=180, peakwidth=85, basewidth=70, maxutil=150, minutil=25, summitdelta=40.

The form in which the utility drops is dependent on the settings of the six parameters shown in the figure. The summit, peakwidth, and basewidth values are given in units native to the decision variable, while the summitdelta, minutil, and maxutil values are given in terms of units of utility.

10.1.2 The ZAIC_PEAK Parameters and Function Form

The ZAIC_PEAK tool accepts six parameters in defining $f(x)$. The summit parameter is the point of maximum utility. The minutil parameter is the minimum value of $f(x)$, with a default value of zero.

The `maxutil` is the maximum value of $f(x)$, with a default value of 100. The utility of the function drops off linearly in two stages. In the first stage the utility drops linearly off from the `maxutil` to `maxutil-summitdelta`, and in the second stage it drops off linearly from `maxutil-summitdelta` to `minutil`. The function has the form:

$$f(x) = \begin{cases} f_1(x) & (\texttt{summit} - \texttt{peakwidth}) \leq x \leq \texttt{summit}, \\ f_2(x) & (\texttt{summit} - \texttt{peakwidth} - \texttt{basewidth}) \leq x < (\texttt{summit} - \texttt{peakwidth}), \\ f_3(x) & \texttt{summit} < x \leq (\texttt{summit} + \texttt{peakwidth}), \\ f_4(x) & (\texttt{summit} + \texttt{peakwidth}) < x \leq (\texttt{summit} + \texttt{peakwidth} + \texttt{basewidth}), \\ \texttt{minutil} & \text{otherwise.} \end{cases} \quad (1)$$

where

$$f_1(x) = (\texttt{maxutil} - \texttt{summitdelta}) + (\texttt{summitdelta} * ((x - (\texttt{summit} - \texttt{peakwidth}))/\texttt{peakwidth}))$$
$$f_2(x) = \texttt{minutil} + ((\texttt{maxutil} - \texttt{minutil} - \texttt{summitdelta}) * ((x - (\texttt{summit} - \texttt{peakwidth} - \texttt{basewidth}))/\texttt{basewidth}))$$
$$f_3(x) = (\texttt{maxutil} - \texttt{summitdelta}) + (\texttt{summitdelta} * (((\texttt{summit} + \texttt{peakwidth}) - x)/\texttt{peakwidth}))$$
$$f_4(x) = \texttt{minutil} + ((\texttt{maxutil} - \texttt{minutil} - \texttt{summitdelta}) * (((\texttt{summit} + \texttt{peakwidth} + \texttt{basewidth}) - x)/\texttt{basewidth}))$$

To correlate the above five cases above with the six pieces in Figure 24, $f_1(x)$ is piece `#3`, $f_2(x)$ is piece `#2`, $f_3(x)$ is piece `#4`, $f_4(x)$ is piece `#5`, `minutil` for pieces `#1` and `#6`. The two stage linear drop-off in utility is there to allow the shape of the function to approximate convex or concave functions as shown in Figure 25.

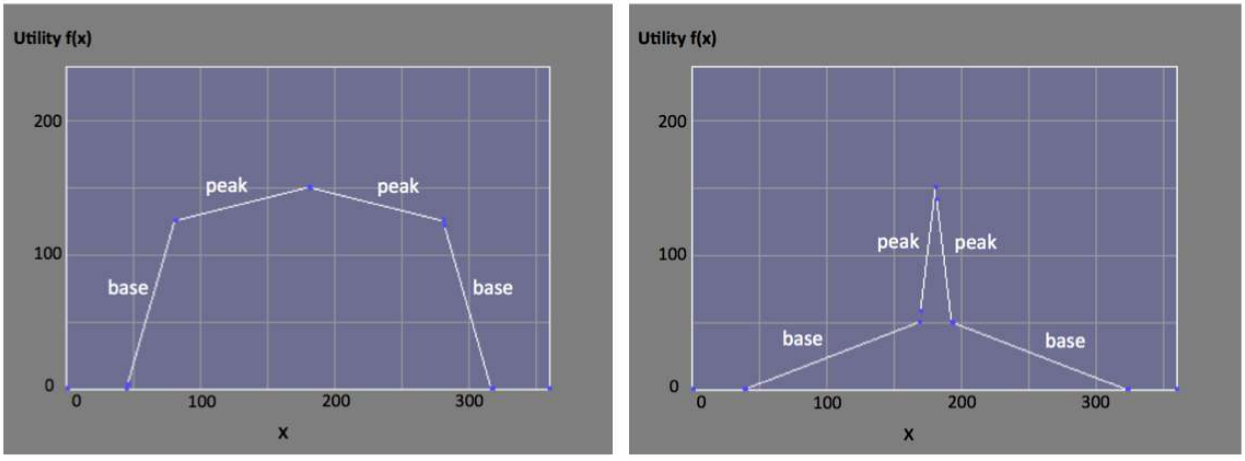

Figure 25: **The `ZAIC_PEAK` tool:** A convex uni-modal function (left) and a non-convex uni-modal function (right).

10.1.3 The `ZAIC_PEAK` Interface Implementation

The following functions define the interface to the `ZAIC_PEAK` tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were

detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested.

Many of the below functions take an optional `index` parameter. This is used for creating functions with multiple modes or peaks as described in Section 10.1.6. The default value is zero, or the zeroth index, when only one mode or peak is being implemented. If the given index references a non-existing component, this is considered a fatal configuration error. Example usage is provided in Listing 1.

- `bool setSummit(double val, int index=0)`: Sets the `summit` value of the component at the given index. If no index parameter is provided, the index is zero. If the summit value is outside the range of the domain, it is clipped to the appropriated end. For example if the domain were [0,359] as the example in Figure 24, and the requested summit value were 550, the summit parameter would be set to 359. This is therefore not regarded as a fatal configuration error, but a warning would be generated anyway. This returns false only if index referencing a non-existent component is provided.
- `bool setPeakWidth(double val, int index=0)`: Sets the peakwidth value of the component at the given index. If no index parameter is provided, the index is zero.
- `bool setBaseWidth(double val, int index=0)`: Sets the basewidth value of the component at the given index. If no index parameter is provided, the index is zero.
- `bool setSummitDelta(double val, int index=0)`: Sets the `summitdelta` value of the component at the given index. If no index parameter is provided, the index is zero. A fatal error is declared and false is returned if the index is out of range, or if the value is less than zero. Otherwise true is returned. If the sumitdelta value is greater than the range determined by `maxutil - minutil`, this is not interpreted as a fatal error, but the `summitdelta` is clipped to the range.
- `bool setMinMaxUtil(double min, double, max, int index=0)`: Sets the minutil and maxutil values of the component at the given index. If no index parameter is provided, the index is zero. A fatal error is declared and false is returned if the index is out of range, or if the min value is greater than or equal to the max value. Otherwise true is returned. If the existing `summitdelta` value is greater than the range determined by `maxutil - minutil`, this is not interpreted as a fatal error, but the `summitdelta` is clipped to the range.
- `bool setParams(double summit, double peakwidth, double basewidth, double summitdelta, double minutil, double maxutil, int index=0)`: Sets the six configuration parameters `summit`, `peakwidth`, `basewidth`, `summitdelta`, `minutil` and `maxutil` all at once. If the `summitdelta` value is greater than the range determined by `maxutil - minutil`, this is not interpreted as a fatal error, but the `summitdelta` is clipped to the range.
- `void setSummitInsist(bool val)`: Sets the `summitinsist` flag to the given value. The default is true. See Section 10.1.4 for more on this parameter.
- `void setValueWrap(bool val)`: Sets the `valuewrap` flag to the given value. The default is false. See Section 10.1.4 for more on this parameter.
- `int addComponent()`: Allocates a new component and returns the index of the new component. Since the first component exists upon `ZAIC` creation, the first call to this function will return 1, and will result in the `ZAIC_PEAK` instance having two components at index 0 and 1. The default values for the new component are `summit`=0, `peakwidth`=0, `basewidth`=0, `summitdelta`=50,

minutil=0, maxutil=100.

- `IvPFunction* extractIvPFunction(bool maxval=true)`: This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. If a fatal error was detected in prior parameter setting attempts, this function will simply return the NULL pointer. When the IvP function is extracted from the ZAIC, an `IvPFunction` instance is created from the heap that needs to be later deleted. The ZAIC tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.
- `string getWarnings()`: When or if fatal (or non-fatal) problems are encountered in setting the parameters, the tool appends a message to a local warning string. This string can be retrieved by this function. A non-empty string does not necessarily mean a fatal configuration error was encountered. Instead, the `stateOK()` function below should be consulted.
- `bool stateOK()`: This function returns true if no fatal errors were encountered during configuration attempts, otherwise it returns false. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the `getWarnings()` function above can be consulted.

10.1.4 The Value-Wrap and Summit-Insist Parameters

Two additional Boolean parameters may be set for an instance of `ZAIC_PEAK`. They are the `valuewrap` and `summitinsist` parameters set with the following functions with the defaults shown:

```
zaic.setValueWrap(false);     // Default value is false
zaic.setSummitInsist(true);   // Default value is true
```

When the `valuewrap` parameter is true, the utility associated with the domain variable value "wraps" around. For example, if the domain variable is the vehicle *heading*, which may have the domain [0, 359], a value of 10 degrees is evaluated as being only 20 degrees different from 350, rather than being different by 330 degrees. The two functions depicted in Figure 26 differ only in the setting of the `valuewrap` parameter.

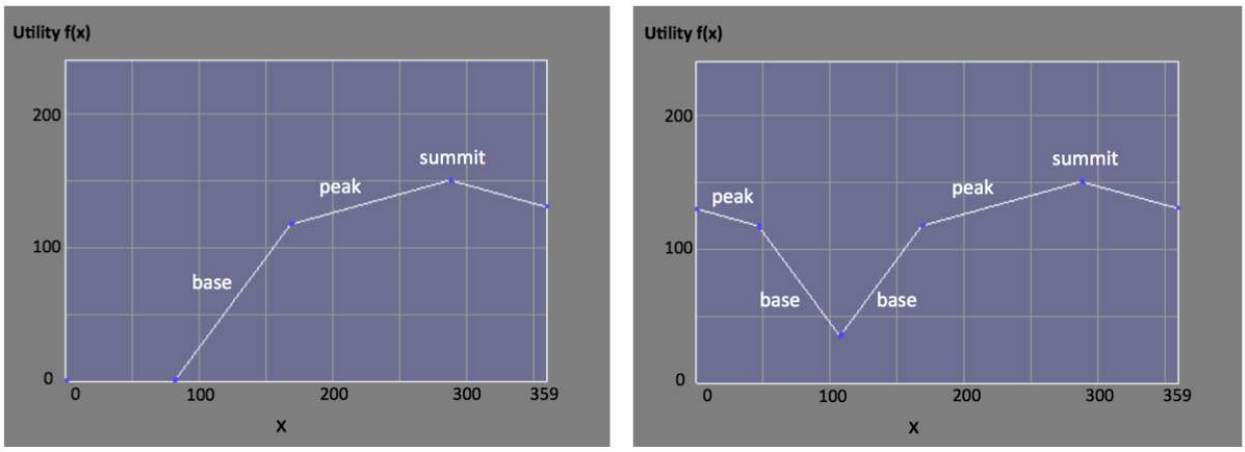

Figure 26: **The `valuewrap` parameter in the `ZAIC_PEAK` tool:** A function generated with `valuewrap=false` on the left, and function generated with `valuewrap=true` and otherwise identical parameters on the right.

The value of the summitinsist parameter can affect the generated objective function in the following two scenarios. In the first case, consider the domain to be the possible headings with 360 discrete choices [0, 359], and the peakwidth and basewidth are both zero. If the summit were then set to 90.25, one way to interpret this is that all 360 discrete heading choices have a utility of zero, since none are equal exactly to 90.25. This is the interpretation when summitinsist is false. When set to true, the same set of parameters would generate an objective function that ranked the heading of 90 degrees with maximum utility and all other heading choices with the minimum utility (an IvP function with 3 pieces, i.e., intervals). In the second case, consider the domain to be possible speeds with 31 discrete choices [0, 3.0], with the summit set to 4.0 with a peakwidth and basewidth of 0.25. The ZAIC_PEAK tool would generate an objective function ranking all speeds equally with the minimum utility when summitinsist is set to false. When set to true, the ZAIC_PEAK tool would generate an objective function ranking the highest speed in the domain (3.0) with maximum utility (maxutil), and all other speeds with minimum utility (minutil). The default setting is summitinsist=true since this seems the more reasonable thing to do in most such cases.

10.1.5 Using the ZAIC_PEAK Tool

Usage of the ZAIC_PEAK tool boils down to the following four steps.

- Step 1: Create the IvP domain, or retrieve it if otherwise already created.
- Step 2: Create the ZAIC_PEAK instance with a domain and domain variable.
- Step 3: Set the ZAIC_PEAK parameters.
- Step 4: Extract the IvP function.

A code example of the four steps is provided in Listing 1 below. This code example describes a function that builds and returns an IvP function using the ZAIC_LEQ tool. It is not too different from the activity inside a typical implementation of onRunState() in an IvP behavior.

Listing 10.1: Example usage of the ZAIC_Peak *tool corresponding to Figure 24.*

```
   #include "ZAIC_PEAK.h"
   ...
 1 IvPFunction *buildIvPFunction()
 2 {
 3   // Step 1 - Create the IvPDomain, the function's domain
 4   IvPDomain domain;
 5   domain.addDomain("depth", 0, 600, 601);
 6
 7   // Step 2 - Create the ZAIC_PEAK with the domain and variable name
 8   ZAIC_PEAK  zaic_peak(domain, "depth");
 9
10   // Step 3 - Configure the ZAIC_PEAK parameters
11   zaic_peak.setSummit(150);
12   zaic_peak.setMinMaxUtil(20, 120);
13   zaic_peak.setBaseWidth(60);
14
15   // Step 4 - Extract the IvP function
16   IvPFunction *ivp_function = 0;
17   if(zaic_peak.stateOK())
18     ivp_function = zaic_peak.extractIvPFunction();
19   else
20     cout << zaic_peak.getWarnings();
21   return(ivp_function)
```

The lines comprising step 4 (lines 15-20) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

```
return(zaic_peak.extractIvPFunction());
```

This is simpler, but the warning information is potentially useful. When the `ZAIC_PEAK` tool is used within an IvP behavior, the warnings can be posted to the MOOSDB in the variable `BHV_WARNING` which is monitored by other tools.

10.1.6 Support for Multi-Modal Functions with the `ZAIC_PEAK` Tool

The `ZAIC_PEAK` tool will allow additional components to be added to provide a multi-modal effect. A *component* refers to the set of six parameters `summit`, `peakwidth`, `basewidth`, `summitdelta`, `minutil`, `maxutil`. Adding a second component creates a multi-modal function as shown in Figure 27.

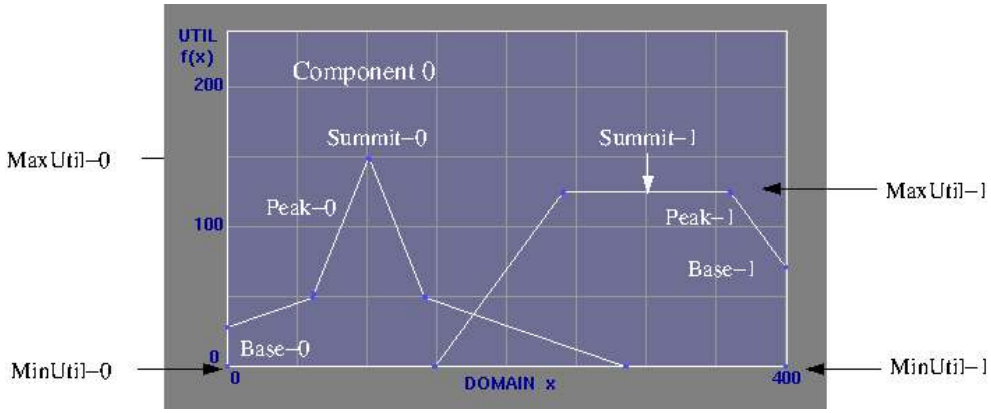

Figure 27: **Multiple modes with the `ZAIC_PEAK` tool**: additional components can be added to create a multi-modal objective function. Each component is comprised of the six parameters `summit`, `peakwidth`, `basewidth`, `summitdelta`, `minutil`, and `maxutil`.

Listing 2 shows how this is done. In lines 2-3, the `ZAIC_PEAK` instance is created and the parameters are set for the first component. A second component is allocated in line 5 with the call to `addComponent()` which returns the index of the newly created component. This index is passed as the last argument to the function call on line 6 to clarify that the parameters are to be applied to the second component (at index 1).

Listing 10.2: Using the `ZAIC_PEAK` tool to build and return a multi-mode IvP Function.

```
   #include "ZAIC_PEAK.h"
   ...
1  IvPFunction *buildIvPFunction(IvPDomain domain, string varname)
2  {
3    ZAIC_PEAK zaic(domain, varname);
```

```
 4    zaic.setParams(300, 80, 100, 15, 0, 100);  // No index given - assumed to be zero.
 5
 6    int index = zaic.addComponent();
 7    zaic.setParams(600, 130, 35, 30, 0, 147, index); // Last param is component index
 8
 9    zaic.setValueWrap(false);     // Not component specific - no index given
10    zaic.setSummitInsist(true);   // Not component specific - no index given
11    bool take_the_max = true
12    IvPFunction *ipf = zaic.extractIvPFunction(take_the_max);
13    return(ipf);
14  }
```

When the IvP function is extracted from the `ZAIC_PEAK`, in line 12, the composition of the multiple components can be interpreted in one of two ways - by taking the *maximum* or the *sum* of the two components. In the former, the combined utility is the maximum of the values given by the individual components. In the latter the combined utility is the sum of the individual components. The `extractIvPFunction(bool)` function on line 12 will return a composition based on taking the max if passed true, and will take the sum otherwise. The default (if no value is passed) is true. The difference between the two extractions is shown in Figure 28 for the two components shown in Figure 27.

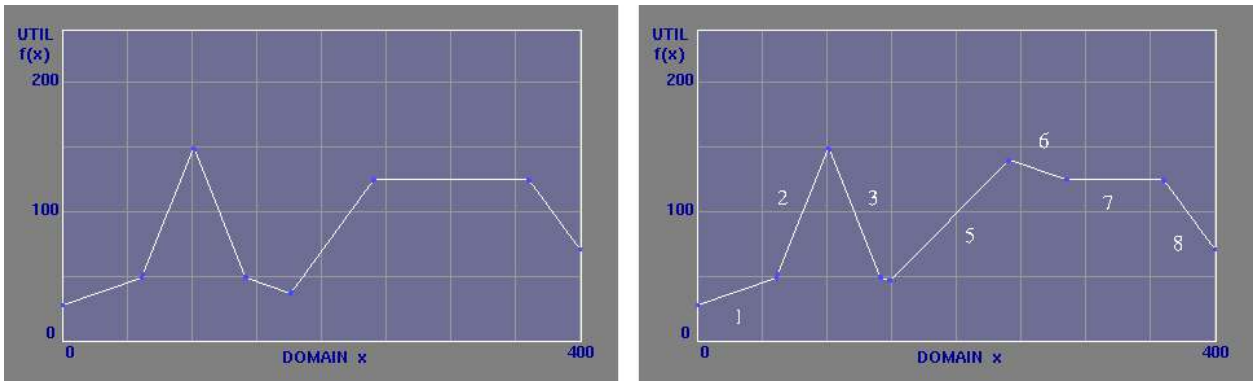

Figure 28: **Options for combining components**: On the left is the result of combining the two components in Figure 27 by using the max value from the two components. On the right is the result of combining the two components by using the sum of the values from the two components.

All component parameter-setting functions defined for the `ZAIC_PEAK` class take an optional final argument indicating the intended component index. If the argument is not provided, it is assumed to be zero, the index of the one component created automatically when the `ZAIC_PEAK` instance is created. The default values of a newly added component are are `summit`=0, `peakwidth`=0, `basewidth`=0, `summitdelta`=50, `minutil`=0, `maxutil`=100. The `valuewrap` and `summitinsist` parameters are not component-specific parameters, and therefore are only called once on a particular `ZAIC_PEAK` instance, and do not have an optional index parameter.

10.2 The ZAIC_LEQ and ZAIC_HEQ Tools

10.2.1 Brief Overview

The ZAIC_LEQ tool is used for generating IvP functions where there is a constant, maximum utility associated with a decision variable whose value is kept *less than or equal* to (LEQ) a given value. For example if a UUV component is known to work reliably up to a certain depth, or if a vehicle's speed is to be kept below a certain value to prevent interference with another sensor or communications equipment. The ZAIC_LEQ tool allows for expressing a linear drop-off in utility between two values. For example, if the fictional UUV component is rated to a depth of 200 meters, the utility function may have a maximum utility for depths up to 150 meters and minimum utility at 210 meters as in the example in Figure 29.

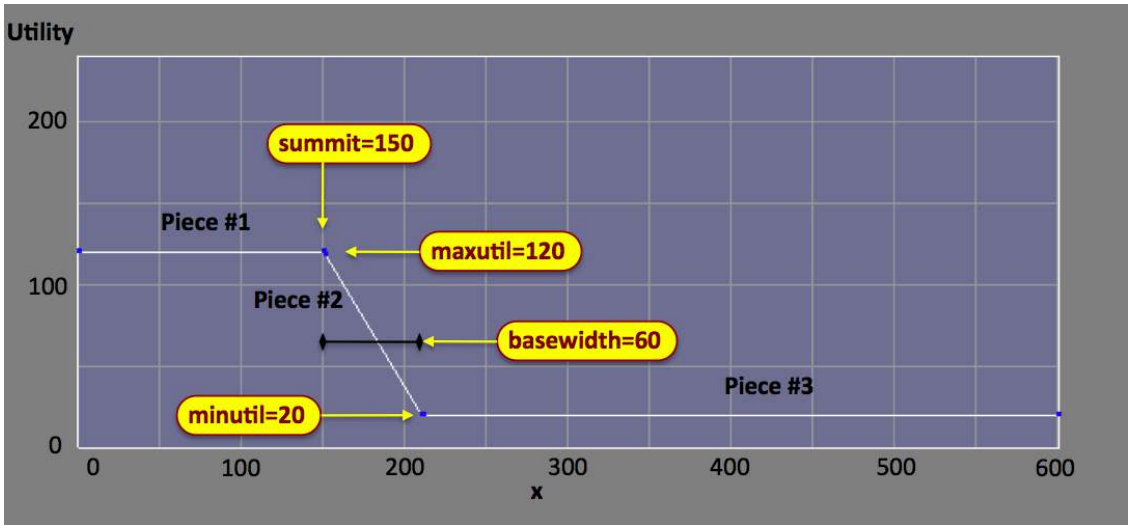

Figure 29: **The ZAIC_LEQ tool**: facilitates building simple 2-3 piece piecewise defined utility functions over a single decision variable whose value is to be kept less than or equal to a given value. It accepts four parameters, the `summit`, `minutil`, `maxutil`, `basewidth`. In the figure `summit`=150, `minutil`=20, `maxutil`=120, `basewidth`=60.

Likewise, the ZAIC_HEQ tool is used for generating objective functions where there is a constant, maximum utility associated with a decision variable whose value is kept *greater than or equal* to (HEQ) a given value. These two tools have a similar interface. Most of what is described below for one tool applies to the other. The differences are distinguished in Section 10.2.5. The IvP functions generated by these ZAIC tools have a small footprint, having either two or three pieces.

10.2.2 The ZAIC_LEQ Parameters and Function Form

The ZAIC_LEQ tool accepts four parameters in defining $f(x)$. The `summit` parameter is the point where maximum utility begins to drop off. The `minutil` parameter is the minimum value of $f(x)$, with a default value of zero. The `maxutil` is the maximum value of $f(x)$, with a default value of 100. The function has the form:

$$f(x) = \begin{cases} \texttt{maxutil} & x \leq \texttt{summit}, \\ \texttt{minutil} & \text{otherwise}. \end{cases} \tag{2}$$

The `basewidth` parameter can be used to soften the drop in utility as shown in Figure 29. When `basewidth` has the default value of zero, the general form is as above. When `basewidth` is configured with a positive value, the general form is:

$$f(x) = \begin{cases} \texttt{maxutil} \\ \texttt{minutil} + ((\texttt{maxutil} - \texttt{minutil}) * ((\texttt{x} - \texttt{summit}) / \texttt{basewidth})) & \begin{array}{l} x \leq \texttt{summit}, \\ \texttt{summit} < x \leq \texttt{summit} + \texttt{basewidth}, \\ \texttt{otherwise}. \end{array} \\ \texttt{minutil} \end{cases} \tag{3}$$

For the example in Figure 29, the function is given below.

$$f(x) = \begin{cases} 120 & x \leq 150, \\ 20 + ((120 - 20) * ((210 - x)/(210 - 150))) & 150 < x \leq 210, \\ 20 & \text{otherwise}. \end{cases} \tag{4}$$

The three above cases corresponds to the three pieces generated for the IvP function shown in Figure 29.

10.2.3 The `ZAIC_LEQ` Interface Implementation

The following functions define the interface to the `ZAIC_LEQ` tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested. Example usage is provided in Listing 3.

- `ZAIC_LEQ(IvPDomain domain, string varname)`: The constructor takes two arguments, an IvP domain and a variable name contained in the domain. The named variable needs to be just *one* of the variables used in the IvP domain; not necessarily the only one. If the named variable is not part of the IvP domain, this is regarded as a fatal error.
- `bool setSummit(double summit)`: Sets the `summit` value. If the summit value is outside the range of the domain, it is clipped to the appropriated end. For example if the domain were [0,600] as the example in Figure 29, and the requested summit value were 650, the summit parameter would be set to 600. This is therefore not regarded as a fatal configuration error, but a warning would be generated anyway. This function always returns true.
- `bool setMinMaxUtil(double min, double max)`: Sets the values for `minutil` and `maxutil`. If the `minutil` is greater or equal to `maxutil`, this is regarded as a fatal configuration error, a warning is generated, and the function returns false.
- `bool setBaseWidth(double basewidth)`: Sets the `basewidth` parameter value. The given value must be greater than or equal to zero. Otherwise this is regarded as a fatal configuration error, a warning is generated, and the function returns false.

- `IvPFunction *extractIvPFunction()`: This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. If a fatal error was detected in prior parameter setting attempts, this function will simply return the NULL pointer. When the IvP function is extracted from the ZAIC, an IvPFunction instance is created from the heap that needs to be later deleted. The ZAIC tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.
- `string getWarnings()`: When or if fatal (or non-fatal) problems are encountered in setting the parameters, the tool appends a message to a local warning string. This string can be retrieved by this function. A non-empty string does not necessarily mean a fatal configuration error was encountered. Instead, the `stateOK()` function below should be consulted.
- `bool stateOK()`: This function returns true if no fatal errors were encountered during configuration attempts, otherwise it returns false. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the `getWarnings()` function above can be consulted.

10.2.4 Using the `ZAIC_LEQ` Tool

Usage of the `ZAIC_LEQ` tool boils down to the following four steps.

- Step 1: Create the IvP domain, or retrieve it if otherwise already created.
- Step 2: Create the `ZAIC_LEQ` instance with a domain and domain variable.
- Step 3: Set the `ZAIC_LEQ` parameters.
- Step 4: Extract the IvP function.

A code example of the four steps is provided in Listing 3 below. This code example describes a function that builds and returns an IvP function using the `ZAIC_LEQ` tool. It is not too different from the activity inside a typical implementation of `onRunState` in an IvP behavior.

Listing 10.3: Example usage of the `ZAIC_LEQ tool` *corresponding to Figure 29.*

```
   #include "ZAIC_LEQ.h"
   ...
 1 IvPFunction *buildIvPFunction()
 2 {
 3   // Step 1 - Create the IvPDomain, the function's domain
 4   IvPDomain domain;
 5   domain.addVar("depth", 0, 600, 601);
 6
 7   // Step 2 - Create the ZAIC_LEQ with the domain and variable name
 8   ZAIC_LEQ  zaic_leq(domain, "depth");
 9
10   // Step 3 - Configure the ZAIC_LEQ parameters
11   zaic_leq.setSummit(150);
12   zaic_leq.setMinMaxUtil(20, 120);
13   zaic_leq.setBaseWidth(60);
14
15   // Step 4 - Extract the IvP function
16   IvPFunction *ivp_function = 0;
17   if(zaic_leq.stateOK())
```

```
18      ivp_function = zaic_leq.extractIvPFunction();
19    else
20      cout << zaic_leq.getWarnings();
21    return(ivp_function)
```

The lines comprising step 4 (lines 15-20) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

```
return(zaic_leq.extractIvPFunction());
```

This is simpler, but the warning information is potentially useful. When the `ZAIC_LEQ` tool is used within an IvP behavior, the warnings can be posted to the `MOOSDB` in the variable `BHV_WARNING` which is monitored by other tools.

10.2.5 The `ZAIC_HEQ` Tool

Like the `ZAIC_LEQ` tool, the `ZAIC_HEQ` too is used for generating objective functions where there is a constant, maximum utility associated with a decision variable whose value is kept *greater than or equal* to a given value. The parameters described in Section 10.2.2 and interface implementation described in Section 10.2.3 for the `ZAIC_LEQ` tool are identical for the `ZAIC_HEQ` tool. The parameters are interpreted differently however. The same parameters used in Figure 29 are used in the `ZAIC_HEQ` tool to give the function shown in Figure 30.

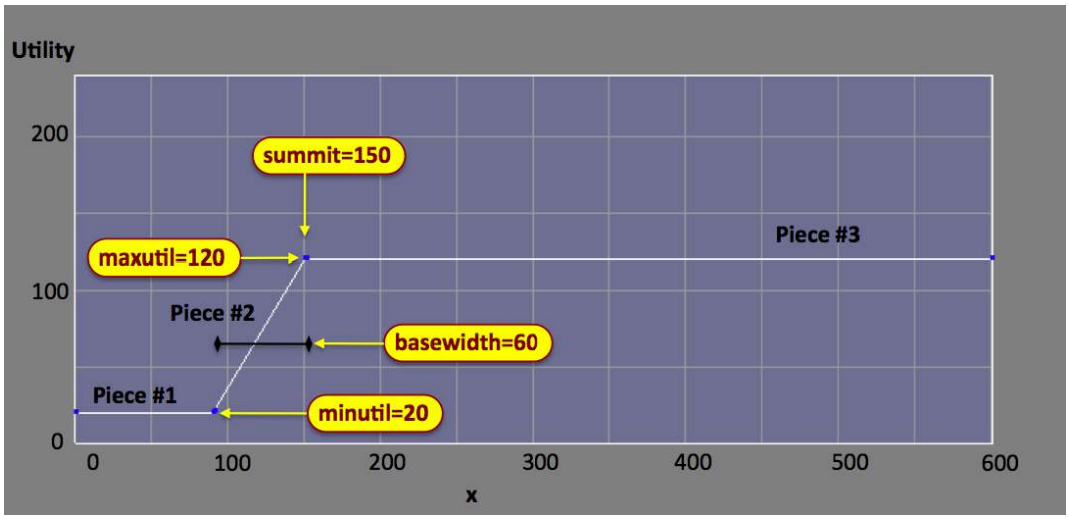

Figure 30: **The `ZAIC_HEQ` tool**: facilitates building simple 2-3 piece piecewise defined utility functions over a single decision variable whose value is to be kept greater than or equal to a given value. It accepts four parameters, the `summit`, `minutil`, `maxutil`, `basewidth`. In the figure `summit`=150, `minutil`=20, `maxutil`=120, `basewidth`=60.

10.2.6 A Warning about the Maximum Utility Plateau

It is worth noting a potential pitfall regarding the maximum utility plateau generated by both the `ZAIC_LEQ` and `ZAIC_HEQ` tools. By having equal utility for all domain values in the plateau range, no one domain value is preferred. If this is the only IvP objective function involved in the decision process *for the particular domain variable*, it is not clear what value will be chosen by the IvP solver. Even more troublesome is that the chosen value may change between iterations giving the appearance that the decision engine is thrashing. In this case the solver is simply faithfully and strictly interpreting the problem it was given. In short, these objective functions are designed to work in conjunction with others that express preferences in a non-plateau manner.

10.3 The `ZAIC_Vector` Tool

10.3.1 Brief Overview

The `ZAIC_Vector` tool is used for generating IvP functions over one variable, given some number of explicit domain-range mappings. These mappings are passed to the `ZAIC_Vector` tool as two equally sized vectors; a vector of domain values and a vector of range values. An IvP function is created that typically has a piece per given domain-range pair, where the slope of each piece simply approximates the domain-range characteristics for domain-range pairs not explicitly given.

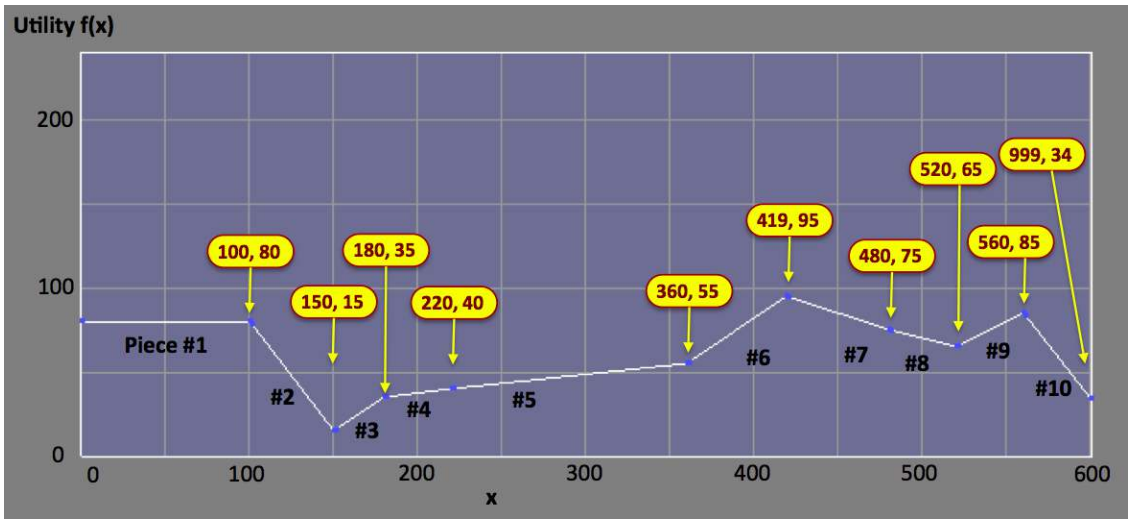

Figure 31: **The `ZAIC_Vector` tool**: defines an IvP function over one variable defined by a set of explicit domain-range mappings.

10.3.2 Using the `ZAIC_Vector` Tool

Usage of the `ZAIC_Vector` tool boils down to the following four steps.

- Step 1: Create the IvP domain, or retrieve it if otherwise already created.
- Step 2: Create the `ZAIC_Vector` instance with a domain and domain variable.

- Step 3: Set the `ZAIC_Vector` parameters.
- Step 4: Extract the IvP function.

A code example of the four steps is provided in Listing 4 below. This code example describes a function that builds and returns an IvP function using the `ZAIC_Vector` tool. It is not too different from the activity inside a typical implementation of `onRunState` in an IvP behavior.

Listing 10.4: Example usage of the `ZAIC_Vector` *tool corresponding to Figure 31.*

```
   #include "ZAIC_Vector.h"
   ...
 1 IvPFunction *buildIvPFunction()
 2 {
 3   // Step 1 - Create the IvPDomain, the function's domain
 4   IvPDomain domain;
 5   domain.addVar("depth", 0, 600, 601);
 6
 7   // Step 2 - Create the ZAIC_Vector with the domain and variable name
 8   ZAIC_Vector  zaic_vector(domain, "depth");
 9
10   // Step 3 - Configure the ZAIC_Vector parameters
11   vector<double> domain_vals;
12   vector<double> range_vals;
13   domain_vals.push_back(100); range_vals.push_back(80);
14   domain_vals.push_back(150); range_vals.push_back(15);
15   domain_vals.push_back(180); range_vals.push_back(35);
16   domain_vals.push_back(220); range_vals.push_back(40);
17   domain_vals.push_back(360); range_vals.push_back(55);
18   domain_vals.push_back(419); range_vals.push_back(95);
19   domain_vals.push_back(480); range_vals.push_back(75);
20   domain_vals.push_back(520); range_vals.push_back(65);
21   domain_vals.push_back(560); range_vals.push_back(85);
22   domain_vals.push_back(999); range_vals.push_back(34);
23   zaic_vecctor.setDomainVals(domain_vals);
24   zaic_vecctor.setRangeVals(range_vals);
25
26   // Step 4 - Extract the IvP function
27   IvPFunction *ivp_function = 0;
28   if(zaic_vector.stateOK())
29     ivp_function = zaic_vector.extractIvPFunction();
30   else
31     cout << zaic_vector.getWarnings();
32   return(ivp_function)
```

The lines comprising step 4 (lines 26-31) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

```
return(zaic_vector.extractIvPFunction());
```

This is simpler, but the warning information is potentially useful. When the `ZAIC_LEQ` tool is used within an IvP behavior, the warnings can be posted to the MOOSDB in the variable `BHV_WARNING` which is monitored by other tools.

11 The Reflector Tool for Building N-Variable IvP Functions

11.1 Overview

The IvPBuild Toolbox contains the `Reflector` tool for building IvP functions over $n \geq 2$ decision variables. Although the tools work with $n = 1$ variables, the `ZAIC` tools are typically used instead. The `Reflector` tool operates on a particular division of labor. The user of the `Reflector` provides a black-box function implementation able to provide a utility value for any queried input. The possible queries are limited to the domain or decision space of the function expressed with an `IvPDomain` instance. This black-box routine in essence is the underlying objective function to be approximated by the generated IvP function (Figure 32).

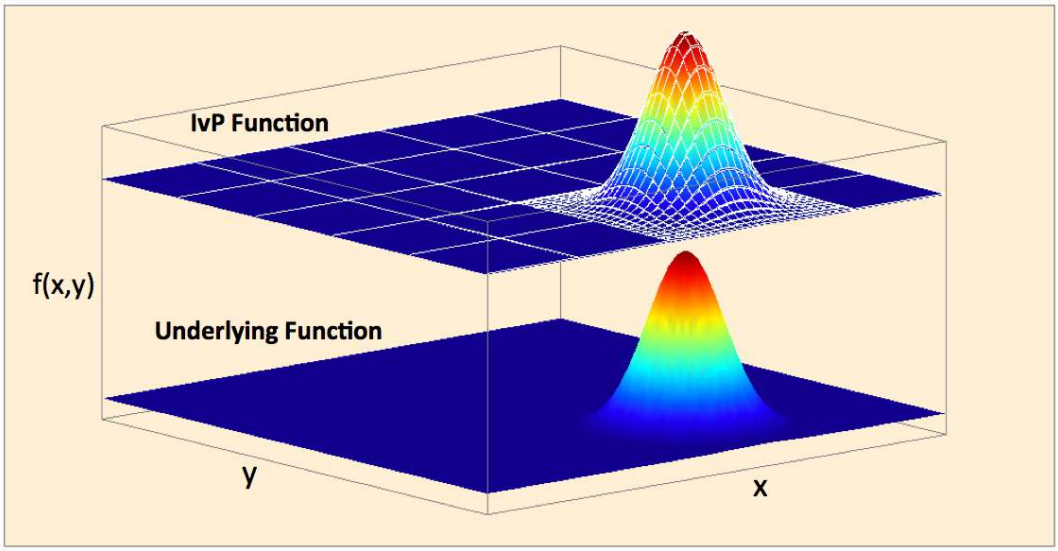

Figure 32: **The Reflector Tool**: An IvP function approximates an underlying function $f(x, y)$ using a piecewise linear structure with 698 pieces. The piece distribution need not be uniform allowing greater resolution over parts of the domain where the function is detected to be less locally linear.

The goal is to generate an acceptable IvP function approximation by querying the underlying function for as a small subset of the total function domain as possible. The CPU time taken to evaluate the underlying function can easily be the most expensive part of building the IvP function for a given behavior. Implementing the underlying function efficiently and in a way that accurately reflects the intent of the behavior can be the most challenging part of building a behavior. The `Reflector` tool can be used with a very simple interface that builds an IvP function given a pointer to the underlying function and the number of pieces to use in the IvP function. Such IvP functions will be constructed with pieces of uniform size. This is discussed in Section 11.3. The Reflector can be configured with more advanced parameters to build an IvP function with non-uniformly distributed pieces as depicted in Figure 32. These methods are used to build functions that more accurately approximate their underlying functions *and* use less pieces. The advanced Reflector parameters are discussed in Section 12.

11.2 Implementing Underlying Functions within the AOF Class

The primary job of a behavior author is to provide a method capable of evaluating any candidate decision in the decision space. Evaluating each decision can be prohibitively time consuming and a piecewise linear approximation with an IvP function is typically built by invoking the evaluation function for only a small subset of the domain. The build toolbox depends on access to the evaluation routine in a generic way, as a pointer to an instance of the class `AOF`, the "actual objective function".

11.2.1 The AOF Class Definition

The `AOF` class itself is abstract, and the AOF pointer actually points to an implemented subclass with a few key virtual functions overloaded. The `AOF` class definition (slightly simplified) is given in Listing 1.

Listing 11.1: The `AOF` *class definition.*

```
1  #include "IvPBox.h"
2  #include "IvPDomain.h"
3  class AOF{
4  public:
5    AOF(IvPDomain domain) {m_domain=domain;};
6    virtual ~AOF() {};
7
8    virtual double evalPoint(vector<double>);
9    virtual bool   setParam(string, double)  {return(false);};
10   virtual bool   setParam(string, string)  {return(false);};
11   virtual bool   initialize()              {return(true);};
12
13   double extract(string, const vector<double>&) const;
14
15 protected:
16   IvPDomain m_domain;
17 };
18 #endif
```

This is essentially a template for a function, defined over the domain given in the constructor. The mapping from the domain to a range is implemented in the `evalPoint()` function which takes a vector of numerical values representing a candidate decision in the IvP domain decision space. The `setParam()` and `initialize()` virtual functions provide a generic way for subclasses to set their parameters.

11.2.2 An Example Underlying Function Implemented as an AOF Subclass

As an example consider the simple linear function $f(x, y) = m \cdot x + n \cdot y + b$, implemented by the class `AOF_Linear` shown in Listing 2 and 3 below. The class contains three member variables, lines 11-13 in Listing 2 for representing the coefficient and scalar parameters.

Listing 11.2: `AOF_Linear.h` *- The class definition for the* `AOF_Linear` *class.*

```
1  #include "AOF.h"
2  class AOF_Linear: public AOF {
3  public:
4    AOF_Linear(IvPDomain domain) : AOF(domain)
5     {m_coeff = 0; n_coeff=0; b_scalar=0;};
```

```
 6    ~AOF_Linear() {};
 7
 8    double evalPoint(vector<double>);
 9    bool   setParam(const string& param, double val);
10
11  private:
12    double m_coeff;
13    double n_coeff;
14    double b_scalar;
15  };
```

In lines 4-5 above, the constructor takes an instance of `IvPDomain` as an argument and passes it to the `AOF` superclass for handling (line 4). The remainder of the class implementation is shown in Listing 3. The m and n coefficients are set in the `setParam()` function and will return true if either the `m_coeff`, `n_coeff`, or `b_scalar` parameters are passed, and false otherwise.

Listing 11.3: `AOF_Linear.cpp` - *The class implementation for the* `AOF_Linear` *class.*

```
 1  //-----------------------------------------------------------
 2  bool AOF_Linear::setParam(string param, double val)
 3  {
 4    if(param == "mcoeff")
 5      m_coeff = val;
 6    else if(param == "ncoeff")
 7      n_coeff = val;
 8    else if(param == "bscalar")
 9      b_scalar = val;
10    else
11      return(false);
12    return(true);
13  };
14
15  //-----------------------------------------------------------
16  double AOF_Linear::evalPoint(vector<double> point)
17  {
18    double x_val = extract("x", point);
19    double y_val = extract("y", point);
20
21    return((m_coeff * x_val) + (n_coeff * y_val) + b_scalar);
22  }
```

The `evalPoint()` function in lines 16-22 is where the actual implementation of $f(x, y) = m \cdot x + n \cdot y + b$ is implemented (on line 21). The argument to this function is a vector of values holding the values for the x and y variables. The ordering of these values i.e., which of the two variables, x or y, is contained in the first value of the vector, is sorted out in the two calls to the `extract()` function in lines 18-19. This sorting out is possible because the ordering is determined by the `IvPDomain` member variable defined at the `AOF` superclass level, and provided in the constructor. The `AOF_Linear` class, used as an example here, is included in the code distribution in `lib_ivpbuild`. It may serve as a template in building a new `AOF_YourAOF` class. It also can be used to verify that the build tools will work in the extreme case of creating a piecewise function with only one piece. And in the case of `AOF_Linear` the piecewise "approximation" is exact.

11.2.3 Another AOF Example Class Implementation for Gaussian Functions

A second function type, implementing Gaussian functions, is implemented as the `AOF_Gaussian` class in the IvP Toolbox in the `lib_ivpbuild` module. This function is a bit more interesting in that a piecewise linear approximation needs multiple pieces to generate a fairly good approximation (understanding that "fairly good" is subjective). It is also interesting in that, depending on the configuration, there may be large portions of the function that are indeed locally linear and in need of relatively few pieces to generate a decent approximation. This function will be used extensively in later examples of usage and performance of the `Reflector` tool. The Gaussian function form is given by:

$$f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$$

The function is defined over the two variables, x and y, and has four parameters. Examples for two groups of parameter settings are shown in Figure 33, and in Figure 34. The coefficient, A, is the amplitude, x_0 and y_0 are the center, and σ represents the spread of the blob. This function is implemented by the class `AOF_Gaussian` shown in Listings 4 and 5. Note that it is a subclass of the `AOF` class and overrides the critical function `evalPoint()`. It also implements the `setParam()` function for setting the four parameters in the above equation..

Listing 11.4: The class definition for the `AOF_Gaussian` *class.*

```
 1  class AOF_Gaussian: public AOF {
 2  public:
 3    AOF_Gaussian(IvPDomain domain) : AOF(domain)
 4      {m_xcent=0; m_ycent=0; m_sigma=1; m_range=100;};
 5    ~AOF_Gaussian() {};
 6
 7    double evalPoint(vector<double> point);
 8    bool   setParam(string param, double value);
 9
10  private:
11    double  m_xcent;
12    double  m_ycent;
13    double  m_sigma;
14    double  m_range;
15 };
```

Listing 11.5: The class implementation for the `AOF_Gaussian` *class.*

```
 1  //-----------------------------------------------------------
 2  // Procedure: setParam
 3
 4  bool AOF_Gaussian::setParam(string param, double value)
 5  {
 6    if(param == "xcent")      m_xcent = value;
 7    else if(param == "ycent") m_ycent = value;
 8    else if(param == "sigma") m_sigma = value;
 9    else if(param == "range") m_range = value;
10    else
11      return(false);
12    return(true);
13  }
```

```
14
15 //---------------------------------------------------------------
16 // Procedure: evalPoint
17
18 double AOF_Gaussian::evalPoint(vector<double> point)
19 {
20   double xval = extract("x", point);
21   double yval = extract("y", point);
22   double dist = hypot((xval - m_x9ent), (yval - m_ycent));
23   double pct  = pow(M_E, -((dist*0ist)/(2*(m_sigma * m_sigma))));
24
25   return(pct * m_range);
26 }
```

An example is shown in Figure 33 below. The domain for both the x and y variables is [-250, 250] containing 501 x 501 = 251,001 points.

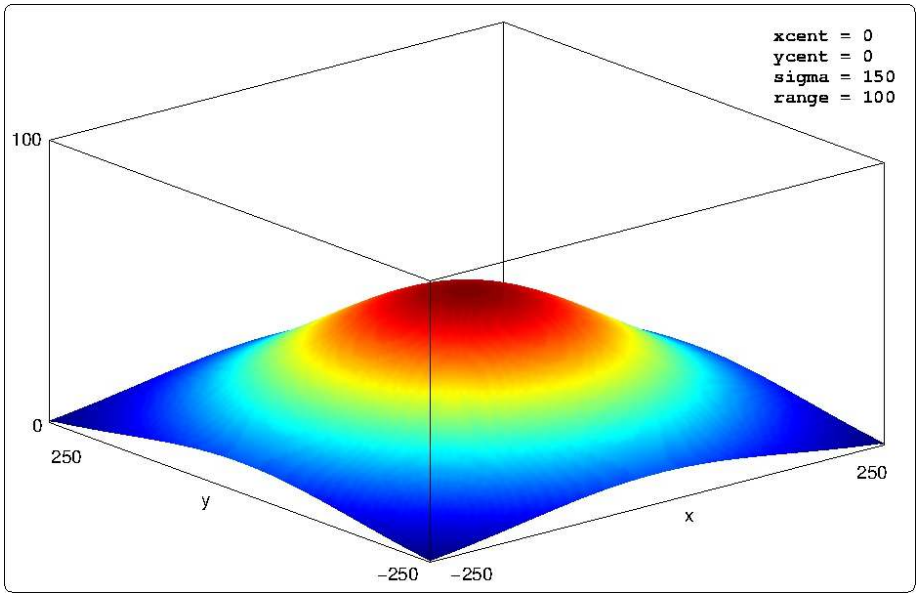

Figure 33: **A Gaussian function**: A rendering of the function $f(x, y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where $A =$ `range` $=$ 100, $\sigma =$ `sigma` $= 150$, $x_0 =$ `xcent` $= 0$, $y_0 =$ `ycent` $= 0$. The domain for x and y ranges from -250 to 250.

11.3 Basic Reflector Tool Usage Tool with Examples

Using the Reflector tool boils down to the four steps below. The third step may be non-existent if the user is building simple uniform functions.

- Step 1: Create the underlying function, AOF instance, and set its parameters.
- Step 2: Create the `Reflector` instance passing it a pointer to the AOF instance.
- Step 3: Set parameters for the `Reflector` if necessary or desired.
- Step 4: Direct the `Reflector` to build the IvP function and then extract it.

A code example of the four steps is provided in Listing 6 below. This code example describes a function that builds and returns an IvP function using the `Reflector` tool. It is not too different from the activity inside a typical implementation of `onRunState()` in an IvP behavior.

Listing 11.6: An example use of the Reflector to create a uniform IvP function.

```
1  IvPFunction *buildIvPFunction(IvPDomain ivp_domain)
2  {
3    // Step 1 - Create the AOF instance and set parameters
4    AOF_Gaussian aof(ivp_domain);
5    aof.setParam("xcent", 50);
6    aof.setParam("ycent", -150);
7    aof.setParam("sigma", 32.4);
8    aof.setParam("range", 150);
9
10   // Step 2 - Create the Reflector instance given the AOF
11   OF_Reflector reflector(&aof);
12
13   // Step 3 - Parameterize the Reflector (None in this case)
14
15   // Step 4 - Build and Extract the IvP Function
16   int  amt_created = reflector.create(1000);
17   IvPFunction *ipf = reflector.extractIvPFunction();
18
19   cout << ``Pieces in the new IvPFunction: `` << amt_created << endl;
20   return(ipf);
21 }
```

The underlying function is created on lines 3-7 creating the Gaussian function with parameters shown in Figure 34. The `Reflector` is created on line 10 with a pointer to the new Gaussian underlying function. In lines 15-16, the `Reflector` creates and returns the IvP function.

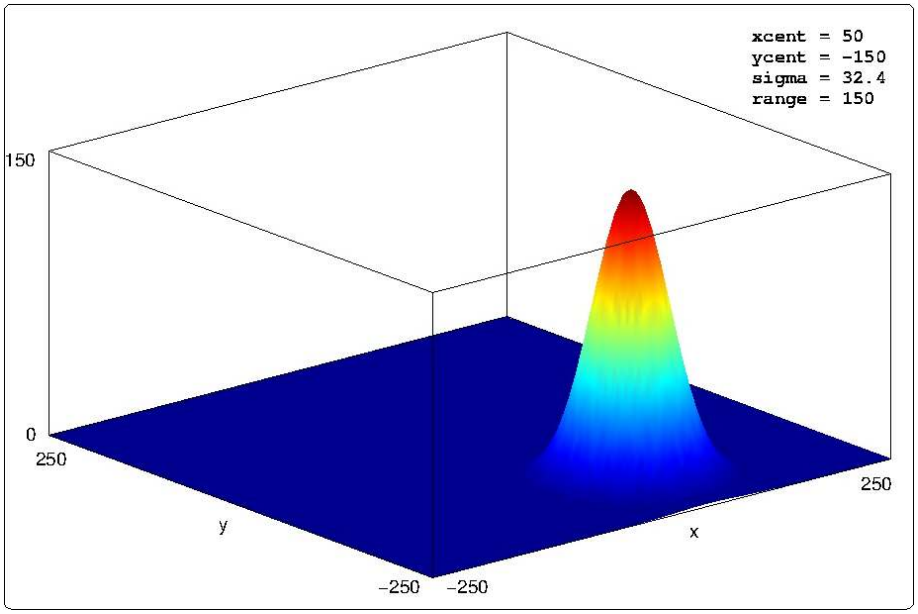

Figure 34: **A Gaussian function**: A rendering of the function $f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where $A =$ `range` $=$ 150, $\sigma =$ `sigma` $= 32.4$, $x_0 =$ `xcent` $= 50$, $y_0 =$ `ycent` $= -150$. The domain for x and y ranges from -250 to 250.

In this simple style of usage, no parameters are set on the `Reflector` after it is created. The result will be an IvP function with uniform piece shape, where the total number of pieces are requested on line 15. (Note that 1000 pieces are requested, but not all requested piece counts are feasible or practical. See Section 12.2.2 for more on this). The requested number of uniform pieces affects three practical metrics of the resulting the IvP function. The error in its representation of the underlying function, the time to create the IvP function, and the number of pieces in the IvP function. The goal is to minimize each, but they are in competition with each other.

Figure 35 depicts four IvP function approximations of the same underlying function, and Table 5 illustrates the relationship between the three metrics of (a) piece count, (b) create time, and (c) accuracy in representing the underlying function. The user determines the most appropriate compromise between these metrics for the application at hand. In general, a gain on one metric is traded off against a sacrifice on other metrics. With the additional tools described in Section 12, it is often possible to make improvements in all three metrics simultaneously. One way to look at this is that there is a fourth metric, *ease-of-use*, that can instead be dialed back to achieve gains in all of the first three metrics. In Listing 6, the absence of Step 3, where insightful parameters could have been provided to the `Reflector` to produce non-uniform functions, could be viewed as optimizing the ease-of-use metric.

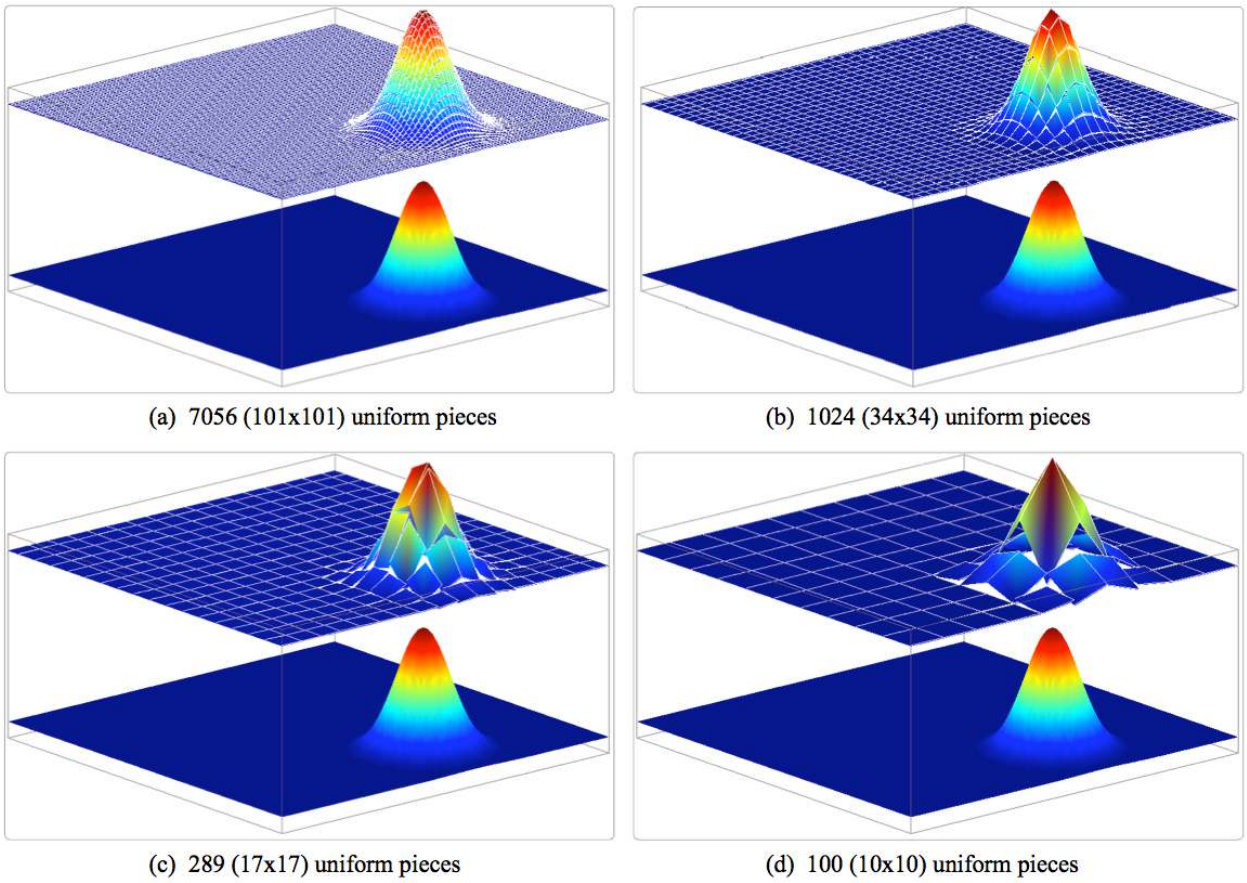

Figure 35: **Four IvP functions approximating the same underlying function**: Each IvP function uses a different number of uniform pieces.

Case	Edge Size	Pieces	Layout	Worst Error	Avg Error	Time (ms)
1	3	27889	(167x167)	0.0761	0.0014	656.4
2	5	1000	(100x100)	0.3019	0.0048	160.0
3	7	5184	(72x72)	0.6720	0.0104	83.3
4	10	2500	(50x50)	1.4589	0.0232	39.9
5	15	1156	(34x34)	3.4532	0.0551	18.9
6	20	625	(25x25)	5.5855	0.1014	10.4
7	25	400	(20x20)	7.79764	0.1585	6.5
8	30	289	(17x17)	12.0347	0.2303	4.7
9	40	169	(13x13)	24.2977	0.3919	2.8
10	50	100	(10x10)	18.2113	0.5917	1.6
11	75	49	(7x7)	42.0652	1.2143	0.9
12	100	25	(5x5)	30.3938	2.0285	0.5

Table 5: **IvP function configurations and metrics**: The relationship between piece size, accuracy and construction time is shown for varying uniform piece size. Four of the row entries are rendered in Figure 35.

11.4 The Full Reflector Interface Implementation

The following functions define the interface to the Reflector tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested. Example usage is provided in Listing 6.

- `OF_Reflector(AOF*)`: The constructor takes a single argument, a pointer to the underlying function to be approximated by the Reflector. The `AOF` instance contains an instance of the IvPDomain which will also be the IvPDomain of any IvP functions created with the Reflector.
- `int create(int pieces=-1)`: This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. Many of the parameters affecting the form of the function are settable separately in the `setParam()` function, including the parameter specifying the number of pieces. If the optional `pieces` argument is provided in this function call, and if the value of the argument is ≥ 1, this overrides any piece count request set otherwise. This function will create an IvP function that the user can then obtain via the function `extractIvPFunction()` described below. The integer value returned is the number of pieces in the newly created IvP function. A value of zero indicates something has gone wrong.
- `IvPFunction *extractIvPFunction()`: This function returns a new IvP function built during a prior invocation of the `create()` function described above. If an error was encountered in either the parameter setting attempts, or in the invocation of the `create()` function, this function will simply return the NULL pointer. When the IvP function is extracted from the Reflector, an IvPFunction instance is created from the heap that needs to be later deleted. The Reflector tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.
- `string getWarnings()`: When or if problems are encountered in setting the parameters, the Reflector appends a message to a local warning string. This string can be retrieved by this function.
- `bool stateOK()`: This function returns true if no errors were encountered during configuration

attempts, otherwise it returns false. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the `getWarnings()` function above can be consulted.

- `bool setParam(string param, string value)`: This function is used for setting parameters on many optional tools more advanced than specifying the number of pieces to be used in a simple uniform function. An overview is provided here, with more detailed deferred to later sections that cover the advanced tools.

- `uniform_amount`: The amount of pieces to use in the creation of a simple uniform function. Alternatively can be supplied in the call to `create()` as described above.
- `uniform_piece`: A string description of the size and shape of a piece used during the creation of a pure uniform function. Details described in Section 12.1.2.
- `strict_range`: When set to true, the range of the linear interior function is guaranteed to stay within the range of any sampled points of the underlying function, even if a better overall fit could be obtained otherwise. The default is true.
- `refine_region`: A string description of a region of the IvP domain within which further directed refinement is requested. See Section 12.1.3.
- `refine_piece`: A string description of the size and shape of uniform pieces to be used within a region of directed refinement. See Section 12.3.
- `refine_point`: A string description of a point within the IvP domain to direct further refinement. See Section 12.3.
- `smart_amount`: The number of pieces to use in the smart refinement algorithm, beyond the number of pieces used in an initial simple uniform function. See Section 12.4.
- `smart_percent`: The number of pieces to use in the smart refinement algorithm specified as a percentage of the number of pieces used in an initial simple uniform function. See Section 12.4
- `smart_thresh`: A threshold given in terms of worst noted error between the IvP function and the underlying function, below which the smart refinement algorithm will cease further refinement. See Section 12.4.
- `auto_peak`: Set to either true or false indicating whether the auto-peak algorithm should be applied. See Section 12.5.
- `auto_peak_max_pcs`: The maximum amount of new pieces added to an IvP function during the auto-peak heuristic. See Section 12.5.

- `bool setParam(string param, double value)`: The parameters that may be set via this function may also be set via the `setParam()` function above where the `value` parameter is a string. This alternate method is implemented solely as a convenience to the caller.

- `uniform_amount`: See above.
- `smart_amount`: See above.
- `smart_percent`: See above.
- `smart_thresh`: See above.
- `auto_peak_max_pcs`: See above.

12 Optional Advanced Features of the Reflector Tool

12.1 Preliminaries

The previous section discussed how to build IvP functions with the `Reflector` tool in the IvP Build Toolbox by simply specifying a desired number of pieces in the resulting piecewise defined function. This section discusses a few further methods for building functions that give the user more control of the build process and typically better overall results in terms of fewer pieces, less time to build, and greater accuracy in the piecewise approximation of the underlying function.

12.1.1 The Reflector-Script

The basic invocation of the Reflector `create()` function may take a single argument requesting the number of pieces to be used in the piecewise function. An example is line 15 in Listing 6. In reality the invocation of `create()` is comprised of a *script* of distinct build heuristics of which the creation of uniform sized pieces is just the first of four parts. The latter three parts are optional and require further user configuration before being included for execution in the script. The four parts are:

- Uniform function creation
- Directed refinement
- Smart refinement
- Auto-Peak refinement

These four heuristics are discussed in the next four sections. Uniform function creation is revisited since finer control can be used (with typically better results) if the choice of piece size and shape is not left to the heuristic that converts the requested total number of pieces into an actual uniform piece shape.

12.1.2 Specifying a Piece Shape or IvP Domain Point in String Format

Aspects of the `Reflector` tool require the specification of the shape of a piece used in a piecewise defined IvP function. The specification is comprised of the length of the piece for each of the n dimensions, i.e., decision variables. There are two ways to describe the lengths. Recall that the IvP domain for a variable is given by a low and high value, and the number of points. For example the variable x could range from 0 to 30 with 31 points, and y could range from -50 to 50 with 21 points. The first way to describe the length of a piece is by specifying the number of discrete points:

```
"discrete @ x:5,y:5"
```

A uniform function built over this domain with the above requested piece shape would have 35 pieces in a manner rendered in Figure 36.

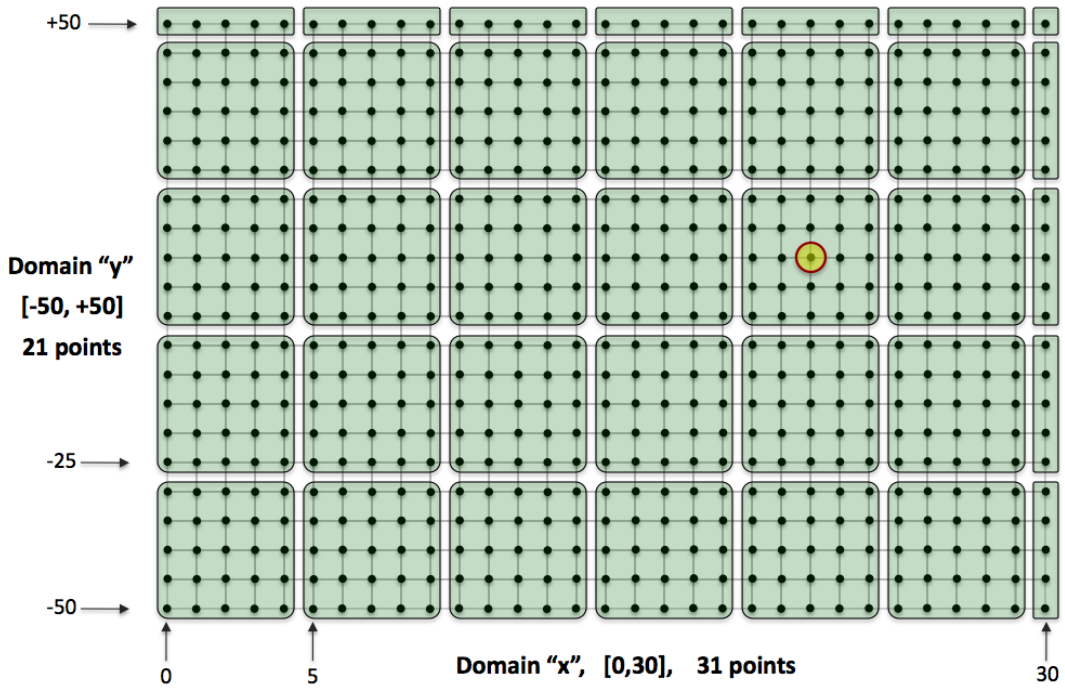

Figure 36: **A uniform IvP function**: An IvP domain is rendered over the two variables x, with 31 elements, and y with 21 elements. Requesting a set of uniform pieces with five elements on each edge results in the piece distribution shown. The circled point represents the 23rd index into the x domain and the 13th index into the y domain. This point can be referenced by the string `"discrete @ x:22,y:12"`. It may also be referenced by the string `"native @ x:22,y:10"`.

Note the distribution of pieces is not completely uniform. Smaller pieces are used at the upper ranges of the domain. A second method of specifying the same piece shape is to use the native lengths of the domain:

```
"native @ x:5,y:25"
```

This piece also has a length of five units along the x dimension and five units along the y dimension, resulting in the same distribution shown in Figure 36. When a "native" value doesn't exactly map onto one of the points in the domain, it is rounded to the nearest domain point. For example, `"native @ x:5,y:22.6"` specifies a piece with *five* units on the y dimension, `"native @ x:5,y:22.4"` specifies a piece with *four* units on the y dimension. And when a native value is given exactly between two domain points, the value is rounded up, so `"native @ x:5,y:22.5"` specifies a piece with *five* units on the y dimension.

A single *point* in the IvP domain can be similarly referenced. When the string `"discrete @ x:5,y:5"` is used to represent a piece *shape*, the numerical values represent the length of the piece. When the same string is used to represent a *point* in the IvP domain, the numerical values represent the index into the domain. For example, the circled point in Figure 36 can be be referenced by the string `"discrete @ x:22,y:12"`. It may also be referenced by the string `"native @ x:22,y:10"`. When a native values does not map exactly to a domain value, the nearest domain point is used.

12.1.3 Specifying a Region of an IvP Domain in String Format

Aspects of the `Reflector` tool require the specification of a `region` of the IvP domain. The specification is comprised of an upper and lower bound for each of the n dimensions, i.e., decision variables. Recall that the IvP domain for a variable is given by a low and high value, and the number of points. For example the variable x could range from 0 to 30 with 31 points, and y could range from -50 to 50 with 21 points. A region can be specified as follows:

```
"native @ x:10:24,y:-25:20"
```

or equivalently,

```
"discrete @ x:10:24,y:5:14"
```

This region is rendered in Figure 37. If the extents specified in the string exceed the boundaries of the IvP domain, the requested region is clipped to be exactly the boundary value. For example, the string `"native @ x:10:24, y:-25:50"` and `"native @ x:10:24, y:-25:50000"` would specify the same region given the example in Figure 37.

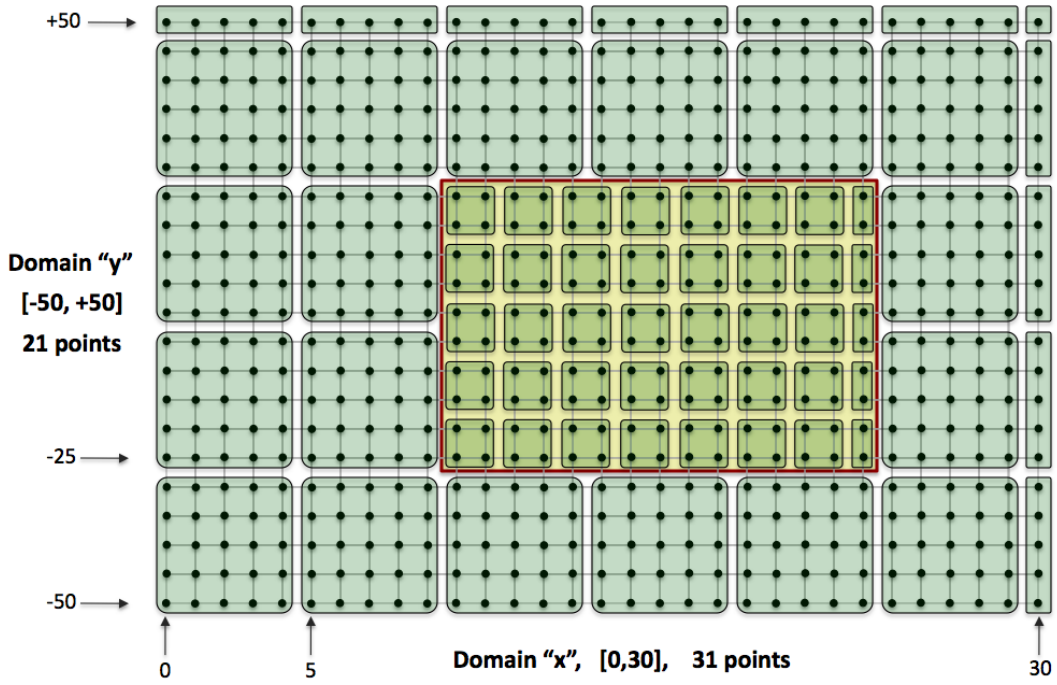

Figure 37: **A non-uniform IvP function**: An IvP domain is rendered over the two variables, x, with 31 elements, and y, with 21 elements. A region of IvP domain is identified for further application of the `Reflector`. The region is specified by the string `"discrete @ x:10:24, y:5:14"` or `"native @ x:10:24, y:-25:20"`. In this case smaller uniform pieces are applied within the region.

When a native value is specified that does not map to a domain value, this case is handled differently for *regions* than it was when specifying a piece *shape*. In a region specification the native value is treated as a strict boundary value. Therefore the string `"native @ x:9.01:24.99, y:-29.99:24.99"` would specify the exact same region as the example above and in Figure 37.

12.2 Optional Feature #1: Choosing the Piece Shape in Uniform Functions

12.2.1 Potential Advantages

By simply specifying the desired number of pieces, the `Reflector` heuristically sets the piece size and aspect ratio of an initial uniform function. This has the advantage of being very simple and independent of the underlying function. (See line 15 in Listing 6.) However, like most heuristics, there may be cases where the result may not be best for a particular situation. If the user has some insight into the underlying function and the IvP domain, the user may not wish to leave this decision to the heuristic, but instead specify the piece shape explicitly. Below, the piece count-to-piece shape heuristic is described as well as how to override the heuristic with an explicit shape request.

12.2.2 Specifying the Piece Shape Implicitly from a Piece Count Request

When the `Reflector` creates a uniform IvP function based on a requested piece count, a heuristic is invoked to generate a single piece to be used in the uniform function based on both the piece count and the IvP domain. This piece is not unlike the 5 x 5 piece in Figure 36, except that a 5 x 5 piece is not explicitly requested, but rather the total pieces in that figure, 35, would be requested. Knowing a little about this heuristic can help determine when its worth the effort to instead explicitly define the shape of the uniform piece. The total requested pieces is an upper limit, and often not exactly achieved. For example, the same 35 pieces in Figure 36 would be created upon piece-count requests of 35, 36, 37, 38, and 39 pieces. The heuristic attempts to keep the aspect ratio of the uniform piece close to 1.0, but will deviate to allow a uniform piece that will result in a total number of pieces closer to the requested amount. The heuristic is given Listing 1 below, and some examples are shown in Table 6.

Listing 12.1: The heuristic for generating a uniform piece based on piece-count and domain.

```
 1  IvPBox buildUniformPiece(IvPDomain domain, int max_amount)
 2  {
 3    int dim = domain.getDim();
 4    vector<int>  pcs_on_edge(dim,1);
 5    vector<bool> pcs_maxed(dim,false);
 6    vector<int>  pts_on_edge(dim,0);
 7
 8    // Store the number of points on an edge for quick reference
 9    for(i=0; i<dim; i++)
10      pts_on_edge[i] = domain.getVarPoints();
11
12    // Augment the number pieces on edges until done
13    bool done = false;
14    while(!done) {
15      // Algorithm done if augmentations for all dimensions are maxed out.
16      done = true;
17      for(i=0; i<dim; i++)
18        done = done && pcs_maxed[i]
19
20      // Find the dimension most worthy of further augmentation
21      if(!done) {
22        int    augment_dim;
23        double biggest = 0;
24        for(d=0; d<dim; d++) {
25            if(!pcs_maxed[d]) {
26            double ratio = (pts_on_edge[d] / pcs_on_edge[d]);
27            if(ratio > biggest) {
```

```
28              biggest = ratio;
29              augment_dim = d;
30            }
31          }
32        }
33
34        // Augment the pieces_on_edge for the chosen dimension
35        pcs_on_edge[augment_dim]++;
36
37        // Calculate hypothetical number of boxes given new augmentation.
38        double hypothetical_total = 1;
39        for(d=0; d<dim; d++)
40          hypothetical_total *= pcs_on_edge[d];
41
43        // If max_amount exceeded, undo the augment, and max-out the dimension
44        if(hypothetical_total > max_count) {
45          pcs_maxed[ix] = true;
47          pcs_on_edge[augment_dim]--;
48        }
49
50        // Cant have more pieces on an edge than points on an edge
51        if(pcs_on_edge[augment_dim] >= pts_on_edge[augment_dim])
52          pcs_maxed[augment_dim] = true;
53      }
54    }
55
56    // Now build the uniform piece based on pts_on_edge and pcs_on_edge
57    IvPBox uniform_piece(dim);
58    for(d=0; d<dim; d++) {
59      double edge_size = ceil(pts_on_edge[d] / pcs_on_edge[d]);
60      uniform_piece.setPTS(d, 0, edge_size-1);
61    }
62    return(uniform_piece);
63  }
```

The heuristic progresses by growing the number of "pieces on an edge", `pcs_on_edge`, on each dimension. The algorithm proceeds to grow the `pcs_on_edge` for each dimension until it cannot grow further. For example, in Figure 36 there are seven pieces on the x edge and five pieces on the y edge. The algorithm is initiated with a single piece on each edge, i.e., dimension, (line 4 in Listing 1). A Boolean is associated with each dimension indicating whether growth in that dimension has been maxed out. This vector is initiated on line 5. A dimension becomes maxed out if additional growth in that dimension means the requested piece count is exceeded (checked for in lines 37-48), or if the number of pieces on an edge is equal to the number of points on and edge of the IvP domain (checked for in lines 49-51). At each chance to grow the size of the uniform piece the most appropriate dimension is identified for growth (lines 22-33) by choosing the dimension with the largest ration of points on the edge to pieces on the edge (line 26).

Some examples of the heuristic are shown in Table 6. The domain shown in table has 1000 discrete choices for both the x and y variables. Given that the domain itself has an aspect ratio of one, not surprisingly, the generated uniform pieces also have roughly an aspect ratio of 1.0, and the number of pieces on each edge of the domain are also nearly equivalent.

Requested Pieces	Aspect Ratio	Shape of Piece	Actual Pieces	Pieces On the 'x' Domain Edge	Pieces On the 'y' Domain Edge
63	0.78	112x143	63	9	7
64	1.0	125x125	64	8	8
500	1.0	46x46	484	22	22
512	0.96	44x46	506	23	22
1000	0.97	32x33	992	32	31
1024	1.0	32x32	1024	32	32
1025	1.0	32x32	1024	32	32
4000	1.0	16x16	3969	63	63

Table 6: **Example 2D results of the uniform-piece heuristic**: Uniform piece characteristics resulting from a heuristic applied to a requested total number of pieces and a given IvP domain with two variables. In this case the IvPDomain is x:200:299:1000, y:0:999:1000.

Consider how the heuristic performs instead on the 3D domain shown in Table 7. The number of choices for the z variable is a tenth of that for the x and y variables. The results provided by the heuristic may or may not be the right overall, depending on the underlying function and application. In particular consider that when requesting 100 or 200 pieces, the z component of the resulting uniform piece is the entire z domain, i.e., there is only one piece on the z domain edge.

Requested Pieces	Shape of Piece	Actual Pieces	Pieces On the 'x' Domain Edge	Pieces On the 'y' Domain Edge	Pieces On the 'z' Domain Edge
100	100x100x100	100	10	10	1
200	72x72x100	196	14	14	1
1000	44x46x50	968	22	22	2
7500	23x24x25	7392	44	42	4

Table 7: **Example 3D results of the uniform-piece heuristic**: Uniform piece characteristics resulting from a heuristic applied to a requested total number of pieces and a given IvP domain with three variables. In this case the IvPDomain is x:200:299:1000, y:0:999:1000, z:0:99:100.

This heuristic has served fairly well in practice, but in cases where the user has insight into a better choice for the size and shape of the uniform piece, this can be overridden as discussed next.

12.2.3 Specifying the Uniform Piece Shape Explicitly

The piece shape used in a uniform IvP function can be set explicitly using the `uniform_piece` parameter in the Reflector `setParam()` function first mentioned in Section 11.4. For example, the uniform piece shown in Figure 38 can be requested as follows:

```
reflector.setParam("uniform_piece", "discrete @ x:6,y:4");
int amt = reflector.create();
```

Compared to generating a uniform function by a simple piece-count request, the above two lines would replace the single line with the `create()` invocation, as in line 15 in Listing 6.

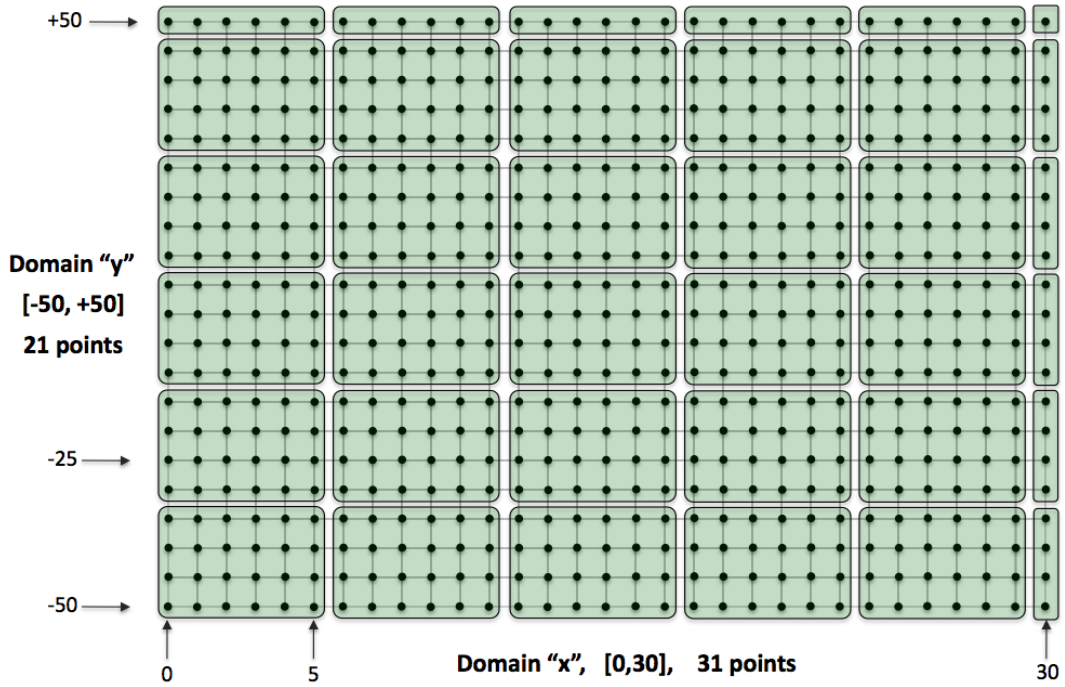

Figure 38: **An IvP function made from an explicit piece shape request**: An IvP domain is rendered over the two variables x, with 31 elements, and y with 21 elements. Requesting a uniform piece of size 6x4 would result in the rendered configuration. This piece can be specified with `"discrete @ x:6,y:4"` or `"native @ x:6,y:20"`. This piece shape would *not* be resulting piece shape had the user simply requested 36 pieces given this domain.

In summary, when the Reflector `create()` function is called, the reflector-script begins and needs to know the size and shape of the piece used for uniform function creation. It may get this information by either explicitly configuring the piece shape, or implicitly by requesting a total number of pieces (as an argument to the `create()` function). If both requests are inadvertently invoked, the latter type of request is ignored and the explicit piece shape configuration is honored. If neither specification of piece shape is provided, a function with a single piece will be created (but perhaps further refined in later parts of the reflector-script). Use of the explicit piece shape request may be the preferred method for example if a domain includes a variable for vehicle heading and a uniform function is desired with pieces split on every three degrees, regardless of whether the domain contains 180, 360, or 720 choices for heading.

12.3 Optional Feature #2: IvP Functions with Directed Refinement

The directed-refinement feature of the Reflector is potentially useful when (a) the underlying function has distinct sub-regions that are harder to accurately represent with a piecewise linear approximation, and (b) when the user has insight into the location of those sub-regions. Use of the tool involves specifying both the region to direct further refinement, and the size of the piece to use in the refinement region. This is done using the `uniform_piece`, `refine_region`, and `refine_piece` parameters in the Reflector `setParam()` function first mentioned in Section 11.4. For example, the IvP function shown in Figure 37 would be generated with the following lines:

Listing 12.2: An example configuration of the `Reflector` *tool using directed refinement.*

```
reflector.setParam("uniform_piece", "discrete @ x:5,y:5");
reflector.setParam("refine_region", "native @ x:10:24,y:-25:20");
reflector.setParam("refine_piece",  "discrete @ x:2,y:2");
reflector.create();
```

When the `create()` function is invoked in the last line above, the reflector-script will involve two of the components of the reflector-script mentioned in Section 12.1.1. The first line configures the initial uniform function phase, and the middle two lines configure the directed refinement phase by declaring a sub-region (second line) and a uniform piece to be applied to that sub-region (third line). Multiple directed refinements can be configured and queued for inclusion in the reflector-script by adding further `refine_region` - `refine_piece` pairs prior to the invocation of the `create()` function. They must be added in pairs however since the `refine_piece` is always associated with the last specified `refine_region`.

For an illustrative case we return to the Gaussian function rendered in Figure 35:

$$f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})},$$

where $A = 150$, $x_0 = 50$, $y_0 = -150$ and $\sigma = 32.4$. This function apparently has a sub-region of the domain where the function is very nonlinear and otherwise quite linear outside the sub-region. The use of directed-refinement begins by building an initial uniform function as shown in Figure 39, conceding for now that the approximation will be poor in the sub-region around the peak.

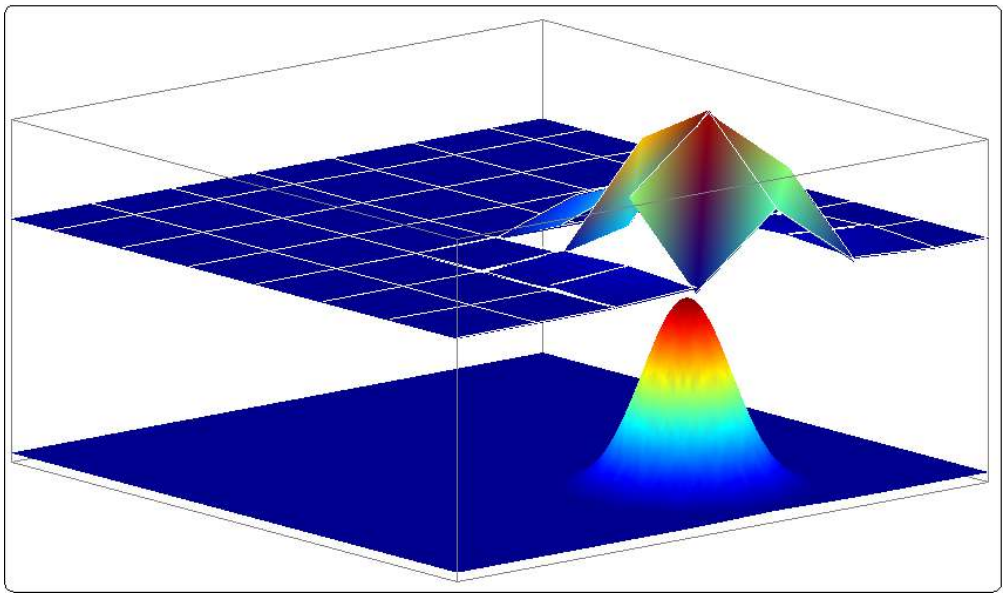

Figure 39: **An initial IvP function approximation**: The `Reflector` first creates an initial simple uniform function with fairly large pieces, conceding for the time being poor performance in approximating the underlying function in areas near the peak of the underlying function.

The initial uniform function was created by requesting 50 pieces, and a function with 49 pieces was subsequently generated. The sub-region shown in Figure 40 was identified for directed refinement, with much smaller pieces used in the sub-region.

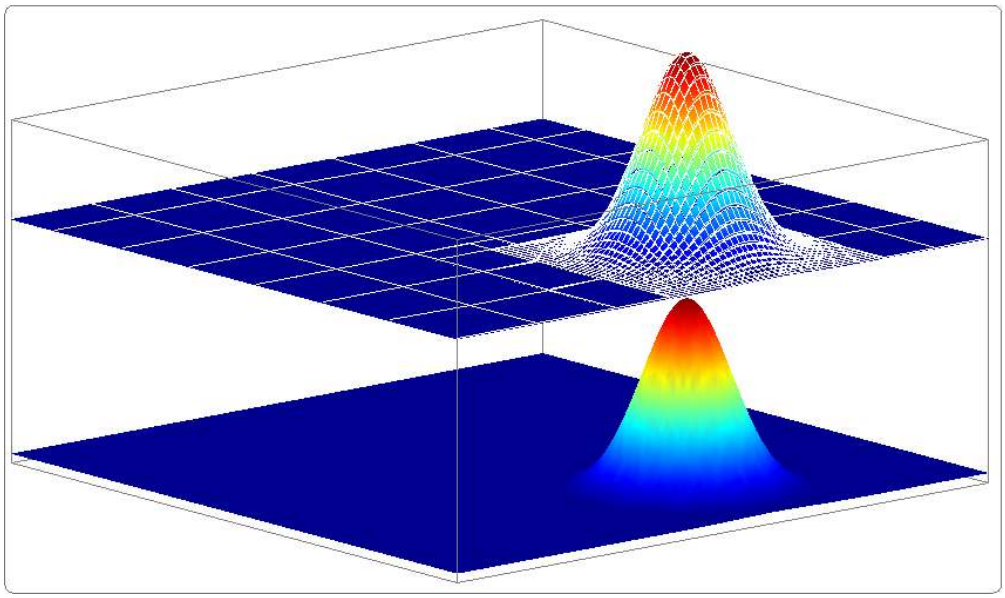

Figure 40: **An IvP function generated with directed-refinement**: After an initial uniform function has been generated, the `Reflector` refines the function on the prescribed sub-domain of the function with much smaller pieces.

The results in Table 8 below were generated by configuring the reflector-script to include directed-refinement on the underlying Gaussian function shown in Figure 40, in a manner similar to the four lines in Listing 2. Each row in the table below differs only in the size of the `refine_piece` shown in the second column.

Case	Refine Edge Size	Total Pieces	Worst Error	Average Error	Time millisecs
A	4	2607	0.7760	0.0104	38.9
B	5	1687	0.7760	0.0123	25.6
C	6	1162	0.7760	0.0149	17.8
D	7	847	0.7760	0.0178	13.0
E	8	682	0.9093	0.0214	10.2
F	9	535	1.1895	0.0254	8.2
G	10	447	1.4589	0.0302	6.7
H	11	367	1.8263	0.0351	5.7
I	12	295	2.2470	0.0409	4.7
J	13	262	2.6527	0.0472	4.1
K	14	231	3.0321	0.0540	3.6
L	15	202	3.5886	0.0615	3.2

Table 8: **Results of directed-refinement**: Characteristics of 12 different IvP functions approximating the underlying function shown in Figure 39 and 40. Each function is built by starting with an initial uniform function and then performing directed refinement over the region $-50 \leq x \leq 150$ by $-250 \leq y \leq -50$. The refinement piece size shown in the second column is the parameter that results in the 12 different functions.

For the observed errors reported in columns 4 and 5, the domain was sampled for each resulting IvP function at 50,000 random points for comparison between the value provided by the IvP function against the underlying function. The average time to create the IvP function, noted in the last column, was taken by averaging 100 creations since the precision of the timer used was 100 milliseconds. The data shown here are meant to show the relationship between parameters, not necessarily an indication of how fast things run on "typical" platforms. That being said, this data is from a Dell laptop containing a Pentium chip with about 2.0 GHz processor, with a codebase compiled without typical gcc optimization options.

The trends in the table are as one would expect. As the number of pieces is decreased, the average error and worst error increase, and the time to create the IvP function is decreased. The question is whether this technique offers the ability to improve in all three metrics, piece-count, function accuracy, and creation time, simultaneously compared to using a simple uniform function without directed refinement. The answer is yes. Evidence can be seen of this by comparing Table 8 with Table 5. We look for cases in Table 8 that dominate cases in Table 5. A case that dominates another is stronger or equal in all three performance metrics simultaneously. Case (a) dominates case (4). Case (b) dominates cases (5),(4). Case (c) dominates cases (6),(5). Case (d) dominates cases (6),(5). Case (e) dominates cases (7),(6).

12.4 Optional Feature #3: IvP Functions with Smart Refinement

12.4.1 Potential Advantages

The smart-refinement algorithm works by further refining an existing IvP function based on an (automated) estimate of which pieces need refinement the most. There are two key ideas in this algorithm. First, no insight into the underlying function form is required by the user, unlike the directed-refinement tool. Second, the prioritization of pieces is based on the apparent fit between a piece's linear function and the underlying function, for the sub-domain of that piece. This determination of fit can be measured by performing very little extra computations beyond the already required calculations performed during linear regression for each piece during the uniform and directed-refinement phases. In short, there is typically very little reason not to invoke this tool to some degree.

12.4.2 The Smart-Refinement Algorithm

The smart-refinement algorithm utilizes information collected during the creation of pieces earlier in the reflector-script, during the initial uniform function phase which is always invoked, and directed-refinement phase which is optionally invoked. During these phases, pieces are formed and linear regression is performed to determine the linear function associated with each piece.

To perform linear regression for a new piece, the underlying function over k variables is sampled at $n = 2k + 1$ points (the corners and the middle point) to produce $f(\vec{x}_1) \ldots f(\vec{x}_n)$, and these n values are used to determine the linear function for that piece:

$$f'(\vec{x}) = c_1 x_1 + \ldots + c_k x_k + b.$$

The same n points are again evaluated using this newly determined linear function instead, producing another set of n values $f'(\vec{x}_1) \ldots f'(\vec{x}_n)$. The `regression score` is determined by:

$$\texttt{regression\_score} = \text{sqrt} \sum_{i=1}^{n} (f'(\vec{x}_i) - f(\vec{x}_i))^2$$

The `regression_score` is then inserted into a priority queue along with a reference to the piece that generated the score. The idea is shown in Figure 41. This algorithm for implementing a priority queue can be found in [**?**]. The priority queue implemented in the IvPBuild Toolbox is modified slightly to be a fixed-length queue. Insertion and retrieval time is $O(n\ log(n))$.

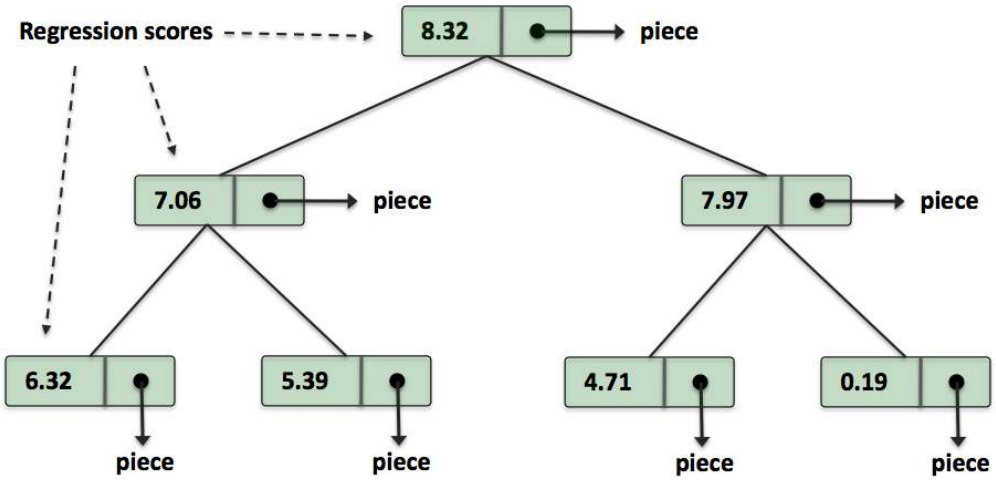

Figure 41: **Priority queue keyed with regression scores**: The `Reflector` uses a balanced priority queue based on a regression score to determine which pieces could benefit the most from further refinement.

The `Reflector` instance maintains this priority queue only if smart-refinement is activated. The pieces made during the initial uniform function and directed-refinement parts of the reflector-script are stored in the priority queue. The smart-refinement proceeds by repeatedly popping the top priority piece from the queue for further refinement. By further refinement of a piece, we mean splitting a piece and replacing the piece with the two new pieces after performing regression on the two new pieces. The piece is split along the dimension with the largest edge. These two new pieces are then also inserted in the priority queue for possible further refinement.

An example of the smart-refinement algorithm applied to the same Gaussian function shown in Figure 40 is shown below in Figure 42.

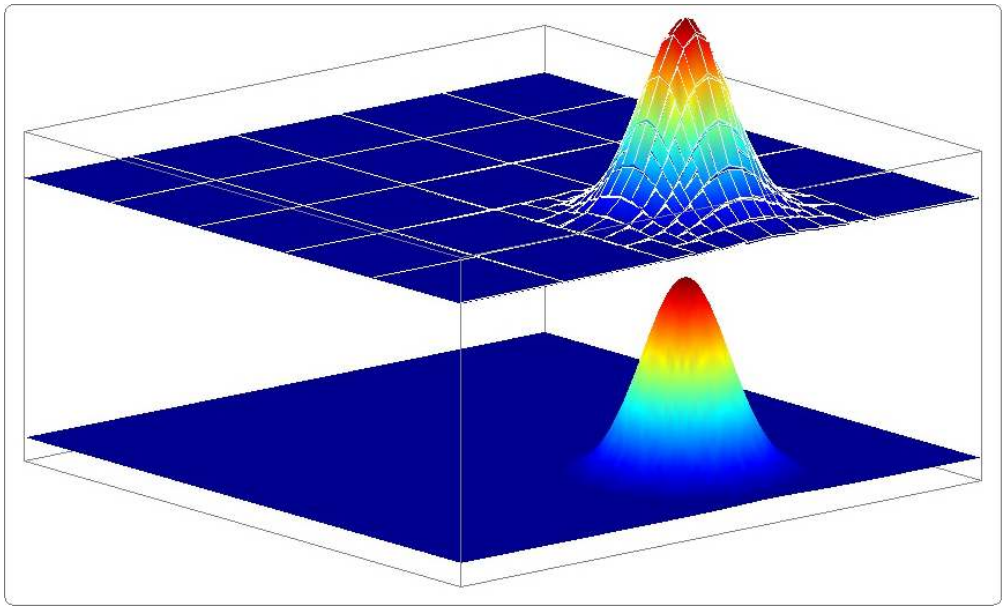

Figure 42: **An IvP function generated with smart-refinement**: Results of smart-refinement on an initial uniform function with 25 pieces and an additional 200 pieces during the smart-refinement phase. The function has a total of 225 pieces and is significantly more accurate and faster to create than a pure uniform function with 625 pieces. Further examples are shown in Table 9.

The results in Table 9 show the results of applying the smart-refinement algorithm to the function in Figure 34. Each row in the table shows the results from creating first a pure uniform function, and then a further refined function using an additional 75% more pieces with smart-refinement. The left-hand side of the table is the same as Table 5, duplicated here for ease of comparison. Compare for example the smart-refine function with 175 pieces in Case 10 against the pure uniform function with 400 pieces in Case 7. The former not only has less pieces, but is more accurate and took less time to create. It dominates the pure uniform function, i.e., is simultaneously better in all measures of performance. This is similar to the way directed-refinement dominates pure uniform functions, but in the case of smart-refinement, no insight into the underlying function form was required!

Case	Edge Size	Pieces	Worst Error	Avg Error	Time msec	Pieces	Worst Error	Avg Error	Time msec
4	10	2500	1.4589	0.0232	39.9	3445	0.8512	0.0139	70.1
5	15	1156	3.4532	0.0551	18.9	2023	1.3241	0.0224	47.3
6	20	625	5.5855	0.1014	10.4	1093	1.9834	0.0297	27.6
7	25	400	7.79764	0.1585	6.5	700	2.5924	0.0362	18.1
8	30	289	12.0347	0.2303	4.7	505	2.3905	0.0480	11.3
9	40	169	24.2977	0.3919	2.8	295	12.6192	0.0885	7.5
10	50	100	18.2113	0.5917	1.6	175	3.4166	0.1194	4.3
11	75	49	42.0652	1.2143	0.9	85	7.9236	0.3100	2.3
12	100	25	30.3938	2.0285	0.5	43	12.3887	0.5188	1.2

Table 9: In each case an initial uniform function was created with the number of pieces indicated in column 3. The qualities of the function in terms of accuracy and time are shown in columns 4-6. Each function was then augmented with 75% additional pieces using the smart-refinement algorithm with the resulting qualities shown in columns 7-10.

The smart-refine algorithm is limited by the degree to which the regression score is accurate for each piece entered into the queue. Note for example, Case 9 in Table 9, where the worst error detected for the smart-refine function is anomalous and significantly higher than that noted in functions with far fewer pieces. The error of 12.6192 occurred in some piece that apparently did not report a high regression score. This is likely due to an unfortunate case where the points sampled for use in generating the linear function all fit the resulting linear function very well, but the non-sampled points did not fit well. The idea is shown in Figure 43.

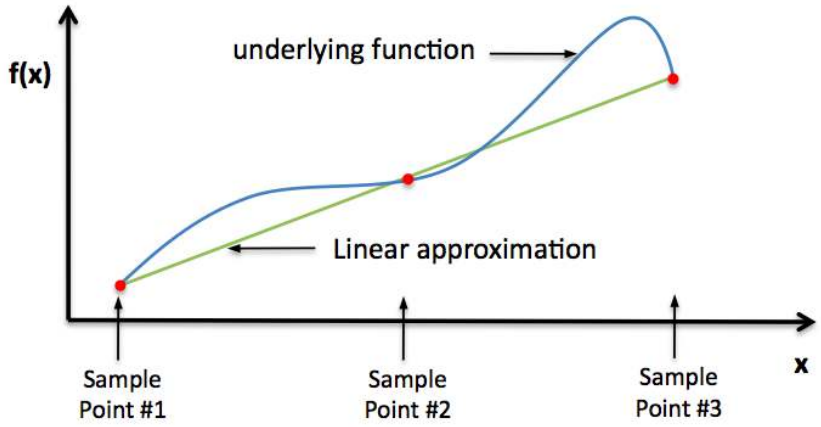

Figure 43: **Regression scoring gone awry**: Assessing the regression score can be misleading in cases where the derived linear function fits each sampled point very well but otherwise poorly fits the underlying function.

12.4.3 Invoking the Smart-Refine Algorithm in the Reflector

The smart-refine tool can be included in the refine-script in a few different ways. The first is to simply indicate how many pieces to use during the smart-refine process. The number of pieces is addition to any pieces that may already be present after the initial uniform function has bee created and after any directed-refinement has been performed.

```
reflector.setParam("smart_amount", 400);
```

Alternatively the number of pieces to be used in smart-refine can be given in terms of the percentage of additional pieces beyond what has already been created at the time of invocation of the smart-refine part of the refine-script. The argument is a non-negative integer value:

```
reflector.setParam("smart_percent", 35);
```

There is a third parameter *smart_thresh* that can affect how many pieces are used in the smart-refine phase. The value passed with this parameter is a regression score such that if the current top element of the priority queue has a score below this threshold, smart-refinement will terminate early, before the target piece amount specified by *smart_amount* or *smart_percent* has been reached:

```
reflector.setParam("smart_thresh", 0.05);
```

Regression scores represent the raw discrepancy between the underlying function and the linear approximation, and in general do not reflect any normalization. For example, depending on the

function, the value of 0.05 above could be a relatively large value resulting in an early termination of refinement, or a relatively small threshold that cannot be met without thousands of additional pieces. The user of this parameter needs to have some knowledge of the range of the underlying function.

Finally, the simplest way of invoking the smart-refinement tool is by specifying the number of pieces as the second argument in the `create()` function call, and the smart threshold as the third argument:

```
reflector.create(1000, 400, 0.5);
```

The above will result in an initial uniform function with 1000 pieces and an additional 400 pieces used for smart-refinement. The full additional 400 pieces will be generated only if the threshold is not reached along the way. The above is equivalent to:

```
reflector.setParam("uniform_amount", 1000)
reflector.setParam("smart_amount", 400)
reflector.setParam("smart_thresh", 0.5)
reflector.create();
```

Accepting these three common parameters as arguments to the `create()` function call is simply for convenience. If provided, they override the setting from prior calls to `setParam()`.

12.5 Optional Feature #4: IvP Functions with Auto-Peak Refinement

12.5.1 Potential Advantages

The auto-peak algorithm is the last optional algorithm in the reflector-script. The objective is also to build a more accurate IvP function representing the underlying function. The metric of accuracy referenced up to this point has been the average error and worst error observed from a number of random sample points. For example Table 9 reported error in this way. Another metric of accuracy is the degree to which the maximum peak of the function, represented by a point in the discrete IvP domain, agree between the underlying function and the IvP function. For example, if the peak of the underlying function is `"heading=133, speed=3.2"` and the peak of the IvP function is `"heading=129, speed=3.0"` the IvP function could still be rated well in terms of error metrics from sampling the entire domain. However, the peak of the function is probably the most important part of the underlying function to represent precisely. When the IvP function happens to be the only function or dominating function influencing the vehicle at that moment, the peak of the function *is* the output of the solver. The auto-peak refinement focuses on this aspect of the function, without user insight into where the actual peak occurs in the underlying function.

12.5.2 The Auto-Peak Algorithm

The auto-peak algorithm proceeds by repeatedly refining the single piece in the IvP function that is believed to contain the maximum peak until that one piece contains only a single point in the IvP domain. It takes advantage of the fact that for a given piece in an IvP function, its maximum value, over the piece interval and linear interior function, can be rapidly calculated. The basic algorithm steps are as follows:

Listing 12.3: An overview of the auto-peak algorithm.

```
Step  1.  Populate the priority queue with the max-value for each piece.
Step  2   If the top piece in the priority queue has only one IvP domain point, go to 10.
Step  3   If the number of pieces added so far has reached an upper limit, go to 10.
Step  4.  Pop the top piece in the priority queue for further refinement.
Step  5.      Split the piece along the longest edge.
Step  6.      Build the new linear function for both pieces, noting max values.
Step  7.      Add the two new pieces back to the IvP function.
Step  8.      Add the two max-values back to the priority queue.
Step  9.      Go to Step 2.
Step 10.  Done.
```

The first step in the algorithm is to build a priority queue similar to the priority queue used in the smart-refinement algorithm. In this case, the score associate with each piece in the queue is the maximum value for the given piece, as depicted in Figure 44 below. The maximum value is calculated quickly and directly from the coefficients of the linear function associated with the piece. When the auto-peak algorithm is initiated, it works with the IvP function generated thus far during the prior phases of the reflector-script. The priority queue is built by evaluating the max-value for all pieces in this function.

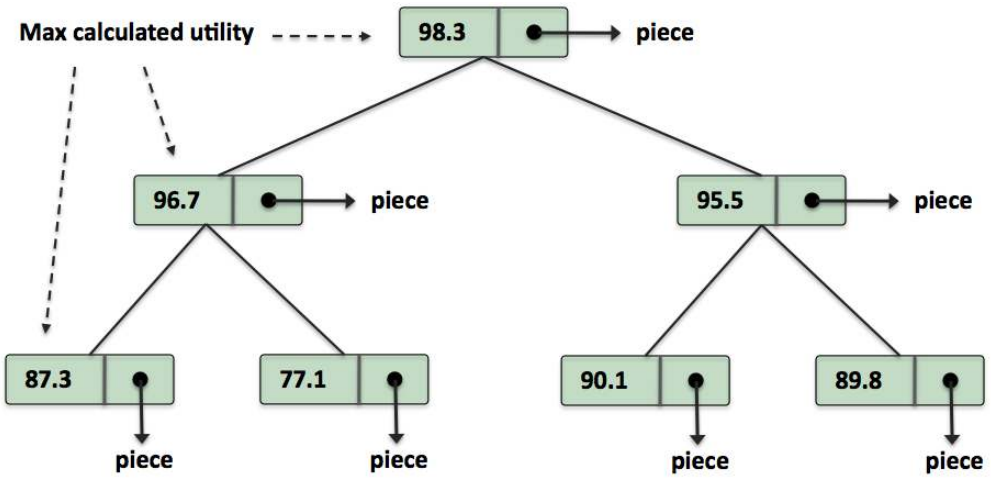

Figure 44: **Priority queue keyed with maximum utility scores**: The `Reflector` uses a balanced priority queue based on a max utility score to determine which pieces could benefit the most from further refinement. Each node in the tree keeps a pointer to the piece that generated the maximum utility key.

The algorithm terminates when either the top piece in the priority queue contains only a single IvP domain point, or when auto-peak refinement has generated a total of new pieces that exceeds a specified optional limit. This is checked in steps 2-3 in Listing 3. When a piece is selected for further refinement, it is split along the longest edge creating two pieces (one new piece) regardless of the number of edges or dimensions. Linear regression is performed on the two pieces and they are added back to the IvP function and the priority queue. Typically, but not necessarily, the piece with the maximum value in the priority queue is a result of the most recent refinement.

12.5.3 Invoking the Auto-Peak Algorithm in the Reflector

Use of the auto-peak tool is done using the `auto_peak` and `auto_peak_max_pcs` parameters in the Reflector `setParam()` function first mentioned in Section 11.4. The `auto_peak` parameter simply turns the tool on or off, with true or false. The `auto_peak_max_pcs` parameter sets an upper limit on the number of new pieces introduced to an IvP function during the auto-peak phase. The following shows an example usage:

```
reflector.setParam("auto_peak", "true")
reflector.setParam("auto_peak_max_pcs", 100)
reflector.create(1000);
```

The upper limit on pieces is typically not needed, and the default value is no limit. The algorithm tends to reach termination quickly because the piece with the maximum point tends to always be at the top of the priority queue, and in cases where the top ranked piece does not contain the maximum point, this is resolved quickly as the piece is split. Nevertheless, this upper limit is available for the conservative user. It is worth noting that, for underlying functions where the maximum value is part of a large plateau, the auto-peak tool is likely to have little benefit.

13 An Implementation Example - the SimpleWaypoint Behavior

In this section an example IvP behavior is presented. It is a simplified waypoint behavior version of the waypoint behavior in the standard suite of behaviors distributed with the MOOS-IvP public software bundle. The class name for this behavior is `BHV_SimpleWaypoint`. This behavior is distributed in the moos-ivp-extend repository and should build out of the box. After going through the class itself, later in this section example missions, also distributed with moos-ivp-extend, for running the behavior are discussed.

13.1 The SimpleWaypoint Behavior Class Definition

The SimpleWaypoint behavior is configured with four parameters: a single waypoint given in terms of local x and y coordinates, a transit speed in meters per second, and a radius in meters around the destination point within which the vehicle will be declared to have arrived at its waypoint. The behavior, at every iteration of the helm loop, notes the vehicle's own position in x and y local coordinates. The idea is shown in Figure 45.

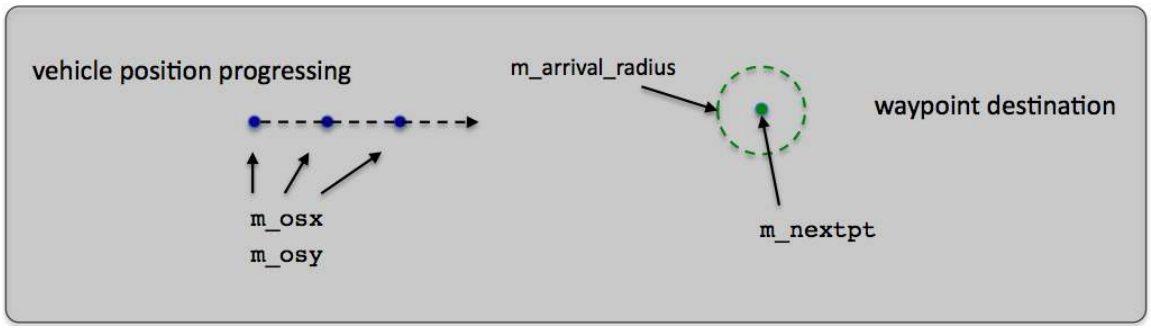

Figure 45: **The SimpleWaypoint behavior**: The SimpleWaypoint behavior works with a single waypoint. The location of the waypoint is stored in the local variable `m_nextpt` and is set during behavior configuration. The local variables `m_osx` and `m_osy` reflect the current vehicle (ownship) position updated at every helm iteration. The `m_arrival_radius` determines how close the vehicle needs to be from the waypoint destination before declaring completion.

The `BHV_SimpleWaypoint` class definition is given below in Listing 1. Note that it is declared to be a subclass of the `IvPBehavior` superclass on line 8. The three helm-invoked overloadable functions are declared on lines 13-15. The constructor is defined to take an `IvPDomain` as an argument. The helm will instantiate each behavior with the same helm-configured domain as an argument to a behavior constructor.

Listing 13.1: BHV_SimpleWaypoint.h - the class definition for the "simple waypoint" behavior.

```
1  #ifndef BHV_SIMPLE_WAYPOINT_HEADER
2  #define BHV_SIMPLE_WAYPOINT_HEADER
3
4  #include <string>
5  #include "IvPBehavior.h"
6  #include "XYPoint.h"
7
8  class BHV_SimpleWaypoint : public IvPBehavior {
```

```
 9  public:
10    BHV_SimpleWaypoint(IvPDomain);
11    ~BHV_SimpleWaypoint() {};
12
13    bool         setParam(std::string, std::string);
14    void         onIdleState();
15    IvPFunction* onRunState();
16
17  protected:
18    void         postViewPoint(bool viewable=true);
19    IvPFunction* buildFunctionWithZAIC();
20    IvPFunction* buildFunctionWithReflector();
21
22  protected: // Configuration parameters
23    double       m_arrival_radius;
24    XYPoint      m_nextpt;
25    double       m_desired_speed;
26    std::string  m_ipf_type;
27
28  protected: // State variables
29    double   m_osx;
30    double   m_osy;
31  };
32
33  extern "C" {
34    IvPBehavior * createBehavior(std::string name, IvPDomain domain)
35    {return new BHV_SimpleWaypoint(domain);}
36  }
37  #endif
```

The two configuration parameters depicted in Figure 45, the waypoint and arrival radius, are declared on lines 23-124. The two remaining configuration parameters, `"speed"` and `"ipf_type"` are on the following two lines. The former sets the ideal speed for waypoint transiting, and the latter indicates the type of IvP function to be generated. Two different ways of generating an IvP function are implemented in this behavior to demonstrate two different tools. The last part, lines 33-36 are the hooks needed for each behavior class to implement the dynamic loading of behaviors into the helm. These lines are therefore not present for behaviors compiled into the IvP helm. These lines are very pertinent to the discussion of extending" the helm.

13.2 The SimpleWaypoint Behavior Class Implementation

The class implementation is given in Listings 2-7 below.

13.2.1 The SimpleWaypoint Behavior Constructor

The first part contains the class constructor in lines 17-34. On line 18, a call to the base-class constructor is made with the given domain. A default for the behavior name is also set on line 20. On line 21, the behavior declares that the domain over which it will produce an IvP function is comprised of both the `course` and `speed` variables. If the domain given to the behavior by the helm in the constructor does not have either of these variables, a null IvP domain will result in line 21. A null domain will make the behavior thereafter not capable of running, and is considered a fatal error, prompting the helm to post all-stop output values. This is purposely drastic. Configuring the behaviors in a vehicle mission where one of the behaviors is not runnable is worthy of stopping the helm and addressing the problem. Since this condition is checked for on all behaviors on each helm

iteration, this problem would always reveal itself at launch time, never during a mission, regardless of any dynamic behavior configurations during a mission.

On lines 25-27, default values for class member variables representing key behavior parameters are set in the constructor. In lines 30-31, class member variables representing behavior state variables are initialized. The grouping of member variables into two sets, one that represent parameter configurations and the other that otherwise represent behavior state maintained during operation, is merely a convention that has provided clarity in practice.

Listing 13.2: BHV_SimpleWaypoint.cpp - The SimpleWaypoint Behavior Constructor.

```
 1  #include <cstdlib>
 2  #include <math.h>
 3  #include "BHV_SimpleWaypoint.h"
 4  #include "MBUtils.h"
 5  #include "AngleUtils.h"
 6  #include "BuildUtils.h"
 7  #include "ZAIC_PEAK.h"
 8  #include "OF_Coupler.h"
 9  #include "OF_Reflector.h"
10  #include "AOF_SimpleWaypoint.h"
11
12  using namespace std;
13
14  //-----------------------------------------------------------
15  // Procedure: Constructor
16
17  BHV_SimpleWaypoint::BHV_SimpleWaypoint(IvPDomain gdomain) :
18    IvPBehavior(gdomain)
19  {
20    IvPBehavior::setParam("name", "simple_waypoint");
21    m_domain = subDomain(m_domain, "course,speed");
22
23    // All distances are in meters, all speed in meters per second
24    // Default values for configuration parameters
25    m_desired_speed  = 0;
26    m_arrival_radius = 10;
27    m_ipf_type       = "zaic";
28
29    // Default values for behavior state variables
30    m_osx  = 0;
31    m_osy  = 0;
32
33    addInfoVars("NAV_X, NAV_Y");
34  }
35
```

Finally, on line 33, the behavior declares two variables, `NAV_X` and `NAV_Y`, representing vehicle ownship position. The IvP helm, containing this behavior, will need to register for these to variables on the behavior's behalf. This is the hook where the behavior tells the helm what it needs from the MOOSDB. It is from these two variables that the behavior will populate its variables `m_osx` and `m_osy` representing the current vehicle position.

13.2.2 The SimpleWaypoint Behavior `setParam()` Function

In Listing 3 below, a key overloadable behavior function is implemented, the `setParam()` function, in lines 39-69. This function handles the configuration of the behavior for its five parameters, `"ptx"`,

"pty", "speed", "radius", and "ipf_type". An example configuration for this behavior is given in Listing 8. Behavior parameters defined at the IvPBehavior superclass level, such as name, condition, endflag, etc., are handled in the setParam() function of the superclass. The helm, when it handles a behavior parameter from a *.bhv file, first attempts to handle the parameter at the superclass level. If the IvPBehavior::setParam() function returns false, the helm passes the parameter-value pair to the behavior's locally implemented version of setParam().

Listing 13.3: BHV_SimpleWaypoint.cpp - The setParam() function.

```
36 //-----------------------------------------------------------
37 // Procedure: setParam - handle behavior configuration parameters
38
39 bool BHV_SimpleWaypoint::setParam(string param, string val)
40 {
41   // Convert the parameter to lower case for more general matching
42   param = tolower(param);
43
44   double double_val = atof(val.c_str());
45   if((param == "ptx")  && (isNumber(val))) {
46     m_nextpt.set_vx(double_val);
47     return(true);
48   }
49   else if((param == "pty") && (isNumber(val))) {
50     m_nextpt.set_vy(double_val);
51     return(true);
52   }
53   else if((param == "speed") && (double_val > 0) && (isNumber(val))) {
54     m_desired_speed = double_val;
55     return(true);
56   }
57   else if((param == "radius") && (double_val > 0) && (isNumber(val))) {
58     m_arrival_radius = double_val;
59     return(true);
60   }
61   else if(param == "ipf_type") {
62     val = tolower(val);
63     if((val == "zaic") || (val == "reflector")) {
64       m_ipf_type = val;
65       return(true);
66     }
67   }
68   return(false);
69 }
70
```

A fair amount of error checking is done for parameter. For example, in setting the "speed" parameter, the string value is checked to ensure that is both numerical and larger than zero. Solid error checking implemented in this function is a very good idea that will save headaches down the road. This function should only return true if it has been passed a proper parameter-value pair. Another common practice is to perform a case insensitive parameter match, e.g., "pty" and "PTY" are both allowable configurations. This is done by converting the string representing the parameter to lower case in line 42. In this case, the tolower() function is defined in a local utility toolbox.

13.2.3 The SimpleWaypoint onIdleState() and postViewPoint() Functions

The onIdleState() function, lines 74-77, is only executed when the behavior is in the *idle* state, i.e., not in the *running* state. See Sections 7.4 and 7.6 for more on behavior states. In this behavior, the only task executed in the onIdleState() function is to publish a waypoint marker in the form of the MOOS variable VIEW_POINT.

Listing 13.4: BHV_SimpleWaypoint.cpp -The onIdleState() and postViewPoint() functions.

```
71  //-----------------------------------------------------------
72  // Procedure: onIdleState
73
74  void BHV_SimpleWaypoint::onIdleState()
75  {
76    postViewPoint(false);
77  }
78
79  //-----------------------------------------------------------
80  // Procedure: postViewPoint
81
82  void BHV_SimpleWaypoint::postViewPoint(bool viewable)
83  {
84    m_nextpt.set_label(m_us_name + "'s next waypoint");
85    m_nextpt.set_type("waypoint");
86    m_nextpt.set_source(m_descriptor);
87
88    string point_spec;
89    if(viewable)
90      point_spec = m_nextpt.get_spec("active=true");
91    else
92      point_spec = m_nextpt.get_spec("active=false");
93    postMessage("VIEW_POINT", point_spec);
94  }
95
```

An example produced by this would be:

```
VIEW_POINT = "active,false:label,alder's next waypoint:
             type,waypoint:source,waypt_return:0,0,0"
```

In this case, due to the "active,false" component, the posting of this variable would serve to "erase" similar postings to this variable made in the onRunState() function described next. For more on how VIEW_POINT is consumed, see the documentation for the pMarineViewer.

13.2.4 The SimpleWaypoint Behavior onRunState() Function

Implementation of the onRunState() function is where the primary unique operation of the behavior is implemented. For the SimpleWaypoint behavior, the full function is in Listing 5 below. It is implemented in four parts:

- Part 1: Get the vehicle position from the information buffer.
- Part 2: Determine if the waypoint has been reached and possibly enter *complete* mode.
- Part 3: Build a status message regarding the waypoint for third party viewers.

- Part 4: Build an IvP function with either the ZAIC or Reflector tool.

In the first part, lines 101-109, information from the information buffer is retrieved regarding the vehicle's own position. This is done with the `getBufferDoubleVal()` function described in Section 7.5.1. In this behavior the result of the query to the buffer is stored in the `ok1` and `ok2` variables and subsequently checked and handled in lines 106-109. In this behavior, if essential information like the vehicle's own position is missing, a warning is posted (line 107) and the `onRunState()` function returns without producing an objective function (line 108). In such a case the behavior would be considered to be in the running state, but not the active state for the present iteration.

A fair point to raise regarding Part 1 is the possibility that the vehicle's position information is in the buffer but has become so old that it no longer reflects the vehicle's true current position. In other words, what if the navigation module on board the vehicle has somehow shut down? First, in most situations with a vehicle implementing the backseat/front-seat driver architecture, a heartbeat monitor for the navigation system is typically put in place at the larger autonomy system level and an all-stop would be invoked overriding the helm. However, for the sake of having some fail-safe redundancy *within* the helm to handle this situation, the `no_starve` parameter could be used (Section 7.2.1) for this behavior, or any behavior since it is defined at the `IvPBehavior` superclass level. An example of it's usage is shown in Listing 8, setting a `no_starve` threshold of 3 seconds for `NAV_X` and `NAV_Y`.

Listing 13.5: BHV_SimpleWaypoint.cpp - the `onRunState()` implementation.

```
 96  //-----------------------------------------------------------
 97  // Procedure: onRunState
 98
 99  IvPFunction *BHV_SimpleWaypoint::onRunState()
100  {
101    // Part 1: Get vehicle position from InfoBuffer and post a
102    // warning if problem is encountered
103    bool ok1, ok2;
104    m_osx = getBufferDoubleVal("NAV_X", ok1);
105    m_osy = getBufferDoubleVal("NAV_Y", ok2);
106    if(!ok1 || !ok2) {
107      postWMessage("No ownship X/Y info in info_buffer.");
108      return(0);
109    }
110
111    // Part 2: Determine if the vehicle has reached the destination
112    // point and if so, declare completion.
113    double dist = hypot((m_nextpt.x()-m_osx), (m_nextpt.y()-m_osy));
114    if(dist <= m_arrival_radius) {
115      setComplete();
116      postViewPoint(false);
117      return(0);
118    }
119
120    // Part 3: Post the waypoint as a string for consumption by
121    // a viewer application.
122    postViewPoint(true);
123
124    // Part 4: Build the IvP function with either the ZAIC tool
125    // or the Reflector tool.
126    IvPFunction *ipf = 0;
127    if(m_ipf_type == "zaic")
```

```
128       ipf = buildFunctionWithZAIC();
129     else
130       ipf = buildFunctionWithReflector();
131     if(ipf == 0)
132       postWMessage("Problem Creating the IvP Function");
133
134     return(ipf);
135   }
136
```

In Part 2 of the `onRunState()` function, in lines 111-118 of Listing 5, the determination of waypoint arrival is made. This is just a simple comparison between the current distance of the vehicle and the waypoint to the configured arrival radius in the parameter `m_arrival_radius`. If a determination of arrival is made, the behavior calls the `setComplete()` function. This function is defined at the behavior superclass level and was described in detail in Section 7.5.1. The invocation of this function will put the behavior in the group of *completed* behaviors on the next helm iteration. In the current iteration this behavior would be considered in the *running* state, but not the *active* state since the `onRunState()` function returns (line 117) without generating an objective function. See Section 6.5.3 for more on behavior run states.

In Part 3, the behavior generates a visual artifact for consumption by a viewer, for rendering the waypoint the behavior is using as its destination. This point is shown in Figures 50 and 51 with the label `"Alder's next waypoint"`. An example produced by this would be:

```
VIEW_POINT = "label=alder's next waypoint,type=waypoint,source=transit,x=60,y=-40"
```

Compare this to the value for `VIEW_POINT` generated in the `onIdleState()` function described in Section 13.2.3. This variable-value pair is generated by the behavior for posting on each invocation of the `onRunState()` even though the value posted does not generally change between iterations. The posting of the variable-value pair is done with the `postMessage()` function, described in Section 7.5.1. The invocation of `postMessage()` will result in an actual post to the MOOSDB only if the value of string posted changes. Successive duplicate postings are filtered out by the Duplication Filter. The Duplication Filter was described in Section 5.8 where alternative functions for overriding the filter are given to accommodate other situations. The `postMessage()` function was discussed in Section 7.5.1.

In Part 4 of the `onRunState()` function, in lines 124-135 of Listing 5, an IvP function is generated over a domain of heading and speed choices to reflect the goal of reaching a waypoint given a current vehicle position. For the purposes of providing an example usage of the IvPBuild Toolbox, the behavior is implemented to produced an IvP function using two different methods of the toolbox. The SimpleWaypoint behavior can be configured with the `"ipf_type"` parameter, as shown in Listing 3, to accept either the configuration of `"zaic"` or `"reflector"`. In the example mission, mission "Alder" described below in Section 13.3, the helm is configured in Listing 8 to use the ZAIC tool on the outbound transit trip, and the Reflector tool on the return trip. The implementation of the `onRunState()` function merely checks the type of IvP function desired and makes the appropriate call to either the `buildFunctionWithZAIC()` function or the `buildFunctionWithReflector()` function. These two functions are described next.

13.2.5 The SimpleWaypoint Behavior `buildFunctionWithZAIC()` Function

When the SimpleWaypoint behavior is configured to generate an IvP function with the ZAIC tool, it invokes the function `buildFunctionWithZAIC()`, shown below in Listing 6. Of the three ZAIC tools described in Section 10, the `ZAIC_PEAK` is used in this behavior. It is used to generate two one-variable IvP functions. The first function is defined over the `speed` decision variable in lines 142-151. The second function is defined over `heading` decision variable. The functions are shown in Figure 46.

Listing 13.6: BHV_SimpleWaypoint.cpp - the `buildFunctionWithZAIC()` implementation.

```
137 //-----------------------------------------------------------
138 // Procedure: buildFunctionWithZAIC
139
140 IvPFunction *BHV_SimpleWaypoint::buildFunctionWithZAIC()
141 {
142   ZAIC_PEAK spd_zaic(m_domain, "speed");
143   spd_zaic.setSummit(m_desired_speed);
144   spd_zaic.setPeakWidth(0.5);
145   spd_zaic.setBaseWidth(1.0);
146   spd_zaic.setSummitDelta(0.8);
147   if(spd_zaic.stateOK() == false) {
148     string warnings = "Speed ZAIC problems " + spd_zaic.getWarnings();
149     postWMessage(warnings);
150     return(0);
151   }
152
153   double rel_ang_to_wpt = relAng(m_osx, m_osy, m_nextpt.x(), m_nextpt.y());
154   ZAIC_PEAK crs_zaic(m_domain, "course");
155   crs_zaic.setSummit(rel_ang_to_wpt);
156   crs_zaic.setPeakWidth(0);
157   crs_zaic.setBaseWidth(180.0);
158   crs_zaic.setSummitDelta(0);
159   crs_zaic.setValueWrap(true);
160   if(crs_zaic.stateOK() == false) {
161     string warnings = "Course ZAIC problems " + crs_zaic.getWarnings();
162     postWMessage(warnings);
163     return(0);
164   }
165
166   IvPFunction *spd_ipf = spd_zaic.extractIvPFunction();
167   IvPFunction *crs_ipf = crs_zaic.extractIvPFunction();
168
169   OF_Coupler coupler;
170   IvPFunction *ivp_function = coupler.couple(crs_ipf, spd_ipf, 50, 50);
171
172   return(ivp_function);
173 }
174
```

The first step in creating an `IvPFunction` with the `ZAIC_PEAK` tool is to create an instance of the `ZAIC_PEAK`, on line 142, passing it the `IvPDomain` used by the behavior and set in the behavior constructor in Listing 2. The ZAIC constructor is also passed the name of the one variable in the `IvPDomain` for which to create the `IvPFunction`. The `ZAIC_PEAK` parameters, set in lines 143-146, are described in detail in Section 10.1.1. In lines 147-151, a check is made to determine whether the `ZAIC_PEAK` instance has been configured properly. The `stateOK()` function returns `false` if there were any configuration problems, and the string returned by `getWarnings()` function on line 148

will provide insight into any configuration errors. The `postWMessage()` function on lines 149 and 162 will result in the helm posting to the MOOSDB the variable `BHV_WARNING` with the contents of string `warnings`. The `postWMessage()` function is discussed in Section 7.5.1 on page **??**.

The second one-variable function, defined over `course`, is created with a second `ZAIC_PEAK` instance in lines 153-164. First, the angle between the current vehicle position and the waypoint destination is calculated on line 153, with a call to the `relAng()` function, defined in one of the MOOS-IvP utility libraries. The creation and configuration of the `ZAIC_PEAK` instance proceeds much in same way as for the first one. In this case, the `valuewrap` parameter is set to `true` on line 159 to indicate that the domain values should "wrap around", that is, a course of 350 is 20 degrees separated from a course of 10 degrees, not separated by 340 degrees. The `valuewrap` parameter is discussed in Section 10.1.3 on page **??**.

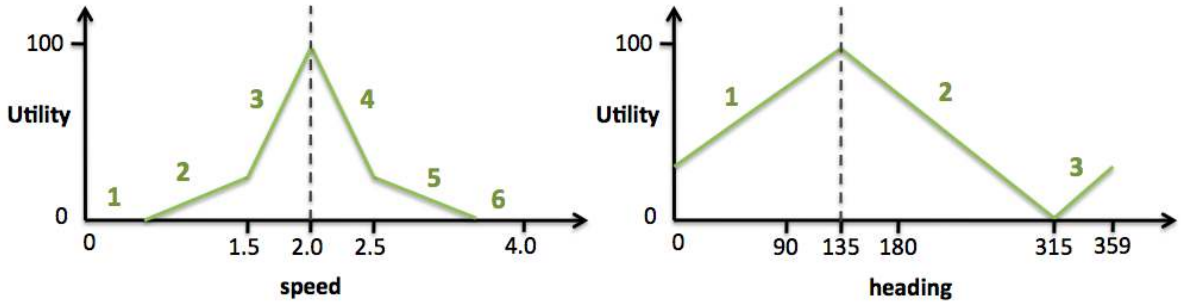

Figure 46: **IvP functions produced by the ZAIC_PEAK Tool**: The two functions produced by the SimpleWaypoint behavior by use of the `ZAIC_PEAK` tool would produce a function over `speed` with six pieces and a function over `heading` with three pieces.

When these two functions are coupled using the `Coupler` tool, lines 169-170, an IvP function over the coupled decision 2D space is created as shown in Figure 47 below. The Coupler tool is discussed in Section 9.2.3 on page 127.

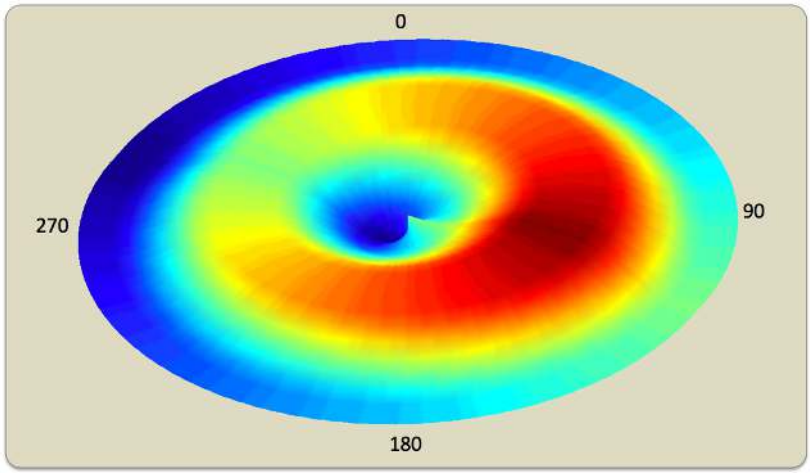

Figure 47: **An IvP function coupled from two one-variable functions**: This function is created by coupling the pair of one-variable functions of Figure 46 using the `OF_Coupler` tool. Decisions of increasing speed are represented by points on the function radially farther from the center.

The two one-variable functions are combined with an equal weight of 50 on line 170. The choice of relative weight has a distinct influence over the resulting function. In Figure 48 below, two IvP functions similarly generated, but with alternative weights are shown. The Coupler, by default, normalizes the combined function to the range of $[0, 100]$. This can be turned off, or normalized with a different range as described in Section 9.2.3. The range of $[0, 100]$ is a common range for functions returned by an IvP behavior to the IvP Solver.

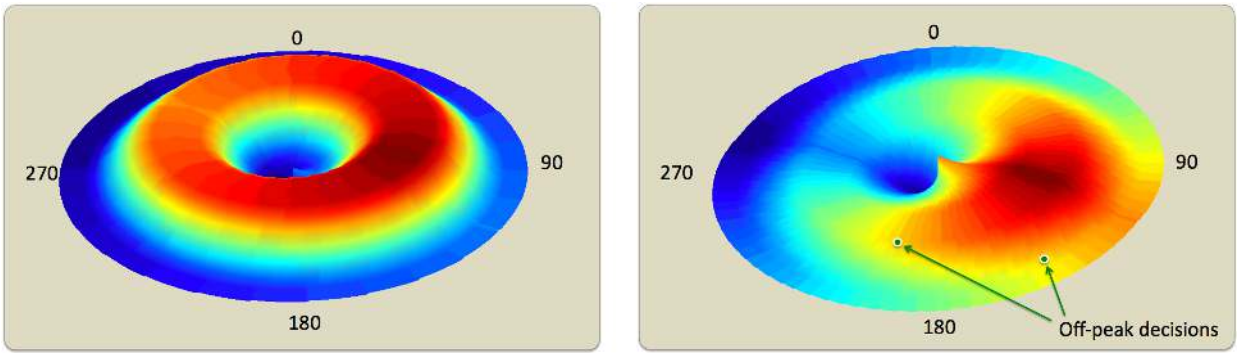

Figure 48: **Two IvP functions coupled from the same two one-variable functions**: These functions were created by coupling the pair of one-variable functions of Figure 46 using the `OF_Coupler` tool. They differ only in the relative weight applied to each function. The one on the left had a weight of 75 to the `speed` function and a weight of 25 to the `course` function. These weights are reversed on the right. The function on the right shows two off-peak decision points with equal utility rating. Decisions of increasing speed are represented by points on the function radially farther from the center.

In each IvP function in Figures 47 and 48, the location of the *peak* of the function is the same. When this behavior is the only active behavior, the path taken by the vehicle will be the same regardless of the weights chosen to combine the two one-variable functions with the Coupler. The

only thing that matters is the value of the `summit` parameters passed to the two ZAIC tools creating the two one-variable functions. The off-peak characteristics of the function begin to matter when the behavior is coordinated with functions from other behaviors. In the function on the right in Figure 48, two off-peak decisions with equal utility values are shown. One point represents a decision with the heading more toward the destination with a speed much higher than the desired speed, and the other decision represents a heading less toward the destination but with speed near the optimal speed. In the absence of mission metrics that clarify the relative utility of sub-optimal transit paths versus sub-optimal speeds, the construction of the off-peak shape of the objective function is typically a subjective decision of the behavior implementor.

13.2.6 The SimpleWaypoint Behavior `buildFunctionWithReflector()` Function

The Reflector tool can be used to generate objective functions that cannot otherwise be formed as the product of the coupling of two independent functions. This gives the behavior implementor more freedom to generate functions with off-peak characteristics more in-line with the goals of the behavior. The Reflector tool is described in detail in Sections 11 and 12.

The use of the Reflector tool in the SimpleWaypoint behavior is given in Listing 7 below. The Reflector generates an IvP function approximation of an underlying function, an instance of the `AOF_SimpleWaypoint` class. The underlying function and the IvP function are defined over the same domain. This domain is passed to the underlying function in its constructor (line 183). The underlying function is passed required parameters (lines 184-188) and initialized (line 189). If any part of the initialization fails, a null IvP function is returned (line 180, 196).

Listing 13.7: BHV_SimpleWaypoint.cpp - the `buildFunctionWithReflector()` implementation.

```
175  //-----------------------------------------------------------
176  // Procedure: buildFunctionWithReflector
177
178  IvPFunction *BHV_SimpleWaypoint::buildFunctionWithReflector()
179  {
180    IvPFunction *ivp_function = 0;
181
182    bool ok = true;
183    AOF_SimpleWaypoint aof_wpt(m_domain);
184    ok = ok && aof_wpt.setParam("desired_speed", m_desired_speed);
185    ok = ok && aof_wpt.setParam("osx", m_osx);
186    ok = ok && aof_wpt.setParam("osy", m_osy);
187    ok = ok && aof_wpt.setParam("ptx", m_nextpt.x());
188    ok = ok && aof_wpt.setParam("pty", m_nextpt.y());
189    ok = ok && aof_wpt.initialize();
190    if(ok) {
191      OF_Reflector reflector(&aof_wpt);
192      reflector.create(1000);
193      ivp_function = reflector.extractIvPFunction();
194    }
195
196    return(ivp_function);
197  }
```

The Reflector tool does its work (lines 191-193) after it has been determined that a proper instance of the underlying function, `AOF_SimpleWaypoint`, has been created and initialized. The Reflector tool has several options for creating a piecewise defined IvP function, described later in Sections 11 and

12. The simplest method is to specify the number of pieces desired in the piecewise function, in this case 1000 pieces were requested, on line 192. After creation, the IvP function is extracted from the Reflector (line 193), and returned (line 196).

An example of the IvP function created with the Reflector is shown in Figure 49. The function represents a preference for maneuvers that bring the vehicle toward the waypoint at a desired speed. The values of off-peak areas are evaluated based on (a) the rate of closure, (b) the rate of detour, and (c) the deviation from the desired speed. For this function generated by the Reflector, and the particular underlying function, it is not possible to generate a function of equal form using the ZAIC tools.

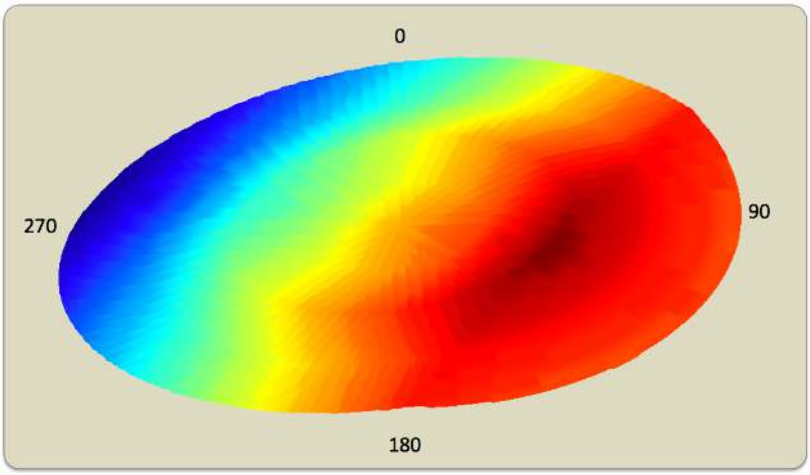

Figure 49: **An IvP function created with the Reflector tool**: The function represents a preference for maneuvers that bring the vehicle toward the waypoint at a desired speed. The values of off-peak areas are evaluated based on (a) the rate of closure, (b) the rate of detour, and (c) the deviation from the desired speed. Decisions of increasing speed are represented by points on the function radially farther from the center.

13.3 Running an Example Mission with the SimpleWaypoint Behavior

An example mission file, `alder.moos`, and behavior file, `alder.bhv`, have been configured to demonstrate the usage of the SimpleWaypoint behavior. The files may be found in the `moos-ivp-extend` tree. Refer back to Section 8.2 for information on obtaining this tree from the web. The example mission with the SimpleWaypoint behavior is called the Alder mission and is found by:

```
$ cd moos-ivp-extend/missions/alder
$  ls
$ README   alder.bhv   alder.moos
```

The behavior file is given in Listing 8 below. The SimpleWaypoint behavior is used twice. It is used to transit to a point and then to return to the starting point. The transiting use of the behavior is configured in lines 7-20, and the returning use of the behavior in lines 23-35.

Listing 13.8: The `alder.bhv` *file - For running the Alder example mission.*

```
1  //--------    FILE: alder.bhv   -------------
2
3  initialize   DEPLOY = false
5  initialize   RETURN = false
6
7  //----------------------------------------------
8  Behavior = BHV_SimpleWaypoint
9  {
10   name       = transit_to_point
11   pwt        = 100
12   condition  = RETURN = false
13   condition  = DEPLOY = true
14   endflag    = RETURN = true
15
16        speed = 2.0   // meters per second
17       radius = 8.0
18          ptx = 60
19          pty = -40
20     ipf_type = zaic
21 }
22
23 //----------------------------------------------
24 Behavior = BHV_SimpleWaypoint
25 {
26   name       = waypt_return
27   pwt        = 100
28   condition  = RETURN = true
29   condition  = DEPLOY = true
30
31        speed = 2.0
32       radius = 8.0
33          ptx = 0
34          pty = 0
35     ipf_type = reflector
36 }
```

Both behaviors are idle upon startup. Presumably the transit behavior is activated first by setting DEPLOY=true, and the second instance of the behavior is activated when the transit behavior completes and sets its endflag. The `alder.moos` file is not discussed here, but may be examined in the tree.

The example mission may be started by:

```
$ cd moos-ivp-extend/missions/alder
$ pAntler alder.moos
```

The pMarineViewer window should launch, and look similar to image in Figure 50. After clicking on the DEPLOY button in the lower right corner, the transiting instance of the SimpleWaypoint behavior becomes active and the vehicle begins to form a track to the waypoint as shown. Clicking on the DEPLOY button initiates a MOOS poke on the MOOSDB connected to both the pMarineViewer, and pHelmIvP application.

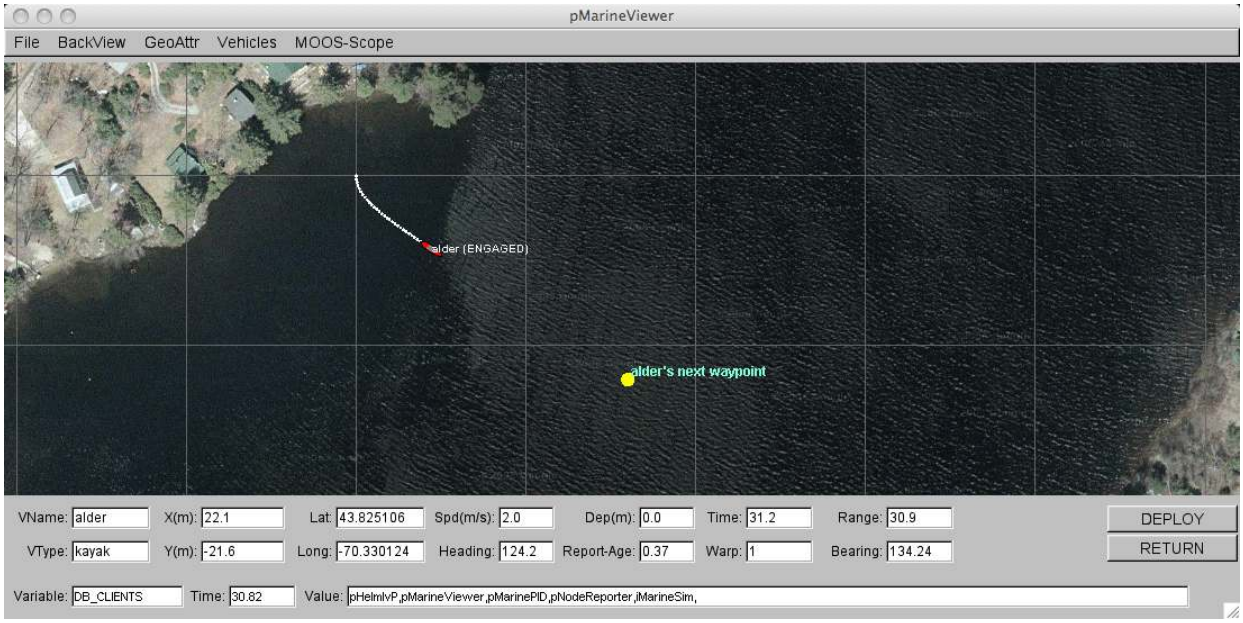

Figure 50: **The SimpleWaypoint behavior in action**: After the user clicks on the `DEPLOY` button, the condition for the transiting SimpleWaypoint behavior is satisfied (line 13 in Listing 8).

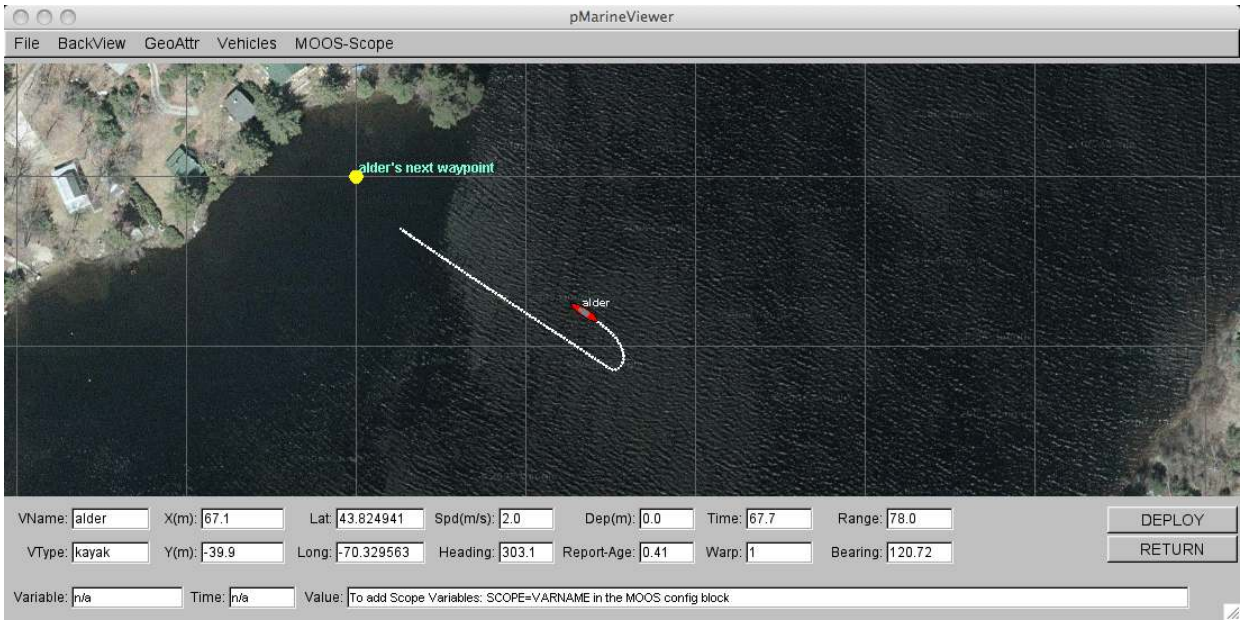

Figure 51: **The SimpleWaypoint behavior in action**: After the vehicle has reached the waypoint prescribed in the transiting instance of the SimpleWaypoint behavior, the second instance of the SimpleWaypoint behavior, returning to the start point, becomes active.

14 Behaviors of the IvP Helm

The following is a description of some single-vehicle behaviors currently written for the helm. The below is a guide for users of these behaviors. The topic of developing new behaviors is addressed separately. The setting of behavior parameters is the primary method for affecting the overall autonomy behavior in a vehicle. Parameters may also be dynamically altered once the mission has commenced. A parameter is set with a single line of the form:

```
parameter = value
```

The left-hand side, the parameter component, is case insensitive, while the value component is typically case sensitive. When the helm is launched, each behavior is created and the parameters are set. If a parameter setting in the behavior file references an unknown parameter, or if the value component fails a syntactic or semantic test, the line is noted and the helm ceases to launch.

Overview of Parameters Common to All Behaviors

The parameters below are common to all IvP behaviors, although they may have more relevance to some behaviors than others. They are defined in the IvPBehavior superclass of which all the behaviors described in this section are a subclass. More information on the functionality behind these parameters was given in Section 7.2.1.

Listing 14.1: Configuration Parameters Common to All IvP Behaviors.

Parameter	Description
activeflag:	A flag (MOOSVar,Data) pair, posted when in the active state.
condition:	A logical condition that must satisfied for the behavior to run.
duration:	Behavior duration, in seconds. Default is `"unlimited"`.
duration_idle_decay:	If true, clock will progress even if in the idle state.
duration_reset:	If limited duration, duration is reset when this variable is received.
duration_status:	MOOS variable informing of remaining duration, if duration is set.
endflag:	A flag (MOOSVar,Data) pair, posted when the behavior completes.
idleflag:	A flag (MOOSVar,Data) pair, posted when in the idle state.
inactiveflag:	A flag (MOOSVar,Data) pair, posted when in the inactive state.
name:	A unique identifier for the behavior instance.
nostarve:	If a given MOOSVar is stale by a given amount, an error is posted.
perpetual:	If true, a behavior may be reset after completion or duration timeout.
post_mapping:	A mapping to change the default MOOS variable output of a behavior.
priority, pwt:	Priority weight of the IvP function produced by behavior.
runflag:	A flag (MOOSVar,Data) pair, posted when in the running state.
templating:	Enable the behavior spec as a template for dynamic spawning.
updates:	A MOOSVar from which behavior parameter updates are received.

Configuring One Variable Objective Functions

Several behaviors use a common tool for constructing objective functions over a single decision variable. These behaviors have a similar interface for configuring this tool, and it is described

here to avoid redundancy. Examples of behaviors that use this tool are the Waypoint, Loiter, PeriodicSurface, ConstantDepth, ConstantSpeed, ConstantHeading, and Shadow behaviors. This tool is called the `ZAIC_PEAK` tool, and is describe in more detail in [**?**]. This tool is designed with the objective function form shown in Figure 52 in mind, where there is an identifiable preferred single decision choice (the `summit`) with maximum utility, and then a gradual drop in utility as the variable varies from the preferred choice.

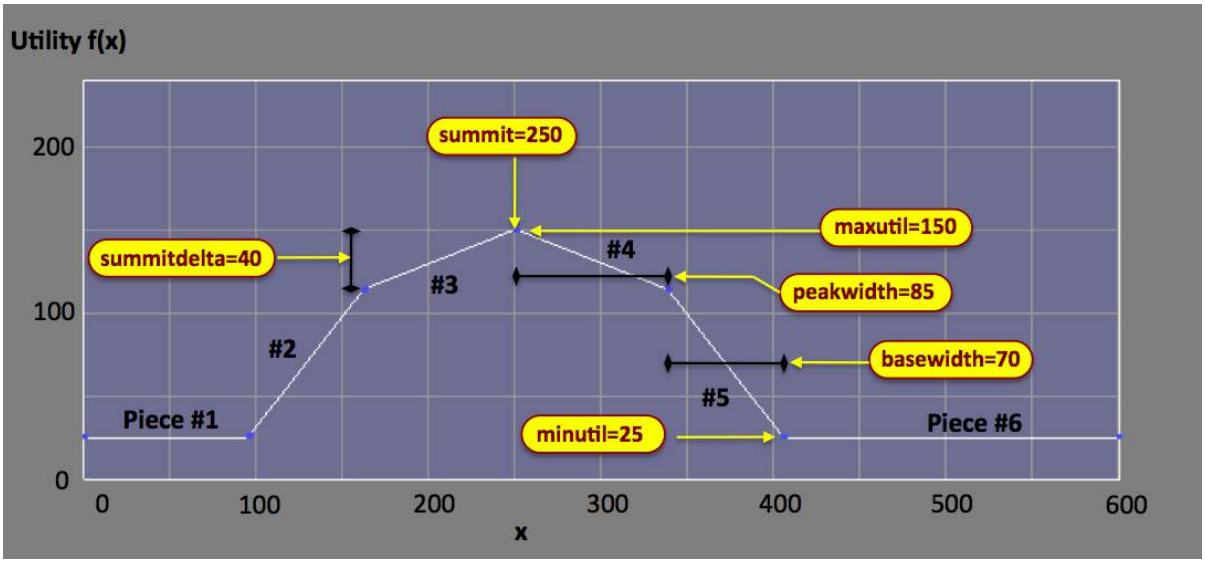

Figure 52: **The `ZAIC_PEAK` tool**: defines an IvP function over one variable defined by the four parameters shown here. In the case rendered here, the tool would create an IvP function with six pieces. The function rendered was created with `summit=180`, `peakwidth=85`, `basewidth=70`, `summitdelta=40`.

The form in which the utility drops is dependent on the settings of other parameters shown in the figure. The `summit`, `peakwidth`, and `basewidth` values are given in units native to the decision variable, while the `summitdelta`, `minutil`, and `maxutil` values are given in terms of units of utility. The latter two variables default to 0 and 100 respectively and are not typically exposed as configuration parameters in behaviors that use this tool, unlike the other four parameters.

15 Contact Related Behaviors of the IvP Helm

The section contains is a description of five behaviors currently written for the helm that reason about relative position to another vehicle or *contact*.

- The AvoidCollision behavior
- The AvdColregsV19 behavior
- The CutRange behavior
- The Shadow behavior
- The Trail behavior

Each behavior needs to be continuously updated with the position and trajectory of a given contact. The helm subscribes to the MOOS variable, `NODE_REPORT`, which may have content similar to:

```
NODE_REPORT= "NAME=alpha,TYPE=UUV,MOOSDB_TIME=39.01,TIME=1252348077.59,
              X=51.71,Y=-35.50,LAT=43.824981,LON=-70.329755,SPD=2.00,HDG=118.85,
              YAW=118.84754,DEPTH=4.63,LENGTH=3.8,MODE=SURVEYING"
```

This contact information is stored in the helm information buffer for any behavior that wishes to retrieve it on any iteration. Since each behavior type needs to reason about contact information, the common code for this is handled in superclass called `IvPContactBehavior`. So for these behaviors, the class hierarchy looks like:

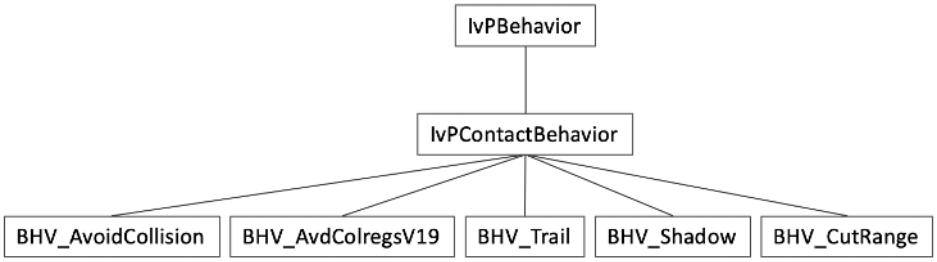

Figure 53: **IvP Contact Behaviors**: The contact-related behaviors share a common superclass to streamline the development of contact behaviors and provide a level consistency between behaviors.

15.1 Static Versus Dynamically Spawned Behaviors

A contact behavior may be configured in one of two ways. The simplest way is to reason about a particular contact with a name known at mission time. This is more appropriate for a behavior that needs to operate on one contact at a time, such as a trail or cut-range behavior. While it is technically possible to try to cut-range or trail multiple contacts simultaneously, and the IvP solver will support such an attempt, typically such behaviors are focused on a single contact. Configuration of the behavior simply names the contact

```
  contact = mothership
```

A second type of configuration is possible when several simultaneous instances of the same behavior are desired, as during collision avoidance with multiple contacts. In this case the behavior is configured to enable *templating*, and the contact name is left unspecified until a new contact emerges:

```
templating = spawn
contact = to_be_determined
```

A templating contact behavior works closely with a contact manager, either the pBasicContactMgr app or the newer version, pContactMgrV20. The contact manager is responsible for generating the MOOS posting events that trigger the spawning of a new behavior. As depicted in Figure 54 below, the contact manager applies a certain configurable criteria to a contact, and if it passes that criteria, it generates an alert to the helm, resulting in a spawned behavior.

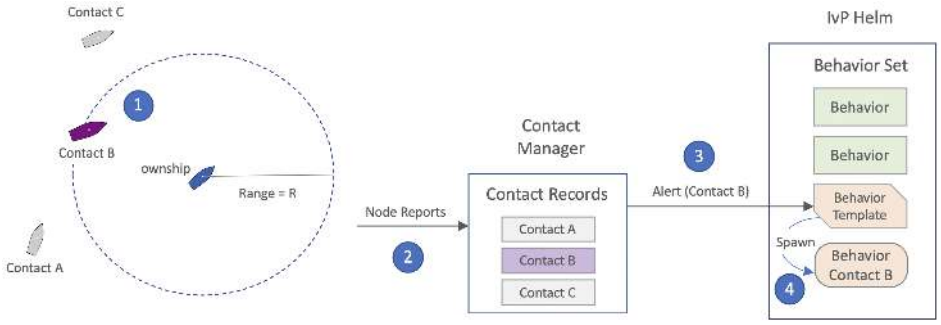

Figure 54: **Relationship between the contact manager and spawned contact-related behaviors**: (1) A contact closes range to ownship, crossing a range threshold. (2) The contact manager makes note and changes the internal record associated with the contact. (3) An alert it generated and received by the helm. (4) The helm spawns a new contact-related behavior dedicated to this contact.x

Typically the contact manager criteria for generating an alert is based on the *range* of the contact to ownship. However, the contact manager may also utilize additional filters, providing user configurable selectivity as to which contacts generate alerts resulting in behavior spawnings. These filters may be configured at the contact manager level, e.g., ignore all contacts of type sailboat. Or they may be configure at the behavior level, e.g., one collision avoidance behavior matches sailboats, and another collision avoidance behavior matches motorboats. This is covered in part in the documentation of pContactMgrV20, but portion that is accessible through behavior configuration is described next.

15.2 Exclusion Filters

An *exclusion filter* is tool for allowing the contact manager and/or the helm to treat distinct contacts differently, based on a certain property. *The contact manager has an exclusion filter, and all behaviors have an exclusion filter.* They can be safely ignored if one wishes to simply treat all contacts equally, but they can provide some powerful options when applied to a particular situation.

15.2.1 Types of Contact Exclusion Filters

Contact exclusion filters may be configured around four different types of properties:

- `name`: This is the most direct method but rather brittle. But if there is a particular named vehicle that you'd like to handle differently, it can be just called out by name.
- `type`: The vehicle type is part of the incoming `NODE_REPORT` message, e.g., UUV, USV and so on.
- `group`: The vehicle group is part of the incoming `NODE_REPORT` message, e.g., friendly, foe and so on.
- `region`: A convex polygon describing a region of the operation area.

Each of the above four properties can be configured as `match` or `ignore` filters. For example, `match_type=kayak` indicates that the contact must by of type `kayak`. Likewise `ignore_type=kayak` indicates that the contact must be of some type other than `kayak`. The same is true for `name`, `group`, and `region` properties. This makes eight possible configuration parameters available to all contact behaviors when templating is enabled:

- `match_group` and `ignore_group`
- `match_name` and `ignore_name`
- `match_type` and `ignore_type`
- `match_region` and `ignore_region`

Multiple entries for each property may be used. If for example multiple `match_name` configurations are used, a contact will pass the filter if it matches at least one. If multiple `ignore_name` configurations are used, a contact will pass the filter only if it has a name different from all.

The `region` property is a bit different from the other three properties. While a contact's `name`, `type`, and `group` properties tend not to change as a mission unfolds, the contact position does. When the `region` is used by the contact manager as a filter it applies only to that moment in time. If, for example, the contact was outside a `match_region` but then enters it, it will now pass the filter and an alert will be generated, provided all other filter and range criteria are met. Likewise if the contact manager generated an alert for a contact, but then moved out of a `match_region` or into an `ignore_region`, the previously generated alert will not somehow be retracted.

One last note about `region` filters: non-convex regions may be implemented by using two or more convex regions that "cover" the desired non-convex region. And consistent with the above discussion, the approximating convex regions may harmlessly overlap.

15.2.2 Configuring Exclusion Filters Globally or Locally

As shown in Figure 54 above, the contact manager is the gatekeeper for sending the helm alert messages that may trigger the spawning of the behavior. Users can configure the exclusion filters in one of two ways:

- Global configuration via the contact manager, `pContactMgrV20`, or
- Local configuration via individual behaviors in the helm.

- A third configuration method, dealing with behaviors *after* they have been spawned, is discussed in Section 15.2.4.

The former method is arguably more convenient but the filters apply to all behavior templates equally. If the contact manager is configured to ignore sailboats, there is no way to otherwise configure the helm to have a spawnable behavior that will handle sailboats. Maybe you really do want to just ignore all sailboats. But if instead you'd like to have one type of behavior ignore sailboats and another type of behavior ignore motorboats, then the filter needs to be configured at the behavior level instead.

Figure 55 below depicts a helm locally configured with two distinct behavior templates, with the filter configuration done in the helm configuration. In this way, one distinct behavior is used for Contact Abe, and another for Contact Ben. It may be that both are collision avoidance behaviors, but perhaps the safety region around Abe is larger than Ben. Likewise the filter could have involved the vehicle type, group or region.

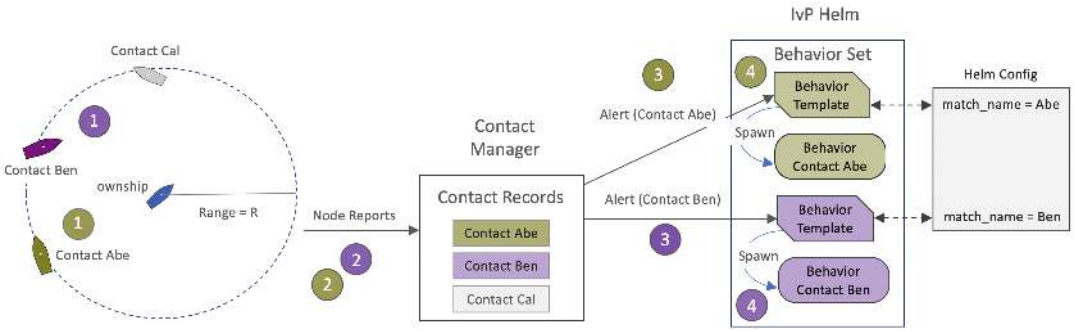

Figure 55: **Configuration of contact filter at the local behavior level**: Compare with Figure 54, the match filters are provided in the helm configuration rather than the contact manager configuration. This allows different behavior types or templates to work with different contacts. In the previous example in Figure 54 every matched contact was handled by the same behavior type.

Using both global and local exclusion filters is certainly allowable, but keep in mind that if a contact is excluded at the global contact manager level, it does not matter what configurations are specified at the local behavior level.

15.2.3 Enabling Strict Filtering

Filtering based on the contact `type` or `group` depends on this information being known about the contacts. Typically this is embedded in each incoming `NODE_REPORT`. In some cases, especially in sensor-based contact management, this information may not be known.

For *match* filtering, this is not an issue. This is because the spirit of match filtering is that, for example if we say that a contact must be of type motor-boat, and the contact type is unknown, it's pretty clear that condition is not satisfied and thus the contact with unknown type is filtered out.

With *ignore* filtering it is more ambiguous. This is because the spirit of ignore filtering is that, for example if we say that a contact cannot be of type motor-boat, and the contact type is unknown,

we might be inclined to say that it should not be filtered out. The user may however want to be strict about this and say that it *should* be filtered out since we don't know the contact type, and for all we know it could be a motor-boat.

To allow the user to have some control and be explicit about the above situation, filtering can be configured to be *strict* by setting `strict_ignore` to be `true` or `false`. The below set of examples hopefully makes this clear.

Contact Type	Ignore Group	Strict Ignore	Filtered Out
sailboat	sailboat	true or false	yes
sailboat	motorboat	true or false	no
unknown	motorboat	false	no
unknown	motorboat	true	yes

The `strict_ignore` parameter, as with other filtering parameters, is also used in configuring the contact manager, applied globally and overriding any local filter behavior configurations.

15.2.4 Failing an Exclusion Filter on Spawned Behavior

The primary use of exclusion filters is to affect which behaviors are spawned. However, the configured criteria is based on the vehicle properties of group, name, platform type and location, and these properties may change *after* a behavior is spawned. Normally a contact behavior does not complete, and exit, until it has gone out of range. The user may also opt to have the behavior exit if its properties change after spawning such that it no longer satisfies the original exclusion filter.

A contact behavior will accept the following three configuration parameters:

- `exit_on_filter_group`: If true, will apply the current known group name to an exclusion filter.
- `exit_on_filter_vtype`: If true, will apply the current known vehicle to an exclusion filter.
- `exit_on_filter_region`: If true, will apply the current known vehicle position to the region component of an exclusion filter.

The default for all parameters is false. A common example is when a detected change of group name or vehicle type is learned via communications or sensors. A vehicle may also move into a region where a behavior may be configured to be no longer concerned with a contact. Failing the exclusion filter in a spawned behavior will simply result in the exit/removal of the behavior from the helm, and deletion of the behavior from memory space.

Unlike the group, vehicle type and region components of an exclusion filter, the vehicle *name* component of the exclusion filter is not re-assesed after a behavior has spawned, since contact behaviors are essentially keyed by vehicle name.

While the above three parameters are supported by all contact behaviors, not all contact behaviors respect this additional feature. Currently (Aug 2021) only the collision avoidance behaviors (CPA based and COLREGS based) support this feature. Support in additional behaviors will be rolled in on future updates.

15.3 Contact Flags

Available after Release 19.8.1, *contact flags* are an additional way to configure behaviors to post configurable information upon events based on the relative position between ownship and the contact. Most other types of behavior flags contain simply a variable value pair. For example:

```
endflag = RETURN = true
```

Contact flags allow the user to check for expected results over the course of a mission. It also allows missions to be configured on contact related events such as posting a message to command-and-control or to another vessel when the relative position or range to a contact has been achieved.

15.3.1 Contact Flag Trigger Tags

With contact flags, a *trigger tag* is included with the flag configuration corresponding to certain events. For example, the `@cpa` trigger tag will ensure the configured flag will be posted at the time of an observed CPA between ownship and the contact:

```
cnflag = @cpa SEND_MESSAGE = true
```

The trigger tag is always at the beginning of the flag configuration with at least one white space between the tag and the rest of the flag configuration. The trigger tags only apply to contact flags, for contact behaviors, using the `cnflag` parameter.
Supported trigger flags:

Trigger Flag	Meaning
`@cpa`:	When the closes point of approach is observed
`@os_passes_cn`:	When ownship passes contact's beam
`@os_passes_cn_port`	When ownship passes contact's port beam
`@os_passes_cn_star`:	When ownship passes contact's starboard beam
`@cn_passes_os`	When ownship passes contact's beam
`@cn_passes_os_port`	When contact passes contact's port beam
`@cn_passes_os_star`	When contact passes contact's starboard beam
`@os_crosses_cn`	When ownship crosses contact's side
`@os_crosses_cn_bow`	When ownship crosses contact's side fore of contact
`@os_crosses_cn_stern`	When ownship crosses contact's side aft of contact
`@cn_crosses_os`	When contact crosses ownship's side
`@cn_crosses_os_bow`	When contact crosses ownship's side fore of ownship
`@cn_crosses_os_stern`	When contact crosses ownship's side aft of ownship

An example is given in Figure 56 below of an encounter between ownship and a contact. The relevant trigger tags are shown.

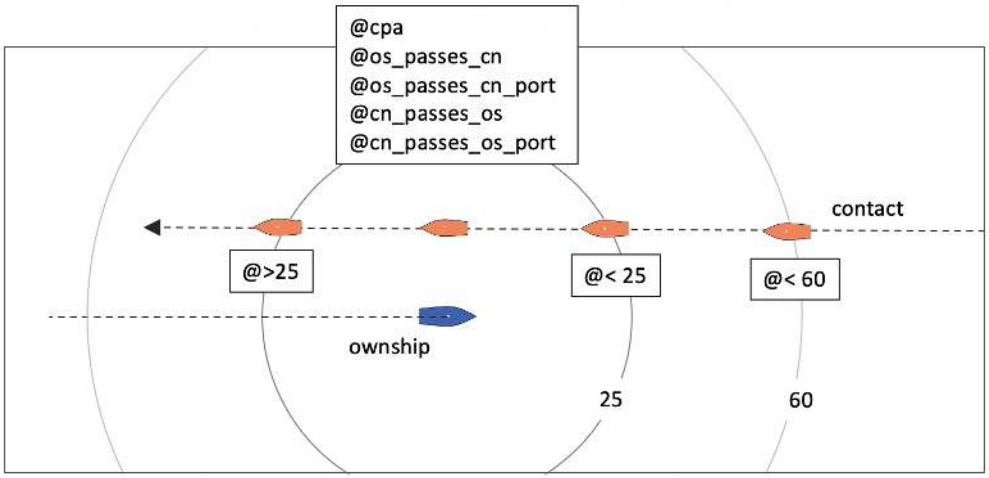

Figure 56: **An example encounter with trigger tags**: As a contact passes ownship various trigger tags become satisfied. The first two trigger tags are based purely on range and are triggered as the vehicle enters the range. The last trigger tag is triggered as the contact opens range to 60. As the two vehicles pass, several trigger tags are satisfied simultaneously.

15.3.2 Contact Flag Macros

Contact flags have a number of supported macros that may be expanded in any value component of a posting. For example, the range between ownship and the contact can be used in a `cnflag` posting:

```
cnflag = @cpa RANGE_TO_CONTACT = $[RANGE]
```

The value of `$[RANGE]` is determined at the moment the flag is triggered. Supported macros:

MACROS	Meaning
$[RANGE]	Range between ownship and contact
$[CN_NAME]	Name of the contact
$[CN_GROUP]	Name of the contact
$[CN_VTYPE]	Vehicle type of the contact
$[ROC]	Rate of Closure between ownship and the contact
$[OS_CN_REL_BNG]	Relative bearing of the contact to ownship
$[CN_OS_REL_BNG]	Relative bearing of ownship to the contact
$[BNG_RATE]	Bearing Rate
$[CN_SPD_IN_OS_POS]	Speed of contact in the direction of ownship position
$[OS_FORE_OF_CN]	true if ownship is currently fore of the contact
$[OS_AFT_OF_CN]	true if ownship is currently aft of the contact
$[OS_PORT_OF_CN]	true if ownship is currently on port side of the contact
$[OS_STAR_OF_CN]	true if ownship is currently on starboard side of the contact
$[CN_FORE_OF_OS]	true if the contact is currently fore of ownship
$[CN_AFT_OF_OS]	true if the contact is currently aft of ownship

$[CN_PORT_OF_OS]	true if the contact is currently on port side of ownship
$[CN_STAR_OF_OS]	true if the contact is currently on starboard side of ownship

15.4 Properties Common to All Contact Related Behaviors

Contact related behaviors are distinct from non contact related behaviors in that they share a fair amount of functionality in dealing with their contact of interest. Contact related behaviors are implemented as a subclass of the `IvPContactBehavior` class, which is a subclass of the `IvPBehavior` class. Much of the shared functionality of contact related behaviors is implemented in the former. The shared functionality includes several common configuration parameters, and mechanisms for reasoning about the closest point of approach (CPA) between the platform and contact for candidate maneuver considerations. These topics are discussed next, prior to the sections on the behaviors themselves.

15.4.1 Common Behavior Configuration Parameters

The following set of parameters are common to all the contact related behaviors:

Listing 15.1: Configuration Parameters Common to Contact Behaviors.

Parameter	Description
bearing_lines	If true, a visual artifict will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.
contact	Name or unique identifier of a contact to be avoided.
decay	Time interval during which extrapolated position slows to a halt.
exit_on_filter_group	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_vtype	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_region	It true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default it false.
extrapolate	If true, contact position is extrapolated from last position and trajectory.
ignore_group:	If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 15.2
ignore_name:	If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 15.2

ignore_region:	If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 15.2
ignore_type:	If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 15.2
match_group:	If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 15.2
match_name:	If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 15.2
match_region:	If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regins. Introduced after Release 19.8.1. Section 15.2
match_type:	If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 15.2
on_no_contact_ok	If false, a helm error is posted if no contact information exists. Applicable in the more rare case that a contact behavior is statically configured for a named contact. The default is true.
strict_ignore	If true, and if one of the ignore exclusion filter components is enabled, then an exclusion filter will fail if the contact report is missing information related to the filter. For example if the contact group information is unknown. The default is true.
time_on_leg	The time on leg, in seconds, used for calculating closest point of approach.

The contact parameter specifies the contact name or identifier. This name is used as a key by the behaviors for querying the contact position and trajectory. It must match the contact name received by the helm in an incoming NODE_REPORT message. The contact name may be specified at helm launch time, but it may also be specified at run time if the behavior is configured as a template for dynamic spawning. The latter is more common, for example, in a collision avoidance behavior where the name or ID of the contact is not known until a contact manager alerts the helm. See the Berta mission for an example of this usage.

The extrapolate and decay parameters are used to address the situation where a contact/node report has significant delays between updates. Extrapolation is enabled by setting the extrapolate parameter to `true`, which is the default. The behavior may be configured to have limited extrapolation by setting a decay time interval. The extrapolated position is based on the last known contact position, heading and speed. The speed used for calculations may begin decaying at the beginning of the decay interval and will have decayed to zero at the end of the decay interval. The default setting is `"decay = 15, 30"`, in seconds. The idea is shown in Figure 57.

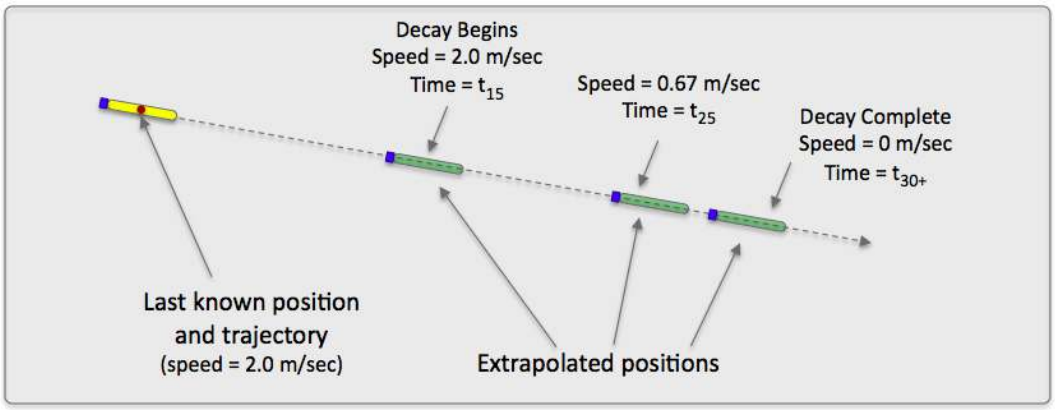

Figure 57: **Contact Extrapolations:** Contact related behaviors may use an extrapolated position of the contact to compensate for periods of no new reports for the contact. A decay period may be used to effectively halt the extrapolated contact position after some specified period of time. In the example in this figure, the decay window is [15, 30] seconds. After 30 seconds, the extrapolated position does not extend further.

The `on_no_contact_ok` parameter determines how the behavior should regard the situation where it is unable to find any information about a given contact. If this parameter is set to true, the default, then the behavior will post a warning, `BHV_WARNING` if no contact information is found. Otherwise the behavior will post an error with `BHV_ERROR`. In the latter case the helm may interpret this as request to halt the helm and come to zero speed and depth.

The `time_on_leg` parameter refers to the behavior's calculations of the closest point of approach (CPA) for candidate maneuver legs. A candidate maneuver leg is defined by a the heading, speed, and time-on-leg components. Longer time-on-leg settings tend to report deceivingly worrisome CPA distances even for contacts at a great distance, and lower time-on-leg settings tend to report deceivingly comfortable CPA distances even for vehicles at relatively low distances. The default setting for this parameter is 60 seconds.

16 A First Example with MOOS-IvP - the Alpha Mission

This section describes a simple mission using the helm. It is designed to run in simulation on a single machine. The mission configuraiton files for this example are distributed with the source code. Information on how to find these files and launch this mission are described below in Section 16.1. In this example the vehicle simply traverses a set of pre-defined given waypoints and returns back to the launch position. The user may return the vehicle any time before completing the waypoints, and may subsequently command the vehicle to resume the waypoints at any time. This example touches on the following issues:

- Launching a mission with a given mission (`.moos`) file and behavior (`.bhv`) file.
- Configuration of MOOS processes, including the IvP Helm, with a `.moos` file.
- Configuration of the IvP Helm (mission planning) with a `.bhv` file.
- Implementation of simple command and control with the IvP Helm.
- Interaction between MOOS processes and the helm during normal mission operation.

Here is a bit of what the Alpha mission should look like:

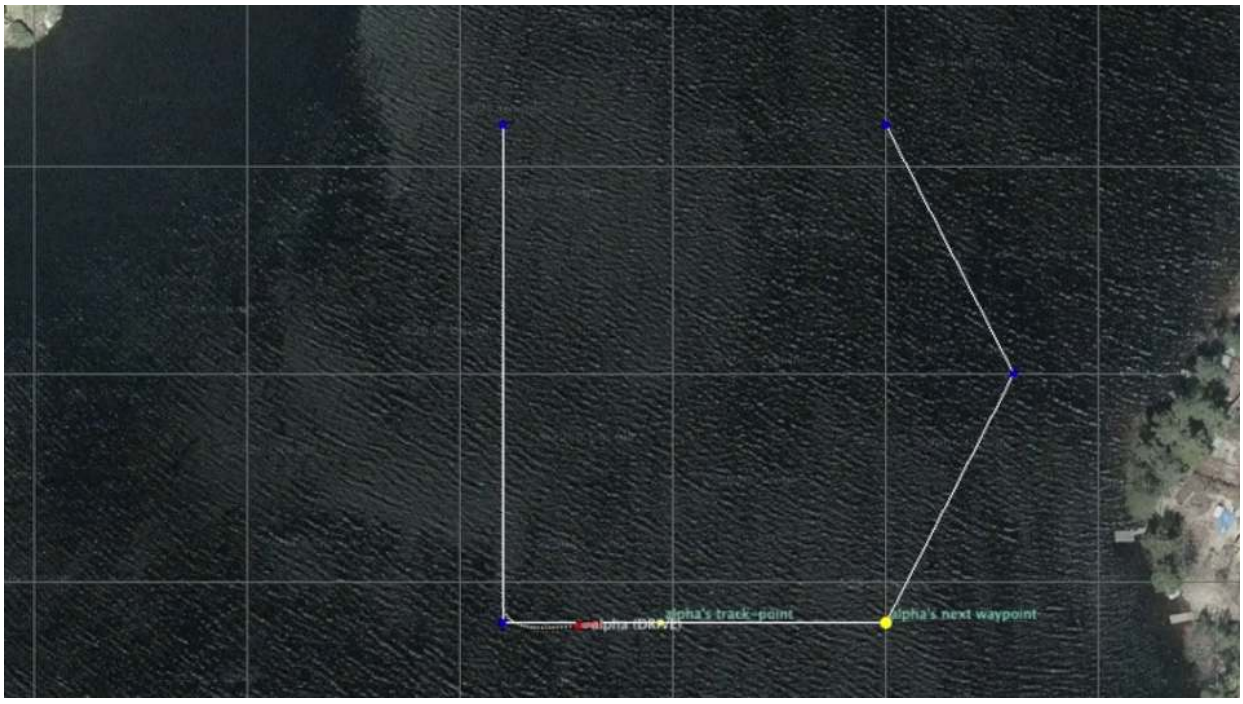

Figure 58: The Alpha mission.

video:(0:19): https://vimeo.com/84549446

16.1 Find and Launch the Alpha Example Mission

The example mission should be in the same directory tree containing the source code. There are two files - a MOOS file, also mission file or `.moos` file, and a behavior file or `.bhv` file:

```
moos-ivp/
   MOOS/
   ivp/
      missions/
         s1_alpha/
            alpha.moos    <---- The MOOS file
            alpha.bhv     <---- The Behavior file
```

To run this mission from a terminal window, simply change directories and launch:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ pAntler alpha.moos
```

After pAntler has launched each process, the pMarineViewer window should be open and look similar to that shown in Figure 59. After clicking the DEPLOY button in the lower right corner the vehicle should start to traverse the shown set of waypoints.

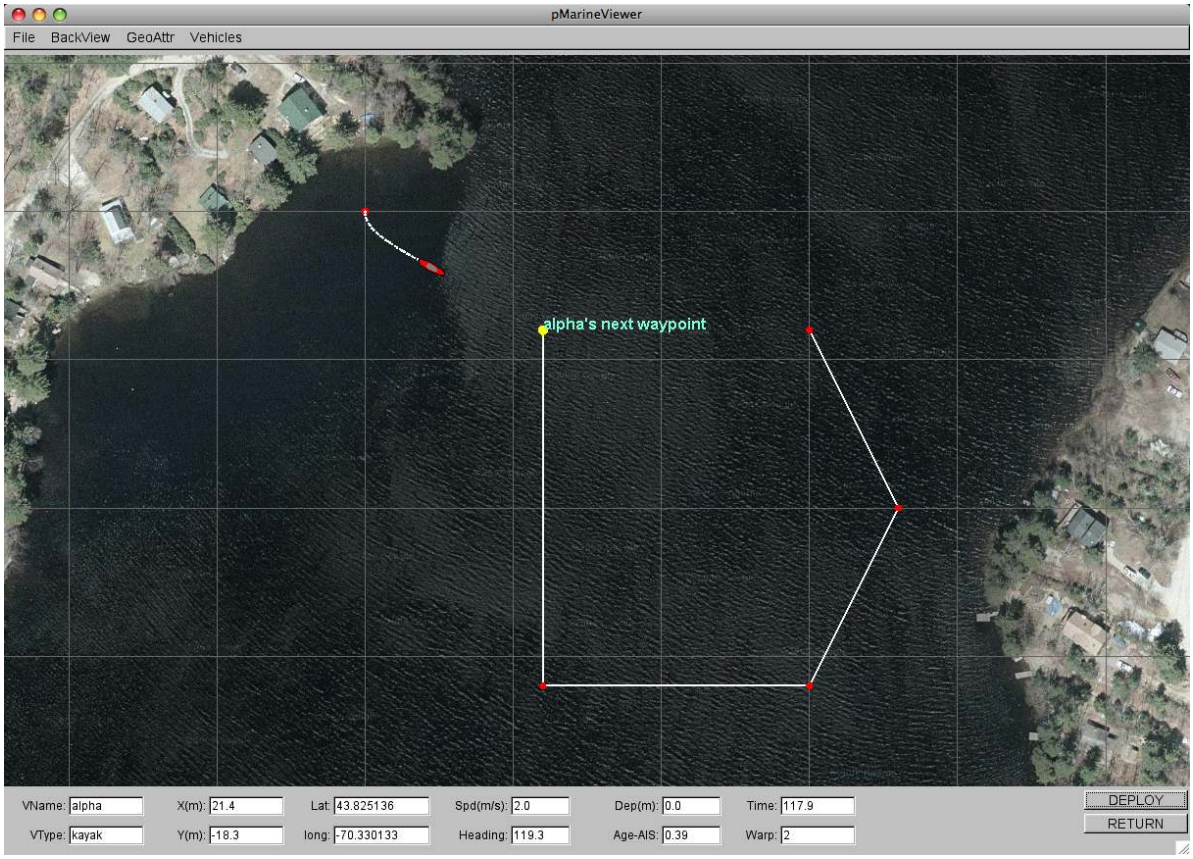

Figure 59: **The Alpha Example Mission - In the Surveying Mode**: A single vehicle is dispatched to traverse a set of waypoints and, upon completion, traverse to the waypoint (0,0) which is the launch point.

This mission will complete on its own with the vehicle returning to the launch point. Alternatively,

by hitting the RETURN button at any time before the points have been traverse, the vehicle will change course immediately to return to the launch point, as shown in Figure 60. When the vehicle is returning as in the figure, it can be re-deployed by hitting the DEPLOY button again.

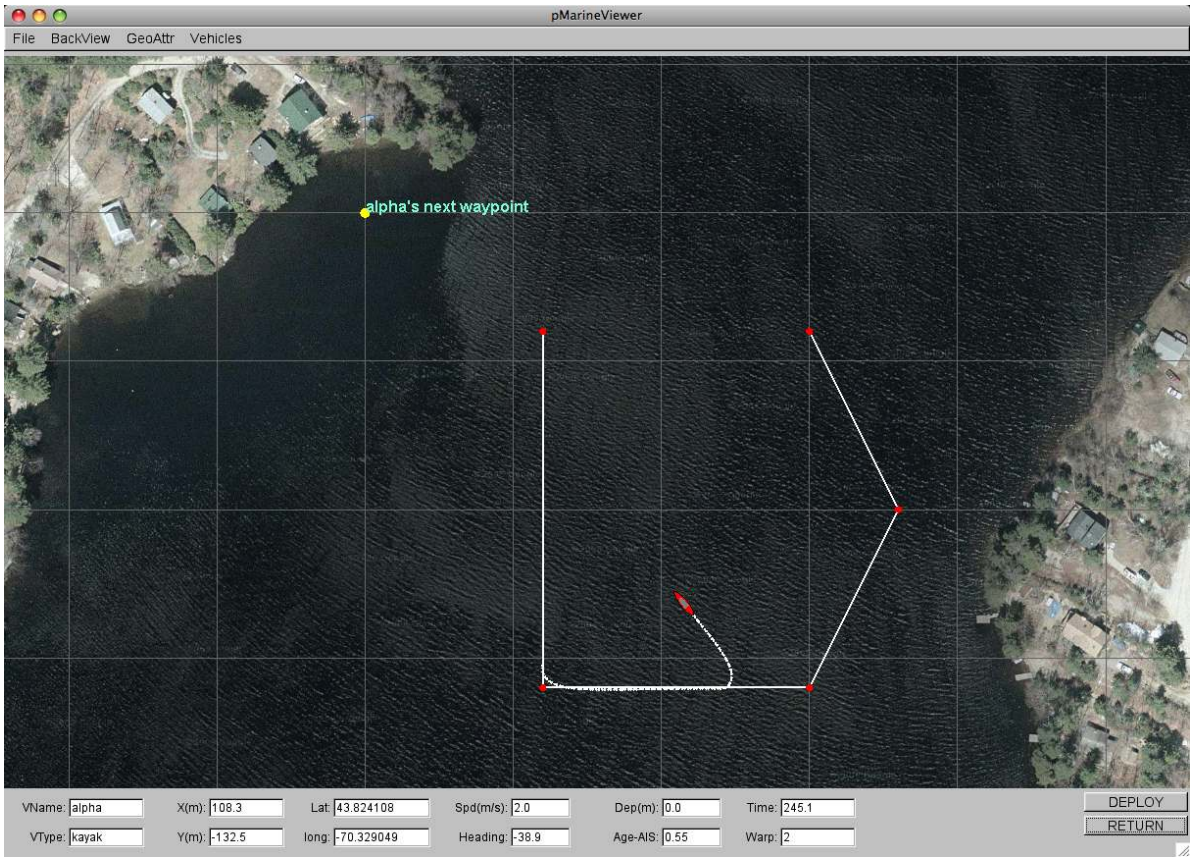

Figure 60: **The Alpha Example Mission - In the Returning Mode**': The vehicle can be commanded to return prior to the completion of its waypoints by the user clicking the RETURN button on the viewer.

The vehicle in this example is configured with two basic waypoint behaviors. Their configuration with respect to the points traversed and when each behavior is actively influencing the vehicle, is discussed next.

16.2 A Look at the Behavior File used in the Alpha Example Mission

The mission configuration of the helm behaviors is provided in a *behavior file*, and the complete behavior file for the example mission is shown in Listing 1. Behaviors are configured in blocks of parameter-value pairs - for example lines 6-17 configure the waypoint behavior with the five waypoints shown in the previous two figures. This is discussed in more detail in Section **??**.

Listing 16.1: The behavior file for the Alpha example.

```
1  initialize   DEPLOY = false
2  initialize   RETURN = false
```

```
 3
 4  //---------------------------------------------
 5  Behavior = BHV_Waypoint
 6  {
 7    name      = waypt_survey
 8    pwt       = 100
 9    condition = RETURN = false
10    condition = DEPLOY = true
11    endflag   = RETURN = true
12    perpetual = true
13
14            lead = 8
15     lead_damper = 1
16           speed = 2.0   // meters per second
17          radius = 4.0
18       nm_radius = 10.0
19          points = 60,-40:60,-160:150,-160:180,-100:150,-40
20          repeat = 1
21  }
22
23  //---------------------------------------------
24  Behavior = BHV_Waypoint
25  {
26    name       = waypt_return
27    pwt        = 100
28    condition  = RETURN = true
29    condition  = DEPLOY = true
30    perpetual  = true
31    endflag    = RETURN = false
32    endflag    = DEPLOY = false
33
34          speed = 2.0
35         radius = 2.0
36      nm_radius = 8.0
37          point = 0,0
38  }
```

The parameters for each behavior are separated into two groups. Parameters such as `name`, `priority`, `condition` and `endflag` are parameters defined generally for all IvP behaviors. Parameters such as `speed`, `radius`, and `points` are defined specifically for the Waypoint behavior. A convention used in `.bhv` files is to group the general behavior parameters separately at the top of the configuration block.

In this mission, the vehicle follows two sets of waypoints in succession by configuring two instances of a basic waypoint behavior. The second waypoint behavior (lines 23-37) contains only a single waypoint representing the vehicle launch point (0,0). It's often convenient to have the vehicle return home when the mission is completed - in this case when the first waypoint behavior has reached its last waypoint. Although it's possible to simply add (0,0) as the last waypoint of the first waypoint behavior, it is useful to keep it separate to facilitate recalling the vehicle pre-maturely at any point after deployment.

Behavior conditions (lines 9-10, 28-29), and endflags (line 110, lines 31-32) are primary tools for coordinating separate behaviors into a particular mission. Behaviors will not participate unless each of its conditions are met. The condtions are based on current values of the MOOS variables involved in the condition. For example, both behaviors will remain idle unless the variable `DEPLOY` is set to true. This variable is set initially to be false by the initialization on line 1, and is toggled by the `DEPLOY` button on the `pMarineViewer` GUI shown in Figures 59 and 60. The `pMarineViewer` MOOS application is one option for a command and control interface to the helm. The MOOS variables in

the behavior conditions in Listing 1 do not care which process was responsible for setting the value. Endflags are used by behaviors to post a MOOS variable and value when a behavior has reached a completion. The notion of completion is different for each behavior and some behaviors have no notion of completion, but in the case of the waypoint behavior, completion is declared when the last waypoint is reached. In this way, behaviors can be configured to run in a sequence, as in this example, where the returning waypoint behavior will have a necessary condition (line 28) met when the surveying behavior posts its endflag on line 11.

16.3 A Closer Look at the MOOS Apps in the Alpha Example Mission

Running the example mission involves five other MOOS applications in addition to the IvP helm. In this section we take a closer look at what those applications do and how they are configured. The full MOOS file, `alpha.moos`, used to run this mission is given in full in the appendix. An overview of the situation is shown in Figure 61.

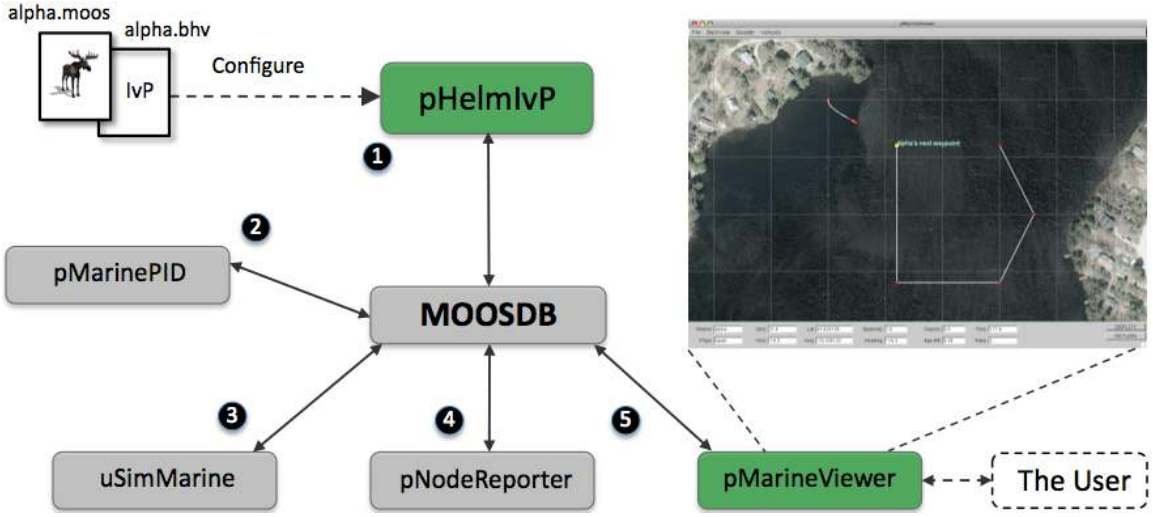

Figure 61: **The MOOS processes in the example "alpha" mission**: In (1) The helm produces a desired heading and speed. In (2) the PID controller subscribes for the desired heading and speed and publishes actuation values. In (3) the simulator grabs the actuator values and the current vehicle pose and publishes a set of MOOS variables representing the new vehicle pose. In (4) all navigation output is wrapped into a single node-report string to be consumed by the helm and the GUI viewer. In (5) the pMarineViewer grabs the node-report and renders a new vehicle posision. The user can interact with the viewer to write limited commmand and control variables to the MOOSDB.

16.3.1 Antler and the Antler Configuration Block

The `pAntler` tool is used to orchestrate the launching of all the MOOS processes participating in this example. From the command line, `pAntler` is run with a single argument the `.moos` file. As it launches processes, it hands each procoess a pointer to this same MOOS file. The Antler configuration block in this example looks like:

Listing 16.2: An example Antler configuration block for the Alpha mission.

```
1  ProcessConfig = ANTLER
2  {
3    MSBetweenLaunches = 200
4
5    Run = MOOSDB          @ NewConsole = false
6    Run = uSimMarine      @ NewConsole = false
7    Run = pNodeReporter   @ NewConsole = false
8    Run = pMarinePID      @ NewConsole = false
9    Run = pMarineViewer   @ NewConsole = false
10   Run = pHelmIvP        @ NewConsole = false
11 }
```

The first parameter on line 2 specifies how much time should be left beteen the launching of each process. Lines 4-9 specify which processes to launch. The MOOSDB is typically launched first. The `NewConsole` switch on each line determines whether a new console window should be opened with each process. Try switching one or more of these to `true` as an experiment.

16.3.2 The pMarinePID Application

The `pMarinePID` application implements a simple PID controller which produces values suitable for actuator control based on inputs from the helm. In simulation the output is consumed by the vehicle simulator rather than the vehicle actuatiors.

In short: The `pMarinePID` application typically gets its info from `pHelmIvP`; produces info consumed by `uSimMarine` or actuator MOOS processes when not running in simulation.

Subcribes to: `DESIRED_HEADING`, `DESIRED_SPEED`.

Publishes to: `DESIRED_RUDDER`, `DESIRED_THRUST`.

16.3.3 The uSimMarine Application and Configuration Block

The `uSimMarine` application is a very simple vehicle simulator that considers the current vehicle pose and actuator commands and produces a new vehicle pose. It can be initialized with a given pose as shown in the configuration block used in this example, shown in Listing 3:

Listing 16.3: An example uSimMarine configuration block for the Alpha mission.

```
1  ProcessConfig = uSimMarine
2  {
3    AppTick  = 10
4    CommsTick = 10
5
6    START_X       = 0
7    START_Y       = 0
8    START_SPEED   = 0
9    START_HEADING = 180
10   PREFIX        = NAV
11 }
```

In short: The `uSimMarine` application typically gets its info from `pMarinePID`; produces info consumed by `pNodeReporter` and itself on the next iteration of `uSimMarine`.

Subcribes to: DESIRED_RUDDER, DESIRED_THRUST, NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING.

Publishes to: NAV_X, NAV_Y, NAV_HEADING, NAV_SPEED.

16.3.4 The pNodeReporter Application and Configuration Block

An Automated Information System (AIS) is commonplace on many larger marine vessels and is comprised of a transponder and receiver that broadcasts one's own vehicle ID and pose to other nearby vessels equiped with an AIS receiver. It periodically collects all latest pose elements, e.g., latitutude and longitude position and latest measured heading and speed, and wraps it up into a single update to be broadcast. This MOOS process collects pose information by subscribing to the MOOSDB for NAV_X, NAV_Y, NAV_HEADING, NAV_SPEED, and NAV_DEPTH and wraps it up into a single MOOS variable called NODE_REPORT_LOCAL. This variable in turn can be subscribed to another MOOS process connected to an actual serial device acting as an AIS transponder. For our purposes, this variable is also subscribed to by pMarineViewer for rendering a vehicle pose sequence.

In short: The pNodeReporter application typically gets its info from uSimMarine or otherwise on-board navigation systems such as GPS or compass; produces info consumed by pMarineViewer and instances of pHelmIvP running in other vehicles or simulated vehicles.

Subcribes to: NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING.

Publishes to: NODE_REPORT_LOCAL

16.3.5 The pMarineViewer Application and Configuration Block

The pMarineViewer is a MOOS process that subscribes to the MOOS variable NODE_REPORT_LOCAL and NODE_REPORT which contains a vehicle ID, pose and timestamp. It renders the updated vehicle(s) position. It is a multi-threaded process to allow both communication with MOOS and let the user pan and zoom and otherwise interact with the GUI. It is non-essential for vehicle operation, but essential for visually confirming that all is going as planned.

In short: The pMarineViewer application typically gets its info from pNodeReporter and pHelmIvP; produces info consumed by pHelmIvP when configured to have command and control hooks (as in this example).

Subcribes to: NODE_REPORT, NODE_REPORT_LOCAL, VIEW_POINT, VIEW_SEGLIST, VIEW_POLYGON, VIEW_MARKER.

Publishes to: Depends on configuration, but in this example: DEPLOY, RETURN.

17 The Charlie Mission

Behaviors:	Loiter, StationKeep, Waypoint
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer, uTimerScript
Primary Topics:	(1) Hierarchical Mode Declarations (2) The Loiter behavior (2a) Loiter traversal - Clockwise vs. counter-clockwise (2b) Loiter polygon shapes, hexagons, ellipses (2c) Loiter dynamic changes to the polygon center position (2d) Loiter robustness to periodic external forces (3) The StationKeep Behavior
Side Topics:	(1) uTimerScript is used to simulate wind gusts, external forces (2) Use of pMarineViewer action buttons and action pull-down menu

17.1 Launching the Charlie Mission

The charlie mission may be launched from the command line in the following manner:

```
$ cd moos-ivp/missions/s3_charlie/
$ ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 63, with a single vehicle, *henry*, initially in the PARK mode. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "LOITERING" mode and begins to proceed to the polygon shown. To get a quick feel for what's possible in this simulation, the following are some things to try. (a) Click on the "CWISE" and "CCWISE" buttons to change the direction of loitering around the polygon. (b) Click the RETURN and DEPLOY buttons repeatedly to see the vehicle switch between two primary modes. (c) Select the "WIND_GUSTS=true" option in the ACTION pull-down menu to see the vehicle adjust to random periodic external forces. (d) Select the "STATION_KEEP=true" option in the ACTION pull-down menu to see how the vehicle station keeps with simulated wind gust. (e) Let the vehicle return to its return point and watch how it automatically switches to the station keeping mode, and then re-deploy it. (f) Select the ellipse polygon from the ACTION pull-down menu to see a different loiter shape. (g) Select the different center points for the loiter ellipse from the ACTION pull-down menu to see just the location of the loiter polygon change and the vehicle adjust.

17.2 Topic #1: Hierarchical Mode Declarations in the Charlie Mission

The Charlie mission is organized by hierarchical mode declarations into four modes: INACTIVE, LOITERING, RETURNING, and STATION-KEEPING. The mode declarations are given a the top of the charlie.bhv file and shown again in Figure 62. The pMarineViewer application by default shows the vehicle's mode alongside the vehicle name, and when the simulation is first launched, the Charlie vehicle is shown on the screen with the mode PARK.

```
// Excerpt from charlie.bhv

set MODE = ACTIVE {
  DEPLOY = true
} INACTIVE

set MODE = LOITERING {
  MODE = ACTIVE
  LOITER = true
  RETURN != true
  STATION_KEEP != true
}

set MODE = RETURNING {
  MODE = ACTIVE
  (RETURN = true)
  STATION_KEEP != true
} STATION-KEEPING
```

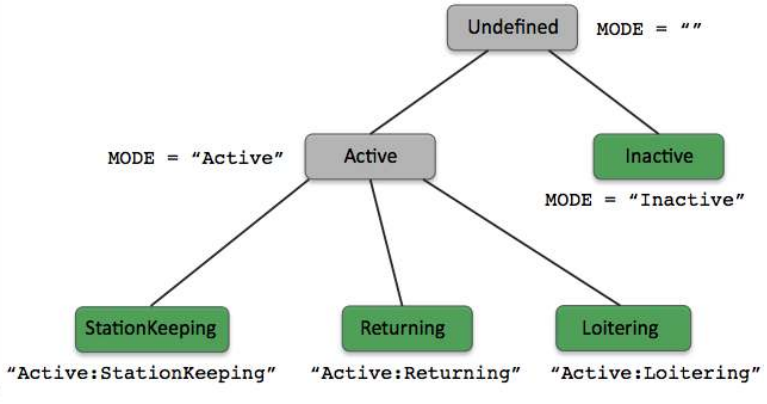

Figure 62: **Charlie Mission Mode Declarations:** The helm, when in drive, will be in one of the four modes indicated by the leaf nodes on the tree. The hierarchical mode structure means, for example, that when the vehicle is in the "RETURNING" mode, it is also considered to be in the "ACTIVE" mode. The vehicle mode, by default, is displayed alongside the vehicle name in the pMarineViewer window.

The "PARK" mode displayed with the vehicle when the simulation is launched is actually the high level helm state. As described in Section 5.2, the helm has a higher level *helm state* which is either PARK or DRIVE, analogous to a car in either the "Park" or "Drive" modes. When the Charlie mission is first launched, the helm is not yet in drive, and is put into drive when the DEPLOY button is hit. This button is configured in the pMarineViewer configuration block to make several MOOS pokes with a single button click. It posts MOOS_MANUAL_OVERRIDE=false, to put the vehicle in the DRIVE helm state. It posts DEPLOY=true to put the helm in the ACTIVE mode, and it posts LOITER=true to put the helm in the ACTIVE:LOITERING mode.

The Charlie mission is configured to allow for easy transition between the other modes via the pMarineViewer interface. The vehicle may be put into the STATION-KEEPING mode at any time via the STATION_KEEP=true option in the ACTION pull-down menu in pMarineViewer. This does nothing more than make the STATION_KEEP=true post to the MOOSDB. The same could have been accomplished by uPokeDB for example:

```
$ cd moos-ivp/missions/s3_charlie/
$ uPokeDB charlie.moos STATION_KEEP=true
```

Referring to the mode declarations in Figure 62, the LOITERING and RETURNING modes both have the condition STATION_KEEP != true, so both of those mode requirements fail to be satisfied. The STATION-KEEPING mode is the default mode, i.e., the "else" mode when the requirements of the RETURNING mode are not met. The STATION_KEEP=true posting could also have been made by another MOOS process, for example, connected to an acoustic modem, an Iridium satellite, or wifi interface.

The helm mode is communicated to pMarineViewer from the helm via the MOOS variable IVPHELM_SUMMARY. This variable is parsed and the mode is rendered next to the vehicle name in the viewer. This rendering may be turned off by toggling through the "name options", with the 'n' key,

or setting the vehicles_name_viewable parameter in the pMarineViewer configuration block to one of the values described in the "Vehicles Pull-Down Menu" Section of [?].

17.3 Topic #2: The Loiter Behavior

The LOITERING mode in the Charlie mission is characterized by the BHV_Loiter behavior, and we explore some of capabilities and options for this behavior here. The behavior configuration, in file charlie.bhv is shown in Listing 1 below. The first configuration block, in lines 3-6, are for parameters defined generally for all IvP Helm behaviors, and the second block, in lines 8-16, are for parameters defined explicitly for the Loiter behavior.

Listing 17.1: Configuration of the Loiter behavior in the Charlie mission, from the file charlie.bhv.

```
1  //-----------------------------------------------
2  Behavior = BHV_Loiter
3  {
4    name      = loiter
5    priority  = 100
6    condition = MODE==LOITERING
7    updates   = UP_LOITER
8
9              speed = 1.3
10         clockwise = false
11   capture_radius = 4.0
12        nm_radius = 15.0
13          polygon = format=radial, x=0, y=-75, radius=40, pts=6, snap=1,
14     visual_hints = nextpt_color=white, nextpt_lcolor=khaki
15     visual_hints = edge_color=blue, vertex_color=yellow
16     visual_hints = edge_size=1, vertex_size=2, label=LOITER_POLYGON
17 }
```

When the vehicle is first deployed, it heads to the hexagon shaped polygon shown in Figure 63, configured in line 13. It traverses the polygon in a counter-clockwise manner due to the configuration in line 10. The Loiter behavior maintains an internal mode, the "acquire_mode" which is true when the vehicle is not sufficiently close to the polygon, defined by the acquire_dist parameter which is by default 10 meters. It also keeps track of how many vertex arrivals come by way of achieving the capture radius, and how many by way of the non-monotonic radius. These internal state variables are summarized by the Loiter behavior by publishing a variable LOITER_REPORT. An example might look like:

```
LOITER_REPORT = "index=2,capture_hits=12,nonmono_hits=3,acquire_mode=false"
```

In the Charlie mission, pMarineViewer is configured to scope on this variable by default, and the evolving LOITER_REPORT may be monitored as the vehicle progresses around the polygon.

17.3.1 Loiter Traversal - Clockwise vs. Counter-Clockwise

The traversal direction of the Loiter behavior is by default clockwise and can be set to counter-clockwise with the parameter clockwise=false. In the Charlie mission, the traversal direction can be changed by selecting UP_LOITER = clockwise=true and UP_LOITER=clockwise=false from the ACTION

pull-down menu. This is a good way to observe the algorithm used by the behavior to acquire the polygon when sufficiently far inside the polygon. The traversal direction could also have been affected by any other MOOS application capable of executing the above two MOOS pokes, including uPokeDB, or any application connected to a communications device.

17.3.2 Loiter Polygon Shapes, Hexagons, and Ellipses

The only restriction on the shape traversed by the Loiter behavior is that it be a convex polygon. The typical specification of the polygon is shown in line 13, via the "radial" syntax, which accepts a center position (`"x=0, y=-75"`), a distance from the center point to each vertex (`"radius=40"`), the number of points in the polygon, (`"pts=6"`), and a snap value, which rounds the vertex positions to, in this case, the nearest whole meter (`"snap=1"`). Ellipse shapes are supported with a string such as:

```
polygon = format=ellipse, x=110, y=-75, degs=150, pts=14, major=80, minor=20
```

The above ellipse specification is selectable in the Charlie mission via the `ACTION` pull-down menu. If selected while in the loiter mode, the vehicle immediately begins traversing to this new polygon, as shown in Figure 64.

17.3.3 Loiter Dynamic Changes to the Polygon Position

In addition to the an outright change to the loiter polygon as described above, which includes shape changes, number of vertices and position, there are a few methods for just changing the polygon position while leaving the other characteristics in tact. The Loiter behavior accepts a parameter for just the center position. The following two such assignments are selectable in the `ACTION` pull-down menu in the `pMarineViewer` for the Charlie mission:

```
center_assign = 40,-40
center_assign = x=100, y=-80
```

Try selecting either of these options, with the polygon set to either the hexagon or ellipse and confirm that only the position changes. One other method for affecting the polygon position is via the `center_activate` parameter, which is `false` by default. When set to `true`, the polygon center is set to be the vehicle's present position when the Loiter behavior enters the running state. In the Charlie mission, try selecting `center_activate=true` from the `ACTION` pull-down menu. If already in the `LOITERING` mode, then nothing happens immediately. If and when the helm exits and returns to the `LOITERING` mode, for example after clicking the `RETURN` button, and then the `DEPLOY` button again, note that the loiter polygon is centered on the vehicle's present position. The `center_activate` feature is particularly useful when the Loiter behavior is used more or less as a station keeping behavior and one wants to be able to command the vehicle to loiter at the present position at any given time.

17.3.4 Loiter Robustness to Periodic External Forces

When the Loiter behavior is active and the vehicle is proceeding around the polygon vertices, it is proceeding more or less like the Waypoint behavior (both behaviors share the same WaypointEngine

class as a sub-component). Things get interesting when the vehicle is approaching the polygon from the outside or from a distance internally (for example when the center_activate parameter set to true. To test and demonstrate this robustness, the Charlie mission is configured with the uTimerScript utility to generate periodic random forces to simulate wind gusts, to push the vehicle off the loiter polygon in unpredictable ways. An example is shown in Figure 65.

The simulated wind gusts may be activated by selecting the WIND_GUSTS=true option from the pMarineViewer ACTION pull-down menu. The details of the simulated forces can be found in the uTimerScript configuration block in the charlie.moos file. The script works by posting external force vectors to the MOOS variable USM_DRIFT_VECTOR_ADD, which is read by the uSimMarine application to alter the prevailing external force vector, which by default has a magnitude of zero. The syntax of the uTimerScript script to implement the drift effects in this example are described in more detail in the "A Script as a Proxy for Simulating Random Wind Gusts" Section of the uTimerScript documentation, [?].

17.4 Topic #3: The StationKeep Behavior

The STATION-KEEPING mode in the Charlie mission is characterized by the BHV_StationKeep behavior, and we explore some of capabilities and options for this behavior here. The behavior configuration, in file charlie.bhv is shown in Listing 2 below. The first configuration block, in lines 3-5, are for parameters defined generally for all IvP Helm behaviors, and the second block, in lines 7-11, are for parameters defined explicitly for the Loiter behavior.

Listing 17.2: Configuration of the StationKeep behavior in the Charlie mission, from the file charlie.bhv.

```
 0  //--------------------------------------------
 1  Behavior = BHV_StationKeep
 2  {
 3    name      = station-keep
 4    priority  = 100
 5    condition = MODE==STATION-KEEPING
 6
 7    center_activate = true
 8       inner_radius = 5
 9       outer_radius = 10
10        outer_speed = 1.0
11         swing_time = 0
12  }
```

17.4.1 Putting the Vehicle into the Station Keeping Mode

The vehicle is put into the station keeping mode by posting STATION_KEEP=true to the MOOSDB. This results in the helm mode, represented by the MOOS variable MODE, being set to station keeping. The hierarchical mode declarations used in the Charlie mission are declared in the top of the charlie.bhv file and were shown in Figure 62. The StationKeep behavior is conditioned on MODE==STATION-KEEPPING, on line 5 above. The StationKeep behavior may be configured to station keep at a specified point in water (with the station_pt parameter), or it may be configured to station keep wherever it happens to be when it enters or re-enters the running state, as it is configured in the Charlie mission by virtue of line 7 above. In the Charlie mission, pMarineViewer is

configured with four on-screen buttons. For more info, see the "Command-and-Control" section of the pMarineVeiwer documentation, [?]. Two of the buttons are used for toggling the STATION_KEEP variable to the MOOSDB.

17.4.2 What is Happening in the Station Keeping Mode

The station keeping behavior (the only running behavior in this mode) attempts to keep the vehicle within a certain distance, given by the inner_radius parameter, to a center point. Inside this radius, it simply lets the vehicle drift. Outside the radius it sets a heading and speed to drive the vehicle back to the center point. The speed decreases as it approaches the inner circle, and is at its highest speed (set by the outer_speed parameter on line 10 above) when its range to the center point is greater than that given by the outer_radius distance. It also makes two postings to the VIEW_POLYGON object to represent the circles implied by the inner_radius and outer_radius parameters, as shown in Figure 66.

17.5 Suggestions for Tinkering in the Charlie Mission

- Station keeping is trivial if the vehicle doesn't drift. Try turning on the artificial wind gusts available in the Charlie mission, courtesy of the uTimerScript script, available by selecting WIND_GUSTS=true from the Action pull-down menu. The vehicle performance is a bit more interesting now in the station keeping mode.

- Try configuring the simulator, uSimMarine, to reflect a vehicle that takes much more time to come to a full stop in the water. This can be done by setting deceleration=0.1 in the uSimMarine configuration block, or by posting USM_DECELERATION=0.1 to the MOOSDB once the mission is running. Note that when the vehicle is put into station keeping mode, its station point is set to the point where it is when it enters this mode. Since the vehicle takes so long to slow down, it immediately drifts out of the inner radius and turns around 180 degrees. This is a typical situation seen in the field. This can be countered a bit by setting the swing_time parameter in the StationKeep behavior. This parameter is the number of elapsed seconds after the vehicle enters the station keeping mode before the station keeping behavior marks its present position as the point to station keep around. Try setting swing_time=5 and re-running the above to see the difference.

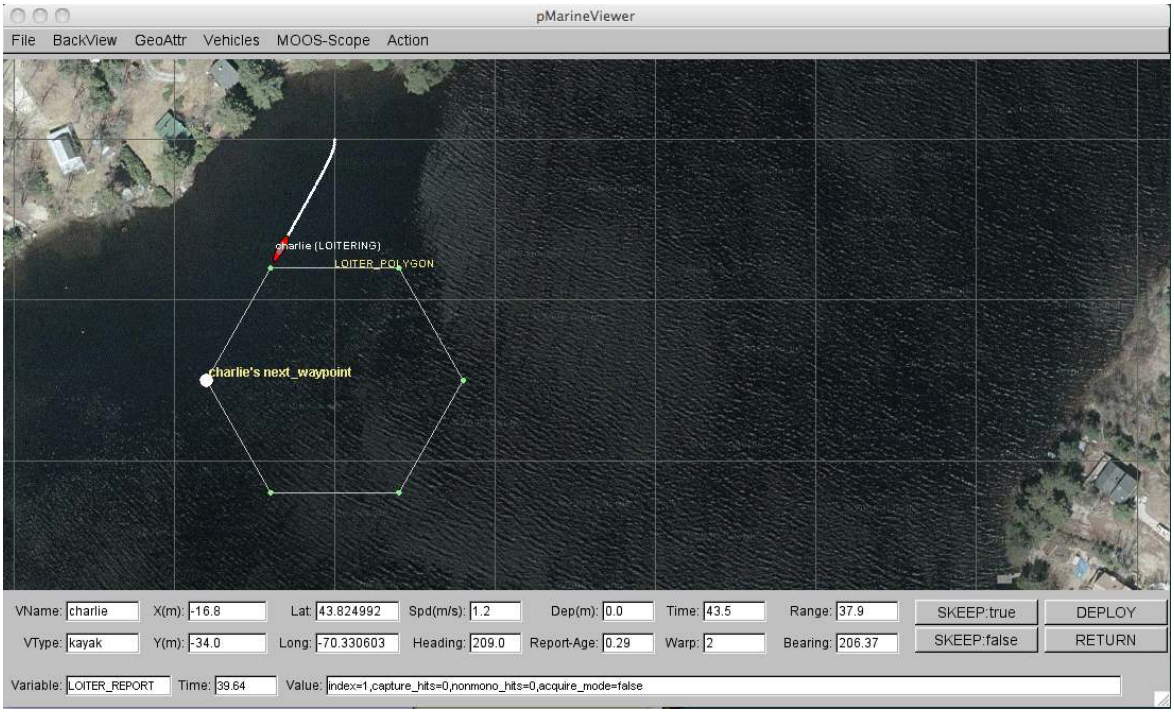

Figure 63: **The Charlie Mission (1):** The vehicle `"charlie"` proceeds to a loiter polygon, traversing in a counter-clockwise manner to the first waypoint labelled `"charlie's next waypoint"`.

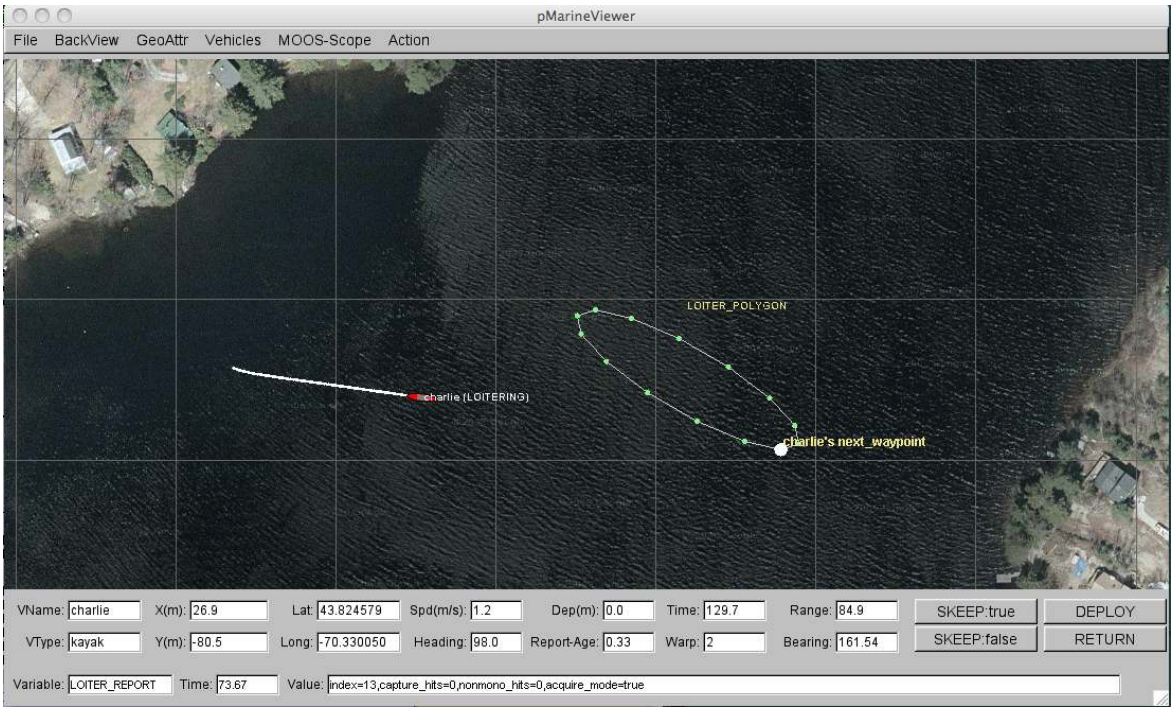

Figure 64: **The Charlie Mission (2):** The loiter traversal polygon for the vehicle `"charlie"` has been dynamically changed to an ellipse via a selection from the `ACTION` pull-down menu.

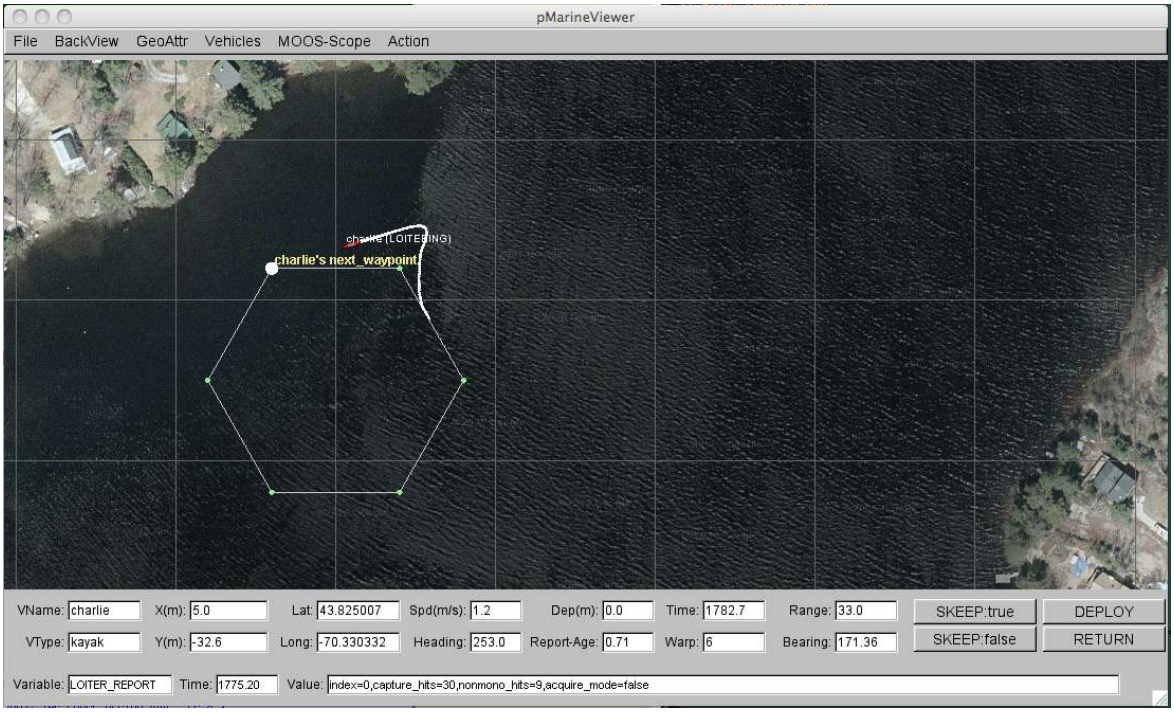

Figure 65: **The Charlie Mission (3):** The vehicle, `"charlie"` recovers from a wind gust and proceeds to re-enter the polygon trajectory at the tangential vertex labelled `"charlie's next waypoint"`.

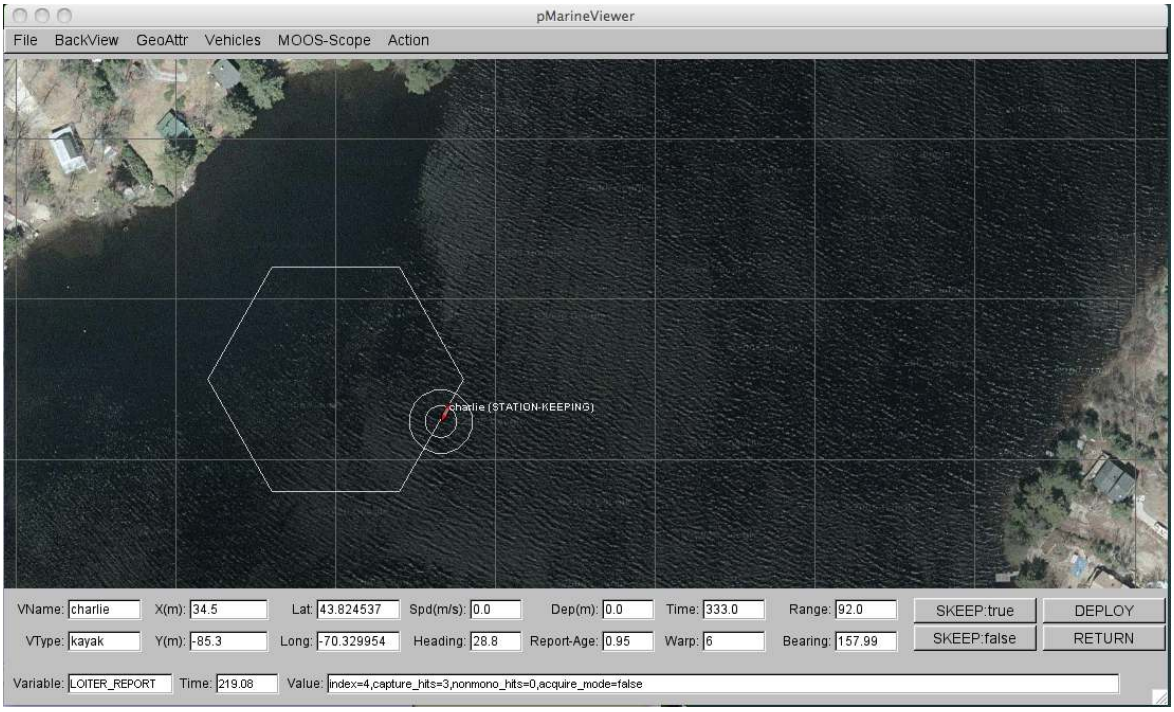

Figure 66: **The Charlie Mission (4):** The vehicle `"charlie"` is put into the station keeping mode. The two rings around the vehicle are the `inner_radius` and `outer_radius` parameters.

18 The Delta Mission

The Delta example mission is used to illustrate the operation of a a vehicle in the depth plane (an underwater as opposed to surface vehicle), the illustration of the PeriodicSurface behavior, the use of the Waypoint behavior with survey patterns, and the use of the pMarineViewer application to send field-control commands to alter the mission as it unfolds. The vehicle will initially deploy using the Loiter behavior, and will periodically come to the surface. If commanded by the user, the vehicle will break off from the Loiter behavior to execute a survey pattern, after which it will resume loitering at its original location.

18.1 Overview of the Delta Mission Components and Topics

Behaviors:	Waypoint, Loiter, PeriodicSurface, ConstantDepth
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer uTimerScript
Primary Topics:	(1) Configuring the Helm for operation at depth (2) The ConstantDepth behavior (3) The PeriodicSurface behavior (4) The Waypoint behavior with survey patterns (5) Using pMarineViewer with Geo-referenced mouse clicks
Side Topics:	(1) uTimerScript is used to simulate a UUV receiving a GPS update.

18.2 Launching the Delta Mission

The delta mission may be launched from the command line:

```
$ cd moos-ivp/missions/s4_delta/
$ ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 67, with a single vehicle, "dudley", initially in the PARK helm state. See Section 5.2 for more on the helm state. After hitting the DEPLOY button in the lower right corner, the vehicle enters the DRIVE state, and the LOITERING helm mode, and begins to proceed along the waypoints as shown. See Section 6.4 for more on more on the helm mode, vs. the helm state. The mode declarations for the Delta mission are defined at the top of the delta.moos file, and amount to the following simple hierarchy:

```
o ROOT
|--o INACTIVE
|--o ACTIVE
   |--o RETURNING
   |--o SURVEYING
   |--o LOITERING
```

In the initial helm mode after deployment, the LOITERING mode, the helm is running the Loitering behavior, Section 26, the ConstantDepth behavior, Section 29, and the PeriodicSurface behavior, Section **??**. It will stay in this mode indefinitely until it is either commanded to return or break off to another region to conduct a survey pattern. In the LOITERING mode the vehicle will periodically

come to the surface for a GPS fix, presumably to correct for accumulated navigation error. The uTimerScript utility is used to simulate the event of receiving a GPS fix. This script is described in more detail in the uTimerScript documentation in the Section "A Script Used as Proxy for an On-Board GPS Unit".

In the SURVEYING mode, the Waypoint behavior becomes active and will execute a survey lawn-mower type pattern, as shown in Figures 68 and 70. In this mode, the periodic surfacing for GPS fixes is suppressed until the pattern is completed. The SURVEYING mode is entered whenever the MOOS variable SURVEY is set to true. In this mission, pMarineViewer is configured to toggle the SURVEY MOOS variable with on-screen buttons, and it is configured to accept mouse clicks which not only put the vehicle into the SURVEYING mode, but also accept the location of the mouse click as the center location of a predefined survey pattern. This kind of pMarineViewer configuration is discussed in Section 18.6.

18.3 Topic #1: Configuring the Helm for Operation at Depth

The Delta mission is configured to simulate a UUV and the helm is configured to produce decisions about the decision variables *heading*, *speed*, as well as *depth*. Adding *depth* to the helm decision space is done in the helm configuration block as shown below in Listing 1, taken from the delta.moos mission file. In this case, the *depth* decision space is defined on line 13 with a range from 0 to 500 meters, with a resolution of 1 meter. See Section 5.4.6 for more on configuring the helm decision space with the domain parameter.

Listing 18.1: Configuring pHelmIvP in the Delta mission to use behaviors concerning vehicle depth.

```
1  // pHelmIvP config block
2
3  ProcessConfig = pHelmIvP
4  {
5    AppTick    = 4
6    CommsTick  = 4
7
8    behaviors  = delta.bhv
9    verbose    = quiet
10   domain     = course:0:359:360
11   domain     = speed:0:4:21
12   domain     = depth:0:500:501
13 }
```

In this example, the *depth* decision variable is a mandatory variable, along with *heading* and *speed*. This means that, on any given helm iteration, there *must* be a decision made about depth, or else the helm will post a helm error. How does the helm ensure that a depth decision is always part of any decision? The helm must be configured such that a behavior that reasons about *depth* is always active regardless of the helm mode. In the Delta mission, the three primary modes, LOITERING, SURVEYING, RETURNING, are all sub-modes of the ACTIVE mode, as shown above. The ConstantDepth behavior is configured to be running whenever the helm is in the Active mode.

18.4 Topic #2: The ConstantDepth Behavior

The ConstantDepth behavior is used in the Delta mission to keep the vehicle a prescribed depth while it is transiting, surveying, and loitering. For simplicity a single ConstantDepth behavior is used, but a different behavior instance could also be used for each vehicle mode. The behavior configuration, in file `delta.bhv` is shown in Listing 2 below. The first part of the configuration block, in lines 3-6, are for parameters defined generally for all IvP Helm behaviors, and the second part, on line 8, is a parameter defined for the ConstantDepth behavior.

Listing 18.2: The ConstantDepth behavior in the Delta mission, from the file delta.bhv.

```
1  Behavior = BHV_ConstantDepth
2  {
3    name      = bhv_const_depth
4    pwt       = 100
5    duration  = no-time-limit
6    condition = MODE==ACTIVE
7
8        depth = 15
9  }
```

The ConstantDepth behavior's primary parameter is the `depth` parameter on line 8. This behavior does make provisions for providing a range of depths, or a more gradually degrading utility function when this behavior is used to work with other behaviors reasoning about depth. In this mission however, the depth is mostly non-contentious between behaviors. The exception is when the PeriodicSurface behavior is active, in which case the priority weight for the PeriodicSurface behavior is simply set to suppress the ConstantDepth behavior.

18.5 Topic #3: The PeriodicSurface Behavior

The PeriodicSurface behavior, described generally in Section **??**, is used in the Delta mission to bring the vehicle to the surface for GPS fixes periodically, to simulate the need for occasional navigation corrections. The behavior configuration, in file `delta.bhv` is shown in Listing 3 below. The first part of the configuration block, in lines 3-6, is for parameters defined generally for all IvP Helm behaviors, and the second part, in lines 8-13, is for parameters defined for the PeriodicSurface behavior.

Listing 18.3: The PeriodicSurface behavior in the Delta mission, from the file delta.bhv.

```
1  Behavior = BHV_PeriodicSurface
2  {
3    name             = bhv_periodic_surface
4    pwt              = 1000
5    condition        = (MODE == LOITERING) or (MODE == RETURNING)
6    condition        = PSURFACE = true
7
8                period = 120
9       zero_speed_depth = 2
10   max_time_at_surface = 60
11          ascent_speed = 1.0
12          ascent_grade = fullspeed
13         mark_variable = GPS_UPDATE_RECEIVED
```

```
14  }
```

The PeriodicSurface behavior outputs an objective function solely on the *depth* decision variable. Note the priority weight in line 4 in Listing 3 is set to 1000, in comparison to the priority weight set for the ConstantDepth behavior on line 4 in Listing 2. When the PeriodicSurface behavior is running, it may or may not be active and producing an objective function over the *depth* decision variable. When it *is* active, the influence on *depth* simply overrules the influence from the ConstantDepth behavior due to these relative priority weights. For this reason, the ConstantDepth behavior may be conditioned on all active modes as in line 6 of Listing 2.

Note the parameter setting on line 13 above, setting the "mark variable" to `GPS_UPDATE_RECEIVED`. This is the default value, and the line could be safely be removed from the configuration block, but it is included anyway to be clear. The PeriodicSurface behavior, once it has noted that the vehicle has reached the surface, will wait until a value has been posted to `GPS_UPDATE_RECEIVED` that is different than its previous posting. Usually a timestamp suffices. The configuration of `uTimerScript` to achieve simulated GPS updates is described in more detail in the `uTimerScript` documentation in the Section "A Script Used as Proxy for an On-Board GPS Unit".

18.6 Topic #4: The Waypoint Behavior with Survey Patterns

The Waypoint behavior, described generally in Section 23, is used in the Delta mission to conduct survey patterns like that shown in Figure 68. The behavior configuration, in file `delta.bhv` is shown in Listing 4 below. Note the survey pattern, described on line 15, does not list a set of points, but instead describes the pattern in terms of its properties such as the lane width and pattern orientation. This format is supported generally for building SegList objects from string specifications, discussed in the Geometry documentation.

Listing 18.4: The Waypoint behavior in the Delta mission, from the file delta.bhv.

```
 1 Behavior = BHV_Waypoint
 2 {
 3   name      = waypt_survey
 4   pwt       = 100
 5   condition = MODE==SURVEYING
 6   perpetual = true
 7   updates   = SURVEY_UPDATES
 8   endflag   = SURVEY = false
 9
10          lead = 8
11    lead_damper = 1
12         speed = 2.0   // meters per second
13        radius = 8.0
14        points = format=lawnmower, label=dudley_survey, x=80, y=-80,
                   width=70, height=30, lane_width=8, rows=north-south,
                   degs=30
15 }
```

The waypoint behavior is configured to execute the survey pattern once, since the `repeat` parameter is not set and defaults to zero. Upon completion, it posts an end flag, configured on line 8, `SURVEY=false` to the `MOOSDB`. This immediately moves the vehicle out of the `SURVEYING` and into either the `LOITERING` or `RETURNING` mode depending on what it was doing when it was commanded

to begin the survey. See the hierarchical mode declarations at the top of the `delta.bhv` file. Since the behavior is set with `perpetual=true` on line 6, completion of the survey pattern merely puts the behavior in a virtual stand-by mode until it is given a new set of waypoints and it once again meets its run conditions.

The waypoint behavior is configured to accept dynamic parameter updates on line 7 through the MOOS variable `SURVEY_UPDATES`. In the Delta mission, updates are initiated by a mouse click in the `pMarineViewer` window, with the result being a post to the MOOSDB of the `SURVEY_UPDATES` variable. Such a posting would consist of a new value for the `points` parameter, replacing the initial configuration on line 14. This is discussed next.

18.7 Topic #4: Using pMarineViewer with Geo-referenced Mouse Clicks

In the Delta mission, the `pMarineViewer` application is used to post messages to the MOOSDB to (a) put the vehicle into the `SURVEYING` mode from either the `LOITERING` or `RETURNING` modes and (b) grab a user specified point in the operation area, from the user's mouse click, to be used as the center point of the commanded survey pattern. This situation is shown in Figure 68. This is achieved by utilizing the Mouse Context feature of `pMarineViewer`, discussed in the `pMarineViewer` documentation, [**?**], in the section *"Contextual Mouse Poking with Embedded OpArea Information"*. The below two lines are inserted into the `pMarineViewer` configuration block in `delta.moos`:

```
left_context[survey-point] = SURVEY = true
left_context[survey-point] = SURVEY_UPDATES = points = vname=$(VNAME),  \
      x=$(XPOS), y=$(YPOS), format=lawnmower, label=delta, width=70,    \
      height=30, lane_width=8, rows=north-south, degs=80
```

Both lines declare that a MOOS variable-value pair is to be posted whenever a left mouse click is generated by the user. The fist line sets `SURVEY=true`, in an attempt to put the helm into the `SURVEYING` mode. This alone will not suffice to switch modes if the current value of the MOOS variable `RETURN` is set to `true`. See the mode declarations near the top of the `delta.bhv` file for this mission - a command to return overrides a command to perform a survey. The second line above associates with a left-mouse click a posting to the `SURVEY_UPDATES` variable. Recall from Listing 4 that this is the very variable through which the surveying waypoint behavior is configured to receive dynamic parameter updates. This posting is configured with a few macros, `$(VNAME)`, `$(XPOS)`, `$(YPOS)`, which are filled in with the name and position of the current vehicle in the `pMarineViewer` window . Each user left-mouse click thus re-assigns the vehicle to a new survey pattern, and switches the helm mode, unless it has been commanded to return.

18.8 Suggestions for Further Experimenting with the Delta Mission

18.8.1 Failing to Reason about Depth

In this mission, *depth* is a mandatory helm decision variable, along with *course* and *speed*, as configured in `delta.moos` and shown in Listing 1. If a mission is not configured properly it's possible to bring about a helm mode where no behavior is reasoning about depth. Declaring *depth* to mandatory is equivalent to declaring that a situation where a helm iteration with no decision on *depth* is an error condition worth of an all-stop.

One way to bring this about in this mission, is to reconfigure the ConstantDepth behavior configuration in `delta.bhv` to replace line 6 in Listing 2 with the following

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

By doing this, there will be no *depth* related behavior when the vehicle is in the `RETURNING` mode. Try re-running the mission. Note when the mission is first launched, the label next to the vehicle in `pMarineViewer` reads `"dudley (PARK)(ManualOverride)"`. This is normal and indicates that the helm state is `PARK`, simply because the operator retains manual control. In addition to seeing this in the `pMarineViewer` window, this can also be confirmed by scoping on a few key helm variables including `IVPHELM_STATE` and `IVPHELM_ALLSTOP`:

```
$ uXMS delta.moos IVPHELM_STATE IVPHELM_ALLSTOP BHV_ERROR --show=source,time,community
```

should produce something similar to:

```
VarName            (S)ource    (T)ime      (C)ommunity   VarValue (MODE = SCOPE:PAUSED)
----------------   ----------  ----------  ----------    ----------- (61)
IVPHELM_STATE      pHelmIvP    81.86       dudley        "PARK"
IVPHELM_ALLSTOP    pHelmIvP    1.69        dudley        "ManualOverride"
BHV_ERROR           n/a         n/a         n/a          n/a
MODE                n/a         n/a         n/a          n/a
```

Note that the `IVPHELM_ALLSTOP` value is posted once, until its value changes, and `IVPHELM_STATE` posts its value on every helm iteration regardless of the value. The latter variable also serves the purpose of a helm heartbeat indicator.

After noting the above, launch the mission by hitting the `DEPLOY` button, and note that the label next to the vehicle has changed to `"dudley (LOITERING)"`. The four MOOS variables from above have also changed to:

```
VarName            (S)ource    (T)ime      (C)ommunity   VarValue (MODE = SCOPE:PAUSED)
----------------   ----------  ----------  ----------    ----------- (103)
IVPHELM_STATE      pHelmIvP    109.56      dudley        "DRIVE"
IVPHELM_ALLSTOP    pHelmIvP    98.07       dudley        "clear"
BHV_ERROR           n/a         n/a         n/a          n/a
MODE               pHelmIvP    98.07       dudley        "ACTIVE:LOITERING"
```

After the vehicle has been running a bit, try hitting the `RETURN` button which changes the helm to the `RETURNING` mode. In this case, due to our tinkering with the mission above, there is no behavior reasoning about depth, and the following message appears next to the vehicle on the screen: `"dudley (Returning)(MissingDecVars:depth)"`. The four scoped MOOS variables also change to the following:

```
VarName            (S)ource    (T)ime      (C)ommunity   VarValue (MODE = SCOPE:PAUSED)
----------------   ----------  ----------  ----------    ----------- (134)
IVPHELM_STATE      pHelmIvP    621.34      dudley        "DRIVE"
IVPHELM_ALLSTOP    pHelmIvP    617.09      dudley        "MissingDecVars:depth"
BHV_ERROR          pHelmIvP    621.59      dudley        "MissingDecVars:depth"
MODE               pHelmIvP    617.09      dudley        "ACTIVE:RETURNING"
```

Note that, despite the error and all-stop event, the helm remains in drive, and may return to loitering or surveying at any time. Try configuring the helm in `delta.moos` to contain the line `park_on_allstop=true` and note the difference in the above experiment. The generated error will bump the helm into the `PARK` helm status.

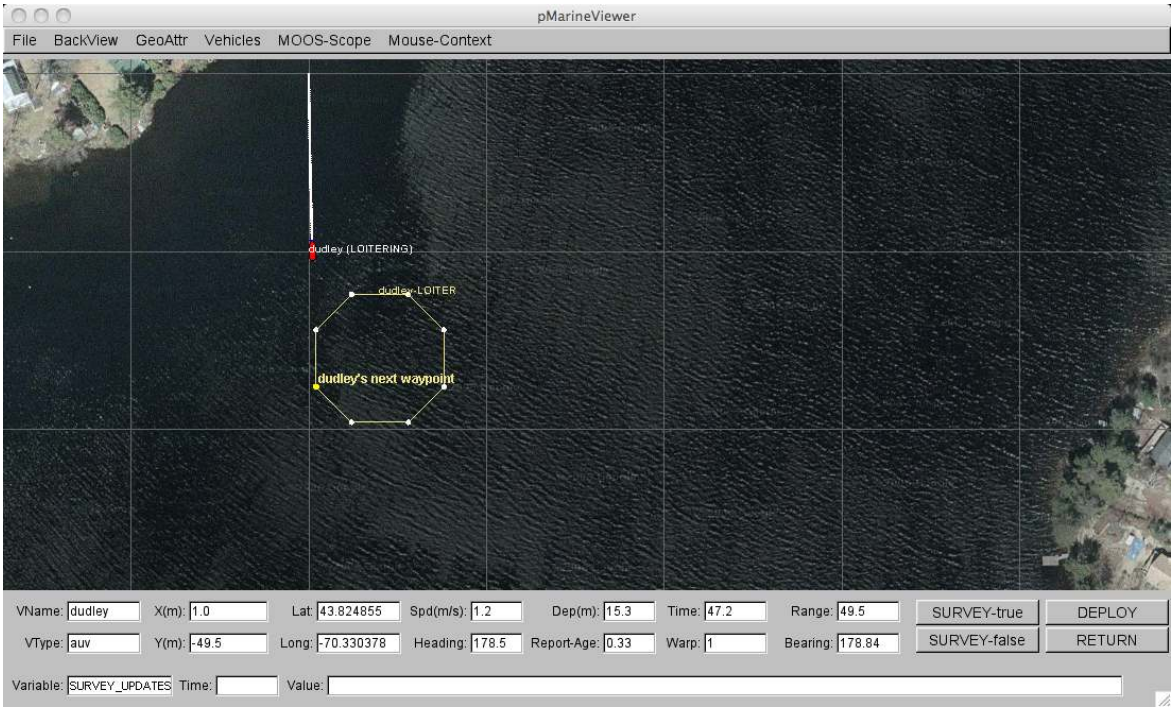

Figure 67: **The Delta Mission (1):** The vehicle `"dudley"` heads to its loiter region where it will remain indefinitely until it is commanded to return or perform a survey pattern. It will periodically surface for a GPS fix.

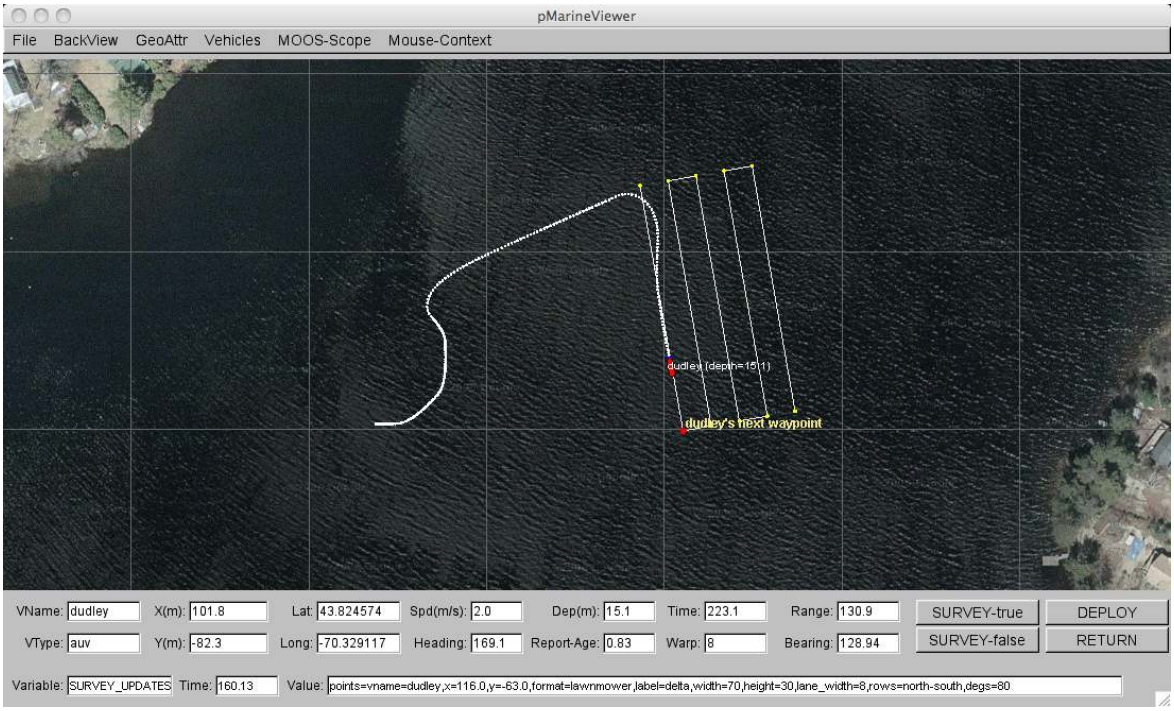

Figure 68: **The Delta Mission (2):** The user has clicked a point in the viewer around which a survey pattern is built. The vehicle exits the loitering mode and begins to execute the survey pattern.

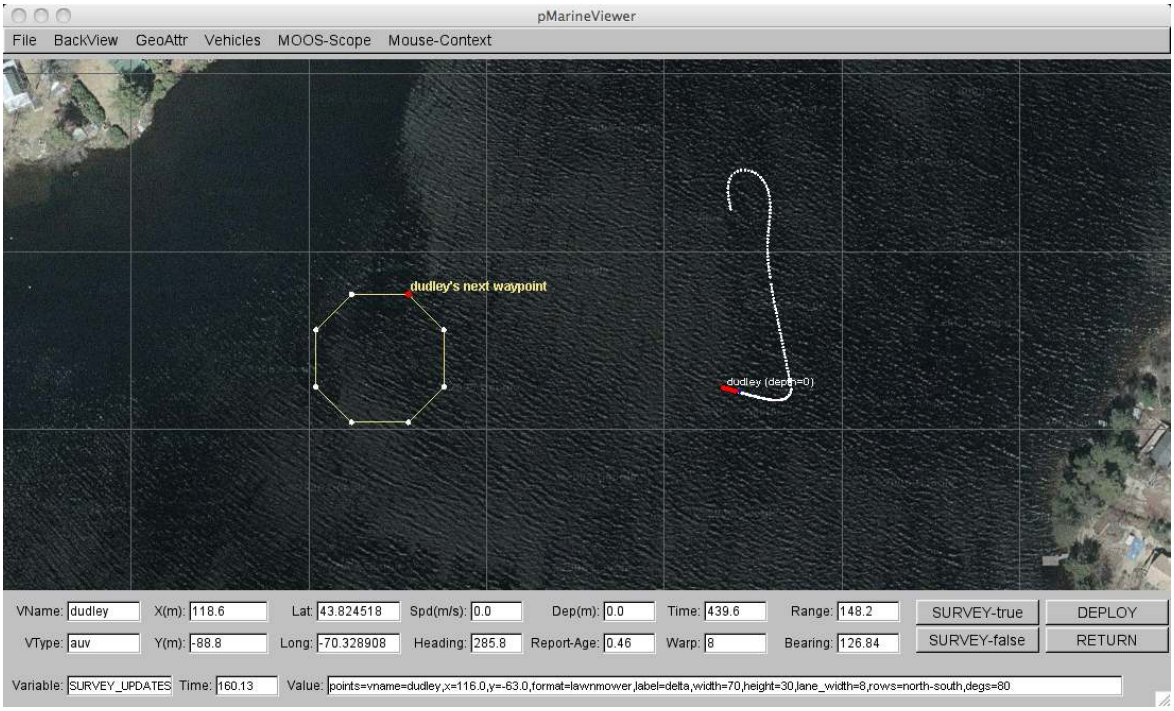

Figure 69: **The Delta Mission (3):** Once the vehicle has finished the survey pattern, it re-enters the loiter mode returning to its loiter region. It resumes periodic surfacing and immediately surfaces for a GPS fix.

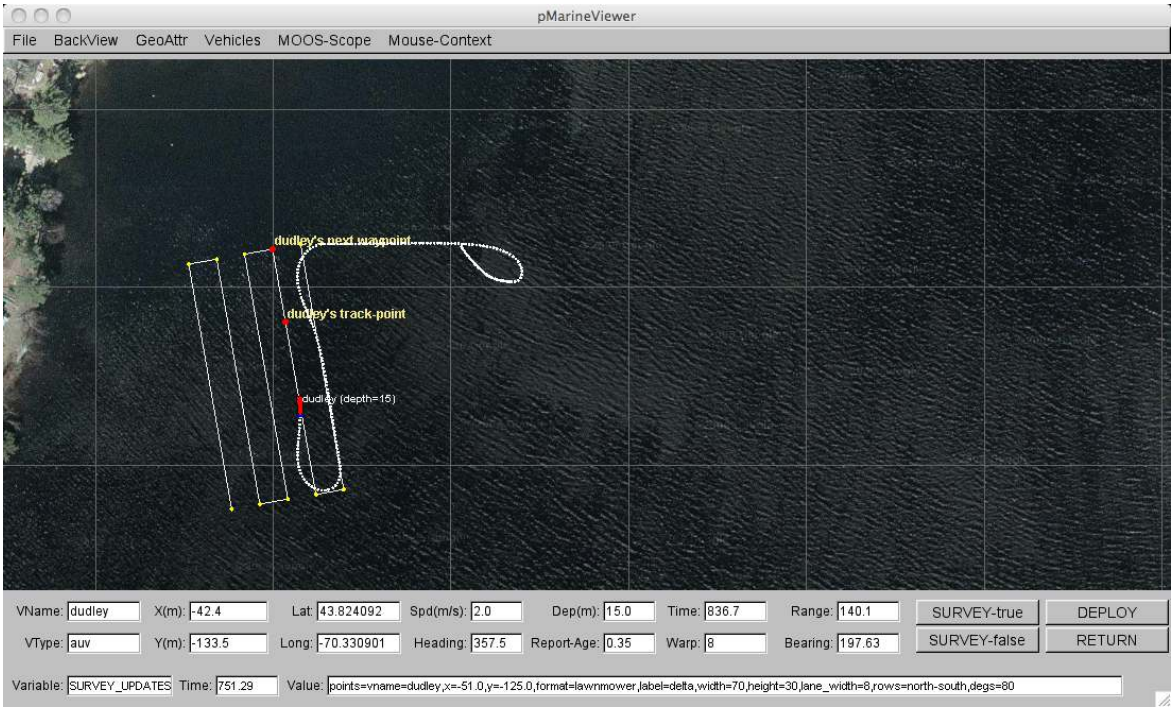

Figure 70: **The Delta Mission (4):** After resumption of loitering, the user clicks in the viewer a new point around which a new survey pattern is built. The entry point of the survey pattern automatically accommodates the vehicle.

19 The Echo Mission

The primary purpose of the Echo example mission is to illustrate the use of dynamically spawned behaviors. A simple behavior, the BearingLine behavior, is used to illustrate the idea. The BearingLine behavior simply posts a viewable point and viewable line segment representing the bearing from the present position of the vehicle to a fixed point in the operation area. Each new "bearing point" posted to the MOOSDB results in a newly spawned BearingLine behavior in the helm.

19.1 Overview of the Echo Mission Components and Topics

Behaviors:	Waypoint, BearingLine
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer
Primary Topics:	(1) The BearingLine behavior. (2) Dynamic behavior spawning. (3) Sending updates to the original and spawned behaviors.
Side Topics:	(1) uTimerScript is used to auto-generate events with random components that lead to behavior spawning. (2) uHelmScope may be used to monitor the spawning and death of behaviors in the helm.

19.2 Launching the Echo Mission

The echo mission may be launched from the command line:

```
$ cd moos-ivp/missions/s5_echo/
$ ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 71, with a single vehicle, *henry*, initially having the "PARK" helm state. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "SURVEYING" mode and begins to proceed along the waypoints as shown.

19.3 Topic #1: The BearingLine Behavior

In Figure 71, note the line segment rendered from the vehicle in the direction of point (100, -100). This line segment is posted to the MOOSDB by the IvP Helm on behalf of the BearingLine behavior. This behavior does not influence the trajectory of the vehicle at all. The line segment acts as an easy visual confirmation that the behavior is instantiated and running properly. In the Echo mission this behavior is configured as in Listing 1. The bearing line originates from the present vehicle position toward the point specified on line 8. The length of the line is 50% of the present distance between the vehicle and the bearing point, as specified on line 7. The bearing point is also configured to be rendered by the configuration on line 9.

Listing 19.1: Configuration of the BearingLine behavior in the Echo example mission.

```
1  Behavior = BHV_BearingLine
2  {
```

```
 3    name       = bng-line
 4    templating = clone
 5    updates    = BEARING_POINT
 6
 7         line_pct = 50
 8    bearing_point = 100,-100
 9          show_pt = true
10  }
```

The BearingLine behavior produces no (IvP) objective function, so by the definition of the behavior run states (Section 6.5.3), it is never in the *active* state. In the Echo mission, the BearingLine behavior is in the *running* state when the vehicle is surveying the five waypoints shown in Figure 71. This can be confirmed by launching a uHelmScope window:

```
$ uHelmScope moos-ivp/ivp/missions/s5_echo/echo.moos -x -p}
```

The output should be similar to that shown in Listing 2 below. Note the Waypoint behavior is active, line 12, and the BearingLine behavior is in the running state, line 14.

Listing 19.2: Output of the uHelmScope tool during the execution of the Echo mission.

```
 1  ==============   uHelmScope Report  ============== DRIVE  (8)
 2    Helm Iteration: 173     (hz=0.25)(5)  (hz=0.25)(100)  (hz=0.26)(max)
 3    IvP functions:  1
 4    Mode(s):
 5    SolveTime:      0.00    (max=0.01)
 6    CreateTime:     0.00    (max=0.01)
 7    LoopTime:       0.00    (max=0.01)
 8    Halted:         false   (0 warnings)
 9  Helm Decision: [speed,0,4,21] [course,0,359,360]
10    course = 189.0
11    speed = 2.0
12  Behaviors Active: ---------- (1)
13    waypt_survey (43.0) (pwt=100.00) (pcs=6) (cpu=0.17)
14  Behaviors Running: --------- (1)
15    bng-line (43.0) (upd=0/0)
16  Behaviors Idle: ------------ (1)
17    waypt_return
18  Behaviors Completed: ------- (0)
19
20  #  MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
21  @  BEHAVIOR-POSTS TO MOOSDB ----------------------- (Hit '@' to en/disable)
```

19.4 Topic #2 Dynamic Behavior Spawning

The Echo mission is configured to allow dynamic behavior spawning for the BearingLine behavior. Lines 4 and 5 in Listing 1 allow the spawning by configuring templating to be enabled on line 4, and specifying the MOOS variable through which spawning requests are received on line 5. Recall that templating can be enabled with either the `"clone"` or `"spawn"` options. In this case, `"clone"`

option was chosen to allow the instantiation of one initial instance upon helm start-up, with the parameter configuration shown.

In this example, the pMarineViewer is configured to convert left-mouse clicks into posts of the BEARING_POINT variable to the MOOSDB, triggering the spawning of new BearingLine instances. A mouse click over the point (-5, -58) results in the post:

```
BEARING_POINT = "name=bng-line-5.0--158.0, bearing_point=-5.0,-158.0"
```

The helm receives mail for the BEARING_POINT variable since it registers automatically for each variable specified in any behavior configured with the updates parameter. The helm examines this string before applying the update, and notes that the behavior name specified is unique (not currently instantiated) and rather than interpreting this as a request to update the existing BearingLine behavior already instantiated, interprets it as a request to spawn a new BearingLine instance if templating is enabled (which it is). The new behavior is spawned, with the behavior name specified. In this case the behavior name is based on the coordinates of the point clicked by the user. Two successive clicks on the same point will result in two posts to BEARING_POINT by pMarineViewer, but the second post will be effectively ignored by the helm. (It is read by the helm, but since the behavior name is one that is already known to the helm, the update is applied to that existing behavior instance. In this case such an update to the bearing_point parameter would be redundant.

After two such user mouse clicks, there will be two new BearingLine behaviors instantiated, and the situation would look similar to that shown in Figure 72. If the uHelmScope tool is still connected as above, the output would looks similar to that shown in Listing 3. Note the existence of three running BearingLine behavior instances reported in lines 15-17. The instance on line 15 was created upon helm startup, and the instances on lines 16-17 were created upon the user mouse clicks.

Listing 19.3: Output of the uHelmScope tool during the later execution of the Echo mission.

```
 1  ==============   uHelmScope Report  ============== DRIVE  (12)
 2    Helm Iteration: 279     (hz=0.25)(5)  (hz=0.25)(100)  (hz=0.26)(max)
 3    IvP functions:  1
 4    Mode(s):        ACTIVE:SURVEYING
 5    SolveTime:      0.00    (max=0.01)
 6    CreateTime:     0.00    (max=0.01)
 7    LoopTime:       0.00    (max=0.01)
 8    Halted:         false   (0 warnings)
 9  Helm Decision: [speed,0,4,21] [course,0,359,360]
10    course = 180.0
11    speed = 2.0
12  Behaviors Active: ---------- (1)
13    waypt_survey (69.6) (pwt=100.00) (pcs=4) (cpu=0.28)
14  Behaviors Running: --------- (3)
15    bng-line (69.6) (upd=0/0)
16    bng-line-5--58 (66.1) (upd=1/1)
17    bng-line13--124 (56.8) (upd=1/1)
18  Behaviors Idle: ------------ (1)
19    waypt_return
```

```
20  Behaviors Completed: ------- (0)
21
22  #  MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
23  @  BEHAVIOR-POSTS TO MOOSDB ---------------------- (Hit '@' to en/disable)
```

19.5 Topic #3: Sending Updates to the Original and Spawned Behaviors

The action associated with the left-mouse click in the Echo mission is configured in the pMarineViewer configuration block in `echo.moos` by:

```
left_context[bng_point] = BEARING_POINT = name=bng-line$(X)-$(Y) # bearing_point=$(X),$(Y)
```

Setting the context for the left and right mouse clicks in pMarineViewer is is achieved by utilizing the Mouse Context feature of pMarineViewer, discussed in the pMarineViewer documentation, [**?**], in the section *"Contextual Mouse Poking with Embedded OpArea Information"*. With the above configuration a left-mouse click may result in the following if clicked at the point `(-5, -58)`:

```
BEARING_POINT = name=bng-line-5--58 # bearing_point=-5,-58
```

Updating or changing the prevailing parameters for existing behaviors is possible via the use of the MOOS variable specified in the `updates` parameter for each behavior. The only difference between an update that changes parameters of an existing behavior, and an update that spawns a new behavior is the inclusion of a `name=<behavior-name>` as above. If this component is present in the string posted to the MOOSDB and the `<behavior-name>` specifies an existing behavior, then that behavior only will have its parameters updated. If it does not specify an existing behavior, a new behavior will be spawned with the specified name. If the `name=<behavior-name>` component is not included in the posting, then the update will be applied to all behaviors configured to receive updates via that particular MOOS variable.

This case is a bit interesting since all newly spawned behaviors specify the same MOOS variable for receiving updates. Indeed a single poke to the MOOS variable BEARING_POINT could result in the simultaneous configuration modification of all instantiated BearingLine behaviors. In the Echo mission, pMarineViewer is configured to make the following pokes to the MOOSDB via the `ACTION` pull-down menu:

```
action  = BEARING_POINT = show_pt=true
action+ = BEARING_POINT = show_pt=false
action  = BEARING_POINT = line_pct=0
action  = BEARING_POINT = line_pct=25
action  = BEARING_POINT = line_pct=50
action  = BEARING_POINT = line_pct=75
action+ = BEARING_POINT = line_pct=100
action  = BEARING_POINT = "name=bng-line # line_pct=0"
action  = BEARING_POINT = "name=bng-line # line_pct=50"
action  = BEARING_POINT = "name=bng-line # line_pct=100"
```

The first two actions will turn on or off the rendering of the bearing points posted by each behavior. The next five actions will adjust the rendering length of the posted bearing line by each behavior. The last two actions will adjust the rendering length of the posted bearing line only for the behavior named "bng-line", the behavior spawned at the time of helm start-up.

19.6 Suggestions for Tinkering

- After the mission is launched, use the ACTION pull-down menu in pMarineViewer to select AUTO_SPAWN=true. This enables a script via uTimerScript that automatically generates new bearing points, spawning new behaviors. These behaviors have their durations set randomly by the script to be in range $[5, 10]$ seconds. Note the new bearing lines emerging and moving with the vehicle until the behavior dies. The script will lead to the spawning (and death) of 5000 behaviors over the course of about two hours of simulation at MOOSTimeWarp=1.

- With AUTO_SPAWN=true as above, let the vehicle return to the launch point (by clicking the RETURN button in the pMarineViewer window. After it reaches the return point and perhaps sits for a bit, hit the DEPLOY button once again to return the vehicle into its SURVEYING mode. Notice that initially there are many bearing lines rendered before returning to only a handful of bearing lines after a few seconds of simulation. This is because newly spawned behaviors do not start their duration clock until the first time the behavior enters the running state. All behaviors spawned while returning to the start point are effectively put on hold until the vehicle is re-deployed. Then they all start their duration clocks simultaneously and all die off 5-10 seconds later.

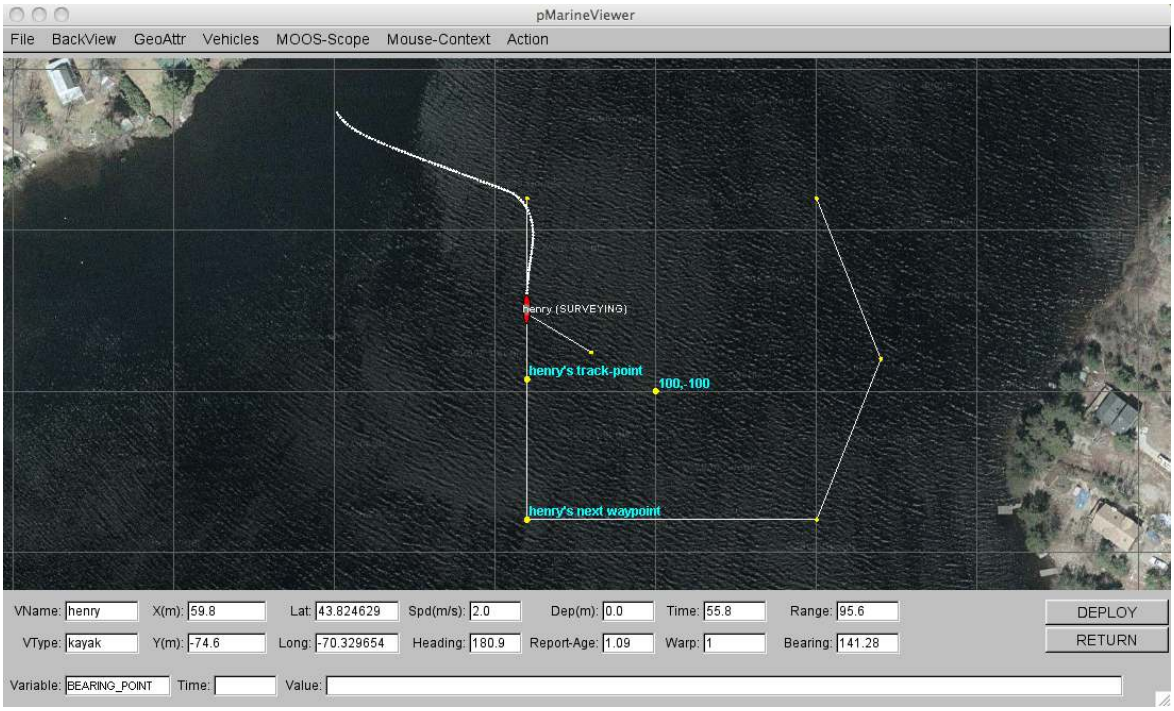

Figure 71: **The Echo Mission (1):** The vehicle `"henry"` traverses waypoints with the Waypoint behavior. The BearingLine behavior generates a viewable "bearing point" at `(100, -100)`, and a viewable "bearing line".

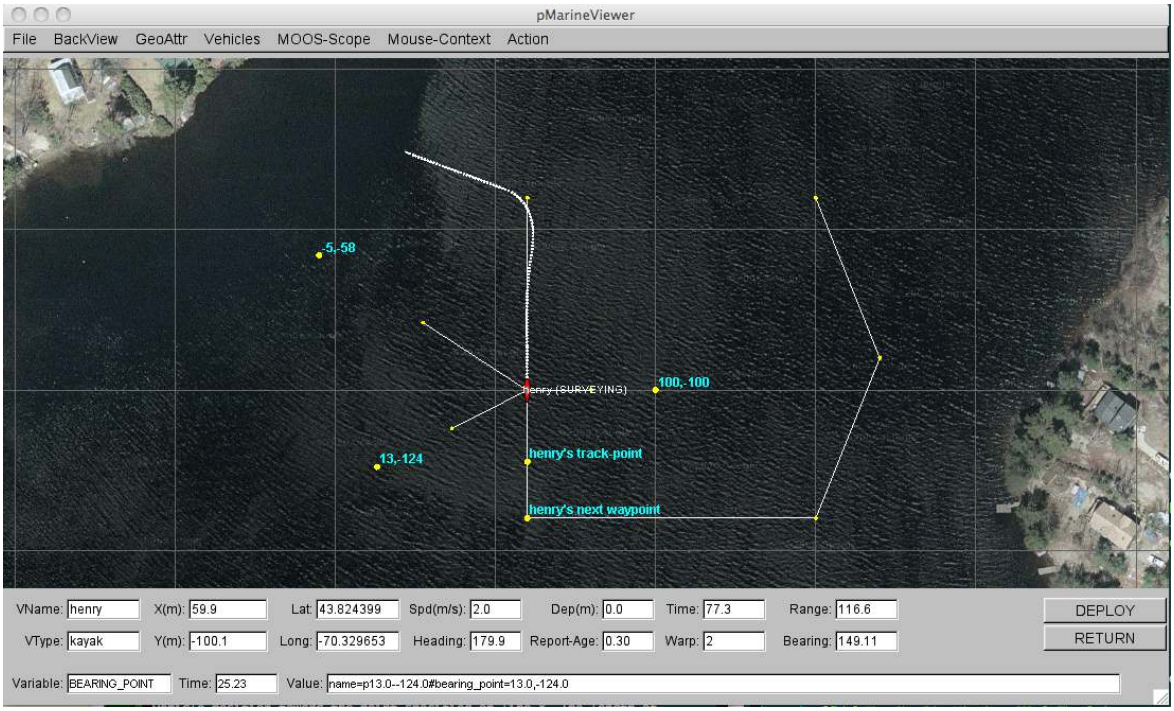

Figure 72: **The Echo Mission (2):** The user has clicked two new bearing points, `(-5, -58)`, and `(13, -124)`, and two new BearingLine behaviors have been spawned, each generating bearing line and bearing point visual outputs.

20 Mission S11: The Kilo Mission

The purpose of the Kilo mission is to illustrate the use of the standby helm, and the use of the TestFailure behavior. The standby helm consists of an otherwise normally configured helm with the additional standby parameter invoked. This helm will wait in standby mode until the heartbeat of another (primary) helm ceases to be posted to the MOOSDB for some period of time. In this example mission, a primary and standby helm are configured with the primary helm executing a simple mission similar to the Alpha mission, but also using an instance of the TestFailure behavior. This behavior will be configured to trigger either a crash or hang of the primary helm to demonstrate the manner in which a standby helm will step in to take over with its own mission. The standby mission in this case is simply a behavior to return the vehicle to its launch position.

20.1 Overview of the Kilo Mission Components and Topics

Behaviors:	Waypoint, TestFailure
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer
Primary Topics:	(1) The use of a standby helm. (2) The TestFailure behavior. (3) Scoping the primary and standby helm states at runtime.
Side Topics:	(1) uHelmScope may be used to monitor the relationship between the shadow helm and primary helm.

20.2 Launching the Kilo Mission

The Kilo mission may be launched from the command line:

```
$ cd moos-ivp/missions/s11_kilo/
$ ./launch.sh --warp=10          (or more simply, ./launch.sh 10)
```

This should bring up a pMarineViewer window like that shown in Figure 76, with a single vehicle, "kilo", initially in the "PARK" mode. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "SURVEYING" mode and begins to proceed along the waypoints as shown.

20.3 Topic #1: The Use of a Standby Helm

The vehicle in this example is configured with both a primary helm and a secondary helm. Configuring a mission with two helm instances is straight forward. The primary helm is configured as done normally, as in lines 3-11 in Listing 1 below. The standby helm is configure identically but with one additional line, as in line 23 in Listing 1 below. This line denotes the helm it to be a *standby* helm, and indicates the time threshold used in determining when to take over from a primary helm.

Listing 20.1: Configuration of the primary and secondary helm in the Kilo mission.

```
1  //---------------------------- pHelmIvP config block
2
3  ProcessConfig = pHelmIvP
4  {
```

```
5    AppTick    = 4
6    CommsTick  = 4
7
8    behaviors  = kilo.bhv
9    domain     = course:0:359:360
10   domain     = speed:0:4:21
11 }
12
13 //---------------------------- pHelmIvP_Standby config block
14
15 ProcessConfig = pHelmIvP_Standby
16 {
17   AppTick    = 4
18   CommsTick  = 4
19
20   behaviors  = kilo_standby.bhv
21   domain     = course:0:359:360
22   domain     = speed:0:4:21
23   standby    = 2
24 }
```

Both helms are instances of the pHelmIvP application, but one of them is named pHelmIvP_Standby. The name of the application is not in any way used to put this helm instance into a standby role. It is chosen simply to be different from the primary helm, since the MOOSDB requires all connected applications to have unique names, and to be clear when or if debugging the mission.

ched by pAntler with the name pHelmIvP_Standby. This is done by giving it the alternative name using the pAntler ExtraProcessParams parameter as shown on lines 15 and 18 below.

Listing 20.2: Configuration of Antler.

```
1  //------------------------------------------
2  // Antler configuration  block
3
4  ProcessConfig = ANTLER
5  {
6    MSBetweenLaunches = 200
7
8    Run = MOOSDB            @ NewConsole = false
9    Run = pLogger           @ NewConsole = false
10   Run = uSimMarine        @ NewConsole = false
11   Run = pNodeReporter     @ NewConsole = false
12   Run = pMarinePID        @ NewConsole = false
13   Run = pMarineViewer     @ NewConsole = false
14   Run = pHelmIvP          @ NewConsole = true
15   Run = pHelmIvP          @ NewConsole = true, ExtraProcessParams=HParams
16   Run = uProcessWatch     @ NewConsole = false
17
18   HParams=--alias=pHelmIvP_Standby
19 }
```

If the shadow helm is launched independently, not with pAntler, it may be launched with:

```
$ pHelmIvP kilo.moos --alias=pHelmIvP_Standby
```

20.4 Topic #2: The TestFailure Behavior

The Kilo mission uses the `TestFailure` behavior to artificially generate a failure of the primary helm. In the Kilo mission it is configured to produce a "hung" helm with the following configuration:

Listing 20.3: Configuration of the TestFailure Behavior.

```
1  //-----------------------------------------------
2  Behavior = BHV_TestFailure
3  {
4    name               = test_failure
5    condition          = DEPLOY=true
6    duration           = 120
7    duration_idle_decay = false
8
9          failure_type  = hang,3
10 }
```

The first four configuration parameters are defined for all IvP behaviors. A run condition of `DEPLOY=true` keeps this behavior idle until the mission is launched. Once it is launched, a countdown of 120 seconds begins until the behavior will hang. The `duration_idle_decay=false` setting ensures that the countdown doesn't begin until the behavior is in the running state. In this case the behavior will fail by hanging for three seconds. This will cause the primary helm to also appear to hang for three seconds, noted by a three second gap between heartbeats, defined by posting to the MOOS variable `IVPHELM_STATE`. Since the standby helm is configured to wait no longer than two seconds (see Listing 1), the standby helm will promptly take over control from the primary helm. The sequence of events in taking over control is described in more detail next.

20.5 Topic #3: Scoping the Helm State(s) at Runtime

The relationship between the primary and standby helm may be monitored by scoping on the variable `IVPHELM_STATE` during the course of the mission. Figure 73 below shows the situation shortly after the vehicle is deployed using the `uXMS` scoping tool.

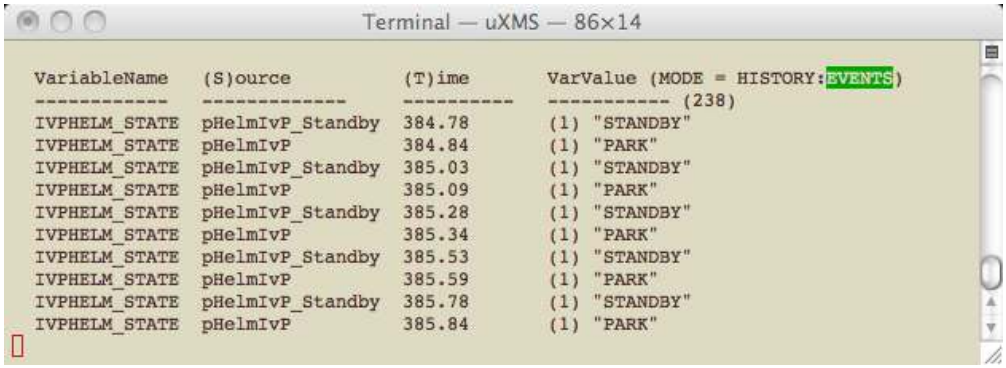

Figure 73: **Helm State in the Kilo Mission:** Both the standby and primary helms are operating normally, posting a helm state message about four times per second.

Initially the standby helm is showing a helm state of `"STANDBY"`, and the primary helm is showing a helm state of `"DRIVE"`. Since both helms are operating at the same frequency, they mostly alternate between postings as shown above.

The events shown in Figure 74 show that during the 9 identical postings ending at time 329.86, the standby helm is the only helm emitting a heartbeat. During this period the primary helm has either crashed or hung. Finally the secondary helm takes over and posts a helm state of `"DRIVE+"` for 107 consecutive heartbeats (iterations). Recall the `"+"` is to further distinquish that this helm was originally configured as a standby helm.

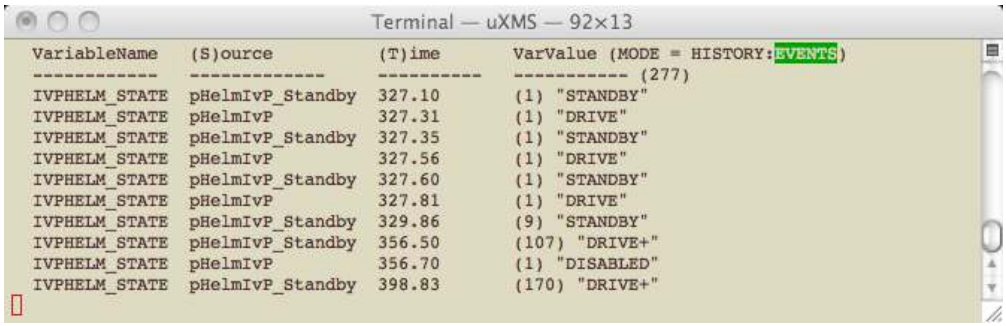

Figure 74: **Helm State in Kilo Mission after Standby Helm Takes Over Hung Primary Helm:** The standby helm has detected a delay in the primary helm heartbeat and takes over about two seconds later. Note the number in parentheses is the number of identical postings with the timestamp showing the time of the last of the identical postings.

Eventually it turns out that the original primary helm did not crash after all but was just temporarily hung. After emerging from its hung state, upon reading mail in the next loop, it realizes the standby helm has taken over and immediately concedes control and posts the `"DISABLED"` helm state, and will never again post. One may wonder why the primary helm, when it finally woke up, did not first mistakely post `"DRIVE"` before conceding control - after all it needs to first read its mail to learn that another helm is in control. The primary helm in fact did post `"DRIVE"` and `"DISABLED"` on two consecutive iterations. The first posting occurred (time 327.81) prior to querying the behaviors for input, and the second posting (time 356.70) occurred on the next iteration of upon reading mail.

In the case where the standby helm takes over for a crashed helm, the sequence of events, viewed from the perspective of posts to IVPHELM_STATE, is similar as shown in Figure 75. When things are going fine the standby helm and primary helm alternately post helm states of STANDBY and DRIVE respectively as the first several posts below shoe. At some point the primary helm crashes and the only posts are made by the standby helm. In the example here, there are 10 IVPHELM_STATE="STANDBY" posts made while the standby helm is getting closer to triggering a take-over. Finally, upon takeover, it posts a helm state of "DRIVE+" thereafter. The primary helm is never heard from again.

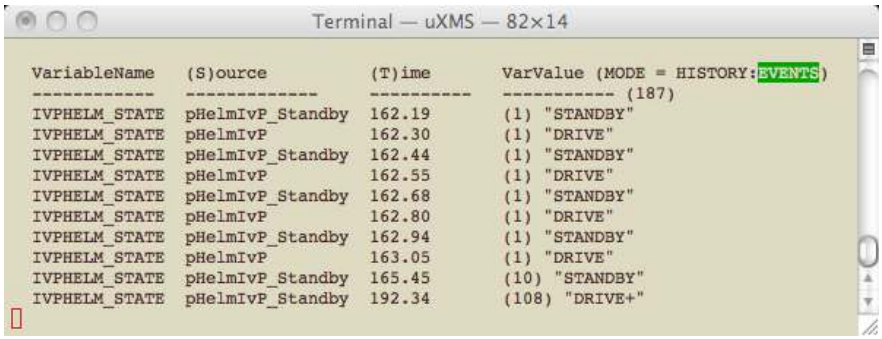

Figure 75: **Helm State in Kilo Mission after Standby Helm Takes Over Crashed Primary Helm:** The standby helm has detected a delay in the primary helm heartbeat and takes over about two seconds later. Note the number in parentheses is the number of identical postings with the timestamp showing the time of the last of the identical postings.

20.6 Suggestions for Tinkering

- *Analyze a helm takeover from the log files.* Run the Kilo mission again and, as opposed to monitoring things with a live scope as in Figures 73 - 75, let the mission play out. Change directories to the folder where the .alog file was created. Take a look at the IVPHELM_STATE postings using the aloggrep tool. Confirm that the sequence of events described in Figures 73 - 75 are consistent with entries of the log file. Hint: This can be achieved without mission configuration or code modifacations.

- *Crashing with helm with a mouse click.* Rather than waiting for the TestFailure behavior to run down its clock to failure, configure it to fail immediately upon entering the run state (duration=0). But also set an additional run-condition, e.g., FAIL=true and configure the pMarineViewer to make this posting upon a mouse-click. Test the modified Kilo mission by launching it, and clicking the mouse when ready to see the failure. This can be achieved solely through mission configuration modification, without code modification.

- *Configure a standby helm that finishes a task.* Create a primary and shadow helm where the primary helm is surveying a set of waypoints. The shadow helm should be able to survey the remaining points when/if the primary helm is taken over. Hint: this will likely require code generation or modification. Extra bonus if achievable solely through mission modification.

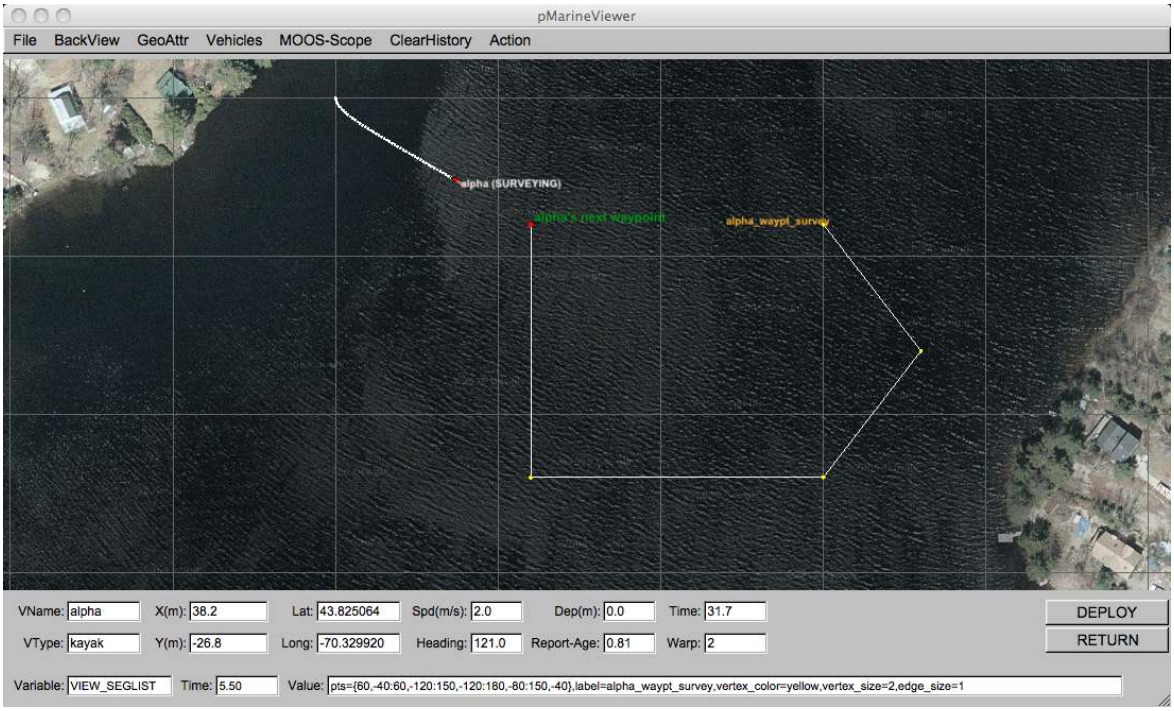

Figure 76: **The Kilo Mission (1):** The vehicle `"kilo"` traverses the waypoints using a primary helm, launched alongside a standby helm. The TestFailure behavior is a ticking bomb that will hang the primary helm momentarily.

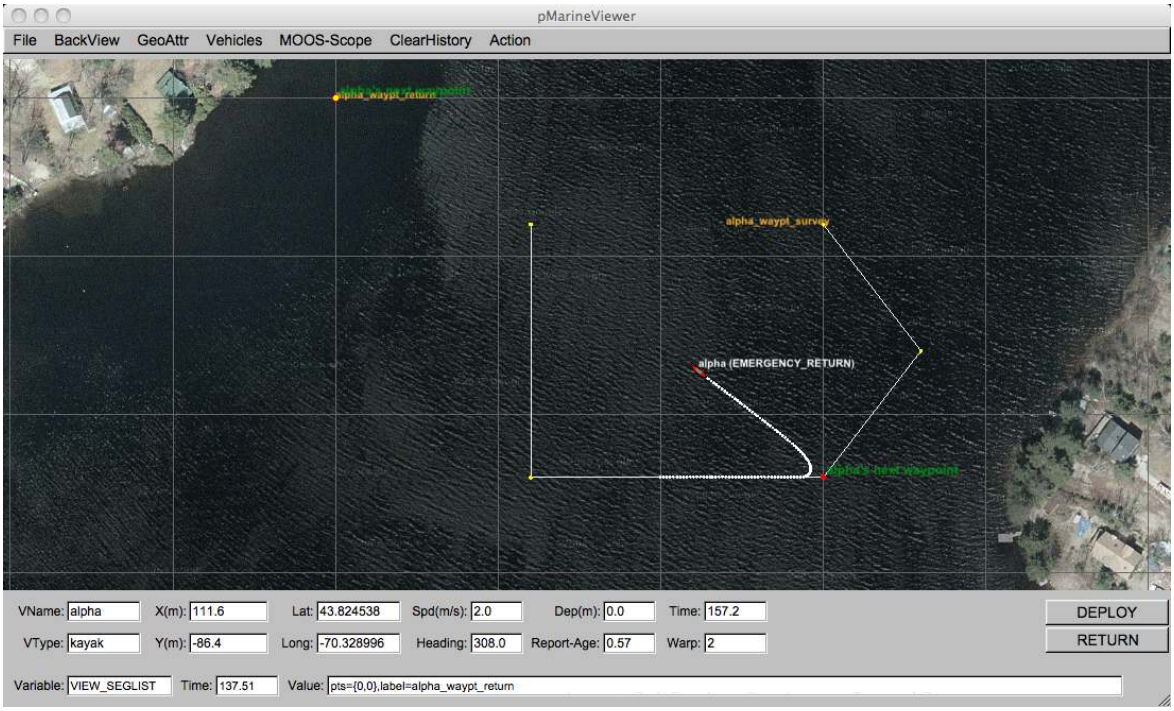

Figure 77: **The Kilo Mission (2):** The TestFailure behavior has hung the primary helm. The standby helm has detected the absence of a heartbeat (the `IVPHELM_STATE` variable), and take over with a simple return mission.

21 Colors

Below are the colors available for for any MOOS-IvP app or behavior that has configurable colors. Colors are case insensitive. A color may be specified by the string as shown, or with the '_' character as a separator. Or the color may be specified with its hexadecimal or floating point form. For example the following are equivalent:

`"darkblue", "DarkBlue", "dark_blue", "hex:00,00,8b", and "0,0,0.545".`

In the latter two styles, the '%', '$', or '#' characters may also be used as a delimiter instead of the comma if it helps when embedding the color specification in a larger string that uses its own delimeters. Mixed delimeters are not supported however.

In most cases, the colors `invisible`, `empty`, `off` are aliases indicating that the relevant object, line or vertex etc is not to be rendered at all.

antiquewhite, (fa,eb,d7)
aquamarine (7f,ff,d4)
beige (f5,f5,dc)
black (00,00,00)
blue (00,00,ff)
brown (a5,2a,2a)
cadetblue (5f,9e,a0)
chocolate (d2,69,1e)
cornsilk (ff,f8,dc)
crimson (de,14,3c)
darkblue (00,00,8b)
darkgoldenrod (b8,86,0b)
darkgreen (00,64,00)
darkmagenta (8b,00,8b)
darkorange (ff,8c,00)
darkred (8b,00,00)
darkseagreen (8f,bc,8f)
darkslategray (2f,4f,4f)
darkviolet (94,00,d3)
deepskyblue (00,bf,ff)
dodgerblue (1e,90,ff)
floralwhite (ff,fa,f0)
fuchsia (ff,00,ff)
ghostwhite (f8,f8,ff)
goldenrod (da,a5,20)
green (00,80,00)
honeydew (f0,ff,f0)
aqua (00,ff,ff)
azure (f0,ff,ff)
bisque (ff,e4,c4)
blanchedalmond (ff,eb,cd)
blueviolet (8a,2b,e2)
burlywood (de,b8,87)
chartreuse (7f,ff,00)
coral (ff,7f,50)
cornflowerblue (64,95,ed)
cyan (00,ff,ff)
darkcyan (00,8b,8b)
darkgray (a9,a9,a9)
darkkhaki (bd,b7,6b)
darkolivegreen (55,6b,2f)
darkorchid (99,32,cc)
darksalmon (e9,96,7a)
darkslateblue (48,3d,8b)
darkturquoise (00,ce,d1)
deeppink (ff,14,93)
dimgray (69,69,69)
firenrick (b2,22,22)
forestgreen (22,8b,22)
gainsboro (dc,dc,dc)
gold (ff,d7,00)
gray (80,80,80)
greenyellow (ad,ff,2f)
hotpink (ff,69,b4)

indianred (cd,5c,5c)
ivory (ff,ff,f0)
lavender (e6,e6,fa)
lawngreen (7c,fc,00)
lightblue (ad,d8,e6)
lightcyan (e0,ff,ff)
lightgray (d3,d3,d3)
lightpink (ff,b6,c1)
lightseagreen (20,b2,aa)
lightslategray (77,88,99)
lightyellow (ff,ff,e0)
limegreen (32,cd,32)
magenta (ff,00,ff)
mediumblue (00,00,cd)
mediumseagreen (3c,b3,71)
mediumspringgreen (00,fa,9a)
mediumvioletred (c7,15,85)
mintcream (f5,ff,fa)
moccasin (ff,e4,b5)
navy (00,00,80)
olive (80,80,00)
orange (ff,a5,00)
orchid (da,70,d6)
paleturquoise (af,ee,ee)
papayawhip (ff,ef,d5)
pelegoldenrod (ee,e8,aa)
pink (ff,c0,cb)
powderblue (b0,e0,e6)
red (ff,00,00)
royalblue (41,69,e1)
salmon (fa,80,72)
seagreen (2e,8b,57)
sienna (a0,52,2d)
skyblue (87,ce,eb)
slategray (70,80,90)
springgreen (00,ff,7f)
tan (d2,b4,8c)
thistle (d8,bf,d8)
turquoise (40,e0,d0)
wheat (f5,de,b3)
indigo (4b,00,82)
khaki (f0,e6,8c)
lavenderblush (ff,f0,f5)
lemonchiffon (ff,fa,cd)
lightcoral (f0,80,80)
lightgoldenrod (fa,fa,d2)
lightgreen (90,ee,90)
lightsalmon (ff,a0,7a)
lightskyblue (87,ce,fa)
lightsteelblue (b0,c4,de)
lime (00,ff,00)
linen (fa,f0,e6)
maroon (80,00,00)
mediumorchid (ba,55,d3)
mediumslateblue (7b,68,ee)
mediumturquoise (48,d1,cc)
midnightblue (19,19,70)
mistyrose (ff,e4,e1)
navajowhite (ff,de,ad)
oldlace (fd,f5,e6)
olivedrab (6b,8e,23)
orangered (ff,45,00)
palegreen (98,fb,98)
palevioletred (db,70,93)
peachpuff (ff,da,b9)
peru (cd,85,3f)
plum (dd,a0,dd)
purple (80,00,80)
rosybrown (bc,8f,8f)
saddlebrowm (8b,45,13)
sandybrown (f4,a4,60)
seashell (ff,f5,ee)
silver (c0,c0,c0)
slateblue (6a,5a,cd)
snow (ff,fa,fa)
steelblue (46,82,b4)
teal (00,80,80)
tomatao (ff,63,47)
violet (ee,82,ee)
white (ff,ff,ff)

whitesmoke (f5,f5,f5)
yellow (ff,ff,00)
yellowgreen (9a,cd,32)

22 Use of Logic Expressions

Logic conditions are employed in both the pHelmIvP, uTimerScript, uQueryDB and alogcheck applications, and others, to condition certain activities based on the prescribed logic state of elements of the MOOSDB. The use of logic conditions in the helm is done in behavior file (.bhv file). For the uTimerScript application, logic conditions are used in the configuration block of the mission file (.moos file). The MOOS application using logic conditions maintains a local buffer representing a snapshot of the MOOSDB for variables involved in the logic expressions. The key relationships and steps are shown in Figure 78:

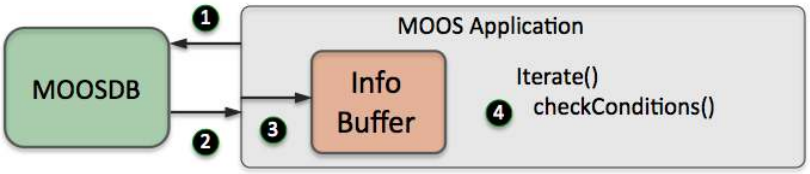

Figure 78: **Logic conditions in a MOOS application**: Step 1: the applications registers to the MOOSDB for any MOOS variables involved in the logic expressions. Step 2: The MOOS application reads incoming mail from the MOOSDB. Step 3: Any new mail results in an update to the information buffer. Step 4: Within the application's Iterate() method, the logic expressions are evaluated based on the contents of the information buffer.

The logic conditions are configured as follows:

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

The parameter condition is case insensitive. When multiple conditions are specified, it is implied that the overall criteria for meeting conditions is the conjunction of all such conditions. In what remains below, the allowable syntax for <logic-expression> is described.

Simple Relational Expressions

Each logic expression is comprised of either Boolean operators (and, or, not) or relation operators ($\leq, <, \geq, >, =, !=$). All expressions have at least one relational expression, where the left-hand side of the expression is a MOOS variable, and the right-hand side is a literal (either a string or numerical value). The literals are treated as a string value if quoted, or if the value is non-numerical. Some examples:

```
condition = (DEPLOY   = true)    // Example 1
condition = (QUALITY >= 75)      // Example 2
```

Variable names are case sensitive since MOOS variables in general are case sensitive. In matching string values of MOOS variables in Boolean conditions, the matching is *case insensitive*. If for example, in Example 1 above, the MOOS variable DEPLOY had the value "TRUE", this would satisfy the condition. But if the MOOS variable deploy (lowercase is unconventional, but legal) had the value "true", this would not satisfy Example 1.

Simple Logical Expressions with Two MOOS Variables

A relational expression generally involves a variable and a literal, and the form is simplified by insisting the variable is on the left and the literal on the right. A relational expression can also involve the comparison of two variables by surrounding the right-hand side with `$()`. For example:

```
condition = (REQUESTED_STATE != $(RUN_STATE))     // Example 3
```

The variable types need to match or the expression will evaluate to `false` regardless of the relation. The expression in Example 3 will evaluate to `false` if, for example, `REQUESTED_STATE="run"` and `RUN_STATE=7`, simply because they are of different type, and regardless of the relation being the inequality relation.

Complex Logic Expressions

Individual relational expressions can be combined with Boolean connectors into more complex expressions. Each component of a Boolean expression must be surrounded by a pair of parentheses. Some examples:

```
condition = (DEPLOY = true) or (QUALITY >= 75)              // Example 4

condition = (MSG != error) and !((K <= 10) or (w != 0))     // Example 5
```

A relational expression such as `(w != 0)` above is false if the variable `w` is undefined. In MOOS, this occurs if the variable has yet to be published with a value by any MOOS client connected to the MOOSDB. A relational expression is also `false` if the variable in the expression is the wrong type, compared to the literal. For example `(w != 0)` in Example 5 would evaluate to `false` even if the variable `w` had the string value `"alpha"` which is clearly not equal to zero.

23 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. Them basic idea is shown in Figure 79.

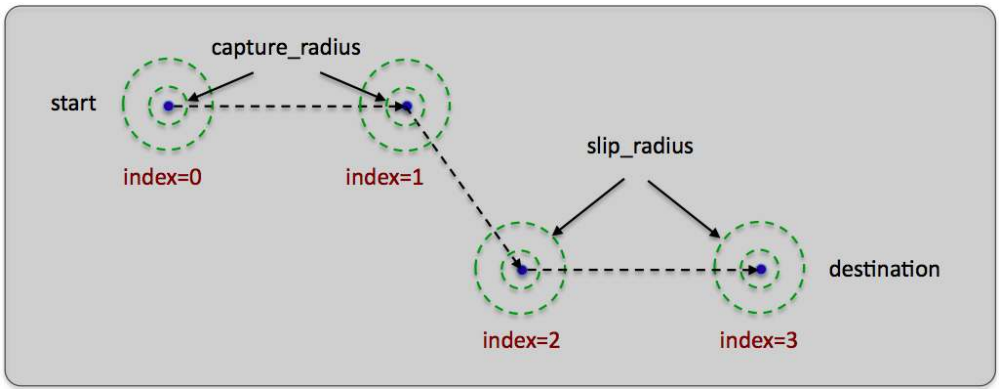

Figure 79: **The Waypoint behavior:** The waypoint behavior basic purpose is to traverse a set of waypoints. A capture radius is specified to define what it means to have achieved a waypoint, and a non-monotonic radius is specified to define what it means to be "close enough" should progress toward the waypoint be noted to degrade.

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 waypoint. This is to ensure 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 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, lawnmower pattern and so on.

23.1 Configuration Parameters

Listing 23.1: Configuration Parameters Common to All Behaviors.

`activeflag`:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
`condition`:	Specifies a condition that must be met for the behavior to be running. [more].
`duration`:	Time in behavior will remain running before declaring completion. [more].
`duration_idle_decay`:	When true, duration clock is running even when in the *idle* state. [more].
`duration_reset`:	A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. [more].

duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 23.2: Configuration Parameters for the Waypoint Behavior.

Parameter	Description
capture_radius:	The radius tolerance, in meters, for satisfying the arrival at a waypoint. The default is 3. Section 23.5.
capture_line:	If set to true, waypoint arrival will be achieved if the vehicle crosses the line perpendicular to the line approaching the waypoint. Default is false. Section 23.6.
crs_spd_zaic_ratio:	Specifies the relative weight of the course portion of the ZAIC. Valid range is $[1, 99]$. Default is 50.
cycleflag:	Optional MOOS variable-value pairs posted at end of each cycle through waypoints.
efficiency_measure:	Determines if leg by leg efficiency is to be measured and/or reported. The default is "off".
lead:	If this parameter is set, track-line following between waypoints is enabled. Section 23.7.
lead_damper:	Distance from trackline within which the lead distance is stretched out. Section 23.7.
lead_to_start:	Boolean indicating whether trackline following is applied to first waypoint. The default is false. Section 23.7.

order:	The order in which waypoints are traversed - "normal", "reverse", and with 14.3 or later, "toggle". Section 23.1.
points:	A colon separated list of x,y pairs given as points in 2D space, in meters. Section 23.1.
point:	A single x,y pair given as a point in 2D space, in meters. Section 23.1.
polygon:	An alias for points. Need not be a convex polygon. Section 23.1.
post_suffix:	A suffix tagged onto the WPT_STATUS, WPT_INDEX and CYCLE_INDEX variables.
radius:	An alias for capture_radius. Section 23.5.
repeat:	The number of *extra* times traversed through the waypoints. Or "forever". The default is "normal" meaning the points will be visited in the order listed. Section 23.4.
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. The default is 15 meters. Section 23.5.
speed:	The desired speed (m/s) at which the vehicle travels through the points. Accepts only non-negative numbers. The default is 0.
speed_alt:	An alternative desired speed (m/s) at which the vehicle travels through the points. Applies only when the use_alt_speed parameter is set to true. The default is -1, which indicates internally that it has not been set. (Introduced after release 15.5.)
use_alt_speed:	If true then attempt to use the altnernate speed set with the speed_alt parameter, if that speed is set to a non-negative value. The default is false. (Introduced after release 15.5.)
visual_hints:	Optional hints for visual properties in variables posted intended for rendering. Section 23.12.
wpt_status_var:	Optional MOOS variable for posting the waypoint status messages as described in Section 23.9.
wpt_index_var:	Optional MOOS variable for posting the waypoint index messages as described in Section 23.9.
wptflag:	Optional MOOS variable-value pairs posted after each waypoint has been reached.

Listing 23.3: Example Configuration Block.

```
Behavior = BHV_Waypoint
{
  // General Behavior Parameters
  // ---------------------------
  name         = transit               // example
  pwt          = 100                   // default
  condition    = MODE==TRANSITING      // example
  updates      = TRANSIT_UPDATES       // example

  // Parameters specific to this behavior
  // ------------------------------------
     capture_radius = 3                     // default
       capture_line = false                 // default
          cycleflag = COMMS_NEEDED=true     // example
               lead = -1                    // default
        lead_damper = -1                    // default
      lead_to_start = false                 // default
              order = normal                // default
             points = pts={-200,-200:30,-60} // example
         post_suffix = HENRY                // example
             repeat = 0                     // default
         slip_radius = 15                   // default
              speed = 1.2                   // default is zero
            wptflag = HITPTS = $(X),$(Y)    // example

       visual_hints = vertex_size  = 3            // default
       visual_hints = edge_size    = 1            // default
       visual_hints = vertex_color = dodger_blue  // default
       visual_hints = edge_color   = white        // default
       visual_hints = nextpt_color = yellow       // default
       visual_hints = nextpt_lcolor = aqua        // default
       visual_hints = nextpt_vertex_size = 5      // default
}
```

23.2 Variables Published

The below MOOS variables 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 described in Section 7.2.8.

- `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 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 Waypoint behavior will also publish any MOOS variables configured with the general behavior flags:

- `runflag`, `idleflag`, `activeflag`, `inactiveflag`, and `endflag`, described in Section 7.2.

as well as the flags defined locally for the Waypoint behavior:

- cyleflag and wptflag.

23.3 Specifying Waypoints with the points or point Parameter

There are a few formats supported for setting the list of waypoints. In the simplest case, when there is just a single point, as with the waypoint return behavior in the alpha mission, the following format may be used:

```
point = 0,0
```

Using the plural points parameter is also ok when there is only one point. Another common method is to specify a colon-separated list of comma-separate pairs. For example, also from the alpha mission:

```
points = 60,-40:60,-160:150,-160:180,-100:150,-40
```

Other formats supported are tied to the XYSegList and XYPolyogn classes. The former represents a set of vertices connected by line segments with a clear first and last vertex. The latter represents a convex polygon over vertices with the notion of the first or last vertex less clear in some cases. So for example, both of the following ways of setting waypoints are supported:

```
points = pts={60,-40:60,-160:150,-160:180,-100:150,-40}
points = format=lawnmower, x=0, y=40, height=60, width=180, lane_width=15
```

When loading a waypoint specification, the behavior will first attempt to build an XYSeglist from one of the above formats. Failing this, it will try to build an XYPolygon from the specification. Therefore any of the XYPologon string formats described in the Geometry documentation is supported. For example:

```
points = format=radial, label=foxtrot, x=0, y=40, radius=60, pts=6, snap=1
points = format=ellipse, label=golf, x=0, y=40, degs=45, pts=14, snap=1,
            major=100, minor=70
```

In specifying a list of points indirectly via the lawnmower, radial or ellipse patterns, the ”first” point in the list may be less predictable.

23.4 The order and repeat Parameters

The order of the parameters may be reversed with the order parameter, and the number of times the waypoints are traversed may be adjusted with the repeat parameter. An example specification:

```
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 and then 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 launch in the MALCONFIG mode. If the syntactic error is passed as part of a dynamic update (see Section 7.2.5), the change in waypoints will be ignored and the a warning posted to the BHV_WARNING variable. See the Geometry documentation for more methods for specifying sets of waypoints. The behavior can be set to repeat its waypoints indefinitely by setting repeat="forever". The point parameter may be used insted of points when there is only a single waypoint.

23.5 The capture_radius and slip_radius Parameters

The capture_radius parameter specifies the distance to a given waypoint 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 79. The slip_radius parameter specifies an alternative criteria for achieving a waypoint.

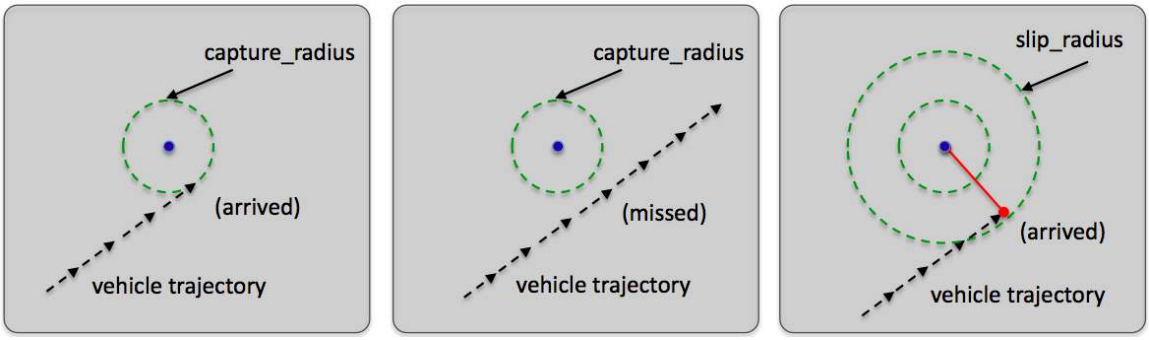

Figure 80: **The capture radius and slip radius:** (a) a successful waypoint arrival by achieving proximity less than the capture radius. (b) a missed waypoint likely resulting in the vehicle looping back to try again. (c) a missed waypoint but arrival declared anyway when the distance to the waypoint begins to increase and the vehicle is within the slip radius.

As the vehicle progresses toward a waypoint, the sequence of measured distances to the waypoint decreases monotonically. The sequence becomes non-monotonic 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 shown in Figure 80. 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 described in Section 23.9 below.

23.6 The capture_line Parameter

The capture_line parameter allows for an alternative or additional arrival criteria to be applied. When set to true, waypoint arrival will be achieved if the vehicle crosses the line perpendicular to the line segment approaching the waypoint from the previous waypoint. In the case of the first

waypoint, the "previous" waypoint is defined by the vehicle position when the behavior first becomes active.

When capture_line is set to true, both the capture line criteria *and* the capture and slip radius criteria apply. If either is satisfied, then arrival is achieved. If the capture_line criteria is set instead to absolute, then the capture and slip radius criteria is disabled, and only the capture line criteria is achieved.

When capture_line is set to false, only the capture and slip radius criteria are applied. A note of caution - setting the capture_line parameter to absolute essentially sets the capture and slip radius to zero. If the capture_line parameter is subsequently set to anything else (true or false), the capture and slip radius values also need to be explicitly re-set to non-zero values.

23.7 Track-line Following using the lead, lead_damper, and lead_to_start Parameters

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.

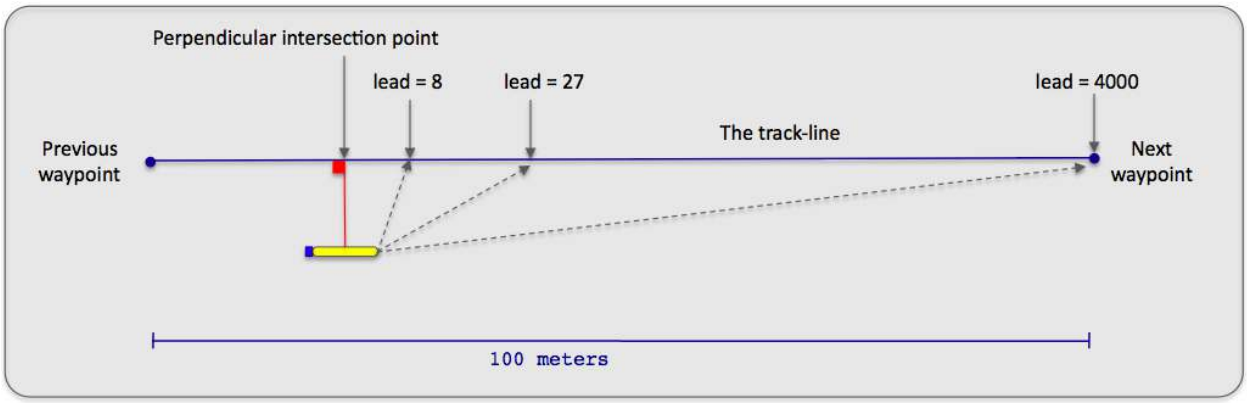

Figure 81: **The track-line mode:** When in track line mode, the vehicle steers toward a point o the track line rather than simply toward the next waypoint. The steering-point is determined by the lead parameter. This is the distance from the perpendicular intersection point toward the next waypoint.

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.

Normally, when trackline following is enabled, it is enabled only between the first waypoint and all succeess waypoints. When the parameter lead_to_start is set to true, trackline following is

attempted even to the first waypoint, by defining a trackline from the vehicle's present position when the behavior first enters the running mode.

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 trackline 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 trackline and reduce overshooting the trackline.

23.8 The wptflag Parameter

The wptflag parameter allows the user to speicify flags to be posted upon the achievement of each waypoint, where waypoint achievement is defined by the normal means, depending on how the user has configured the capture_radius, slip_radius or capture_line parameters. The wptflags are in the same format as idle, end, active, or inactive flags defined generally for IvP behaviors, i.e, they consist of a MOOS variable and value. For example:

```
wptflag = STATION_KEEP=true
```

The flags also support a handleful of macro expansions that allow the certain information about ownship or the next waypoint to be embedded in the posting that may not be available until run-time. These macros are:

- $[X] or $(X): Expanded to ownship's x position in local coordinates.
- $[Y] or $(Y): Expanded to ownship's x position in local coordinates.
- $[NX] or $(NX): Expanded to the x position of the next waypoint in local coordinates.
- $[NY] or $(NY): Expanded to the y position of the next waypoint in local coordinates.

Here are a couple examples of how the wptflag parameter may be useful in practice.

In the first case, in the example provided above, the mission may be configured to station-keep after arriving at each waypoint. This may be useful for a surface vehicle with a primary mission of collecting sensor measurements at periodic points in a survey area corresponding to waypoints. In this case, another station-keeping behavior is activated with the wptflag posting, which presumably temporarily idles the waypoint behavior while measurements are collected.

A second case utilizes the wptflag parameter to post the x-y position of the next waypoint in the list.

23.9 Variables Published by the Waypoint Behavior

The waypoint behavior publishes five variables for monitoring the performance of the behavior as it progresses: WPT_STAT, WPT_INDEX, CYCLE_INDEX, VIEW_POINT, VIEW_SEGLIST. The WPT_STAT contains information identifying the vehicle, the index of the current waypoint, the type of hits recorded for each waypoint, the distance to the current waypoint, and the estimated time of arrival to the current waypoint. Example output:

```
WPT_STAT = "vname=alpha,behavior=traverse1,index=0,hits=10/11,dist=43,eta=23"
```

The "hits=10/11" component in the above example indicates that, of the 11 waypoint arrivals achieved so far, 10 of them were achieved by meeting the capture radius criteria, and one of them was achieved by meeting the nonmonotonic radius criteria.

The WPT_INDEX variable simply publishes the index of the current waypoint. This is a bit redundant since this information is also in the WPT_STAT posting, but this variable is logged as a numerical variable, not a string, and facilitates the plotting of the index value as a step function in post mission analysis tools. The CYCLE_INDEX variable publishes the number of times the behavior has traversed the entire set of waypoints. The behavior may be configured to post the information in these three variables using alternative variables of the user's liking:

```
post_mapping  = WPT_STAT, MY_WPT_STATUS_VAR
post_mapping  = WPT_INDEX, MY_WPT_INDEX_VAR
post_mapping  = CYCLE_INDEX, MY_CYCLE_INDEX
```

or, to suppress the reports completely:

```
post_mapping  = WPT_STAT, SILENT
post_mapping  = WPT_INDEX, SILENT
post_mapping  = CYCLE_INDEX, SILENT
```

Further posts to the MOOSDB can be configured to be made at the end of each cycle, that is, after reaching the last waypoint. Normally, if the repeat parameter remains at its default value of zero, then the end of a cycle and completing are identical and endflags can be used to post the desired information. However, when the behavior is configured to repeat the set of waypoints one or more times before completed, the cycleflag parameter may be used to post one or more variable-value pairs at the end of each cycle. Likewise, if the repeat parameter is zero, but the behavior is set with perpetual=true, the cycle flags will posted each new time that the behavior completes.

The VIEW_POINT and VIEW_SEGLIST variables provide information consumable by a GUI application such as pMarineViewer or alogview for rendering the set of waypoints traversed by the behavior (VIEW_SEGLIST) and the behavior's next waypoint (VIEW_POINT). These two variables are responsible for the visual output in the Alpha Example Mission in Section 16 in Figure 59.

23.10 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 functions rendered in Figure 82 are built in the first manner.

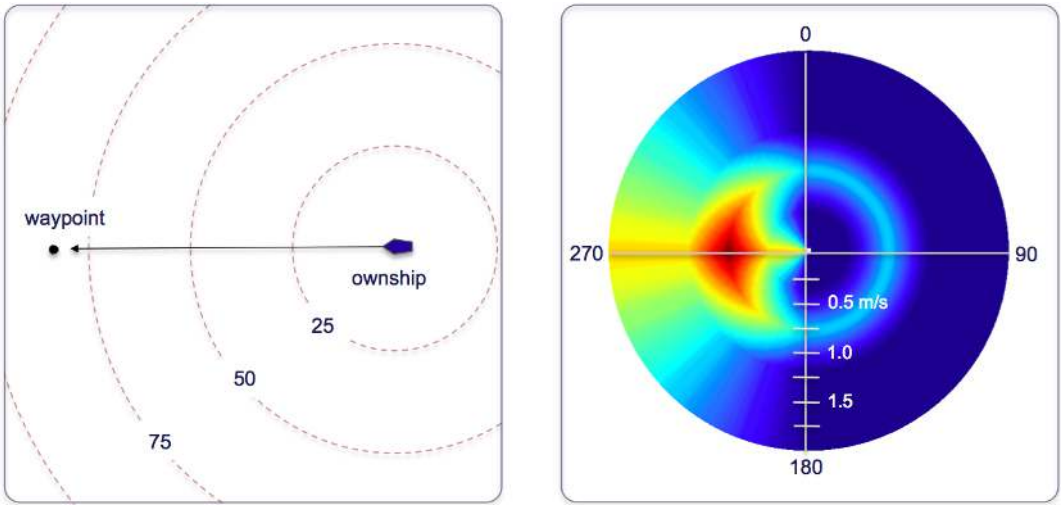

Figure 82: **A waypoint objective function:** The objective function produced by the waypoint behavior is defined over possible heading and speed values. Depicted here is an objective function favoring maneuvers to a waypoint 270 degrees from the current vehicle position and favoring speeds closer to the mid-range of capable vehicle speeds. Higher speeds are represented farther radially out from the center.

23.11 Further Clarification on the `repeat` vs. `perpetual` Parameter

It's worth clarifying the difference in usage and effect between the `repeat` parameter, which is specific to the Waypoint behavior, and the `perpetual` parameter which is defined for all helm behaviors. Normally when a behavior completes, it is entered into the completed stated, never again to be called upon by the helm. See Section 6.5.3 for more on behavior run states. It is up to the behavior implementor to decide what it means to be complete, and the implentor typically invokes the `setComplete()` function from within the code. See Section 7.5.1 for more on this function. In the case of the Waypoint behavior, completion by default occurs when the vehicle has hit all its waypoints. By setting `perpetual=true` the behavior, upon hitting all its waypoints, still invokes the `setComplete()` function, which causes it to post its endflags, but it does not enter the completed state. This feature is used, for example, in the Alpha example mission to allow the behavior to repeatedly return to the start point and be re-deployed, and later return to its start point again using the same Waypoint behavior.

The `repeat` parameter is used to change the criteria for completion. Whereas the normal criteria for completion is hitting all waypoints once, using `repeat=N` changes the criteria to be hitting all waypoints `N+1` times. A few rules of thumb may be helpful in keeping things straight:

- When `perpetual=false`, the default setting, the behavior will permanently enter the completed state once it has hit all its waypoints.
- When `perpetual=true` and the behavior does not post endflags leading to its run conditions being unsatisfied, the behavior will repeat its waypoints indefinitely.
- When `perpetual=true` and it does indeed post endflags that lead to its run conditions being unsatisfied, it will remain in a running state until all its waypoints are hit. If the `repeat` parameter is used, it won't post its endflags until it has repeated all waypoints the specified

number of times. During the course of traversal, the cycleflag posts will be made each time it has completed the set of waypoints. Upon completion, it will post its endflags and reset its cycle counter.

- By setting repeat=N, where N>0, the perpetual parameter is automatically set to true.

23.12 Visual Hints Defined for the Waypoint Behavior

Although the primary output of the Waypoint behavior is an IvP Function, a number of visual properties are also published for convenience in mission monitoring. This includes (a) the set of waypoints, (b) the point the behavior is presently progressing toward, and (c) the trackpoint, if trackpoint following is enabled. These visual artifacts have default properties in size and color that may be altered to the user's preferences. These preferences are configurable through the visual_hints parameter. Each parameter below is used in the following way by example:

```
visual_hints = vertex_size=3, edge_size=2
visual_hints = vertex_color=khaki
```

- vertex_size: The size of vertices rendered in the loiter polygon. The default is 1.
- edge_size: The width of edges rendered in the loiter polygon. The default is 1.
- vertex_color: The color of vertices rendered in the loiter polygon. The default is "dodger_blue".
- edge_color: The color of edges rendered in the loiter polygon. The default is "white".
- label_color: The color of label for set of waypoints. The default is "white".
- nextpt_color: The color of the point rendered as the present next waypoint. The default is "yellow".
- nextpt_lcolor: The color of the label for the point rendered as the present next waypoint. The default is "aqua".
- nextpt_vertex_size: The size of the vertex for the point rendered as the present next waypoint. The default is 1.

Rendering of vertices or the next waypoint may be shut off with a size of zero, and labels may be shut off with the special color "invisible". For a list of legal colors, see Appendix 21.

24 The AvoidObstacle Behavior

The AvoidObstacleV24 behavior is designed to produce IvP functions for avoiding a single convex polygon obstacle. It is typically configured as a behavior *template* in the helm and works with an *obstacle manager* with the latter spawning new behavior instances for each known obstacle when the vehicle is sufficiently close to the obstacle. In the example in Figure **??** below, two behavior instances are spawned, one for each of the two shown obstacles. Each instance starts exactly in the same way, evolving in state differently as it maneuvers around its respective obstacle. As an obstacle recedes behind the vehicle, its behavior at some point is completed, deleted and forgotten.

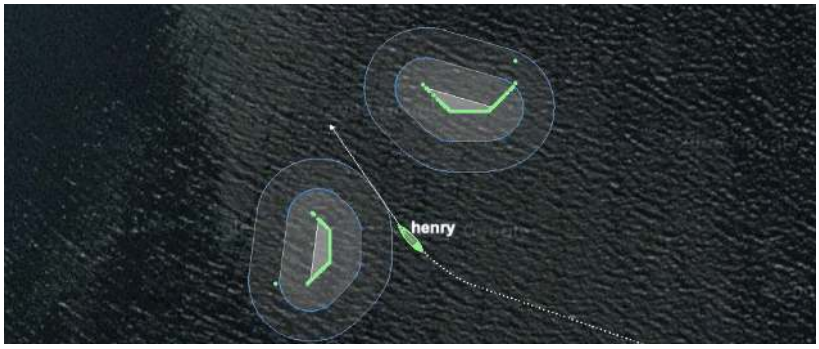

Figure 83: **AvoidObstacleV24 Behavior**: The operation area

24.1 Overview

The `AvoidObstacle` behavior acts upon a single obstacle given by a convex polygon, Figure 84. When a robot has multiple obstacles, multiple versions of the behavior are spawned, one for each obstacle. During the course of a mission, it is expected that multiple such behavior instances will be spawned and deleted.

The `AvoidObstacle` behavior is typically configured as a template for spawning instances as obstacles emerge. In less common cases, one or more static behaviors may be configured, instantiated at mission launch time, with a obstacle of a known position and location. Both use cases will be described. In the case of dynamically spawned behaviors, the helm and the AvoidObstacle behavior work in coordination with an obstacle manager which sits between the helm and sensor processing applications to reason about obstacles and post messages at the approprite time to the helm, for generating new AvoidObstacle behaviors. The obstacle manager currently used is another MOOS app called `pObstacleMgr` and is documented separately. The interface that triggers AvoidObstacle instances will be described.

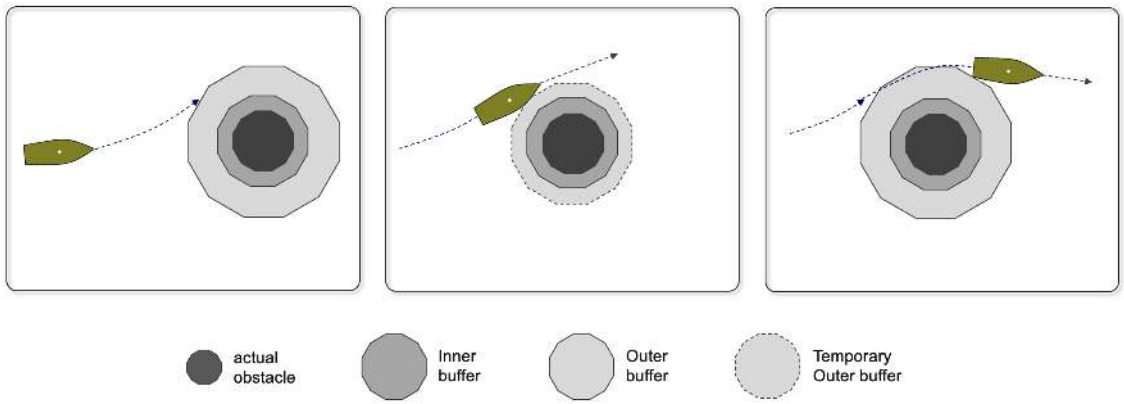

Figure 84: **The AvoidObstacle Behavior**: The behavior reasons about a single convex polygon obstacle. It uses an inner buffer and outer buffer, treating the inner buffer as essentially an extension of the obstacle, and the outer buffer as a region where it is preferred not to be in.

The AvoidObstacle behavior is not a path planner. It is a behavior that works in conjuction with a planned path, typically being executed by the Waypoint behavior. The AvoidObstacle behavior is typically used for avoid locally sensed obstacles like buoys, stationary vessels, or any other hazard that may be detected or known about beforehand. The AvoidObstacle behavior, like any IvP Helm behavior, is capable of accepting dynamic updates. This includes obstacle size and location. So the AvoidObstacle behavior is capable of dealing with moving obstacles including vessels. However, when a vessel is known to be a vessel, and has a known position, heading, and speed, the AvoidCollision or AvdColregs behaviors are more appropriate.

24.2 The Nature and Origin of Obstacles

The AvoidObstacle behavior may be configured *statically* or *dynamically*. In a static behavior the obstacle is provided in the mission configuration using the `polygon` configuration parameter. For example:

```
polygon = 60,-40 :  60,-160 :  150,-160 :  180,-100 :  150,-40
```

The vertices may also be specified indirectly using one more supported patterns. For example an octogon polygon, perhaps for avoiding a 1 meter sized buoy at a local coordinates `(45,90)` could be given as:

```
polygon = format=radial, x=45, y=90, radius=2, pts=8
```

In a dynamic behavior, the polygon specification is typically provided by an external source, typically the obstacle manager. In the mission file this behavior leaves the `polygon` parameter unspecified, but does specify a MOOS variable for receiving updates. For example:

```
templating = spawn
updates = OBSTACLE_ALRERT
```

A new AvoidObstacle behavior

In both cases, they may be altered during run time.

24.3 Configuration Parameters

Listing 24.4: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 24.5: Configuration Parameters for the AvoidObstacle Behavior.

Parameter	Description

completed_dist:	Range to contact outside of which the behavior completes and dies. The default is 500 meters.
max_util_cpa_dist:	Range to contact outside which a considered maneuver will have max utility. Section **??**. The default is 75 meters
min_util_cpa_dist:	Range to contact within which a considered maneuver will have min utility. Section **??**. The default is 10 meters.
pwt_inner_dist:	Range to contact within which the behavior has maximum priority weight. Section 24.6. The default is 50 meters.
pwt_outer_dist:	Range to contact outside which the behavior has zero priority weight. Section 24.6. The default is 200 meters.
use_refinery:	If true, the behavior will produce an optimized objective function that is faster to produce, uses a smaller memory footprint, and contributes to faster helm solution time. The default is false, simply for continuity with prior releases, but there is no downside to enabling this feature. Section **??**.
polygon:	A convex polygon representing the obstacle. Section **??**.
poly:	Same as polygon.
rng_flag:	A flag to be posted on all iterations. It may be conditioned on a range threshold. Section **??**.
cpa_flag:	A flag to be posted upon reaching the closest point of approach to the obstacle. Section **??**.
visual_hints:	A request to override the default visual parameters of the rendered obstacle or its buffer region. Section **??**.
allowable_ttc:	The allowable time-to-collision, in seconds, within which a candidate trajectory will begin to be penalized. Section **??**. id:

Listing 24.6: Example Configuration Block.

```
Behavior = BHV_AvoidObstacleV24
{
  // General Behavior Parameters
  // ---------------------------
  name       = avdob_
  pwt        = 300
  condition  = DEPLOY = true
  templating = spawn
  updates    = OBSTACLE_ALERT

  // Parameters specific to this behavior
  // ------------------------------------
       allowable_ttc = 20   // default
      pwt_outer_dist = 50   // default
      pwt_inner_dist = 10   // default
      completed_dist = 60   // default
   min_util_cpa_dist = 8    // default
   max_util_cpa_dist = 16   // default
        use_refinery = true // default is false

    visual_hints = obstacle_edge_color   = white     // default
    visual_hints = obstacle_vertex_color = gray60    // default is white
    visual_hints = obstacle_vertex_size  = 1         // default is white
    visual_hints = obstacle_fill_color   = gray60    // default
    visual_hints = obstacle_fill_transparency = 0.7  // default

    visual_hints = buffer_min_edge_color = gray60    // default
    visual_hints = buffer_min_vertex_color = blue    // default is dodger_blue
    visual_hints = buffer_min_vertex_size = 1        // default
    visual_hints = buffer_min_fill_color = gray70    // default
    visual_hints = buffer_min_fill_transparency = 0.25  // default

    visual_hints = buffer_max_edge_color = gray60         // default
    visual_hints = buffer_max_vertex_color = dodger_blue  // default
    visual_hints = buffer_max_vertex_size = 0             // default is 1
    visual_hints = buffer_max_fill_color = gray70         // default
    visual_hints = buffer_max_fill_transparency = 0.1     // default
}
```

24.4 Variables Published

The below MOOS variables 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 described in Section 7.2.8.

- AVD_OB_SPAWN: A request to the obstacle manager specifying the conditions for obstacle alerts.
- NOTED_RESOLVED: A posting made when the behavior notes an obstacle has been reported be resolved by the obstacle manager.
- VIEW_POLYGON: A polygon rendering of either the obstacle, inner, or outer buffer.

24.5 Configuring and Using the AvoidObstacle Behavior

The AvoidObstacle behavior produces an objective function based on the relative position and trajectory between the vehicle and its obstacle.

The objective function is based on applying a utility to the calculated closest point of approach (CPA) for a candidate maneuver. The user may configure a priority weight, but this weight is typically degraded in proportion to increasing range to the obstacle. The behavior may be configured for avoidance with respect to an obstacle known prior to the start of the mission, or it may be configured to spawn a new instance upon demand as obstacles become known through the obstacle manager.

24.6 Specifying the Behavior Priority Weight Policy

The AvoidObstacle behavior may be configured to increase its priority as it closes range to the obstacle. The priority weight specified in its configuration represents the *maximum* possible priority applied to the behavior, presumably in close range to the obstacle. The range at which this maximum priority applies is specified in the `pwt_inner_dist` parameter. Likewise, the `pwt_outer_dist` parameter specifies a range to the obstacle where the priority weight becomes zero, regardless of the priority weight specified in the configuration file.

So the *current priority* will always be between zero and the maximum priority set in the behavior `priority` configuration parmeter. To be more precise:

Current Priority =

- 0 if current range to obstacle is greater than or equal to `pwt_outer_dist`
- 100 if current range to obstacle is less than or equal to `pwt_inner_dist`
- otherwise ((`pwt_outer_dist` - current range) / (`pwt_outer_dist` - `pwt_inner_dist`)) * priority

This relationship is shown in Figure 85.

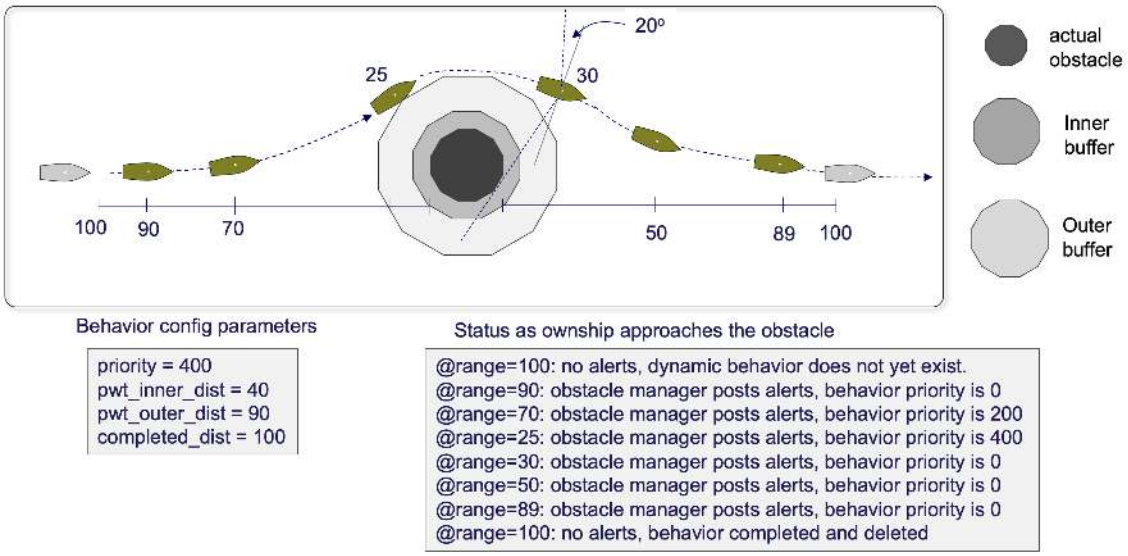

Figure 85: **Scaling priority weights based on ownship range to obstacle**: The range between the vehicle and the obstacle affects whether the behavior is spawned, is active and with what priority weight. Beyond the range specified by `pwt_outer_dist` the behavior will have a zero priority weight, if it even exists. Within the range of `pwt_outer_dist`, the behavior is active with a non-zero priority weight growing as the obstacle comes closer. Within the range of `pwt_inner_dist`, the behavior is active with 100% of its configured priority weight.

The example shown below in Figure **??** shows the effect of the `pwt_outer_dist` parameter. The vehicle on the left is proceeding east, oblivious to the two approaching vessels. The two westbound vessels, `ben` and `cal` are simulated exactly on top of one another. They are oblivious to one another, but will use the collision avoidance behavior to avoid the eastbound vessel, abe. The only difference between `ben` and `cal` is that `cal` begins winding up its priority weight at 80 meters range to `abe`, as opposed to 30 meters for `ben`. The short simulation shows the resulting difference in trajectory.

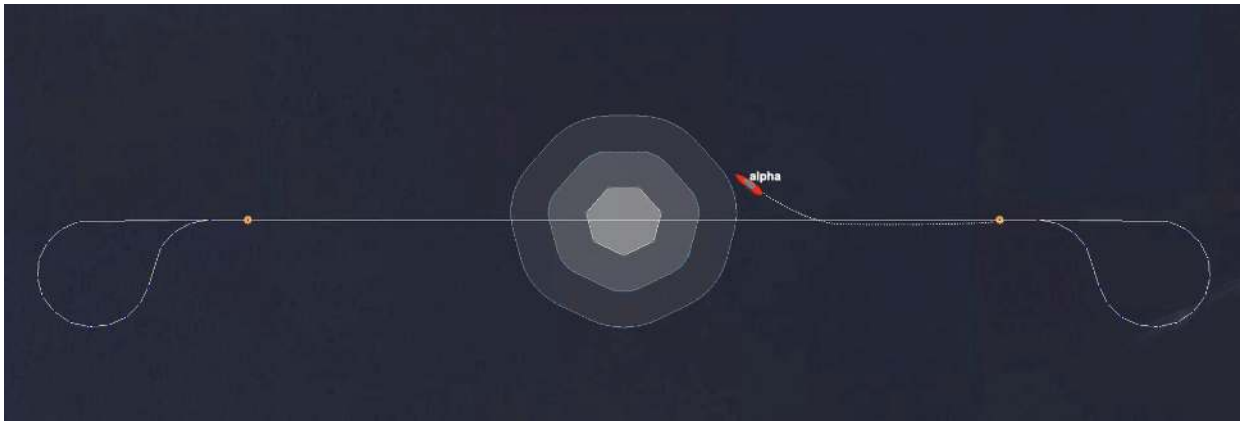

Figure 86: A vehicle approaches an obstacle. The physical obstacle is shown by the inner polygon. The middle inner polygon represents a buffer region around the physical obstacle regarded with the same severity of consequence as hitting the physical obstacle. The outer polygon represents the all clear boundary. The buffer region of the outer polygon has a linearly scaled utility.

video:(0:15): https://vimeo.com/856588988

By default, the priority weight decreases linearly between the two depicted ranges. The `pwt_grade` parameter allows the degradation from maximum priority to zero priority to fall more steeply by setting `pwt_grade=quadratric`.

Visual Configuration Options

The visual artifacts generated by the `AvoidCollisionV24` behavior are related to the three polygon regions representing (a) the obastacle, (b) the inner buffer, and (c) outer buffer polygons. There are also seven visual aspects for each polygon that may be modified: (1) the vertex size, (2) the edge size, (3) the vertex color, (4) the edge color, (5) the label color, (6) the polygon fill color, and (7) polygon transparency.

By setting `edge_color=green`, this sets the default edge color for all three polygons. This can be overruled for say the outer buffer polygon by setting `buff_max_edge_color=white`. If the latter is left unspecified, the default edge colore is used. In the example lines below, the first group of lines are the default aspects for all polygons. The second block of lines pertain to the obstacle polygon and overrule the default aspects. The third block of lines pertain to the inner buffer polygon, and the final block of lines pertain ot the outer buffer polygon. All lines below represent the defaults used by the `AvoidCollisionV24` behavior if no visual hints were provided in the behavior configuration.

```
m_hints.setMeasure("vertex_size", 0);
m_hints.setMeasure("edge_size", 1);
m_hints.setColor("vertex_color", "gray50");
m_hints.setColor("edge_color", "gray50");
m_hints.setColor("fill_color", "off");
m_hints.setColor("label_color", "white");

m_hints.setColor("obst_edge_color", "white");
m_hints.setColor("obst_vertex_color", "white");
m_hints.setColor("obst_fill_color", "gray60");
m_hints.setMeasure("obst_vertex_size", 1);
m_hints.setMeasure("obst_fill_transparency", 0.7);

m_hints.setColor("buff_min_edge_color", "gray60");
m_hints.setColor("buff_min_vertex_color", "dodger_blue");
m_hints.setColor("buff_min_fill_color", "gray70");
m_hints.setColor("buff_min_label_color", "off");
m_hints.setMeasure("buff_min_vertex_size", 1);
m_hints.setMeasure("buff_min_fill_transparency", 0.25);

m_hints.setColor("buff_max_edge_color", "gray60");
m_hints.setColor("buff_max_vertex_color", "dodger_blue");
m_hints.setColor("buff_max_fill_color", "gray70");
m_hints.setColor("buff_max_label_color", "off");
m_hints.setMeasure("buff_max_vertex_size", 1);
m_hints.setMeasure("buff_max_fill_transparency", 0.1);
```

The posting of the inner and outer polygons can be disabled simply with:

```
draw_buff_max_poly = false
draw_buff_min_poly = false
```

24.7 Flags and Macros

25 The OpRegionV24 Behavior

The `OpRegionV24` behavior was created to combine and replace the `OpRegion` and `OpRegionRecover` behaviors. The former was simply a pass-fail check to ensure operating constraints are not exceeded, resulting in an all-stop if they were. No attempt was made in the `OpRegion` behavior to influence the vehicle to avoid exceeding the operating constraints. On the other hand, in the `OpRegionRecover` behavior, the goal is indeed to influence the vehicle to return to an the operation area once it has exited. The `OpRegionV24` behavior can perform both functions of the older behaviors. It works with the notion of three polygons, the *core*, *save* and *halt* polygons shown in Figure 87 below.

Figure 87: **OpRegionV24 Behavior Regions**: The operation area is defined by the *core* polygon. The buffer region is defined by the *save* polygon. The all-stop region is the area outside the *halt* polygon.

The *core* polygon is the main operation area, presumably where the mission will be contained under normal circumstances. The *save* polygon extends the core polygon and once the vehicle is outside this polygon buffer zone, this behavior will begin to produce an objective function influencing the the vehicle back into the save polygon. The *halt* polygon extends the buffer region even further. If the vehicle breaches the halt polygon, an all-stop will be enforced.

25.1 Intended Mission Use

The `OpRegionV24` behavior is typically running in every helm mode of a mission, to guard against exceeding certain safety boundaries. The operation area is one, but there are three other bounds: (1) maximum depth, (2) maximum altitude, and (3) maximum mission time. The first two are only relevant for underwater vehicles, but the maximum mission time is another way to ensure that, if all other safeguards fail, the mission can be halted after a certain duration. For an underwater vehicle that has disappeared, the maximum time can significantly reduce the search box area.

Defining and constraining operation area is the most visual aspect of the `OpRegionV24` behavior. As shown in Figure 87, three regions or polygons are supported. The *core*, *save*, and, *halt* polygons.

The former is simply the area declared to be the intended operation area. Even if there are no hazards around this area, this conveys an important element of the mission plan to the operators and others observing a deployment. The *save* polygon usually subsumes the core polygon by some buffer distance. This conveys the degree to which deviation from the core area may be tolerated. For vessels that breach the save polygon, the `OpRegionV24` behavior will produce an IvP Function to influence the vehicle back within the save polygon. The *halt* polygon is a final boundary, beyond which the behavior will produce an all-stop message. Figure 88 shows the effects of breaching the save and halt polygons.

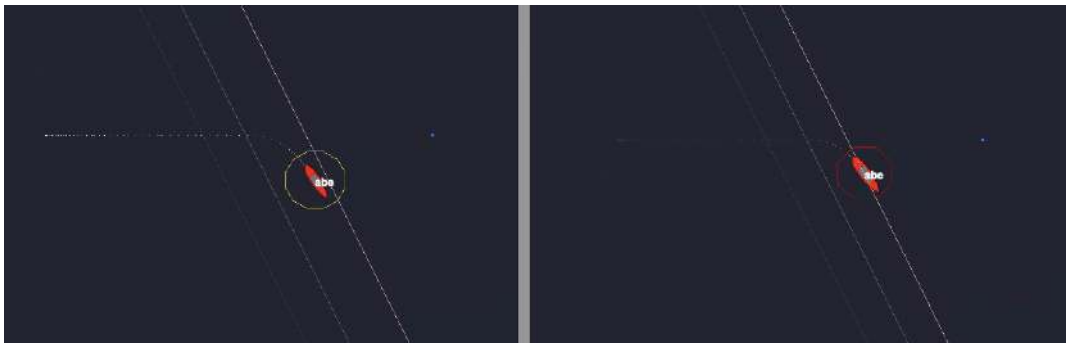

Figure 88: **OpRegionV24 Behavior**: Left: A vehicle exits the operating area and *save polygon* buffer zone, and amends its course to return to the the buffer zone. Right: A vehicle exits the buffer zone and is unable to complete a turn before breaching the halt polygon. An all-stop state results.

25.2 The Core Algorithm

The `OpRegionV24` behavior may be configured to use none or all of the possible constraints, e.g., (1) max time, (2) max depth, (3) min altitude, (4) halt polygon, or (5) save polygon. The `onRunState()` algorithm will first check for all the constraints that would result in an all-stop condition. If it passes these, then the vehicle is checked for containment of the save polygon. An IvP function is only generated if the behavior is attempting to return the vehicle to the save polygon.

OnRunState() Flow Chart

OnRunState() Pseudo Code

Algorithm 1: OpRegionV24 Behavior Main Loop

```
1: procedure OnRunState()
2:     V ← updatePositions()                                   ▷ Step 1
3:     if (max_time > 0) and (time_elapsed > max_time) then    ▷ Step 2
4:         postBreachedTimeFlags()                             ▷ Step 3
5:         all_stop ← true
6:     else if (max_depth > 0) and (depth > max_depth) then    ▷ Step 4
```

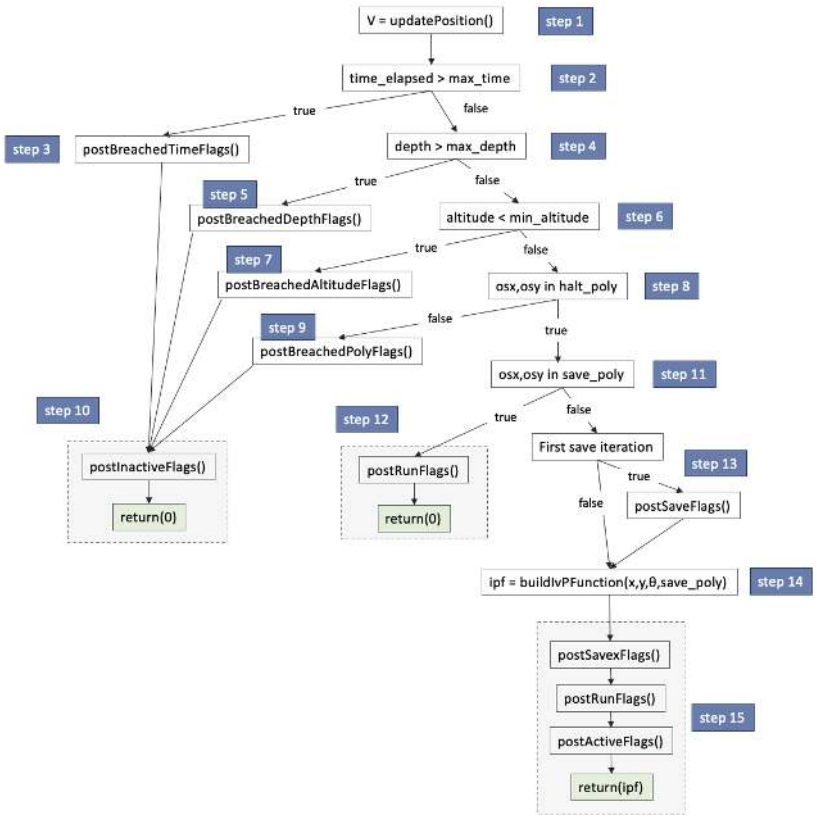

Figure 89: **OpRegionV24 Behavior Loop Flow Chart:** Each iteration of the OpRegionV24 behavior will proceed through the steps shown, with branching determined by sensed events and the behavior configuration. Details of each step are discuss in their own sections as indicated.

```
7:      postBreachedDepthFlags()                                          ▷ Step 5
8:      all_stop ← true
9:  else if (min_altitude > 0) and (altitude < min_altitude) then          ▷ Step 6
10:     postBreachedAltitudeFlags()                                       ▷ Step 7
11:     all_stop ← true
12: else if halt_poly.isConvex() and (x_os, y_os ∉ halt_poly) then        ▷ Step 8
13:     postBreachedPolyFlags()                                           ▷ Step 9
14:     all_stop ← true
15: end if
16: if (all_stop = true) then                                             ▷ Step 10
17:     postInactiveFlags()
18:     return 0
19: end if
20: if save_poly.isConvex() and (x_os, y_os ∈ save_poly) then             ▷ Step 11
21:     postRunFlags()                                                    ▷ Step 12
22:     return 0
23: end if
24: if first_save_iteration then                                          ▷ Step 13
25:     postSaveFlags()
26: end if
```

27: $ipf \leftarrow$ buildIvPFunction($x_{os}, y_{os}, \theta_{os}$,save_poly) ▷ Step 14
28: postRunFlags() ▷ Step 15
29: postActiveFlags() ▷ Step 15
30: **return** *ipf*
31: **end procedure**

In Step 1, ownship information is updated, including the ownship local x-y position, heading, speed, and depth if the vehicle is an underwater vehicle, and altitude if the underwater vehicle is equipped with an altitude sensor.

In Step 2, if a maximum mission time is set, with the parameter max_time, the duration of the mission is checked against this maximum. The duration clock is started on the first helm iteration, i.e., when the helm first enters the *drive* state. More discussion on this topic in Section 25.3.3. If the maximum time is exceeded, in Step 3, any flags tied to the timeout event, set with the breached_time_flag parameter, are posted.

In Step 4, if a maximum depth is set, with the parameter max_depth, the current depth of the vehicle is checked against this maximum. More discussion on this topic in Section 25.3.3. If the maximum depth is exceeded, in Step 5, any flags tied to the breached depth event, set with the breached_depth_flag parameter, are posted.

In Step 6, if a minimum altitude is set, with the parameter min_altitude, the current altitude of the vehicle is checked against this minimum. More discussion on this topic in Section 25.3.3. If the minimum altitude is exceeded, in Step 7, any flags tied to the breached altitude event, set with the breached_altitude_flag parameter, are posted.

In Step 8, if a halt polygon is set, the current position of the vehicle is checked for containment in the polygon. If the vehicle is outside the polygon, in Step 9, any flags tied to the breached halt polygon event, set with the breached_poly_flag parameter, are posted. Two further notes: (1) the halt polygon can be set either by explicitly defining a convex polygon with the halt_poly parameter, or relative to the core polygon with the halt_dist parameter. See Section 25.3.1. (2) a breach of the halt polygon is typically not possible until the vehicle has first been within the halt polygon, and typically the breach will not be declared until some time has passed being continually outside the polygon. The defaults and configuration options for these aspects are described in Section 25.3.2.

In Step 10, if any of the four constraints on time, depth, altitude or halt polygon have been breached, the behavior will post any defined inactive flags, post an all-stop indicator, and return without providing an IvP function. This will normally result in the helm entering an all-stop state and will be the last iteration of this behavior.

In reaching Step 11, all dire halt conditions have been checked against. If a valid convex *save* polygon has been provided, *and* the current position of the vehicle is contained in the polygon, then Step 12 will be invoked. In this step, any configured run flags will be posted and the function will return without providing an IvP function.

Reaching Step 13 means the vehicle has breached a valid convex save polygon. If this is the first iteration of a breach, then any configured *save* flags will be posted. Finally, in Step 14, an IvP function will be generated to influence the vehicle on a trajectory back toward the save polygon. The formation algorithm for this IvP function is discussed in Section 25.4. In Step 15 any *savex*

flags, run flags or active flags are posted before finally returning the generated IvP function. The *savex* flags are published on every iteration when the behavior detects the vehicle to be outside the save poly. The *save* flags are only published on the first such iteration.

25.3 Configuration and Operation Mechanics

25.3.1 Configuring the Polygon Regions

The three operation areas are the *core* polygon, the *save* polygon, and the *halt* polygon. They are configured with the following parameters and examples, from Figure 87.

```
core_poly = pts={-70.38,-49.97:56.12,10:90.39,-62.28:-36.11,-122.25}
save_poly = pts={-74.9,-52.1:-75.4,-49.6:-74.5,-47.1:-72.5,-45.5:54,14.5: \
                 56.5,15:60.6,12.1:94.9,-60.1:95.4,-62.7:94.5,-65.1:      \
                 92.5,-66.8:-34,-126.8:-36.5,-127.2:-40.6,-124.4}
halt_poly = pts={-79.4,-54.3:-80.3,-49.2:-78.6,-44.3:-74.7,-40.9:51.8,19:    \
                 56.9,20:65.2,14.3:99.4,-58:100.4,-63.1:98.6,-68:94.7,-71.3: \
                 -31.8,-131.3:-36.9,-132.2:-45.1,-126.5}
```

These three polygons, in this case, are more or less evenly concentric, and the same configuration could have been achieved with:

```
core_poly = pts={-70.38,-49.97:56.12,10:90.39,-62.28:-36.11,-122.25}
save_dist = 5
halt_dist = 10
```

If the `save_dist` parameter is provided, then (a) the `core_poly` parameter must be provided and be a convex polygon, (b) the contents of the `save_poly` parameter will be ignored if it was provided, and (c) the value of the `save_dist` parameter must not be negative. If it is zero, then the save polygon is essentially the same as the core polygon.

If the `halt_dist` parameter is provided, then (a) the `core_poly` parameter must be provided and be a convex polygon, (b) the contents of the `halt_poly` parameter will be ignored if it was provided, and (c) the value of the `halt_dist` parameter must not be negative. If it is zero, then the halt polygon is essentially the same as the core polygon.

Both the `save_dist` and `halt_dist` parameters are distances relative to the core polygon. If both the `save_dist` and `halt_dist` parameters are provided, but the value of the `save_dist` parameter is larger, then it will be automatically reduced to the value of the `halt_dist` parameter, without a posted warning.

25.3.2 Halt Polygon Triggers

By default, the halt polygon is not enabled until the vehicle has first entered the halt polygon. This feature can be reversed by setting:

```
trigger_on_poly_entry = false
```

By requiring the vehicle to first enter the halt poly, this allows a vehicle to be launched and transit to the operation area, before regarding a halt poly breach as an all-stop event.

Due to the relatively common occurrence in early years of a spurious navigation solution, there is also a requirement that the vehicle enter the halt polygon for some duration of time before it is considered entered. By default this is 1 second. Likewise the vehicle must be outside the halt polygon for a duration of 0.5 seconds before a halt poly breach is declared. These values can be changed with:

```
trigger_entry_time = 2  // Default is 1
trigger_exit_time = 3   // Default is 0.5
```

25.3.3 Mission Time, Depth, and Altitude Operation Envelope

While the most outwardly visible component of the OpRegionV24 behavior relates to its position in the x-y plane relative to the containment polygons, it also supports operational constraints on overall mission time, depth and altitude. On most commercial platforms where MOOS-IvP is running in the backseat of a mature front-seat computing system, these aspects are also monitored and constrained on the front-seat. Use of the OpRegionV24 behavior not only provides a redundancy, but also an option for the mission planner to configure the backseat operating envelope within the front-seat envelope. This could allow the backseat MOOS-IvP autonomy the opportunity to make a last-ditch effort to correct the vehicle trajectory before a front-seat all-stop is triggered.

The three relevant parameters are: `max_time`: The maximum allowable time (in seconds) that the helm is allowed to run. The clock starts when the pHelmIvP process first takes control, i.e., enters the `DRIVE` state. If no maximum time is specified, then no time checks are made. `max_depth`: The maximum allowable depth of the vehicle (in meters). If no depth is provided, no depth checks are made. `min_altitude`: The minimum allowable altitude of the vehicle (in meters). If no altitude is provided, no altitude checks are made.

For any of the operation envelope constraints, a corresponding macro is supported for conveying status. These are, `ALT_LEFT`, `DEPTH_LEFT`, and `TIME_LEFT`. For example, the following configuration would enable the remaining depth to be published on each behavior iteration:

```
runx_flag = REMAINING_DEPTH=$[DEPTH_LEFT]
```

The choice of the `runx_flag` ensures that it is published on every iteration while in the run state, not just when the behavior enters the run state. Note also that the `*_LEFT` flags publish as integer values unless within 10 meters or seconds, in which case they are published with floating point values.

25.3.4 Resetting a Breach Condition

The behavior may be configured to accept a reset after one of the breach conditions occur. This is done by publishing to the MOOS variable configured with the behavior's `updates` parameter. If this variable is say `OPR_UPDATES`, then the following would accomplish the reset:

```
OPR_RESET = reset = true
```

This configuration parameter is somewhat non-sensical as an initial configuration parameter. Typically it would be sent mid-mission by an operator when or if there has been a breach and the operator deems it safe to reset the behavior.

Term	ZAIC Param	Behavior Config	Meaning
$u(v)$	n/a	n/a	Utility of candidate speed v
$\hat{v}$	`summit`	`save_spd`	Set speed, with top utility=100
v_p	`peak_width`	hard-coded	ZAIC off-peak range
v_b	`base_width`	hard-coded	ZAIC next off-peak range
u_{med}	`summit_delta`	hard-coded	Utility at ZAIC off-peak range

Table 24: The terms for the speed utility function of the `OpRegionV24` behavior.

25.4 IvP Function Formulation

Once the vehicle has been detected within the save polygon, a subsequent exit from the save polygon will result in the production of an IvP function designed to return the vehicle to the polygon. The IvP function created for the `BHV_OpRegionV24` behavior is comprised of an independent utility function related to desired speed, and one related to desired heading. In a typical mission configuration, the `OpRegionV24` behavior may be the only behavior running while recovering to the save polygon, and thus the formulation of the IvP function may be overkill. However, this behavior, like most if not all IvP behaviors, assumes there may be other behaviors vying for influence, e.g., collision or obstacle avoidance while recovering. Therefore the `OpRegionV24` behavior produces the IvP function described here.

Speed Utility

When the behavior is in the mode of recovering, and returning back to the save polygon, the presumption is that a moderation of safe speed is called for while turning the vehicle around back to the polygon. This speed is configurable with the parameter `save_speed` with a default of 1 meter per second. Speeds a bit higher or lower are tolerated up to a point, as the IvP function described below indicate. Another consideration is to allow enough speed for the vehicle to turn effectively since some vehicles experience a "bare steerage" effect of not having sufficient ability to turn a low speeds.

For the speed utility function, the `OpRegion` behavior uses the `ZAIC_PEAK` utility, a MOOS-IvP C++ defined class (in `lib_ivpbuild`) that accepts a few parameters and will produce a valid IvP function defined over a single domain, in this case desired speed. The utility function, is given by:

$$u(v) = \begin{cases} \frac{|v_p - v|}{v_p} \cdot (100 - u_{med}) + u_{med} & |\ v - \hat{v}\ | \leq v_p \\ \frac{|v_b - v|}{v_b} \cdot u_{med} & |\ v - \hat{v}\ | \leq v_b \\ 0 & otherwise \end{cases} \tag{5}$$

The terms of the above equation are provided in the table below. The only term that is configurable as behavior parameter is `save_spd` which by default is 1.0 meters per second. This value will likely need to be adjusted based on the particular vehicle in a mission.

The `peak_width`, `base_width` are set at 0.3 m/s. The `summit_delta` is set to the value of 20.

Term	Meaning
$u(\theta)$	Utility of candidate ownship heading θ
$r(\theta, \texttt{poly})$	Range from ownship to save poly in direction θ. Range is -1 if no poly intersection in direction θ
r_{max}	Max range for all possible θ values, excluding non intersection values
r_{min}	Min range for all possible θ values, excluding non intersection values

Table 25: The terms for the heading utility function of the `OpRegionV24` behavior.

Heading Utility

When the behavior is in the mode of recovering, a heading utility function is used, independently to complement the speed utility function described above. The heading utility function is given below. It utilizes a directional range calculation, the range from ownship to the save polygon, given a candidate heading maneuver. If the trajectory along a given heading does not result in the intersection of the polygon, it is given a value -1.

$$u(\theta) = \begin{cases} \frac{r_{max} - r(\theta, \texttt{poly})}{r_{max} - r_{min}} \cdot 100 & r(\theta, \texttt{poly}) \geq 0 \\ 0 & otherwise \end{cases} \tag{6}$$

The remaining terms of the above equation are provided in the table below.

The Heading and Speed IvP Function

The independent speed and utility functions are combined into a single utility function over heading and speed:

$$u(\theta, v) = w_\theta \cdot u(\theta) \oplus w_v \cdot u(v) \tag{7}$$

The construction of this IvP function is achieved through a MOOS-IvP C++ class called an `OF_Coupler`, which accepts the two one-variable functions, and a weight for each function. A two-variable IvP function is produced. This coupling utilizes two weights, for each component utility function. Five examples of different relative weights, and the resulting couple IvP function are shown in Figure 90 below.

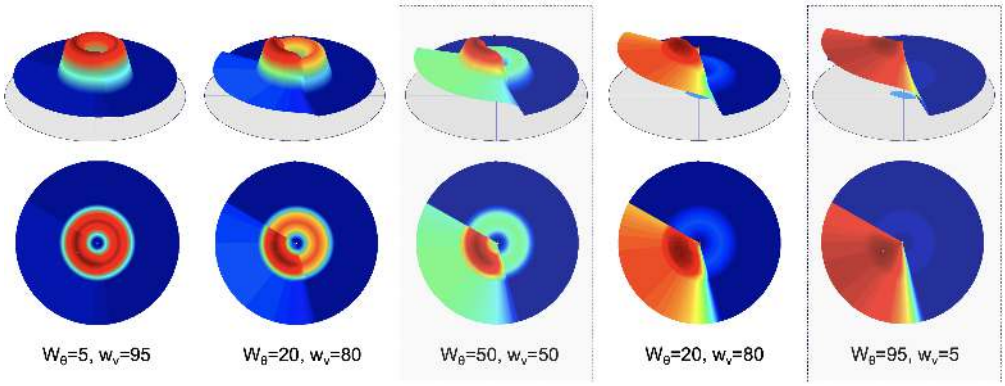

Figure 90: **Five Couplings with Different weights**: The heading-speed IvP function is created from a weighted combination of the individual heading and speed functions. Five examples are shown. In the left, the function primarily influences the desired speed, and on the right influences desired heading. In the middle, there is a balance on both heading and speed influence.

When the vehicle is not heading toward the save polygon, e.g., when it has first exited the save polygon, then the weights are 50-50. When the vehicle's current heading is on an intersection course with the save polygon, the weights are 95-5. These two cases are highlighted in Figure 90. The motivation for this is to have an IvP function that is more influential about ownship speed initially, to enhance safety when heading away from the save polygon. Once the vehicle is on a course to return to the save polygon, the influence on speed is reduced, while the influence on heading is retained.

An Example with IvP Function Output

In the example below in Figure 91, the vehicle has just exited the save polygon. Prior to exiting the save polygon, the behavior produces no IvP function. On initial exit, the produced IvP function is structured to prefer headings that would result in a trajectory returning to the polygon. Once the vehicle has turned and its current heading is pointed toward the save polygon, the shape of the IvP function is based on coupling with a weighting like the example shown on the far right in Figures 90.

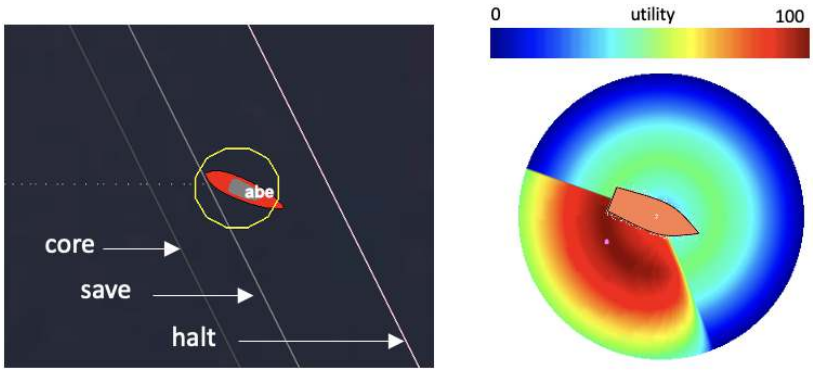

Figure 91: **OpRegionV24 IvP Function in Save Mode**: The vehicle has breached the save polygon and is producing an objective function to influence the vehicle to turn to the starboard to return to the save polygon.

25.5 Configuration Parameters and Examples

The following configuration parameters are supported for the OpRegionV24 behaviors, in addition to the parameters support for all IvP behaviors.

Listing 25.7: Configuration Parameters for the OpRegion Behavior.

Parameter:	Description
breached_altitude_flag:	A MOOS variable-value pair to be posted when or if the vehicle breaches the altitude limit set by this behavior. Section 25.6.
breached_depth_flag:	A MOOS variable-value pair to be posted when or if the vehicle breaches the depth limit set by this behavior. Section 25.6.
breached_poly_flag:	A MOOS variable-value pair to be posted when or if the vehicle breaches the convex polygon op area limit set by this behavior. Section 25.6.
breached_time_flag:	A MOOS variable-value pair to be posted when or if the vehicle breaches the time limit set by this behavior. Section 25.6.
core_poly:	The nominal operation area, a convex polygon. Section 25.3.1.
draw_halt_status:	If true, a small (by default red) polygon is drawn around the vehicle when or if it exits the halt polygon, following the vehicle until it returns to the within the halt polygon. The default is true. Section 25.5.
draw_save_status:	If true, a small (by default yellow) polygon is drawn around the vehicle when or if it exits the save polygon, following the vehicle until it returns to the within the save polygon. The default is true. Section 25.5.
halt_poly:	A convex polygon area, outside of which the behavior will generation an all-stop indicator to the helm. Section 25.3.1.
max_time:	The max allowable time in seconds.
max_depth:	The max allowable depth in meters.
min_altitude:	The min allowable altitude in meters.
save_poly:	A convex polygon area, outside of which the behavior will generation an IvP function to return within the polygon. Section 25.3.1.
reset:	If set to true, and the behavior is currently in a mode where one of the operating envelopes (time, depth, altitude or position) has been breached, the behavior will be reset to original launch state. Section 25.3.4.
trigger_entry_time:	The time required for the vehicle to have been within the halt polygon before triggering the halt polygon requirement. Section 25.3.2.
trigger_exit_time:	The time required to have been outside the polygon before declaring a polygon containment failure. Section 25.3.2.
trigger_on_poly_entry:	If true, the vehicle must first be in the halt polygon before the halt polygon constraint is enabled. The default is true. Section 25.3.2.
visual_hints:	Hints for visual properties in variables posted intended for rendering. See Section 25.5.

Example Configuration (Minimal)

Below is an minimal configuration block example, accepting default values for all other missing parameters:

Listing 25.8: Example Minimal Config Block.

```
Behavior = BHV_OpRegionV24
{
  name = opr24
  pwt  = 300

  core_poly = pts={-80,-50:-30,-175:150,-100:95,25}
  save_dist = 5
  halt_dist = 15
}
```

Example Configuration (Full)

The below configuration shows all configuration parameters with defaults indicated where appropriate:

Listing 25.9: Example Configuration Block.

```
Behavior = BHV_OpRegionV24
{
  // General Behavior Parameters
  // ---------------------------
  name         = opr24              // example
  pwt          = 300                // example
  updates      = OPREGION_UPDATES   // example

  // Params specific to this behavior
  // --------------------------------
                max_time = 0         // default (seconds)
               max_depth = 0         // default (meters)
            min_altitude = 0         // default (meters)
      trigger_entry_time = 1         // default (seconds)
       trigger_exit_time = 0.5       // default (seconds)

               core_poly = pts={-80,-50:-30,-175:150,-100:95,25}
               save_dist = 5
               halt_dist = 15

   breached_altitude_flag = SAY_MOOS = min altitude has been exceeded
      breached_depth_flag = SAY_MOOS = max depth has been exceeded
       breached_poly_flag = SAY_MOOS = halt region has been violated
       breached_time_flag = SAY_MOOS = max mission time has been exceeded
                 save_flag = SAY_MOOS = save region has been violated
                savex_flag = RECOVER_POSITION=$[OSX],$[OSY]

             visual_hints = vertex_size  = 0      // default
             visual_hints = edge_color   = aqua   // default
             visual_hints = edge_size    = 1      // default
}
```

Visual Configuration Options

The visual artifacts generated by the `OpRegionV24` behavior are related to the (a) the polygon regions and (b) small polygon indicators drawn around the vehicle to indicate state. The polygon regions

are shown in Figure 87. The polygon indicators drawn around the vehicle are shown in Figure 88. The rendering style shown in these figures is the default, but the visual artifacts can be either turned off, or rendered in different colors or sizes.

There are five polygon renderings that can be modified: (1) the core polygon, (2) the save polygon, (3) the halt polygon, (4) the save status and (5) halt status polygons. There are also five visual aspects for each polygon that may be modified: (a) the vertex size, (b) the edge size, (c) the vertex color, (d) the edge color, (e) the label color.

By setting `edge_color=green`, this sets the default edge color for all five polygons. This can be overruled for say the core polygon by setting `core_edge_color=white`. If the latter is left unspecified, the default aspects are used. In the example lines below, the first five lines are the default aspects for all polygons. The vertex color is moot since the vertex size is zero. The middle three lines overrule the default edge colors for the three region polygons with the colors shown in Figure 87. The last two lines below override the edge colors for the save and halt status polygons to yellow and red as shown in Figure 88. All lines below represent the defaults used by the `OpRegionV24` behavior if no visual hints were provided in the behavior configuration.

```
visual_hints = vertex_size = 0
visual_hints = edge_size = 1
visual_hints = vertex_color = gray50
visual_hints = edge_color = gray50
visual_hints = label_color = off

visual_hints = core_edge_color = gray30
visual_hints = save_edge_color = gray50
visual_hints = halt_edge_color = pink

visual_hints = save_status_edge_color = yellow
visual_hints = halt_status_edge_color = red
```

The posting of the save status and halt status polygons (Figure 88) can be disabled simply with:

```
draw_save_status = false
draw_halt_status = false
```

25.6 Flags and Macros

The OpRegionV24 behavior supports the below set of event flags in addition to the standard behavior flags, e.g., endflags, runflags. These are:

- `breached_altitude_flag`: If the minimum altitude constraint is enabled, this flag or flags are posted once, when or if the vehicle altitude first becomes lower than the limit.
- `breached_depth_flag`: If the maximum depth constraint is enabled, this flag or flags are posted once, when or if the vehicle depth first becomes greater than the limit.
- `breached_poly_flag`: If the halt polygon is specified, this flag or flags are posted once, when or if the vehicle exits the polygon, and has been outside the polygon for the duration specified by the `trigger_exit_time`.

- breached_time_flag: If the maximum time constraint is enabled, this flag or flags are posted once, when or if the vehicle first exits the maximum time limit.
- save_flag: If the save polygon is enabled, this flag is posted once, when or if the vehicle first exits the save polygon.
- savex_flag: If the save polygon is enabled, this flag is posted when or if the vehicle first exits the save polygon, and on each iteration thereafter so long as the vehicle remains outside the save polygon.

The following macros are supported in the `OpRegionV24` behavior. These macros will be expanded in any event flag, including event flags defined for all IvP behaviors as well as event flags defined only for the `OpRegionV24` behavior.

- $[CORE_POLY]: The specification of the polygon representing the core operation area.
- $[SAVE_POLY]: The specification of the polygon representing the operation area, outside which the vehicle will actively attempt to return to.
- $[HALT_POLY]: The specification of the polygon representing the halt area, outside which the vehicle will post an emergency all-stop message.
- $[ALT_LEFT]: If a minimum altitude is enabled, this is the number of meters remaining before the minimum altitude has been breached.
- $[DEPTH_LEFT]: If a maximum depth is enabled, this is the number of meters remaining before the maximum depth has been breached.
- $[TIME_LEFT]: If a maximum mission time is enabled, this is the number of seconds remaining before the maximum mission time has been breached.
- $[SECS_IN_HALT_POLY]: The number of seconds the vehicle has been inside the halt polygon.
- $[SECS_OUT_HALT_POLY]: The number of seconds the vehicle has been outside the halt polygon (once it has been inside).
- $[SECS_OUT_SAVE_POLY]: The number of seconds the vehicle has been outside the save polygon (once it has been inside).
- $[DIST_TO_CORE]: Distance in meters of the vehicle to the boundary of the core polygon.
- $[TRAJ_DIST_TO_CORE]: Distance in meters of the vehicle to the boundary of the core polygon, in the direction of the current vehicle trajectory.
- $[ETA_TO_CORE]: Estimated time of arrival (ETA), in seconds, of the vehicle if it were to instantaneously change direction and drive directly to the closest point of exit of the core polygon at the top vehicle speed.
- $[TRAJ_ETA_TO_CORE]: ETA of the vehicle on its current heading and speed reaching the boundary of the core polygon.
- $[DIST_TO_SAVE]: Distance in meters of the vehicle to the boundary of the save polygon.
- $[TRAJ_DIST_TO_SAVE]: Distance in meters of the vehicle to the boundary of the save polygon, in the direction of the current vehicle trajectory.
- $[ETA_TO_SAVE]: Estimated time of arrival (ETA), in seconds, of the vehicle if it were to instantaneously change direction and drive directly to the closest point of exit of the save polygon at the top vehicle speed.
- $[TRAJ_ETA_TO_SAVE]: ETA of the vehicle on its current heading and speed reaching the boundary of the core polygon.

- $[DIST_TO_HALT]: Distance in meters of the vehicle to the boundary of the halt polygon.
- $[TRAJ_DIST_TO_HALT]: Distance in meters of the vehicle to the boundary of the halt polygon, in the direction of the current vehicle trajectory.
- $[ETA_TO_HALT]: Estimated time of arrival (ETA), in seconds, of the vehicle if it were to instantaneously change direction and drive directly to the closest point of exit of the halt polygon at the top vehicle speed.
- $[TRAJ_ETA_TO_HALT]: ETA of the vehicle on its current heading and speed reaching the boundary of the halt polygon.

26 The Loiter Behavior

A behavior for transiting to and repeatedly traversing a set of waypoints forming a convex polygon. A similar effect can be achieved with the Waypoint behavior but this behavior assumes a set of waypoints forming a convex polygon to exploit certain useful algorithms discussed below. It also utilizes the non-monotonic arrival criteria used in the Waypoint behavior to avoid loop-backs upon waypoint near-misses. It also robustly handles dynamic exit and re-entry modes when or if the vehicle diverges from the loiter region due to external events. t 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.

26.1 Configuration Parameters

Listing 26.10: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].

templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 26.11: Configuration Parameters for the Loiter Behavior.

Parameter	Description
acquire_dist:	Distance to polygon outside which the behavior will be in "acquire" mode. Section 26.3.7.
capture_radius:	The radius tolerance, in meters, for satisfying the arrival at a point. Section 26.3.9.
center_activate:	If true, center of polygon is set to present position when behavior activates. Section 26.3.2.
center_assign:	An x,y pair, in meters, indicating a (new) center of the loiter polygon. Section 26.3.2.
clockwise:	If true, loitering is done clockwise (the default setting). Section 26.3.11.
polygon:	Polygon about which the vehicle will traverse and loiter. Section 26.3.1.
post_suffix:	A string to add as a suffix to variables posted by this behaviors. Section 26.3.10.
radius:	An alias for capture_radius. Section 26.3.9.
slip_radius:	An "outer" capture radius. Arrival declared when the vehicle is in this range and the distance to the point begins to increase. Section 26.3.9.
speed:	Speed in meters per second. Section 26.3.4.
speed_alt:	An alternative desired speed (m/s) at which the vehicle travels through the points. Applies only when the use_alt_speed parameter is set to true. The default is -1, which indicates internally that it has not been set. Section 26.3.5.
(Introduced after release 15.5.)	
spiral_factor:	The degree of spiraling when activated at the center of the polygon. Section 26.3.6.
use_alt_speed:	If true then attempt to use the alternate speed set with the speed_alt parameter, if that speed is set to a non-negative value. The default is false. Section 26.3.5.
(Introduced after release 15.5.)	
visual_hints:	Hints for visual properties in variables posted intended for rendering. Section 26.3.12.
xcenter_assign:	A x-value, in meters, indicating a (new) x-position of the loiter polygon. Section 26.3.2.
ycenter_assign:	A y-value, in meters, indicating a (new) y-position of the loiter polygon. Section 26.3.2.

Listing 26.12: Example Configuration Block.

```
Behavior = BHV_Loiter
{
  // General Behavior Parameters
  // ---------------------------
  name         = transit               // example
  pwt          = 100                   // default
  condition    = MODE==LOITERING       // example
  updates      = LOITER_UPDATES        // example

  // Parameters specific to this behavior
  // ------------------------------------
       acquire_dist = 10               // default
     capture_radius = 3                // default
    center_activate = false            // default
          clockwise = true             // default
        slip_radius = 15               // default
              speed = 0                // default
      spiral_factor = -2               // default

            polygon = radial:: x=5,y=8,radius=20,pts=8  // example
       post_suffix = HENRY                              // example

      center_assign = 40,50            // example
     xcenter_assign = 40               // example
     ycenter_assign = 50               // example


       visual_hints = vertex_size  = 1             // default
       visual_hints = edge_size    = 1             // default
       visual_hints = vertex_color = dodger_blue   // default
       visual_hints = edge_color   = white         // default
       visual_hints = nextpt_color = yellow        // default
       visual_hints = nextpt_lcolor = aqua         // default
       visual_hints = nextpt_vertex_size = 5       // default
       visual_hints = label        = zone3         // example
}
```

26.2 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be suppressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.8.

- LOITER_ACQUIRE: Posts 1 when in the "acquire" mode, 0 otherwise.
- LOITER_DIST_TO_POLY: Current distance, in meters, to the loiter polygon.
- LOITER_ETA_TO_POLY: Estimated time of arrival to the polygon at present speed and trajectory.
- LOITER_INDEX: The index of the vertex in the loiter polygon currently heading toward.
- LOITER_MODE: A string indicating details of the acquire mode, e.g., "acquiring_external".
- LOITER_REPORT: A status string with current mode, current vertex, and prior arrivals.
- VIEW_POINT: A visual cue for rendering the next point in the loiter polygon.
- VIEW_POLYGON: A visual cue for rendering the loiter polygon.

The following are some examples:

```
      LOITER_REPORT = index=5,capture_hits=51,nonmono_hits=0,acquire_mode=false
       LOITER_INDEX = 5
    LOITER_ACQUIRE = 1
LOITER_DIST_TO_POLY = 2.2
 LOITER_ETA_TO_POLY = 294.4
       LOITER_MODE = stable
      VIEW_POLYGON = edge_size,0.0:vertex_size,0.0:0,-50:0,-150:150,-150:150,-50
```

26.3 Detailed Discussions on Loiter Behavior Parameters

26.3.1 The `polygon` Parameter for Setting the Loiter Region

The Loiter behavior is configured with a *loiter region*, defined by a convex polygon. The behavior will influence the vehicle to repeatedly traverse the set of points on the polygon. The polygon is specified typically in the behavior configuration block. Any string that properly defines a convex polygon is acceptable. The following two configuration lines, for example, will result in the same polygon.

```
polygon = 20,-40:40,-75:20,-110:-20,-110:-40,-75:-20,-40:label,Lima

polygon = format=radial, x=0, y=-75, radius=40, pts=6, snap=1, label=Lima
```

Each would produce the polygon shown in Figure 92:

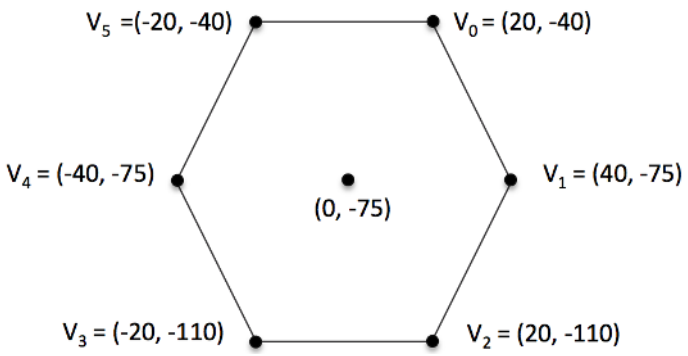

Figure 92: A typical loiter polygon with six vertices.

The shape and position polygon may be alterered dynamically (after the helm is launched and running) in one of two manners by either specifying new parameters explicitly, or by tying the loiter position to the vehicle position when the loiter behavior enters the running behavior mode.

26.3.2 The `updates` Parameter for Updating the Loiter Region

In the first case, altering the polygon parameter dynamically is accomplished by using the standard `updates` parameter described in Section 7.2.5. For example, the behavior may be configured to

receive new updates by adding the following line to its configuration block:

```
updates = NEW_CONFIG
```

Its position may be then moved 75 meters North by posting the following to the MOOSDB:

```
NEW_CONFIG = "polygon = format=radial, x=0, y=0, radius=40, pts=6, snap=1, label=Lima"
```

Alternatively, if one wants to simply move the polygon to a new `(x,y)` position, the Loiter behavior also implements the `center_assign` configuration parameter. The same shift as above can be made without the source making the post having to know anything about the other parameters of the polygon:

```
NEW_CONFIG = "center_assign = 0,0"
```

Likewise, if one simply wants to move the polygon in either the x or y direction without knowing anything about the position in the other direction, the Loiter behavior implements the `xcenter_assign` and `ycenter_assign` parameters. Thus the above shift could also be accomplished without the source making the post knowing anything about the current x position of the polygon:

```
NEW_CONFIG = "ycenter_assign = 0"
```

26.3.3 The `center_activate` Parameter for Using the Current Vehicle Position

The Loiter behavior may also be configured to reset the (x,y) center position of the loiter polygon to the present position of the vehicle at the very the moment the behavior transitions from the idle to the running state. See Section 6.5.3 for more on behavior run states. This configuration is declared with the following line in the behavior configuration block:

```
center_activate = true
```

This setting is useful if one wants to, for example, send a command to the vehicle to exit some other mission mode and simply loiter at its present location until further notice. Of course this configuration as well, may also be dynamically toggled through a variable specified in the `updates` parameter.

26.3.4 The `speed` Parameter for Setting the Loiter Speed

The desired speed, in meters/second, at which the vehicle travels through the points.

26.3.5 The `speed_alt` and `use_alt_speed` Parameters

When the `use_alt_speed` parameter is set to `"true"`, the loiter behavior will use speed set in the `speed_alt` parameter, if it has been set. By default `use_alt_speed` is `"false"` and `speed_alt` is set to -1. This provides a convenient way for other behaviors or apps to periodically influence the loiter behavior to a different speed. The other behavior or app does not need to read, remember and reset the loiter behavior back to its original speed, but can instead just toggle `use_alt_speed` to `false` to return the loiter back to its original speed.

26.3.6 The spiral_factor Parameter

A value in the range $[0, 100]$ indicating the degree to which the vehicle will spiral out to the polygon if started on the inside. A setting of zero indicates it will move directly to the first polygon vertex. This parameter only relates to the situation of starting inside the polygon. The default value is 98.

26.3.7 The acquire_dist Parameter

The acquire_dist parameter is the distance in meters between the vehicle and the polygon that will trigger the vehicle to return to *acquire* mode. This notion applies to the case where the vehicle is both inside and outside the polygon. The re-acquire algorithms are different however. The default is 10 meters.

26.3.8 The xcenter_assign and ycenter_assign Parameters

26.3.9 The capture_radius and slip_radius Parameters

radius: The radius tolerance, in meters, for satisfying the arrival at a waypoint. As soon as the vehicle is within this distance to the waypoint the waypoint behavior begins operating on the next waypoint in the sequence, or completes and posts its endflags if there are no more waypoints.

slip_radius:As the vehicle progresses toward a waypoint, the sequence of measured distances to the waypoint decreases monotonically. The sequence becomes non-monotonic when it hits its waypoint or when there is a near-miss of the waypoint arrival radius. The slip_radius, a.k.a, the nm_radius, short for *non-monotonic radius* is an arrival 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 arrival radius to have any effect (see Figure 80). As a rule of thumb, a distance of twice the arrival radius is practical.

26.3.10 The post_suffix Parameter

The post_suffix parameter names a string to be added as a suffix to each of the status variables posted by the behavior (LOITER_REPORT, LOITER_INDEX, LOITER_ACQUIRE, LOITER_DIST2POLY). By default, the suffix is the empty string and the variables will be posted as above. When multiple Loiter behaviors are configured in the helm it may help to distinguish the posted variabes by a suffix. A given suffix of "FOO" would result in the posting of LOITER_INDEX_FOO for example. The extra '_' character is inserted automatically.

26.3.11 The clockwise Parameter for Setting the Loiter Direction

The user may configure the direction the vehicle traverses the loiter polygon by setting the parameter:

```
clockwise = <value>
```

The possible case insensitive settings for <value> are "true", "false", "best". In the first two cases, the directions are explicitly set and will not vary regardless of the pose of the vehicle with respect to the polygon. In the case where clockwise=best, the direction is re-evaluated once, whenever the behavior run state transitions from idle to running. Thus the direction depends on the pose relative

position to the polygon at that particular point in time. This is shown in the lower case in Figure 93 below.

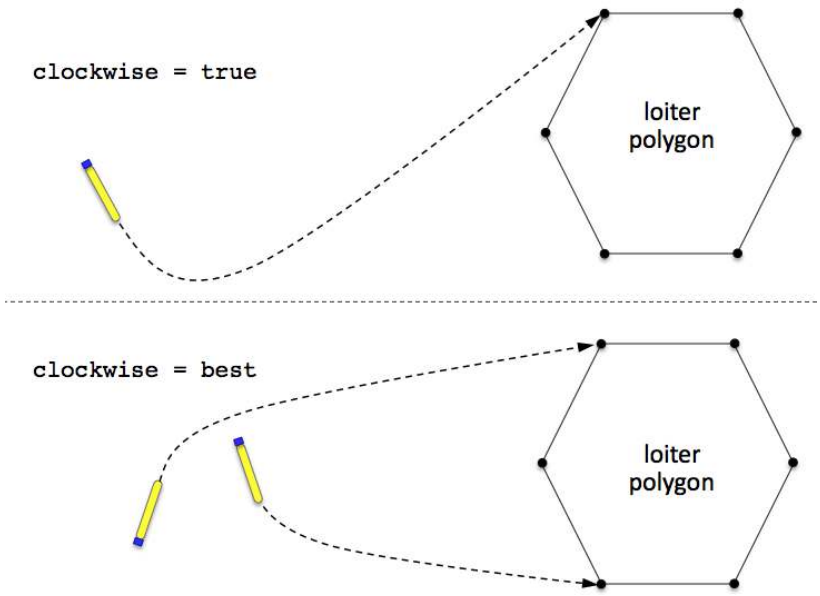

Figure 93: **The Loiter Direction.** The polygon traversal direction is determined by the `clockwise` configuration parameter. It may be explicitly set as shown on the top, or determined at run time when the behavior becomes non-idle as shown on the bottom.

Regardless of the prevailing direction, as the vehicle is transiting to the loiter polygon, the behavior will influence the vehicle toward the vertex that allows for the smoothest entry, given the chosen direction. If the polygon were a perfect circle, the vehicle would approach on one of the two tangent lines.

26.3.12 The `visual_hints` Parameter

Although the primary output of the Loiter behavior an IvP Function, a number of visual properties are also published for convenience in mission monitoring. This includes both the loiter polygon and the point on the polygon the behavior is presently progressing toward. These visual artifacts have default properties in size and color that may be altered to the user's preferences. These preferences are configurable through the `visual_hints` parameter. Each parameter below is used in the following way by example:

```
visual_hints = vertex_size=3, edge_size=2
visual_hints = vertex_color=khaki
```

- `vertex_size`: The size of vertices rendered in the loiter polygon. The default is 1.
- `edge_size`: The width of edges rendered in the loiter polygon. The default is 1.
- `vertex_color`: The color of vertices rendered in the loiter polygon. The default is "dodger_blue".

- edge_color: The color of edges rendered in the loiter polygon. The default is "white".
- nextpt_color: The color of the point rendered as the present next waypoint. The default is "yellow".
- nextpt_lcolor: The color of the label for the point rendered as the present next waypoint. The default is "aqua".
- label: The label of rendered for the loiter polygon.

Rendering of vertices or the next waypoint may be shut off with a size of zero, and labels may be shut off with the special color "invisible". For a list of legal colors, see Appendix 21.

26.4 The Loiter Acquisition Mode

When the Loiter behavior is running (non-idle), it behaves differently depending on where the vehicle is in relation to the loiter polygon. The behavior has a notion of (a) when it is progressing in a "stable" manner around the polygon and (b) when it is trying to move the vehicle back to (a). The difference between the two is solely determined by whether or not the vehicle is within some *acquire distance* from the loiter polygon. This distance is set with the behavior configuration parameter:

```
acquire_dist = <distance>
```

The <distance> component must simply be a non-negative value. A value of zero should work fine, but essentially disables some useful ways in which the behavior may be coordinated with other parts of the mission. This relationship is shown in Figure 94.

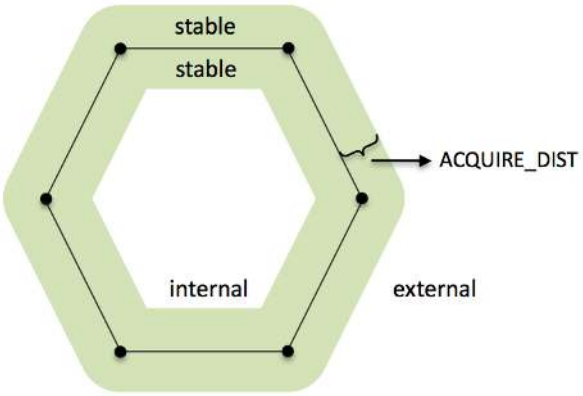

Figure 94: **The Loiter Polygon Zones:** The Loiter behavior regards itself to be in one of three distinct possible zones relative to the polygon boundary - *stable*, *internal*, *external*, depending on the setting of the acquire_dist parameter as shown.

When the vehicle is *not* within the stable zone, it is trying to acquire the polygon in one of four distint manners, depending on whether (a) the vehicle is internal or external and (b) whether it is "acquiring" the polygon for the first time, or whether it is "recovering" from having drifted out of the stable zone. The idea is depicted in Figure 95.

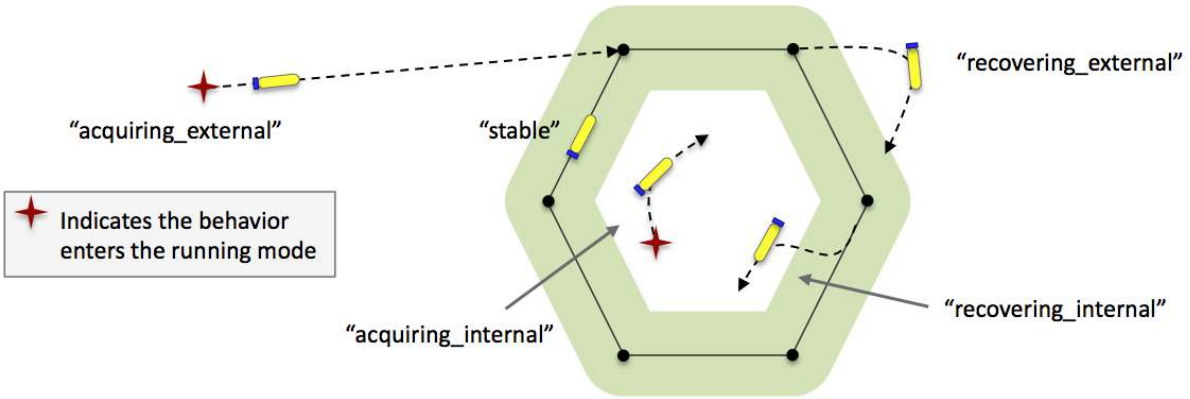

Figure 95: **Loiter Modes**: The behavior is one of five possible modes, *stable*, *acquiring_external*, *recovering_external*, *recovering_internal*, or *acquiring_internal*.

The current loiter mode is published by the behavior in the MOOS variable `LOITER_MODE`. As with any MOOS variable published by a behavior, this may be remapped to a different variable of the users liking with the `post_mapping` configuration parameter.

When the behavior is in any loiter mode other than the stable mode, it recalculates on each iteration which vertex it should be heading toward, the *approach vertex*. In the case of an external approach, the chosen vertex should remain steady unless there are external forces such as wind or current, or if the vehicle changes its aspect to the polygon significantly as it is executing a turn. In the case of an internal approach, the approach vertex will likely change during the approach, outward toward the polygon boundary, creating a pseudo outward spiral trajectory. Note that re-evaluating the approach vertex is not the same as re-evaluating the traversal direction of the polygon. The latter is only re-evaluated dynamically if the behavior is configured with `clockwise=best`, and then only when the behavior becomes non-idle.

The circumstance most common for triggering the acquire mode is the initial assignment to the vehicle to loiter at a new given region in the X,Y plane. This assignment *could* occur while the vehicle happens to already be within the polygon for a number of reasons. Furthermore, the vehicle could be driven off the polygon loiter trajectory due to environmental (wind or current) forces or the temporary dominance of other vehicle behaviors such as collision avoidance or tracking of another vehicle.

Once the behavior enters the acquire mode, it remains in this mode until arriving at the first waypoint (defined by the arrival and non-monotonic radii settings), after which it switches to normal mode until the acquire mode is re-triggered or the behavior run conditions are no longer met. There is currently no "complete" condition for this behavior other than a time-out which is defined for all behaviors.

27 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 (busy), and a period specifying when the influence if off (lazy), as depicted in Figure 96.

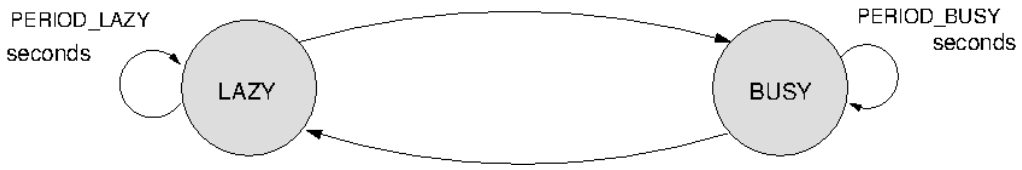

Figure 96: **Busy and Lazy Modes:** In the busy mode the behavior will produce an objective function defined over speed that will potentially influence the speed of the vehicle. In the lazy mode, it simply will not produce an objective function.

It was conceived for use on an AUV equipped with an acoustic modem to periodically slow the vehicle to reduce self-noise and reduce communication difficulty. One can also specify a flag (a MOOS variable and value) to be posted at the start of the period to prompt an outside action such as the start of communication attempts.

27.1 Configuration Parameters

Listing 27.13: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].

perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 27.14: Configuration Parameters for the Periodic Speed Behavior.

Parameter	Description
basewidth:	The width of the ZAIC basewidth, in m/sec, in the IvP function.
initially_busy:	If true the initial state is busy. The default is false.
peakwidth:	The width of the ZAIC peakwidth, in m/sec in the IvP function.
period_busy:	The duration of the busy period, in seconds.
period_lazy:	The duration of the lazy period, in seconds.
period_speed:	The desired speed, in m/sec, in the IvP function.
reset_upon_running:	Initial conditions reset upon entering the running state. Default is `true`.
summit_delta:	The extent of the ZAIC summit delta in the IvP function.

Listing 27.15: Example Configuration Block.

```
Behavior = BHV_PeriodicSpeed
{
  // General Behavior Parameters
  // ---------------------------
  name         = transit            // example
  pwt          = 100                // default
  condition    = MODE==TRANSITING   // example
  updates      = PSPD_UPDATES       // example

  // Parameters specific to this behavior
  // ------------------------------------
           basewidth = 0            // default
       initially_busy = false       // default
           peakwidth = 0            // default
         period_busy = 0            // default
         period_lazy = 0            // default
        period_speed = 0            // default
  reset_upon_running = true         // default
        summit_delta = 25           // default
}
```

27.2 Variables Published

Listing 27.16: Variables Published the Periodic Speed Behavior.

Variable	Description
PS_PENDING_BUSY:	Publishes the amount of time in seconds until the behavior reaches the busy mode. During the busy mode this value is zero, and it resets to the value given by the period_lazy parameter upon transitioning into the lazy mode.
PS_PENDING_LAZY:	Publishes the amount of time in seconds until the behavior reaches the lazy mode. During the lazy mode this value is zero, and it resets to the value given by period_busy upon transitioning back into the busy mode. To reduce posting volume, the values posted will be rounded to the nearest second until less than one second remains, after which fractions are posted.
PS_BUSY_COUNT:	Thi variable is posted by the behavior each time it enters the busy mode. The value is an integer indicating the number of times it has entered the busy mode, posting zero initially.

27.3 State Transition Policy and Initial Condition Parameters

The behavior alternates between one of two modes, the busy mode or the lazy mode. In the former, it will produce an objective function over the *speed* decision variable, and in the latter mode it will simply refrain from producing the objective function. Note that these modes are different from the general behavior run states described in Section 6.5.3. If this behavior is idle, i.e., has not met the conditions for being in the running state, it will not generate an objective function regardless of whether it is in the busy or lazy mode.

When the behavior enters the busy mode, it resets a timer which counts down from period_busy seconds, after which it enters the lazy mode. When it enters the lazy mode, it resets a timer which counts down from period_lazy seconds, after which it goes back to the busy mode. By default the

behavior is initially in the lazy mode, but it can be configured in the opposite manner by setting `initially_busy` to `true`.

By default, when `reset_upon_running` is true, each time the behavior enters the running state, the busy/lazy mode is set to its initial value, and all timers are reset to their initial values. This is true regardless of whether it is entering the running state upon initial helm engagement, or due to transitioning from the idle state. This can be changed by setting `reset_upon_running` to false. In this case the timers are counting down immediately upon helm engagement and continue to do so regardless of the behavior run state.

28 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.

28.1 Configuration Parameters

Listing 28.17: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 28.18: Configuration Parameters for the PeriodicSurface Behavior.

Parameter	Description
acomms_mark_variable	The incoming MOOS variable for resetting the acomms period clock.
ascent_grade	Manner in which desired speed approaches zero on the approach to the surface. The default is `linear`. Section 28.2.6.
ascent_speed	Desired speed of the vehicle during the ascent mode. Section 28.2.5.
atsurface_status_var	MOOS variable indicating number of seconds at the surface. Section 28.2.4.
mark_variable	The incoming MOOS variable for resetting the period clock. Section 28.2.2.
max_time_at_surface	The maximum time, in seconds, the vehicle will wait at the surface. Section 28.2.8.
pending_status_var	MOOS variable written to with remaining time on idle clock. Section 28.2.3.
period	Duration of the `WAITING` mode. Section 28.2.1.
zero_speed_depth	The depth, in meters, at which the desired speed becomes zero on ascent. Section 28.2.7.

Listing 28.19: Example Configuration Block.

```
Behavior = BHV_PeriodicSurface
{
  // General Behavior Parameters
  // ---------------------------
  name         = periodic_surface   // example
  pwt          = 100                // default
  condition    = MODE==TRANSITING   // example
  updates      = PSURFACE_UPDATES   // example

  // Parameters specific to this behavior
  // ------------------------------------
  acomms_mark_variable = ACOMMS_RECEIVED       // example
          ascent_grade = linear                // default
          ascent_speed = -1                    // default
  atsurface_status_var = TIME_AT_SURFACE       // default
         mark_variable = GPS_UPDATE_RECEIVED   // default
   max_time_at_surface = 300                   // default
    pending_status_var = PENDING_SURFACE       // default
                period = TIME_AT_SURFACE       // default
     zero_speed_depth = 0                      // default
}
```

28.2 Detailed Discussions of Behavior Parameters

28.2.1 The period Parameter

The period parameter sets the duration of the period, in seconds, during which the behavior will remain in the IDLE_WAITING state.

28.2.2 The mark_variable Parameter

The mark_variable names a variable used for indicating when the behavior witnesses the event that would reset the period clock. On each iteration, the variable is checked against its last known value and if different, the clock is reset. The default value for this parameter is GPS_UPDATE_RECEIVED. If this variable is populated by another process with a value indicating the time a GPS fix is obtained, then the mark will occur on each GPS fix. Since the value of this argument names a MOOS variable, it is case sensitive.

28.2.3 The pending_status_var Parameter

The pending_status_var names a variable to be written to with the value of the remaining time on the idle clock, rounded to integer seconds. The default value is PENDING_SURFACE. Since the value of this argument names a MOOS variable, it is case sensitive.

28.2.4 The atsurface_status_var Parameter

The atsurface_status_var parameter names a variable to be written to with the number of seconds that the vehicle has been waiting at the surface (for the event indicated by the MOOS variable specified in the mark_variable parameter). The number of seconds is rounded to the nearest integer and will be zero when the vehicle is not at the surface. The default value is time_at_surface. Since the value of this parameter names a MOOS variable, it is case sensitive.

28.2.5 The ascent_speed Parameter

The ascent_speed parameter indicates the desired speed (m/s) of the vehicle during the ascent state. If left unspecified, the ascent speed will be equal to the current noted speed at moment it transitions into the ascent state.

28.2.6 The ascent_grade Parameter

The ascent_grade parameter indicates the manner in which the ascent speed approaches zero as the vehicle progresses toward the zero_speed_depth. It has four legal values: `fullspeed`, `linear`, `quadratic`, and `quasi`. The default is `linear`. In all four cases, the initial speed is determined by the parameter ascent_speed, and the desired speed will be zero once the zero_speed_depth has been achieved. The four settings determine the manner of slowing to zero speed during the ascent. The `fullspeed` setting indicates that desired speed should remain constant through the ascent right up to the instant the vehicle achieves zero_speed_depth. For the other three settings the speed reduction is relative to the starting depth (the depth noted at the outset of the ascent state) and the zero_speed_depth. With the `linear` setting, the speed reduction is linear. With the `quadratic` setting, the speed reduction is quadratic (quicker initial speed reduction). With the `quasi` setting the speed reduction is between linear and quadratic. The value passed to this parameter is not case sensitive.

28.2.7 The zero_speed_depth Parameter

The zero_speed_depth parameter sets the depth (in meters) during the ascent state at which the desired speed becomes zero, and presumably further ascent is achieved through positive buoyancy.

28.2.8 The max_time_at_surface Parameter

The max_time_at_surface parameter sets the maximum time (in seconds) spent in the AT_SURFACE state, waiting for the event indicated by the mvar_variable, before the behavior transitions into the IDLE state.

28.3 Internal States of Periodic Surface Behavior

The behavior can be in one of four states as described in Figure 97 below.

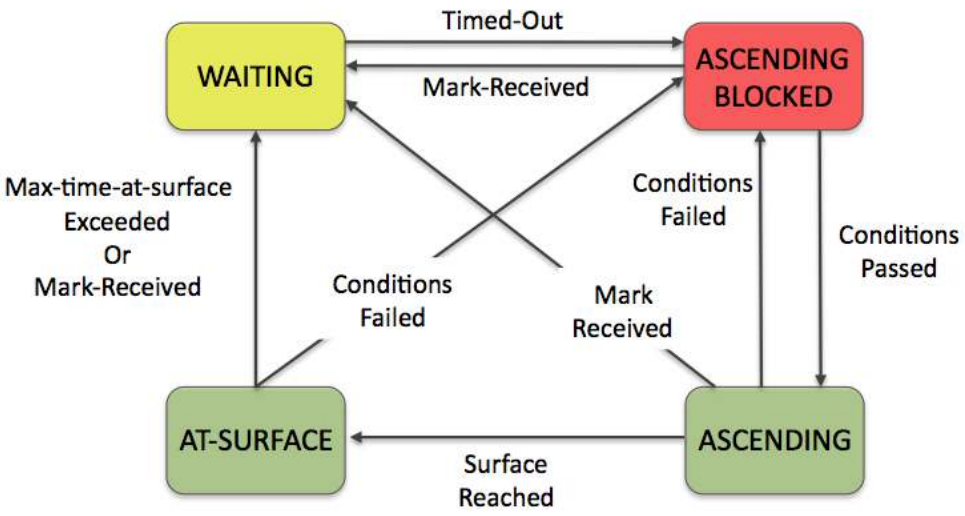

Figure 97: Possible modes of the PeriodicSurface behavior.

In the WAITING state the behavior is simply waiting for its clock to wind down to zero. The duration is given by the period parameter listed below. The clock is active despite any other run conditions that may apply to the behavior. It is started when the behavior is first instantiated and also when the desired event occurs at the surface. The ASCENDING_BLOCKED state indicates that the behavior timer has reached zero, but another run condition has not been met. This is to prevent the behavior from trying to surface the vehicle when other circumstances override the need to surface. In the ASCENDING state, the behavior will produce an objective function over depth and speed to bring the vehicle to the surface. A couple parameters described below can determine the trajectory of the vehicle during ascent. This state can transition back to the ASCENDING_BLOCKED state if run conditions become no longer satisfied prior to the vehicle reaching the surface. In the AT_SURFACE state the vehicle is at the surface waiting for a specified event.

29 The ConstantDepth Behavior

This behavior will drive the vehicle at a specified depth. This behavior merely expresses a preference for a particular depth. If other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

29.1 Configuration Parameters

Listing 29.20: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 29.21: Configuration Parameters for the ConstantDepth Behavior.

Parameter	Description
basewidth:	The width of the base, in meters, in the produced ZAIC-style IvP function. Section 29.4.
depth:	The desired depth of the vehicle, in meters.
duration:	Behavior duration, in seconds. Mandatory configuration for this behavior. Section 29.3.
peakwidth:	The width of the peak, in meters, in the produced ZAIC-style IvP function. Section 29.4.
summitdelta:	The height of the summit delta parameter in the produced ZAIC-style IvP function. See Figure 52. Section 29.4.
depth_mismatch_var:	Name of the MOOS variable indicating the present delta between the desired depth and the current depth. Section 32.2.

Listing 29.22: Example Configuration Block.

```
Behavior = BHV_ConstantDepth
{
  // General Behavior Parameters
  // ---------------------------
  name         = const_dep_survey   // example
  pwt          = 100                // default
  condition    = MODE==SURVEYING    // example
  updates      = CONST_DEP_UPDATES  // example

  // Parameters specific to this behavior
  // ------------------------------------
          basewidth = 100          // default
              depth = 0            // default
  depth_mismatch_var = DEPTH_DIFF  // example
           duration = 0            // default (Choose something higher!)
          peakwidth = 3            // default
       summitdelta = 50            // default
}
```

29.2 Variables Published

The behavior may be optionally configured to publish a variable indicating the discrepancy between the requested depth and the actual observed depth. The MOOS variable is named using the depth_mismatch_var parameter. This variable will only be published when the behavior is in the running state.

29.3 The duration Parameter

The duration parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to

another behavior or event via condition variables, then setting `duration=no-time-limit` will result in no time duration checks for this behavior.

29.4 The ConstantDepth Objective Function

See Figure 52.

- `basewidth`: The width of the base, in meters in the produced objective function. The default is 100. See Figure 52 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.
- `peakwidth`: The width of the peak in meters in the produced objective function. The default is 3. See Figure 52 for more on the peak parameter used in the ZAIC tool for building IvP functions.
- `summitdelta`: The width of the base, in meters in the produced objective function. The default is 50. See Figure 52 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.

30 The ConstantHeading Behavior

This behavior will drive the vehicle at a specified heading. 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 multi-objective optimization process.

30.1 Configuration Parameters

Listing 30.1: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 30.2: Configuration Parameters for the ConstantHeading Behavior.

Parameter	Description
basewidth	The width of the base, in degrees, in the produced ZAIC-style IvP function. Section 30.5.
complete_thresh	The threshold in discrepancy between the the requested heading and present heading before the behavior completes. The default is −1. Section 30.4.
duration	Behavior duration, in seconds. Mandatory configuration for this behavior. Section 30.3.
heading	The desired heading of the vehicle, in degrees (North=0).
peakwidth	The width of the peak, in degrees, in the produced ZAIC-style IvP function. Section 30.5.
summitdelta	The height of the summit delta parameter in the produced ZAIC-style IvP function. Section 30.5.
heading_mismatch_var	Name of the MOOS variable indicating the present delta between the desired heading and the current heading. Section 30.2.

Listing 30.3: Example Configuration Block.

```
Behavior = BHV_ConstantHeading
{
  // General Behavior Parameters
  // ---------------------------
  name         = const_hdg            // example
  pwt          = 100                  // default
  condition    = MODE==GO_STRAIGHT    // example
  updates      = CONST_HDG_UPDATES    // example

  // Parameters specific to this behavior
  // ------------------------------------
            basewidth = 10           // default
             duration = 0            // default
              heading = 0            // default
  heading_mismatch_var = HDG_DIFF    // example
            peakwidth = 10           // default
           summitdelta = 25          // default
}
```

30.2 Variables Published

The behavior may be optionally configured to publish a variable indicating the discrepancy between the requested heading and the actual observed heading. The MOOS variable is named using the heading_mismatch_var parameter. This variable will only be published when the behavior is in the running state.

30.3 The duration Parameter

The duration parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to another behavior or event via condition variables, then setting duration = no-time-limit will result in no time duration checks for this behavior.

30.4 Behavior Completion

The behavior, by default, remains active so long as its conditions are met, just like any other behavior. If the user optionally declares a threshold via the complete_thresh parameter, the behavior will complete once the discrepancy between the observed heading and the goal heading falls below that threshold. In this case the behavior will complete, as any other helm behavior, by posting its endflags.

30.5 The ConstantHeading Objective Function

See Figure 52.

- basewidth: The width of the base, in degrees in the produced objective function. The default is 170. See Figure 52 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.
- peakwidth: The width of the peak in degrees in the produced objective function. The default is 10. See Figure 52 for more on the peak parameter used in the ZAIC tool for building IvP functions.
- summitdelta: The width of the base, in meters in the produced objective function. The default is 25. See Figure 52 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.

31 The ConstantSpeed Behavior

This behavior will drive the vehicle at a specified speed. This behavior merely expresses a preference for a particular speed. If other behaviors also have a speed preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

31.1 Configuration Parameters

Listing 31.1: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 31.2: Configuration Parameters for the ConstantSpeed Behavior.

Parameter	Description
basewidth	The width of the base, in meters per second, in the produced ZAIC-style IvP function. Section 31.4.
duration	Behavior duration, in seconds. Mandatory configuration for this behavior. Section 31.3.
peakwidth	The width of the peak, in meters per second, in the produced ZAIC-style IvP function. Section 31.4.
speed	The desired speed of the vehicle, in meters per second.
summitdelta	The height of the summit delta parameter in the produced ZAIC-style IvP function. Section 31.4.
speed_mismatch_var	Name of the MOOS variable indicating the present delta between the desired speed and the current speed. If left unspecified, no posting is made. Section 31.2.

Listing 31.3: Example Configuration Block.

```
Behavior = BHV_ConstantSpeed
{
  // General Behavior Parameters
  // ---------------------------
  name         = const_spd_transit  // example
  pwt          = 100                // default
  condition    = MODE==TRANSITING   // example
  updates      = CONST_SPD_UPDATES  // example

  // Parameters specific to this behavior
  // ------------------------------------
          basewidth = 0.2          // default
           duration = 0            // default
              speed = 0            // default
  speed_mismatch_var = SPEED_DIFF   // example
          peakwidth = 0            // default
       summitdelta = 0             // default
}
```

31.2 Variables Published

The behavior may be optionally configured to publish a variable indicating the discrepancy between the requested speed and the actual observed speed. The MOOS variable is named using the speed_mismatch_var parameter. This variable will only be published when the behavior is in the running state.

31.3 The duration Parameter

The duration parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to

another behavior or event via condition variables, then setting `duration = no-time-limit` will result in no time duration checks for this behavior.

31.4 The ConstantSpeed Objective Function

See Figure 52.

- `peakwidth`: The width of the peak in meters/second in the produced objective function. The default is 0. See Figure 52 for more on the peak parameter used in the ZAIC tool for building IvP functions.
- `basewidth`: The width of the base, in meters/second in the produced objective function. The default is 0.2. See Figure 52 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.
- `summitdelta`: The width of the base, in meters/second in the produced objective function. The default is 0. See Figure 52 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.

32 The MaxDepth Behavior

This behavior will drive the vehicle within a configured depth tolerance. This behavior merely expresses a preference for a particular depth. If other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process.

This behavior differs from the maximum depth in the `OpRegion` or newer `OpRegionV24` behavior In the `MaxDepth` behavior, an objective function is produced to prevent the vehicle from exceeding the maximum depth, perhaps tempering other behaviors that may prefer deeper depths. In the `OpRegionV24` behavior there is no attempt to influence the vehicle depth, but rather only to monitor the observed depth and produce an error if the depth is exceeded.

The following parameters are defined for this behavior:

32.1 Configuration Parameters

Listing 32.4: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as `priority`.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].

spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 32.5: Configuration Parameters for the MaxDepth Behavior.

Parameter	Description
basewidth:	The width of the base, in meters, in the produced ZAIC-style IvP function. Section 32.3.
max_depth:	The maximum allowable depth of the vehicle, in meters.
tolerance:	Another way (alias) of setting the basewidth parameter Section 32.3.
depth_slack_var:	Name of the MOOS variable indicating the present delta between the maximum depth and the current depth. Section 32.2.

Listing 32.6: Example Configuration Block.

```
Behavior = BHV_MaxDepth
{
  // General Behavior Parameters
  // ---------------------------
  name      = maxdepth
  pwt       = 1000
  condition = MODE==ACTIVE

  // Parameters specific to this behavior
  // ------------------------------------
       max_depth = 80              // example (meters)
       tolerance = 100             // default (meters)
  depth_slack_var = DEPTH_SLACK    // example
}
```

32.2 Variables Published

The behavior may be optionally configured to publish a variable indicating the discrepancy between the maximum allowed depth and the actual observed depth. The MOOS variable is named using the depth_slack_var parameter. This variable will only be published when the behavior is in the running state.

32.3 The MaxDepth Objective Function

- basewidth: The width of the base, in meters in the produced objective function. The default is 100. See Figure 29 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.
- tolerance: This is an alias, or another way of specifying the basewidth parametr.

33 The GoToDepth Behavior

This behavior will drive the vehicle to a sequence of specified depths and duration at each depth. This behavior merely expresses a preference for a particular depth. If other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process. A log of vehicle depth similar to figure below may result.

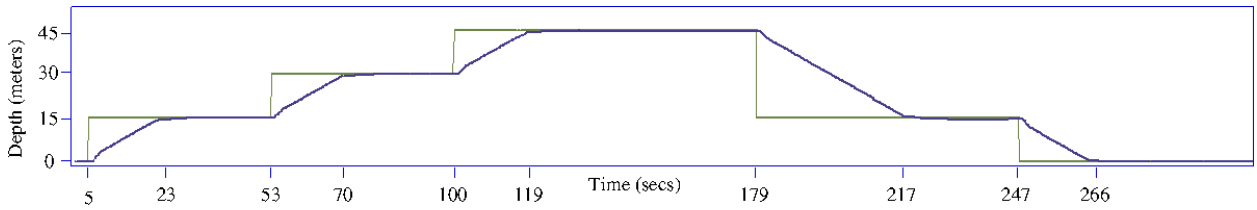

Figure 98: Depth log from simulation with the depth parameters shown in Listing 9. The lighter, step-like line indicates the values of DESIRED_DEPTH generated by the helm, and the darker line indicates the recorded depth value of the vehicle. The depth plateaus start from the moment the vehicle achieves depth. For example, the vehicle achieved a depth of 45 meters at 119 seconds and retained that desired depth for another 60 seconds as requested in the configuration shown in Listing 9.

33.1 Configuration Parameters

Listing 33.7: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].

post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 33.8: Configuration Parameters for the GoToDepth Behavior.

Parameter	Description
capture_delta	The delta depth, in meters, between the current observed depth and the current target depth, below which the behavior will declare the depth to have been achieved. The default value is 1 meter. Section 33.2.2.
capture_flag	The name of a MOOS variable incremented each time a target depth level has been achieved. Section 33.2.1.
depth	A colon-separated list of comma-separated pairs. Each pair contains a desired depth and a duration at that depth. Section 33.2.3.
repeat	The number of times the vehicle will traverse through the evolution of depths. Section 33.2.4.

Listing 33.9: Example Configuration Block.

```
Behavior = BHV_GoToDepth
{
  // General Behavior Parameters
  // ---------------------------
  name         = gotodepth                // example
  pwt          = 100                      // default
  condition    = MODE==Alpha              // example
  updates      = GOTO_DEPTH_UPDATES       // example

  // Parameters specific to this behavior
  // ------------------------------------
    capture_delta = 1                     // default (meters)
     capture_flag = DEPTH_ACHIEVED        // example
            depth = 40,60:30,45:20,45     // example
           repeat = 0                     // default
}
```

33.2 A Detailed Discussion of GoToDepth Behavior Parameters

33.2.1 The capture_flag Parameter

This parameter names a MOOS variable incremented each time a target depth level has been achieved. It may be useful for logfile analysis and also allows other behaviors to be conditioned on a depth event. If this behavior is completed in *perpetual* mode, the counter is reset to zero. If the behavior is repeating a set of depths by setting repeat greater than zero, the counter will continue to increment through evolutions. The default value is the empty string, meaning nothing will be posted. Note the named MOOS variable will automatically have the prefix "GTD_" applied.

33.2.2 The capture_delta Parameter

When the GoToDepth behavior is running and actively influencing the depth of the vehicle to a target depth level, it monitors the discrepancy between the obvserved depth and the current target depth. The capture_delta parameter is the delta depth, in meters, below which the behavior will declare the depth to have been achieved. The default value is 1 meter.

As an example, consider a target depth of say 100 meters, and a capture_delta of 5. When a diving vehicle reaches 95 meters, it will consider the depth achieved. However, it will continue to influence the vehicle to a depth of 100 meters for as a long as prescribed by the depth setting. The "achieving" of the depth results in two things: (a) the duration clock for time spent at that depth will begin, and (b) if there is a capture_flag set, this flag will be posted by the helm to the MOOSDB.

33.2.3 The depth Parameter

The depth parameter is a colon-separated list of comma-separated pairs. Each pair contains a desired depth and a duration at that depth. The duration applies from the point in time that the depth is first achieved. If a time duration is not provided for any pair, it defaults to zero. Thus depth=20 is a valid parameter setting.

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 capture_delta parameter. The behavior subscribes for NAV_DEPTH to examine the current vehicle depth against the target depth. If the current depth is within the delta given by capture_delta, that depth is considered to have been achieved. The behavior also stores the previous 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 whether the prior or current depth is actually within the capture_delta.

33.2.4 The repeat Parameter

The number of times the vehicle will traverse through the evolution of depths, proceeding to the 1st depth after the nth depth has been hit. The default value is zero.

33.2.5 The perpetual Parameter

The perpetual parameter is defined at the superclass level, but it's worth discussing its function here. If equal to true, when the vehicle completes its evolution of depths (perhaps several evolutions

if `repeat` is non-zero), the endflags will be posted. But rather than setting the complete variable to true and thus never receiving any further run consideration, the behavior is reset to its initial state. Presumably the user sets endflags that will cause the condition flags to be not immediately satisfied, thus putting the behavior in a state waiting again for an external event flag to be posted. The default value of this parameter is false.

34 The MemoryTurnLimit Behavior

The objective of the Memory-Turn-Limit behavior is to avoid vehicle turns that may cross back on its own path and risk damage to the towed equipment. Its configuration is determined by the two parameters described below which combine to set a vehicle turn radius limit. 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 multi-objective optimization process. The following parameters are defined for this behavior:

34.1 Configuration Parameters

Listing 34.10: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].

templating: Turns a behavior into a template for spawning behaviors dynamically. [more].

updates: A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 34.11: Configuration Parameters for the MemoryTurnLimit Behavior.

Parameter	Description
memory_time:	The duration of time for which the heading history is maintained and heading average calculated. The default value is -1, indicating that the parameter is un-set. In this case the behavior will not produce an objective function.
turn_range:	The range of heading valuesdeviating from the current heading average outside of which the behavior reflects sharp penalty in its objective function. The default value is -1, indicating that the parameter is un-set. In this case the behavior will not produce an objective function.

Listing 34.12: Example Configuration Block.

```
Behavior = BHV_MemoryTurnLimit
{
  // General Behavior Parameters
  // ---------------------------
  name         = mem_turn_limit       // default
  pwt          = 100                  // default
  condition    = MODE=TRANSITING      // example
  condition    = ARRAY=connected      // example
  updates      = MEM_TURN_UPDATES     // example

  // Parameters specific to this behavior
  // ------------------------------------
        memory_time = 60              // example (seconds)
         turn_range = 30              // example (degrees)
}
```

34.2 Calculation of the Heading History

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 $[0, 359]$.

$$\text{heading\_avg} = \text{atan2}(s, c) \cdot 180/\pi,$$

where s and c are given by:

$$s = \sum_{k=0}^{n-1} \sin\left(h_k \pi/180)\right), \qquad c = \sum_{k=0}^{n-1} \cos\left(h_k \pi/180)\right).$$

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:

$$u = \texttt{turn\_range}/\texttt{memory\_time}.$$

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.

34.3 Variables Published

The only variable published by this behavior is MEM_TURN_AVG which indicates the present heading average, rounded to the nearest integer.

34.4 The MemoryTurnLimit Objective Function

A Rendering of the MemoryTurnLimit Objective Function

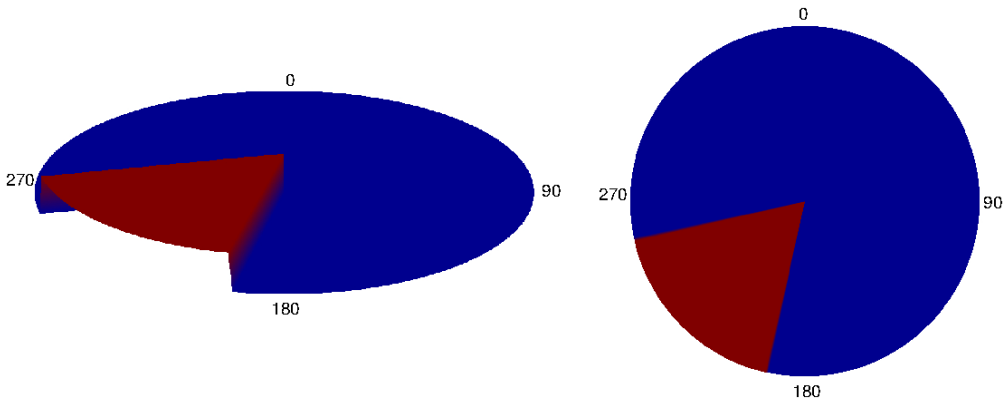

Figure 99: **The MemoryTurnLimit objective function**: The objective function produced by the MemoryTurnLimit behavior is defined over possible heading values. Depicted here is an objective function formed when the recent heading history is 225 degrees and the turn_range parameter is set to 30 degrees. The resulting objective function highly favors headings in the range of 190-240 degrees. One the right is a "birds-eye" view of the function, and on the right the function is viewed at an angle to appreciate the 3D quality of the function. Higher (red) values correspond to higher utility.

35 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.

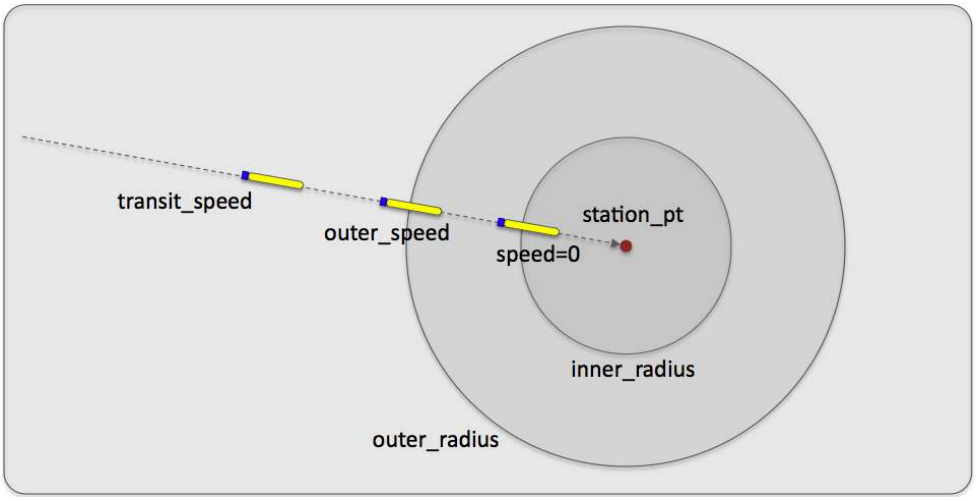

Figure 100: **The station-keep behavior parameters:** The station-keep behavior can be configured to approach the outer station circle with a given transit speed, and will decrease its preference for speed linearly between the outer radius and inner radius. The preferred speed is zero when the vehicle is at or inside the inner radius.

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 pre-set 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. It's worth pointing out that a vehicle's control system, i.e., the front-seat driver described in Section 2.3, 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's also worth pointing out that most UUVs are positively buoyant and will simply come to the surface if commanded with a zero-speed.

35.1 Configuration Parameters

Listing 35.13: Configuration Parameters Common to All Behaviors.

`activeflag`: A MOOS variable-value pair posted when the behavior is in the *active* state. [more].

`condition`: Specifies a condition that must be met for the behavior to be running. [more].

duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 35.14: Configuration Parameters for the StationKeep Behavior.

Parameter	Description
center_activate:	If true, station-keep at the vehicle's present position upon activation. Section 35.2.
hibernation_radius:	A radius used for low-power, passive station-keeping. Section 35.5.
inner_radius:	Distance to station-point within which the preferred speed is zero. Section 35.4.
outer_radius:	Distance within which the preferred speed begins to decrease. Section 35.4.
outer_speed:	Preferred speed at outer radius, decreasing toward inner radius. Section 35.4.
station_pt:	An x,y pair given as a point in local coordinates. Section 35.2.
swing_time:	Duration of drift of station circle with vehicle upon activation. Section 35.3.
transit_speed:	Preferred speed beyond the outer radius. Section 35.4.
visual_hints:	Preferences for rendering visual artifacts produced by the behavior. Section 35.7.

Listing 35.15: Example Configuration Block.

```
Behavior = BHV_StationKeep
{
  // General Behavior Parameters
  // ---------------------------
  name         = station-keep          // example
  pwt          = 100                   // default
  condition    = MODE==SKEEPING        // example
  inactiveflag = STATIONING = false    // example
  activeflag   = STATIONING = true     // example


  // Parameters specific to this behavior
  // ------------------------------------
     center_activate = false   // default
  hibernation_radius = -1      // default
        inner_radius = 4       // default
        outer_radius = 15      // default
        outer_speed  = 1.2     // default
     transit_speed  = 2.5      // default
         station_pt = 0,0      // default
         swing_time = 0        // default

        visual_hints = vertex_size  = 1           // default
        visual_hints = edge_color   = light_blue  // default
        visual_hints = edge_size    = 1           // default
        visual_hints = label_color  = white       // default
        visual_hints = vertex_color = red         // default
}
```

35.2 The station_pt and center_activate Parameters

The station-keep point is set in one of two ways: either with a pre-specified 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_activate = true  // "true" is case insensitive
```

35.3 The swing_time Parameter

At the outset of station-keeping via center_activate, 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 "over-shooting" 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 seconds
```

The time duration is given in seconds and should be in the range [0, 60]. If found to be outside this range it it simply clipped to the boundary value.

If the behavior enters the running state, but center-activation is not set to true, and no pre-specified fixed position is given, the behavior will not produce an objective function. It will remain in the running state, but not the active state. (Section 6.5.3 discusses run states.) In this situation, a warning will be posted via the helm appcast structure and by posting to the BHV_WARNING variable. In both cases the warning will read: "STATION_POINT_NOT_SET".

35.4 The inner_radius, outer_radius, and outer_speed Parameters

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 than the outer, this will not trigger an error, but the two radii parameters will be collapsed to the value of the inner radius on the first iteration of the behavior. The transit_speed parameter indicates the desired speed when the vehicle is outside the outer_radius. The default value for transit_speed is 2.5 meters per second. If the outer_speed is set higher than the transit_speed the transit_speed will automatically be raised to the outer_speed.

35.5 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 non-zero 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 hibernation_radius before it resumes outputting a preferred heading and non-zero speed. The idea is shown in Figure 101.

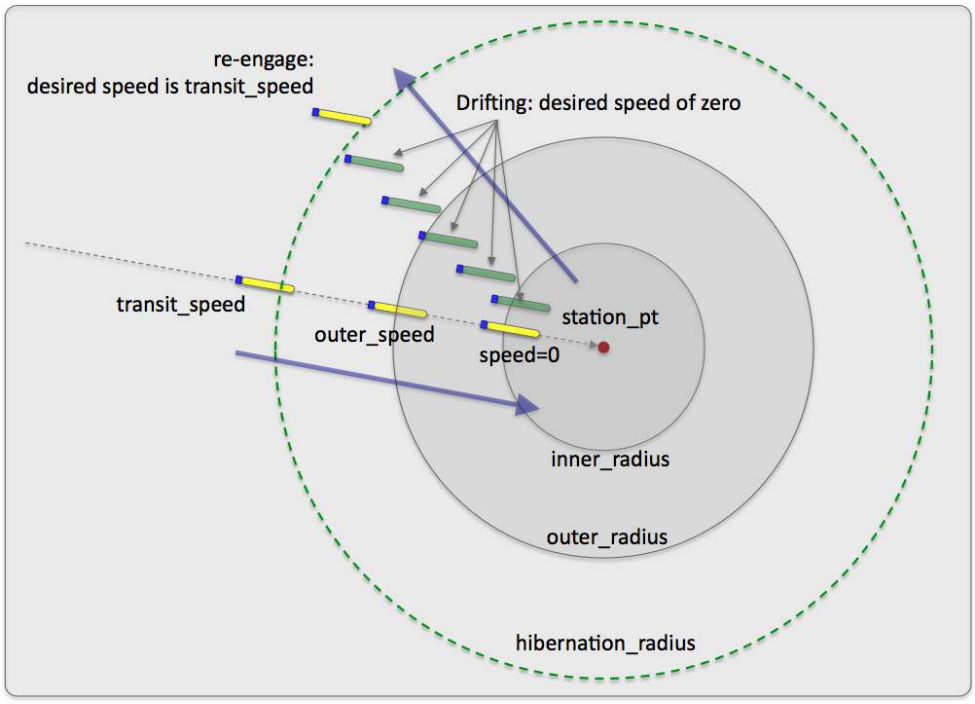

Figure 101: **Passive station-keeping:** The station-keep behavior can be configured in the "passive" mode. The vehicle will move toward the station point until it reaches the `inner_radius` or until progress ceases. It will then drift until its distance to the station point is beyond the `hibernation_radius`. At this point it will re-engage to reach the station-point and may trigger another behavior to dive.

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 non-zero 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 shown in Figure 102.

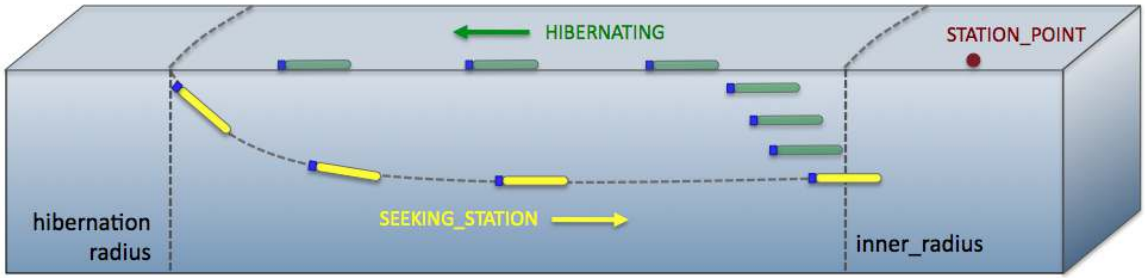

Figure 102: **Passive station-keeping with depth coordination:** The passive mode can be coordinated with the ConstantDepth behavior to dive each time the StationKeep behavior enters the "SEEKING_STATION" mode. This ensures that a UUV needing to be at depth to have reliable heading control will indeed be at depth when it needs to be.

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, that 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 where the vehicle is neither progressing toward or way from the inner_radius, or by 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" mode when it detects the vehicle speed toward the station-point reach zero. To calculate this vehicle speed, a ten-second history of range to the station-point is kept by the behavior. A zero speed, or "stale-progress" criteria 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 ten seconds ago in the history list. The behavior will transition into the "HIBERNATING" mode if either the inner-radius or stale-progress criteria are met.

It is also possible that when the StationKeep behavior enters the "SEEKING_STATION" mode from the "HIBERNATING" mode, that 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's quite possible that the behavior would correctly, but unwantingly, infer that the stale-progress criteria has been met. For this reason, the stale-progress criteria is not applied until an "initial-progress" criteria is met after entering the "SEEKING_STATION" mode. The same ten second history is used to detect when the vehicle begins to make initial progress, i.e., closing range, toward the station-point.

35.6 Station Keeping On Demand

A common, and perhaps recommended configuration, is to have one station-keep behavior defined for a given helm configuration and have it set to be usable in one of three ways: (a) station-keep at a default pre-specified position, (b) station-keep at a specified position dynamically provided, or (c)

station-keep at the vehicle's present position when activated. The behavior would be configured as follows:

```
station_pt      = 100,200 // The default station-keep point
center_activate = false
updates         = STATION_UPDATES
condition       = STATION_REQUEST = true
```

Then, to use the station-keep behavior in the above three ways, the following three pairs of postings, i.e., pokes, to the MOOSDB would be used. See Section 7.2.5 for more on the `updates` parameter defined for all behaviors - by utilizing this dynamic configuration hook, the one behavior configuration above can be used in these different manners. The first pair would result in the behavior keeping station at its pre-arranged point of (100,200):

```
STATION_REQUEST = true
STATION_UPDATES = center_activate=false
```

The second line above dynamically configures the behavior parameter `center_activate` to be false to ensure that the point given by the original `station_pt` parameter is used. Even though the `center_activate` parameter is initially set to false, the above usage sets it to false anyway, to be safe, and in case it has been dynamically set to true in a prior usage.

In the second case below, again the `center_activate` parameter is dynamically set to false for the same reasons. In this case the `station_point` parameter is also dynamically configured with a given point:

```
STATION_REQUEST = true
STATION_UPDATES = station_pt=45,-150 # center_activate=false
```

In the last case, below, the behavior is activated and configured to station-keep at the vehicle's present position when activated. There is no need to tinker with the `station_pt` parameter since this parameter is ignored when `center_activate` is true:

```
STATION_REQUEST = true
STATION_UPDATES = center_activate=true
```

It's worth noting that above variable-value pairs that trigger the StationKeep behavior could have come from a variety of sources. They could be endflags from another behavior. They could have come from a poke using `uPokeDB`, `uTimerScript`, `pMarineViewer` or any third party command and control interface.

35.7 The `visual_hints` Parameter

Although the primary output of the StationKeep behavior is an IvP Function, a number of visual properties are also published for convenience in mission monitoring. This includes (a) the `inner_radius`, (b) the `outer_radius`, and (c) the `hibernation_radius` if used. These visual artifacts

have default properties in size and color that may be altered to the user's preferences. These preferences are configurable through the visual_hints parameter. Each parameter below is used in the following way by example:

```
visual_hint = vertex_size=3, edge_size=2
visual_hint = vertex_color=khaki
```

- edge_color: The color of edges rendered in the loiter polygon. The default is "white".
- edge_size: The width of edges rendered in the loiter polygon. The default is 1.
- label_color: The color of labels rendered with the innner and outer radii. The default is "gray50".
- vertex_color: The color of vertices rendered in the loiter polygon. The default is "dodger_blue".
- vertex_size: The size of vertices rendered in the loiter polygon. The default is 1.

Rendering of vertices may be shut off with a size of zero, and labels may be shut off with the special color "invisible". For a list of legal colors, see Appendix 21.

36 The Timer Behavior

The Timer behavior is a somewhat unique behavior in that it never produces an objective function. It has virtually no functionality beyond what is derived from the parent IvPBehavior class. It can be used to set a timer between the observation of one or more events (with condition flags) and the posting of one or more events (with end flags). The `duration`, `duration_status`, `duration_idle_decay`, `condition`, `runflag` and `endflag` parameters are all defined generally for behaviors. There are no additional parameters defined for this behavior.

36.1 Configuration Parameters

Listing 36.16: Configuration Parameters Common to All Behaviors.

`activeflag`:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
`condition`:	Specifies a condition that must be met for the behavior to be running. [more].
`duration`:	Time in behavior will remain running before declaring completion. [more].
`duration_idle_decay`:	When true, duration clock is running even when in the *idle* state. [more].
`duration_reset`:	A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. [more].
`duration_status`:	The name of a MOOS variable to which the vehicle duration status is published. [more].
`endflag`:	A MOOS variable-value pair posted when the behavior has completed. [more].
`idleflag`:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
`inactiveflag`:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
`name`:	The (unique) name of the behavior. [more].
`nostarve`:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
`perpetual`:	If true allows the behavior to to run even after it has completed. [more].
`post_mapping`:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
`priority`:	The priority weight of the behavior. [more].
`pwt`:	Same as `priority`.
`runflag`:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
`spawnflag`:	A MOOS variable and a value posted when a behavior is spawned. [more].
`spawnxflag`:	A MOOS variable and a value posted when a behavior is spawned. [more].
`templating`:	Turns a behavior into a template for spawning behaviors dynamically. [more].

`updates`: A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 36.17: Example Configuration Block.

```
Behavior = BHV_Timer
{
  // General Behavior Parameters
  // ---------------------------
  name         = bhv_timer            // default
  condition    = WAIT_REQUEST=true    // example
  duration     = 120                  // example
  idleflag     = WAITING=false        // example
  runflag      = WAITING=true         // example
  runflag      = WAITED=false         // example
  endflag      = WAITED=true          // example
  updates      = TIMER_UPDATES        // example

  // Parameters specific to this behavior
  // ------------------------------------
  // None
}
```

36.2 Variables Published

This behavior publishes two variables for monitoring and logging performance - `TIMER_IDLE`, `TIMER_RUNNING`.

37 The TestFailure Behavior

The TestFailure behavior is used to test the helm in two conceivable behavior failure modes. First, it may be used to simulate a behavior that crashes and thereby results in the crash of the helm. Second, it may be used to simulate a behavior that consumes a sufficiently large enough amount of time so as cause the helm to be considered "hung" by consumers of the helm output.

Recall that the helm is compiled, with behaviors, into a single MOOS application. Although some behaviors may be compiled into shared libraries loaded at run time, thereby not requiring a recompile, all behaviors do run as part of a single helm process. *A crashed behavior results in a crashed helm.* Furthermore the helm, on each iteration, queries each participating behavior for its input. It does not do this in separate threads, and there is no timeout with a default reply should a behavior never answer. *A hung behavior results in a hung helm.* These are architecture decisions that on one hand allow a substantial amount of simplicity in the helm implementation and debugging. Furthermore, it's not clear that a graceful and safe policy exists to safely handle a rogue behavior other than to either (a) abort the mission or (b) put the vehicle in the hands of a much more conservatively configured "standby" instance of the helm, perhaps just to get the vehicle home. This behavior is used to simulate both kinds of rogue behaviors, a behavior that crashes and a behavior the hangs. The crash is implemented simply with an `assert(0)` statement, and the hang is implemented with a long for-loop.

37.1 Configuration Parameters

Listing 37.18: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].

post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 37.19: Configuration Parameters for the TestFail Behavior.

Parameter	Description
failure_type:	Fail by `crash`, or fail by `hang`.

Choosing to fail with a *crash* will result in behavior executing an `assert(0)` as soon as the behavior enters the running state. Configuring for a *hang* will likewise result in the execution of a long for-loop upon entering the running state. The hang time is by default three seconds, but may be altered by specifying a time alongside the hang parameter as in line 9 below.

Below are a couple typical usage configurations. In the first case, the TestFailure behavior is set with a condition as on line 4, to remain idle until the vehicle is deployed. The duration parameter on line 6, defined for all helm behaviors, ensures the behavior will not run until two minutes into the mission. This is done presumably to have the failure occur while the vehicle is "in the middle of doing something". Setting the duration_idle_decay parameter to false indicates the duration countdown is not to proceed while the behavior is still idle. In this case the failure type is set to hang for two seconds. This behavior configuration is used in the Kilo example mission, described in Section XYZ, and can be run to illustrate.

Listing 37.20: Example Configuration Block.

```
Behavior = BHV_TestFailure
{
  // General Behavior Parameters
  // ---------------------------
  name      = test_failure         // default
  condition = DEPLOY=true          // example
  duration  = 120                  // example
  duration_idle_decay = false      // example (default is true)


  // Parameters specific to this behavior
  // ------------------------------------
  failure_type = hang,2            // example
}
```

Another configuration style below will result in the behavior crashing as soon as the behavior meets its run condition. Presumably the user can simply poke the MOOSDB at any time to invoke the failure, or another MOOS app may generate the poke at the key desired moment. The `failure_type` is left unspecified since `crash` is the default failure type.

Listing 37.21: Example Configuration Block.

```
Behavior = BHV_TestFailure
{
  // General Behavior Parameters
  // ---------------------------
  condition = MUD_HITS_THE_FAN = true
}
```

38 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 closest point of approach (CPA) between the candidate maneuver leg, and the contact leg formed from the contact's position and trajectory. A further user-defined utility function is applied to the CPA calculation for a candidate maneuver.

38.1 Configuration Parameters

Listing 38.22: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].

templating: Turns a behavior into a template for spawning behaviors dynamically. [more].

updates: A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 38.23: Configuration Parameters Common to Contact Behaviors.

Parameter	Description
bearing_lines	If true, a visual artifict will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.
contact	Name or unique identifier of a contact to be avoided.
decay	Time interval during which extrapolated position slows to a halt.
exit_on_filter_group	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_vtype	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_region	It true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default it false.
extrapolate	If true, contact position is extrapolated from last position and trajectory.
ignore_group:	If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 15.2
ignore_name:	If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 15.2
ignore_region:	If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 15.2
ignore_type:	If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 15.2
match_group:	If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 15.2
match_name:	If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 15.2

match_region:	If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regins. Introduced after Release 19.8.1. Section 15.2
match_type:	If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 15.2
on_no_contact_ok	If false, a helm error is posted if no contact information exists. Applicable in the more rare case that a contact behavior is statically configured for a named contact. The default is true.
strict_ignore	If true, and if one of the ignore exclusion filter components is enabled, then an exclusion filter will fail if the contact report is missing information related to the filter. For example if the contact group information is unknown. The default is true.
time_on_leg	The time on leg, in seconds, used for calculating closest point of approach.

Listing 38.24: Configuration Parameters for the AvoidCollision Behavior.

Parameter	Description
completed_dist:	Range to contact outside of which the behavior completes and dies. The default is 500 meters.
max_util_cpa_dist:	Range to contact outside which a considered maneuver will have max utility. Section 38.6. The default is 75 meters
min_util_cpa_dist:	Range to contact within which a considered maneuver will have min utility. Section 38.6. The default is 10 meters.
no_alert_request:	If true, the behavior will not send an automatic configuration message to the contact manager at startup. Section 38.4. The default is false.
pwt_grade:	Grade of priority growth as the contact moves from the pwt_outer_dist to the pwt_inner_dist. Choices are `linear`, `quadratic`, or `quasi`. The default is `quasi`. Section 38.5.
pwt_inner_dist:	Range to contact within which the behavior has maximum priority weight. Section 38.5. The default is 50 meters.
pwt_outer_dist:	Range to contact outside which the behavior has zero priority weight. Section 38.5. The default is 200 meters.
use_refinery:	If true, the behavior will produce an optimized objective function that is faster to produce, uses a smaller memory footprint, and contributes to faster helm solution time. The default is false, simply for continuity with prior releases, but there is no downside to enabling this feature. Section 38.8.

Listing 38.25: Example Configuration Block.

```
Behavior = BHV_AvoidCollision
{
  // General Behavior Parameters
  // ---------------------------
  name         = avdcollision_                     // example
  pwt          = 200                               // example
  condition    = AVOID = true                      // example
  updates      = CONTACT_INFO                      // example
  endflag      = CONTACT_RESOLVED = $[CONTACT]     // example
  templating   = spawn                             // example


  // General Contact Behavior Parameters
  // -----------------------------------
      bearing_lines = white:0, green:0.65, yellow:0.8, red:1.0   // example

            contact = henry            // example
              decay = 15,30            // default (seconds)
        extrapolate = true             // default
   on_no_contact_ok = true             // default
         time_on_leg = 60              // default (seconds)


  // Parameters specific to this behavior
  // ------------------------------------
     completed_dist = 500              // default
  max_util_cpa_dist = 75               // default
  min_util_cpa_dist = 10               // default
   no_alert_request = false            // default
          pwt_grade = quasi            // default
     pwt_inner_dist = 50               // default
     pwt_outer_dist = 200              // default
}
```

38.2 Variables Published

The below MOOS variables 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 described in Section 7.2.8.

- BCM_ALERT_REQUEST: A request to the contact manager specifying the conditions for contact alerts.
- CLOSING_SPD_AVD: The current closing speed, in meters per second, to the contact.
- CONTACT_RESOLVED: Posted with contact name when the behavior completes and dies.
- RANGE_AVD: The current range, in meters, to the contact.
- VIEW_SEGLIST: A bearing line between ownship and the contact if configured for rendering.

38.3 Configuring and Using the AvoidCollision Behavior

The AvoidCollision behavior produces an objective function based on the relative positions and trajectories between the vehicle and a given contact. The objective function is based on applying a

utility to the calculated closest point of approach (CPA) for a candidate maneuver. The user may configure a priority weight, but this weight is typically degraded as a function of the range to the contact. The behavior may be configured for avoidance with respect to a known contact, or it may be configured to spawn a new instance upon demand as contacts present themselves.

38.4 Automatic Requests for Contact Manager Alerts

The collision avoidance behavior is most commonly configured as a *template*, meaning instances will not be spawned until an outside event, i.e, posting to the MOOSDB, is received by the helm. This was discussed in detail in Section 7.7. The spawning event for the collision avoidance behavior typically comes from the pBasicContactMgr application. This app therefore needs to be informed, by the collision avoidance behavior, of the desired conditions for generating behavior-spawning alerts. This is done automatically upon helm startup with a posting of the form:

```
BCM_ALERT_REQUEST = id=avd, onflag=CONTACT_INFO=name=$[VNAME] # contact=$[VNAME],
                    alert_range=80, cpa_range=100
```

The values for this posting are chosen as follows:

- The value from the `onflag` component is a posting to be made by the contact manager when an alert is triggered. Like all MOOS postings, the posting has a variable and value component. The variable component is `CONTACT_INFO`. The value component is

  ```
  name=$[VNAME] # contact=$[VNAME]
  ```

- When the contact manager posts the onflag, it will expand the `$[VNAME]` macros with the actual name of the contact. The variable `CONTACT_INFO` is the variable provided in the `updates` parameter for the behavior.
- The value from the `alert_range` component, e.g., 80 in the example above, is the value specified in the `pwt_outer_dist` parameter for the behavior. This is the range between ownship and contact beyond which the behavior assigns a priority weight of zero.
- The value from the `cpa_range` component, e.g., 100 in the example above, is the value specified in the `completed_dist` parameter for the behavior. This is the range, in meters, between ownship and contact beyond which the behavior will initiate its own completion and de-instatiation.

If some other contact manager regime is being used other than pBasicContactMgr, the above automatic posting is probably harmless. However, if one really wants to disable this automatic posting, it can be turned off by setting the configuration parameter `no_alert_request` to true.

Implementation note: One may wonder when or how the behavior can make this automatic posting when the behavior is configured as a template. An instance is never spawned until an alert is received, but the alert parameters are posted by the behavior, creating a bit of a chicken or the egg conundrum. The alert request is actually posted by an instance of the behavior created ever so briefly at helm startup. At startup, the helm creates instances of *all* behaviors, even templates, to ensure the configuration parameters are correct. The ultimate confirmation of behavior parameter correctness is obtained by a behavior instance itself confirming each parameter. The helm will then immediately, before its first iteration, delete any behaviors temporarily created from template

behaviors. During this brief startup period, the helm will invoke the function `onHelmStart()` for all behaviors. This function is defined at the IvPBehavior superclass level just like `onRunState()`. In the case of the collision avoidance behavior, this function is implemented to make the automatic alert configuration posting to `BCM_ALERT_REQUEST`.

38.5 Specifying the Behavior Priority Weight Policy

The AvoidCollision behavior may be configured to increase its priority as it closes range to the contact. The priority weight specified in its configuration represents the *maximum* possible priority applied to the behavior, presumably in close range to the contact. The range at which this maximum priority applies is specified in the `pwt_inner_dist` parameter. Likewise, the `pwt_outer_dist` parameter specifies a range to the contact where the priority weight becomes zero, regardless of the priority weight specified in the configuration file.

So the *current priority* will always be between zero and the maximum priority set in the behavior `priority` configuration parmeter. To be more precise:

Current Priority =

- 0 if current range to contct is greater than or equal to `pwt_outer_dist`
- 100 if current range to contact is less than or equal to `pwt_inner_dist`
- otherwise ((`pwt_outer_dist` - current range) / (`pwt_outer_dist` - `pwt_inner_dist`)) * priority

This relationship is shown in Figure 103.

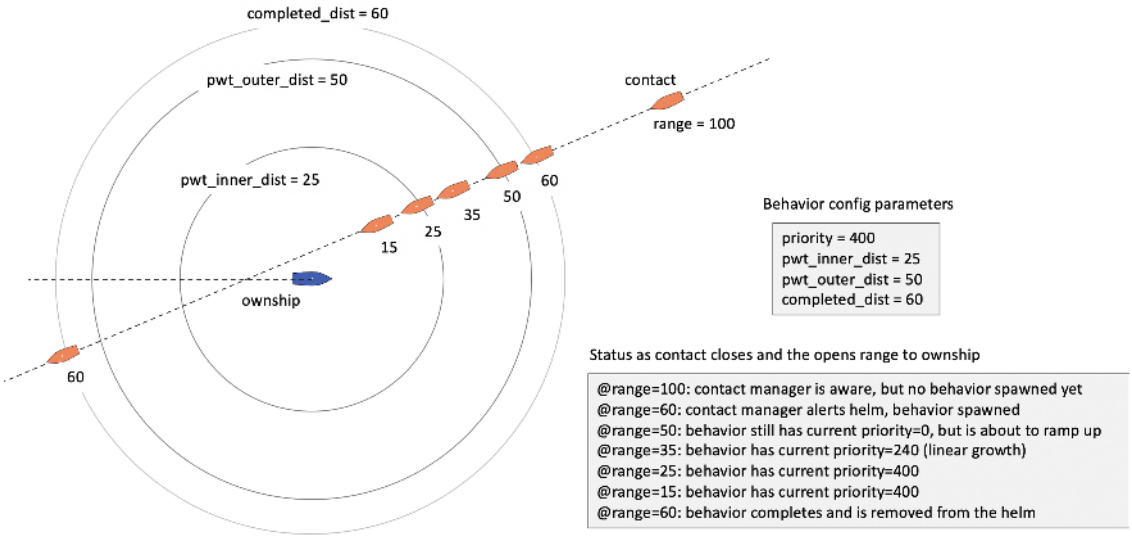

Figure 103: **Scaling priority weights based on ownship range to contact**: The range between the two vehicles affects whether the behavior is spawned, active and with what priority weight. Beyond the range specified by `completed_dist` the behavior is likely not inexistence. Beyond the range specified by `pwt_outer_dist`, the behavior is not yet active. Within the range of `pwt_outer_dist`, the behavior becomes active with a non-zero priority weight growing as the contact closes range. Within the range of `pwt_inner_dist`, the behavior is active with 100% of its configured priority weight.

The example shown below in Figure 104 shows the effect of the pwt_outer_dist parameter. The vehicle on the left is proceeding east, oblivious to the two approaching vessels. The two westbound vessels, ben and cal are simulated exactly on top of one another. They are oblivious to one another, but will use the collision avoidance behavior to avoid the eastbound vessel, abe. The only difference between ben and cal is that cal begins winding up its priority weight at 80 meters range to abe, as opposed to 30 meters for ben. The short simulation shows the resulting difference in trajectory.

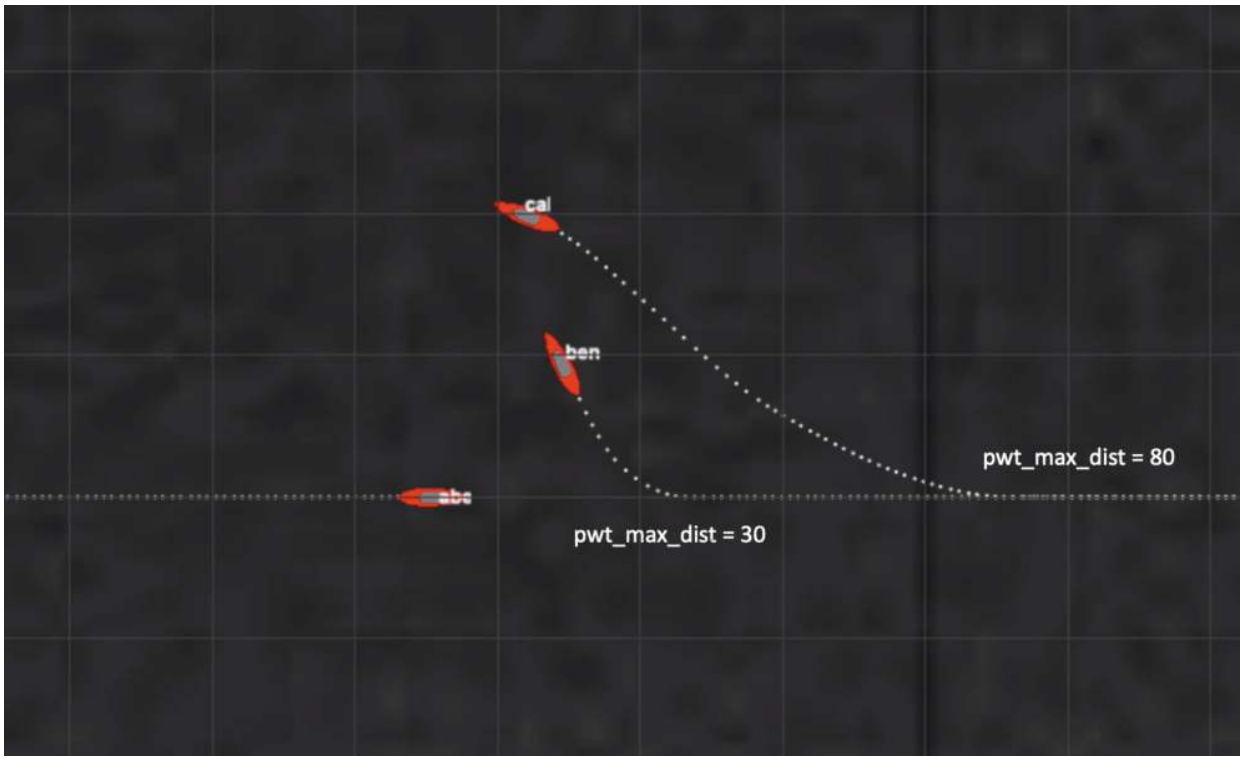

Figure 104: Two vehicles use the AvoidCollision behavior to avoid the oblivious and non-maneuvering Eastbound vehicle. One vehicle has pwt_outer_dist set to 80 and the other set to 30. This affects when each vehicle begins its maneuver to avoid.
video:(0:12): https://vimeo.com/457500757

By default, the priority weight decreases linearly between the two depicted ranges. The pwt_grade parameter allows the degradation from maximum priority to zero priority to fall more steeply by setting pwt_grade=quadratric.

38.6 Specifying the Utility Policy of the Behavior

Whereas the behavior *weight* discussed above in Section 38.5 determines the influence of the behavior relative to other behaviors, the behavior *utility function* specifies the relative utility of candidate maneuvers from the perspective of the collision avoidance goals of this behavior.

The utility function for the avoid collision behavior is based on two factors:

- The range at the closest point of approach (CPA) for a candidate maneuver

- The determination risk associated with any CPA range.

The first component is just physics. Given ownship and contact current positions and trajectories, and the assumption that the contact will stay on its current heading and speed for at least the near future, the projected CPA range may be calculated for any given heading-speed maneuver. Consider the example in Figure 105 below with particular ownship and contact positions and trajectories shown on the left. On the right, the projected CPA range values are plotted for all candidate ownship maneuvers.

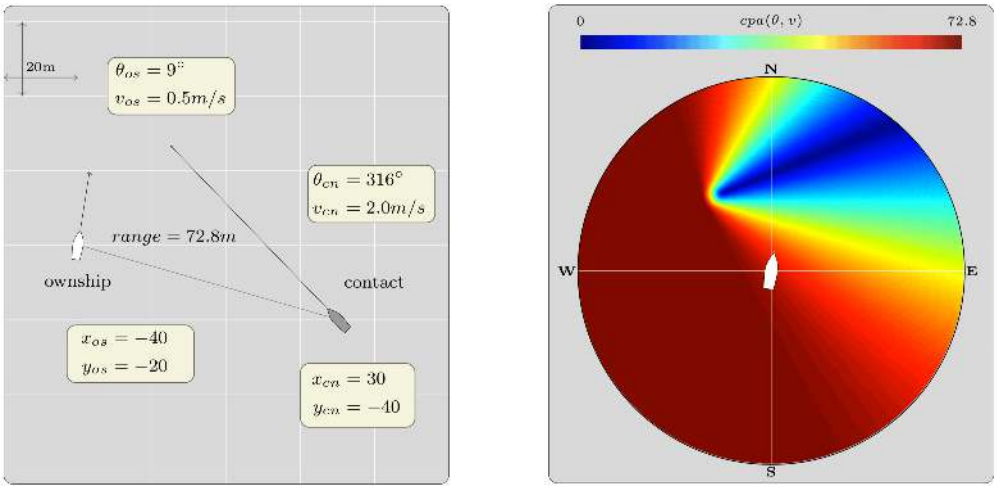

Figure 105: (left) An example collision avoidance situation, where ownship would cross behind the contact if it remained on its present heading and speed. (right) The evaluation of closest point of approach for all possible maneuvers with ownship max speed of 4 m/s. The darkest blue maneuvers will result in a collision or near collision.

The second component of the utility function is the mapping of "risk" to certain CPA range values. This part is very subjective, and of course is why there are configuration parameters, allow different users to align the behavior with their own notion of risk. The two key parameters are `min_util_cpa_dist` and `max_util_cpa_dist`. The former refers the the CPA range with the minimum utility, essentially equivalent to a collision. For this value, perhaps pick a range that, while not a collision, someone would get written up for a safety violation for breaching this range. The `max_util_cpa` range, on the other hand, is the range that, above which there is not increase in utility. Everything at this range or higher is "all clear".

These two parameters form a function, $g()$, which takes as input the CPA range value and outputs the utility of a canditate maneuver. This function is depicted on the left in Figure 106, and the overall utility function is shown on the right:

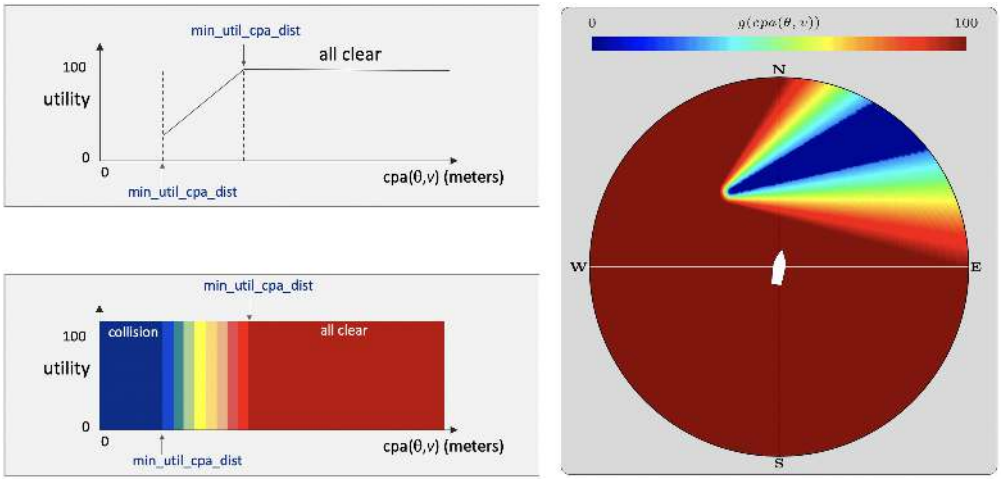

Figure 106: (left) A utility function mapping CPA range values to a single risk utility is shown. Up to the range specified by the `min_util_cpa` configuration parameter, maneuvers resulting in these ranges are considered essentially collisions. Above the range specified by the `max_util_cpa` configuration parameter, maneuvers resulting in these ranges are considered essentially equivalently optimal. In between are the ranges that are in between disaster and optimal. These represent compromise maneuvers that the helm may opt for if there are other behaviors that find such maneuvers useful for accomplishing their objectives.

The `min_util_cpa_dist` and `max_util_cpa_dist` parameters have a default value of 10 and 75 meters respectively. These values are very subjective and will need to be adjusted per vehicle and mission. Currently these defaults are used if not specified by the user, but in future releases overriding these values may be enforced.

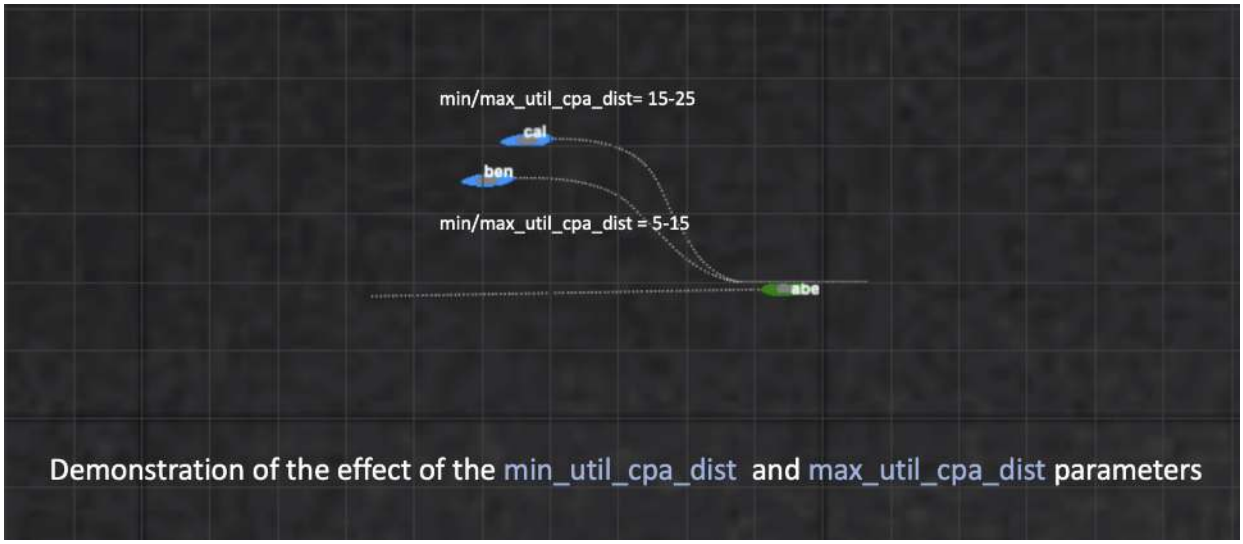

Figure 107: Two vehicles use the AvoidCollision behavior to avoid the oblivious and non-maneuvering eastbound vehicle. One vehicle, ben, is configured with the `min_util_cpa_dist` and `max_util_cpa_dist` paremeters set to 5 and 15 meters respectively. The other vehicle, cal, is more risk averse and has these to parameters set to 15 and 25 meters respectively. The resulting trajectories of ben and cal are shown. The two vehicles on the right are simulated separately, unaware of each other, with the same starting position and parameters except for two above parameters. The divergence in trajectory is soley due to the differences around these two parameters.
video:(0:13): https://vimeo.com/458207817

38.7 Specifying Contact Flags

Contact flags are new feature for all contact behaviors, available in releases after Release 19.8.1.

38.8 Using the CPA Refinery

38.9 Relevant Example Missions

39 The AvdColregs Behavior

The AvdColregs behavior will produce IvP objective functions designed to avoid collisions (and near collisions) with another specified vehicle, based on the protocol found in the US Coast Guard Collision Regulations (COLREGS) citecolregs. 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 closest point of approach (CPA) between the candidate maneuver leg, and the contact leg formed from the contact's position and trajectory. A further user-defined utility function is applied to the CPA calculation for a candidate maneuver.

Note: This behavior is invoked as `BHV_AvdColregsV17` in mission files. The `m2_berta_detect` mission is an example simulation mission for this behavior. Note finally that the configuration interface for this behavior is very similar to `BHV_AvoidCollision`, even though the effects of the behavior are quite different. Documenation is evolving for this behavior. If you would like a more detail discussion of the algorithms implementing the COLREGS behavior, please see MIT CSAIL TR-2017-09. As this remains a very active area of research in this lab, further releases and documentation are expected later in 2017 and beyond.

39.1 Configuration Parameters

Listing 39.26: Configuration Parameters for the AvdColregs Behavior.

Parameter	Description
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. Section 38.6.
giveway_bow_dist:	In the giveway consideration, when determining if ok to cross the contact's bow, this range to contact at time of crossing from contact's port to starboard, must be met in order for ownship to consider crossing the contac'ts bow. This is a new option in V19 of this vehavior. If left unspecified, it will revert to the criteria used in previous versions.
min_util_cpa_dist:	Range to contact within which a considered maneuver will have min utility. Section 38.6.
no_alert_request:	If true will turn off automatic handshaking with the contact manager. Section 38.4.
pwt_grade:	Grade of priority growth as the contact moves from the pwt_outer_dist to the pwt_inner_dist. Choices are `linear`, `quadratic`, or `quasi`. Section 38.5.
pwt_inner_dist:	Range to contact within which the behavior has maximum priority weight. Section 38.5.
pwt_outer_dist:	Range to contact outside which the behavior has zero priority weight. Section 38.5.
refinery:	If true, the behavior will create an IvP Function using the refinery, using large pieces for plateau regions. This is a new feature after Release 19.8, and currently only is used in the CPA or in-extremis modes. The default is false.

Listing 39.27: Example Configuration Block.

```
Behavior = BHV_AvdColregsV17
{
  // General Behavior Parameters
  // ---------------------------
  name         = avdcollision_                    // example
  pwt          = 200                              // example
  condition    = AVOID = true                     // example
  updates      = CONTACT_INFO                     // example
  endflag      = CONTACT_RESOLVED = $[CONTACT]    // example
  templating   = spawn                            // example


  // General Contact Behavior Parameters
  // -----------------------------------
      bearing_lines = white:0, green:0.65, yellow:0.8, red:1.0   // example

            contact = henry            // example
              decay = 15,30            // default (seconds)
        extrapolate = true             // default
   on_no_contact_ok = true             // default
         time_on_leg = 60              // default (seconds)


  // Parameters specific to this behavior
  // ------------------------------------
     completed_dist = 500              // default
  max_util_cpa_dist = 75               // default
  min_util_cpa_dist = 10               // default
   no_alert_request = false            // default
          pwt_grade = quasi            // default
     pwt_inner_dist = 50               // default
     pwt_outer_dist = 200              // default
}
```

39.2 Variables Published

The below MOOS variables 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 described in Section 7.2.8.

- BCM_ALERT_REQUEST: A request to the contact manager specifying the conditions for contact alerts.
- CLOSING_SPD_AVD: The current closing speed, in meters per second, to the contact.
- CONTACT_RESOLVED: Posted with contact name when the behavior completes and dies.
- RANGE_AVD: The current range, in meters, to the contact.
- VIEW_SEGLIST: A bearing line between ownship and the contact if configured for rendering.

39.3 Configuring and Using the AvdColregs Behavior

The AvdColregs behavior produces an objective function based on the relative positions and trajectories between the vehicle and a given contact. The objective function is based on applying a

utility to the calculated closest point of approach (CPA) for a candidate maneuver. The user may configure a priority weight, but this weight is typically degraded as a function of the range to the contact. The behavior may be configured for avoidance with respect to a known contact, or it may be configured to spawn a new instance upon demand as contacts present themselves.

39.4 Automatic Requests for Contact Manager Alerts

The collision avoidance behavior is most commonly configured as a *template*, meaning instances will not be spawned until an outside event, i.e, posting to the MOOSDB, is received by the helm. This was discussed in detail in Section 7.7. The spawning event for the collision avoidance behavior typically comes from the pBasicContactMgr application. This app therefore needs to be informed, by the collision avoidance behavior, of the desired conditions for generating behavior-spawning alerts. This is done automatically upon helm startup with a posting of the form:

```
BCM_ALERT_REQUEST = id=avd, var=CONTACT_INFO, val="name=$[VNAME] # contact=$[VNAME]",
                    alert_range=80, cpa_range=100
```

The values for this posting are chosen as follows:

- The value from the `var` component, e.g., `CONTACT_INFO` in the example above, is the variable named in the `updates` parameter for the behavior.
- The value from the `alert_range` component, e.g., 80 in the example above, is the value specified in the `pwt_outer_dist` parameter for the behavior. This is the range between ownship and contact beyond which the behavior assigns a priority weight of zero.
- The value from the `alert_range_cpa` component, e.g., 100 in the example above, is the value specified in the `completed_dist` parameter for the behavior. This is the range between ownship and contact beyond which the behavior will initiate its own completion and de-instatiation.

If some other contact manager regime is being used other than pBasicContactMgr, the above automatic posting is probably harmless. However, if one really wants to disable this automatic posting, it can be turned off by setting the configuration parameter `no_alert_request` to true.

Implementation note: One may wonder when or how the behavior can make this automatic posting when the behavior is configured as a template. An instance is never spawned until an alert is received, but the alert parameters are posted by the behavior, creating a bit of a chicken or the egg conundrum. The alert request is actually posted by an instance of the behavior created ever so briefly at helm startup. At startup, the helm creates instances of *all* behaviors, even templates, to ensure the configuration parameters are correct. The ultimate confirmation of behavior parameter correctness is obtained by a behavior instance itself confirming each parameter. The helm will then immediately, before its first iteration, delete any behaviors temporarily created from template behaviors. During this brief startup period, the helm will invoke the function `onHelmStart()` for all behaviors. This function is defined at the IvPBehavior superclass level just like `onRunState()`. In the case of the collision avoidance behavior, this function is implemented to make the automatic alert configuration posting to `BCM_ALERT_REQUEST`.

39.5 Specifying the Priority Policy - the pwt_*_dist Parameters

The AvdColregs behavior may be configured to increase its priority as its range to the contact diminishes. The priority weight specified in its configuration represents the *maximum* possible priority applied to the behavior, presumably in close range to the contact. The range at which this maximum priority applies is specified in the pwt_inner_dist parameter. Likewise, the pwt_outer_dist parameter specifies a range to the contact where the priority weight becomes zero, regardless of the priority weight specified in the configuration file. This relationship is shown in Figure 108.

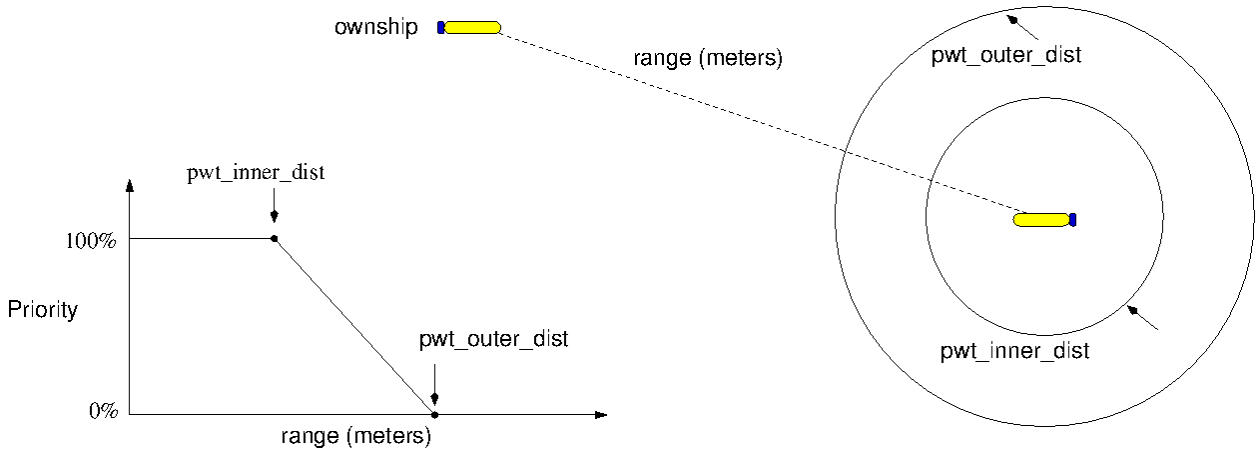

Figure 108: **Parameters for the BHV_AvdColregs behavior:** The *ownship* vehicle is the platform running the helm. The range between the two vehicles affects whether the behavior is active and with what priority weight. Beyond the range specified by pwt_outer_dist, the behavior is not be active. Within the range of pwt_inner_dist, the behavior is active with 100% of its configured priority weight.

By default, the priority weight decreases linearly between the two depicted ranges. The pwt_grade parameter allows the degradation from maximum priority to zero priority to fall more steeply by setting pwt_grade=quadratric.

39.6 Associating Utility to CPA of Candidate Maneuvers

- min_util_cpa_dist: The distance (in meters) between ownship and the contact at the closest point of approach (CPA) for a candidate maneuver, below which the behavior treats the distance as it would an actual collision between the two vehicles.
- max_util_cpa_dist: The distance (in meters) between ownship and the contact at the closest point of approach (CPA) for a candidate maneuver, above which the behavior treats the distance as having the maximum utility.

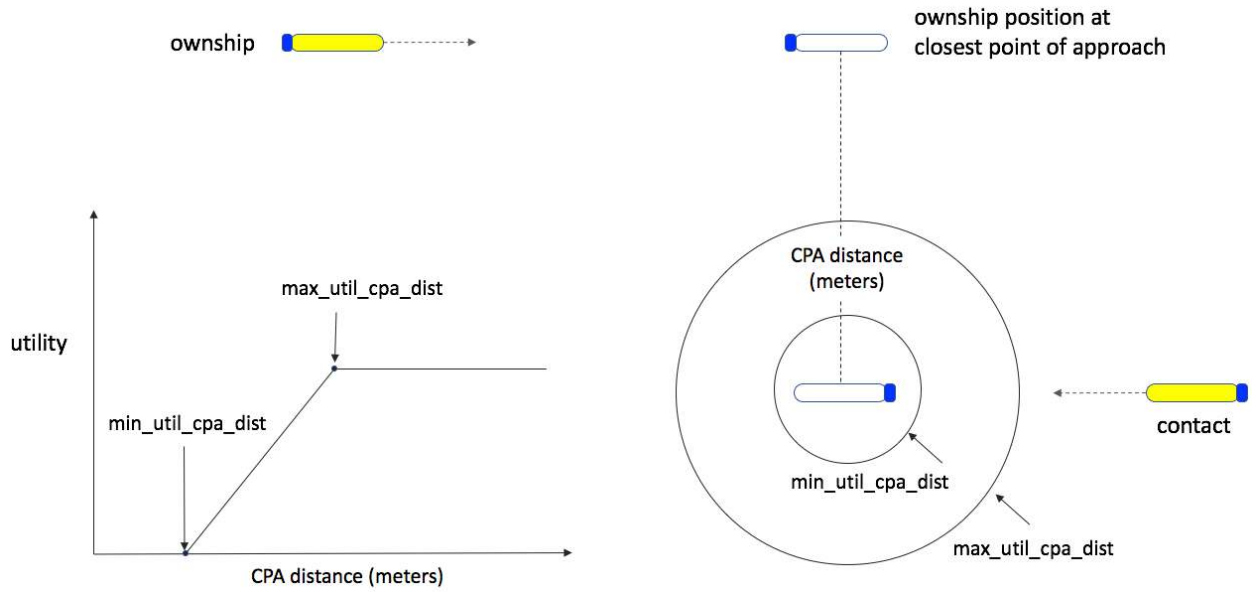

Figure 109: Parameters for the BHV_AvdColregs behavior. The *ownship* vehicle is the platform running the helm. The `collision_distance` is used when applying a utility metric to a calculated closest point of approach (CPA) for a candidate maneuver. A CPA less than or equal to the `collision_distance` is treated as an actual collision with the lowest utility rating.

40 The CutRange Behavior

The CutRange behavior will drive ownship to reduce the range between itself and another specified vehicle (nearly the opposite of the BHV_AvoidCollision behavior).

40.1 The Patience Parameter

The behavior reasons about candidate ownship maneuvers and will use a combination of immediate-rate-of-closure (IROC) and expected closest point of approach (CPA) distance, to evaluate candidate maneuvers. Using solely IROC is regarded as having zero patience, and using solely CPA is regarded as having maximal patience. The idea is conveyed in Figure 110 below. The former method may be preferred for a contact with frequent heading changes, while the latter may be preferred for contacts on a trajectory that rarely changes.

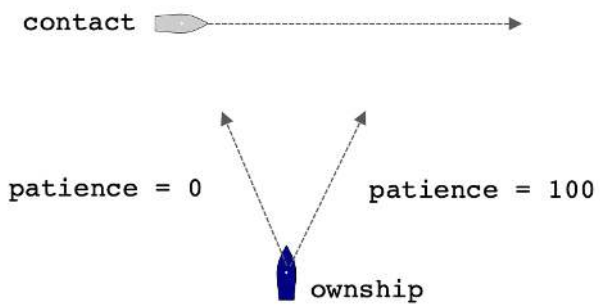

Figure 110: The *CutRange* behavior will either drive ownship directly toward the current contact position (zero patience setting) or drive ownship to be on the fastest intercept course based on the contact current position, heading and speed (maximal patience setting).

40.2 Automatic Priority Weight Variation Based on Range

The CutRange behavior may optionally be configured to automatically decrease its priority weight down to zero as it closes range to the contact. This can be used to ensure ownship does not collide with the contact or to transition the helm to a different mode of operation as it closes range to the contact. The idea is shown in Figure 111

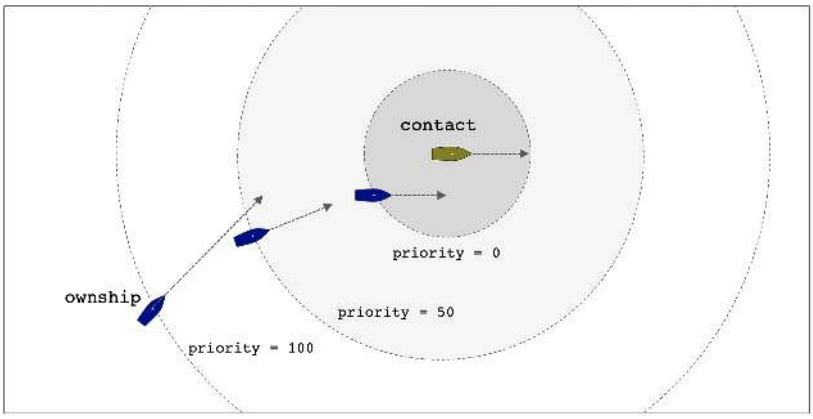

Figure 111: When ownship is at the outer range (pwt_outer_range), the priority is at its maximal value. When ownship is at the inner most range (pwt_inner_range), the priority weight drops to zero. The priority weight decreases linearly as the range is closed. If the overall configured priority weight for the behavior were set to 80 instead of 100, the outer weight would 80, and 40 at the intermediate range.

When the contact is at a range at or beyond pwt_outer_dist, the priority weight will be 100 percent of the behavior's configured priorty weight. When the contact is at or closer than the range given by pwt_inner_dist the priorty weight will be zero. At contact ranges between these two values the priority weight will linearly decrease in value. In the situation depicted in Figure 111, the vehicle transistion to a dominant Shadow behavior as the range is closed. When the range decreases to pwt_inner_dist, the Shadow behavior is solely determining ownship heading and speed to match the contact's current heading and speed.

40.3 Initiating and Abandoning Pursuit

The giveup_dist parameter sets a contact range beyond which the behavior will assign, a zero priority weight, essentially giving up pursuit and becoming an inactive behavior. The parameter is typically set to be a bit higher than the pwt_outer_dist parameter, but any non-zero configured value will be respected. A giveup_dist of zero is the default, and disables this feature.

As ownship range to the contact increases and crosses the giveup_dist threshold, the behavior may be configure to post one or more messages, using the giveupflag parameter. Similarly, when ownship range to contact decreases below the giveup range, the behavior may be configure to post one or more messages, using the pursueflag parameter. For example:

```
pursueflag = PURSUIT=true
giveupflag = PURSUIT=false
```

In the above case, the two flags could be used to transition the helm out of its present mission mode and back again based on the status of closing range to the contact.

40.4 Configuration Parameters

Listing 40.28: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 40.29: Configuration Parameters Common to Contact Behaviors.

Parameter	Description
bearing_lines	If true, a visual artifict will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.

contact	Name or unique identifier of a contact to be avoided.
decay	Time interval during which extrapolated position slows to a halt.
exit_on_filter_group	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_vtype	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_region	It true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default it false.
extrapolate	If true, contact position is extrapolated from last position and trajectory.
ignore_group:	If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 15.2
ignore_name:	If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 15.2
ignore_region:	If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 15.2
ignore_type:	If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 15.2
match_group:	If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 15.2
match_name:	If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 15.2
match_region:	If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regins. Introduced after Release 19.8.1. Section 15.2
match_type:	If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 15.2
on_no_contact_ok	If false, a helm error is posted if no contact information exists. Applicable in the more rare case that a contact behavior is statically configured for a named contact. The default is true.

strict_ignore	If true, and if one of the ignore exclusion filter components is enabled, then an exclusion filter will fail if the contact report is missing information related to the filter. For example if the contact group information is unknown. The default is true.
time_on_leg	The time on leg, in seconds, used for calculating closest point of approach.

Listing 40.30: Configuration Parameters for the CutRange Behavior.

Parameter	Description
giveup_dist:	Range to contact, in meters, outside which the behavior will give up (become inactive). Default is zero. Section 40.3.
giveupflag:	Optional MOOS variable-value pairs posted when ownship range to contact increases above giveup_dist. Section 40.3.
patience:	Linear scale choice between preferring heading directly to the contact (patience=0) or heading on the lowest closest point of approach (patience=100). Section 40.1.
pursueflag:	Optional MOOS variable-value pairs posted when ownship range to contact drops below giveup_dist. Section 40.3.
pwt_outer_dist:	Range to contact outside which the behavior has maximum priority. Section 40.2.
pwt_inner_dist:	Range to contact within which the behavior has zero priority. Section 40.2.

Listing 40.31: Example Configuration Block.

```
Behavior = BHV_CutRange
{
  // General Behavior Parameters
  // ---------------------------
  name         = cutrange_                          // example
  pwt          = 200                                // example
  condition    = AVOID = true                       // example
  updates      = CONTACT_INFO                       // example
  endflag      = CONTACT_RESOLVED = $[CONTACT]      // example
  templating   = spawn                              // example


  // General Contact Behavior Parameters
  // -----------------------------------
      bearing_lines = white:0, green:0.65, yellow:0.8, red:1.0   // example

            contact = henry            // example
              decay = 15,30            // default (seconds)
        extrapolate = true             // default
   on_no_contact_ok = true             // default
        time_on_leg = 60               // default (seconds)


  // Parameters specific to this behavior
  // ------------------------------------
        giveup_dist = 225              // Meters. Default is 0, disabled
           patience = 45               // [0,100]. Default 0
     pwt_inner_dist = 50               // Meters. Default is 0.
     pwt_outer_dist = 200              // Meters. Default is 0.
         pursueflag = PURSUIT=true
         giveupflag = PURSUIT=false
}
```

40.5 Variables Published

The below MOOS variables 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 described in Section 7.2.8.

- `VIEW_SEGLIST`: A bearing line between ownship and the contact if configured for rendering.
- Any variables involved in the `giveupflag` or `pursueflag` configurations, or other general behavior flags.

41 The Shadow Behavior

This behavior will drive the vehicle to match the trajectory of another specified vehicle. This behavior in conjunction with the BHV_CutRange behavior can produce a "track and trail" capability.

41.1 Configuration Parameters

Listing 41.32: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].
name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 41.33: Configuration Parameters Common to Contact Behaviors.

Parameter	Description
bearing_lines	If true, a visual artifict will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.
contact	Name or unique identifier of a contact to be avoided.
decay	Time interval during which extrapolated position slows to a halt.
exit_on_filter_group	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_vtype	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_region	It true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default it false.
extrapolate	If true, contact position is extrapolated from last position and trajectory.
ignore_group:	If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 15.2
ignore_name:	If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 15.2
ignore_region:	If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 15.2
ignore_type:	If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 15.2
match_group:	If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 15.2
match_name:	If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 15.2
match_region:	If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regins. Introduced after Release 19.8.1. Section 15.2
match_type:	If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 15.2

on_no_contact_ok	If false, a helm error is posted if no contact information exists. Applicable in the more rare case that a contact behavior is statically configured for a named contact. The default is true.
strict_ignore	If true, and if one of the ignore exclusion filter components is enabled, then an exclusion filter will fail if the contact report is missing information related to the filter. For example if the contact group information is unknown. The default is true.
time_on_leg	The time on leg, in seconds, used for calculating closest point of approach.

Listing 41.34: Configuration Parameters for the Shadow Behavior.

Parameter	Description
heading_peakwidth:	Width of the peak, in degrees, in the produced ZAIC-style IvP function.
heading_basewidth:	Width of the base, in degrees, in the produced ZAIC-style IvP function.
speed_peakwidth:	Width of the peak, in m/sec, in the produced ZAIC-style function.
speed_basewidth:	Width of the base, in m/sec, in the produced ZAIC-style IvP function.

Listing 41.35: Example Configuration Block.

```
Behavior = BHV_CutRange
{
  // General Behavior Parameters
  // ---------------------------
  name         = shadow_henry          // example
  pwt          = 200                   // example
  condition    = SHADOW = true         // example
  updates      = CONTACT_INFO          // example


  // General Contact Behavior Parameters
  // -----------------------------------
           contact = henry             // example
             decay = 15,30             // default (seconds)
       extrapolate = true              // default
   on_no_contact_ok = true             // default
       time_on_leg = 60                // default (seconds)


  // Parameters specific to this behavior
  // ------------------------------------
       pwt_outer_dist = 0              // default (meters)
    heading_peakwidth = 20             // default
    heading_basewidth = 160            // default
      speed_peakwidth = 0.1            // default
      speed_basewidth = 2.0            // default
}
```

- pwt_outer_dist: The distance (in meters) that the contact must be within for the behavior to be active and produce an objective function. The default is zero meaning it will be active regardless of the distance to the contact.
- heading_peakwidth: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the peakwidth parameter of the heading component.
- heading_basewidth: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the basewidth parameter of the heading component.
- speed_peakwidth: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the peakwidth parameter of the speed component.
- speed_basewidth: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the basewidth parameter of the speed component.

41.2 Variables Published

The below MOOS variables 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 described in Section 7.2.8.

Listing 41.36: Variables Published by the Shadow Behavior.

Variable	Description
SHADOW_CONTACT_X:	description
SHADOW_CONTACT_Y:	description
SHADOW_CONTACT_SPEED:	description
SHADOW_CONTACT_HEADING:	description
SHADOW_CONTACT_RELEVANCE:	description

42 The Trail Behavior

This behavior will drive the vehicle to trail or follow another specified vehicle at a given relative position. A tool for ”formation flying”. The following parameters are defined for this behavior:

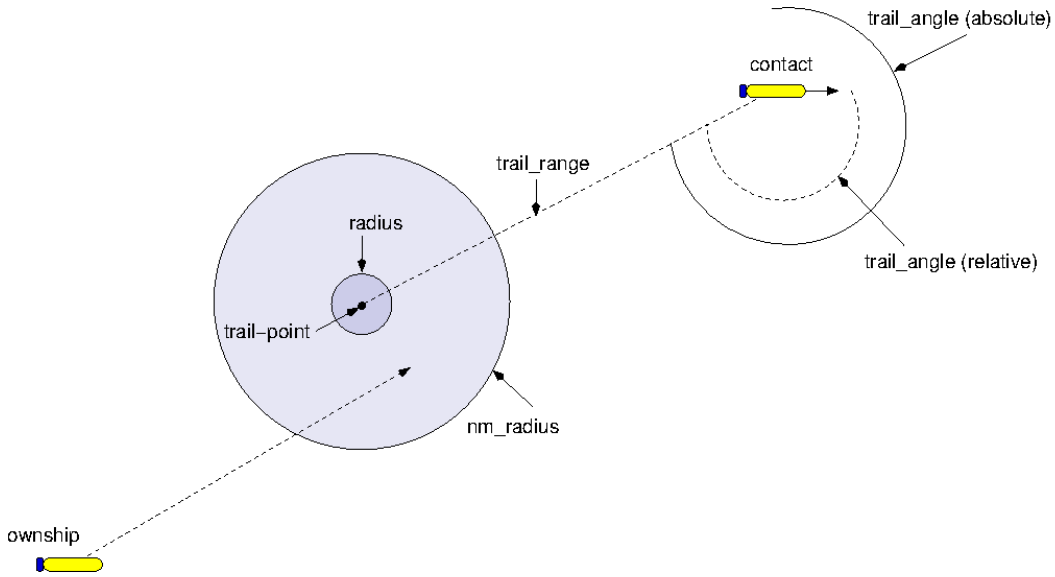

Figure 112: Interpolation of vehicle speed inside the radius set by nm_radius relative to the extrapolated trail position.

.

42.1 Configuration Parameters

Listing 42.37: Configuration Parameters Common to All Behaviors.

activeflag:	A MOOS variable-value pair posted when the behavior is in the *active* state. [more].
condition:	Specifies a condition that must be met for the behavior to be running. [more].
duration:	Time in behavior will remain running before declaring completion. [more].
duration_idle_decay:	When true, duration clock is running even when in the *idle* state. [more].
duration_reset:	A variable-pair such as MY_RESET=true, that will trigger a duration reset. [more].
duration_status:	The name of a MOOS variable to which the vehicle duration status is published. [more].
endflag:	A MOOS variable-value pair posted when the behavior has completed. [more].
idleflag:	A MOOS variable-value pair posted when the behavior is in the *idle* state. [more].
inactiveflag:	A MOOS variable-value posted when the behavior is *not* in the *active* state. [more].

name:	The (unique) name of the behavior. [more].
nostarve:	Allows a behavior to assert a maximum staleness for a MOOS variable. [more].
perpetual:	If true allows the behavior to to run even after it has completed. [more].
post_mapping:	Re-direct behavior output normally to one MOOS variable to another instead. [more].
priority:	The priority weight of the behavior. [more].
pwt:	Same as priority.
runflag:	A MOOS variable and a value posted when a behavior has met its conditions. [more].
spawnflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
spawnxflag:	A MOOS variable and a value posted when a behavior is spawned. [more].
templating:	Turns a behavior into a template for spawning behaviors dynamically. [more].
updates:	A MOOS variable from which behavior parameter updates are read dynamically. [more].

Listing 42.38: Configuration Parameters Common to Contact Behaviors.

Parameter	Description
bearing_lines	If true, a visual artifict will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.
contact	Name or unique identifier of a contact to be avoided.
decay	Time interval during which extrapolated position slows to a halt.
exit_on_filter_group	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_vtype	If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.
exit_on_filter_region	It true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default it false.
extrapolate	If true, contact position is extrapolated from last position and trajectory.
ignore_group:	If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 15.2
ignore_name:	If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 15.2

ignore_region:	If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 15.2
ignore_type:	If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 15.2
match_group:	If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 15.2
match_name:	If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 15.2
match_region:	If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regins. Introduced after Release 19.8.1. Section 15.2
match_type:	If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 15.2
on_no_contact_ok	If false, a helm error is posted if no contact information exists. Applicable in the more rare case that a contact behavior is statically configured for a named contact. The default is true.
strict_ignore	If true, and if one of the ignore exclusion filter components is enabled, then an exclusion filter will fail if the contact report is missing information related to the filter. For example if the contact group information is unknown. The default is true.
time_on_leg	The time on leg, in seconds, used for calculating closest point of approach.

Listing 42.39: Configuration Parameters for the Trail Behavior.

Parameter	Description
nm_radius:	If in this range to contact and ahead of it, slow down. The default is 20 meters.
no_alert_request:	If true, no request will be made to the contact manager for contact alerts. The default is true.
post_trail_dist_on_idle:	If true, post TRAIL_DISTANCE even during the idle state. The default is true.
pwt_outer_dist:	Range to contact outside which the behavior has zero priority.
radius:	If outside this radius to the contact, head to nm_radius ahead of trail point. The default is 5 meters.
trail_angle:	Relative angle to the contact to set the trail-point. The default is 180.
trail_angle_type:	Either ”relative” or ”absolute” bearing/angle to the contact. The default is relative.
trail_range:	Relative distance to the contact to set the trail-point. The default is 50 meters.

Listing 42.40: Example Configuration Block.

```
Behavior = BHV_Trail
{
  // General Behavior Parameters
  // --------------------------
  name         = trail_                // example
  pwt          = 100                   // default
  condition    = TRAIL_ALLOWED = true  // example
  updates      = TRAIL_INFO            // example
  templating   = spawn                 // example


  // General Contact Behavior Parameters
  // -----------------------------------
            contact = to-be-set        // example
              decay = 15,30            // default (seconds)
        extrapolate = true             // default
   on_no_contact_ok = true             // default
        time_on_leg = 60               // default (seconds)


  // Parameters specific to this behavior
  // ------------------------------------
               nm_radius = 20          // default (meters)
       no_alert_request = false        // default
  post_trail_dist_on_idle = true       // default
         pwt_outer_dist = 0            // default (meters)
                 radius = 5            // default (meters)
            trail_angle = 180          // default (degrees)
        trail_angle_type = relative    // default (or absolute)
             trail_range = 50          // default (meters)
}
```

42.2 Variables Published

The below MOOS variables 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 described in Section 7.2.8.

Listing 42.41: Variables Published by the Trail Behavior.

Variable	Description
PURSUIT:	1 if in behavior is active and producing an objective function, 0 otherwise.
REGION:	
TRAIL_HEADING:	
TRAIL_SPEED:	
TRAIL_RANGE:	
MAX_RANGE:	
VIEW_POINT:	
TRAIL_DISTANCE:	

42.3 Parameters

- `trail_range`: The range component of the relative position to the contact to trail.
- `trail_angle`: The relative angle of the relative position to the contact to trail. (180 is directly behind, 90 is a parallel track to the contacts starboard side, -90 is on the port side of the contact.)
- `trail_angle_type`: The trail angle may be set to either `relative` (the default), or `absolute`.
- `radius`: The distance (in meters) from the trail position that will result in the behavior "cutting range" to the trail position, and inside of which will result in the behavior "shadowing" the contact. The default is 5 meters.
- `nm_radius`: The distance in meters from the trail point within which the speed will be gradually change from the outer chase speed (max speed) and the speed of the contact, as illustrated in Figure 112. This parameter should typically be set to several times the value of `radius` to achieve smooth formation flying. Default is 20 meters.
- `max_range`: The distance (in meters) that the contact must be within for the behavior to be active and produce an objective function. The default is `max_range` value is zero meaning it will be active regardless of the distance to the contact.

43 The Convoy Behavior

This behavior will drive the vehicle to follow another specified vehicle by following virtual markers dropped by the lead vehicle. The vehicle following the lead vehicle is referred to as the *convoying* vehicle. As the lead vehicle transits, the convoying vehicle will make note of the lead vehicle's position and periodically note as *markers*, the position of the lead vehicle. The convoying vehicle will continuously transit to the rear-most marker. Upon reaching the rear-most marker it will proceed to the next rear-most marker, and so on.

This behavior can be compared to the Trail behavior. The goal of the Trail behavior is to continually aim toward a point at a relative angle and distance from the lead vehicle. When the Trail behavior is using a trail angle of 180 degrees, it may look similar to a Convoy behavior when both vehicles are on a long linear transit. However, as shown in Figure 113, when the lead vehicle turns, the Convoy behavior will continue to follow the trajectory of the lead vehicle, while the Trail behavior will deviate to steer directly toward the point currently 180 degrees behind the lead vehicle.

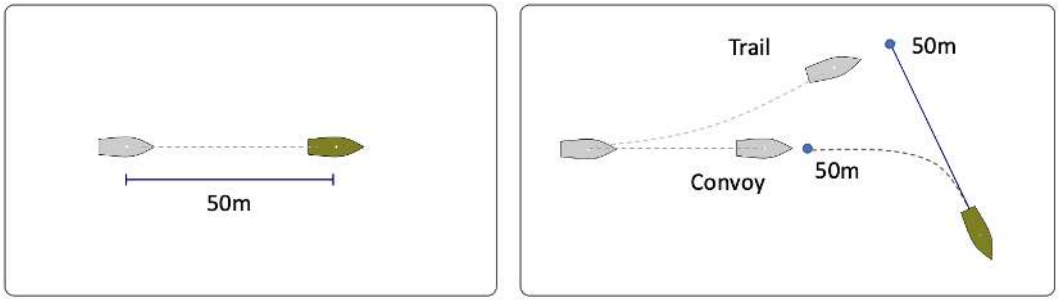

Figure 113: **Convoy vs. Trail** When traveling on a linear path, the Convoy behavior is similar to the Trail behavior with the trail angle set to 180 degrees, as shown on the left. But as the lead vehicle maneuvers, the Convoy behavior will follow the path of the lead vehicle, whereas the Trail behavior will continuously aim for the point currently 180 degrees behind the lead vehicle, as shown on the right.

.

The Convoy behavior may be preferred when it is important to have the following vehicle traverse the same path of the lead vehicle, regardless of their relative angle at any one given instant. For example, if the lead vehicle is driven by a human, or is autonomous but has a more capable sensor system, it may serve as a guide through obstacles, bridge pylons, or other hazards.

43.1 Generation and Removal of Markers

A *marker* is comprised of the observed X-Y or Lat/Lon position of the lead vehicle at some previous point in time. The lead vehicle typically does not communicate marker information explicitly to the convoying vehicle. Rather the convoying vehicle notes and stores this information from incoming node reports of the lead vehicle. Incoming node reports may be derived via AIS, direct communications, or through sensors on the convoying vehicle. The source of the lead vehicle position information is irrelevant to this behavior.

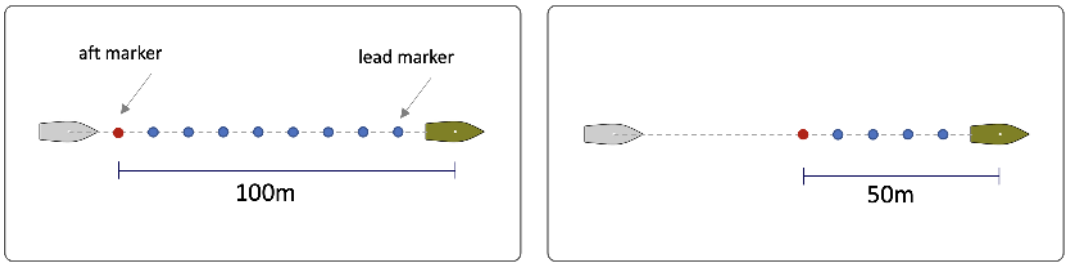

Figure 114: **Convoy Markers**: The convoying vehicle heads toward the rear-most marker behind the lead vehicle. As the convoying vehicle reaches each marker, the reached marker is dropped, and then proceeds to next rear-most marker as shown on the left. In a *lagging* situation, shown on the right, the convoying vehicle is still heading toward the rear-most marker, but the convoying vehicle's path, while it is lagging, may not closely follow the lead vehicle's path.

The *marker tail* is the set of all markers. The *marker tail length* is the sum of line segments between markers plus the range from the lead vehicle to the *lead marker*, which is the most recently added marker just aft of the lead vehicle. The *aft marker* is the oldest marker in the marker tail and presumably the marker that the convoy behavior is driving toward. The range between markers is set with the configuration parameter `inter_mark_range`. The default value is 10 meters.

A new lead marker is added as the lead vehicle moves, whenever the range between the lead vehicle and the current lead marker becomes greater than the `inter_mark_range`. A marker is removed from the rear of the tail if the tail length becomes longer than `tail_length_max`. The default of this parameter is 150 meters. By this policy, the tail length should always be less than or equal to `tail_length_max`. The rear-most marker may also be removed when or if the convoying vehicle captures the marker. Capturing is discussed next.

43.2 Capturing a Marker

As the convoying vehicle approaches its next marker, there are conditions continually examined to determine when it has reached, or *captured*, the marker. The simplest criteria is when the vehicle has reduced its range to the marker to be less than the `capture_radius`, a configuration parameter with default value of 5 meters. This is shown on the left in Figure 115. Occasionally the convoying vehicle may miss the marker, perhaps due to external environmental forces, or other mission considerations that pull the vehicle off a direct path to the marker. If the `capture_radius` were the only criteria for capturing a marker, the convoying vehicle may be forced to loop back to try again. If the `capture_radius` is too large, then the vehicle would be free, perhaps too early, to begin heading toward the next marker. If the lead vehicle is on a highly nonlinear path, a very large `capture_radius` may result in the convoying vehicle deviating too much from the lead vehicle's path.

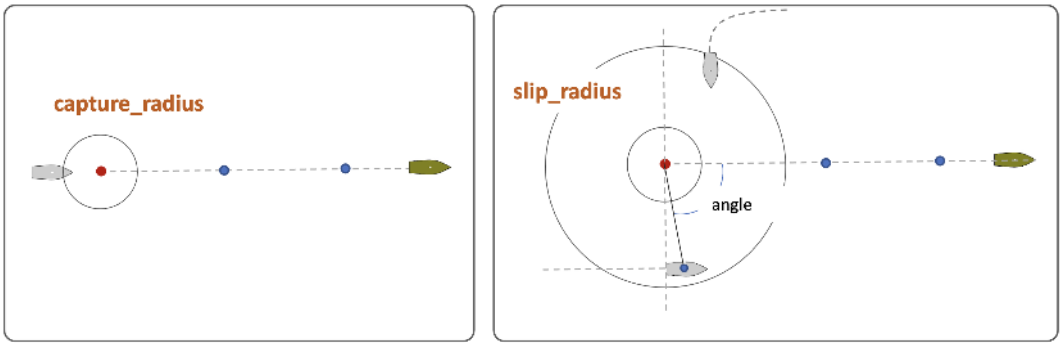

Figure 115: **Capturing a Marker**: The `capture_radius` determines when a convoying vehicle reaches a marker (left). If the convoying vehicle gets within the `slip_radius` range of the marker, the marker will be considered reached when or if the range to the marker begins to increase (right).

The `slip_radius` configuration parameter specifies a larger range to the marker than the `capture_radius`. Once the convoy vehicle enters within range of the `slip_radius`, the marker will be considered captured if the angle between ownship, the marker and next marker, is less than 90 degrees.

The default value for the `slip_radius` is 20 meters. The `slip_radius` range, as a rule of thumb, should be 2-4 times bigger than the `capture_radius`. A configuration warning will be generated if the `slip_radius` is less than the `capture_radius`.

When a marker is captured, the Convoying behavior removes the marker from its internal marker tail. Markers are removed from the tail when either they are captured by the convoying vehicle, or when the marker tail length exceeds the maximal length set by the `max_tail_length` parameter.

43.3 The Ideal Convoying Steady State

The ideal, or steady-state, situation for the convoying vehicle is to be positioned, on the marker tail, at the proper convoy range to the lead vehicle, and matching the speed of the lead vehicle. The *convoy range* is the length of the marker tail plus the distance between the rear-most marker and position of the convoying vehicle, as shown in Figure 116.

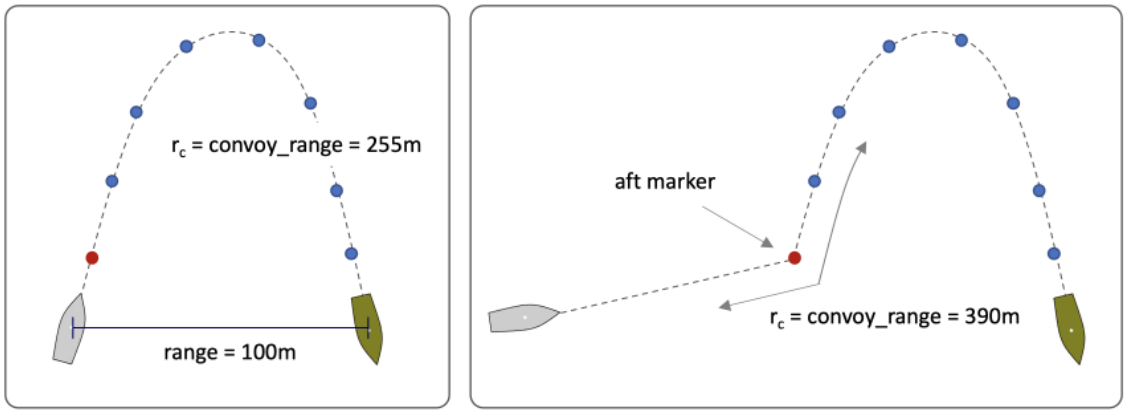

Figure 116: **Convoy Range**: The *convoy range* is the length of the marker tail plus the range from the rear-most marker to the current position of the convoying vehicle. This will be greater than or equal to the linear *range* between the two vehicles.

The *ideal convoy range* is synonymous with the value given by the parameter `tail_length_max`. The *convoy range delta* is the difference between the current convoy range and the ideal convoy range. The speed of the convoying vehicle in the ideal convoying steady state, is simply the current speed of the lead vehicle. The *convoy speed delta* is the difference between the current speed of the convoy vehicle and the lead vehicle.

It is also sometimes useful to have a notion of when the vehicle is in the ideal convoying steady state. As a general approximation we will say it is in the ideal state if:

- the convoy range delta is less than the inter mark range plus the slip radius, and
- the convoy speed delta is less than a quarter of the current lead vehicle speed, and less than a quarter of the convoy vehicle speed.

43.4 Speed of the Convoying Vehicle - The Speed Policy

The convoy behavior uses a *speed policy* to determine which speed best keeps the vehicle at the ideal convoy range to the lead vehicle. The policy is based on (a) the current convoy range, (b) the speed of the lead vehicle, and (c) the direct range between vehicles. The latter is only used as a measure for triggering a safety full-stop. Depending on the current convoy range delta, the speed policy will identify a *set speed*. The set speed will either match, overshoot, or undershoot the lead vehicle speed depending on whether the current convoy range is ideal, too far, or too close respectively.

When the current convoy range is not ideal, the degree of the overshoot or undershoot in the set speed is linearly proportional to the difference between the convoy range and ideal convoy range. There are several configuration parameters that determine the speed policy. These parameters are conveyed in Figure 117. Note that for the ideal convoy range there is a tolerance. Within this tolerance range, the set speed will simply be the lead vehicle speed, even if the current convoy range is not quite equal to the ideal convoy range.

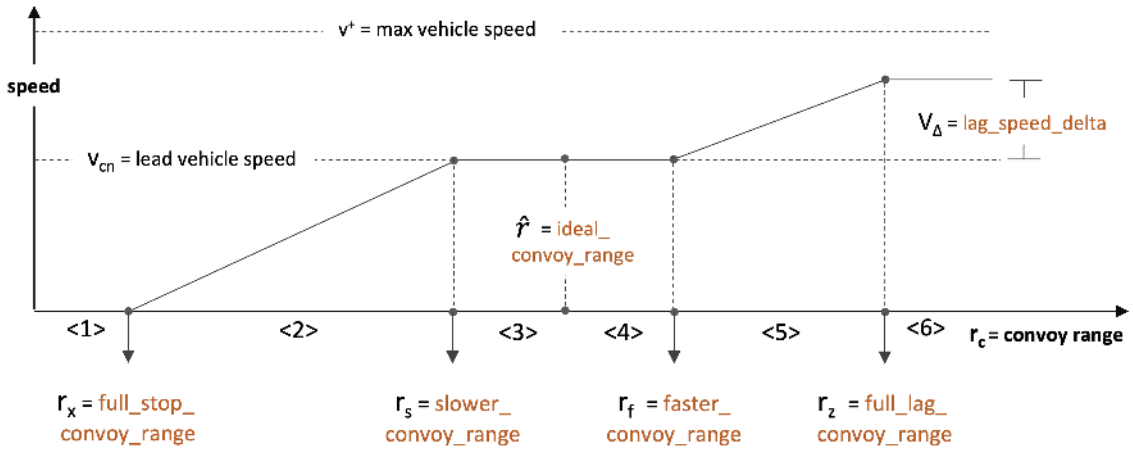

Figure 117: **The Convoy Speed Policy**: The preferred speed of the convoy behavior is determined by the speed policy, which depends on (a) the speed of the lead vehicle, (b) the current convoy range, and (c) the current direct range between vehicles. The speed policy is configured by the user and may be dynamically adjusted during a mission. There are six correction modes identified, based on the relationship between the current convoy range and the ideal convoy range.

The Correction Mode:

The relationship between the current convoy range and the ideal convoy range determines which of six *correction modes* apply at the moment. They are shown in Figure 117, and named:

1. `full stop`: The current convoy range is less than or equal to the `full_stop_convoy_range`.
2. `close`: The current convoy range is greater than the `full_stop_convoy_range`, but less than or equal to the `slower_convoy_range`.
3. `ideal close`: The current convoy range is greater than the `slower_convoy_range`, but less than or equal to the `ideal_convoy_range`.
4. `ideal far`: The current convoy range is greater than the `ideal_convoy_range`, but less than or equal to the `faster_convoy_range`.
5. `far`: The current convoy range is greater than the `faster_convoy_range`, but less than or equal to the `full_lag_convoy_range`.
6. `full lag`: The current convoy range is greater than the `full_lag_convoy_range`

43.4.1 Speed Policy Configuration Parameters

The speed policy is set directly with the following five parameters as an example. Note the value of the ideal convoy range does not directly influence the speed policy. Speed adjustments faster or slower are only made when the current convoy range is outside the steady speed span. The `ideal_convoy_range` parameter is used in setting the correction mode, and to have a metric to measure mission performance against. If left unspecified, it will be automatically set to the mid point between the `faster_convoy_range` and the `slower_convoy_range`

```
full_stop_convoy_range=20
slower_convoy_range=40
ideal_convoy_range=50
```

```
faster_convoy_range=60
full_lag_convoy_range=80
```

From Figure 117, the legal relative values are conveyed. For example, the `slower_convoy_range` cannot be less than the `full_stop_convoy_range`. The relative magnitude for each adjacent pair of parameters is checked both on helm startup, and after any dynamic update to the behavior during a mission. If a violation is detected upon startup, the helm will start in the `malconfig` state and will be unable to deploy until the behavior file is fixed and the helm re-launched. If the violation is detected at runtime, the behavior update will simply be rejected and a run warning will be posted.

43.4.2 Speed Policy Compression

The speed policy, comprised of the configuration parameters described above, may be dynamically adjusted mid-mission to compress or expand the convoy range between vehicles. This could be done using the `updates` variable, by providing new values for the speed policy parameters. For example, if the `updates` variable were `CONVOY_UPDATES`, then the following posting would adjust the speed policy:

```
CONVOY_UPDATES = full_lag_convoy_range=50 # faster_convoy_range=40 # slower_convoy_range=30
```

While the above is supported, a perhaps easier method is to specify a single *compression* value. This parameter ranges between 0 (no compression), and 1 (full compression). Since a fully compressed policy would essentially amount to a policy comprised of either full catch up speed or full stop, compression is capped at 0.9. For example, given the example speed policy above in Section 43.4.1, the full `CONVOY_UPDATES` line above could be accomplished with the simpler line:

```
CONVOY_UPDATES = compression=0.5
```

Compression is applied to all speed policy intervals except the safety-critical `full_stop_convoy_range`. The idea is conveyed in Figure 118 below.

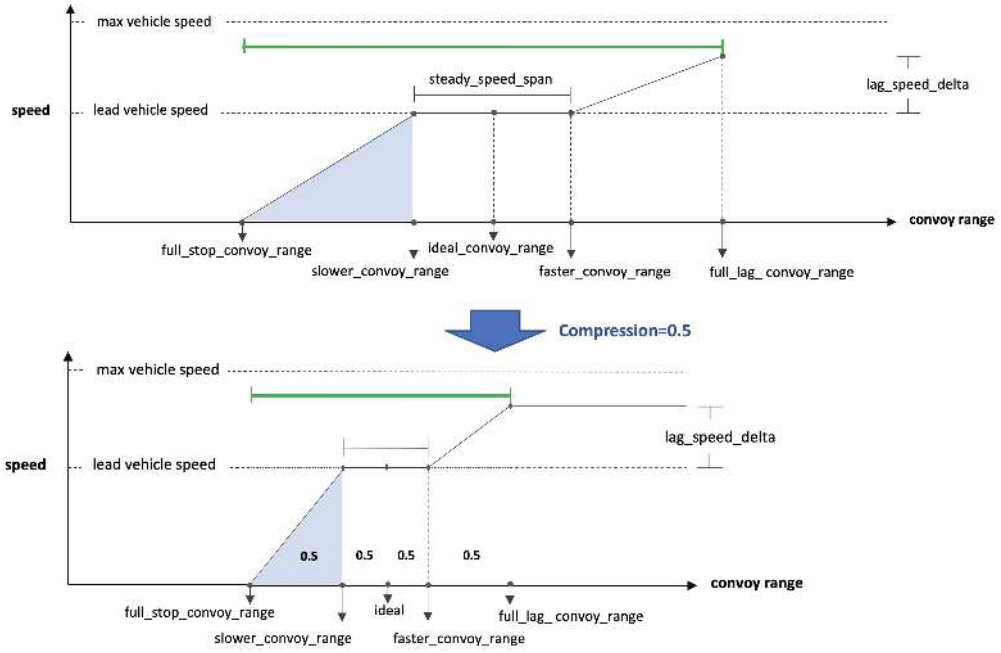

Figure 118: **Speed Policy Compression**: A single compression value in the range of [0,0.9] may be applied to a compression policy. All intervals will be compressed by this value, except the safety-critical `full_stop_convoy_range`.

Note that by setting `compression=N` either on startup, or through the `updates` variable, the compression is applied to the original uncompressed policy. In other words, two successive updates with a compression value of 0.5, will result in a policy compressed at 0.5, not 0.25.

43.4.3 Off-Peak Speed Preferences

The convoy behavior, like all behaviors of the IvP helm, will produce an objective function. The objective function of the convoy behavior is defined over possible course (desired heading) and speed choices. The convoy behavior constructs this 2D objective function by combining an objective function (utility function) defined over speed and one defined over heading. In this section the speed utility function is discussed.

The helm is configured with a decision space over all decision variables. This decision space is propagated to all behaviors of the helm, including the convoy behavior. The decision space for the speed component will contain both a min speed value (typically zero) and a max speed value. The max speed may be the max speed of the vessel, or the highest allowed operational speed, or may be set for some other reason. In any event, the speed utility function maps speed to utility, from the min speed to the max speed. As shown in Figure 119, the peak of this function is the set speed, produced from the speed policy. The set speed and speed utility function are dynamic, recalculated at each iteration of the helm based on situational input.

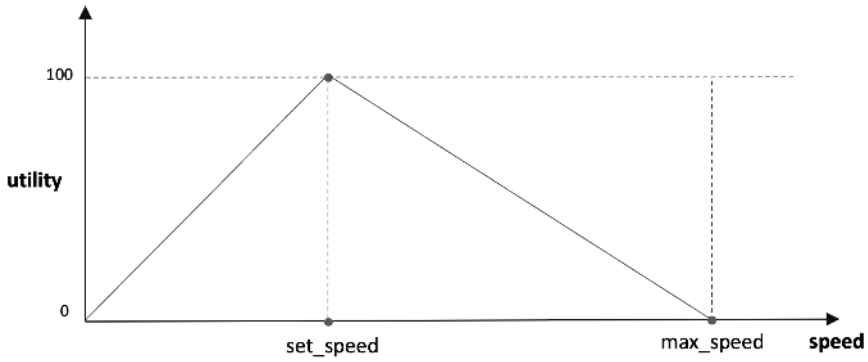

Figure 119: **Speed Utility Function**: Given a set speed, the speed utility function is marked by the optimal utility at the set speed with linearly decreasing utility for speed differing from the set speed.

Note in Figure 119, by default, the utility at both zero speed and the maximum speed are set to zero. However, the convoy behavior will vary either the zero speed or max speed utility to a non-zero value depending on the correction mode, described in Section 43.4. For example,

Figure 120 shows the speed utility function when the correction mode is `far`. Since the current convoy range is larger than the ideal convoy range, the correction mode is `far`, and the set speed already is somewhat higher than the lead vehicle speed, to close the convoy range. However, in this correction mode, off-peak utility of speeds higher than the set speed are skewed higher by setting the utility of the max speed to 50 percent of the max utility. In this way, if the convoy behavior needs to coordinate and compromise with another behavior, it is more likely to strike a compromise with a speed higher than the set speed.

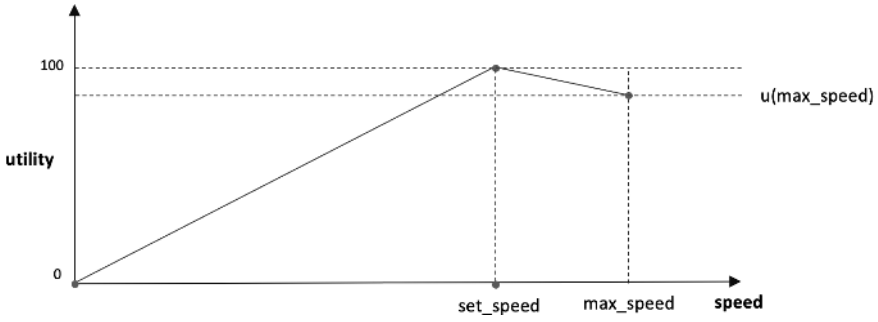

Figure 120: **Speed Utility Function When Lagging**: In a lagging situation, the set speed is chosen accordingly to close range on the lead vehicle. Additionally, the utility of higher speeds degrade linearly at a slower rate than the utility of speeds lower than the set speed. If the behavior is in a compromise situation with another behavior and the optimal set speed is precluded by the other behavior, compromise speeds are more likely to be higher than the set speed.

Likewise, if the convoy vehicle has a correction mode of `close`, Figure 121 shows the speed utility function. Since the current convoy range is smaller than the ideal convoy range, and the correction mode is `close`, the set speed already is somewhat lower than the lead vehicle speed, to open the convoy range. However, in this correction mode, off-peak utility of speeds lower than the set speed

are skewed lower by setting the utility of the zero speed to 50 percent of the max utility. In this way, if the convoy behavior needs to coordinate and compromise with another behavior, it is more likely to strike a compromise with a speed lower than the set speed.

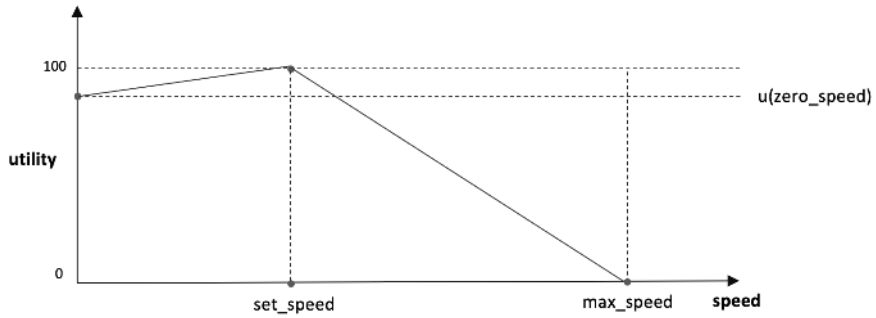

Figure 121: **Speed Utility Function When Close-In**: In a close-in or tailgating situation, the set speed is chosen accordingly to open range to the lead vehicle. Additionally, the utility of slower speeds degrade linearly at a slower rate than the utility of speeds higher than the set speed. If the behavior is in a compromise situation with another behavior and the optimal convoy set speed is precluded by the other behavior, compromise speeds are more likely to be slower than the set speed.

The speed utility function is skewed in this manner, based on the correction mode as described in the two examples above. For the other correction modes, the following policy is used:

Correction Mode	Zero-Speed Utility	Max-Speed Utility
close	50	0
ideal_close	25	0
ideal_far	0	25
far	0	50
full lag	0	75

43.5 Visual Preferences

The primary visual artifact produced by the convoy behavior is a posting to MOOS variable VIEW_POINT, for representing the markers in the marker tail. This variable is recognized by the pMarineViewer GUI application. An example posting is:

```
VIEW_POINT = x=-5.1,y=-50.31,label=ben_deb_14,vertex_color=red,vertex_size=6
```

The label is formed from the convoy vehicle name plus the lead vehicle name plus an index. As markers are captured and deleted by the behavior they are ”erased” by re-posting with the same label, setting the field active=false.

```
VIEW_POINT = x=-5.1,y=-50.31,active=false,label=ben_deb_14
```

An example of the artifacts rendered in pMarineViewer in a typical mission is shown in Figure 122.

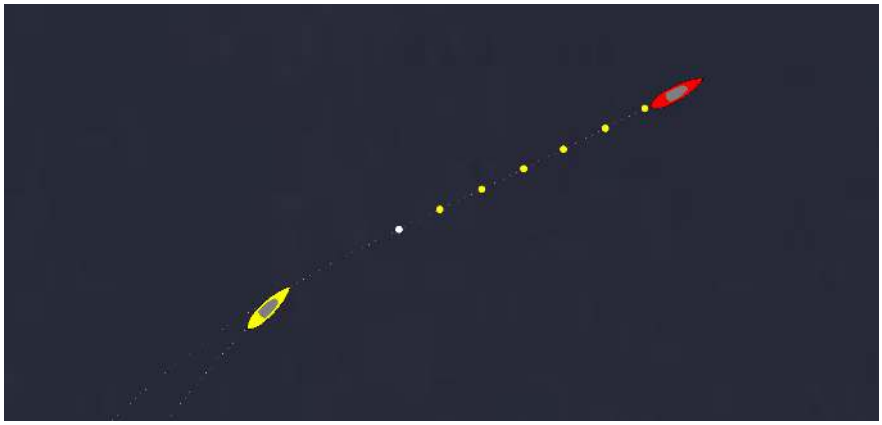

Figure 122: **Convoy Behavior Visual Artifacts**: The Convoy behavior will post points of a chosen size and color to represent the working marker tail. The aft marker is typically rendered in white. In this case the marker colors are set to automatically match the vehicle color for extra clarity.

The marker size and color, and the marker label color, may be set with configuration parameters:

```
visual_hint = marker_color = green
visual_hint = marker_label_color = white
visual_hint = marker_size = 6
```

The default marker color is `blue`. The default marker label color is `off`. The default marker point size is 4.

43.6 Output of the Convoy Behavior in the Form of Publications

All behaviors produce two kinds of output, (a) an objective function, and (b) one or more publications to MOOS variables. The former was discussed in Section **??**, the latter is discussed here. MOOS variable publications take two forms, (a) publications that are baked into the code that are not user configurable, and (b) user configurable publication based on event flags. The tendency in newer behaviors is to use fewer baked in publications and instead support a rich set of event flags that are further configurable with event macros. The specifics of publications for the Convoy behavior are discussed here.

43.6.1 Baked In Publications of the Convoy Behavior

Baked in publications are part of a behavior's implementation and the format of these publications are not configurable by the user. The Convoy behavior has three core publications:

- VIEW_POINT: Intended to be consumed by pMarineViewer during a mission, or alogview during mission playback. This variable was discussed in detail in Section 43.5.
- CONVOY_SPD_POLICY: A description of the prevailing speed policy.

- CONVOY_STAT_RECAP: A recap of convoy configuration state that typically does not change between iterations. See the example below.
- CONVOY_RECAP: A recap of convoy state that changes often and includes the core performance metrics and other state variables. See the example below.

An example of the static recap publication:

```
CONVOY_STAT_RECAP = follower=abe,leader=deb,ideal_rng=25,compression=0,index=0
```

An example of the speed policy publication:

```
CONVOY_SPD_POLICY = full_stop_rng=2,vname=eve,slower_rng=23,ideal_rng=25, \
                    faster_rng=27,full_lag_rng=40,lag_spd_delta=2,        \
                    max_compress=0.9
```

An example of the dynamic recap publication:

```
CONVOY_RECAP = convoy_rng=17.4,vname=gus,rng_delta=-7.5,tail_cnt=6,    \
               cmode=close,tail_rng=4.5,tail_ang=3.44,mark_bng=46.41,  \
               almnt=49.84,set_spd=0.661,cnv_avg2=0.905,cnv_avg5=0.867,\
               mx=30.6,my=-11.8,mid=0
```

The `mx`, `my`, and `mid` components are the aft marker X, Y location and marker ID respectively. The `cmode` compoent it the most recently observed correction mode.

Although the *format* of these variables cannot be changed in the mission file, they can be remapped to a different variable name, or they may be suppressed if desired, using the IvP Behavior post_mapping parameter:

```
post_mapping = CONVOY_RECAP,silent
post_mapping = CONVOY_RECAP,MY_PREFERRED_RECAP_VAR
```

By default, the CONVOY_RECAP will be posted whenever the behavior drops the aft marker, typically by reaching it. If the user prefers a more frequent posting, on every iteration of the behavior, the following parameter is used:

```
post_recap_verbose=true
```

The default setting for this parameter is false.

If the preference is to publish a smaller subset of the recap, or a simple numerical posting more readily consumable for plotting, use the event macros instead. For example, if one only wants the alignement value (the `almt` field in the recap), an event flag can be used:

```
convoy_flag = CONVOY_ALIGNMENT=$[ALIGNMENT]
```

which is published on every iteration of the convoy behavior when active. Or for less frequent publications, whenever a marker is reached:

```
marker_flag = CONVOY_ALIGNMENT=$[ALIGNMENT]
```

See Section 43.6.3 for further event flag and macro options.

43.6.2 Event Flag Publications of the Convoy Behavior

43.6.3 Event Flag Macros of the Convoy Behavior

Listing 43.42: Macros supported by the Convoy Behavior.

Macro	Description
AVG_SPD2:	The average speed of the lead vehicle over the last previous two seconds.
AVG_SPD5:	The average speed of the lead vehicle over the last previous five seconds.
COMP:	The current compresssion setting. See Section 43.4.2.
CONVOY_RNG:	The current convoy range. See Section 43.3 for the definition of the convoy range.
IDEAL_RNG:	The current ideal convoy range. See Section 43.3 for the definition of the ideal convoy range.
LEADER:	The name of the leader vehicle, in lower case.
CMODE:	The correction mode for the most recent set speed calculations. See Section 43.4. for the definition of the correction mode.
SET_SPD:	The current *set speed*. See Section 43.4 for the definition of the set speed.
RECAP:	A summary of the Convoy behavior settings and state.
TAIL_CNT:	The number of markers currently in the *marker tail*. See Section 43.1 for the definition of the marker tail.
TRK_ERR:	The current range to the marker tail augmented with recent history of retired markers. Meant to measure the deviation from the path of the lead vehicle. See Figure 124 for the definition of the track error.

An example of the RECAP macro expansion is:

$(RECAP) = `leader=ben, follower=abe, convoy_range=32.1, ideal_range=30, compression=0.25`

43.7 Performance Metrics

Performance metrics are values derived from the current ownship and contact positions and headings, plus the current marker tail and the parameter set by the user for the ideal contact range.

- *tail range*: The range between ownship and the aft marker.
- *tail angle*: The angle between the three points given by ownship position, the aft marker and the near-aft marker. By definition this value cannot be more than 180. This number is reported at 180-x so that minimal, near zero is optimal. See Figure 123.
- *marker bearing*: The absolute relative bearing of the aft marker to ownship. This number is in the range of $[0, 180]$. See Figure 123.

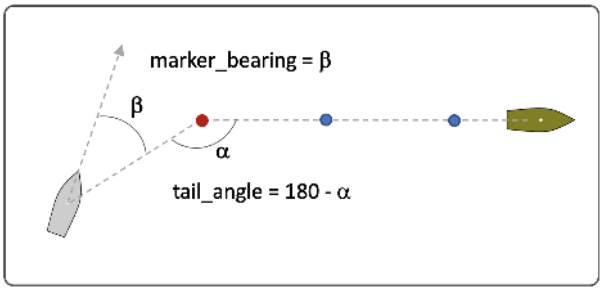

Figure 123: **Tail Angle and Marker Bearing**: The tail angle is angle at the aft marker between ownship and the near-aft marker. The marker bearing is the absolute relative bearing of the aft marker to ownship.

- *alignment*: Alignment is the sum of tail angle and `marker_bearing`.
- *convoy range delta*: The difference between the current convoy range and the ideal convoy range.
- *track error*: The distance to the marker tail with the marker tail augmented by a few recently retired markers. This is meant to convey how well ownship is following the lead vehicle. See Figure 124.

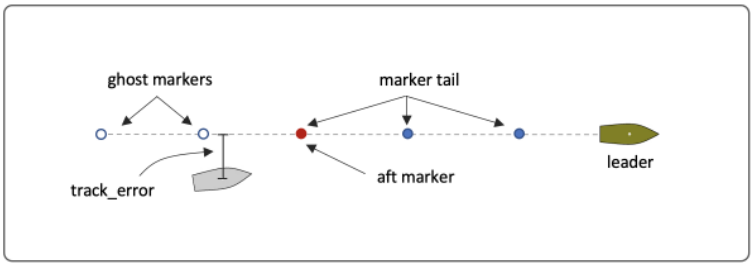

Figure 124: **Track Error**: The track error is the distance between ownship and the track line segments, formed by the marker tail and few recently retired markers, ghost markers.

The metrics are published in the `CONVOY_RECAP` posting as discussed in Section 43.6.1. They may also be published via event flags and macros as discussed in Section 43.6.3.

43.8 Configuration Parameters

Certain configuration parameters available to Convoy behavior are are unique to the Convoy behavior and others are inherited one or more base classes. Here all parameters are presented, with details on those unique to the Convoy behavior.

Parameters for the Convoy Behavior

Listing 43.43: Configuration Parameters for the Convoy Behavior.

Parameter	Description
capture_radius:	The range between the convoying vehicle and the next marker before the marker can be considered captured. Section 43.2.
compression:	The degree to which the ideal convoy range and the entire speed policy is compressed to achieve closer convoy ranges. This parameter ranges between `[0, 0.9]`. Section 43.4.2.
faster_convoy_range:	The convoy range above which the behavior will produce preferred speeds that are faster than the lead vehicle speed. Section 43.4.
full_stop_convoy_range:	Range between ownship and the lead vehicle, within which the preferred ownship speed is zero. Section 43.4.
full_lag_convoy_range:	The convoy range above which ownship will apply maximum speed in order to close the convoy range. Maximum speed is given by the lead vehicle speed plus the lag_speed_delta. Section 43.4.
inter_mark_range:	The range between neighboring markers. Section 43.1.
ideal_convoy_range:	The ideal or target convoy range to be achieved by the behavior. Section 43.4.
radius:	Same as capture_radius.
lag_speed_delta:	The difference between preferred ownship speed and the lead vehicle speed when the convoy range is at or higher than the full_lag_convoy_range. Section 43.4.
slip_radius:	The range between the convoying vehicle and the next marker, within which the marker will be considered captured when or if the vehicle begins to open range to the marker. Section 43.2.
slower_convoy_range:	The convoy range within which the behavior will produce preferred speeds that are slower than the lead vehicle speed. Section 43.4.
tail_length_max:	The maximum marker tail length, beyond which the behavior will drop the oldest marker. Section 43.1.
visual_hints:	One or more hints governing the preferred size and color of markers. Section 43.5.

Listing 43.44: Example Configuration Block.

```
Behavior = BHV_ConvoyV21
{
  // General Behavior Parameters
  name       = convoy_
  pwt        = 100
  condition  = MODE = convoying
  updates    = CONVOY_UPDATES
  templating = spawn

  // Contact Behavior Parameters
  contact    = unset_ok

  // Convoy Behavior Parameters
            radius = 3.0
       slip_radius = 15.0
  inter_mark_range = 3
   tail_length_max = 40

  full_stop_convoy_range = 2
     slower_convoy_range = 23
      ideal_convoy_range = 25
     faster_convoy_range = 27
   full_lag_convoy_range = 40
         lag_speed_delta = 2.0

   marker_flag = CONVOY_RECAP=$[RECAP]
   marker_flag = CONVOY_RANGE=$[CONVOY_RNG]
  visual_hints = marker_color=$(COLOR)
  visual_hints = marker_color_label=off
  visual_hints = marker_size=12
}
```

43.9 Terms and Definitions

- *aft marker*: The oldest marker, typically farthest to the lead vehicle of all the markers, and typically the marker currently being driven toward by the Convoy behavior. Section 43.1.
- *captured marker*: The marker reached by the following vehicle. Section 43.1.
- *convoy range*: The actual current distance along the marker tail from the lead vehicle to the following vehicle. Section 43.3.
- *compression*: Modification of the speed policy to adjust all convoy range parameters, except the full stop convoy range, to be smaller than the original speed policy values. Section 43.4.
- *convoy range delta*: The absolute difference between the ideal convoy range and the desired convoy range. Section 43.3.
- *convoy speed delta*: The absolute difference between the lead vehicle speed and the following vehicle speed. Section 43.3.
- *ideal convoy range*: The desired convoy range. Section 43.3.
- *lagging*: A following vehicle is lagging the lead vehicle when the convoy range is beyond the ideal convoy range, namely beyond the range set in the `faster_convoy_range`. Section 43.4.
- *lead marker*: The most recently created marker, typically closest to the lead vehicle of all the markers. Section 43.1.

- *marker*: A point in the X-Y plane representing a prior location of the lead vehicle, to be driven toward by the convoying or following vehicle. Section 43.1.
- *marker tail*: The set of markers used for determining the following vehicle's path to match the lead vehicle. Section 43.1.
- *marker tail length*: The distance between the lead vehicle and the the newest marker plus the total length of segments between markers. Section 43.1.
- *range*: The direct range between the lead and following vehicle.
- *set speed*: The speed chosen to modify or hold the convoy actual convoy range compared to the ideal convoy range. It is the output of the speed policy. Section 43.4.
- *speed policy*: The policy governing the selection of the set speed given the current convoy range, lead vehicle speed, and absolute direct range between vehicles. Section 43.4.
- *track error*: The distance between ownship and the marker tail, where the marker tail is augmented by a few recently retired ghost markers. See Figure 124.

43.10 Known Shortcomings - Work in Progress

- Automatic compression. Speed policy compresses as the lead vehicle slows, and decompresses as the lead vehicle speeds up.
- Name of the leader is a macro option
- Follow auto-informs heartbeat style to the leader

44 The FixedTurn Behavior

The *FixedTurn* behavior allows a vehicle to execute a turn with parameters for configuring the duration of the turn (in degrees), and the commanded changes in desired heading during the turn. The behavior is primarily motivated by supporting surface vehicle shake-out tests, but may be used in any mission where this pattern is desired. The behavior can be configured with varying number of turns, order of turns, and the number of degrees for each turn, and the manner in which the turn is requested.

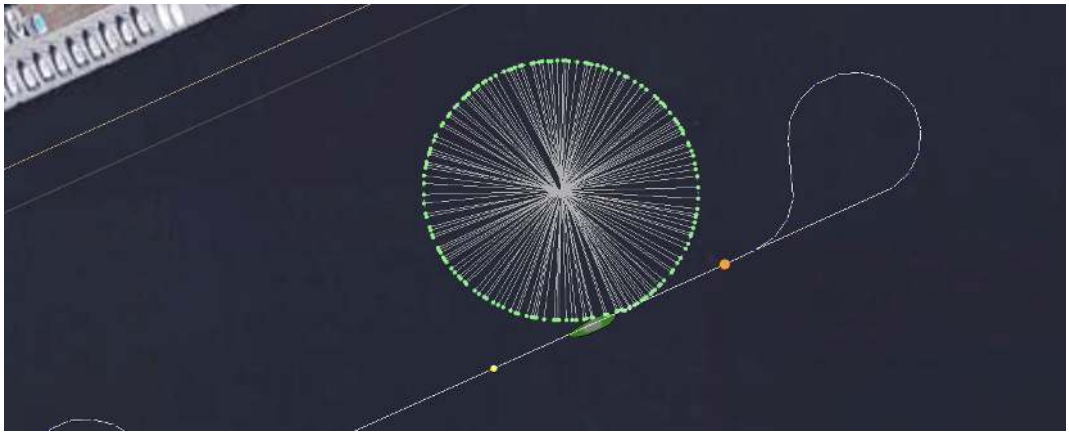

Figure 125: **FixedTurn Behavior**: A vehicle executes a FixedTurn behavior, with a single turn shown. The behavior starts and finishes within a pause of the LegRun behavior using the basic LegRun ability to initiate an event at given point in a leg. (From demo mission `s6_legrun`).

Turns in a FixedTurn behavior are accomplished by continuously requesting a heading change of a configured fixed value, monitoring the number of degrees in a turn, until the turn completion criteria is reached. Completion criteria is in number of degrees, e.g, a 180 degree turn, or a 270 degree turn.

44.1 Intended Mission Use

A FixedTurn behavior is configured with a sequence of parameterized turns. A turn configuration will prescribe the turn speed, the fixed delta heading, the total number of degrees for the turn, and the turn direction. For example, the following two turns only differ in direction.

```
turn_spec = spd=1.5, mhdg=10, fix=355, turn=star
turn_spec = spd=1.5, mhdg=10, fix=355, turn=port
```

Once the turn has been completed, an end flag is published, presumably resulting in the FixedTurn behavior becoming idle. The next time the FixedTurn behavior enters the running state, the next turn in the list of turns will be executed. Presumably, in the meanwhile, while in the idle state, other behaviors in the mission will maneuver the vehicle to a position where the nexted turn is safe for execution. In the intended use case for this behavior, there is typically no obstacle or collision

avoidance behaviors running during the turn. The mission author of course is free to do otherwise. However, the behavior was conceived to support the goal of conducting a controlled turn in a safe, hazard-free space, to gauge vehicle performance.

44.2 The Core Algorithm

The basic FixedTurn behavior will be configured with a sequence of turns, where each turn can be configured in terms of turn direction and size. The intended use for this behavior is for it to be run as a mission interuption of sorts, while the behavior is otherwise transiting. In the example in Figure 127, the turn is executing in the middle of the long straight leg of the vehicle running the LegRun behavior. This is from Example Mission N.

OnRunState() Flow Chart

The primary operation of the behavior is contained within the `OnRunState()` function. This is shown the flow chart and the pseudocode that follows. `Step` 1 of this sequence is to update the current value of ownship position and heading, a common first step in many behaviors. For this behavior, ownship heading is the key value for reasoning about turn completion.

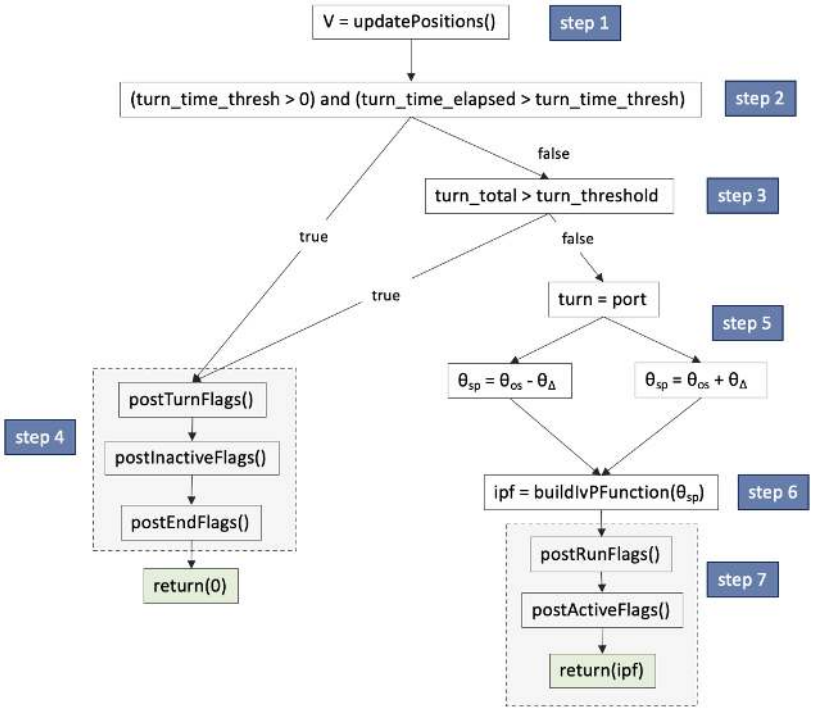

Figure 126: **FixedTurn Behavior Loop Flow Chart:** Each iteration of the FixedTurn behavior will proceed through the steps shown, with branching determined by sensed events and the behavior configuration. Details of each step are discuss in their own sections as indicated.

OnRunState() Pseudo Code

Algorithm 2: FixedTurn Behavior Main Loop

```
1:  procedure OnRunState()
2:      V ← updatePositions()                                   ▷ Step 1
3:      if timeout or turn_complete then                        ▷ Steps 2,3
4:          postTurnFlags()                                     ▷ Step 4
5:          postRunFlags()                                      ▷ Step 4
6:          postEndFlags()                                      ▷ Step 4
7:          return 0
8:      end if
9:      if turn = port then                                     ▷ Step 5
10:         θ_sp ← θ_sp − θ_Δ
11:     else
12:         θ_sp ← θ_sp + θ_Δ
13:     end if
14:     ipf ← buildIvPFunction(θ_sp)                            ▷ Step 6
15:     postRunFlags()                                          ▷ Step 7
16:     postActiveFlags()                                       ▷ Step 7
17:     return ipf
18: end procedure
```

In Step 2, the behavior checks to see if there is a timeout threshold specified for this turn. If there is, the timer is started as soon as the behavior transitions into the running state. The elapsed time since then is compared to the timeout threshold and, if it is exceeded, the behavior regards the turn as completed. In Step 3, the turn is checked for completion based on the turn degrees specified, and how much the vehicle has turned thus far. Although a full 360 degree turn is common, the specified turn may be any value, even beyond 360 degrees.

In Step 4, a behavior completion sequence is conducted. This includes the posting of end flags and inactive flags, if they are provided by the user in the behavior configuration. As well as turn flags, which are separate flags supported uniquely for this behavior, posted whenever a turn has been completed.

In Step 5, depending on whether the current turn is to the port or starboard, the desired heading set point θ_{sp}, is set to be offset from ownship current heading θ_{OS} by the number of degrees, θ_{Δ} prescribed in the configuration parameter mod_hdg. This heading set point is passed, in Step 6 to the function where the IvP objective function (ipf) is created. See Section **??**.

In Step 7, finally the run flags and active flags are posted this at this stage the behavior is in both the running (non-idle) and active (generates an IvP Function) state. The IvP function is returned to the helm to contribute to the selection of the heading and speed for the current helm iteration.

44.3 Configuration and Operation Mechanics

44.3.1 Configuring the Polygon Regions

The `FixedTurn` behavior is configured with one or more *turns*, constituting a *turn sequence*. A single turn is comprised of the following *turn elements*:

- *turn length*: number of degrees constituting a turn
- *heading offset*: requested change of heading during the turn
- *turn speed*: desired speed during the turn
- *turn direction*: port or starboard
- *turn timeout*: a max time, in seconds, allowed for the turn
- *turn flags*: one or more flags to be posted at turn completion.

```
turn_spec = spd=2.5, mhdg=20, fix=360, turn=star, timeout=60
turn_spec = spd=2.5, mhdg=20, fix=360, turn=port, timeout=60
turn_spec = spd=4.0, mhdg=10, fix=180, turn=star, flag=STAGE=complete
turn_spec = spd=4.0, mhdg=10, fix=180, turn=port, flag=STAGE=complete
```

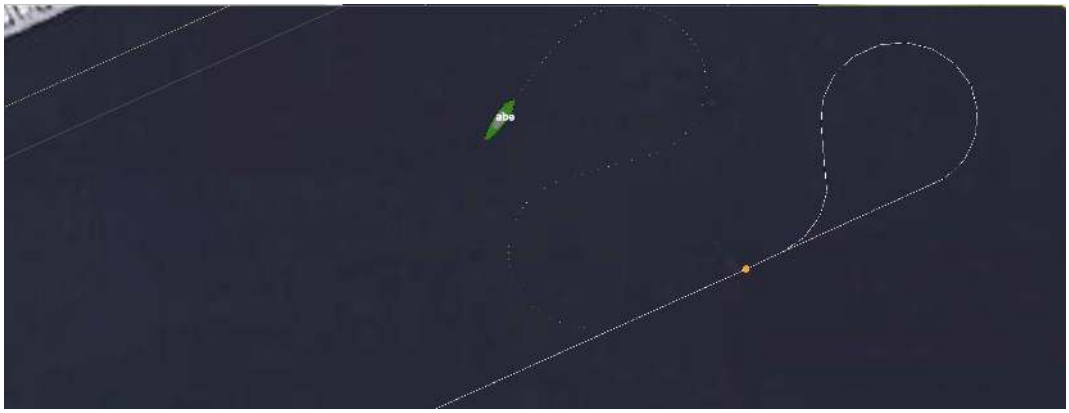

Figure 127: **Back-to-back Turns**: A vehicle executes a FixedTurn mission `s6_legrun`).

44.4 Configuration Parameters

Listing 44.45: Configuration Parameters for the FixedTurn Behavior.

Parameter	Description
`fix_turn`:	The default turn threshold for any turn that does not provide one. The default value is 0 degrees. Section **??**.
`mod_hdg`:	The default heading offset, θ_Δ for any turn that does not specify a heading offset. The default value is 20 degrees. Section **??**.

turn_dir:	The default turn direction for any turn that does specify a direction. The default direction is *port*. Section **??**.
speed:	The default speed for any turn that does not specify a turn speed. The default value is 1 meter per second. Section **??**.
timeout:	The default timeout duration, in seconds, for any turn that does not specify one. The default value is -1, indicating there is not timeout for the turn. Section **??**.
turn_delay:	feading. Section **??**.
turn_spec:	A full specification for a scheduled turn. Section **??**.
stale_nav_thresh:	The number of seconds, after which an update nav position and heading has not been received, befor the behavior throws a warning. Section **??**.
radius_rep_var:	A MOOS variable where the numerical value of the most recently measure turn radius is posted. By default this variable is unspecified and no publication is made. Section **??**.
schedule_repeat:	feading. The number of times the full list of turns in the turn schedule will be repeated. The default is zero. Section **??**.
visual_hints:	Color and other settings for posted visual artifacts. Section **??**.

Listing 44.46: Example Configuration Block.

```
Behavior = BHV_ZigZag
{
  name      = zig
  pwt       = 100
  condition = MODE==ZIGZAG
  endflag   = ZIGGING = false

  updates   = ZIG_UPDATE
  perpetual = true

            speed = 2.0   // meters per sec
   stem_on_active = true
        zig_first = star
     max_zig_zags = 2
        zig_angle = 45
 zig_angle_fierce = 30

     max_stem_odo = 100

     visual_hints = draw_set_hdg = true
     visual_hints = draw_req_hdg = true

       hdg_thresh = 2
   fierce_zigging = false
}
```

44.5 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags.

- VIEW_SEGLIST: A visual cue for rendering the requested zig angle, if either draw_set_hdg=true or draw_req_hdg=true in the visual_hints. See Section ??.

44.6 Flags and Macros

The ZigLeg behavior supports the below set of event flags in addition to the standard behavior flags, e.g., endflags, runflags. These are:

- zigflag: Posted each time a zig leg is completed. These flags are posted twice as frequent as zagflag postings.
- zagflag: Posted each time a zig zag is completed. These flags will be posted half as frequent as zigflag postings.
- starflag: Posted each time a zig leg started in the starboard direction.
- portflag: Posted each time a zig leg started in the port direction.
- starflagx: Posted each time a zig leg has *completed* in the starboard direction.
- portflagx: Posted each time a zig leg has *completed* in the port direction.

The folowing macros are supported in the ZigZag behavior. These macros will be expanded in any event flag, including event flags defined for all IvP behaviors as well as event flags defined only for the ZigZag behavior.

- $[ODO]: Total odometry distance since the behavior became active.
- $[ZIG_ODO]: Total odometry distance since the behavior began its current zig leg.
- $[ZAG_ODO]: Total odometry distance since the behavior began its current zig zag.
- $[STEM_DIST]: The current ownship distance perpendicular from the stem line. Section ??.
- $[STEM_ODO]: The current ownship distance parallel along the stem line. Section ??.
- $[STEM_HDG]: The stemline heading direction.
- $[ZIGS]: Total zig legs completed since behavior activation. Section ??.
- $[ZAGS]: Total zig zags completed since behavior activation. Section ??.
- $[ZIGS_EVER]: Total zig legs completed since this activation and all prior activations. For example if perpetual=true and repeated activations as in example mission s6_zigzag. Section ??.
- $[ZAGS_EVER]: Total zig zags completed since this activation and all prior activations. For example if perpetual=true and repeated activations as in example mission s6_zigzag. Section ??.
- $[ZIG_START_SPD]: Ownship speed at the start of the current zig leg.
- $[ZIG_MIN_SPD]: Minimum ownship speed recorded since the start of the current zig leg.
- $[ZIG_SPD_DELTA]: Change in ownship speed since the start of the current zig leg.
- $[ZIGS_TOGO]: Remaining zig legs to execute before completion, including the current zig leg being executed. Section ??.
- $[ZAGS_TOGO]: Remaining zig zags to complete before completion, including the current zig zag being executed. Section ??.

45 pMarineViewer: A GUI for Mission Monitoring and Control

45.1 Overviewx

The pMarineViewer application is a MOOS application written with FLTK and OpenGL for rendering vehicles and associated information and history during operation or simulation. A screen shot of a simple one-vehicle mission is shown below in Figure 128.

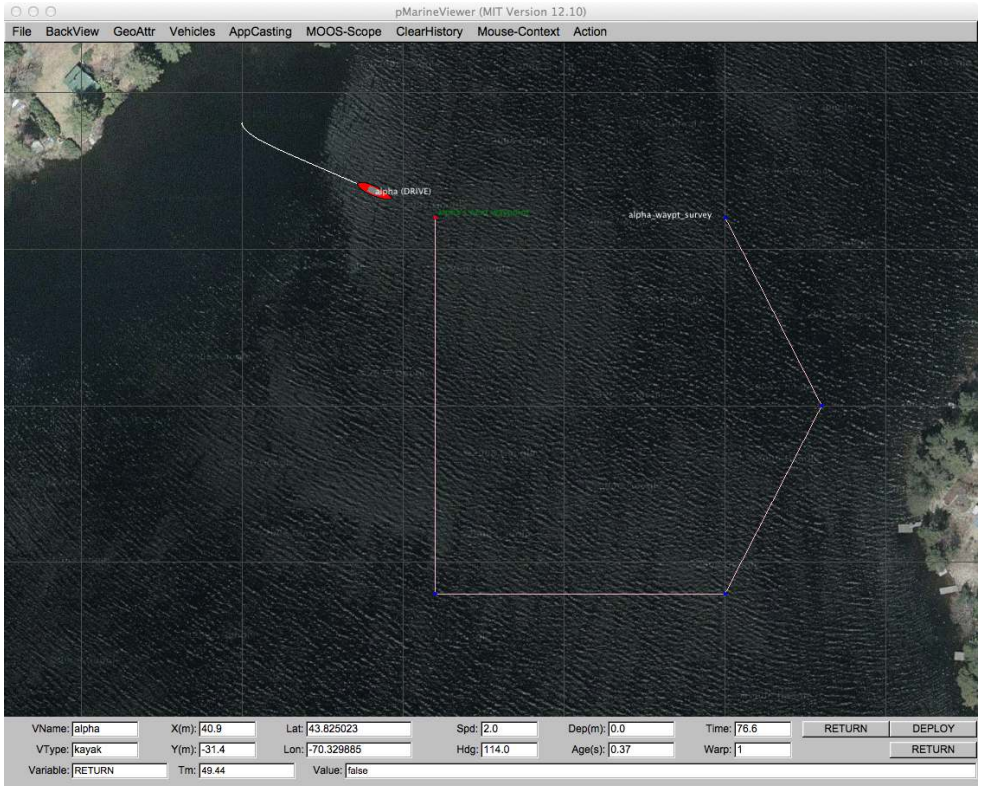

Figure 128: A **pMarineViewer** screen-shot executing a simple one-vehicle mission. The track of the vehicle is shown along with the set of waypoints it will traverse during this mission.

The user is able manipulate a geo display to see multiple vehicle tracks and monitor key information about individual vehicles. In the primary interface mode the user is a passive observer, only able to manipulate what it sees and not able to initiate communications to the vehicles. However there are hooks available and described later in this section to allow the interface to accept field control commands. With Release 12.11, appcasting viewing is supported to allow the pMarineViewer user to view appcasts across multiple fielded vehicles within a single optional window pane. In the release following 19.8, appcasting was complemented with realmcasting allowing the user to view publications and subscriptions for any app on multiple fielded vehicles. The user may also set up a watch cluster of variables to monitor over a set of vehicles in a single pane. The realmcasting utilities add quite a bit of power to developers of multi-vehicle, i.e., swarm, autonomy missions. This is described more fully in Section 46.10.

45.1.1 The Shoreside-Vehicle Topology

In some simple simulation single-vehicle arrangements pMarineViewer may co-exist in the same MOOS community as the helm and other components of a simulated vehicle. This is the case in the Alpha example mission. A more typical module topology, however, is that shown in Figure 129, where pMarineViewer is running in its own dedicated local MOOS community while simulated vehicles, or real vehicles on the water, transmit information in the form of a stream of *node reports* to the local community.

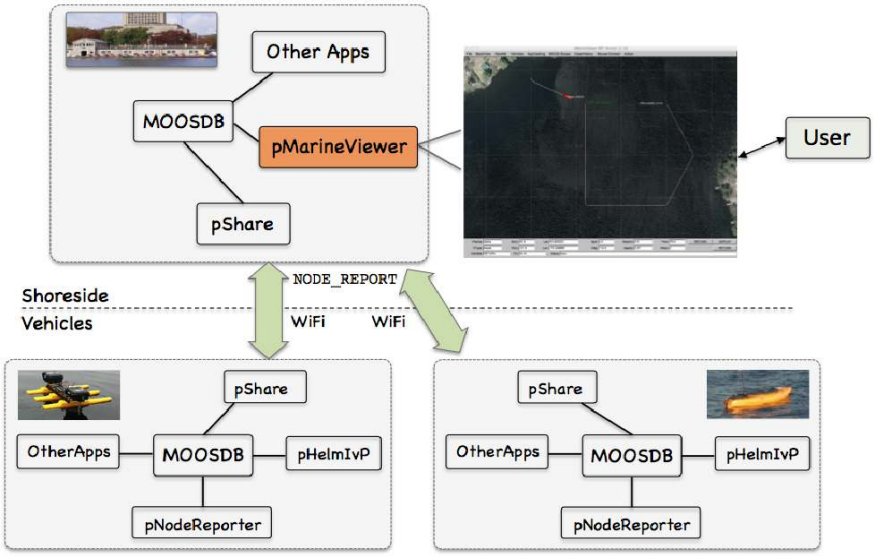

Figure 129: A common usage of the **pMarineViewer** is to have it running in a local MOOSDB community while receiving node reports on vehicle poise from other MOOS communities running on either real or simulated vehicles. The vehicles can also send messages with certain geometric information such as polygons and points that the view will accept and render.

A key variable subscribed to by pMarineViewer is the variable NODE_REPORT, which has the following structure given by an example:

```
NODE_REPORT = "NAME=henry,TYPE=uuv,TIME=1195844687.236,X=37.49,Y=-47.36,SPD=2.40,
              HDG=11.17,LAT=43.82507169,LON=-70.33005531,TYPE=KAYAK,MODE=DRIVE,
              ALLSTOP=clear,index=36,DEP=0,LENGTH=4"
```

Reports from different vehicles are sorted by their vehicle name and stored in histories locally in the pMarineViewer application. The NODE_REPORT is generated by the vehicles based on either sensor information, e.g., GPS or compass, or based on a local vehicle simulator.

In addition to node reports, pMarineViewer subscribes to several other types of information typically originating in the individual vehicle communities. This include several types of geometric shapes for which pMarineViewer has been written to handle. This includes points, polygons, lists of line segments, grids and so on. This is described further in Section 46.8.

In addition to consuming the above information, pMarineViewer may also be configured to post certain information, usually for command and control purposes. Since this is mission-specific, this

information is completely configured by the user to suit the mission. Posted information may also be tied to mouse clicks to allow, for example, a vehicle to be deployed to a point clicked by the users. This is described further in Section 45.2.

45.1.2 Description of the pMarineViewer GUI Interface

The viewable area of the GUI has three parts as shown in Figure 130 below. In the upper right, there is a geo display area where vehicles and perhaps other objects are rendered. The blue panes on the upper left displays appcast information. These panes hold appcast output from any appcast-enabled MOOS application running on any node, including the shoreside node. This is a new feature of Release 12.11 and may be toggled off and on with the `'a'` key, and may be configured to be either open or closed by setting the `appcast_viewable` parameter inside the `pMarineViewer` MOOS configuration block.

In the lower pane, certain data fields associated with the *active* vehicle are updated. Multiple vehicles may be rendered simultaneously, but only one vehicle, the *active* will be reflected in the data fields in the lower pane. Changing the designation of which vehicle is active can be accomplished by repeatedly hitting the 'v' key. The active vehicle is always rendered as red, while the non-active vehicles have a default color of yellow. Individual vehicle colors can be given different default values (even red, which could be confusing) by the user.

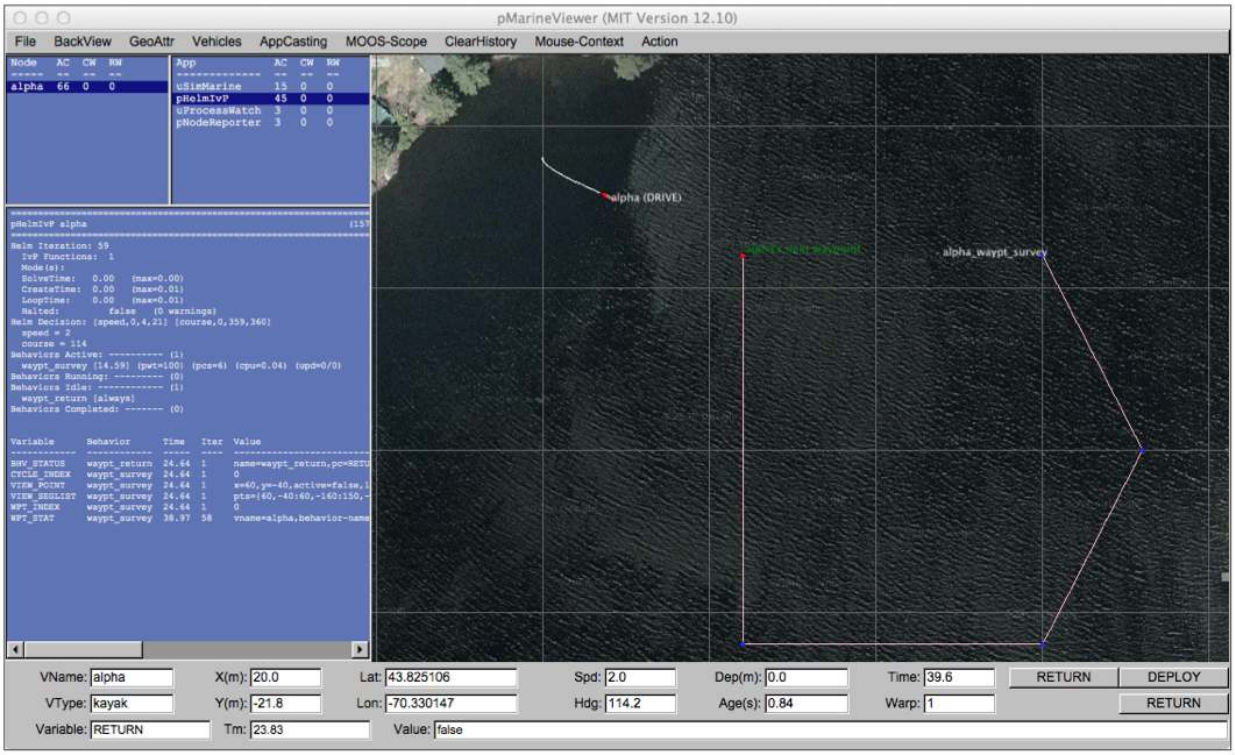

Figure 130: A screen shot of the `pMarineViewer` application running the alpha example mission. The position, heading, speed and other information related to the vehicle is reflected in the data fields at the bottom of the viewer.

Properties of the vehicle rendering such as the trail length, size, and color, and vehicle size and

color, and pan and zoom can be adjusted dynamically in the GUI. They can also be set in the pMarineViewer MOOS configuration block. Both methods of tuning the rendering parameters are described later in this section. The individual fields of the data section are described below:

- **VName:** The name of the active vehicle associated with the data in the other GUI data fields. The active vehicle is typically indicated also by changing to the color red on the geo display.
- **VType:** The platform type, e.g., auv, uuv, glider, kayak, ship, heron, wamv, buoy, mokai, longship, swimmer or unknown. The buoy shape is just a circle. The swimmer shape is a triangle.
- **X(m):** The x (horizontal) position of the active vehicle given in meters in the local coordinate system.
- **Y(m):** The y (vertical) position of the active vehicle given in meters in the local coordinate system.
- **Lat:** The latitude (vertical) position of the active vehicle given in decimal latitude coordinates.
- **Lon:** The longitude (horizontal) position of the active vehicle given in decimal longitude coordinates.
- **Spd:** The speed of the active vehicle given in meters per second.
- **Hdg:** The heading of the active vehicle given in degrees $(0 - 359.99)$.
- **Dep(m):** The depth of the active vehicle given in meters.
- **Age(s):** The elapsed time in seconds since the last received node report for the active vehicle.
- **Time:** Time in seconds since pMarineViewer was launched.
- **Warp:** The MOOS Time-Warp value. Simulations may run faster than real-time by this warp factor. MOOSTimeWarp is set as a global configuration parameter in the .moos file.

The age of the node report is likely to remain zero in simulation as shown in the figure, but when operating on the water, monitoring the node report age field can be the first indicator when a vehicle has failed or lost communications. Or it can act as an indicator of communications link quality.

The lower three fields of the window are used for scoping on a single MOOS variable. See Section 46.11 for information on how to configure the pMarineViewer to scope on any number of MOOS variables and select a single variable via an optional pull-down menu. The scope fields are:

- **Variable:** The variable name of the MOOS variable currently being scoped, or "n/a" if no scope variables are configured.
- **Time:** The variable name of the MOOS variable currently being scoped, or "n/a" if no scope variables are configured.
- **Value:** The actual current value for the presently scoped variable.

45.1.3 The AppCasting, FullScreen and Traditional Display Modes

As mentioned above, appcasting is new to release 12.11, pMarineViewer supports three display modes. The first mode is the *normal* mode familiar to pre-12.11 users of pMarineViewer as it was the only mode. A second mode, the *appcasting* mode, also shows the three appcasting panes shown above in Figure 130. The third mode is the *full-screen* mode which shows only the geo-display part to maximize viewing of the operation area. The modes may be toggled by single hot-key actions as shown in the figure.

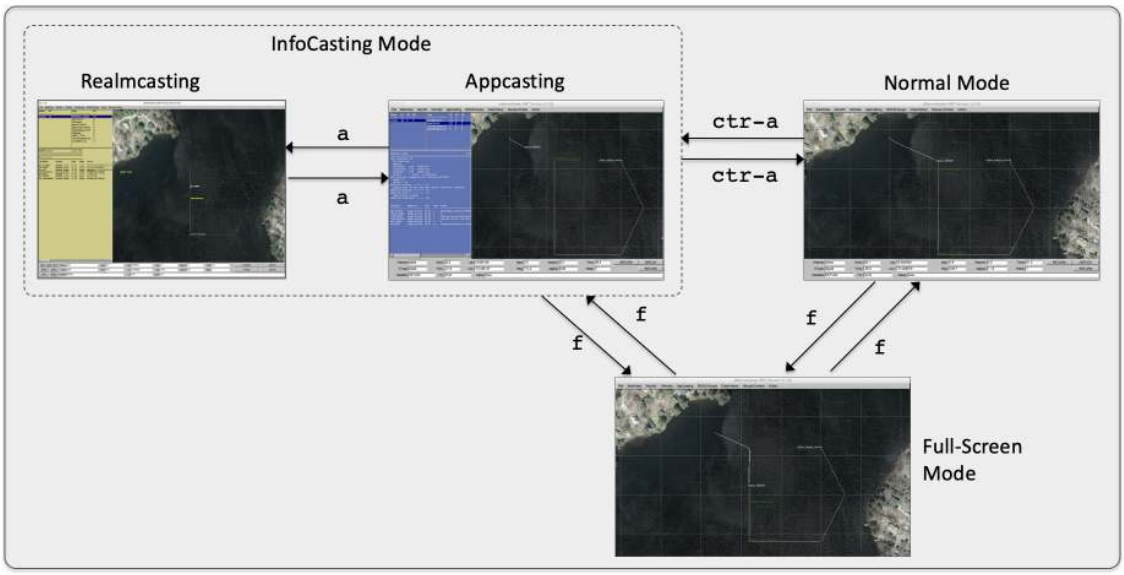

Figure 131: Three viewing modes are supported by `pMarineViewer`. The *normal* mode, the *infocasting* mode which renders appcast or realmcast output from any connected vehicle, or the *full-screen* mode to maximize viewing of the operation area and vehicles. The modes may be toggled with the hot-keys shown. When typing `'f'` in the full-screen mode, the viewer will return to the mode prior to entering the full-screen mode. The modes may also be changed via pull-down menu items, or set to personal preferences in the `.moos` configuration block. In software released after 19.8.x, the infocasting mode has two sub-modes, appcasting and realmcasting.

To launch a mission in the appcasting mode, set `appcast_viewable=true` in the `pMarineViewer` configuration block. To launch in the full-screen mode, set `full_screen=true` in the configuration block instead.

45.1.4 Run-Time and Mission Configuration

Nearly all `pMarineViewer` configuration parameters may be configured both at run-time, via pull-down menu selections, and prior to launch via configuration lines in the `pMarineViewer` configuration block of the `.moos` mission configuration file. To reduce the need to consult the documentation, the text of the pull-down menu selection is identical to the text of the parameter in the configuration file. Furthermore, most parameter selections are a choice from a fixed set of options. The present option for a parameter is typically indicated by a radio button in the pull-down menu.

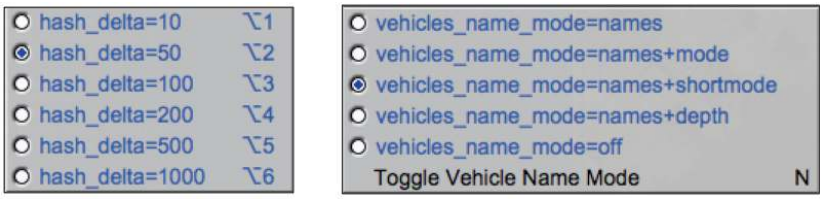

Figure 132: Most configuration parameters may be altered with pull-down menu selections. The radio-button shows the present parameter value and its neighbors show other legal settings. The text of the pull-down menu selection may be placed verbatim in the `.moos` configuration block to determine the setting upon the next mission launch. In general, menu items rendered in blue text are legally accepted parameters for placing in the `.moos` configuration block. Items in black are not.

Most parameter options have either a hot key associated with each option as shown in the left in Figure 132, or a hot key for toggling between options as on the right in the figure.

45.1.5 Recent Changes and Bug Fixes

Release 22.8.x Changes in Release 22.8.x (the first release after 22.8, until 22.8.1 is released, this means trunk/main).

- A bug in loading multiple tiff files was fixed
- Loading of multiple tiff files is supported, no longer limited to just two.
- Tiff files can now be located using a shell path variable, `IVP_IMAGE_DIRS`.

Release 22.8 (Aug 2022)

- Major new augmentation to support RealmCasting, a powerful new tool for scoping on any app in a multi-vehicle mission. Clusters of variables can be configured to scope across multiple vehicles in a single table. This works in conjunction with a new app called pRealm, which requires no configuration and runs in each MOOS community. Toggling between appcasting and realmcasting is done with the `'a'` key.
- Augmented the GUI to accept up to twenty buttons for command poking, up from the previous limit of four buttons. Buttons and info fields will automatically resize to accommodate however many buttons are used.
- Command buttons, when hovering with the mouse, will show what is being commanded upon a button click.
- `VPlug_GeoShapes` class was modified to explicitly drop from memory shapes that arrive with `active=false`. Previously this would just result in the object being ignored, but not removed. This mod guards against unbounded memory growth in pMarineViewer in some longer missions.
- `VPlug_AppCastSettings` in `lib_geometry` was replaced with `InfoCastSettings` in `lib_apputil`. This class stores all the user preferences applicable to appcasting and realmcasting. The new class is also used by `uMACView`.
- An additional variable, `REGION_INFO`, is published upon startup, and whenever a new vehicle has been detected. This info holds info about the background image, the zoom, the datum, and the pan x/y info. It is intended simply to be logged, and used by alogview upon startup to replicate the background image and orientation to be similar to how pMarineViewer was launched.
- Points and circles now have support to publish with a duration, and once the duration has been exceeded with now new publication (keyed on label), the object will be dropped from memory.
- Improved rapid drawing for large sets of Polygons
- Fixed bug where text/labels for objects off screen would be rendered on screen, appended to other objects' text, rather than simply not being drawn.
- New Option of ingesting `NODE_REPORT` info from uFldNodeComms as an intermediary. This enables smoother operation of pMarineViewer in missions with very high number of vehicles and very high time warp. Enabled with `node_report_unc=true` configuration. Of course must also be running `uFldNodeComms`.

45.2 Command-and-Control

For the most part pMarineViewer is intended to be only a receiver of information from the vehicles and the environment. Adding command and control capability, e.g., widgets to re-deploy or manipulate vehicle missions, can be readily done, but make the tool more specialized, bloated and less relevant to a general set of users. However, pMarineViewer does have a few powerful extendible command and control capabilities under the hood. Each are simply ways to conveniently post to the MOOSDB, and come in three forms: (a) configurable pull-down menu actions, and (b) contextual mouse poking with embedded oparea information, (c) configurable action buttons, and in Release 17.7 and later, (d) a configure commander pop-up window.

45.2.1 Configurable Pull-Down Menu Actions

The Action pull-down menu described in Section 46.13 provides a way to pre-define a set of MOOS postings, each selectable from the pull-down menu. For example, the alpha mission is configured with the below action:

```
action = RETURN = true
```

This post to the MOOSDB correlates to a behavior condition of the helm waypoint behavior with the return position. Actions may also be grouped into a single pull-down selection, discussed in Section 46.13.

45.2.2 Contextual Mouse Poking with Embedded OpArea Information

The mouse left and right buttons may be configured to make a post to the MOOSDB with value partly comprised of the point in the oparea under the mouse when clicked. For example, rather than commanding the vehicle to return to a pre-defined return position as the case above implies, the user may use this feature to command the vehicle to a point selected by the user at run time with a mouse click. The configuration might look like:

```
left_context[return] = RETURN_POINT = points = $(XPOS),$(YPOS)
left_context[return] = RETURN = true
```

This is discussed further in Section 46.14.

45.2.3 Action Button Configuration

Perhaps the most visible form of command and control is with the few action buttons configurable for on-screen use. For example, the DEPLOY and RETURN buttons in the lower right corner as in Figures 128, and 130. These buttons, for example, are configures as follows:

```
button_one = DEPLOY # DEPLOY=true
button_one = MOOS_MANUAL_OVERIDE=false # RETURN=false
button_two = RETURN # RETURN=true
```

The general syntax is:

```
button_one   = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
button_two   = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
button_three = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
button_four  = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
```

The left-hand side contains one of the four button keywords, e.g., `button_one`. The right-hand side consists of a `'#'`-separated list. Each component in this list is either a `'='`-separated variable-value pair, or otherwise it is interpreted as the button's label. The ordering does not matter and the `'#'`-separated list can be continued over multiple lines as in the simple example above.

The variable-value pair being poked on a button call will determine the variable type by the following rule of thumb. If the value is non-numerical, e.g., `true`, `one`, it is poked as a string. If it is numerical it is poked as a double value. If one really wants to poke a string of a numerical nature, the addition of quotes around the value will suffice to ensure it will be poked as a string.

The couple of macros are supported for button clicks:

- `$[BIX]`: An integer that increments on each button click, regardless of whether other button clicks contain this macro. It starts at zero.
- `$[UTC]`: The current time in UTC seconds, in millisecond precision.

If either of these macros appear in isolation, they will be published as type double, not type string. Support for this in the release after Release 19.8.1.

Commander Pop-Up Window In Releases 17.7 the *commander pop-up* feature was added to `pMarineViewer`. This can be thought of as the tool to use when your mission requires the more than the four buttons allowed on the lower right corner of the main screen (the *Action Buttons* described above in Section 45.2.3. The commander pop-up window contains user-configured commands to either all vehicles, or particular vehicles, and commands to the shoreside if desired. If `pMarineViewer` is configured to use the commander pop-up, this window is toggled open/closed with the space-bar key. It also contains a log at the bottom of the window, showing exactly what pokes to the MOOSDB are made upon each button click.

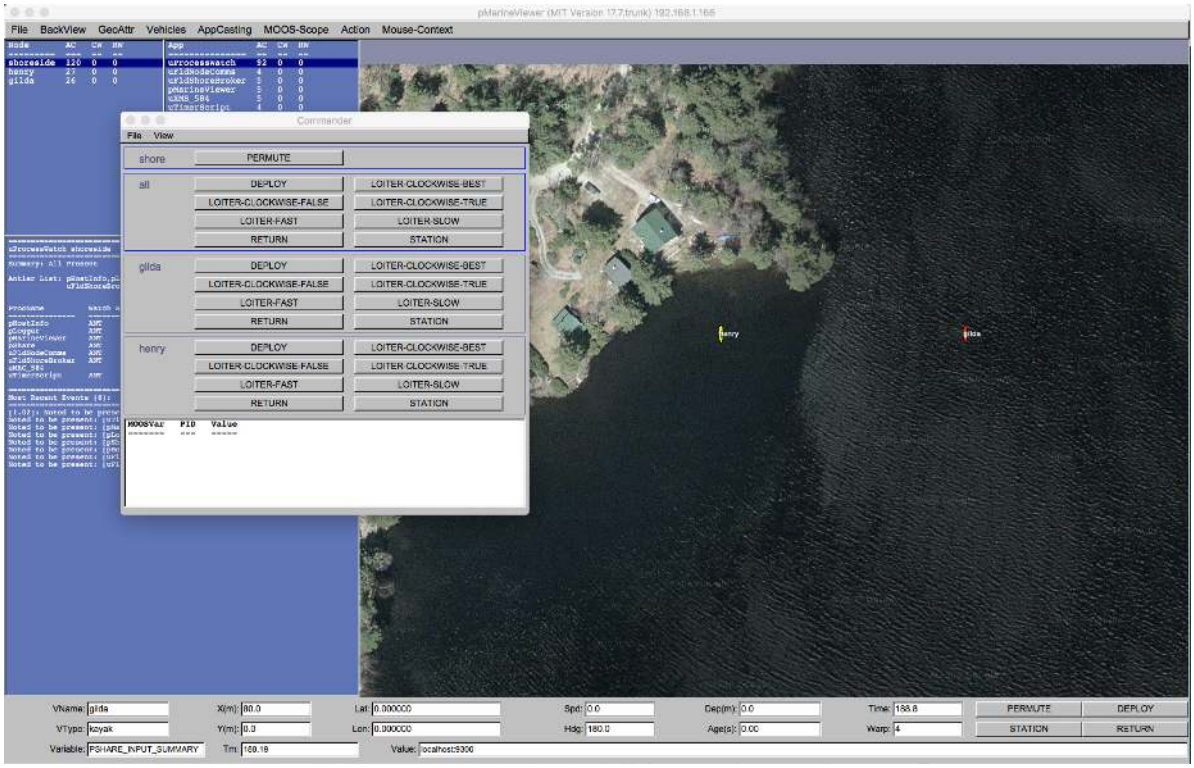

Figure 133: The commander pop-up window is toggled open/closed with the space-bar key and presents the user with a set of user-configured commands to either the all vehicles, individual vehicles, or the the pMarineViewer shoreside community itself.

The above figure was generated from the `m2_berta` mission, which may be used as a good starting example. This pop-up window is configured in the pMarineViewer configuration block as described further in Section 46.15.

45.3 The BackView Pull-Down Menu

The BackView pull-down menu deals mostly with panning, zooming and issues related to the rendering of the background on which vehicles and mission artifacts are rendered. The full menu is shown in Figure 134. Although panning and zooming is not something typically done via the pull-down menu, they are included in this menu primarily to remind the user of their hot-keys. The zooming commands affect the viewable area and apparent size of the objects. Zoom in with the 'i' or 'I' key, and zoom out with the 'o' or 'O' key. Return to the original zoom with ctrl+'z'.

45.3.1 Panning and Zooming

Panning is done with the keyboard arrow keys. Three rates of panning are supported. To pan in 20 meter increments, just use the arrow keys. To pan "slowly" in one meter increments, use the Alt + arrow keys. And to pan "very slowly", in increments of a tenth of a meter, use the Ctrl + arrow keys. The viewer supports two types of "convenience" panning. It will pan to put the active vehicle in the center of the screen with the 'C' key, and will pan to put the average of all vehicle

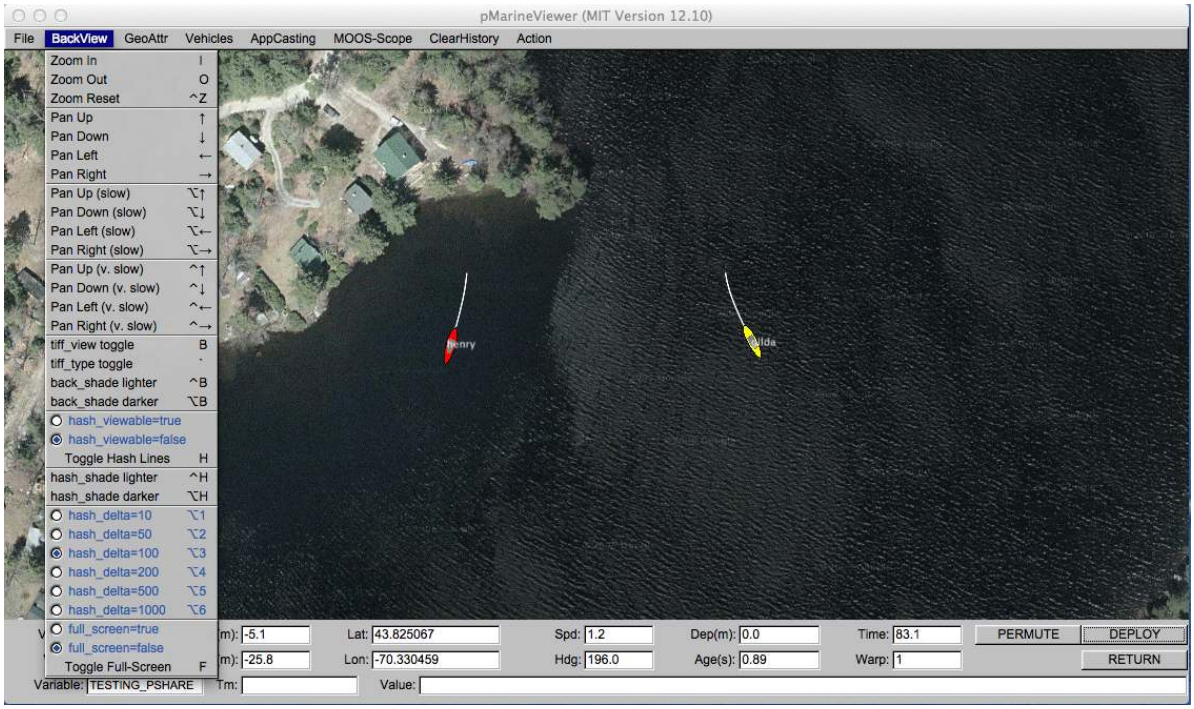

Figure 134: **The BackView menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering aspects of the geo-display background.

positions at the center of the screen with the 'c' key. These are part of the 'Vehicles' pull-down menu discussed in Section 46.9.

45.3.2 Background Images

The background can be in one of two modes; either displaying a gray-scale background, or displaying a geo image read in as a texture into OpenGL from an image file. The default is the geo display mode if provided on start up, or the grey-scale mode if no image is provided. The mode can be toggled by typing the 'b' or 'B' key. The geo-display mode can have two sub-modes if two image files are provided on start-up. This is useful if the user has access to a satellite image *and* a map image for the same operation area. The two can be toggled by hitting the back tick key. After Release 22.8.x, multiple tiff files may be provided, and toggling will cycle through all tiff files.

When in the grey-scale mode, the background can be made lighter by hitting the ctrl+'b' key, and darker by hitting the alt+'b' key.

To use an image in the geo display, the input to `pMarineViewer` comes in two files, an image file in TIFF format, and an information text file correlating the image to the local coordinate system. The file names should be identical except for the suffix. For example `dabob_bay.tif` and `dabob_bay.info`. Only the `.tif` file is specified in the `pMarineViewer` configuration block of the MOOS file, and the application then looks for the corresponding `.info` file. The info file correlates the image to the local coordinate system and specifies the location of the local `(0,0)` point. An example is given in Listing 47.

Listing 45.47: An example .info file associated with a background image.

```
1  // Lines may be in any order, blank lines are ok
2  // Comments begin with double slashes
3
4  datum_lat  = 47.731900
5  datum_lon  = -122.85000
6  lat_north  = 47.768868
7  lat_south  = 47.709761
8  lon_west   = -122.882080
9  lon_east   = -122.794189
```

All four latitude/longitude parameters are mandatory. The two datum lines indicate where $(0, 0)$ in local coordinates is in earth coordinates. However, the datum used by pMarineViewer is determined by the LatOrigin and LongOrigin parameters set globally in the MOOS configuration file. The datum lines in the above information file are used by applications other than pMarineViewer that are not configured from a MOOS configuration file. The lat_north parameters correlate the upper edge of the image with its latitude position. Likewise for the other three parameters and boundaries. Two image files may be specified in the pMarineViewer configuration block. This allows a map-like image and a satellite-like image to be used interchangeably during use. An example of this is shown in Figure 135 with two images of Dabob Bay in Washington State.

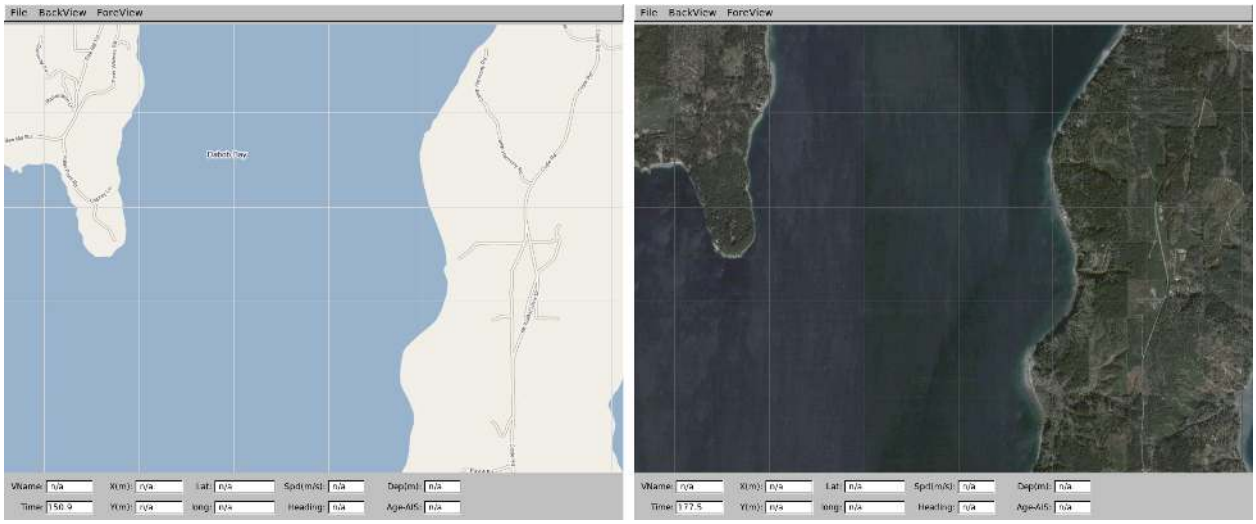

Figure 135: **Dual background geo images:** Two images loaded for use in the geo display mode of pMarineViewer. The user can toggle between both as desired during operation.

In the configuration block, the images can be specified by:

```
tiff_file   = dabob_bay_map.tif
tiff_file_b = dabob_bay_sat.tif
```

In Release 22.8.x, more than two tiff files may be provided, with successive tiff_file lines. The parameter tiff_file_b has been deprecated. By default pMarineViewer will look for the files Default.tif and DefaultB.tif in the local directory unless alternatives are provided in the configuration block.

In Release 22.8.x pMarineViewer (and alogview) make use of a shell environment variable IVP_IMAGE_DIRS. This is a colon-separated path of directories on the local computer where images files are searched for upon pMarineViewer startup. Image files with paths relative to the mission directory are still suported. One directory, moos-ivp/ivp/data/ is hard-coded into the pMarineViewer containing the Alpha mission image (Forest Lake Maine), and the MIT Sailing Pavilion.

By default a copies of the background image and info files are *not* logged by pLogger. This may be changed by the setting the following parameter: log_the_image = true. This is only a request to pLogger in the form of the PLOGGER_CMD posting:

```
PLOGGER_CMD = COPY_FILE_REQUEST = /home/jake/images/lake_george.tif
PLOGGER_CMD = COPY_FILE_REQUEST = /home/jake/images/lake_george.info
```

The result should be that the files are included in the folder created by pLogger with the ._tif and ._info suffixes. These may then be used by post-mission analysis tools to re-convey the operation area.

45.3.3 Local Grid Hash Marks

Hash marks can be overlaid onto the background. By default this mode is off, but can be toggled with the 'h' or 'H' key, or set in the configuration file with the hash_viewable parameter. The hash marks are drawn in a grey-scale which can be made lighter by typing the ctrl+'h' key, and darker by typing the alt+'h' key, or set in the configuration file with the hash_shade parameter. The hash mark spacing may only be set to one of the values shown in the menu. If set to different value, the closest legal value will be chosen.

45.3.4 Full-Screen Mode

The viewer may be put into full-screen mode by toggling with the 'f' key. This will result in the data fields at the bottom of the viewer being replace with a bit more viewing area for the geo display. As with all other blue items in this pull-down menu, the full-screen mode may be set in the MOOS configuration block with full_screen=true. The default if false. Full-screen mode is useful when running simulations while connected to a low-resolution projector for example.

46 Background Region Images

In both pMarineViewer and alogview, typical use involves a background region image upon which vehicles and other geometric objects are overlayed. Both utilies can be configured to use images of their own choosing. See examples in Figure 222. The user provides these images from whatever source they wish. Options for obtaining images and proper format are discussed below in the section *Obtaining Image Files*.

Figure 136: **Example Background Region Images**: The example images downloaded from freely available tile servers that may be utilized in either pMarineViewer or alogview.

46.1 Default Packaged Images

Image files can be large, and of course there are endless possibilities depending on where you are operarating and which kind of background images you prefer. That being said, a couple images are distributed with MOOS-IvP. This allows example missions distributed with the MOOS-IvP code to having working images out-of-the-box without requiring the new user to fetch images. These two images are for (a) the MIT Sailing Pavilion, and (b) Forest Lake in Gray Maine, the site of some of the earliest in-water experiments circa 2004. This files are:

- `MIT_SP.tif`
- `forrest19.tif`

Both are distributed with MOOS-IvP and can be found in `moos-ivp/ivp/data`. When pMarineViewer or alogview is launched, this directory will be examined for the specified image file. Instructions for loading user-provided images is are given below in section *Loading Images at Run Time*.

46.2 Image File Format and Meta Data (Info Files)

Images loaded into pMarineViewer and alogview are in the format of `"Tiff"` files. These files have the suffix `".tif"`. Tiff files have been around since the eighties, use lossless compression, are typically high quality, but are not as common as formats such as jpeg. There are many freely available tools for converting back and forth between tiff and jpeg and other formats. Tiff files are used in pMarineViewer and alogview primarily due to the availability of the `libtiff` library, readily available through package managers on both the MacOS and Linux platforms.

Each `.tif` file used in pMarineViewer and alogview has a corresponding `.info` file, containing (a) the latitude and longitude coordinates of the image edges, and (b) the datum, or `(0,0)` coordinate, on the image. For example, the `MIT_SP.tif` file has a corresonding `MIT_SP.info` file found in the same directory.

```
lat_north = 42.360264
lat_south = 42.35415
lon_east  = -71.080058
lon_west  = -71.094214
datum_lat = 42.358436
datum_lon = -71.087448
```

46.3 Obtaining Image Files

Image files may be obtained and used in pMarineViewer and alogview from any source convenient to the user. This includes opening an image in say Google Maps on a web browser and performing a screen grab. The drawback of this method however is that it may be hard to precisely determine the lat/lon coordinates of the edges used in the corresponding `.info` file.

There are several open *tile servers* which allow a user to download an image tile, or set of tiles, provided with a range of lat/lon coordinates. These tiles can then be stitched together to make a single image. Although this sounds cumbersome, this process can be automated in a script. One such script is Anaxi Map, written by Conlan Cesar:

https://github.com/HeroCC/AnaxiMap

This utility is capable of using one of several tile servers with various background styles, such as Google Maps, maps with terrain or bathymetry information, or maps with street data. See Figure 222.

AnaxiMap, or similar utilities, may produce images in jpeg or png format. MacOS and Linux provide native utilities for converting the format, or "exporting" the file, to tiff format. On MacOS or Linux, if the free ImageMagick package is installed, you can use the "convert" utility:

```
$ convert region.jpg region.tif
```

46.4 Loading Images at Run Time

Post Release 22.8, both pMarineViewer and alogview support operation with multiple background images. Toggling between images at run time is done by selecting the `BackView` pull-down menu and selcting `tiff_type toggle`, or by simply hitting the ‘ (back-tick) key.

In pMarineViewer, the multiple background images may be specified with multiple configuration lines, for example:

```
tiff_file  = MIT_SP.tif
tiff_file  = mit_sp_osm18.tif
```

In alogview, the multiple background images may be specified on the command line:

```
$ alogview --bg=MIT_SP.tif --bg=mit_sp_osm18.tif file.alog
```

46.5 Automatic alogview Detection of Background Image

When launching alogview typically the user wants to use the same background region image used by pMarineViewer during the course of the mission that generated the alog file. In a new feature, post Release 22.8, alogview will automatically attempt to detect the image file used by pMarineViewer. The name name of region image is now published by pMarineViewer during the execution of the

mission. This information is contained in the variable REGION_INFO. For example:

```
REGION_INFO = lat_datum=42.358436, lon_datum=-71.087448, img_file=MIT_SP.tif,\
              zoom=2.5, pan_x=129, pan_y=-364
```

The region info contains the name of the image (tiff) file used during the course of the mission, as well as the pan and zoom information as hints for alogview for use upon startup. The images found in REGION_INFO in the alog file will be loaded, as well as any images specified on the command line with the --bg options.

46.6 Background Image Path

Support for the IVP_IMAGE_DIRS shell path is a new feature, post Release 22.8, relevant to both pMarineViewer and alogview. This is explained below.

Image files are named in either the pMarineViewer config block of the .moos mission file, or named on the command line when launching alogview, as specified in Section *Loading Images at Run Time*. For alogview, the file may also be named in the REGION_INFO logged variable as discussed above.

Both apps need to *find* the named .tif file. When found, it looks for the corresponding .info file in the same directory. There are four options for making this work:

- The image file is in the same directory as the mission file.
- The image file is in the special directory moos-ivp/ivp/data.
- The full or relative path name of the file is specified.
- The file exists in a directory on your IVP_IMAGE_DIRS path.

The first option has the drawback of duplicating the image file in potentially many places. The second option has the drawback that the directory moos-ivp/ivp/data is part of the MOOS-IvP code distribution which users otherwise consider to be read-only. A fresh check out of MOOS-IvP would reset this directory and users would need to take care to migrate files to a new checkout. The third option is that full or relative path name may not be the same between different users or machines. The fourth option is the newest option and arguably has the least downside.

The IVP_IMAGE_DIRS is shell (e.g., bash) environment variable. It contains a list of one or more directories on your local computer where pMarineViewer or alogview will look when attempting to load image files. Shell environment variables are already common settings that users will customize on their particular machine.

The recommended way for users to use a set of custom image files is to (a) organize them in one or more directories, preferably under version control, (b) install them at a convenient location on your local machine, (c) configure the IVP_IMAGE_DIRS shell variable to contain the one or more directories where your image files reside.

For example, if you have a folder of image files with the following structure:

```
my_images/
   napa_bay/
      napa_bay_gmaps.tif
      napa_bay_gmaps.info
      napa_bay_osm.tif
      napa_bay_osm.info
   happy_harbor/
      happy_harbor_gmaps.tif
      happy_harbor_gmaps.info
      happy_harbor_osm.tif
      happy_harbor_osm.info
```

If this folder is installed on your machine in the home dirctory folder call "project", then you would set your IVP_IMAGE_DIRS path in your shell configuration file, e.g., .bashrc, as follows:

```
IVP_IMAGE_DIRS=~/project/napa_bay
IVP_IMAGE_DIRS+=:~/project/happy_harbor
```

To verify which file has been loaded, the appcasing output of pMarineViewer shows the full path name of the loaded file(s). And when alogview is launched, the terminal output indicates which directories are being searched, in order, for the image files. This information may be obscured however when the alogview window pops up, but you can find it if you go back to it and perhaps scroll up. Note: It is not sufficient, in the example above, to simply set IVP_IMG_DIR= /project, the parent directory of all image folders. Each image folder must be named.

46.7 Troubleshooting

46.7.1 pMarineViewer fails to load the image (see only gray screen)

1. Check the appcasting output of pMarineViewer. The top few lines should show which image file is loaded. Is this a file you recognize?
2. Does this file exist on your computer? Verify it is where you think it is.
3. How are you specifying this file in your pMarineViewer config block? If it is specified as a relative path, e.g., ../my_images/napa_bay.tif, make sure that relative path location is correct.
4. If you are specifying the image file with just the file name (no path information), then check you have your IVP_IMAGE_DIRS variable set properly. Run echo $IVP_IMAGE_DIRS on the command line.
5. Simplest but most common: Make sure your image file name (configuration and actual name) end in the suffix .tif and not .tiff.

46.7.2 pMarineViewer or alogview image is fine but no vehicles

1. Check the .info file. Make sure the lat/lon values sanity check, e.g., rough magnitude, relative values.
2. Make sure the datum_lat and datum_lon values are in the range of the image.
3. Make sure the datum_lat and datum_lon values match the datum set at the top of the vehicle and shoreside mission files.

46.7.3 alogview fails to load the image (see only gray screen)

In the newer version of alogview, when launching it will attempt to read the name of the image file used by pMarineViewer. So if pMarineViewer launched ok, chanches are good alogview will also launch with the same image. However, it could be the case that (a) the mission was run on some other computer that contained the image file and your current computer does not. The image file is not logged. (b) the mission was named

1. Does this file exist on your computer? Verify it is where you think it is. It is possible that you are trying to run alogview on your computer with alog files generated on someone else's computer. Make sure you have the image file.
2. Check the terminal output of alogview as it is loading. To demonstrate the below output from an alogview launch purposely use the the file `dforrest19.tif` instead of `forrest19.tif`. Note the sequences of folders searched in the attempt to find the image file. The first attempt is the in the `moos-ivp/ivp/data` directory. The next attempts are based on the value of `IVP_IMAGE_DIRS`. The final attempt is in the current working directory where alogview was launched. Does this match up with your expectations?

```
TIFF FILES COUNT:1
[1] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/data]
    Not found.
[2] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/pavlab_map_images/popolopen]
[3] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/pavlab_map_images/mit]
[4] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/datax]
[5] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/data-local]
[6] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/data]
[*] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [./]
    Not found.
Could not find the pair of files:
dforrest19.tif and dforrest19.info
Opening Tiff:
TIFFOpen: : No such file or directory.
  Failed!!!!!!!!!
```

3. Simplest but most common: Make sure your image file name (configuration and actual name) end in the suffix `.tif` and not `.tiff`.

46.8 The GeoAttributes Pull-Down Menu

The GeoAttributes pull-down menu allows the user to affect viewing properties of geometric objects capable of being rendered by the pMarineViewer. The viewer subscribes for and supports the following geometric objects, typically generated by the helm or other MOOS applications:

- Polygons
- SegLists
- Points
- Vectors
- Circles
- Markers
- RangePulses
- CommsPulses

The viewer will also render the following other geometric objects set either in the configuration file or interactively by the user:

- Datum
- OpArea
- DropPoints

The Datum is simply the point in local coordinates representing `(0,0)`. The pull-down menu allows the user to toggle off or on this rendering of the datum point as well as adjust its size and color. The OpArea is used to render the boundaries, if they exist, of an area of operation. DropPoints (described further in Section 46.8.7) are labeled points the user may drop on the viewing area for reference or mission planning

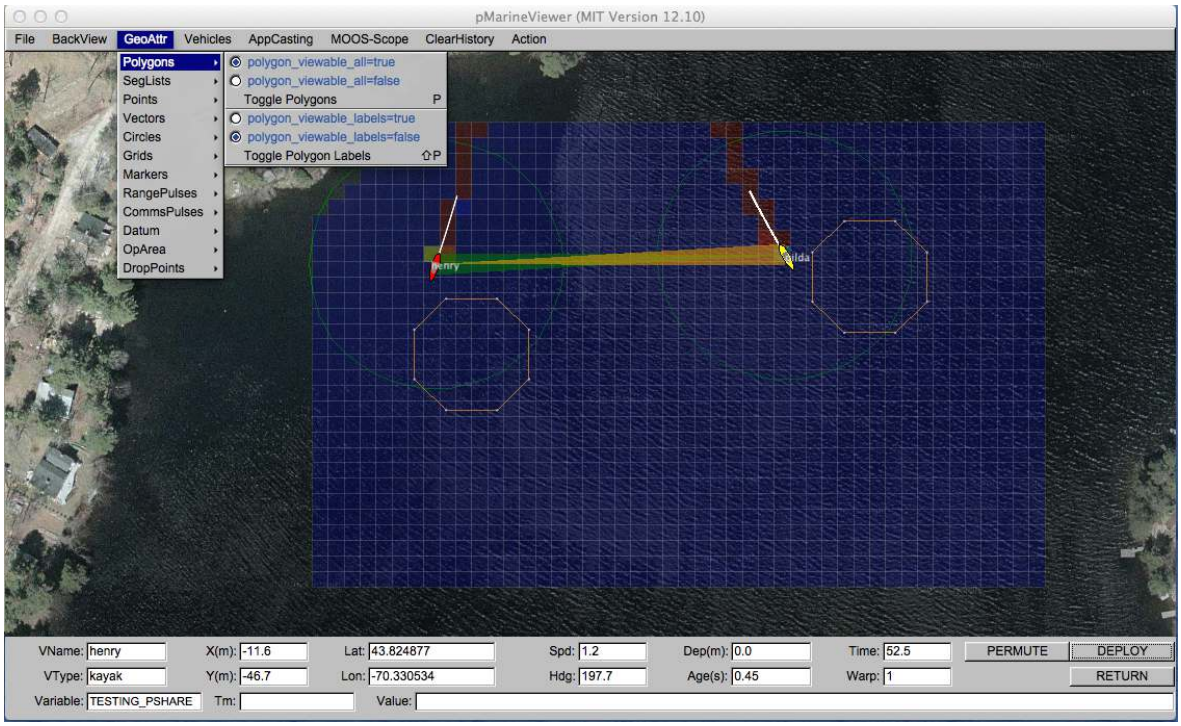

Figure 137: **The "GeoAttr" menu:** This pull-down menu lists the options and hot keys for affecting the rendering of geometric objects.

The possible parameters settings for rendering the geometric objects received by pMarineViewer via MOOS mail is provided in Section 46.16.2.

46.8.1 Polygons, SegLists, Points, Circles and Vectors

The five geometric objects, polygons, seglists, points, circles and vectors, provide a core rendering capability for applications (like the helm and its behaviors) to render visual artifacts related to the unfolding of a mission. For example, in Figure 128, a seglist is used to render the vehicle waypoints, and a labeled point is used to render the vehicles current next waypoint.

Objects are passed to pMarineViewer as strings via normal MOOS mail. An example is given below for the seglist shown in Figure 128. The string is a comma-separated list of variable=value pairs. Note the last pair is a label. Labels are used by all five object types to distinguish uniqueness.

```
VIEW_SEGLIST =  pts={60,-40:60,-160:150,-160:180,-100:150,-40},label=waypt_survey
```

Uniqueness is used to either overwrite or erase previously rendered object instances. For example the above seglist could be "moved" five meters south by posting an identical message with the same label and adjusted coordinates. The *source* of the object is also tracked by pMarineViewer. This is given by the MOOS community from which the message originated, and typically represents the vehicle's name. Thus the above seglist could also be "moved" if the posting originated from a second vehicle community, in the type of arrangement shown in Figure 129.

46.8.2 Parameters Common to Polygons, SegLists, Points, Circles and Vectors

Other optional parameters may be associated with an object to specify rendering preferences. They include:

- active
- msg
- vertex_size:
- vertex_color
- edge_size
- edge_color
- fill_color
- fill_transparency

For example, the VIEW_SEGLIST specification above may be augmented with the below string to specify edge and vertex size and color preferences:

```
edge_color=pink,vertex_color=blue,vertex_size=4,edge_size=1
```

The active parameter may be set to false to indicate that an object, previously received with the same label, should not be drawn by pMarineViewer. The msg parameter may be used to override

the string rendered as the object's label. Since labels are used to uniquely identify an object, the `msg` parameter may be used to, for example, draw five points all with same rendered text. The other six parameters are self-explanatory and not necessarily relevant to all objects. For example, pMarineViewer will ignore an `edge_size` specification when drawing a point, and a `fill_color` will only be relevant for a polygon and a circle.

46.8.3 Serializing Geometric Objects for pMarineViewer Consumption

Geometric objects are only *consumed* by pMarineViewer, but it's worth discussing the issue of *generating* and serializing an object into a string. It is possible to simply post a string in the right format, as with:

```
string str = "x=5,y=25,label=home,vertex_size=3";  // Not recommended
m_Comms.Notify("VIEW_POINT", str);
```

It is highly recommended that this be left to the serialization function native to the C++ class.

```
#include "XYPoint.h"

XYPoint my_point(5, 25);                          // Recommended
my_point.set_label("home");
my_point.set_vertex_size(3);
string str = my_point.get_spec();
m_Comms.Notify("VIEW_POINT", str);
```

The latter code is less prone to user error, and is more likely to work in future code releases if the underlying formats need to be altered. (This is the idea behind Google Protocol Buffers, but here the geometric classes are implemented with various geometry function relations defined in addition to the serialization and de-serialization.) The full set of interface possibilities for creating and manipulating geometry objects is beyond the scope of the discussion here however.

46.8.4 Markers

A set of marker object types are defined for rendering characteristics of an operation area such as buoys, fixed sensors, hazards, or other things meaningful to a user. The six types of markers are shown in Figure 138. They are configured in the pMarineViewer configuration block of the MOOS file with the following format:

```
marker  = type=efield,x=100,y=20,label=alpha,color=red,width=4.5
marker  = type=square,lat=42.358,lon=-71.0874,color=blue,width=8
```

Each entry is a string of comma-separated pairs. The order is not significant. The only mandatory fields are for the marker type and position. The position can be given in local x-y coordinates or in earth coordinates. If both are given for some reason, the earth coordinates will take precedent. The `width` parameter is given in meters drawn to scale on the geo display. Shapes are roughly 10x10

meters by default. The GUI provides a hook to scale all markers globally with the ALT-m and CTRL-m hot keys and in the GeoAttributes pull-down menu.

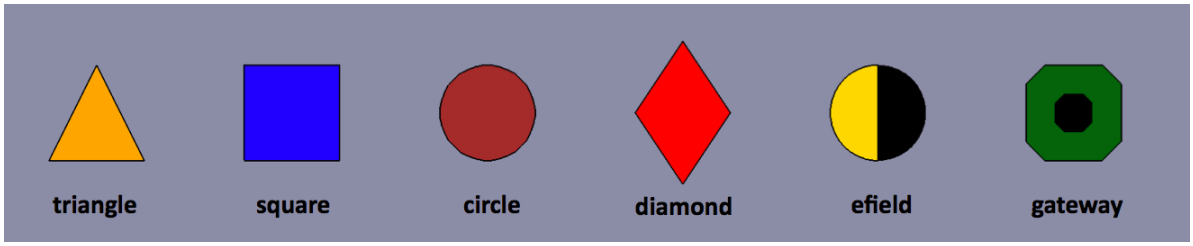

Figure 138: **Markers:** Types of markers known to the pMarineViewer.

The color parameter is optional and markers have the default colors shown in Figure 138. Any of the colors described in the Colors Appendix are fair game. The black part of the Gateway and Efield markers is immutable. The label field is optional and is by default the empty string. Note that if two markers of the same type have the same non-empty label, only the first marker will be acknowledged and rendered. Two markers of different types can have the same label.

In addition to declaring markers in the configuration file, markers can be received dynamically by pMarineViewer through the VIEW_MARKER MOOS variable, and thus can originate from any other process connected to the MOOSDB. The syntax is exactly the same, thus the above two markers could be dynamically received as:

```
VIEW_MARKER  = "type=efield,x=100,y=20,scale=4.3,label=alpha,color=red,width=4.5"
VIEW_MARKER  = "type=square,lat=42.358,lon=-71.0874,scale=2,color=blue,width=8"
```

The effect of a "moving" marker, or a marker that changes color, can be achieved by repeatedly publishing to the VIEW_MARKER variable with only the position or color changing while leaving the label and type the same. To dynamically alter the text rendered with a marker, the msg=value field may be used instead. When the message is non-empty, it will be rendered instead of the label text.

46.8.5 Comms Pulses

Comms pulse objects were designed to convey a passing of information from one node to another. At this writing, they are only used by the uFldNodeComms application, but from the perspective of pMarineViewer it does not matter the origin. The MOOS variable is VIEW_COMMS_PULSE. They look something like that shown in Figure 139. There are two pulses shown in this figure. In this case they were posted by uFldNodeComms to indicate that the two vehicles are receiving each other's node reports.

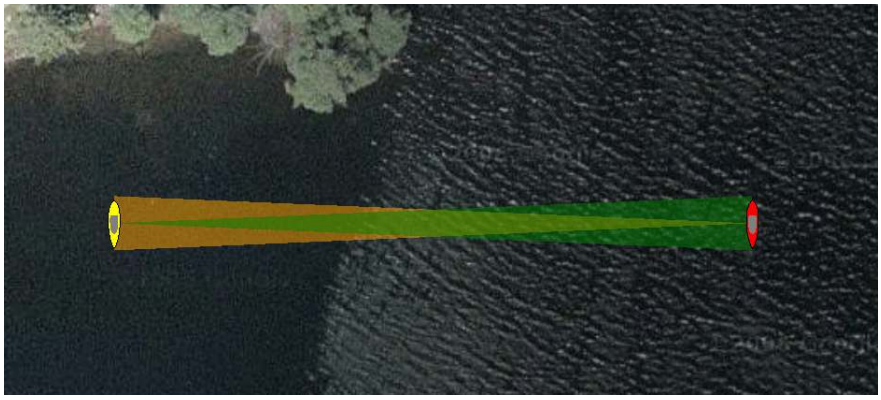

Figure 139: **Comms Pulses:** A comms pulse directionally renders communication between vehicles. Here each vehicle is communicating with the other, and two different colored pulses are rendered.

The term "pulse" is used because the object has a duration (by default three seconds), after which it will no longer be rendered by pMarineViewer. The pulse will fade (become more transparent) linearly with time as it approaches its expiration. If a subsequent comms pulse is received with an identical label before the first pulse times out, the second pulse will replace the first, in the style of other geometric objects discussed previously. Although serializing and de-serializing comms pulse messages is outside the scope of this discussion, it it worth examining an example comms pulse message:

```
VIEW_COMMS_PULSE = sx=91,sy=29,tx=6.7,ty=1.4,beam_width=7,duration=10,fill=0.35,
                   label=GILDA2HENRY_MSG,fill_color=white,time=1350201497.27
```

As with the object types discussed previously, the construction of the above type messages should be handled by the `XYCommsPulse` class along the line of something like:

```
#include "XYCommsPulse.h"

XYCommsPulse my_pulse(91, 29, 6.7, 1.4);
my_pulse.set_label("GILDA2HENRY_MSG");
my_pulse.set_duration(10);
my_pules.set_beam_width(7);
my_pules.set_fill(0.35);
my_pulse.set_color("fill", "white");
string str = my_pulse.get_spec();
m_Comms.Notify("VIEW_COMMS_PULSE", str);
```

The white comms pulse shown in Figure 140 indicates that a message has been sent from one vehicle to the other. The fat end of the pulse indicates the receiving vehicle. The color scheme is not a convention of pMarineViewer, but rather a convention of the uFldNodeComms application which generated the object in this case. A white pulse is typically rendered long enough to allow the user to visually register the information. It also typically does not move with the vehicle, to convey to the user the vehicle positions at the time of the communication.

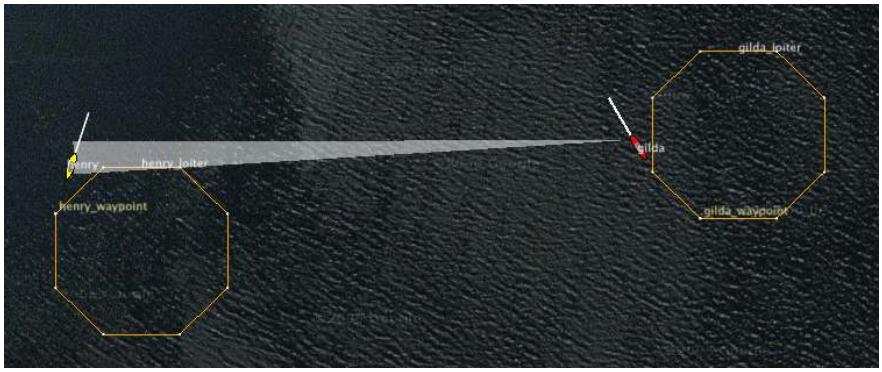

Figure 140: **Comms Pulses for Messaging:** In this figure the white comms pulse indicates that a message is being sent from one vehicle to another, via `uFldNodeComms`.

The rendering of comms pulses may be toggled on or off in `pMarineViewer` via a selection in the GeoAttr pull-down menu, or via the `'@'` hot key. It is not possible in `pMarineViewer` to show just the white comms pulses, and hide the colored node report comms pulses, or vice versa. It is possible however in the `uFldNodeComms` configuration to shut off the node report pulses with `view_node_rpt_pulses=false`.

46.8.6 Range Pulses

Range pulse objects were designed to convey a passing of information or sensor energy from one node to any other node in the vicinity, up to a certain range. At this writing they are only used by the `uFldContactRangeSensor` and `uFldBeaconRangeSensor` applications, but from the perspective of `pMarineViewer` it does not matter the origin. The MOOS variable is `VIEW_RANGE_PULSE`. They look something like that shown in Figure 141. Here the pulse is shown over three successive times.

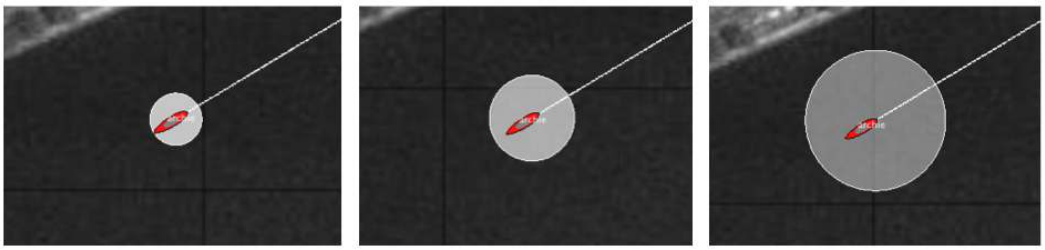

Figure 141: **Comms Pulses:** A comms pulse directionally renders communication between vehicles. Here each vehicle is communicating with the other, and two different colored pulses are rendered.

The term "pulse" is used because the object has a duration (by default 15 seconds), after which it will no longer be rendered by `pMarineViewer`. The pulse will grow in size and fade (become more transparent) linearly with time as it approaches its expiration. If a subsequent range pulse is received with an identical label before the first pulse times out, the second pulse will replace the first, in the style of other geometric objects discussed previously. Although serializing and de-serializing range pulse messages is outside the scope of this discussion, it it worth examining an example range pulse message:

```
VIEW_RANGE_PULSE = x=99.2,y=68.9,radius=50,duration=6,fill=0.9,label=archie_ping,
                   edge_color=white,fill_color=white,time=2700438154.35,edge_size=1
```

As with the object types discussed previously, the construction of the above type messages should be handled by the `XYRangePulse` class along the line of something like:

```
#include "XYRangePulse.h"

XYRangePulse my_pulse(99.2, 68.9);
my_pulse.set_label("archie_ping");
my_pulse.set_duration(6);
my_pules.set_edge_size(1);
my_pules.set_radius(50);
my_pules.set_fill(0.9);
my_pulse.set_color("edge", "white");
my_pulse.set_color("fill", "white");
string str = my_pulse.get_spec();
m_Comms.Notify("VIEW_RANGE_PULSE", str);
```

46.8.7 Drop Points

A user may be interested in determining the coordinates of a point in the geo portion of the pMarineViewer window. The mouse may be moved over the window and when holding the `SHIFT` key, the point under the mouse will indicate the coordinates in the local grid. When holding the `CTRL` key, the point under the coordinates are shown in lat/lon coordinates. The coordinates are updated as the mouse moves and disappear thereafter or when the `SHIFT` or `CTRL` keys are release. Drop points may be left on the screen by hitting the left mouse button at any time. The point with coordinates will remain rendered until cleared or toggled off. Each click leaves a new point, as shown in Figure 142.

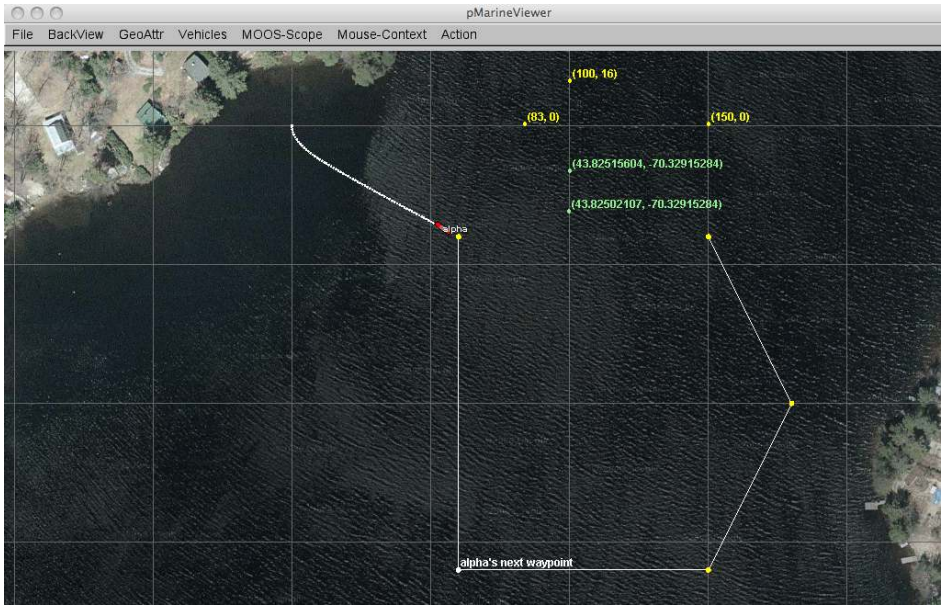

Figure 142: **Drop points:** A user may leave drop points with coordinates on the geo portion of the pMarineViewer window. The points may be rendered in local coordinates or in lat/lon coordinates. The points are added by clicking the left mouse button while holding the SHIFT key or CTRL key. The rendering of points may be toggled on/off, cleared in their entirety, or reduced by popping the last dropped point.

Parameters regarding drop points are accessible from the GeoAttr pull-down menu. The rendering of drop points may be toggled on/off by hitting the 'r' key. Drop points may also be shut off in the mission configuration file with drop_point_viewable_all=false. The set of drop points may be cleared in its entirety via the pull-down menu. Or the most recently dropped point may be removed by typing the CTRL-r key. The pull-down menu may also be used to change the rendering of coordinates from "as-dropped" where some points are in local coordinates and others in lat/lon coordinates, to "local-grid" where all coordinates are rendered in the local grid, or "lat-lon" where all coordinates are rendered in the lat/lon format. By default the mode is "as-dropped". The startup default mode may be changed with drop_point_coords=local-grid for example in the mission file.

46.9 The Vehicles Pull-Down Menu

The *Vehicles* pull-down menu deals with rendering properties of vehicles, vehicle labels, and vehicle trails. The options are shown in Figure 143. The very first option is to turn on or off the rendering of all vehicles. The can be done at run time via the menu selection, or toggled with the `Ctrl-'a'` hot key. Like all blue options in this menu, the text in the menu item may be placed verbatim in the mission configuration file to reflect the user's startup preferences.

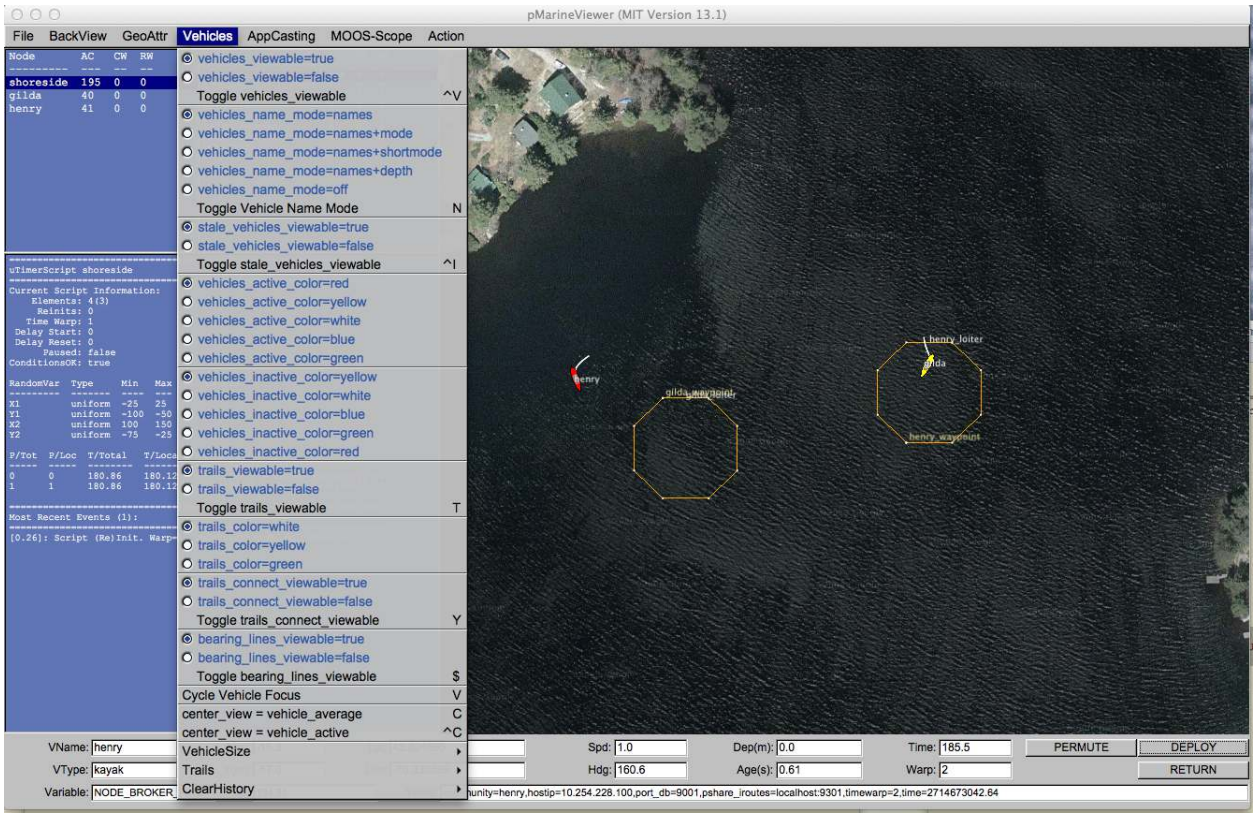

Figure 143: **The Vehicles menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering vehicles and vehicle track history.

46.9.1 The Vehicle Name Mode

Each vehicle rendered in the viewer has an optional label rendered with it. This label may be rendered in one of five modes:

- *names*: Just the vehicle name is rendered.
- *names+mode*: The vehicle name and the full helm mode is rendered.
- *names+shortmode*: The vehicle name and the short helm mode is rendered.
- *names+depth*: The vehicle name and its current depth are rendered.
- *off*: No label is rendered.

The default is *names+shortmode*. The *names*, *off* and *depth* modes are self explanatory. The *names+mode* and *names+shortmode* involve information typically provided in vehicle node reports

about the state of the IvP helm. The helm uses hierarchical mode declarations as a way of configuring behaviors for missions. The helm mode for example be described with string looking something like "MODE@ACTIVE:LOITERING". In pMarineViewer the text next to the vehicle would be either this whole string if configured with the *names+mode* setting, or just "LOITERING" if configured with the *names+shortmode* setting.

The color of the rendered text may be changed from the default of white to any color in the Color Appendix with the vehicles_name_color configuration parameter.

46.9.2 Dealing with Stale Vehicles

A stale vehicle is one who has not been heard from for a some time, perhaps because the vehicle is disabled, out of range, or recovered from the field. These vehicles can be a distraction. Their history may be outright cleared as described in Section 46.9.6, but this requires action by the user or a posting to the MOOSDB.

Stale vehicles are also automatically dealt with by pMarineViewer in another way. After some number of seconds (30 by default), the vehicle label indicates the staleness. The label may look something like "henry (Stale Report: 231)" where the number indicates the number of seconds since the last node report received for this vehicle. After another period of time (30 by default), the vehicle may no longer rendered and removed from the appcasting pane.

A few features of this policy are configurable through the mission configuration file. The duration of time after which a vehicle is reported as stale may be changed from its default of 30 seconds with the stale_report_thresh parameter. The duration of time after which a vehicle is removed may be changed from its default of 30 seconds with the stale_remove_thresh parameter.

Stale vehicles are handled a bit differently when running in simulation and when running vehicles in the field. The difference between the two is determined by the MOOS time warp. Although it's possible to simulate with a time warp of one, here a time warp of one is interpreted as running physical vehicles. Simulated vehicles will be automatically removed from the viewer after stale_report_thresh + stale_remove_thresh seconds. When running actual vehicles, stale vehicles must be explicitly removed using the alt+'k' key to remove all stale vehicles, or ctrl+'k' key to remove the currently selected vehicle in the appcast pane.

46.9.3 Supported Vehicle Shapes

The shape rendered for a particular vehicle depends on the *type* of vehicle indicated in the node report received in pMarineViewer. There are several types that are currently handled:

- kayak
- uuv
- auv (same as uuv)
- glider
- mokai
- ship
- longship

- heron
- buoy
- swimmer
- cray, crayx, bcray, bcrayx

Note: the swimmer, buoy, cray, bcray, crayx, and bcrayx, types were introduced *after* release 22.8.

Some shapes are shown in Figure 144.

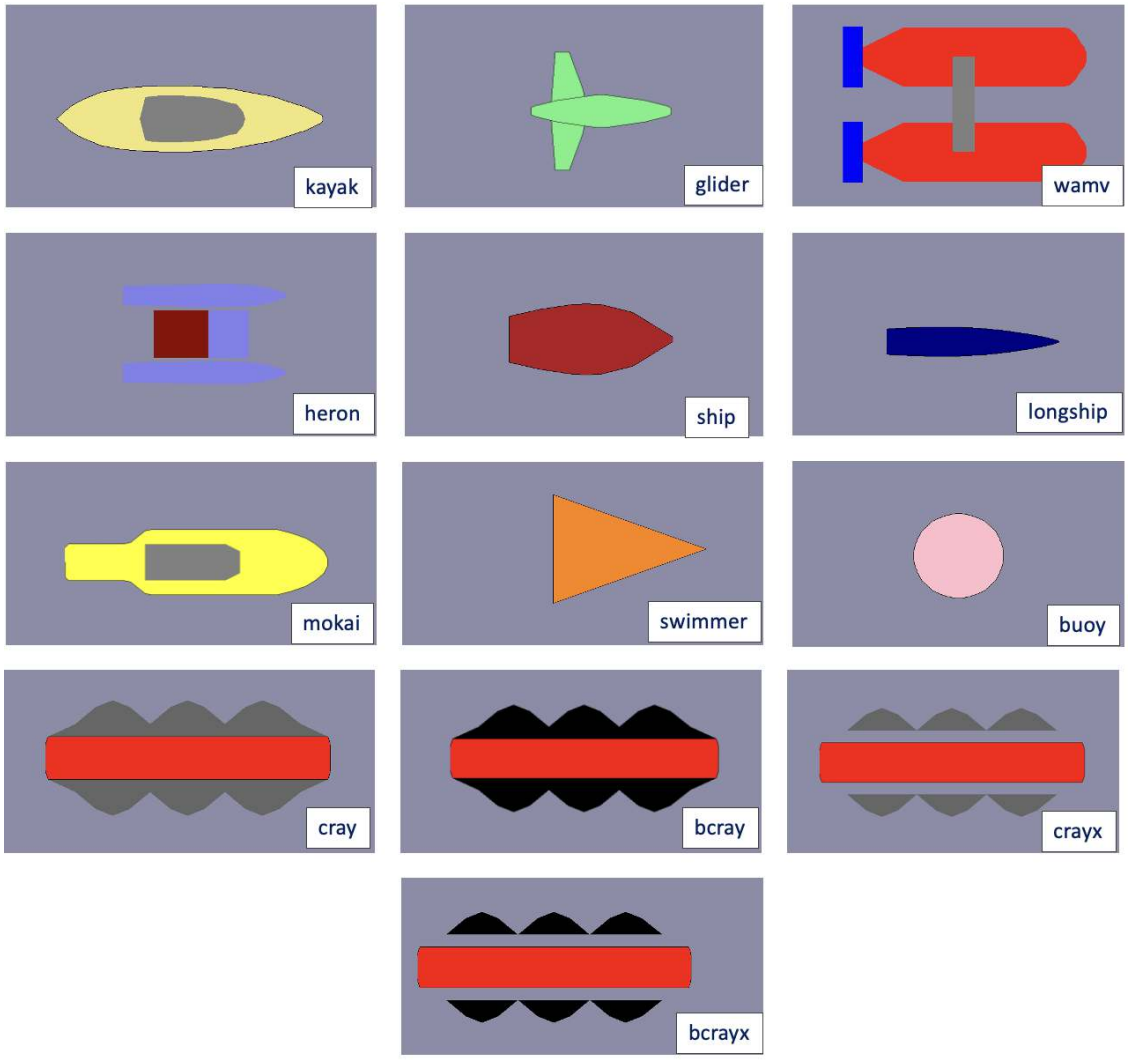

Figure 144: **Vehicles:** Types of vehicle shapes supported by pMarineViewer. The C-Ray shapes were added *after* release 22.8.

The default shape for an unknown vehicle type is currently set to be the shape "ship".

46.9.4 Vehicle Colors

Vehicles are rendered in one of two colors, the *active vehicle color* and the *inactive vehicle color*. The active vehicle is the one who's data is being rendered in the data fields at the bottom of the pMarineViewer window, and who's name is in the `VName:` field. The active vehicle may be changed by selecting `"Cycle Vehicle Focus"` from the Vehicles pull-down menu, or toggling through with the `'v'` key. The default color for the active vehicle is red, and the default for the inactive vehicle is yellow. These can be changed via the pull-down menu, or with the following parameters in the configuration file:

```
vehicles_active_color   = <color>    // default is red
vehicles_inactive_color = <color>    // default is yellow
```

The parameters and colors are case insensitive. All colors of the form described in the Color Appendix are acceptable.

46.9.5 Centering the Image According to Vehicle Positions

The `center_view` menu items alters the center of the view screen to be panned to either the position of the active vehicle, or the position representing the average of all vehicle positions. Once the user has selected this, this mode remains *sticky*, that is the viewer will automatically pan as new vehicle information arrives such that the view center remains with the active vehicle or the vehicle average position. As soon as the user pans manually (with the arrow keys), the viewer breaks from trying to update the view position in relation to received vehicle position information. The rendering of the vehicles can made larger with the '+' key, and smaller with the '-' key, as part of the `VehicleSize` pull-down menu as shown. The size change is applied to all vehicles equally as a scalar multiplier. Currently there is no capability to set the vehicle size individually, or to set the size automatically to scale.

46.9.6 Vehicle Trails

Vehicle trail (track history) rendering can be toggled off and on with the `'t'` key. The default is on. The startup default setting may be changed to off in the mission configuration file with `trails_viewable=false`.

46.9.7 Trail Color and Point Size

The trail color by default is white. A few other colors are available in the Vehicles pull-down menu. A color may also be chosen in the mission configuration file with `trail_color=<color>` using any color listed in the Color Appendix. The trail point size may range from $[1, 10]$. The default setting is 2. The size may be changed at runtime by the Vehicles/Trails pull-down menu, or with the `'{'` and `'}'` hot keys. The startup trail size may also be set in the mission configuration file with `trails_point_size=<int>` parameter.

46.9.8 Trail Length and Connectivity

Trails have a fixed-length history by default of 100 points. This may be changed via the Vehicles/Trails pull-down menu, or with the hot keys `'('` and `')'`. The startup default length may also be set in

the mission configuration file with trails_length=<int> with values in range of $[0, 10000]$.

Individual trail points can be rendered with a line connecting each point, or by just showing the points. When the node report stream is flowing quickly, typically the user doesn't need or want to connect the points. When the viewer is accepting input from an AUV with perhaps a minute or longer delay in between reports, the connecting of points is helpful. This setting can be toggled with the 'y' or key, with the default being off. The startup default may be set to on with the mission file parameter trails_connect_viewable=true.

46.9.9 Resetting or Clearing the Trails

A vehicle's history sometimes needs to be cleared, for example when a vehicle has not been heard from in a long time, or has been recovered. Its trails and other geometric objects posted to the viewer can become a distraction. This may be done in a couple ways. First via the Action pull-down menu, the last menu item allows the user to clear the history of all vehicles or a selected vehicle. The Ctrl-9 hot key can be used to clear all vehicle histories. A select vehicle history may also be cleared by posting to the MOOS variable TRAIL_RESET with the name of the vehicle.

46.10 The InfoCasting Pull-Down Menu

Infocasting refers to two optional modes for operating pMarineViewer. The first is *appcasting*, introduced in 2012 in Release 12.11. The second mode is *realmcasting*, developed in December 2020 for the first release following 19.8. With these tools, pMarineViewer has been augmented to serve as an appcast and realmcast viewer. Other apps may be used for appcast viewing, such as uMAC, and uMACView. Realmcast viewing is also supported in uMACView, but not uMAC.

Appcasting requires that each app be "appcast enabled", and virtually all MOOS apps distributed with MOOS-IvP are appcast enabled. The motivation for appcasting and how to build appcast enabled MOOS applications are discussed elsewhere in the appcasting documentation. Realmcasting, on the other hand, requires nothing of the user other than running an app called pRealm in each MOOS community. In the simplest case, pRealm can be run with no configuration parameters, allowing it to be very easily added to legacy missions.

Quick points to note:

- The term *infocasting* refers generally to include either appcasting or realmcasting.
- By default, when pMarineViewer launches, Infocasting panes on the left are rendering appcast content, typically with a white-on-indigo color scheme.
- The content of the infocasting panes can be switched between appcasting and realmcasting by hitting the 'a' key.
- The infocasting panes can be toggled/hidden entirely by hitting the ctrl-'a' key.

46.10.1 Turning On and Off InfoCast Viewing

The InfoCasting pull-down menu, shown in Figure 145 allows adjustments to be made to the infocast rendering and content. The very first set of menu options allows the user to control whether the infocasting panes are shown or not (toggling with ctrl-'a', followed by options for controlling the content, appcasting or realmcasting. As with all pMarineViewer pull-down menu items, if they are

rendered in blue, then the same text can be placed in the pMarineViewer configuration block to be applied automatically at startup.

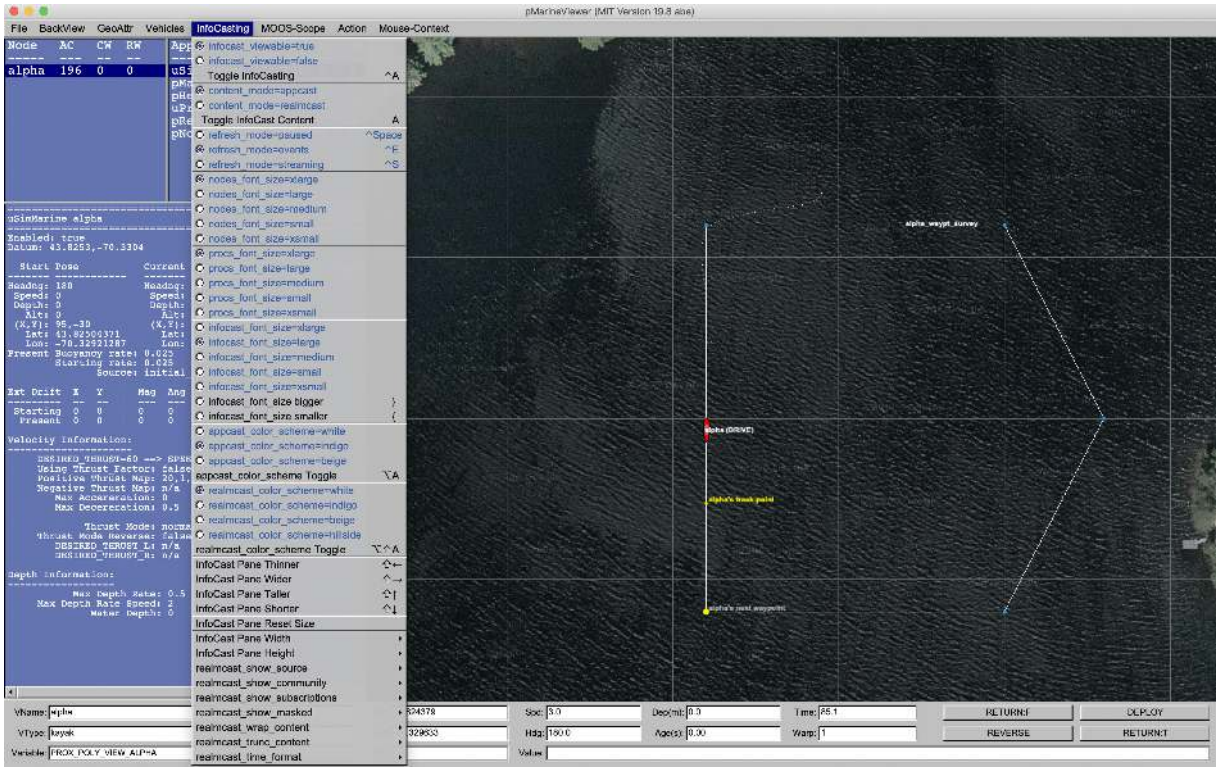

Figure 145: **The InfoCast menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering aspects of the appcast panels, and policy for soliciting appcasts from known vehicles and applications.

46.10.2 Adjusting the InfoCast Viewing Panes Height and Width

The next set of menu items allow the relative size of the infocasting panes to be adjusted. The width of the three panes may be increased or decreased with the shift-left and shift-right arrow keys, and the height of the lower infocasting window relative to the two upper windows may be adjusted with the shift-up and shift-down arrow keys.

The infocast pane extents may also be set to the user's liking in the mission configuration file with the parameters `infocast_width` and `infocast_height`. The allowable range of values for each may be seen by pulling down the "InfoCast Pane Width" and "InfoCast Pane Height" sub-menus of the InfoCasting pull-down menu.

46.10.3 Adjusting the InfoCast Refresh Mode

The infocast *refresh mode* refers to the policy of sending appcast and realmcast requests to known vehicles and applications. This is discussed more fully in appcast documentation, but summarized here. Appcasting and Realmcasting are implemented to be lazy with respect to generating appcast and infocast reports - they will not generate them unless asked. And even when asked, the request

comes with an expiration after which, if no new request has been received, the application returns to the lazy mode of producing no appcasts. So, for pMarineViewer to function as an appcast and realmcast viewer, under the hood it must be also generating appcast requests (APPCAST_REQ postings) and realmcast requests (REALMCAST_REQ postings) to the MOOSDB. The refresh mode refers to this under-the-hood policy.

In the *paused* refresh mode, pMarineViewer is not generating any infocast requests at all. This is not the default and typically not very helpful, but it may be useful when the viewer is situated in the field with only a low-bandwidth connection to remote vehicles. The refresh mode may be set to *paused* via the pull-down menu selection, with the CTRL+Spacebar hot key, or set in the mission configuration file with refresh_mode=paused.

In the *events* refresh mode, the default mode, pMarineViewer is generating appcast requests only to the selected vehicle and the selected MOOS application. Even this is only partly true. In fact it is generating another kind of appcast request to all vehicles and apps, but this kind of request comes with the caveat that an app should only generate an appcast report if a new run warning has been detected. Otherwise these apps remain lazy. In this mode you should expect to see regular appcasts received for the selected app, and updates for the other apps only if something worthy of a run warning has occurred. You can confirm this for yourself by looking at the counter reflecting the number of appcasts received for any application. This counter is under the AC column in the upper panes. The refresh mode may be set to *events* via the pull-down menu selection, with the CTRL+'e' hot key, or set in the mission configuration file with refresh_mode=events. The latter would be redundant however since this is the default mode.

In the *streaming* refresh mode, pMarineViewer is generating appcast requests to all vehicles and all apps to generate appcasts all the time. This mode is a bandwidth hog, but it may be useful at times, especially to debug why a particular application is silent. If it is not generating and appcast in this mode, then something may indeed be wrong. The refresh mode may be set to *streaming* via the pull-down menu selection, with the CTRL+'s' hot key, or set in the mission configuration file with refresh_mode=streaming. The *streaming* mode is not relevant to realmcast generation. Realmcast generation may be only in either the paused for events mode.

46.10.4 Adjusting the InfoCast Fonts

The font size of the text in the infocasting panes may be adjusted. There are three panes:

- *Nodes Pane*: The upper left pane shows the list of nodes (typically synonymous with vehicles), presently known to the viewer.
- *Procs Pane*: The upper right pane show the list of apps, for the chosen node, presently known to the viewer.
- *InfoCast Pane*: The bottom pane shows the contents of the presently selected appcast or realmcast report.

For each pane the possible font settings are xlarge, large, medium, small, and xsmall. The default for the upper panes is large, and the default for the infocast pane is medium. Font sizes may be changed via the pull-down menu or set to the user's liking in the mission configuration file with nodes_font_size, procs_font_size, and infocast_font_size parameters.

46.10.5 AppCasting Versus RealmCasting

Although both types of infocast content, appcast and realmcast reports, can be viewed from other client apps like uMACView and uMAC, they were originally and primarily designed to be accessed from within pMarineViewer. It is worth discussing here their similarities and differences. In distributed multi-vehicle simulations and in-field operations, the content available through infocasting is priceless for development and safe field operation. Much care was taken to allow the user access to an enormous variety of information across all vehicles, while ensuring that the information flow is restricted to be only what the user is presently monitoring through the current infocast pane. In both modes, if bandwidth is an issue, information flow can be halted immediately by hitting `ctrl-spacebar`.

With appcasting, the basic idea is shown in Figure 146. The pMarineViewer user selects a node (either the shoreside or one of the simulated or deployed vehicles), and a process/app running on that node. This information is contained in an APPCAST_REQ message published by pMarineViewer which is shared out to the particular node and app. If this app is appcast enabled (most are), it will respond with an APPCAST message. This single string message will be expanded to a multi-line report displayed in the infocast pane when received by pMarineViewer.

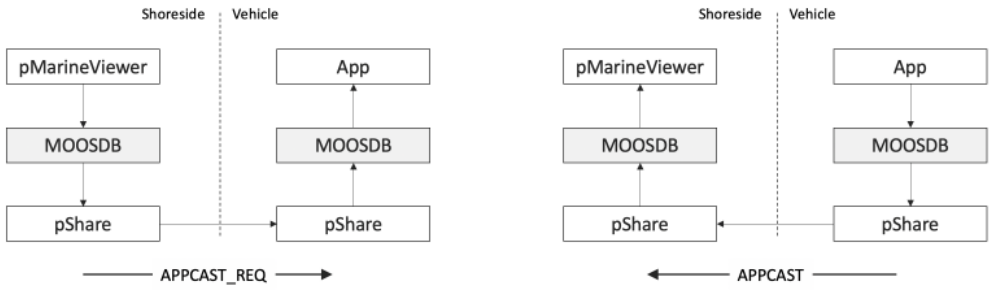

Figure 146: **Appcasting Information Flow:** An appcast viewing client, like pMarineViewer, uMACView, or uMAC will generate an appcast request to a targeted remote application. If the remote application is appcast enabled, it will respond with an appcast report for viewing in the appcast viewing client.

Realmcasting works a bit differently. It does not involve direct communication with remote apps as with appcasting. Instead, realmcasting is supported by a single instance of another app, pRealm. This app is run on each node, including the shoreside. This app is more fully described in [**?**]. It utilizes a special variable published by the local MOOSDB, DB_RWSUMMARY which informs pRealm what variables are subscribed for and published for each app connected to the MOOSDB. With this information, pRealm also subscribes for all variables and is able to form a realmcast report. Typically the form of this report is a summary, for a specified app, of all variables involved in subscriptions or publications. As with appcasting, the report generation is on-demand, requiring a realmcast request followed by a realmcast reply, as shown in Figure 147.

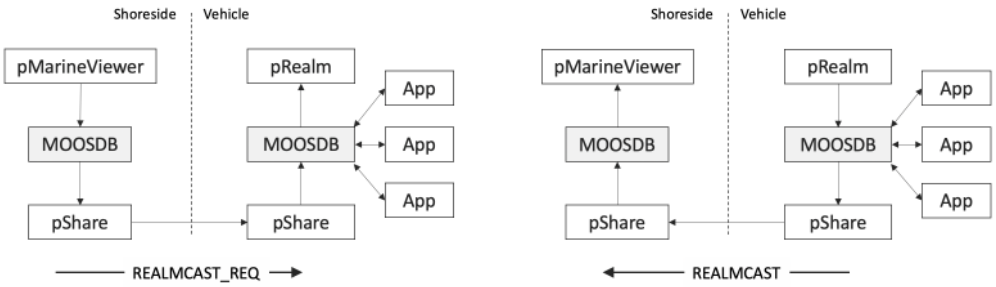

Figure 147: **Realmcasting Information Flow:** An realmcast viewing client, like pMarineViewer or uMACView will generate an realmcast request to a targeted node running an instance of pRealm. The realmcast request will specify a particular app and the realmcast response will report the status of variables involved in publications or subscriptions for that app.

The realmcast report is split into two parts. The top part shows all variables subscribed for by the given application, and the bottom part similarly shows all variables published by the application. In this way, a realmcast report is another way to glean the pub/sub interface for any application. A typical report is shown in Figure 241 below.

```
====================================
pContactMgrV20             gilda (140)
====================================
Subscriptions
====================================
Variable          Source          Time    Comm       Value
----------------  --------------  ------  ---------  ------------------------------------------------------------------------------------------
APPCAST_REQ       uMAC_220        241.52  shoreside  node=all,app=all,duration=3.0,key=uMAC_220:app,thresh=run_warning
BCM_ALERT_REQUEST pHelmIvP        0.28    gilda      id=avdcol_, on_flag=CONTACT_INFO=name=$[VNAME] # contact=$[VNAME],alert_range=80, cpa_rang
                                                     e=85,match_type=mokai
NAV_HEADING       uSimMarine      242.21  gilda      180
NAV_SPEED         uSimMarine      242.21  gilda      0
NAV_X             uSimMarine      242.21  gilda      180
NAV_Y             uSimMarine      242.21  gilda      0
NODE_REPORT       uFldNodeComms   241.73  shoreside  NAME=henry,X=0,Y=0,SPD=0,HDG=180,TYPE=kayak,MODE=PARK,ALLSTOP=ManualOverride,INDEX=479,TIM
                                                     E=3218502913.32,LENGTH=4

====================================
Publications
====================================
Variable                 Source          Time    Comm   Value
-----------------------  --------------  ------  -----  ------------------------------------------------------------------------------------------
ALERT_VERBOSE            pContactMgrV20  2.91    gilda  new_contact=henry
APPCAST                  uProcessWatch   7.68    gilda  formatted report
CONTACTS_LIST            pContactMgrV20  2.91    gilda  henry
CONTACTS_RECAP           pContactMgrV20  242.05  gilda  vname=henry,range=180.00,age=0.69
CONTACT_CLOSEST          pContactMgrV20  2.91    gilda  henry
CONTACT_CLOSEST_TIME     pContactMgrV20  2.91    gilda  3218502674.874138
PCONTACTMGRV20_ITER_GAP  pContactMgrV20  242.04  gilda  1.004112
PCONTACTMGRV20_ITER_LEN  pContactMgrV20  242.05  gilda  0.000828
PCONTACTMGRV20_PID       pContactMgrV20  -0.12   gilda  42130
PCONTACTMGRV20_STATUS    pContactMgrV20  241.04  gilda  AppErrorFlag=false,Uptime=241.812,cpuload=0.2981,memory_kb=3224,memory_max_kb=3224,
```

Figure 148: **Example RealmCast Report:** A realmcast message is expanded into a multi-line report, consisting of a report on subscribed variables and published variables for a given application. In this case the report is for the pContactMgrV20 application on vehicle `gilda`.

46.10.6 Adjusting the RealmCast Content

The content of the report can be adjusted by the client, e.g., pMarineViewer user, to help visualize the important information. There are seven options:

- Source: The Source column of the report may be suppressed.
- Community: The Community column of the report may be suppressed.
- UTC Time: The time format may be shown in absolute UTC time, or relative to the start of the local MOOSDB.

- Subscriptions: The subscriptions portion of the report may be suppressed.
- Mask: In the subscriptions portion, virgin variables may be suppressed.
- Wrap: Long string content may be wrapped over several lines, e.g., as in Figure 241.
- Truncate: Long string content may be truncated.

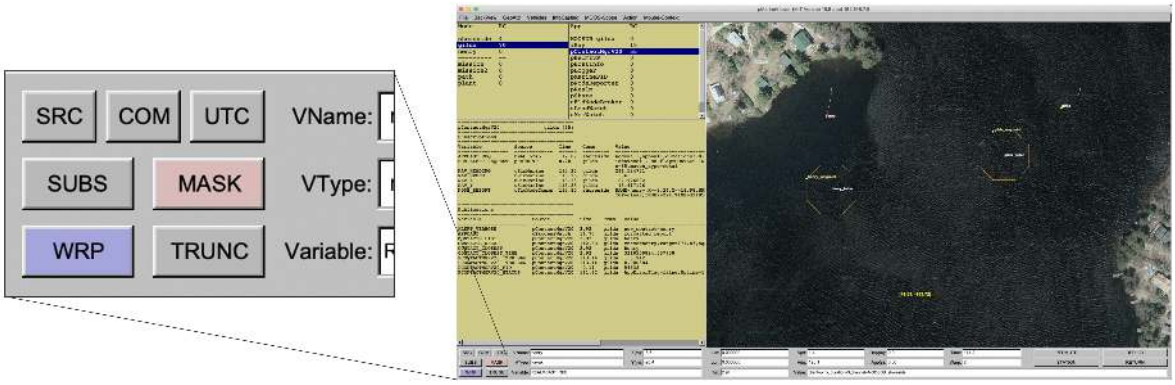

Figure 149: **Adjusting RealmCast Buttons:** Realmcast content may be adjusted through the seven buttons on the lower left corner of pMarineViewer. These buttons are only present when realmcasting is enabled.

The start-up value of these settings may also be set to the user's liking in the pMarineViewer configuration block:

```
realmcast_show_source        = true/false   // Default is true
realmcast_show_community     = true/false   // Default is true
realmcast_show_subscriptions = true/false   // Default is true
realmcast_show_masked        = true/false   // Default is true
realmcast_wrap_content       = true/false   // Default is false
realmcast_trunc_content      = true/false   // Default is false
realmcast_time_format_utc    = true/false   // Default is false
```

46.10.7 Additional RealmCast Capability: Watch Clusters

The normal realmcast reports allow the user to peer into any vehicle and any app to examine the state of variables involved in the subscriptions and publications for that app. However, in some multi-vehicle missions, it may be very useful to quickly see the value of certain variables for all vehicles *simultaneously*. For this purpose, pMarineViewer may be optionally configured with one or more *watch clusters*.

A watch cluster is a group of variables, with an associated grouping key, to allow simultaneous monitoring of these variable over all vehicles in the realmcast pane. This configuration is done on the client (pMarineViewer) side, and will result in a modified posting of REALMCAST_REQ. The pRealm applications knows how to receive these requests, and will generate a posting to the WATCHCAST variable, one for each cluster variable, containing the most recent information for that variable.

We will explain by way of an example. Suppose we have a mission with six vehicles, `abe`, `ben`, `cal`, `deb`, `eve`, and `fin`. And we would like to monitor the local MOOS variables `DEPLOY`, `RETURN`, `STATION_KEEP`, and `COVER`. These four variables constitute our watch cluster, and we pick a keyname, say `mission_top`. The watch cluster would be configured with the following line in the `pMarineViewer` configuration block:

```
watch_cluster = key=mission_top, vars=DEPLOY:RETURN:STATION_KEEP:COVER
```

The watch cluster key, `mission_top`, should then appear in the Nodes pane when launched, as shown in Figure 150

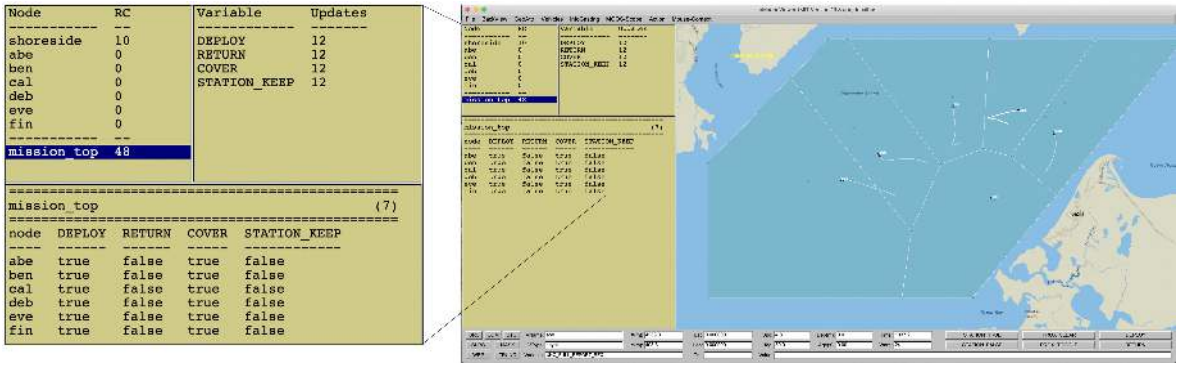

Figure 150: **RealmCast Watch Cluster Example:** A watch cluster has been configured with four variables, with the cluster name of `mission_top`. When the user selects `mission_top` in the nodes pane, the values of the four watched variables, for the six vehicles in the mission, are shown in a table in the bottom realmcast pane. This table is only refreshed when one or more of the variables has been changed on one or more of the vehicles.

The table in Figure 150 is quite concise, providing a substantial amount of information. However, it comes at the expense of leaving out certain information for each cell, such as the source, time, and community associated with the posting in that cell. For each variable in the cluster, the user has the option of finding out more. Notice the four cluster variables are listed in the upper right pane. By clicking on any one of them, the realmcast pane switches from the collective table format of Figure 150 to the single variable table, with examples shown in Figure 151 below.

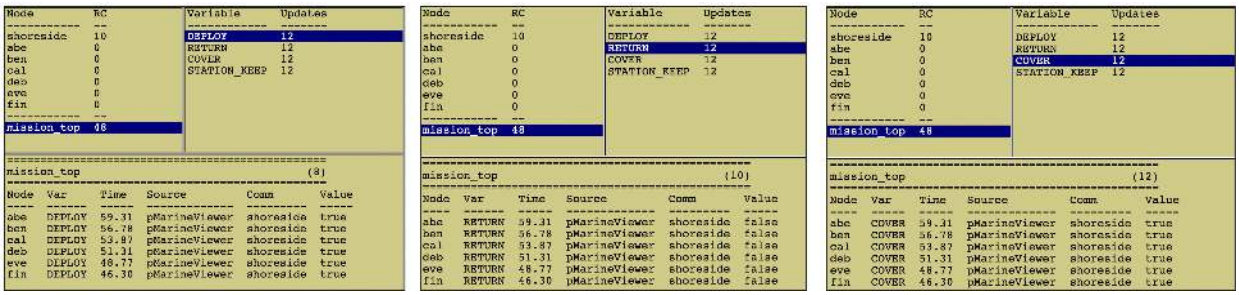

Figure 151: **Examining a Single Variable in a Watch Cluster:** When a single variable is selected in the watch cluster, the realmcast pane reports only on this variable, with one row for each vehicle. In this way, the user may see more information about the current posted value, such as the publisher (source), time posted, and the community of the app that posted the variable. When the user clicks again on the cluster keyname (in this case `mission_top`), the realmcast pane reverts to reporting on all variables in the cluster as in Figure 150.

46.10.8 Adjusting the AppCast and RealmCast Color Scheme

The infocast color schemes are by default different when the infocasting panes are in appcasting or realmcasting modes. There are a few different color schemes available to select from. The default appcast color scheme is `"indigo"`, reflected for example in Figures 130 or 145. The default realmcast color scheme is `"hillside"`, reflected for example in Figures 149 or 150. The color scheme may be changed by selecting a different option from the `InfoCasting` pull-down menu, or toggling through options with the hot key `Alt-'a'` for appcasting or `Alt-'A'` for realmcasting. Alternatively they may be configured in the `pMarineViewer` configuration block to have a different color scheme at launch time:

```
appcast_color_scheme   = indigo/white/beige           // Default is indigo
realmcast_color_scheme = indigo/white/beige/hillside  // Default is hillside
```

46.11 The MOOS-Scope Pull-Down Menu

The MOOS-Scope pull-down menu allows the user to configure `pMarineViewer` to scope on one or more variables in the `MOOSDB`. The viewer allows visual scoping on only a single variable at a time, but the user can select different variables via the pull-down menu, or toggle between the current and previous variable with the `'/'` key, or cycle between all registered variables with the `CTRL+'/'` key. The scope fields are on the bottom of the viewer as shown in Figure 152.

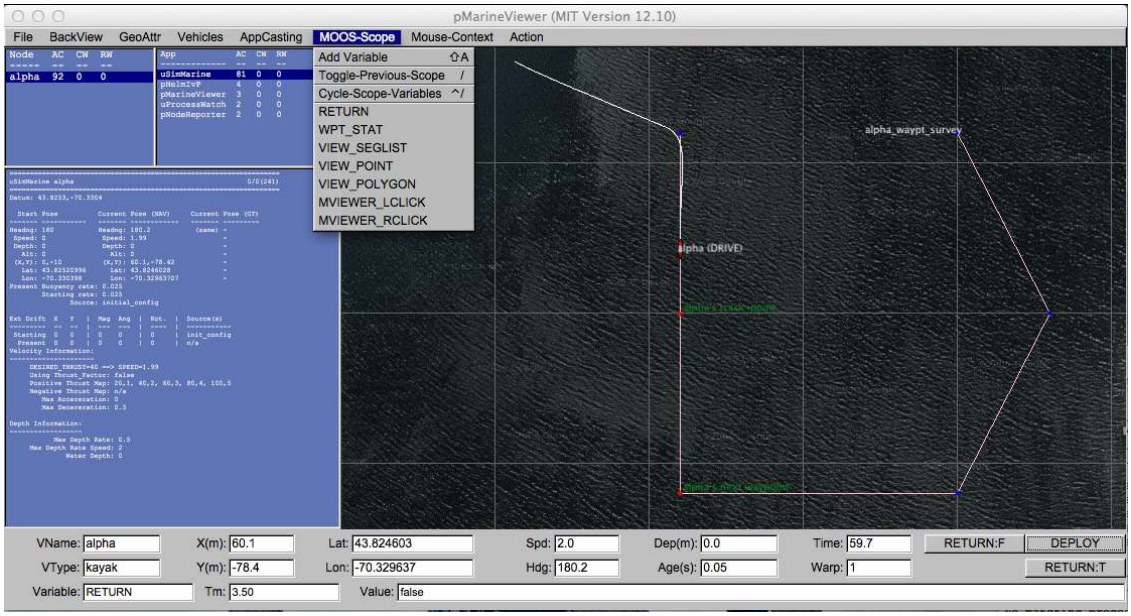

Figure 152: **The Scope Menu:** This pull-down menu allows the user to adjust which pre-configured MOOS variable is to be scoped, or to add a new variable to the scope list.

The three fields show (a) the variable name, (b) the last time is was updated, and (c) the current value of the variable. Configuration of the menu is done in the MOOS configuration block with entries similar to the following (which correlate to the particular items in the pull-down menu in Figure 152):

```
scope = RETURN, WPT_STAT, VIEW_SEGLIST, VIEW_POINT, VIEW_POLYGON
scope = MVIEWER_LCLICK, MVIEWER_RCLICK
```

The keyword `scope` is not case sensitive, but the MOOS variables are. If no entries are provided in the MOOS configuration block, the pull-down menu contains a single item, the `"Add Variable"` item. By selecting this, the user will be prompted to add a new MOOS variable to the scope list. This variable will then immediately become the actively scoped variable, and is added to the pull-down menu.

46.12 The Exclusion Filter

A new feature added after Release 19.8.2: In situations where incoming information from a particular vehicle, or set of vehicles, is to be ignored, an *exclusion filter* may be used. `pMarineViewer` contains a filter that may be configured by specifying one or more vehicle names to ignore:

```
ignore_name = abe
ignore_name = abe, ben, cal
```

It may also be configured to consume information only by vehicles explicitly named:

```
match_name = ben, deb
match_name = abe
```

For an incoming posting to survive the exclusion filter it must (a) match any one of the list of match names, if any are provided, *and*, not match any one of the ignore names, if any are provided. Of course if no match_name or ignore_name parameters are provided, then nothing is filtered out.

The filter is applied to incoming node reports (NODE_REPORT), and incoming geometric messages such as VIEW_POINT or VIEW_POLYGON.

Note: Certain geometric objects such as VIEW_COMMS_PULSE, that are published by apps in the shoreside community, will not be filtered. Filtering is carried out by using the MOOS host community information of the incoming MOOS mail, which typically matches the vehicle name. Information such as VIEW_COMSS_PULSE, are generated by a shoreside MOOS app, such as uFldNodeComms in this case. As such, it will not be filtered.

46.13 The Action Pull-Down Menu

The Action pull-down menu allows the user to invoke pre-define pokes to the MOOSDB (the MOOSDB to which the pMarineViewer is connected). While hooks for a limited number of pokes are available by configuring on-screen buttons (Section 45.2.3), the number of buttons is limited to four. The "Action" pull-down menu allows for as many entries as will reasonably be shown on the screen. Each action, or poke, is given by a variable-value pair, and an optional grouping key. Configuration is done in the MOOS configuration block with entries of the following form:

```
action = menu_key=<key> # <MOOSVar>=<value> # <MOOSVar>=<value> # ...
```

If no such entries are provided, this pull-down menu will not appear. The fields to the right of the action are separated by the '#' character for convenience to allow several entries on one line. If one wants to use the '#' character in one of the variable values, putting double-quotes around the value will suffice to treat the '#' character as part of the value and not the separator. If the pair has the key word menu_key on the left, the value on the right is a key associated with all variable-value pairs on the line. When a menu selection is chosen that contains a key, then all variable-value pairs with that key are posted to the MOOSDB. The following configuration will result in the pull-down menu depicted in Figure 153.

```
action  = menu_key=deploy # DEPLOY = true # RETURN = false
action+ = menu_key=deploy # MOOS_MANUAL_OVERIDE=false
action  = RETURN=true
```

The action+ variant hints to the viewer that a line should be rendered in the pull-down menu separating it from following items.

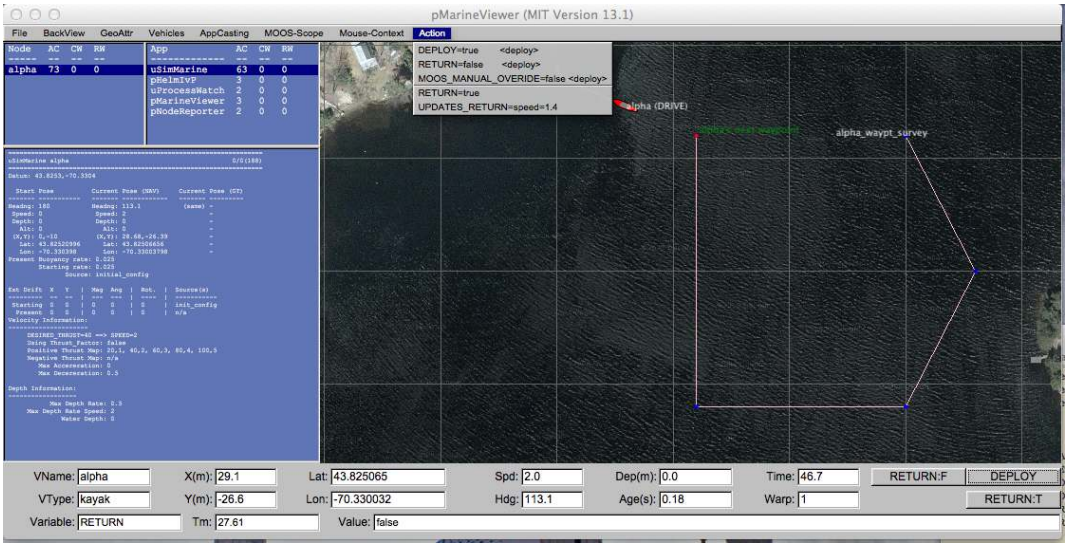

Figure 153: **The Action menu:** The variable value pairs on each menu item may be selected for poking or writing the MOOSDB. The three variable-value pairs above the menu divider will be poked in unison when any of the three are chosen, because they were configured with the same key, `<deploy>`, shown to the right on each item.

The variable-value pair being poked on an action selection will determine the variable type by the following rule of thumb. If the value is non-numerical, e.g., `true`, `one`, it is poked as a string. If it is numerical it is poked as a double value. If one really wants to poke a string of a numerical nature, the addition of quotes around the value will suffice to ensure it will be poked as a string. For example:

```
action = Vehicle=Nomar # ID="7"
```

As with any other publication to the MOOSDB, if a variable has been previously posted with one type, subsequent posts of a different type will be ignored.

46.14 The Mouse-Context Pull-Down Menu

The Mouse-Context pull-down menu is an optional menu - it will not appear unless it is configured for use. It is used for changing the context of left and right mouse clicks on the operation area.

46.14.1 Generic Poking of the MOOSDB with the Operation Area Position

When the user clicks the left or right mouse in the geo portion of the pMarineViewer window, the variables MVIEWER_LCLICK and MVIEWER_RCLICK are published respectively with the operation area location of the mouse click, and the name of the active vehicle. A left mouse click may result in a publication similar to:

```
MVIEWER_LCLICK = x=19.0,y=57.0,lat=43.8248027,lon=-70.3290334,vname=henry,counter=1
```

A counter is maintained by pMarineViewer and is incremented and included on each post. The above style posting presents a generic way to convey to other MOOS applications an operation

area position. In this case the other MOOS applications need to conform to this generic output. But, with a bit of further configuration, a similar *custom* post to the MOOSDB is possible to shift the burden of conformity away from the other MOOS applications where typically a user does not have the ability to change the interface.

46.14.2 Custom Poking of the MOOSDB with the Operation Area Position

Custom configuration of mouse clicks is possible by (a) allowing the MOOS variable and value to be defined by the user, and (b) exposing a few macros in the custom specification to embed operation area information. Configuration is done in the MOOS configuration block with entries of the following form:

```
left_context[<key>]  = <var-data-pair>
right_context[<key>] = <var-data-pair>
```

The left_context and right_context keywords are case insensitive. If no entries are provided, this pull-down menu will not appear. The <key> component is optional and allows for groups of variable-data pairs with the same key to be posted together with the same mouse click. This is the selectable *context* in the Mouse-Context pull-down menu. If the <key> is empty, the defined posting will be made on all mouse clicks regardless of the grouping, as is the case with MVIEWER_LCLICK and MVIEWER_RCLICK.

Macros may be embedded in the string to allow the string to contain information on where the user clicked in the operation area. These patterns are: $(XPOS) and $(YPOS) for the local x and y position respectively, and $(LAT), and $(LON) for the latitude and longitude positions. The pattern $(IX) will expand to an index (beginning with zero by default) that is incremented each time a click/poke is made. This index can be configured to start with any desired index with the lclick_ix_start and rclick_ix_start configuration parameters for the left and right mouse clicks respectively. The following configuration will result in the pull-down menu depicted in Figure 154.

```
left_context[surface_point] = SPOINT = $[XPOS],$[YPOS]
left_context[surface_point] = COME_TO_SURFACE = true
left_context[return_point]  = RETURN_POINT = point=$[XPOS],$[YPOS], vname=$[VNAME]
left_context[return_point]  = RETURN_HOME = true
left_context[return_point]  = RETURN_HOME_INDEX = $[IX]
right_context[loiter_point] = LOITER_POINT = lat=$[LAT], lon=$[LON]
right_context[loiter_point] = LOITER_MODE = true
```

Note in the figure that the first menu option is "no-action" which shuts off all MOOS pokes associated with any defined groups (keys). In this mode, the MVIEWER_LCLICK and MVIEWER_RCLICK pokes will still be made, along with any other poke configured without a <key>.

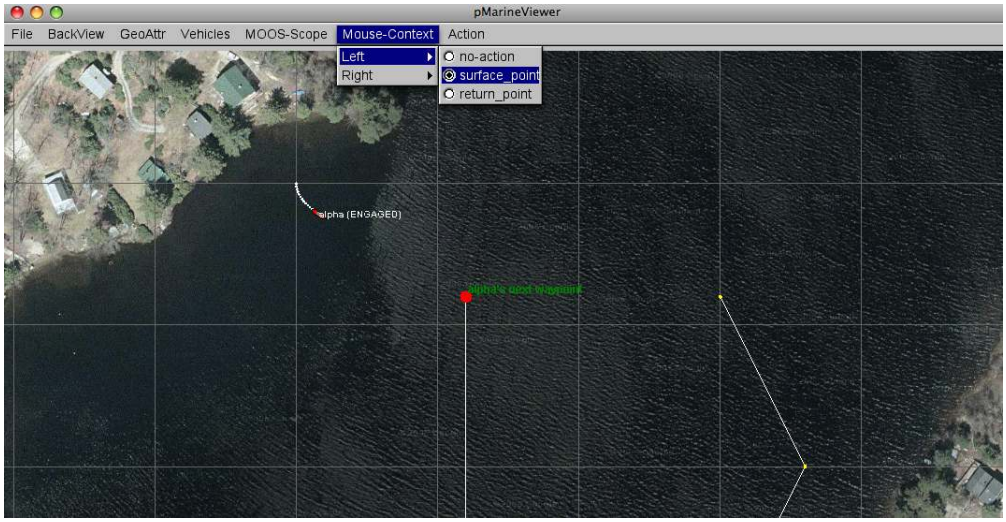

Figure 154: **The Mouse-Context menu:** Keywords selected from this menu will determine which groups of MOOS variables will be poked to the MOOSDB on left or mouse clicks. The variable values may have information embedded indicating the position of the mouse in the operation area at the time of the click.

The full set of macros are:

- `$[XPOS]`: The local x coordinate of the mouse click, to 0.1 meters.
- `$[X]`: The local x coordinate of the mouse click, to 1 meter.
- `$[YPOS]`: The local y coordinate of the mouse click, to 0.1 meters.
- `$[Y]`: The local y coordinate of the mouse click, to 1 meter.
- `$[IX]`: An integer that increments each each left mouse click. It starts at zero, but can be made to start at N with the config parameter `lclick_ix_start`.
- `$[BIX]`: An integer that increments each each right mouse click. It starts at zero, but can be made to start at N with the config parameter `rclick_ix_start`.
- `$[UTC]`: The current time in UTC seconds, in millisecond precision.
- `$[LAT]`: The local latitude coordinate of the mouse click.
- `$[LON]`: The local longitude coordinate of the mouse click.
- `$[VNAME]`: The active vehicle name
- `$[VNAME_CLOSEST]`: The name of the vehicle closest to the mouse click
- `$[UP_VNAME]`: The active vehicle name in upper case
- `$[UP_VNAME_CLOSEST]`: The name of the vehicle, in upper case, closest to the mouse click
- `$[HDG]`: The heading from the active vehicle to the x-y position of the mouse click.

Note macros above are of the form `$[MACRO]` using square brackets. Macros specified with parentheses, e.g., `$(MACRO)`, are also supported. The preferred form is the former, since parentheses macros are used by the `splug` utility for expanding mission files. Using square macros in `pMarineViewer` can help ward off confusion.

Certain macros are also available for expansion in the MOOS variable name component of a posting. For example, consider the following configuration entry:

```
left_context[return_point]  = RETURN_POINT_$[UP_VNAME_CLOSEST] = point=$(XPOS),$(YPOS)
```

the macro `$[UP_VNAME_CLOSEST]` will expand to the upper case name of the vehicle closest to the mouse click, and may result in a posting like:

```
RETURN_POINT_HENRY = point=87.2,99.8
```

The list of macros available for expansion in the MOOS variable name are:

- `$[VNAME]`: The active vehicle name
- `$[VNAME_CLOSEST]`: The name of the vehicle closest to the mouse click
- `$[UP_VNAME]`: The active vehicle name in upper case
- `$[UP_VNAME_CLOSEST]`: The name of the vehicle, in upper case, closest to the mouse click

46.15 Configuring and Using the Commander Pop-Up Window

The *commander pop-up* window is meant to provide an extendible command and control interface for missions requiring more than a few on-screen buttons. It needs to be configured and thought through prior to mission launch, to reflect whatever the mission operator may need to do during the mission. The window conveniently pops up and is hidden by toggling the space bar. An example is shown below.

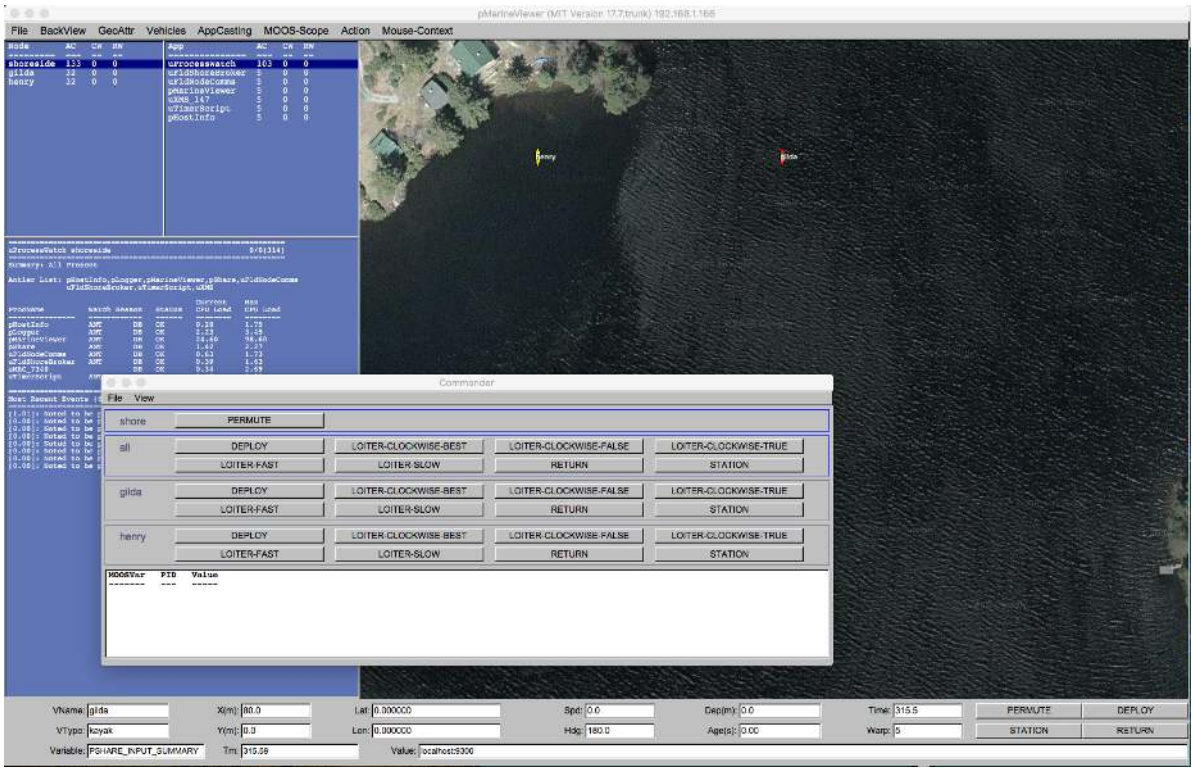

Figure 155: The commander pop-up window is toggled open/closed with the space-bar key and presents the user with a set of user-configured commands to either the all vehicles, individual vehicles, or the the pMarineViewer shoreside community itself.

Below we discuss how to work with the window and its content, how to configure the pop-up window, and the related assumptions of configuration in `uFldShoreBroker` to ensure proper command out to the vehicles.

46.15.1 Commander Pop-Up Window Actions and Content

The commander pop-up window has typically four components as indicated in Figure 156. The first component contains a single pane of buttons with postings to the local shoreside MOOSDB, not intended to be passed on to any vehicles. The second component contains a single pane of buttons for postings intended to be sent to all vehicles. The third component holds a dedicated pane for each known vehicle, with buttons intended for direct communication to the named vehicle. The fourth component contains the *command history*, a list of the actual postings made from recent button clicks, with the most recent postings at top.

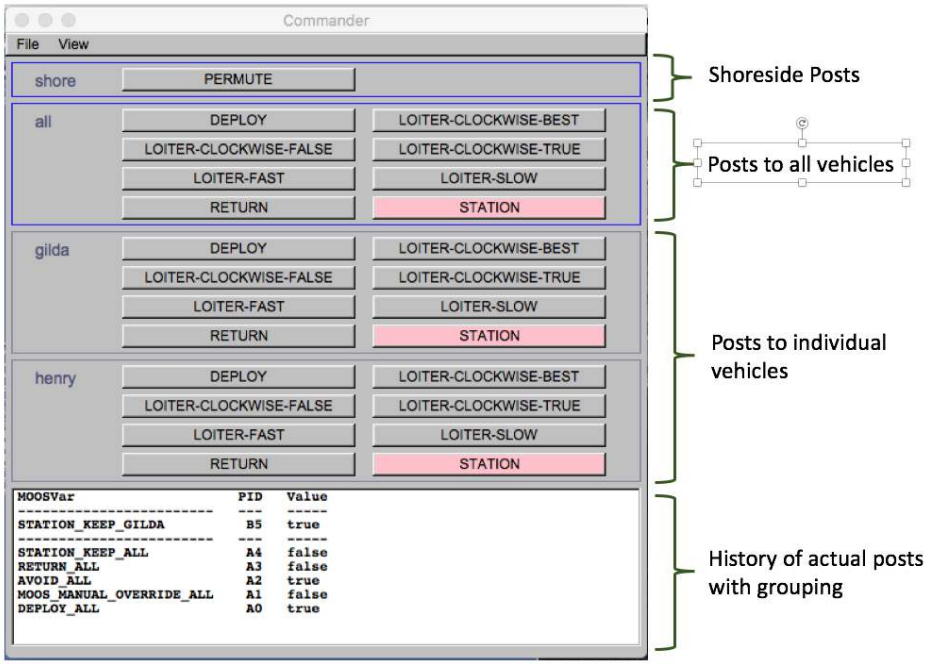

Figure 156: The commander pop-up window is contains four parts as shown. The button grouping is done automatically depending on the user configuration. Each button may be configured to make one or more posts to the MOOSDB.

The command history also indicates the clustering of posts associated with recent button clicks. The Post-ID, is shown in the middle column. All postings from the same button click, begin with the same letter, allowing the user to discern the group of postings resulting from a single button click. The user may also may a hypothetical post by clicking any button with the shift-key held down. In this case, no post will be made to the MOOSDB, but the actions that would have been posted are added to the command history, separated by a dashed line as in Figure 156. A subsequent actual button click would erase he hypothetical lines from the command history. Finally, if screen real estate is an issue, the command history pane may be hidden by toggling with the `'p'` character.

The position and sizing of the command-popup window may be adjusted upon opening, with the

buttons retaining their size, but the button positions automatically adjusted to best fit the size of the window. As the user adjusts wider, the number of buttons in each row will grow, and the height will be as high as needed to fit the content. As the user adjusts taller, the command history pane is grown to show more history. The whole command pop-up window may be hidden and re-shown by toggling the spacebar-key. When doing so, note that not only the shape, but also the position w.r.t. the pMarineViewer window is also retained.

46.15.2 Commander Pop-Up Configuration

The set of possible command buttons and their corresponding posts to the MOOSDB may be referred to as a command portfolio, or *commandfolio* for short. This configuration is achieved with a set of lines of the following format:

```
cmd = label=<val>, var<val>, sval=<val>, dval=<val>, receivers=<val>, color=<val>
```

The label field: The button label is the text that will appear on the viewer button. It is also used as a unique identifier to associate several configuration lines together to make several posts with a single button.

The `var` field:
The `var` field names a single MOOS variable to be used in a post when this button is clicked. As such it should follow MOOS conventions and be upper case, no white space, not begin with a number, no use of special characters like `'@', '?', '('` etc., and use the underscore character `'_'` to separate components. These are only conventions and not enforced.

The `sval` field:
If a posting is to be made with a string value, the `sval` field should be used to specify this value.

The `dval` field: If a posting is to be made with a numerical (double) value, the `dval` field should be used to specify this value.

The `receivers` field: This field specifies a colon-separated list of vehicles, e.g., `henry:gilda` to be receivers of the posts associated with this button. If, for example, a vehicle named `henry` is named, with the MOOS variable `DEPLOY`, a button click will result in a posting to the variable `DEPLOY_HENRY`. Two special names are reserved, `"all"` and `"shoreside"`. If `"all"` is used, `DEPLOY_ALL` will be posted, *not* `DEPLOY_GILDA` and `DEPLOY_HENRY` for example. If `"shoreside"` is used, simply `DEPLOY` will be posted.

The `colors` field:
If the user desires a certain command button to visually stand out, a color may be specified. For example, in Figure 156, the button to order a vehicle to station keep is highlighted in red, since this is often regarded as the go-to action when or if a surface vehicle encounters intervention from an operator.

46.15.3 Commander Pop-Up Example Configuration from m2_berta Mission

Below is an example commander pop-up window configuration from the `m2_berta` mission. It will result in the commander window shown in Figure 156. Note the macro `$(VNAMES)` in many of the configuration lines. This macro is filled in at launch time by the colon-separated list of all vehicles being launched. See the launch script in the `m2_berta` mission for an example.

```
cmd = label=DEPLOY, var=DEPLOY,  sval=true,  receivers=all:$(VNAMES)
cmd = label=DEPLOY, var=MOOS_MANUAL_OVERRIDE,  sval=false,  receivers=all:$(VNAMES)
cmd = label=DEPLOY, var=AVOID,  sval=true,  receivers=all:$(VNAMES)
cmd = label=DEPLOY, var=RETURN,  sval=false,  receivers=all:$(VNAMES)
cmd = label=DEPLOY, var=STATION_KEEP,  sval=false,  receivers=all:$(VNAMES)

cmd = label=RETURN, var=RETURN, sval=true, receivers=all:$(VNAMES)
cmd = label=RETURN, var=STATION_KEEP, sval=false, receivers=all:$(VNAMES)

cmd = label=PERMUTE, var=UTS_FORWARD,  dval=0,  receivers=shore

cmd = label=STATION, var=STATION_KEEP,  sval=true,  receivers=all:$(VNAMES), color=pink

cmd = label=LOITER-FAST, var=UP_LOITER,  sval=speed=2.8,  receivers=all:$(VNAMES)
cmd = label=LOITER-SLOW, var=UP_LOITER,  sval=speed=1.4,  receivers=all:$(VNAMES)

cmd = label=LOITER-CLOCKWISE-TRUE,  var=UP_LOITER, sval=clockwise=true,  receivers=all:$(VNAMES)
cmd = label=LOITER-CLOCKWISE-FALSE, var=UP_LOITER, sval=clockwise=false, receivers=all:$(VNAMES)
cmd = label=LOITER-CLOCKWISE-BEST,  var=UP_LOITER, sval=clockwise=false, receivers=all:$(VNAMES)
```

46.15.4 Commander Pop-Up Coordination with pShare and uFldShoreBroker

Neither the commander popup window, nor pMarineViewer for that matter, are able to communicate with vehicles directly. The convention here is that if posting such as `DEPLOY=true` is desired to be sent to say vehicle `"henry"`, then a post of `DEPLOY_HENRY=true` is made instead. And if all vehicles are to be deployed with a single button click, then a posting of `DEPLOY_ALL=true` is made.
The assumption is made that an inter-MOOSDB share arrangement of some form, typically with pShare, is made from the shoreside community to each vehicle. This can be made directly with pShare with a pair of lines as the following:

```
Output = src_name=DEPLOY_ALL, dest_name=DEPLOY, route=192.168.1.12:9300
Output = src_name=DEPLOY_GUS, dest_name=DEPLOY, route=192.168.1.12:9300
```

The above pair of lines would be needed for *each* vehicle command, for *each* vehicle. This is accomplished much more conveniently if the uField Toolbox tool uFldShoreBroker is used to automatically coordinate known vehicles, and commands with pShare. The above pair of lines is instead:

```
qbridge = DEPLOY
```

And for multiple commands, with say five vehicles for example:

```
qbridge = DEPLOY, RETURN, STATION_KEEP, UP_LOITER, MOOS_MANUAL_OVERRIDE
```

This single line using uFldShoreBroker would require 30 separate pShare lines.

46.16 Configuration Parameters for pMarineViewer

The blue items in pull-down menus are also available as mission file configuration parameters. The configuration parameter is identical to the pull-down menu text. For example in the BackView menu shown in Figure 134, the menu item full_screen=true may also be set in the pMarineViewer configuration block verbatim with full_screen=true.

46.16.1 Configuration Parameters for the BackView Menu

The parameters in Listing 48 relate to the BackView menu described more fully in Section 45.3. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, the text in the pull-down menu is identical to a similar entry in the configuration file.

Listing 46.48: Configuration Parameters for pMarineViewer *BackView Menu.*

back_shade:	Shade of gray background when no image is used. Legal value range: [0, 1]. Zero is black, one is white.
full_screen:	If true, viewer is in full screen mode (no appcasts, no fields rendered at the bottom). Legal values: true, false. Section 45.3.4.
hash_delta:	Sets the hash line spacing. Legal values: 50, 100, 200, 500, 1000. The default is 100. Section 45.3.3.
hash_shade:	Shade of hash marks. Legal value range: [0, 1]. Zero is black, one is white. Section 45.3.3.
hash_viewable:	If true, hash lines are rendered over the op area. Legal values: true, false. The default is false. Section 45.3.3.
log_the_image:	If true, a request is posted to pLogger to log a copy of the image and info file. Legal values: true, false. The default is false. Section 45.3.2.
tiff_file:	Filename of a tiff file background image. After Release 22.8.x, multiple lines of this parameter will load multiple tiff files. Section 45.3.2.
tiff_file_b:	Filename of another tiff file background image. Deprecated after Release 22.8.x: just use multiple tiff_file lines. Section 45.3.2.
tiff_type:	Parameter no longer supported, after Release 22.8.x
tiff_viewable:	Use the tiff background image if set to true. Otherwise a gray screen is used as a background. Legal values: true, false. The default it true. Section 45.3.2.
view_center:	Sets the center of the viewing image in x,y local coordinates. Legal values: (double,double). The default is (0,0).

46.16.2 Configuration Parameters for the GeoAttributes Menu

The parameters in Listing 49 relate to the GeoAttributes pull-down menu described more fully in Section 46.8. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, the text in the pull-down menu is identical to a similar entry in the configuration file.

Listing 46.49: Configuration Parameters for `pMarineViewer` *Geometry Menu.*

`circle_viewable_all`:	If false, circles are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.
`circle_viewable_labels`:	If false, circle labels are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.
`comms_pulse_viewable_all`:	If false, comms pulses are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.5.
`datum_viewable`:	If false, the datum is suppressed from rendering. Legal values: true, false. The default is true. Sections 45.3.2 and 46.8.
`datum_color`:	The color used for rendering the datum. Legal values: Any color listed in the Colors Appendix. The default is red. Sections 45.3.2 and 46.8.
`datum_size`:	The size of the point used to render the datum. Legal values: Integers in the range $[1, 10]$. The default is 2. Sections 45.3.2 and 46.8.
`drop_point_viewable_all`:	If false, drop points are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.7.
`drop_point_coords`:	Specifies whether the drop point labels are in earth or local coordinates. Legal values are: as-dropped, lat-lon, local-grid. The default is as-dropped. Section 46.8.7.
`drop_point_vertex_size`:	The size of the point used to render a drop point. Legal values: Integers in the range $[1, 10]$. The default is 2. Section 46.8.7.
`grid_viewable_all`:	If false, grids are suppressed from rendering. Legal values: true, false. The default is true.
`grid_viewable_labels`:	If false, grid labels are suppressed from rendering. Legal values: true, false. The default is true.
`grid_viewable_opaqueness`:	The degree to which grid renderings are opaque. Legal range: $[0, 1]$. The default is 0.3.
`marker`	A marker may be stated in the configuration file with the same format of the `VIEW_MARKER` message. Section 46.8.4.
`marker_scale`:	The scale applied to marker renderings. Legal range: $[0.1, 100]$. The default is 1.0. Section 46.8.4.
`marker_viewable_all`:	If false, markers are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.4.
`marker_edge_width`:	Markers are rendered with an outer black edge. The edge may be set thicker to aid in viewing. Legal values: Integer values in the range $[1, 10]$. The default is 1. Section 46.8.4.

marker_viewable_labels: If false, marker labels are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.4.

oparea_viewable_all: If false, oparea lines are suppressed from rendering. Legal values: true, false. The default is true.

oparea_viewable_labels: If false, oparea label is suppressed from rendering. Legal values: true, false. The default is true.

point_viewable_all: If false, points are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

point_viewable_labels: If false, point labels are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

polygon_viewable_all: If false, polygons are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

polygon_viewable_labels: If false, polygon labels are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

range_pulse_viewable_all: If false, range pulses are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.5.

seglist_viewable_all: If false, seglists are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

seglist_viewable_labels: If false, seglist labels are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

vector_viewable_all: If false, vectors are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

vector_viewable_labels: If false, vector labels are suppressed from rendering. Legal values: true, false. The default is true. Section 46.8.1.

46.16.3 Configuration Parameters for the Vehicles Menu

The parameters in Listing 50 relate to the Vehicles pull-down menu described more fully in Section 46.9. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, text in the pull-down menu is identical to a similar entry in the configuration file.

Listing 46.50: Configuration Parameters for pMarineViewer Vehicles Pull-Down Menu.

bearing_lines_viewable: If false, bearing lines will be suppressed from rendering. Legal values: true, false. The default is true.

center_view: Sets the pan position to be either directly above the active vehicle, or the average of all vehicles. Legal values: active, average. The default is neither, resulting in the pan position being set to either (0,0) or set via other configuration parameters. Section 46.9.5.

stale_remove_thresh: Number of seconds after a stale vehicle has been detetected before being removed. When time warp is one, vehicles are *not* automatically removed at all, and this number is meaningless. Legal values: Any non-negative number. The default is 30. Section 46.9.2.

stale_report_thresh:	Number of seconds after which a vehicle report will be considered stale. Legal values: Any non-negative number. The default is 60. Section 46.9.2.
trails_color:	The color of trail points rendered behind vehicles to indicate recent vehicle position history. Legal values: Any color listed in the Colors Appendix. The default is white. Section 46.9.6.
trails_connect_viewable:	If true the vehicle trail points are each connected by a line. Useful when node reports have large gaps in time. Legal values: true, false. The default is true. Section 46.9.6.
trails_length:	The number of points retained for the rendering of vehicle trails. Legal values: Integers in the range $[1, 100000]$. The default is 100. Section 46.9.6.
trails_point_size:	The size of the points rendering the vehicle trails. Legal values: Integers in the range $[1, 10]$. The default is 1. Section 46.9.6.
trails_viewable:	If false, vehicle trails are suppressed from rendering. Legal values: true, false. The default is true. Section 46.9.6.
vehicles_active_color:	The color of the active vehicle (the one who's data is being shown in the bottom data fields). Legal values: Any color listed in the Colors Appendix. The default is red. Section 46.9.4.
vehicles_inactive_color:	The color of inactive vehicles. Legal values: Any color listed in the Colors Appendix. The default is yellow. Section 46.9.4.
vehicles_shape_scale:	The scale factor applied to vehicle size rendering. Legal values in the range: $[0.1, 100]$. The default is 1.0. Section 46.9.3.
vehicles_name_mode:	Sets the mode for rendering the vehicle label. Legal values are: names, names+mode, names+shortmode, names+depth, off. The default is names+shortmode. Section 46.9.1.
vehicles_name_color:	Sets the color for rendering the vehicle label. Legal values are any color in the Colore Appendix. The default is white. Section 46.9.4.
vehicles_viewable:	If false, vehicles are suppressed from rendering. Legal values: true, false. The default is true. Section 46.9.

46.16.4 Configuration Parameters for the InfoCasting Menu

The parameters in Listing 51 relate to the InfoCasting pull-down menu described more fully in Section 46.10. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, text in the pull-down menu is identical to a similar entry in the configuration file.

Note: Starting with the first release after 2019's Release 19.8, this pMarineViewer was augmented to include realmcasting in addition to appcasting. The term *infocasting* is the general term refering to either appcasting or realmcasting. As such, some of the earlier configuration parameters, e.g., appcast_font_size, have been replace with a term like infocast_font_size. The older configuration parameters are deprecated, still supported forseeable releases.

Listing 46.51: Configuration Parameters for pMarineViewer AppCast Pull-Down Menu.

appcast_color_scheme: The color scheme used in all three appcasting panes, affecting background color and font color. Possible settings, `default`, `beige`, `indigo` or `white`. The default is `indigo`. Section 46.10.8.

content_mode: Sets the on startup content, which can be changed by the user at any time. Either `appcast` or `realmcast`. The default is `appcast`.

infocast_font_size: The font size uses in the *infocast* pane of the set of infocasting panes. Legal values: `xlarge`, `large`, `medium`, `small`, `xsmall`. The default is `medium`. Section 46.10.4.

infocast_height: The height of the appcasting bottom pane as a percentage of the total pMarineViewer window height. Legal values: [30, 35, 40, 45,..., 85, 90]. The default is 75. Section 46.10.2.

infocast_viewable: If `true`, the infocasting set of panes are rendered on the left side of the viewer. Legal values: `true` or `false`. The default is `true`. Section 46.10.1.

infocast_width: The width of the infocasting panes as a percentage of the total pMarineViewer window width. Legal values: [20, 25, 30, 35,..., 65, 70]. The default is 30. Section 46.10.2.

nodes_font_size: The font size used in the *nodes* pane of the set of infocasting panes. Possible settings, `xsmall`, `small`, `medium`, `large`, or `xlarge`. The default is `large`. Section 46.10.4.

procs_font_size: The font size used in the *procs* pane of the set of infocasting panes. Possible settings, `xsmall`, `small`, `medium`, `large` or `xlarge`. The default is `large`. Section 46.10.4.

realmcast_color_scheme: Either `indigo`, `beige`, `hillside`, `white`, or `default`. The default is `hillside`. Section 46.10.8.

realmcast_show_source: If `true`, the Source column is shown on realmcast output. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`. Section 46.10.6.

realmcast_show_community: If `true`, the Community column is shown on realmcast output. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`. Section 46.10.6.

realmcast_show_subscriptions: If `true`, the Subscriptions block is shown on realmcast output. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`. Section 46.10.6.

realmcast_show_masked: If `true`, certain variables are excluded in realmcast output, e.g., virgin variables. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`. Section 46.10.6.

realmcast_wrap_content: If `true`, the variable Value column in realmcast output will wrap onto several lines. Setting to `true` possibly enhances readability for very long output. The default is `false`. Section 46.10.6.

realmcast_trunc_content: If `true`, the variable Value column in realmcast output will be truncated. Setting to `true` possibly enhances readability for very long output. The default is `false`. Section 46.10.6.

realmcast_time_format_utc: If `true` the Time column in realmcast output will be shown in UTC time instead of local time since app startup. The default is `false`. Section 46.10.6.

refresh_mode: Determines the manner in which appcast requests are sent to apps. Legal values: `paused`, `events`, `streaming`. The default is `events`. Section 46.10.3.

46.16.5 Configuration Parameters for the Scope, MouseContext and Action Menus

Listing 46.52: Configuration Parameters the Scope, MouseContext and Action Menus.

scope: A comma separated list of MOOS variables to scope. Section 46.11.

oparea: A specification of the operation area boundary for optionally rendering.

button_one: A configurable command and control button. The parameter button_1 may also be used. Section 45.2.3.

button_two: A configurable command and control button. The parameter button_2 may also be used. Section 45.2.3.

button_three: A configurable command and control button. The parameter button_3 may also be used. Section 45.2.3.

button_four: A configurable command and control button. The parameter button_4 may also be used. Section 45.2.3.

button_five-twenty: Additional configurable command and control button release following Release 19.8.1. Section 45.2.3.

action: A MOOS variable-value pair for posting, available under the Action pull-down menu. Section 46.13.

left_context: Allows the custom configuration of left mouse click context. Section 46.14.

right_context: Allows the custom configuration of right mouse click context. Section 46.14.

lclick_ix_start: Starting index for the left mouse index macro. Section 46.14.

rclick_ix_start: Starting index for the right mouse index macro. Section 46.14.

46.16.6 Configuration Parameters for Optimizing in Extreme Load Situations

In missions with a very high number of vehicles and very high time warp, and perhaps also a very high number of geometric objects to render, pMarineViewer will strain, slow or even freeze. The breaking point is partly dependent on the speed and number of CPU cores on your machine. There are a few measures that can be taken to reduce the CPU load and squeeze more performance from pMarineViewer.

Listing 46.53: Configuration Parameter for Optimizing with High Number of Vehicles.

node_report_unc: If true, node reports will be ingested from uFldNodeComms rather than directly from the vehicles. (Introduced *after* Release 19.8.1.)

By setting the node_report_unc parameter to true, pMarineViewer will register for NODE_REPORT_UNC as its source for receiving node reports. Normally this information comes from NODE_REPORT. Normally the latter is shared directly from vehicles. A typical vehicle will produce this message at 4Hz. In simulations with a time warp of say 50, and simulating say 50 vehicles, this results in 10,000 node reports per second arriving in the mailbox of pMarineViewer. While this may be managable, depending on the computer, it consumes a sizeable portion of the CPU load. If there are also happen to be very many geometric objects to render, then the workload will strain.

By ingesting node reports instead from uFldNodeComms, the rate of incoming node reports can be reduced to a level suitable for smooth trajectory rendering. In practice this seems to be about 10-20Hz before the human eye regards the motion as "jumpy". For the extreme case described above, the uFldNodeComms app can reduce the rate from 200Hz to 10Hz. Since uFldNodeComms is handling all these node reports anyway, and since it is likely relegated to a CPU different from pMarineViewer on multi-core machines, it is an effective way to aid pMarineViewer.

46.17 Publications and Subscriptions for pMarineViewer

The interface for pMarineViewer, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pMarineViewer --interface or -i
```

46.17.1 Variables Published by pMarineViewer

It is possible to configure pMarineViewer to poke the MOOSDB via either the Action pull-down menu (Section 46.13), or via configurable GUI buttons (Section 45.2.3). It may also publish to the MOOSDB variables configured to mouse clicks (Section 46.14). So the list of variables that pMarineViewer publishes is somewhat user dependent, but the following few variables may be published in all configurations.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- APPCAST_REQ_<COMMUNITY>: As an appcast viewer, pMarineViewer also generates outgoing appcast requests to MOOS communities it is aware of, including its own MOOS community. These postings are typically bridged to the other named MOOS community with the variable renamed simply to APPCAST_REQ when it arrives in the other community.
- HELM_MAP_CLEAR: This variable is published once when the viewer connects to the MOOSDB. It is used in the pHelmIvP application to clear a local buffer used to prevent successive identical publications to its variables.
- PMV_CONNECT: This variable is published once when the viewer connects to the MOOSDB. It is always published with the value 0. It can be used by any application that would like to publish or republish visual artifacts ensuring that pMarineViewer receives all mail in a sequence.

Otherwise if the viewer comes up after the sequences is published, it may only get the last element in the sequence.

- MVIEWER_LCLICK: When the user clicks the left mouse button, the position in local coordinates, along with the name of the active vehicle is reported. This can be used as a command and control hook as described in Section 46.14.
- MVIEWER_RCLICK: This variable is published when the user clicks with the right mouse button. The same information is published as with the left click.
- MVIEWER_UNHANDLED_MAIL: If the viewer receives mail that it is unable to parse, info on the MOOS variable and its content will be posted in this variable for debugging.
- PLOGGER_CMD: This variable is published with a "COPY_FILE_REQUEST" to log a copy of the image and info file, only if log_the_image is set to true. Section 45.3.2.

46.17.2 Variables Subscribed for by pMarineViewer

- APPCAST: As an appcast enabled *viewer*, pMarineViewer also subscribes for appcasts from other other applications and communities to provide the content for its own viewing capability.
- APPCAST_REQ: As an appcast enabled MOOS application, pMarineViewer also subscribes for appcast requests. Each incoming message is a request to generate and post a new appcast report, with reporting criteria, and expiration.
- REALMCAST: As an realmcast enabled *viewer*, pMarineViewer also subscribes for realmcasts from MOOS communities running pRealm, and renders them in the infocast panes described in Section 46.10.5.
- WATCHCAST: As an realmcast enabled *viewer*, pMarineViewer also subscribes for watchcasts from MOOS communities running pRealm, and renders them in the infocast panes described in Section 46.10.7.
- NODE_REPORT: This is the primary variable consumed by pMarineViewer for collecting vehicle position information.
- NODE_REPORT_LOCAL: This serves the same purpose as the above variable. In some simulation cases this variable is used.
- PHI_HOST_INFO: A string representing the detected IP address, published by pHostInfo. Used for augmenting the pMarineViewer title bar with the current IP address.
- TRAIL_RESET: When the viewer receives this variable it will clear the history of trail points associated with each vehicle. This is used when the viewer is run with a simulator and the vehicle position is reset and the trails become discontinuous.
- VIEW_CIRCLE: A string representation of an XYCircle object.
- VIEW_COMMS_PULSE: A string representation of an XYCommsPulse object.
- VIEW_GRID: A string representation of a XYConvexGrids object.
- VIEW_GRID_CONFIG: A string representation of a XYGrid configuration.
- VIEW_GRID_DELTA: A string representation of a XYGrid configuration.
- VIEW_POINT: A string representation of an XYPoint object.
- VIEW_POLYGON: A string representation of an XYPolygon object.
- VIEW_SEGLIST: A string representation of an XYSegList object.

- VIEW_MARKER: A string designation of a marker type, size and location.
- VIEW_RANGE_PULSE: A string representation of an XYRangePulse object.
- VIEW_VECTOR: A string representation of an XYVector object

47 uHelmScope: Scoping on the IvP Helm

47.1 Overview

The uHelmScope application is a console based tool for monitoring output of the IvP helm, i.e., the pHelmIvP process. The helm produces a few key MOOS variables on each iteration that pack in a substantial amount of information about what happened during a particular iteration. The helm scope subscribes for and parses this information, and writes it to a console window for the user to monitor. The user can dynamically pause or alter the output format to suit one's needs, and multiple scopes can be run simultaneously. The helm scope in no way influences the performance of the helm - it is strictly a passive observer.

The example console output shown in Listing 54 is used for explaining the uHelmScope fields.

Listing 47.54: Example uHelmScope output.

```
 1  (alpha)(PAUSED)===========   uHelmScope Report  ============= DRIVE (133)
 2    Helm Iteration: 85
 3    IvP Functions:  1
 4    Mode(s):
 5    SolveTime:   0.00   (max=0.01)
 6    CreateTime:  0.00   (max=0.01)
 7    LoopTime:    0.00   (max=0.02)
 8    Halted:         false   (0 warnings)
 9  Helm Decision: [speed,0,4,21] [course,0,359,360]
10    speed = 2
11    course = 114
12  Behaviors Active: ---------- (1)
13    waypt_survey [21.29] (pwt=100) (pcs=6) (cpu=0.08) (upd=0/0)
14  Behaviors Running: --------- (0)
15  Behaviors Idle: ------------ (1)
16    waypt_return [always]
17  Behaviors Completed: ------- (0)
18
19
20  #  MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
21  #
22  #  VarName            Source         Time  Commty  VarValue
23  #  -----------------  ------------  -----  ------  ----------------------
24  #  DEPLOY             pMari..iewer  25.05  alpha   "true"
25  #  IVPHELM_STATEVARS  pHelmIvP       5.42  alpha   "DEPLOY,MISSION,RETURN"
26  #  MISSION            n/a             n/a          n/a
27  #  RETURN             pMari..iewer  25.05  alpha   "false"               8
28
29
30  @  BEHAVIOR-POSTS TO MOOSDB ---------------------- (Hit '@' to en/disable)
31  @
32  @  MOOS Var      Behavior      Iter  Value
33  @  ------------  ------------  ----  ---------------------------------------
34  @  BHV_STATUS    waypt_return     1  name=waypt_return,p..te=idle,updates=n/a
35  @  ------------  ------------  ----  ---------------------------------------
36  @  CYCLE_INDEX   waypt_survey     1  0
37  @  VIEW_POINT    waypt_survey     1  x=60,y=-40,active=f..r=red,vertex_size=4
38  @  VIEW_SEGLIST  waypt_survey     1  pts={60,-40:60,-160.._size=4,edge_size=1
39  @  WPT_INDEX     waypt_survey     1  0
40  @  WPT_STAT      waypt_survey    84  vname=alpha,behavio..es=0,dist=30,eta=15
```

There are three groups of information in the uHelmScope output on each report to the console - the general helm overview (lines 1-17), a MOOSDB scope for a select subset of MOOS variables (lines

20-27), and a report on the MOOS variables published by the helm on the current iteration (lines 30-40). The output of each group is explained in the next three subsections.

47.2 The Helm Summary Section of the uHelmScope Output

The first block of output produced by uHelmScope provides an overview of the helm. This is lines 1-17 in Listing 54, but the number of lines may vary with the mission and state of mission execution. This block is virtually identical to the appcast report generated by the helm itself. So another way of doing uHelmScope style scoping is with an appcast viewing tool (uMAC, uMACView, and pMarineViewer). But with these tools, you would only see part of the information found in uHelmScope. The MOOSDB-Scope and Behaviors-Post portion of of uHelmScope is not part of the appcast report posted by the helm.

47.2.1 The Helm Status (Lines 1-8)

The integer value at the end of line 1 indicates the number of uHelmScope reports written to the console. This can confirm to the user that an action that should result in a new report generation has indeed worked properly. The integer on line 2 is the counter kept by the helm, incremented on each helm iteration. The value on Line 3 represents the the number of IvP functions produced by the active helm behaviors, one per active behavior. The solve-time on line 5 represents the time, in seconds, needed to solve the IvP problem comprised the n IvP functions. The number that follows in parentheses is the maximum solve-time observed by the scope. The create-time on line 6 is the total time needed by all active behaviors to produce their IvP function output. The loop time on line 7 is simply the sum of lines 5 and 6.

The Boolean on line 8 is true only if the helm is halted on an emergency or critical error condition. Also on line 8 is the number of warnings generated by the helm. This number is reported by the helm and *not* simply the number of warnings observed by the scope. This number coincides with the number of times the helm writes a new message to the variable BHV_WARNING.

47.2.2 The Helm Decision (Lines 9-11)

The helm decision space, i.e., IvP domain, is displayed on line 9. Each decision variable is given by its name, low value, high value, and the number of decision points. So `[speed,0,4,21]` represents values {0, 0.25, 0.5, ..., 3.75, 4.0}. The following lines used to display the actual helm decision. Occasionally the helm may be configured with one of its decision variables configured to be *optional*. The helm may not produce a decision on that variable on some iteration if no behaviors are reasoning about that variable. In this case the label `"varbalk"` may be shown next to the decision variable to indicate no decision.

47.2.3 The Helm Behavior Summary (Lines 12-17)

Following this is a list of all the active, running, idle and completed behaviors. At any point in time, each instantiated IvP behavior is in one of these four states and each behavior specified in the behavior file should appear in one of these groups. Technically all *active* behaviors are also *running* behaviors but not vice versa. So only the running behaviors that are not active (i.e., the behaviors that could have, but chose not to produce an objective function), are listed in the

"Behaviors Running:" group. Immediately following each behavior the time, in seconds, that the behavior has been in the current state is shown in parentheses. For the active behaviors (see line 13) this information is followed by the priority weight of the behavior, the number of pieces in the produced IvP function, and the amount of CPU time required to build the function. If the behavior also is accepting dynamic parameter updates the last piece of information on line 13 shows how many successful updates where made against how many attempts. A failed update attempt also generates a helm warning, counted on line 8. The idle and completed behaviors are listed by default one per line. This can be changed to list them on one long line by hitting the 'b' key interactively.

47.3 The MOOSDB-Scope Section of the uHelmScope Output

A built-in generic scope function is built into uHelmScope, not different in style from uXMS. The scope ability in uHelmScope provides two advantages: first, it is simply a convenience for the user to monitor a few key variables in the same screen space. Second, uHelmScope automatically registers for the variables that the helm reasons over to determine the behavior activity states. It will register for all variables appearing in behavior conditions, runflags, activeflags, inactiveflags, endflags and idleflags. It will also register for variables involved in the helm hierarchical mode definitions. The list of these variables is provided by the helm itself when it publishes IVPHELM_STATEVARS.

For example, the output in Listing 54 was derived from scoping on the alpha mission, and launching from the terminal with:

```
$ uHelmScope alpha.moos IVPHELM_STATEVARS
```

In this case the variable IVPHELM_STATEVARS itself is added to the scope list, and the *value* of this variable contains the three other variables on the scope list, reported by the helm to be involved in conditions or flags. The MISSION variable has not been written to because MISSION="complete" is the endflag of the return behavior in the alpha mission. At the point where this snapshot was taken, this behavior had not completed.

The lines comprising the MOOSDB-Scope section of the uHelmScope output are all preceded by the # character. This is to help discern this block from the others, and as a reminder that the whole block can be toggled off and on by typing the # character. The columns in Listing 54 are truncated to a set maximum width for readability. The default is to have truncation turned on. The mode can be toggled by the console user with the 't' character, or set in the MOOS configuration block or with a command line switch. A truncated entry in the VarValue column has a '+' at the end of the line. Truncated entries in other columns will have ".." embedded in the entry. Line 24 shows an example of both kinds of truncation.

The variables included in the scope list can be specified in the uHelmScope configuration block of a MOOS file. In the MOOS file, the lines have the form:

```
var = <MOOSVar>, <MOOSVar>, ...
```

An example configuration is given in Listing 58. Variables can also be given on the command line. Duplicates requests, should they occur, are simply ignored. Occasionally a console user may want to suppress the scoping of variables listed in the MOOS file and instead only scope on a couple

variables given on the command line. The command line switch `-c` will suppress the variables listed in the MOOS file - unless a variable is also given on the command line. In line 26 of Listing 54, the variable `MISSION` is a *virgin* variable, i.e., it has yet to be written to by any MOOS process and shows `n/a` in the five output columns. By default, virgin variables are displayed, but their display can be toggled by the console user by typing `'v'`.

47.4 The Behavior-Posts Section of the uHelmScope Output

The Behavior-Posts section is the third group of output in `uHelmScope`. It lists MOOS variables and values posted by the helm. Each variable was posted by a particular helm behavior and the grouping in the output is accordingly by behavior. Unlike the variables in the MOOSDB-Scope section, entries in this section only appear if they were written by the helm. The lines comprising the Behavior-Posts section of the `uHelmScope` output are all preceded by the '@' character. This is to help discern this block from the others, and as a reminder that the whole block can be toggled off and on by typing the '@' character. As with the output in the MOOSDB-Scope output section, the output may be truncated. A value that has been truncated will contain the `".."` characters around the middle of the string as in lines 34, 37-38, and 40.

47.5 Console Key Mapping and Command Line Usage

User input is accepted at the console during a `uHelmScope` session, to adjust either the content or format of the reports. It operates in a couple different *refresh* modes. In the *paused* refresh mode, after a report is posted to the console no further output is generated until the user requests it. In the *streaming* refresh mode, new helm summaries are displayed as soon as they are received. The refresh mode is displayed in the report on the very first line as in Listing 54.

The key mappings can be summarized in the console output by typing the 'h' key, which also sets the refresh mode to *paused*. The key mappings shown to the user are shown in Listing 55.

Listing 47.55: Key mapping summary shown after hitting 'h' in a console.

```
 1  KeyStroke  Function
 2  ---------  ---------------------------------------------
 3  Getting Help:
 4      h      Show this Help msg - 'r' to resume
 5
 6  Modifying the Refresh Mode:
 7     Spc     Refresh Mode: Pause (after updating once)
 8      r      Refresh Mode: Streaming (throttled)
 9      R      Refresh Mode: Streaming (unthrottled)
10
11  Modifying the Content Mode:
12      d      Content Mode: Show normal reporting (default)
13      w      Content Mode: Show behavior warnings
14      l      Content Mode: Show life events
15      m      Content Mode: Show hierarchical mode structure
16
17  Modifying the Content Format or Filtering:
18      b      Toggle Show Idle/Completed Behavior Details
19      `      Toggle truncation of column output
20      v      Toggle display of virgins in MOOSDB-Scope output
```

```
21      #       Toggle Show the MOOSDB-Scope Report
22      @       Toggle Show the Behavior-Posts Report
23
24  Hit 'r' to resume outputs, or SPACEBAR for a single update
```

Several of the same preferences for adjusting the content and format of the uHelmScope output can be expressed on the command line, with a command line switch. Command line usage is shown in Listing 56, and may be obtained from the command line by invoking:

```
$ uHelmScope --help or -h
```

Listing 47.56: Command line usage of uHelmScope.

```
 1  ============================================================
 2  Usage: uHelmScope file.moos [OPTIONS] [MOOS Variables]
 3  ============================================================
 4
 5  Options:
 6    --alias=<ProcessName>
 7        Launch uHelmScope with the given process name rather
 8        than uHelmScope.
 9    --clean, -c
10        MOOS variables specified in given .moos file are excluded
11        from the MOOSDB-Scope output block.
12    --example, -e
13        Display example MOOS configuration block.
14    --help, -h
15        Display this help message.
16    --interface, -i
17        Display MOOS publications and subscriptions.
18    --noscope,-x
19        Suppress MOOSDB-Scope output block.
20    --noposts,-p
21        Suppress Behavior-Posts output block.
22    --novirgins,-g
23        Suppress virgin variables in MOOSDB-Scope output block.
24    --streaming,-s
25        Streaming (unpaused) output enabled.
26    --trunc,-t
27        Column truncation of scope output is enabled.
28    --version,-v
29        Display the release version of uHelmScope.
30
31  MOOS Variables
32    MOOS_VAR1 MOOSVAR_2, ..., MOOSVAR_N
33
34  Further Notes:
35    (1) The order of command line arguments is irrelvant.
36    (2) Any MOOS variable used in a behavior run condition or used
37        in hierarchical mode declarations will be automatically
38        subscribed for in the MOOSDB scope.
```

The command line invocation also accepts any number of MOOS variables to be included in the MOOSDB-Scope portion of the uHelmScope output. Any argument on the command line that does

not end in .moos, and is not one of the switches listed above, is interpreted to be a requested MOOS variable for inclusion in the scope list. Thus the order of the switches and MOOS variables do not matter. These variables are added to the list of variables that may have been specified in the uHelmScope configuration block of the MOOS file. Scoping on *only* the variables given on the command line can be accomplished using the -c switch. To support the simultaneous running of more than one uHelmScope connected to the same MOOSDB, uHelmScope generates a random number N between 0 and 10,000 and registers with the MOOSDB as uHelmScope_N.

47.6 Helm-Produced Variables Used by uHelmScope

There are six variables published by the helm to which uHelmScope subscribes. These provide critical information for generating uHelmScope reports.

The first two variables, IVPHELM_STATE and IVPHELM_SUMMARY are published on each iteration of the helm. The former is published regardless of the helm state. This variable serves as the helm heartbeat. The latter is only published when the helm is in the DRIVE state. The below examples give a feel for the content:

```
IVPHELM_STATE    = "DRIVE"
IVPHELM_SUMMARY  = "iter=66,ofnum=1,warnings=0,utc_time=1209755370.74,solve_time=0.00,
                   create_time=0.02,loop_time=0.02,var=speed:3.0,var=course:108.0,
                   halted=false,running_bhvs=none,
                   active_bhvs=waypt_survey$6.8$100.00$1236$0.01$0/0,
                   modes=MODE@ACTIVE:SURVEYING,idle_bhvs=waypt_return$55.3$n/a,
                   completed_bhvs=none"
```

The IVPHELM_SUMMARY variable contains all the dynamic information included in the general helm overview (top) section of the uHelmScope output. It is a comma-separated list of var=val pairs. The helm publishes this in a journal style, ommitting certain content if they are unchanged between iterations. When uHelmScope launches, it publishes to the variable IVPHELM_REJOURNAL which the helm interprets as a request to send a full-content message on the next iteration, before resuming journaling.

The IVPHELM_LIFE_EVENT is posted only when a behavior is spawned or dies. For missions without dynamic behavior spawning, this variable will only be posted upon startup for each initial static behavior. Note that the helm only publishes life events as they occur, so if the helm scope is launched after the helm, earlier events may not be reflected in the life event report. The below example gives a feel for the content of this variable:

```
IVHELM_LIFE_EVENT = "time=814.09, iter=3217, bname=bng-line-bng-line132--104,
                    btype=BHV_BearingLine, event=spawn,
                    seed=name=bng-line132--104#bearing_point=132,-104"
```

The IVPHELM_DOMAIN, IVPHELM_STATEVARS, and IVPHELM_MODESET variables are typically only produced once, upon startup.

```
IVPHELM_DOMAIN    = "speed,0,4,21:course,0,359,360"
```

```
IVPHELM_STATEVARS = "RETURN,DEPLOY"
IVPHELM_MODESET   = "---,ACTIVE#---,INACTIVE#ACTIVE,SURVEYING#ACTIVE,RETURNING"
```

The IVP_DOMAIN variable also contributes to this section of output by providing the IvP domain used by the helm. The IVPHELM_STATEVARS variable affects the MOOSDB-Scope section of the uHelmScope output by identifying which MOOS variables are used by behaviors in conditions, runflags, endflags and idleflags.

47.7 Configuration Parameters for uHelmScope

Configuration for uHelmScope amounts to specifying a set of parameters affecting the terminal output format. An example configuration is shown in Listing 58, with all values set to the defaults. Launching uHelmScope with a MOOS file that does not contain a uHelmScope configuration block is perfectly reasonable.

Listing 47.57: Configuration Parameters for uHelmScope.

behaviors_concise:	If true, the idle and completed behaviors are reported all on one line rather than separate lines. Legal values: true, false. The default is true.
display_bhv_posts:	If true, the behavior-posts section of the report is shown. This can also be toggled at run time with the '@' key. Legal values: true, false. The default is true. Section 47.4.
display_moos_scope:	If true, the MOOS variable scope section of the report is shown. This can also be toggled at run time with the '#' key. Legal values: true, false. The default is true. Section 47.3.
display_virgins:	If true, variables in the MOOS scope section of the report is shown will be shown even if they have never been written to. This can also be toggled at run time with the 'g' key. Legal values: true, false. The default is true. Section 47.3.
paused:	If true, uHelmScope launches in the paused mode. Legal values: true, false. Default value is true.
tuncated_output:	If true, output in the MOOS-scope or behavior-scope section of the report is truncated. This can also be toggled at run time with the '‘' key. Legal values: true, false. The default is false.
var:	A comma-separated list of variables to scope on in the MOOS-Scope block. Multiple lines may be provided. Section 47.3.

An example configuration file may be obtained from the command line with:

```
$ uHelmScope --example or -e
```

This will show the output shown in Listing 58 below.

Listing 47.58: Example configuration of the uHelmScope application.

```
1  ============================================================
2  uHelmScope Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = uHelmScope
6  {
7    AppTick   = 1    // MOOSApp default is 4
8    CommsTick = 1    // MOOSApp default is 4
9
10   paused    = false   // default
11
12   display_moos_scope = true     // default
13   display_bhv_posts  = true     // default
14   display_virgins    = true     // default
15   truncated_output   = false    // default
16   behaviors_concise  = true     // default
17
18   var  = NAV_X, NAV_Y, NAV_SPEED, NAV_DEPTH   // MOOS vars are
19   var  = DESIRED_HEADING, DESIRED_SPEED       // case sensitive
20 }
```

Each of the parameters can also be set on the command line, or interactively at the console, with one of the switches or keyboard mappings listed in Section 47.5. A parameter setting in the MOOS configuration block will take precedence over a command line switch.

47.8 Publications and Subscriptions for uHelmScope

The interface for uHelmScope, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uHelmScope --interface or -i
```

47.8.1 Variables Published by uHelmScope

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- IVPHELM_REJOURNAL: A request to the helm to rejournal its summary output. Section 47.6.

47.8.2 Variables Subscribed for by uHelmScope

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- <USER-DEFINED>: Variables identified for scoping by the user in the uHelmScope will be subscribed for. See Section 47.3.
- <HELM-DEFINED>: As described in Section 47.3, the variables scoped by uHelmScope include any variables involved in the preconditions, runflags, idleflags, activeflags, inactiveflags, and endflags for any of the behaviors involved in the current helm configuration.
- IVPHELM_LIFE_EVENT: A description of a helm life event, the birth or death of a behavior and the manner in which it was spawned. See Section 47.6.

- IVPHELM_SUMMARY: A comprehensive summary of the helm status including behavior status summaries and most recent helm decision. See Section 47.6.
- IVPHELM_STATEVARS: A helm-produced list of MOOS variables involved in the logic of determining behavior activation. Any variable involved in mode conditions or behavior conditions. See Section 47.6.
- IVPHELM_DOMAIN: The specification of the IvP Domain presently used by the helm. See Section 47.6.
- IVPHELM_MODESET: A description of the helm's hierarchical mode specification. See Section 47.6.
- IVPHELM_STATE: A short description of the helm state: either STANDBY, PARK, DRIVE, or DISABLED. See Section 47.6.

48 uTimerScript: Scripting Events to the MOOSDB

48.1 Overview

The uTimerScript application allows the user to script a set of pre-configured posts to a MOOSDB. In its most basic form, it may be used to initialize a set of variables to the MOOSDB, and immediately terminate itself if a quit event is included. The following configuration block, if placed in the alpha example mission, would mimic the posts to the MOOSDB behind the DEPLOY button, simply disabling manual control, deploying the vehicle and quitting the script: Listing 48.1.59.

Listing 48.59: A Simple Timer Script.

```
1  ProcessConfig = uTimerScript
2  {
3    event = var=MOOS_MANUAL_OVERIDE, val=false
4    event = var=DEPLOY, val=true
5    event = quit
6  }
```

Additionally, uTimerScript may be used with the following advanced functions:

- Each entry in the script may be scheduled to occur after a specified amount of elapsed time.
- Event timestamps may be given as an exact point in time relative to the start of the script, or a range in times with the exact time determined randomly at run-time.
- The execution of the script may be paused, or fast-forwarded a given amount of time, or forwarded to the next event on the script by writing to a MOOS variable.
- The script may be conditionally paused based on user defined logic conditions over one or more MOOS variables.
- The variable value of an event may also contain information generated randomly.
- The script may be reset or repeated any given number of times.
- The script may use its own time warp, which can be made to vary randomly between script executions.

In short, uTimerScript may be used to effectively simulate the output of other MOOS applications when those applications are not available. A few examples are provided, including a simulated GPS unit and a crude simulation of wind gusts.

48.2 Using uTimerScript

Configuring a script minimally involves the specification of one or more events, with an event comprising of a MOOS variable and value to be posted and an optional time at which it is to be posted. Scripts may also be reset on a set policy, or from a trigger by an external process.

48.2.1 Configuring the Event List

The event list or script is configured by declaring a set of event entries with the following format:

```
event = var=<MOOSVar>, val=<value>, [time=<time-of-event>]
```

The keywords `event`, `var`, `val`, and `time` are not case sensitive, but the values `<moos-variable>` and `<var-value>` are case sensitive. The `<var-value>` type is posted either as a string or double based on the following heuristic: if the `<var-value>` has a numerical value it is posted as a double, and otherwise posted as a string. If one wants to post a string with a numerical value, putting quotes around the number suffices to have it posted as a string. Thus `val=99` posts a double, but `var="99"` posts a string. If a string is to be posted that contains a comma such as `"apples, pears"`, one must put the quotes around the string to ensure the comma is interpreted as part of `<var-value>`. The value field may also contain one or more macros expanded at the time of posting, as described in Section 48.4.

48.2.2 Setting the Event Time or Range of Event Times

The value of `<time-of-event>` is given in seconds and must be a numerical value greater or equal to zero. The time represents the amount of elapsed time since the uTimerScript was first launched and un-paused. The list of events provided in the configuration block need not be in order - they will be ordered by the uTimerScript utility. The `<time-of-event>` may also be specified by a interval of time, e.g., `time=0:100`, such that the event may occur at some point in the range with uniform probability. The only restrictions are that the lower end of the interval is greater or equal to zero, and less than or equal to the higher end of the interval. By default the timestamps are calculated once from their specified interval, at the the outset of uTimerScript. The script may alternatively be configured to recalculate the timestamps from their interval each time the script is reset, by setting the `shuffle` parameter to true. This parameter, and resetting in general, are described in the next Section 48.2.3.

48.2.3 Resetting the Script

The timer script may be reset to its initial state, resetting the stored elapsed-time to zero and marking all events in the script as pending. This may occur either by cueing from an event outside uTimerScript, or automatically from within uTimerScript. Outside-cued resets can be triggered by posting UTS_RESET with the value `="reset"`, or `"true"`. The `reset_var` parameter names a MOOS variable that may be used as an alternative to UTS_RESET. It has the format:

```
reset_var = <moos-variable> // Default is UTS_RESET
```

The script may be also be configured to auto-reset after a certain amount of time, or immediately after all events are posted, using the `reset_time` parameter. It has the format:

```
reset_time = <time-or-condition> // Default is "none"
```

The `<time-or-condition>` may be set to `"all-posted"` which will reset after the last event is posted. If set to a numerical value greater than zero, it will reset after that amount of elapsed time, regardless of whether or not there are pending un-posted events. If set to `"none"`, the default, then no automatic resetting is performed. Regardless of the `reset_time` setting, prompted resets via the UTS_RESET variable may take place when cued.

The script may be configured to accept a hard limit on the number of times it may be reset. This is configured using the `reset_max` parameter and has the following format:

```
reset_max = <amount> // Default is "nolimit"
```

The <amount> specified may be any number greater or equal to zero, where the latter, in effect, indicates that no resets are permitted. If unlimited resets are desired (the default), the case insensitive argument "unlimited" or "any" may be used.

The script may be configured to recalculate all event timestamps specified with a range of values whenever the script is reset. This is done with the following parameter:

```
shuffle = false // Default is "true"
```

The script may be configured to reset or restart each time it transitions from a situation where its conditions are not met to a situation where its conditions are met, or in other words, when the script is "awoken". The use of logic conditions is described in more detail in Section 48.3.1. This is done with the following parameter:

```
upon_awake = restart // Default is "n/a", no action
```

Note that this does not apply when the script transitions from being paused to un-paused as described in Section 48.3.1. See the example in Section 48.9.2 for a case where the upon_awake feature is handy.

48.3 Script Flow Control

The script flow may be affected in a number of ways in addition to the simple passage of time. It may be (a) paused by explicitly pausing it, (b) implicitly paused by conditioning the flow on one or more logic conditions, (c) fast-forwarded directly to the next scheduled event, or fast-forwarded some number of seconds. Each method is described in this section.

48.3.1 Pausing the Timer Script

The script can be paused at any time and set to be paused initially at start time. The paused parameter affects whether the timer script is actively unfolding at the outset of launching uTimerScript. It has the following format:

```
paused = <Boolean>
```

The keyword paused and the string representing the Boolean are not case sensitive. The Boolean simply must be either "true" or "false". By setting paused to true, the elapsed time calculated by uTimerScript is paused and no variable-value pairs will be posted. When un-paused the elapsed time begins to accumulate and the script begins or resumes unfolding. The default value of paused is false.

The script may also be paused through the MOOS variable UTS_PAUSE which may be posted by some other MOOS application. The values recognized are "true", "false", or "toggle", all case insensitive. The name of this variable may be substituted for a different one with the pause_var parameter in the uTimerScript configuration block. It has the format:

```
pause_var = <MOOSVar> // Default is UTS_PAUSE
```

If multiple scripts are being used (with multiple instances of uTimerScript connected to the MOOSDB),

setting the pause_var to a unique variable may be needed to avoid unintentionally pausing or un-pausing multiple scripts with single write to UTS_PAUSE.

48.3.2 Conditional Pausing of the Timer Script and Atomic Scripts

The script may also be configured to condition the "paused-state" to depend on one or more logic conditions. If conditions are specified in the configuration block, the script must be both un-paused as described above in Section 48.3.1, and all specified logic conditions must be met in order for the script to begin or resume proceeding. The logic conditions are configured as follows:

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

The <logic-expression> syntax is described in Logic Appendix, and may involve the simple comparison of MOOS variables to specified literal values, or the comparison of MOOS variables to one another. See the script configuration in Section 48.9.2 for one example usage of logic expressions.

An *atomic* script is one that does not check conditions once it has posted its first event, and prior to posting its last event. Once a script has started, it is treated as unpausable with respect to the the logic conditions. This is configured with:

```
script_atomic = <Boolean>
```

It can however be paused and unpaused via the pause variable, e.g., UTS_PAUSE, as described in Section 48.3.1. If the logic conditions suddenly fail in an atomic script midway, the check is simply postponed until after the script completes and is perhaps reset. If the conditions in the meanwhile revert to being satisfied, then no interruption should be observable.

48.3.3 Fast-Forwarding the Timer Script

The timer script, when un-paused, moves forward in time with events executed as their event times arrive. However, the script may be moved forwarded by writing to the MOOS variable UTS_FORWARD. If the value received is zero (or negative), the script will be forwarded directly to the point in time at which the next scheduled event occurs. If the value received is positive, the elapsed time is forwarded by the given amount. Alternatives to the MOOS variable UTS_FORWARD may be configured with the parameter:

```
forward_var = <MOOSVar> // Default is UTS_FORWARD
```

If multiple scripts are being used (with multiple instances of uTimerScript connected to the MOOSDB), setting the forward_var to a unique variable may be needed to avoid unintentionally fast forwarding multiple scripts with single write to UTS_FORWARD.

48.3.4 Quitting the Timer Script

The timer script may be configured with a special event, the *quit* event, resulting in disconnection with the MOOSDB and a process exit. This is done with the configuration:

```
event = quit [time=<time-of-event>]
```

Before quitting, a final posting to the MOOSDB is made with the variable EXITED_NORMALLY. The value is

"uTimerScript", or its alias if an alias was used. This indicates to any other watchdog process, such as uProcessWatch, that the exiting of this script is not a reason for concern. When uTimerScript receives its own posting in the next incoming mail, it assumes all pending posts have been made and will then quit.

48.4 Macro Usage in Event Postings

Macros may be used to add a dynamic component to the value field of an event posting. This substantially expands the expressive power and possible uses of the uTimerScript utility. Recall that the components of an event are defined by:

```
event = var=<MOOSVar>, val=<var-value>, time=<time-of-event>
```

The <var-value> component may contain a macro of the form $[MACRO], where the macro is either one of a few built-in macros available, or a user-defined macro with the ability to represent random variables. Macros may also be combined in simple arithmetic expressions to provide further expressive power. In each case, the macro is expanded at the time of the event posting, typically with different values on each successive posting.

48.4.1 Built-In Macros Available

There are five built-in macros available:

- $[DBTIME]: The estimated time since the MOOSDB started, similar to the value in the MOOS variable DB_UPTIME published by the MOOSDB.

An example usage:

```
event = var=DEPLOY_RECEIVED, val=$[DBTIME], time=10:20
```

- $[UTCTIME]: The UTC time, in seconds, at the time of the posting.
- $[UTC]: The UTC time, in seconds, at the time of the posting.
- $[COUNT]: Expands to the integer total of all posts thus far in the current execution of the script, and is reset to zero when the script resets.
- $[TCOUNT]: Expands to the integer total of all posts thus far since the application began, i.e., it is a running total that is not reset when the script is reset.
- $[IDX]: Expands to the integer value representing an event's count or index into the sequence of events. However, it will always post as a string and will be padded with zeros to the left, e.g., "000", "001", ..., and so on.

If the value of a MOOS posting is solely a macro, the variable type of the posting is a double, not a string. For example val=$[DBTIME] will post a type double, whereas val="time:$[DBTIME]" will post a type string.

48.4.2 User Configured Macros with Random Variables

Further macros are available for use in the <var-value> component of an event, defined and configured by the user, and based on the idea of a random variable. In short, the macro may expand to a

numerical value chosen within a user specified range, and recalculated according to a user-specified policy. The general format is:

```
rand_var = varname=<variable>, min=<value>, max=<value>, key=<key_name>
```

The <variable> component defines the macro name. The <low_value> and <high_value> components define the range from which the random value will be chosen uniformly. The <key_name> determines when the random value is reset. It must be set to one of the following three values: "at_start", "at_reset", and "at_post". Random variables with the key name "at_start" are assigned a random value only at the start of the uTimerScript application. Those with the "at_reset" key name also have their values re-assigned whenever the script is reset. Those with the "at_post" key name also have their values re-assigned after any event is posted.

48.4.3 Support for Simple Arithmetic Expressions with Macros

Macros that expand to numerical values may be combined in simple arithmetic expressions with other macros or scalar values. The general form is:

```
{<value> <operator> <value>}
```

The <value> components may be either a scalar or a macro, and the <operator> component may be one of '+', '-', '*', '/'. Nesting is also supported. Below are some examples:

```
{$[FOOBAR] * 0.5}
{-2-$[FOOBAR]}
{$[APPLES] + $[PEARS]}
{35 / {$[FOOBAR]-2}}
{$[DBTIME] - {35 / {$[UTCTIME]+2}}}
```

If a macro should happen to expand to a string rather than a double (numerical) value, the string evaluates to zero for the sake of the remaining evaluations.

48.5 Time Warps, Random Time Warps, and Restart Delays

A time warp and initial start delay may be optionally configured into the script to change the event schedule without having to edit all the time entries for each event. They may also be configured to take on a new random value at the outset of each script execution to allow for simulation of events in nature or devices having a random component.

48.5.1 Random Time Warping

The time warp is a numerical value in the range $(0, \infty]$, with a default value of 1.0. Lower values indicate that time is moving more slowly. As the script unfolds, a counter indicating "elapsed_time" increases in value as long as the script is not paused. The "elapsed_time" is multiplied by the time warp value. The time warp may be specified as a single value or a range of values as below:

```
time_warp = <value>
time_warp = <low-value>:<high-value>
```

When a range of values is specified, the time warp value is calculated at the outset, and re-calculated whenever the script is reset. See the example in Section 48.9.3 for a use of random time warping to simulate random wind gusts.

48.5.2 Random Initial Start and Reset Delays

A start delay may be provided with the `delay_start` parameter, given in seconds in the range $[0, \infty]$, with a default value of 0. The effect of having a non-zero delay of n seconds is to have `elapsed_time=n` at the outset of the script, on the first time through the script only. Thus a delay of n seconds combined with a time warp of 0.5 would result in observed delay of $2 * n$ seconds. The start delay may be specified as a single value or a range of values as below:

```
delay_start = <value>
delay_start = <low-value>:<high-value>
```

To specify a delay applied at the beginning if the script *after a reset*, use the `delay_reset` parameter instead.

```
delay_reset = <value>
delay_reset = <low-value>:<high-value>
```

When a range of values is specified, the start ore reset delay value is calculated at the outset, and re-calculated whenever the script is reset. See the example in Section 48.9.2 for a use of random start delays to the simulate the delay in acquiring satellite fixes in a GPS unit on an UUV coming to the surface.

48.5.3 Status Messages Posted to the MOOSDB by uTimerScript

The `uTimerScript` periodically publishes a string to the MOOS variable `UTS_STATUS` indicating the status of the script. This variable will be published on each iteration if one of the following conditions is met: (a) two seconds has passed since the previous status message posted, or (b) an event has been been posted, or (c) the paused state has changed, or (d) the script has been reset, or (e) the state of script logic conditions has changed. A posting may look something like:

```
UTS_STATUS = "name=RND_TEST, elapsed_time=2.00, posted=1, pending=5, paused=false,
              conditions_ok=true, time_warp=3, start_delay=0, shuffle=false,
              upon_awake=restart, resets=2/5"
```

In this case, the script has posted one of six events (`posted=1, pending=5`). It is actively unfolding, since `paused=false` (Section 48.3.1) and `conditions_ok=true` (Section 48.3.2). It has been reset twice out of a maximum of five allowed resets (`resets=2/5`, Section 48.2.3). Time warping is being deployed `time_warp=3` (Section 48.5), there is no start delay in use `start_delay=0` (Section 48.5.2).

The shuffle feature is turned off `shuffle=false` (Section 48.2.3). The script is not configured to reset upon re-entering the un-paused state, `awake_reset=false` (Section 48.2.3).

When multiple scripts are running in the same MOOS community, one may want to take measures to discern between the status messages generated across scripts. One way to do this is to use a unique MOOS variable other than `UTS_STATUS` for each script. The variable used for publishing the status may be configured using the `status_var` parameter. It has the following format:

```
status_var = <MOOSVar> // Default is UTS_STATUS
```

Alternatively, a unique name may be given to each to each script. All status messages from all scripts would still be contained in postings to `UTS_STATUS`, but the different script output could be discerned by the name field of the status string. The script name is set with the following format.

```
script_name = <string> // Default is "unnamed"
```

48.6 Terminal and AppCast Output

The script configuration and progress of script execution may also be monitored from an open console window where `uTimerScript` is launched, or through an appcast viewer. Example output is shown below in Listing 60. On line 2, the name of the local community or vehicle name is listed on the left. On the right, `"0/0(450)` indicates there are no configuration or run warnings, and the current iteration of `uFldTimerScript` is 450.

48.6.1 Lines 4-16: Script Configuration

Lines 4-11 show the script configuration. Line 5 shows the number of elements in the script and in parentheses the last element to have been posted. Line 6 shows the number of times the script has restarted. Line 7 shows the present time warp and the range of time warps possible on each script restart in brackets (Section 48.5.1. Lines 8-9 show the delay applied at the start and after a script reset (Section 48.5.2). Line 10 indicates the script is presently not paused (Section 48.3.1). Line 11 indicates the script presently meets any prevailing logic conditions (Section 48.3.2). Lines 13-16 show that there are two random variables defined for this script that may be used in event definitions. They are both uniform random variables. The first varies over possible directions, and the second over possible speed magnitudes. Section 48.4.2.

Listing 48.60: Example uTimerScript console and appcast output.

```
1  ============================================================
2  uTimerScript charlie                                 0/0(450)
3  ============================================================
4  Current Script Information:
5      Elements: 10(8)
6       Reinits: 2
7     Time Warp: 1.09 [0.2,2]
8   Delay Start: 0
9   Delay Reset: 23.66 [10,60]
10        Paused: false
11 ConditionsOK: true
12
```

```
13  RandomVar  Type     Min  Max  Parameters
14  ---------  -------  ---  ---  ----------
15  ANG        uniform  0    359
16  MAG        uniform  1.5  3.5
17
18  P/Tot  P/Loc  T/Total   T/Local  Variable/Var
19  -----  -----  --------  -------  ------------
20  19     9      196.77    72.15    DRIFT_VECTOR_ADD = 193,-0.6
21  20     0      219.42    24.08    DRIFT_VECTOR_ADD = 12,0.4
22  21     1      220.93    25.71    DRIFT_VECTOR_ADD = 12,0.4
23  22     2      222.95    27.91    DRIFT_VECTOR_ADD = 12,0.4
24  23     3      224.96    30.09    DRIFT_VECTOR_ADD = 12,0.4
25  24     4      226.46    31.73    DRIFT_VECTOR_ADD = 12,0.4
26  25     5      228.47    33.91    DRIFT_VECTOR_ADD = 12,-0.4
27  26     6      230.49    36.10    DRIFT_VECTOR_ADD = 12,-0.4
28
29  ================================================================
30  Most Recent Events (3):
31  ================================================================
32  [192.78]: Script Re-Start. Warp=0.48922, DelayStart=0.0, DelayReset=54.1
33  [44.80]: Script Re-Start. Warp=1.61692, DelayStart=0.0, DelayReset=18.9
34  [0.51]: Script Start. Warp=1, DelayStart=0.0, DelayReset=0.0
```

48.6.2 Lines 18-27: Recent Script Postings

Lines 18-27 show recent postings to the MOOSDB by the script. The first column shows the total postings so far for the script. The second column shows the index within the script. In the above example, there are ten elements in the script. The most recent posting on line 27, shows the script has been reset twice and the most recent posting is of the seventh element of the script (index 6). The third column shows the total time since script started, and the fourth column shows the time since the script was re-started. Note the time delay between lines 20 and 21, due to the delay_reset shown on line 9. The last column shows the actual variable value pair posted.

48.6.3 Lines 29-34: Recent Events

Lines 29-34 show recent events (other than event postings). In this case it shows the script has been started, and re-started twice. Notice the delay reset on line 32 is different than that on line 9. The delay reset time of 23.66 seconds shown on line 9 is the delay reset to be applied on the *next* reset.

48.7 Configuration Parameters for uTimerScript

The following parameters are defined for uTimerScript. A more detailed description is provided in other parts of this section. Parameters having default values are indicated.

Listing 48.61: Configuration Parameters for uTimerScript.

block_on: A comma-separated list of MOOS apps on which the script will block until seen in the list of DB_CLIENTS.

condition: A logic condition that must be met for the script to be un-paused. Section 48.3.2.

delay_reset: Number of seconds added to each event time, on each script reset. Legal values: any non-negative numerical value, or range of values separated by a colon. The default is zero. Section 48.5.2.

delay_start: Number of seconds, or range of seconds, added to each event time, on first pass only. Legal values: any non-negative numerical value, or range of values separated by a colon. The default is zero. Section 48.5.2.

event: A description of a single event in the timer script. Section 48.2.1.

forward_var: A MOOS variable for taking cues to forward time. The default is UTS_FORWARD). Section 48.3.3.

paused: A Boolean indicating whether the script is paused upon launch. Legal values: true, false. The default is false. Section 48.3.1.

pause_var: A MOOS variable for receiving pause state cues (UTS_PAUSE). Section 48.3.1.

rand_var: A declaration of a random variable macro to be expanded in event values. Section 48.4.2.

reset_max: The maximum amount of resets allowed. Legal values: any non-negative integer, or the string "nolimit". The default is "nolimit". NOTE: If resetting is desired, the reset_time parameter must also be changed from its default value of "none". Section 48.2.3.

reset_time: The time or condition when the script is reset Legal values: Any non-negative number, the string "none", or "all-posted", or equivalently, "end". The default is "none". Section 48.2.3.

reset_var: A MOOS variable for receiving reset cues. The default is UTS_RESET.

script_atomic: When true, a started script will complete if conditions suddenly fail. Legal values: true, false. The default is false.

script_name: Unique (hopefully) name given to this script. The default is "unnamed".

shuffle: If true, timestamps are recalculated on each reset of the script. Legal values: true, false. The default is true. Section 48.2.3.

status_var: A MOOS variable for posting status summary. The default is UTS_STATUS. Section 48.5.3

time_warp: Rate at which time is accelerated in executing the script. Legal values: any non-negative number. The default is zero. Section 48.5.

time_zero: The base time upon which script event times are based. The default is "script_start", the time at which uTimerScript is launched. The one alternative is "db_start", the time at which the MOOSDB was launched. The default is "script_start".

upon_awake: Reset or re-start the script upon conditions being met after failure ("n/a"). Section 48.2.3.

verbose: If true, progress output is generated to the console (true).

48.8 Publications and Subscriptions for uTimerScript

The interface for uTimerScript, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uTimerScript --interface or -i
```

48.8.1 Variables Published by uTimerScript

The primary output of uTimerScript to the MOOSDB is the set of configured events, but one other variable is published on each iteration, and another upon purposeful exit with the event=quit event configuration.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 48.6.
- EXITED_NORMALLY: A posting made when the script contains and executes a event=quit event, to let other applications know that the disconnection of uTimerScript is not a concern for alarm.
- UTS_STATUS: A status string of script progress. Section 48.5.3.

48.8.2 Variables Subscribed for by uTimerScript

The uTimerScript application will subscribe for the following four MOOS variables to provide optional control over the flow of the script by the user or other MOOS processes:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- EXITED_NORMALLY: When uTimerScript receives its own posting, it is assumed that all outgoing posts needed to be made before quitting have been received by the MOOSDB. Upon this receipt uTimerScript will quit. See Section 48.3.4.
- UTS_NEXT: When received with the value "next", the script will fast-forward in time to the next event. See Section 48.3.3.
- UTS_RESET: When received with the value of either "true" or "reset", the timer script will be reset. See Section 48.2.3.
- UTS_FORWARD: When received with a numerical value greater than zero, the script will fast-forward by the indicated time. See Section 48.3.3.
- UTS_PAUSE: When received with the value of "true", "false", "toggle", the script will change its pause state correspondingly. See Section 48.3.1.

In addition to the above MOOS variables, uTimerScript will subscribe for any variables involved in logic conditions, described in Section 48.3.2.

48.8.3 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uTimerScript --example
```

This will show the output shown in Listing 62 below.

Listing 48.62: Example configuration of the uTimerScript application.

```
1  ============================================================
2  uTimerScript Example MOOS Configuration
3  ============================================================
4  Blue lines:     Default configuration
5
6  ProcessConfig = uTimerScript
7  {
8    AppTick   = 4
9    CommsTick = 4
10
11   // Logic condition that must be met for script to be unpaused
12   condition      = WIND_GUSTS = true
13   // Seconds added to each event time, on each script pass
14   delay_reset    = 0
15   // Seconds added to each event time, on first pass only
16   delay_start    = 0
17   // Event(s) are the key components of the script
18   event          = var=SBR_RANGE_REQUEST, val="name=archie", time=25:35
19   // A MOOS variable for taking cues to forward time
20   forward_var    = UTS_FORWARD   // or other MOOS variable
21   // If true script is paused upon launch
22   paused         = false    // or {true}
23   // A MOOS variable for receiving pause state cues
24   pause_var      = UTS_PAUSE     // or other MOOS variable
25   // Declaration of random var macro expanded in event values
26   randvar        = varname=ANG, min=0, max=359, key=at_reset
27   // Maximum number of resets allowed
28   reset_max      = nolimit  // or in range [0,inf)
29   // A point when the script is reset
30   reset_time     = none     // or {all-posted} or range (0,inf)
31   // A MOOS variable for receiving reset cues
32   reset_var      = UTS_RESET     // or other MOOS variable
33   // If true script will complete if conditions suddenly fail
34   script_atomic  = false    // or {true}
35   // A hopefully unique name given to the script
36   script_name    = unnamed
37   // If true timestamps are recalculated on each script reset
38   shuffle        = true
39   // If true progress is generated to the console
40   verbose        = true     // or {false}
41   // Reset or restart script upon conditions being met after failure
42   upon_awake     = n/a      // or {reset,resstart}
43   // A MOOS variable for posting the status summary
44   status_var     = UTS_STATUS    // or other MOOS variable
45   // Rate at which time is accelerated in execuing the script
46   time_warp      = 1
47 }
```

48.9 Examples

The examples in this section demonstrate the constructs thus far described for the uTimerScript application. In each case, the use of the script obviated the need for developing and maintaining a separate dedicated MOOS application.

48.9.1 A Script for Generating 100 Random Numbers

The below script will generate 100 postings with random numbers preceded by an initial "start" posting, and followed by an "end" posting.

Listing 48.63: A uTimerScript configuration for generating 100 random numbers.

```
//------------------------------------------
ProcessConfig = uTimerScript
{
  rand_var    = varname=RND_VAL, min=0, max=50, key=at_post

  event = var=REPORT, val="start", time=0
  event = var=REPORT, val="accidents=$[RND_VAL], unique_id=$[TCOUNT]", time=0, amt=100
  event = var=REPORT, val="end", time=0
}
```

The above may result in postings like those below (take from an alog file):

```
8.817     REPORT         uTimerScript    start
8.817     REPORT         uTimerScript    accidents=35.234,unique_id=1
8.817     REPORT         uTimerScript    accidents=32.01,unique_id=2
8.818     REPORT         uTimerScript    accidents=26.982,unique_id=3
...
8.856     REPORT         uTimerScript    accidents=32.3,unique_id=95
8.857     REPORT         uTimerScript    accidents=13.324,unique_id=96
8.857     REPORT         uTimerScript    accidents=27.624,unique_id=97
8.858     REPORT         uTimerScript    accidents=28.901,unique_id=98
8.858     REPORT         uTimerScript    accidents=18.9,unique_id=99
8.859     REPORT         uTimerScript    accidents=35.72,unique_id=100
8.859     REPORT         uTimerScript    end
```

To make the numerical values integers, use the `snap=1` option (available in the next release after 19.8). Using `snap=0.1` will round to the nearest tenth and so on. For example:

```
  rand_var    = varname=RND_VAL, min=0, max=50, key=at_post, snap=1
```

48.9.2 A Script Used as Proxy for an On-Board GPS Unit

Typical operation of an underwater vehicle includes the periodic surfacing to obtain a GPS fix to correct navigation error accumulated while under water. A GPS unit that has been out of satellite communication for some period normally takes some time to re-acquire enough satellites to resume providing position information. From the perspective of the helm and configuring an autonomy mission, it is typical to remain at the surface only long enough to obtain the GPS fix, and then resume other aspects of the mission at-depth.

Consider a situation as shown in Figure 157, where the autonomy system is running in the payload on a payload computer, receiving not only updated navigation positions (in the form of `NAV_DEPTH`, `NAV_X`, and `NAV_Y`), but also a "heartbeat" signal each time a new GPS position has been received (`GPS_RECEIVED`). This heartbeat signal may be enough to indicate to the helm and mission configuration that the objective of the surface excursion has been achieved.

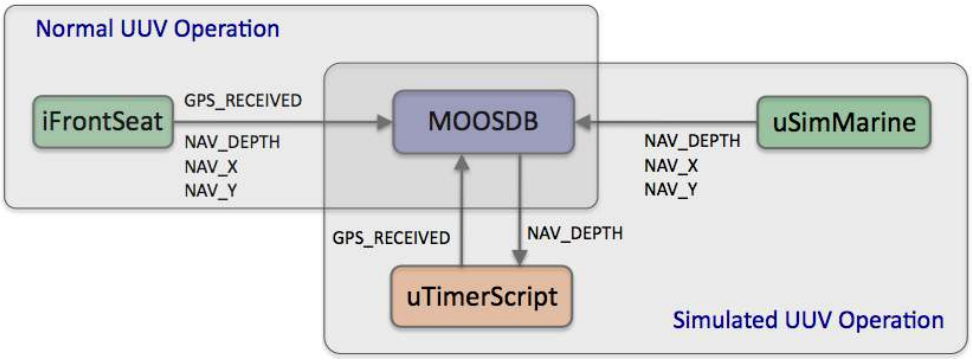

Figure 157: **Simulating a GPS Acknowledgment:** In a physical operation of the vehicle, the navigation solution and a GPS_UPDATE_RECEIVED heartbeat are received from the main vehicle (front-seat) computer via a MOOS module acting as an interface to the front-seat computer. In simulation, the navigation solution is provided by the simulator without any GPS_UPDATE_RECEIVED heartbeat. This element of simulation may be provided with uTimerScript configured to post the heartbeat, conditioned on the NAV_DEPTH information and a user-specified start delay to simulate GPS acquisition delay.

In simulation, however, the simulator only produces a steady stream of navigation updates with no regard to a simulated GPS unit. At this point there are three choices: (a) modify the simulator to fake GPS heartbeats and satellite delay, (b) write a separate simple MOOS application to do the same simulation. The drawback of the former is that one may not want to branch a new version of the simulator, or even introduce this new complexity to the simulator. The drawback of the latter is that, if one wants to propagate this functionality to other users, this requires distribution and version control of a new MOOS application.

A third and perhaps preferable option (c) is to write a short script for uTimerScript simulating the desired GPS characteristics. This achieves the objectives without modifying or introducing new source code. The below script in Listing 64 gets the job done.

Listing 48.64: A uTimerScript configuration for simulating aspects of a GPS unit.

```
1  //------------------------------------------------------------
2  // uTimerScript configuration block
3
4  ProcessConfig = uTimerScript
5  {
6    AppTick   = 4
7    CommsTick = 4
8
9    paused      = false
10   reset_max   = unlimited
11   reset_time  = end
12   condition   = NAV_DEPTH < 0.2
13   upon_awake  = restart
14   delay_start = 20:120
15   script_name = GPS_SCRIPT
16
17   event  = var=GPS_UPDATE_RECEIVED, val="RCVD_$[COUNT]", time=0:1
18 }
```

This script posts a GPS_UPDATE_RECEIVED heartbeat message roughly once every second, based on the event time "time=0:1" on line 17. The value of this message will be unique on each posting due to the $[COUNT] macro in the value component. See Section 48.4.1 for more on macros. The script is configured to restart each time it awakes (line 13), defined by meeting the condition of (NAV_DEPTH < 0.2) which is a proxy for the vehicle being at the surface. The delay_start simulates the time needed for the GPS unit to reacquire satellite signals and is configured to be somewhere in the range of 20 to 120 seconds (line 14). Once the script gets past the start delay, the script is a single event (line 17) that repeats indefinitely since reset_max is set to unlimited and reset_time is set to end in lines 10 and 11. This script is used in the IvP Helm example simulation mission labeled "s4_delta" illustrating the PeriodicSurface helm behavior.

48.9.3 A Script as a Proxy for Simulating Random Wind Gusts

Simulating wind gusts, or in general, somewhat random external periodic drift effects on a vehicle, are useful for testing the robustness of certain autonomy algorithms. Often they don't need to be grounded in very realistic models of the environment to be useful, and here we show how a script can be used simulate such drift effects in conjunction with the uSimMarine application.

The uSimMarine application is a simple simulator that produces a stream of navigation information, NAV_X, NAV_Y, NAV_SPEED, NAV_DEPTH, and NAV_HEADING (Figure 158), based on the vehicle's last known position and trajectory, and currently observed values for actuator variables. The simulator also stores local state variables reflecting the current external drift in the x-y plane, by default zero. An external drift may be specified in terms of a drift vector, in absolute terms with the variable USM_DRIFT_VECTOR, or in relative terms with the variables USM_DRIFT_VECTOR_ADD.

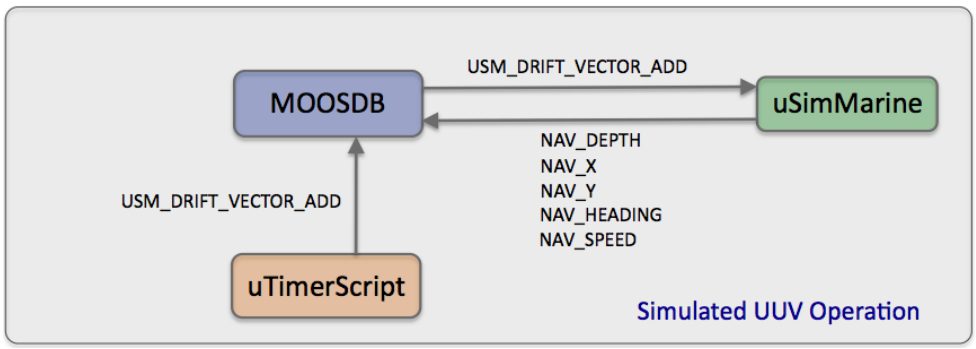

Figure 158: **Simulated Wind Gusts:** The uTimerScript application may be configured to post periodic sequences of external drift values, used by the uSimMarine application to simulate wind gust effects on its simulated vehicle.

The script in Listing 65 makes use of the uSimMarine interface by posting periodic drift vectors. It simulates a wind gust with a sequence of five posts to increase a drift vector (lines 18-22), and complementary sequence of five posts to decrease the drift vector (lines 24-28) for a net drift of zero at the end of each script execution.

Listing 48.65: A uTimerScript configuration for simulating simple wind gusts.

```
1  //------------------------------------------
```

```
 2  // uTimerScript configuration  block
 3
 4  ProcessConfig = uTimerScript
 5  {
 6    AppTick   = 2
 7    CommsTick = 2
 8
 9    paused        = false
10    reset_max     = unlimited
11    reset_time    = end
12    delay_reset   = 10:60
13    time_warp     = 0.25:2.0
14    script_name   = WIND
15    script_atomic = true
16
17    randvar = varname=ANG, min=0,   max=359, key=at_reset
18    randvar = varname=MAG, min=0.5, max=1.5, key=at_reset
19
20  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*0.2}", time=0
21  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*0.2}", time=2
22  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*0.2}", time=4
23  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*0.2}", time=6
24  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*0.2}", time=8
25
26  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*-0.2}", time=10
27  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*-0.2}", time=12
28  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*-0.2}", time=14
29  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*-0.2}", time=16
30  event = var=DRIFT_VECTOR_ADD, val="$(ANG),{$(MAG)*-0.2}", Time=18
31  }
```

The drift *angle* is chosen randomly in the range of $[0, 359]$ by use of the random variable macro `$[ANG]` defined on line 16. The peak *magnitude* of the drift vector is chosen randomly in the range of $[0.5, 1.5]$ with the random variable macro `$[MAG]` defined on line 17. Note that these two macros have their random values reset each time the script begins, by using the `key=at_reset` option, to ensure a stream of wind gusts of varying angles and magnitudes.

*The duration of each gust sequence also varies between each script execution. The default duration is about 20 seconds, given the timestamps of 0 to 18 seconds in lines 19-29. The `time_warp` option on line 12 affects the duration with a random value chosen from the interval of $[0.25, 2.0]$. A time warp of 0.25 results in a gust sequence lasting about 80 seconds, and 2.0 results in a gust of about 10 seconds. The time between gust sequences is chosen randomly in the interval $[10, 60]$ by use of the `delay_restart` parameter on line 11. Used in conjunction with the `time_warp` parameter, the interval for possible observed delays between gusts is $[5, 240]$. The `reset_time` parameter set to `end`, on line 10 is used to ensure that the script posts all drift vectors to avoid any accumulated drifts over time. The `reset_max` parameter is set to `"unlimited"` to ensure the script runs indefinitely.

49 pContactMgrV20: Managing Platform Contacts

49.1 Overview

Note: The pContactMgrV20 app is a substantially re-written version of its predecessor, pBasicContactMgr. The new version is meant to be largely backward compatible with the previous version. The new version has better support for larger streams of contact information over longer missions. It has better guards on memory management and should be more efficient in terms of CPU load. It also has much better support for filtering contacts with greater control over how different contacts may be handled uniquely. The latter is in support of apps in the Swarm Autonomy Toolbox which require these features from a contact manager.

The pContactMgrV20 application deals with information about other known vehicles in its vicinity. It is not a sensor application, but rather handles incoming "contact reports" which may represent information received by the vehicle over a communications link, or may be the result of on-board sensor processing. The primary use case involves the generation of alert messages corresponding to contact relative position criteria configured by the consumers of the alerts. The pContactMgrV20 posts to the MOOSDB summary reports about known contacts, which supports functionality of other MOOS applications.

The basic idea is shown below:

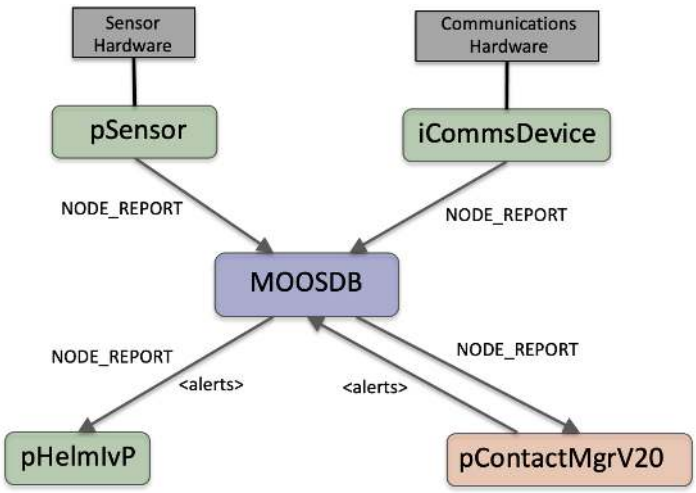

Figure 159: **The pContactMgrV20 Application:** The pContactMgrV20 utility receives NODE_REPORT information from other MOOS applications and manages a list of unique contact records. It may post additional user-configurable alerts to the MOOSDB based on the contact information and user-configurable conditions. The source of contact information may be external (via communications) or internal (via on-board sensor processing). The pSensor and iCommsDevice modules shown here are fictional applications meant to convey these two sources of information abstractly.

The contact manager is typically used in coordinating with the IvP Helm, but a key design goal of the contact manager is to remain helm agnostic. The design or configuration of the contact manager need not know anything about the helm or any other application for that matter. The configuration

or design of the helm, and helm behaviors, on the other hand needs to coordinate its function with the contact manager. Although contact manager alerts are simply MOOS variables, with no explicit linking or dependency on the helm or helm libraries, the alerts are commonly used by the helm to trigger the spawning of a behavior associated with a contact (Section 49.4). The configuration of the contact manager, for use with the helm, is also the job of the helm behaviors which post a MOOS variable to request dynamic alert configurations of the contact manager (Section 49.4.1).

49.2 Contact Alert Structure and Configuration

A *contact alert* is an internal data structure of the contact manager, which may be configured to comprise several types of alerts in a given mission. Configuration of alerts may be done through the mission (`.moos`) file, or they may be configured by other apps in the MOOS community that publish an alert request. The latter is the most common, but both may coexist. Here the data structure is defined, start-up configuration from the mission file, and run-time configuration through MOOS mail messages.

49.2.1 The Contact Alert Structure

The *contact alert* data structure contains the following fields:

The core components:

- `alert id`: A unique name for the alert.
- `alert_range`: A minimal range in meters that will cause an alert trigger.
- `cpa_range`: A larger range in meters that may trigger an alert before breaching the minimal range.
- `on_flag`: An alert posting, or `VarDataPair`, to be published when the alert has been triggered. Multiple flags may be provided.
- `off_flag`: An alert posting, or `VarDataPair`, to be published when the alert trigger criteria has switched off. Multiple flags may be provided.

The filter components:

- `ignore_name`: One or more contact names to be ignored.
- `match_name`: One or more contact names to be matched.
- `ignore_type`: One or more contact types to be ignored.
- `match_type`: One or more contact types to be matched.
- `ignore_group`: One or more contact groups to be ignored.
- `match_group`: One or more contact groups to be matched.
- `ignore_region`: A region where contacts will be ignored.
- `match_region`: A region where contacts will be ignored.

Range Components The `alert_range` component represents a threshold range to a contact, in meters. When a contact moves within this range, an alert will be generated. This is discussed in Section 49.3.1. The `cpa_range` component also represents a threshold range, in meters. Typically this

cpa_range is greater than the alert_range parameter. It is never less than the alert_range. When a contact is noted to be within the cpa_range, the contact and ownship trajectories are considered, to calculate the closest point of approach (CPA). If the CPA range is determined to be within the alert_range, an alert is generated, even if the present range between ownship and contact is outside the alert_range. This is also discussed in Section 49.3.1.

Flag Posting Components A flag describes a MOOS posting. It will be a pair, in the form of MOOS_VARIABLE = value. The value component may be string containing macros to allow the alert posting to contain information about the contact or ownship. This is discussed more in Section 49.2.3.

Filter Components The filter components allow the user to require that a contact meet a configured set of criteria in order for the alert to be applicable. This criteria can be based on either the contact name, platform type, group name, or region of the operation area where the contact presently resides. The *ignore* options indicate the contact should be ignored if one or more of the criteria hold. The *match* options indicate the contact should be ignored *unless* the match options are satisfied. This is discussed further in Section 49.5.

Alert Validity The alert ID, the alert range, and one or more postings are regarded as mandatory for the alert to be considered valid. All other fields are optional. The *cpa range* must be a value equal to or larger than the *alert range*. An invalid alert provided on startup will result in a configuration warning. An invalid alert received via an incoming mail message will result in a run warning.

49.2.2 Alert Configuration from the Mission File

Alerts may be configured in the mission file with the alert parameter. A single alert type may be defined over several lines, where each line contains the alert id of the alert type being configured. For example:

```
alert = id=<alert-id>, on_flag=<VarDataPair>, off_flag=<VarDataPair>
alert = id=<alert-id>, alert_range=<distance>,  cpa_range=<distance>
```

A simple example:

```
alert = id=K8, on_flag=SAY_MOOS=alarm, off_flag=SAY_MOOS=ok, alert_range=20, cpa_range=30
```

Or, equivalently:

```
alert = id=K8, on_flag=SAY_MOOS=alarm
alert = id=K8, off_flag=SAY_MOOS=ok
alert = id=K8, alert_range=20
alert = id=K8, cpa_range=30
```

49.2.3 Alert Configuration from Incoming Mail

Alert configuration can also be requested at run time, originating from another MOOS app via a message to BCM_ALERT_REQUEST. The contents of this message are identical to the one-line configuration used in a mission file shown above.

```
BCM_ALERT_REQUEST = id=k8, on_flag=SAY_MOOS=alert, alert_range=20, cpa_range=40
```

This kind of dynamic request is essential to the operation of the IvP Helm for contact related behaviors such as collision avoidance. This is discussed in Section 49.4.

49.2.4 Flag Macros

The on_flag component of an alert is a posting, or VarDataPair, that will be published by pContactMgrV20 to the MOOSDB when an alert is triggered. The data component of this posting may be a string, and may contain one or more supported macros to be expanded at the time of the post. For example the $[VNAME] macro contains the name of the contact related to the alert. This can be used to spawn a collision avoidance behavior as discussed in Section 49.4. The full set of supported macros is below:

- $[VNAME]: The name of the contact.
- $[X]: The position of the contact in local x coordinates.
- $[Y]: The position of the contact in local y coordinates.
- $[LAT]: The latitude position of the contact in earth coordinates.
- $[LON]: The longitude position of the contact in earth coordinates.
- $[HDG]: The reported heading of the contact.
- $[SPD]: The reported speed of the contact.
- $[DEP]: The reported depth of the contact.
- $[VTYPE]: The reported vessel type of the contact.
- $[UTIME]: The UTC time of the last report for the contact.

Multiple macro types may be used in the same on_flag or off_flag.

49.3 Alert Triggering

49.3.1 Trigger Criteria Base on Range and CPA Range

Alerts are triggered for all contacts based on range between ownship and the reported contact position. It is assumed that each incoming contact report minimally contains the contact's name and present position. An alert will be triggered if the current range to the contact falls within the distance given by `alert_range`, as in Contact-A in Figure 160.

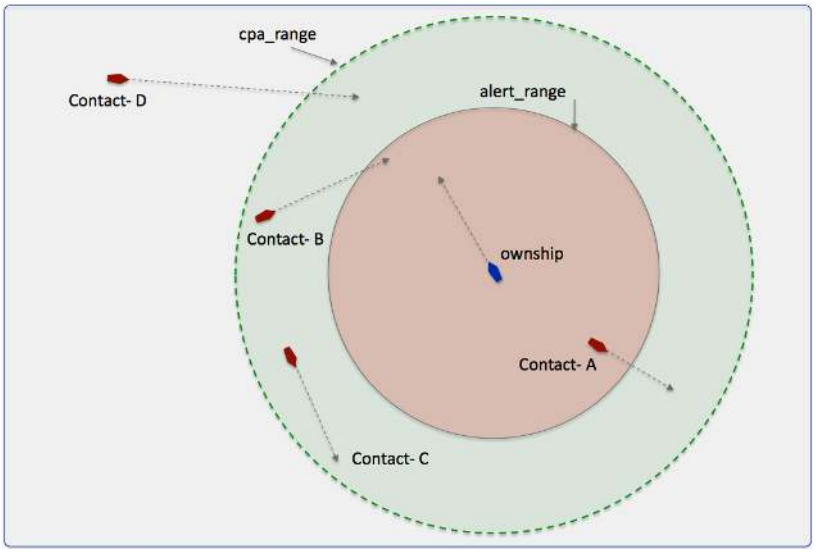

Figure 160: **Alert Triggers:** An alert may be triggered by the contact manager if the contact is within the `alert_range`, as with Contact-A. It may also be triggered if the contact is within the `cpa_range`, and the contact's CPA distance is within the `alert_range`, as with Contact-B. Contact-C shown here would not trigger an alert since its CPA distance is its current range and is not within the `alert_range`. Contact-D also would not trigger an alert despite the fact that its CPA with ownship is apparently small, since its current absolute range is greater than `cpa_range`.

An alert may also be configured with a second trigger criteria consisting of another range to contact, `cpa_range`. The range is enforced to be equal to or larger than the `alert_range`. If `cpa_range` is left unspecified, it will be set to the `alert_range`, effectively disabling this feature. When a contact is outside the `alert_range`, but within the `cpa_range`, as with Contact-B in Figure 160, the closest point of approach (CPA) between the contact and ownship is calculated given their presently-known position and trajectories. If the CPA distance falls below the `alert_range` value, an alert is triggered.

Turning Off an Alert and a Schmitt Trigger Once an alert has been triggered, presumably a posting has been made, specified with the `on_flag` parameter. Typically there is nothing further to do when the trigger criteria is no longer met. As discussed later in the case of the helm, a collision avoidance behavior will spawn with initial alert, and the behavior will be deleted later when the contact passes beyond the `cpa_range` of ownship. The only catch is that the same contact, with the same contact ID, may later change course and once again close range on ownship. A new alert will need to be triggered, and new behavior spawned. For this reason, when the contact opens beyond the `cpa_range`, the contact manager marks the contact/alert pair as *off*. If that contact does indeed

change course and approach ownship, a new alert will be trigger, a new posting will be published and a new collision avoidance behavior will be spawned. However, when events are triggered on/off based on a range threshold, it opens the door that a noisy sensor may create a thrashing of a rapid succession of alert on/off declarations and behavior spawning and deletions. In physics there is the notion of a *Schmitt trigger* which is a switch with a small positive feedback that increases the threshold for changing state in both directions. For example this could be accomplished in our situation by adding a few meters to the `cpa_range` as the contact is opening range, and subtracting a few meters from `cpa_range` when the contact is closing range to ownship. This isn't necessary however since the nature of the `cpa_range` is such that, for a contact to trigger an alert, it must not only be closing range to ownship, it must be on a heading that creates a CPA to ownship within the `range` setting (like Contact-B in Figure 160). As a contact is opening range, and turning off the alert, even if it were magically within the `cpa_range` an instant later, it would not have a CPA within the `range` circle until it comes about with a significant heading change. This creates natural Schmitt trigger effect.

49.3.2 Alert Triggering Internal Data Structures

The contact manager maintains a few key data structures for managing alert generation and other status updates:

- Contact Status Table
- Contact Alert Table
- Retired Contacts

As contact information is updated in the contact manager, the contact's position and timestamp of the most recent update is stored in the contact status table. This range is the simple linear distance. The CPA range projects both vehicles into the future to calculate the expected CPA range. The extrapolated range is based on an extrapolated contact position based on the current time and time of the last position update for that contact.

Each data structure is depicted below:

Contact Status

Contact = C134	Position/TimeStamp	range	cpa range	extrapolated range
Contact = C008	Position/TimeStamp	range	cpa range	extrapolated range
Contact = C344	Position/TimeStamp	range	cpa range	extrapolated range
Contact = C083	Position/TimeStamp	range	cpa range	extrapolated range

Alert Status

	Alert ID=K34	Alert ID=K07	Alert ID = K04
Contact = C134	**on**	off	**on**
Contact = C008	off	off	off
Contact = C344	**true**	**on**	off
Contact = C083	off	off	off

Retirement List

Retired Contacts	C091, C002, C023, C430, C055, C493, C588, C008, C331, C692

Figure 161: **Internal Data Structures:** Information is updated based on the arrival new contact information, updates on internally calculated status, and retiring of contacts out of range and not recently updated.

Based on the updated values of the contact status table, each contact is judged based on the alert range criteria for each alert. The contact alert table is updated accordingly. If a cell switches from `false` to `true`, the `on_flag(s)` for that alert and contact are posted. If the cell switches from `true` to `false`, the `off_flag(s)` are posted.

A contact may be *retired* for one of two reasons. (1) It may have been a long time since its last position update, longer than the threshold set by `contact_max_age`, which has a default of 600 seconds. (2) Or it may now be far away, beyond the range set by the parameter `reject_range` which has a default value of 2000 meters. If it is retired, the contact is reclaimed from the two tables, and is placed on the retirement list. This list also has a maximum size, pruned first-in-first-out.

49.3.3 Publications about Internal Status

There are several MOOS variables published by the contact manager that reflect some of the information in the above internal data structures:

- `CONTACTS_LIST`: A comma-separated list of names for all contacts in the contact status table.
- `CONTACTS_ALERTED`: A list of contacts that are currently in the alerted state, per alert ID.
- `CONTACTS_COUNT`: The number of currently alerted contacts.
- `CONTACTS_UNALERTED`: A list of contacts that are currently in the unalerted state, per alert ID.
- `CONTACTS_RETIRED`: A comma-separated list of names for retired contacts.
- `CONTACTS_RECAP`: A summary, name, range and report age, for contacts in the contact status table.

Here are some examples:

```
CONTACTS_LIST      = delta,gus,charlie,henry
CONTACTS_ALERTED   = (delta,avd)(charlie,avd)
```

```
CONTACTS_COUNT     = 2
CONTACTS_UNALERTED = (gus,avd)(henry,avd)
CONTACTS_RETIRED   = bravo,foxtrot,kilroy
CONTACTS_RECAP     = name=ike,age=11.3,range=193.1 # name=gus,age=0.7,range=48.2 #
                     name=charlie,age=1.9,range=73.1 # name=henry,age=4.0,range=18.2
```

All variables are only published when changed. The CONTACTS_RECAP variable will very likely change content on each iteration since it contains the numerical ranges to moving contacts from a moving ownship. If published and logged on every iteration this will generate large amounts of perhaps unnecessary log data. By default, this variable will publish once per second, but this is changeable with the contacts_recap_interval. It may be set to any non-negative value, or the string "off" to turn it off completely. Of course if log file size is a concern, the frequency of logging can always be adjusted in the pLogger settings.

A fair question might be - why retain in memory any information related to contacts that currently do not have an active alert. As will be discussed in Section 49.6.1 and Section 49.6.2, the contact manager generates other reports that may include contacts that may not presently warrant an alert.

49.4 Contact Manager Coordination with the Helm

The contact manager was designed primarily to support the IvP Helm, but also designed in a manner to make as few assumptions as possible about the helm implementation. The helm is a contact manager user, a consumer of contact manager alerts. It is the helm's responsibility to submit registrations to the contact manager for alerts, with the alert registration defining the conditions and format for the desired alert. The contact manager will blindly abide, and will generate alerts of the requested format, when the requested range conditions and filter conditions (if any) are met. This process is described in more detail in this section. More details on helm behavior templates, updates and behavior spawning can be found in the helm documentation.

49.4.1 Helm Registration for Alerts

When the helm starts up, it populates a set of behaviors. Some behaviors are static, created at launch time, and persist throughout the mission. Other behaviors are configured as templates, placeholders for perhaps many future behaviors of that type. This is the case for contact-related behaviors such as collision avoidance. As Figure 162 shows below, the behavior template is implemented to post an alert registration message upon startup.

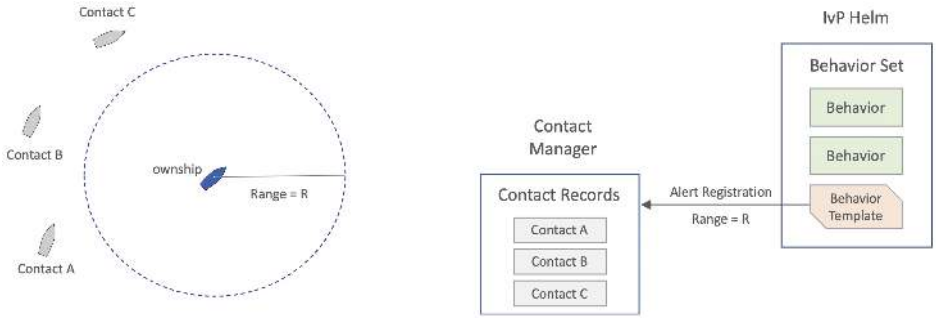

Figure 162: **Alert Registration**: When the helm starts, contact-related behaviors, configured as templates, will post a registration to the contact manager. The registration contains information about the desired format and threshold criteria for future alerts.

The alert registration format was discussed previously in Section 49.2.3. For helm contact-related behaviors, the `on_flag` specifies a posting to the MOOS variable named in the behavior's `updates` parameter configuration, e.g., `CONTACT_INFO` in the below example.

```
BCM_ALERT_REQUEST = id=avoid, alert_range=20, cpa_range=40, ignore_type=cargo \
                    on_flag=CONTACT_INFO=name=avd_$[VNAME] # contact=$[VNAME]
```

The rest of the alert request specifies the alert and cpa range values so the contact manager knows the conditions this behavior template is expecting to receive alert postings, i.e., publications to `CONTACT_INFO` in this case. After this start up process, the helm, with respect to this behavior is in listen-and-wait mode, until an alert is received.

49.4.2 Helm Action Upon Receiving an Alert

Figure 163 below depicts alert sequence of events. First a contact crosses a range threshold relative to ownship. Second, this is detected and noted in the contact manager. This results in an alert posting of a format that a behavior template in the helm is expecting. Finally the helm will spawn a behavior dedicated exclusively to dealing with this contact.

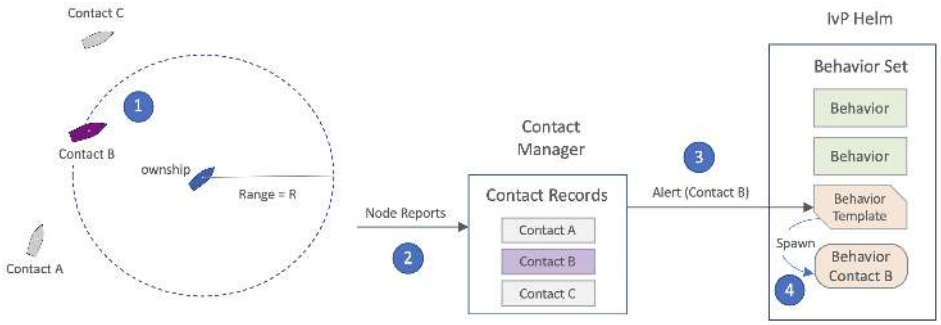

Figure 163: **Alert to Spawning**: (1) A contact closes range to ownship, crossing a range threshold. (2) The contact manager makes note and changes the internal record associated with the contact. (3) An alert it generated and received by the helm. (4) The helm spawns a new contact-related behavior dedicated to this contact.

The posting, following the `BCM_ALERT_REQUEST` example above, will have the following form:

```
CONTACT_INFO=name=avd_$[VNAME] # contact=$[VNAME]
```

The contact manager will use the contact name, e.g., `vessel_23`, to expand the `$[VNAME]` macros and the posting will look more like:

```
CONTACT_INFO=name=avd_vessel_023 # contact=vessel_023
```

See the helm documentation for more information on behavior updates and template behaviors. In short, the helm will first check to see if a behavior already exists that (1) is listening for parameter updates through the MOOS variable `CONTACT_INFO`, and (2) has the unique name of `avd_vessel_23`. If so, it would regard the remainder of the `CONTACT_INFO` message as an update to the existing behavior. If a behavior by that name does not already exist, and there is a behavior template that is listening for parameter updates on the MOOS variable `CONTACT_INFO`, a new behavior is then spawned with the name `avd_vessel_023`, and the remainder of the `CONTACT_INFO` message is treated as an update to this newly spawned behavior.

49.4.3 Helm Action Upon Alert Retirement

When a contact opens range to ownship sufficiently far, beyond the `cpa_range`, the contact manager will alter its internal state associated with the contact/alert pair. See the alert status table in Figure 161. From the helm's perspective no further action is required. Contact-related behaviors are implemented to *complete*, and be promptly deleted by the helm, once the contact goes beyond a

certain range. As you might guess, this range is automatically set to be equal to the `cpa_range` the contact manager uses. So, roughly at the same time, the contact manager denotes the alert for this contact to be *off*, and the helm lets the contact behavior complete and be removed from memory. If the contact and ownship should later cross paths again, the contact manager would simply generate a new alert and the helm would spawn a new behavior.

An example mission is available for showing the use of the contact manager and its coordination with the helm to spawn behaviors for collision avoidance. This mission is `m2_berta` and is described in the IvP Helm documentation. In this mission two vehicles are configured to repeatedly go in and out of collision avoidance range, and the contact manager repeatedly posts alerts that result in the spawning of a collision avoidance behavior in the helm. Each time the vehicle goes out of range, the behavior completes and dies off from the helm and is declared to the contact manager to be resolved.

49.5 Exclusion Filters

In certain missions it is desirable to treat different contacts differently. This may be based on the contact platform type, e.g., sailboats versus motorboats. It may be based on contact group affiliation, e.g., red team versus blue team. It may be based on contact location in the operation area, or perhaps simply relative to ownship. One example that comes up frequently in field trials is the desire for the robot to ignore the nearby test boat holding the test operators.

There are a number of options for configuring the contact manager to achieve the desired effect. In all cases, the configuration may be done at the alert level, or at the contact manager level. At the contact manager level, one may simply want to ignore all sailboats. At the alert level it's also possible that the helm may have one collision avoidance behavior configured and tuned for motorboats and should ignore sailboats, and another behavior configured and tuned for sailboats and should ignore motorboats.

Filters are also generally of a *match* type or an *ignore* type. For example the contact manager may be configured to ignore all sailboats by specifying an ignore filter based on sailboats. Or it may be configured to only regard sailboats by setting a match filter for sailboats. If multiple ignore filters are provided, a contact will be ignored if any one ignore filter applies to the contact. If multiple match filters are provided, a contact will be ignored only if all match filters do not apply to the contact. If any criteria fail, the contact will be ignored. For example, for a given sailboat, if sailboats are explicitly named in a match filter, but also named in an ignore filter, it will be ignored.

49.5.1 Exclusion Filters Based on Contact Name, Type or Group

Name, type and group filters are available at the contact manager level with the following pContactMgrV20 configuration parameters:

- match_name: A comma-separated list of match names
- match_group: A comma-separated list of match groups
- match_type: A comma-separated list of match types
- ignore_name: A comma-separated list of names to ignore
- ignore_group: A comma-separated list of groups to ignore

- ignore_type: A comma-separated list of types to ignore

Here are some examples:

```
match_name   =  henry
match_group  =  blue_team
match_type   =  motor_boat, crew_shell
ignore_name  =  abe,ben, cal
ignore_group =  red_team, work_boat
ignore_type  =  sail_boat
```

If a contact survives the filters at the contact manager level, it still must survive any filters that may have been declared at the individual alert level. The match_group, match_type, ignore_group, and ignore_type components work the same, but are embedded in an alert configuration. As a pContactMgrV20 configuration parameter, it may look like:

```
  alert = id=K8, on_flag=SAY_MOOS=alarm, off_flag=SAY_MOOS=ok, ignore_name=hal   \
          alert_range=20, cpa_range=30, match_group=blue_team
```

As an alert configured by a mail request to the contact manager, it may look like:

```
BCM_ALERT_REQUEST = id=k8, onflag=SAY_MOOS=alert, alert_range=20, cpa_range=40, \
                    match_group=blue_team
```

If a contact does not report its type information, it may be rejected if (a) there is at least one ignore_type specified, *and*, (2) the contact manager configuration parameter strict_ignore is set to true, which is the default value. The same holds true if the contact has missing group information.

49.5.2 Exclusion Filters Based on Contact Position Relative to a Region

Like the group and type filters, region filters are available at the contact manager level with the following pContactMgrV20 configuration parameters:

- match_region: A convex polygon
- ignore_region: A convex polygon

Multiple regions of each kind may be specified. These regions may be contiguous, overlapping or non-overlapping. Thus non-convex regions can be specified by splitting into a set of convex regions. For match regions, the contact must be in at least one region or else it is filtered out. For ignore regions, if the contact is any one (or more) regions, it will be filtered out. If the contact fails according to any kind filter, it fails overall. Here are some examples:

```
match_region  =  pts={10,-40:-50,-80:-30,-140:50,-100}
ignore_ignore =  pts={50,-60:80,-70:120,-110:100,-150:60,-160:30,-110}
```

If a contact survives the region filters at the contact manager level, it still must survive any filters that may have been declared at the individual alert level. The `match_region`, and `ignore_region` components work the same, but are embedded in an alert configuration. As a `pContactMgrV20` configuration parameter, it may look like:

```
alert = id=K8, on_flag=SAY_MOOS=alarm, off_flag=SAY_MOOS=ok,    \
        alert_range=20, cpa_range=30, match_group=blue_team     \
        match_region=pts={10,-40:-50,-80:-30,-140:50,-100}
```

As an alert configured by a mail request to the contact manager, it may look like:

```
BCM_ALERT_REQUEST = id=k8, onflag=SAY_MOOS=alert, alert_range=20, cpa_range=40, \
                    match_region=pts={10,-40:-50,-80:-30,-140:50,-100}
```

Note: Whereas type and group information tend not to change for a contact during operation, its position in or out of a region may very well change. If a contact that was previously being track, with an entry in memory, fails a region filter, it is also wiped from memory in the contact manager. This is to prevent contact information from becoming stale in memory.

49.5.3 Exclusion Filters Based on Contact Range to Ownship

Last but not least, the simplest exclusion filter is to ignore any contact substantially far from ownship. Using the `reject_range` parameter this range can be set from the configuration block of `pContactMgrV20`:

```
reject_range = 2000
```

The default value is 2000 meters. To disable this filter, either set it to a very large value, or give it the string value `"off"`. Keep in mind that the `reject_range` serves a second purpose, as discussed in Section 49.3.2, as a threshold for retiring a contact from the contact manager's internal data structures. Retiring occurs if either (a) the contact range exceeds the `reject_range`, or (b) if contact reports are stale beyond a certain time. By turning off the `reject_range`, contact retirement can only be caused by staleness. Presently there are no warnings produced if the `reject_range` is lower than any of the alert ranges. This check will likely be added in future releases.

49.6 Additional Contact Manager Reports

49.6.1 Closest Contact Reports

The contact manager will publish information about the contact currently with the closest range to ownship. It will publish four variables, `CONTACT_CLOSEST` naming the vehicle that is closest, `CONTACT_CLOSEST_TIME` indicating the time at which the named contact became the closest contact, `CONTACT_CLOSEST_RANGE` indicating the actual current range to the closest contact, and `CONTACT_RANGES` with an ordered list of ranges for all known contacts. For example:

```
CONTACT_CLOSEST = "vessel_0971"
CONTACT_CLOSEST_TIME = 7725262005.34
CONTACT_CLOSEST_RANGE = 48.32
CONTACT_RANGES = 48.32,56.3,112.08
```

The first two variables above in practice will not change very often, perhaps on the order of once per several seconds. The contact range however will likely change on every iteration. While negligible in terms of MOOS bandwidth, it may be annoying in terms of logged data, so it can be configured to be off, by setting the parameter `post_closest_range=false`. The same is true for the variable `CONTACT_RANGES`, which can be disabled by setting `post_all_ranges=false`. The default value for both is true. The rate of data logged can also always be adjusted in the `pLogger` configuration.

49.6.2 Customizable Contact Reports Based on a Range Threshold

The contact manager may optionally be configured to produce customizable contact reports based on a range threshold. These reports are simply a list of contacts. A subscriber to the contact manager could otherwise glean this information via the `CONTACTS_RECAP` message which contains the full list of all contacts and ranges. The customizable contact report described here is simply a more convenient low-bandwidth alternative option.

Format of the Contact Reports The format of the report is simply a comma-separated list of contacts. For example:

```
CONTACTS_050 = vid_0027, vid_0197, vid_8645
```

This report is published only when the list of contacts changes, so it is considerably less bandwidth and easier to parse than the `CONTACTS_RECAP` message if the user simply wants to know which contacts are currently within 50 meters. Note the variable name, `CONTACTS_050`, was chosen with the `_050` suffix as an easy reminder of the meaning of the list. This is just a useful convention, and not enforced in any way. Selecting the name of the variable, and range is described next.

Requesting a Customized Contact Report The MOOS variable used for posting the report, and the range threshold, are specified via the incoming MOOS variable `BCM_REPORT_REQUEST`. For example:

```
BCM_REPORT_REQUEST = "var=CONTACTS_050, range=50, group=blueteam"
BCM_REPORT_REQUEST = "var=CONTACTS_100, range=100"
BCM_REPORT_REQUEST = "var=CONTACTS_200, range=200, type=sailboat, refresh=true"
```

The order in the components in the string do not matter. It is recommended to use a MOOS variable naming convention, such as `CONTACTS_050` above, to indicate the meaning of the report. The variable name and range components are mandatory, while the group and type components are optional. If the group or type component is used, then a contact must satisfy all criteria to be included on the contact list for that report. When the `refresh=true` component is detected, a new report will be generated immediately, even if the content of the report has not changed from a previous posting.

Timeout of Custom Reports While a valid custom report request will generate at least one custom report immediately, the generation of further reports will cease after some point in time unless further requests arrive. By default the duration is 10 seconds, but may be changed to say 60 seconds with the following parameter: `range_report_timeout=60`.

49.7 Guarding Against Unbounded Memory Growth

In some systems it may be possible for the sensors or sensor system to be generating many contacts, with unique IDs, over time. This problem may arise simply from long duration (days+) missions in high contact density situations. It may also be partly a result of a sensor that easily loses track on a contact, generating several IDs for the same contact as it approaches and ultimately departs the ownship vicinity.

The contact manager reasons about two types of data, tied to (a) active contacts, and (b) retired contacts. The key data structures were shown in Figure 161. In this section, the contact manager policies to guard against unbounded growth are described.

49.7.1 Managing Retired Contacts

As a contact is retired, it is moved to an internal list of retired contacts, and this list is published as `CONTACTS_RETIRED`. This is useful for debugging and verifying contact manager operation. With shore test missions having only a handful of contacts, this list can hold all contacts ever retired. In more realistic missions, the list needs to be managed.

As discussed in Section 49.4.3, the newly retired contact will stay on the `CONTACTS_RETIRED` list until it is pushed off, first-in first-out, when newer retired contacts are added. The size of the list is set by the `max_retired_history` parameter to be in the range $[1, 50]$, ensuring that even this list does not grow unbounded. The default is 5. The sequence of steps at the end of each iteration of the contact manager is the following:

- Add newly retired contacts to the retired list
- Publish the retired list to `CONTACTS_RETIRED`
- If needed, prune the retired list to size `max_retired_history`

This means that (a) the list published to `CONTACTS_RETIRED` on some iterations may be longer than `max_retired_history`, and (b) every contact that is ever retired will show up on at least one publication to `CONTACTS_RETIRED`.

49.7.2 Managing Active Contacts into Retirement

To have an absolute guard against unbounded memory, the contact manager has a strict upper limit on the number of contacts, configured with the parameter, `max_contacts`. This is set to 500 by default. Although most systems have enough memory to probably handle many thousands of contacts, 500 is probably generously high. The user takes responsibility to set this value.

Normal Retirement As ownship and contacts pass each other, coming in and out range, or comms and sensor range, contact retirement proceeds simply. At the end of each contact manager

iteration, after receiving all incoming contact updates and generating any applicable alerts or status postings, the contact manager will retire any contact that meets the following criteria:

- The contact is beyond the range set by the parameter `reject_range`, or
- The timestamp of the contact's most recent report is stale beyond the time set by the parameter `max_contact_age`.

The default value for `reject_range` is 2000 meters, and the default value for `max_contact_age` is 60 seconds. If, at this point, the number of contacts is still greater than `max_contacts`, then contacts will need to be forced into retirement, or *culled*.

Forced Retirement If normal retirement doesn't achieve the desired result of reducing the total contacts to be less than or equal to `max_contacts`, the following two additional steps will be taken:

- A tighter maximum age is applied to each contact's staleness. Contacts with a timestamp larger than *half* the time set by the parameter of the `max_contact_age` will be removed.
- Finally, if the above step leaves more culling to be done, contacts will be removed based range, until only the N closest contacts remain, where N is equal to `max_contacts`.

The Consequences of Forced Retirement A contact that has been forced into retirement may have been in the alerted state with respect to one or more alert types. The forced retirement means it did not have a normal transition from being alerted to unalerted. If there was an `off_flag` configured for the alert, it will not get posted as normal. This should be considered when/if using an `off_flag`.

If a culled contact was in an alerted state for one or more alert types, it might reappear a short time later as the contact manager again receives information about this contact. In fact, if the contact happens to be the (N+1)th contact in terms of range, it conceivably could reappear and be culled on successive iterations. What are the consequences of this in terms of alerts? The alert requests used by the contact-related behaviors are designed such that successive identical alerts are harmlessly redundant.

For example, consider the `on_flag` from the earlier example:

```
CONTACT_INFO=name=avd_vessel_023 # contact=vessel_02
```

If a contact behavior did not exist, one would be spawned as normal. If one had already been spawned, the helm would direct this update to that behavior (re)setting the `name` and `contact` parameters to the same value as before. In either case the regeneration of the alert is either doing its job or is harmless.

A culled contact may however affect the accuracy of customized range reports. If the user has asked to be notified of all contacts within say 10,000 meters, but there are more than `max_contact` vessels within this range, then the range report may not be accurate.

49.8 Disabling and Re-enabling Contacts

Note: This feature was introduce after Release 24.8 for the following release.

In certain USV applications, especially with an operator in the loop either on-board or remotely, contacts arrive in the contact manager that may be identified as being of no risk for collision. In these cases, the operator may wish to communicate to the autonomy system, that no behaviors should be spawned for the contact, i.e., the contact should be *disabled*. Likewise the operator may change their mind and may subsequently request that a contact be *enabled* or re-enabled.

This augmentation to contact manager is mostly implemented through changes to the IvP Helm and behaviors. The contact manager is simply one way to initiate the disabling or enabling of a behavior. In the below, first the mods to the helm and behaviors are described, followed by the manner in which the contact manager can be used for this purpose.

49.8.1 The Helm Role in Disabling and Enabling Behaviors

A new state, the *disabled* state, was added added generally to IvP behaviors. When a behavior is disabled, it will simple not participate in the helm. A disabled behavior is not deleted from the helm, in case there is a subsequent request to re-enable the behavior. By default a behavior cannot be disabled. Only certain contact behaviors can be disabled. For now, these are only the collision avoidance behaviors. To allow a behavior to be disabled, the below configuration parameter is used:

```
can_disable = true
```

If a helm behavior is not designed to allow it to be disabled, the above setting will produce a helm configuration warning.

The helm registers for the variable `BHV_ABLE_FILTER`, which can contain messages like the following:

```
BHV_ABLE_FILTER = action=disable, contact=ID345
BHV_ABLE_FILTER = action=enable, contact=ID345
BHV_ABLE_FILTER = action=disable, bhv_type=AvoidCollision
BHV_ABLE_FILTER = action=disable, contact=ID345, bhv_type=AvoidCollision
BHV_ABLE_FILTER = action=disable, contact=ID345, gen_type=safety
```

In the first case, simply the contact ID is given. This will match to a behavior with the name `foo_ID345`. Note, this will also apply to behavior a behavior named `bar_ID345`, as with any other behavior associated with a given contact. In the second case, the action is to (re-enable) the same behavior. In the third case, the `AvoidCollision` behavior will be disabled for all such behaviors for all contacts. In the fourth case, the `AvoidCollision` behavior will be disabled only if contact name ends in `ID345`. In the fifth case, all safety behaviors will be disabled if the contact ends in `ID345`.

The helm stores an ordered list of filter messages, based on time of arrival. On each iteration, this ordered list is applied to all instantiated behaviors. Applying a disable message to a behavior that is already disabled, has no effect. But by applying the whole recent list to all behaviors on each iteration in ensures that disable requests that arrive in the helm before a behavior is spawned, will be applied as soon as the behavior is spawned.

49.8.2 The Contact Manager Role in Disabling Contact/Behaviors

The contact manager can broker disabling and (re)enabling helm behaviors first by naming a MOOS variable for receiving such requests:

```
disable_var = XYZ_DISABLE_TARGET
```

If no variable is configured as above, the contact manager will not be able to receive requests for disabling. Note also the subtle difference between the two terms:

- *disabling a contact*: this implies disabling all disableable helm behaviors associated with a given contact.
- *disabling a behavior*: this implies disabling a particular disableable helm behavior.

In practice the two are equivalent, since typically there is only one disableable (collision avoidance) behavior configured for the helm. This may not be true when/if other contact related behaviors are augmented to allow disabling.

Simple Disabling by Contact ID The simplest way to use the contact manager to disable a behavior is to send mail to the `disable_var` with the contact ID. If the ID is, for example, `ID345`, and the `disable_var` is `XYZ_DISABLE_TARGET`, then any app making the following post will do

```
XYZ_DISABLE_TARGET = ID345
```

This result in the contact manager will be to publish the message the helm needs to receive for the desired effect.

```
BHV_ABLE_FILTER = action=disable, contact=ID345
```

Note if the behavior in the helm has the name, say `avd_ID345`, the match will be made.

Simple Re-Enabling by Contact ID The simplest way to use the contact manager to (re)enable a behavior is to send mail to the `enable_var` with the contact ID. If the ID is, for example, `ID345`, and the `enable_var` is `XYZ_REENABLE_TARGET`, then any app making the following post will do

```
XYZ_REENABLE_TARGET = ID345
```

This result in the contact manager will be to publish the message the helm needs to receive for the desired effect.

```
BHV_ABLE_FILTER = action=enable, contact=ID345
```

Precision Disabling by Contact ID and Behavior Type If the contact manager user wishes to be more precise about exactly which behavior to disable, then following message should be used:

```
XYZ_DISABLE_TARGET = action=disable, contact=ID345
```

This format follows the format used by the helm for input expected on the `BHV_ABLE_FILTER`. Essentially, if the contact manager sees a disable message in the form of something other than a simple contact ID, then it will simply pass through the message to the helm in the format expected by the helm.

Disable and Enable History Held by the Contact Manager The contact manager keeps an ordered list of disabled and enabled contact IDs. This allows the contact manager to know (a) the most recently disabled contact ID, (b) the most recently enabled contact ID, and (c) the total disabled contacts.

Disable/Enable Events, Flags and Macros There are three supported event flags tied to the disabling or enabling a contact:

- The able_flag is posted whenever a contact is enabled or disabled.
- The disable_flag is posted whenever a contact is disabled.
- The enable_flag is posted whenever a contact is enabled.

Like most other flags in other apps and behaviors, the value of the flag is a MOOS variable and value. The value component can contain anything the user wants. The following macros are also supported in the value part of the flag:

- $[RECENT_DISABLE]: The name of the most recently disabled contact. Or "none" if no contact has been disabled.
- $[RECENT_ENABLE]: The name of the most recently enabled contact. Or "none" if no contact has been (re)enabled.
- $[TOTAL_DISABLED]: The total number of currently disabled contacts.
- $[ALL_DISABLED]: A comma-separated list of all currently disabled contacts, or "none" if no contacts are disabled.

Note: the macros described in Section 49.2.4, for use in the on_flag and off_flag, are not available when posting the able_flag, disable_flag, or enable_flag.

49.8.3 Early Warnings

An *early warning* is an event about a contact that can be configured to post a flag to bring the contact to the attention of an operator. Presumably the early warning is generated with enough time for the operator to have the chance to disable a contact if they wish. The warning is configured with a few optional parameters, with the first being the amount of time, in seconds, at which a warning event should happen:

```
early_warning_time = 10
```

The time interval, 10 seconds in this example, is the amount of time before a contact reaches a range to ownship at which an alert would otherwise be posted. For example, if a collision avoidance behavior in the helm has arranged to receive an alert at 40 meters, the early warning will event will happen 10 seconds before reaching the range of 40 meters. The *rate of closure*, based on both ownship and contact position, speed and relative bearing, is applied to the early warning time and current range, to ensure that the warning is roughly 10 seconds prior to breaching the alert range, assuming both ownship and contact remain on their current trajectories. If ownship and/or contact change their speed or headings, the rate of closure is automatically updated and the early warning event will shift correspondingly.

The event that occurs with the early warning is configurable by the user with another parameter for pContactMgrV20, using the following event flag parameter:

```
early_warning_flag = MOOS_VAR = value
```

For example:

```
early_warning_flag = CMGR_WARNING = Warning=$[CONTACT], at $[RNG], at $[UTC]
```

Like most other flags in other apps and behaviors, the value of the flag is a MOOS variable and value. The value component can contain anything the user wants. The following macros are also supported in the value part of the early warning flags:

- `$[CONTACT]`: The name of contact.
- `$[RNG]`: The range to the contact based on the last node report received from the contact, or about the contact.
- `$[RNG_EXT]`: The extrapolated range to the contact based on the last node report received from the contact, or about the contact, and the time since the report was received.
- `$[UTC]`: The current UTC time, in seconds.
- `$[CPA]`: The closest of approach to the contact for this encounter. Mostly useful in a cease_warning_flag.
- `$[ROC]`: The current rate of closure to the contact based on ownship and contact position, speed and relative bearing.

Note: the macros described in Section 49.2.4, for use in the on_flag and off_flag, are not available when posting the able_flag, disable_flag, or enable_flag, and the macros defined in Section 49.8.2 are not available when posting with early_warning_flag or cease_warning_flag.

When a contact generates an early warning, the range at that moment is noted and stored relative to the contact. When or if the contact passes and opens range to ownship and once again beyond that stored range, a cease_warning_flag is published. To guard against thrashing, the cease warning range is `1.05` times the stored range. A warning flag may look like:

```
cease_warning_flag = CMGR_WARNING = off
```

Or, for example, if there is a desire to keep track of the CPA for all encounters, the following could be added. There is no limit to the number of warning flags that can be configured.

```
cease_warning_flag = ENCOUNTER_CPA = $[CPA]
```

Finally, an early warning range visual can be enabled. This is a viewable circle with ownship at the center, rendering the range at which a contact will trigger an early warning. To enable this:

```
early_warning_radii = true
ewarn_radii_color = yellow
```

The default for the former is `false`, and the default for the latter is `"yellow"`.

49.9 Deferring to Earth Coordinates over Local Coordinates

Incoming node reports contain the position information of the contact and may be specified in either local x-y coordinates, or earth latitude longitude coordinates or both. By default pContactMgrV20 uses the local coordinates for calculations and the earth coordinates are merely redundant. It may instead be configured, with the contact_local_coords parameter, to have its local coordinates set from the earth coordinates if the local coordinates are missing:

```
contact_local_coords = lazy_lat_lon
```

It may also be configured to always use the earth coordinates, even if the local coordinates are set:

```
contact_local_coords = force_lat_lon
```

The default setting is `verbatim`, meaning no action is taken to convert coordinates. If either of the other two above settings are used, the latitude and longitude coordinates of the local datum, or `(0,0)` point must be specified in the MOOS mission file, with LatOrigin and LongOrigin configuration parameters. (They are typically present in all mission files anyway.)

49.10 Terminal and AppCast Output

The status of the contact manager may be monitored from from an open console window where pContactMgrV20 is launched. Example output is shown below in Listing 66.

Listing 49.66: Example pContactMgrV20 terminal and appcast output.

```
1  ================================================================
2  pContactMgrV20 abe                                      0/0(377)
3  ================================================================
4  X/Y from Lat/Lon:   false
5  Contact Max Age:    3600
6  Reject Range:       2000
7  BCM_ALERT_REQUESTs: 2
8  DisplayRadii:       false
9
10 Alert Configurations (1):
11 ---------------------
12 1 ID: avdcol_
13   Alert Ranges: 35/45
14   Source: pHelmIvP
15   OnFlag: CONTACT_INFO=name=$[VNAME] # contact=$[VNAME]
16
17 Alert Status Summary:
18 ----------------------
19        List: ben,cal,deb
20     Alerted: (deb,avdcol_)
21     Retired:
22       Recap: vname=ben,range=82.72,age=0.83 # vname=cal,range=73.42,age=0.66 #
23              vname=deb,range=26.32,age=0.81
24
25 Contact Status Summary:
```

```
26  ------------------------
27  Contact      Range     Alerts     Alerts
28               Actual    Total      Active
29  -------      ------    ------     ------
30  ben            82.7    0          0
31  cal            73.4    1          0
32  deb            26.3    3          1
33
34  Custom Contact Reports:
35  ------------------------
36  VarName      Range      Group      VType      Contacts
37  -------      -----      -----      -----      --------
38
39  ======================================================================
40  Most Recent Events (4):
41  ======================================================================
42  [179.51]: CONTACT_INFO=name=deb # contact=deb
43  [72.09]: CONTACT_INFO=name=deb # contact=deb
44  [71.06]: CONTACT_INFO=name=deb # contact=deb
45  [33.44]: CONTACT_INFO=name=cal # contact=cal
```

On line 2, the `"0/0"` indicates there were no configuration warnings and no run-time warnings (thus far). The `"(377)"` represents the iteration counter of pContactMgrV20. Line 4 indicates that contact position in local coordinates is being derived directly from reported local coordinates in incoming node reports (Section 49.9). Line 5 shows the maximum age of a contact report before the contact is dropped (Section 49.3.2). Line 6 indicates the contact range beyond which it will be ignored (Section 49.5.3). Line 7 indicates the number of alert request messages that have been received (Section 49.5.2). Line 8 indicates whether visual circles are to be displayed showing the range and cpa range for a named alert (Section 49.13). In lines 10-15, the alerts configured by the mission file or incoming requests are shown. If multiple alerts types are configured, they would each be listed here separated by their alert id.

In lines 17-23, the alert status of the contact manager is output. These five lines are equivalent to the content of the CONTACTS_* variables described in Section 49.3.3. The block beginning line 34 would show any custom contact report configurations (Section 49.6.2), but there are no custom reports in this example. In lines 25-32, the status and alert history for each known contact is shown. Finally, starting on line 42, a limited list of recent events is shown. In this example, four alerts are shown, three for the vehicle `deb`, and one for the vehicle `cal`. In this particular mission, vehicles routinely proceed in and out of range to ownship, so the helm has been alerted about `deb` three times.

49.11 Configuration Parameters for pContactMgrV20

The following parameters are defined for pContactMgrV20. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 49.67: Configuration Parameters for pContactMgrV20.

able_flag: A flag to be posted whenever a contact is disabled or (re)enabled. Section 49.8.2.

alert: A description of a single alert. Section 49.2.2.

alert_range_color: The default color for rendering the alert range radius. Legal values: any color in Colors Appendix . The default is *gray70*. Section 49.13.

alert_verbose: If true, verbose debugging output will be published to the variable ALERT_VERBOSE. The default is `false`.

cease_warning_flag: A flag to be posted when or if an early warning event is configured and had been triggered, but is now no longer relevant. Section 49.8.3.

contact_local_coords: Determines if the local coordinates of incoming node reports are filled by translated latitude longitude coordinates. Legal values: `verbatim`, `lazy_lat_lon`, `force_lat_lon`. The default is `verbatim`, meaning no translation action is taken. Section 49.9.

contact_max_age: Seconds between reports before a contact is retired and dropped from the list. Legal values: any non-negative number. The default is 600. Sections 49.4.3, and 49.7.

cpa_range_color: The default color for rendering the cpa range radius. Legal values: any color in the Colors Appendix. The default is *gray30*. Section 49.13.

decay: An optional way of treating delays in between incoming contact reports. Legal values are comma-separated pair of numbers, e.g., `"15,30"`, where the first number represents the interval of time where the contact position will be extrapolated linearly into the future given its last known heading and speed. After 15 seconds, the contact speed used for extrapolation will linearly reduce to zero at 30 seconds. The default value is `"15,30"`. Extrapolation may be turned off by setting this value to `"0,0"`.

display_radii: If true, the two alert ranges are posted as viewable circles. Legal values: `true`, `false`. The default is `false`.

disable_var: The name of a MOOS variable for receiving requests for disabling contacts. If no variable is configured, the contact manager will not be able to receive requests for disabling. Section 49.8.2.

disable_flag: A flag to be posted whenever a contact is disabled. Section 49.8.2.

display_radii_id: If multiple alert IDs are configured, this parameter can be used to indicate which alert will have its range circles rendered when rendering is enabled. Section 49.13.

early_warning_flag: A flag to be posted when or if an early warning event is configured. Section 49.8.3.

early_warning_radii: If `true`, a visual range circle is posted to convey the range at which an early warning will be triggered. The default is `false`. Section 49.8.3.

early_warning_time: The number of seconds, before a contact alert is generated, when an early warning event posting is made, if configured. Section 49.8.3.

enable_var: The name of a MOOS variable for receiving requests for enabling contacts. If no variable is configured, the contact manager will not be able to receive requests for enabling. Section 49.8.2.

enable_flag: A flag to be posted whenever a contact is enabled. Section 49.8.2.

ewarn_radii_color: The color of a range circle associated with a configured early warning. Section 49.8.3.

ignore_group: A comma-separated list of contact groups to ignore, defined at the global contact manager level. Section 49.5.1.

ignore_name: A comma-separated list of contact names to ignore, defined at the global contact manager level. Section 49.5.1.

ignore_region: A convex polygon wherein contacts will be ignored, defined at the global contact manager level. Section 49.5.2.

ignore_type: A comma-separated list of contact types to ignore, defined at the global contact manager level. Section 49.5.1.

match_group: A comma-separated list of contact groups to match, defined at the global contact manager level. Section 49.5.1.

match_name: A comma-separated list of contact names to match, defined at the global contact manager level. Section 49.5.1.

match_region: A convex polygon outside of which contacts will be ignored, defined at the global contact manager level. Section 49.5.2.

match_type: A comma-separated list of contact types to match, defined at the global contact manager level. Section 49.5.1.

max_retired_hist: Maximum amount of contacts held in the contact managers memory at any given time before increasingly more dire measures are taken to remove/retire contacts. Section 49.7.

max_retired_hist: Limit the number of contacts listed in the CONTACTS_RETIRED list to this number. The default is 5, and legal values are in the range $[1, 50]$. Regardless of this setting, all retired contacts will be included in a CONTACTS_RETIRED posting at least once. (Section 49.3.2).

post_all_ranges: If true, the range to all contacts is posted in an ordered, comma-separated list. The default is `false`. Section 49.6.1.

post_closest_range: If true, the range to the closest contact is posted, rounded to the nearest meter. The default is `false`. Section 49.6.1.

recap_interval: Sets a threshold number of seconds between successive postings of the CONTACTS_RECAP variable. This variable may be just too verbose for certain use cases, and if logged, can contribute to log file bloat. The default is one second. It can be shut off completely by setting it to string value `"off"`. Section 49.3.3.

reject_range: A range, in meters, beyond which new contacts will be ignored, and existing contacts will be dropped. This is one of the guards against unbounded memory growth, discussed in Section 49.7. The default value is 2000 meters.

strict_ignore: if true, both group and type exclusion filtering will reject a contact if the contact is not reporting its group or type information and there are ignore filters. Section 49.5.1.

49.11.1 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ pContactMgrV20 --example or -e
```

This will show the output shown in Listing 68 below.

Listing 49.68: Example configuration of the pContactMgrV20 application.

```
1  ============================================================
2  pContactMgrV20 Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pContactMgrV20
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10 // Alert configurations (one or more, keyed by id)
11 // Note: Alert config in this block is deprecated. Normally
12 // now configured dynamically via BCM_ALERT_REQUEST mail.
13 alert = id=say, on_flag=SAY_MOOS=hello
14 alert = id=say, off_flag=SAY_MOOS=bye
15 alert = id=say, alert_range=80, cpa_range=100
16
17 // Type and group info about ownship
18 ownship_group = blue_team
19 ownship_type  = kayak
20
21 // Global alert filters (providing nothing is fine)
22 ignore_name = henry,gilda
23 match_name  = abe, cal
24
25 ignore_group = alpha
26 match_group  = bravo,gamma
27
28 ignore_type = usv,uuv
29 match_type  = ship
30
31 ignore_region = pts={60,-40:60,-160:150,-160:150,-40}
32 match_region  = pts={60,70:60,160:150,160:150,70}
33
34 strict_ignore = true             // Default is true
35
36 // Policy for retaining potential stale contacts
37 contact_max_age  = 60            //  Default is 60 secs.
38 max_retired_history = 5          //  Default is 5 contacts
39
40 // Configuring other output
41 display_radii     = false        // Default is false
42 alert_verbose     = false        // Default is false
43 alert_range_color = color        // default is gray65
44
```

```
45  // Policy for linear extrapolation of stale contacts
46  decay = 15,30                    // Default is 15,30 secs
47
48  recap_interval       = 1         // Default is 1 sec
49  range_report_timeout = 10        // Default is 10 secs
50  range_report_maxsize = 20        // Default is 20 reports
51
52  contact_local_coords = verbatim  // Default is verbatim
53  post_closest_range   = false     // Default is false
54  post_closest_relbng  = false     // Default is false
55  post_all_ranges      = false     // Default is false
56
57  reject_range = 2000              // Default is 2000 meters
58  max_contacts = 500               // Default is 500
59  alert_verbose = false            // Default is false
60
61  // 24.8.x support for disabling/enabling contacts
62  disable_var  = DISABLE_TARGET    // Default is null string
63  enable_var   = REENABLE_TARGET   // Default is null string
64  able_flag    = MOOS_VAR = val
65  disable_flag = MOOS_VAR = val
66  enable_flag  = MOOS_VAR = val
67
68  // 24.8.x support for disabling/enabling contacts
69  early_warning_time  = 30        // Default is -1 (off)
70  early_warning_radii = true      // Default is false
71  ewarn_radii_color   = yellow    // Default is yellow
72
73  early_warning_flag  = MOOS_VAR = val
74  cease_warning_flag  = MOOS_VAR = val
75
76  app_logging = true               // Default is false
77
78  hold_alerts_for_helm = true      // Default is false
79
80 }
```

49.12 Publications and Subscriptions for pContactMgrV20

The interface for pContactMgrV20, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pContactMgrV20 --interface or -i
```

49.12.1 Variables Published by pContactMgrV20

The primary output of pContactMgrV20 to the MOOSDB is the set of user-configured alerts, early warnings, or enable flags. Other variables are published on each iteration where a change is detected on its value:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 49.10.

- ALERT_VERBOSE: Verbose or debugging output published to this variable if alert_verbose is set to true.
- CONTACT_CLOSEST: Name of the closest contact. Section 49.6.1.
- CONTACT_CLOSEST_TIME: The time at which the closest contact became the closest contact. Section 49.6.1.
- CONTACT_CLOSEST_RANGE: The range of the closest contact. Section 49.6.1.
- CONTACT_MGR_WARNING: A warning message indicating possible mishandling of or missing data.
- CONTACT_RANGES: A comma-separated and ordered list of all contact ranges. Section 49.6.1.
- CONTACTS_ALERTED: A list of contacts for which alerts have been posted. Section 49.3.3.
- CONTACTS_COUNT: The number of contacts currently alerted. Section 49.3.3.
- CONTACTS_LIST: A comma-separated list of contacts. Section 49.3.3.
- CONTACTS_RECAP: A comma-separated list of contact summaries. Section 49.3.3.
- CONTACTS_RETIRED: A list of contacts removed due to the information staleness. Section 49.3.3.
- CONTACTS_UNALERTED: A list of contacts for which alerts are pending, based on the range criteria. Section 49.3.3.
- PCONTACTMGRV20_PID: The system process ID of the contact manager.
- VIEW_CIRCLE: A rendering of the alert ranges. Section 49.13.

Some examples:

```
CONTACTS_LIST      = gus,joe,ken,kay
CONTACTS_ALERTED   = gus,kay
CONTACTS_UNALERTED = ken,joe
CONTACTS_RETIRED   = bravo,foxtrot,kilroy
CONTACTS_RECAP     = name=gus,age=7.3,range=13.1 # name=ken,age=0.7,range=48.1 # \
                     name=joe,age=1.9,range=73.1 # name=kay,age=4.0,range=18.2
```

49.12.2 Variables Subscribed for by pContactMgrV20

The pContactMgrV20 application will subscribe for the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new appcast report, with reporting criteria, and expiration.
- BCM_ALERT_REQUEST: If false, no postings will be made for rendering the alert and cpa range circles. Section 49.2.3.
- BCM_DISPLAY_RADII: If false, no postings will be made for rendering the alert and cpa range circles. Section 49.13.
- BCM_REPORT_REQUEST: If false, no postings will be made for rendering the alert and cpa range circles. Section 49.6.2
- NAV_HEADING: Present ownship heading in degrees.
- NAV_SPEED: Present ownship speed in meters per second.
- NAV_X: Present position of ownship in local x coordinates.
- NAV_Y: Present position of ownship in local y coordinates.
- NODE_REPORT: A report about a known contact.

49.12.3 Command Line Usage of pContactMgrV20

The pContactMgrV20 application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. To see command-line options enter the following from the command-line:

```
$ pContactMgrV20 --help or -h
```

This will show the output shown in Listing 69 below.

Listing 49.69: Command line usage for the pContactMgrV20 application.

```
1  ============================================================
2  Usage: pContactMgrV20 file.moos [OPTIONS]
3  ============================================================
4
5  SYNOPSIS:
6  ------------------------------------
7    The contact manager deals with other known vehicles in its
8    vicinity. It handles incoming reports perhaps received via a
9    sensor application or over a communications link. Minimally
10   it posts summary reports to the MOOSDB, but may also be
11   configured to post alerts with user-configured content about
12   one or more of the contacts.
13
14 Options:
15   --alias=<ProcessName>
16       Launch pContactMgrV20 with the given process
17       name rather than pContactMgrV20.
18   --example, -e
19       Display example MOOS configuration block.
20   --help, -h
21       Display this help message.
22   --interface, -i
23       Display MOOS publications and subscriptions.
24   --version,-v
25       Display the release version of pContactMgrV20.
26
27 Note: If argv[2] does not otherwise match a known option,
28       then it will be interpreted as a run alias. This is
29       to support pAntler launching conventions.
```

49.13 Visuals

The contact manager will optionally generate visual artifacts, in the form of postings to the variable VIEW_CIRCLE, for consumption by pMarineViewer for rendering the alert range and cpa range. Rendering of ranges is done with the following three configuration parameters:

```
diplay_radii = true
alert_range_color = <color>
cpa_range_color = <color>
```

The default value for display_radii is false. The default value for alert_range_color is gray70, and the default value for cpa_range_color is gray30. See the Colors Appendix for more on colors.

If multiple alerts are configured in the contact manager, the user may need to specify which alert ID to render. If no alert ID is specified, the contact manager will choose the first one in its memory. To specifify the ID, the following parameter may be used:

```
diplay_radii_id = <alert_id>
```

The posting of rendering circles by pContactMgrV20 may also be be adjusted dynamically via the incoming MOOS message BCM_DISPLAY_RADII. The value of this message may be either

- true or on,
- false of off,
- toggle,
- A configured alert ID

50 uProcessWatch: Monitoring MOOS Application Health

50.1 Overview

The uProcessWatch application monitors the health of a set of MOOS application. It does two things:

- It monitors the presence of a set of MOOS apps,
- It monitors the CPU load of a set of MOOS apps.

In the case of the former, uProcessWatch continually monitors the DB_CLIENTS list for applications it has responsibility for watching. In the case of the latter, MOOS apps are already monitoring and reporting their own CPU load and uProcessWatch is doing nothing more than gathering that information for display in the terminal or appcast message. An example terminal output is shown below:

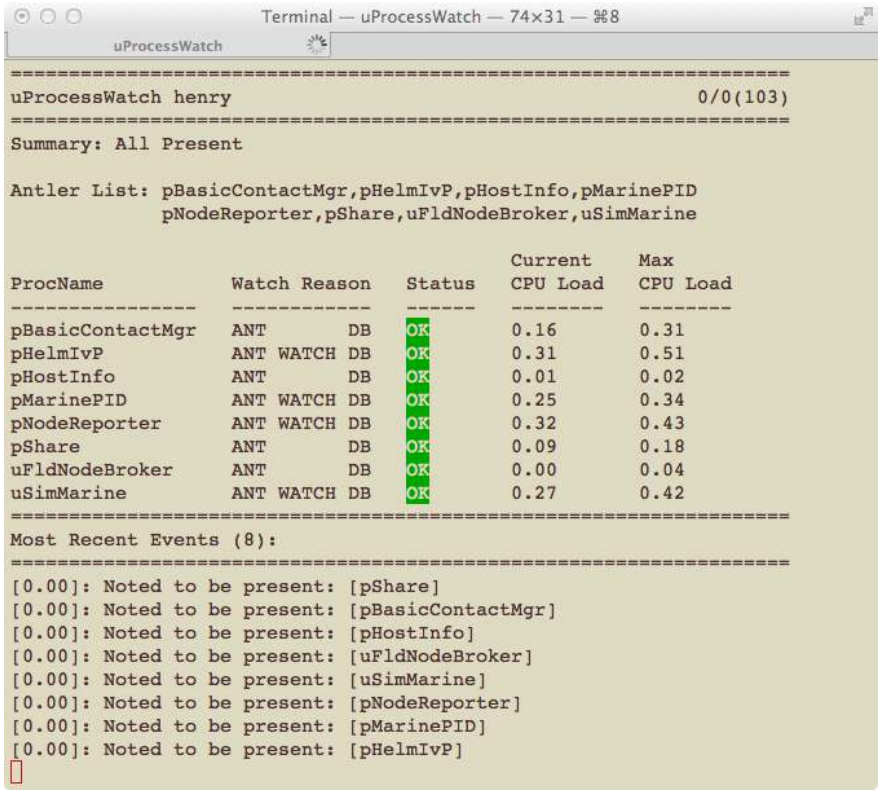

Figure 164: **A typical terminal or appcast report for uProcessWatch.**

The first line, "Summary: All Present", tells you that no processes are missing. This is also posted in the variable PROC_WATCH_SUMMARY. The list of applications launched with pAntler is shown in the next block. The main body shows, for each watched application, the reason why the process is on the watch list, the status, current CPU load as reported by the process itself, and the maximum CPU load noted so far.

The bottom section of the terminal output shows the events (similar to all appcasting output). In the case of uProcessWatch, events note either the arrival or disappearance of a watched process. Each event also results in the posting of the variable PROC_WATCH_EVENT. The user may configure uProcessWatch to *not* watch certain named applications or patterns of applications, such as uXMS*, to avoid unwarranted alerts.

50.2 Typical uProcessWatch Usage Scenarios

50.2.1 Using uProcessWatch with AppCasting and pMarineViewer

The first usage scenario is perhaps the most typical since it is viable both in simulation and in the field where the vehicles have a network connection to a shoreside MOOS community. The idea is shown in Figure 165 below. Multiple vehicles each run uProcessWatch locally. A network connection from each vehicle to a shoreside MOOS community is used to share appcast messages using pShare. The shoreside community is running pMarineViewer which supports a multi-vehicle appcast viewing mode. In this way, the shoreside user may monitor the health of all vehicles' applications with one tool. If a process on a single vehicle goes missing, the menu item on the shoreside viewer for that vehicle turns red.

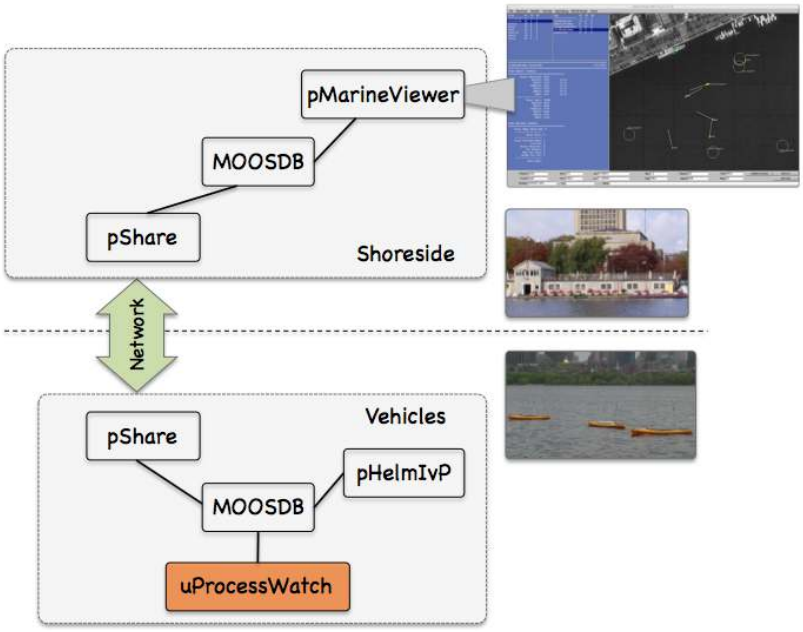

Figure 165: **Using uProcessWatch with AppCasting:** uProcessWatch is run locally on each fielded vehicle, generating appcasts posted to the local MOOSDB. Appcasts are shared to the shoreside and collected by the pMarineViewer tool for monitoring processes across all vehicle. This scenario is relevant when there is a network connection from the vehicles to the shoreside.

50.2.2 Directly Accessing the PROC_WATCH_SUMMARY Output

The main health indicator produced by uProcessWatch in the MOOS variable PROC_WATCH_SUMMRY. This can be checked on a remote machine by simply scoping on this variable. The uMS application

distributed with MOOS will do the trick for example. To focus solely on this variable, the uXMS tool may also be used as follows:

```
$ uXMS --server_host=10.25.0.72 --server_port=9000 PROC_WATCH_SUMMARY
```

Or one may ssh onto the vehicle and launch a scope locally on this variable.

50.3 Using and Configuring the uProcessWatch Utility

The primary configuration of uProcessWatch is defining the *watch list*, the set of other MOOS processes to monitor. By default all processes ever noted to be connected to the MOOSDB are on the watch list. The same with all processes name in the Antler configuration block of the mission file. This default is used because it is simple, and a good rule of thumb is that if any process disconnects from the MOOSDB, it's probably a sign of trouble.

50.3.1 The DB_CLIENTS Variable for Detecting Missing Processes

The MOOSDB, about once per second, posts the variable DB_CLIENTS containing a comma-separated list of clients (other MOOS apps) currently connected to the MOOSDB. When uProcessWatch is configured to watch for all processes, it simply augments the watch list for any process that ever appears on the incoming DB_CLIENTS mail. If the process is then missing at a later reading of this DB_CLIENTS mail, is is considered AWOL.

50.3.2 Defining the Watch List

The *watch list* is a list of processes, i.e., MOOS apps. Each item on the list will be reported missing if it does not appear in the list of clients shown by the current value of DB_CLIENTS. By default, all clients that ever appear on the DB_CLIENTS list will be added to the watch list. The same is true for all processes named in the Antler configuration block of the mission file. This default can be overridden by the following configuration option:

```
watch_all = false
```

The value of watch_all may also be set to "antler" to indicate that items on the Antler list are to be watched by default, but not those that appear in the DB_CLIENTS list. Likewise it may be set to "dbclients" to indicate that items in the DB_CLIENTS list are to be watched by default, but not those in the Antler list. Processes may be added to the watch list by explicitly naming them in configuration block as follows:

```
watch = process_name[*]
```

A process may be named explicitly, or the prefix of the process may be given, e.g., watch=uXMS*. This will match all processes fitting this pattern, e.g., uXMS_845, uXMS_23.

Processes may be explicitly *excluded* from the watch list with configurations of the following:

```
nowatch = process_name[*]
```

This is perhaps most appropriate when coupled with watch_all=true. If a process is for some reason both explicitly included and excluded, the inclusion takes precedent.

50.3.3 Reports Generated

There are three reports generated in the variables:

- PROC_WATCH_EVENT
- PROC_WATCH_SUMMARY,
- PROC_WATCH_FULL_SUMMARY

All reports are generated only when there is a change of status in one of the watched processes. The first type of report is generated for each event when a watched process is noted to have connected or disconnected to the MOOSDB. The following are examples:

```
PROC_WATCH_EVENT = "Process [pMarinePID] is noted to be present."
PROC_WATCH_EVENT = "Process [pMarinePID] has died!!!!"}
PROC_WATCH_EVENT = "Process [pMarinePID] is resurected!!!"
```

In the first line above, a process is reported to be present that was never previously on the watch list. In the third line a process that was previously noted to have left the watch list is reported to have returned or been restarted.

The PROC_WATCH_SUMMARY variable will list the set of processes missing from the watch list or report `"All Present"` if no items are missing. For example:

```
PROC_WATCH_SUMMARY = "All Present"
PROC_WATCH_SUMMARY = "AWOL: pMarinePID,uSimMarine"
```

The PROC_WATCH_FULL_SUMMARY variable will list a more complete and historical status for all processes on the watch list. For example:

```
PROC_WATCH_FULL_SUMMARY = "pHelmIvP(1/0),uSimMarine(1/1),pMarinePID(2/2)"
```

The numbers in parentheses indicate how many times the process has been noted to connect to the MOOSDB over the number of times it has been noted to have disconnected. A report of `"(1/0)"` is the healthiest of possible reports, meaning it has connected once and has never disconnected.

50.3.4 Watching and Reporting on a Single MOOS Process

If desired, uProcessWatch may be configured to generate a report dedicated a single MOOS process with the following example configuration:

```
watch = pBasicContactMgr : BCM_OK
```

In this case a MOOS variable, BCM_OK, will be set to either true or false depending on whether the process pBasicContactMgr presently appears on the list of connected clients in listed in DB_CLIENTS.

50.3.5 A Heartbeat for the Watch Dog

By default uProcessWatch will only post PROC_WATCH_SUMMARY when its value changes. A long stale PROC_WATCH_SUMMARY = "All Present" likely means that everything is fine. It could also mean the uProcWatchSummary process itself died without ever posting anything to suggest otherwise. The user has the configuration option to post PROC_WATCH_SUMMARY every N seconds regardless of whether or not the value has changed:

```
summary_wait = 30  // Summary posted at least every 30 seconds
```

This in effect creates a heartbeat for monitoring uProcessWatch. The default value for this parameter is -1. Any negative value will be interpreted as a request for postings to be made only when the posting value changes. Regardless of the summary_wait setting, the other two reports, PROC_WATCH_EVENT and PROC_WATCH_FULL_SUMMARY, will only be made when the values change.

50.3.6 Excusing a Process

An application on the watch list may be excused and disconnect from the MOOSDB if it posts to the variable EXITED_NORMALLY with the name of itself. A check is made by uProcessWatch of the *source* of the posting to ensure that message was posted by the exiting process. It's up to the developer of an application to build in the feature declaring a normal exit. An example is the uTimerScript application which has the ability to execute a script of postings to the MOOSDB followed by an exit. In this case, the *status* of application becomes EXCUSED, as shown in Figure 166.

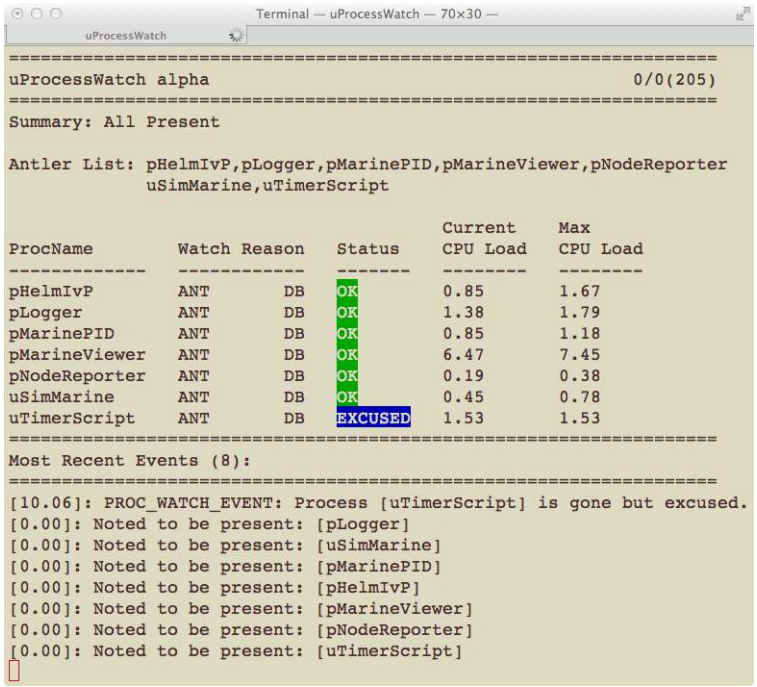

Figure 166: The process uTimerScript has gone missing due to its own intentional exit. Before exiting it posted EXITED_NORMALLY=uTimerScript, and therefore uTimerScript is regarded as excused.

Excusing a missing process is different than choosing to not include it on the watch list. Some applications are, by their nature, designed to disappear at some point. Scoping tools like uXMS, uPokeDB or uHelmScope are examples applications that regularly appear and disappear. They are best excluded entirely from the watch list with the nowatch configuration parameter.

50.3.7 Allowing Retractions if a Process Reappears

With the addition of appcasting to uProcessWatch, a missing process also triggers an appcast run warning, as shown on the top in Figure 167 below.

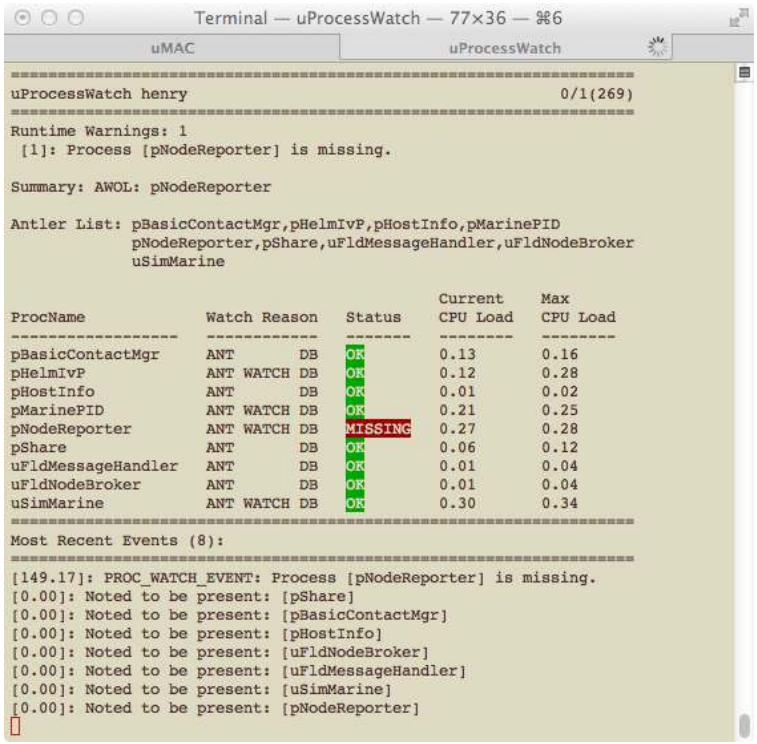

Figure 167: The process pNodeReporter has gone missing. This is noted as a runtime warning at the top, and the MISSING status in the body of the report.

However, there are cases where a process goes missing but then returns, in which case the warning is a distraction. These reasons include.

- The application may actually exit and then restart a short time later.
- The application my be running at a very slow apptick in which case it may be dropped from the DB_CLIENTS list momentarily.
- The application may be missing only because it hasn't launched yet.

To address this, uProcessWatch will allow retractions on appcast run warnings. If the application disappears and reappears, the appcast run warning will disappear. The successive posted values of PROC_WATCH_SUMMARY will show that the process was at some point declared AWOL, but once the process returns, the appcast output will no longer raise attention to the issue.

Of course there are situations where a process that disappears and then reappears is an indication of a problem, not to be swept under the rug. In this case the default behavior of uProcessWatch may be changed by setting allow_retractions to false. Presently this setting cannot be set on a per-process manner.

50.4 Configuration Parameters of uProcessWatch

The following parameters are defined for uProcessWatch. A more detailed description is provided in other parts of this section. Parameters having default values indicate so in parentheses below.

Listing 50.70: Configuration Parameters for uProcessWatch.

allow_retractions:	If true, run warnings are retracted if a process reappears after disappearing. Legal values: true, false. The default is true.
nowatch:	A process or list of MOOS processes to *not* watch.
post_mapping:	A mapping from one posting variable name to another.
summary_wait:	A maximum amount of time between PROC_WATCH_SUMMARY postings. Negative value indicates posting occurs only when the value changes regardless of elapsed time. Legal values: any numerical value. The default is −1.
watch:	One or more comma-separated MOOS process to watch and report on.
watch_all:	If true, watch all processes that become known either via the DB_CLIENTS list or the Antler list. Legal values: true, false. The default is true.

50.4.1 An Example MOOS Configuration Block

Listing 71 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uProcessWatch --example or -e
```

Listing 50.71: Example configuration of the uProcessWatch application.

```
ProcessConfig = uProcessWatch
{
  AppTick   = 4
  CommsTick = 4

  watch_all = true   // The default is true.

  watch  = pMarinePID:PID_OK
  watch  = uSimMarine:USM_OK

  nowatch = uXMS*

  allow_retractions = true   // Always allow run-warnings to be
                             // retracted if proc re-appears

  // A negative value means summary only when status changes.
  summary_wait = 10 // Seconds. Default is -1.
```

```
18
19    post_mapping = PROC_WATCH_FULL_SUMMARY, UPW_FULL_SUMMARY
20  }
```

50.5 Publications and Subscriptions for uProcessWatch

The interface for uProcessWatch, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uProcessWatch --interface or -i
```

50.5.1 Variables Published by uProcessWatch

The primary output of uProcessWatch to the MOOSDB is a summary indicating whether or not certain other processes (MOOS apps) are presently connected.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- PROC_WATCH_EVENT: A report indicating a particular process has been noted to be gone missing or noted to have (re)joined the list of active processes.
- PROC_WATCH_FULL_SUMMARY: A single string report for each process indicating how many times it has connected and disconnected from the MOOSDB.
- PROC_WATCH_SUMMARY: A report listing all missing processes, or "All Present" if no processes are missing.

The user may also configure uProcessWatch to make a posting dedicated to a particular watched process. For example, with the configuration watch=pNodeReporter:PNR_OK, the status of this process is conveyed in the MOOS variable PNR_OK, set to either true or false depending on whether or not it is present.

The variable name for any posted variable may be changed to a different name with the post_mapping configuration parameter. For example, post_mapping=PROC_WATCH_EVENT, UPW_EVENT will result in events being posted under the UPW_EVENT variable rather than PROC_WATCH_EVENT variable.

50.5.2 MOOS Variables Subscribed for by uProcessWatch

The following variable(s) will be subscribed for by uProcessWatch:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- DB_CLIENTS: A comma-separated list of clients currently connected to the MOOSDB, posted by the MOOSDB. Used for detecting missing processes. Section 50.3.1.
- EXITED_NORMALLY: An indication made by a process, potentially on the watch list, that it has exited normally and should be excused, and not regarded as missing. Section 50.3.6.
- <PROCNAME>_STATUS: The status string generated for all MOOSApps, containing the current CPU load among other things.

51 uLoadWatch: Monitoring Application System Load

51.1 Overview

The uLoadWatch application monitors data produced by individual applications and reports a warning when the MOOS community begins to exhibit a strain in the CPU load. Load data is derived solely from two MOOS variables published by participating MOOS apps. For an example application pFooBar, the following two variables are published:

- PFOOBAR_ITER_GAP: indicates the ratio of (a) observed time between invocations to the application's `Iterate()` method, and (b) the scheduled time between iterations based on the configured AppTick. For example, for an application running at 4 Hz, a gap of 1.0 means things are running normally with 0.25 seconds in between iterations. A gap of 3 indicates that there was 0.75 seconds between iterations - an an idication that the CPU cannot keep up with the work being done in the iterate loop. This could, however, be due in part to other processes demanding CPU resources.
- PFOOBAR_ITER_LEN: indicates the ratio of (a) observed time between the begin and end of the `Iterate()` loop, and (b) again, the scheduled time between iterations based on the configured AppTick. In this case, a value of 1 is an indication that the work being done in the iterate loop is dangerously close to capacity.

The ITER_GAP variable indicates when strain is actually being exhibited. The ITER_LEN variable indicates the run-up to problems. The uLoadWatch application monitors the situation by registering for the *_ITER_GAP and *_ITER_LEN from all apps and doing three things:

1. Threshold detection. A user configured threshold for ITER_GAP values may be set, with a posting to LOAD_WARNING when it is exceeded.
2. An AppCast run warning will also be posted when the user configured threshold is exceeded. This will bring a critical load situation to the attention of an operator through the existing AppCast mechanisms.
3. An AppCast report is generated reporting the average and maximumn ITER_GAP and ITER_LEN noted thus far. This can be monitored via one of the AppCast Viewers to monitor the load fluctuation.

51.2 Configuration Parameters for uLoadWatch

The uLoadWatch application may be configured with a configuration block within a MOOS mission file, typically with a `.moos` file suffix. The following parameters are defined for uLoadWatch.

Listing 51.72: Configuration Parameters for uLoadWatch.

breach_trigger: The number of times a threshold can be breached before a warning is posted. The default is 1 meaning the first offense is forgiven, since it is not uncommon for some apps to have a longer initial gap as they perform one-time start-up work.

near_breach_thresh: In addition to normal threshold breaches, the uLoadWatch app may also monitor for *near* breaches. A near breach occurs when the normal breach thresh is nearly met. For example, if the threshold for a certain app is 2.0, this means breach is declared if time between app iterations is greater than 2x the normal gap between iterations. If the near_breach_thresh is set to 0.6, a near breach is declared if the gap reaches $1.6x$ the normal gap.

thresh: A gap threshold for reporting a LOAD_WARNING. Each threshold line specifies the application name and the gap threshold value.

51.2.1 An Example MOOS Configuration Block

An example MOOS configuration block may be obtained from the command line with the following:

```
$ uLoadWatch --example or -e
```

Listing 51.73: Example configuration of the uLoadWatch application.

```
1  ============================================================
2  pHostInfo Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pHostInfo
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   thresh = app=pHelmIvP, gapthresh=1.5
11   thresh = app=any,      gapthresh=2.0
12
13   near_breach_thresh = 0.9   // default is off
14
15   breach_trigger = 5         // default is 1
16 }
```

51.3 Publications and Subscriptions for uLoadWatch

The interface for uLoadWatch, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uLoadWatch --interface or -i
```

51.3.1 Variables Published by uLoadWatch

The primary output of uLoadWatch is the occasional load warning plus the appcasting report. In Release 20.2, a few additional variables were added to summarize breaches and near breaches.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.

- LOAD_WARNING: A warning indicating an app has been detected reporting an gap greater than the configured threshold.
- ULW_BREACH: Posted with value `true`, if and only when a breach has been detected, for any app.
- ULW_BREACH_COUNT: The total amount of breaches detected, for all all apps.
- ULW_BREACH_LIST: The list of all apps for which at least one breach has been detected.
- ULW_NEAR_BREACH: Posted with value `true`, if and only when a *near* breach has been detected, for any app.
- ULW_NEAR_BREACH_COUNT: The total amount of *near* breaches detected, for all all apps.
- ULW_NEAR_BREACH_LIST: The list of all apps for which at least one *near* breach has been detected.

51.3.2 Variables Subscribed for by uLoadWatch

The uLoadWatch application subscribes all variables ending with the suffix _ITER_GAP and the suffix _ITER_LEN:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- *_ITER_GAP: A report from an application indicating its most recent observed gap between iterations.
- *_ITER_LEN: A report from an application indicating its most recent observed itertion duration.

51.3.3 Command Line Usage of uLoadWatch

The uLoadWatch application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uLoadWatch --help or -h
```

Listing 51.74: Command line usage for the uLoadWatch tool.

```
1  Usage: uLoadWatch file.moos [OPTIONS]
2
3  Options:
4    --alias=<ProcessName>
5        Launch pHostInfo with the given process
6        name rather than pHostInfo.
7    --example, -e
8        Display example MOOS configuration block
9    --help, -h
10       Display this help message.
11   --interface, -i
12       Display MOOS publications and subscriptions.
13   --version,-v
14       Display the release version of pHostInfo.
15
16 Note: If argv[2] is not of one of the above formats
17       this will be interpreted as a run alias. This
18       is to support pAntler launching conventions.
```

51.4 Usage Scenarios the uLoadWatch Utility

A typical usage of the uLoadWatch utility is shown in Figure 168 below. If the mission is being monitored by a human (vs. an offline simulation), a load warning will present itself in any of the appcast viewing tools, e.g., pMarineViewer, uMACView, or uMAC. The pNodeReporter application also registers for LOAD_WARNING mail and will include the load warning in vehicle node reports. Depending on how pMarineViewer is configured, the vehicle label may indicate a load warning if it occurs.

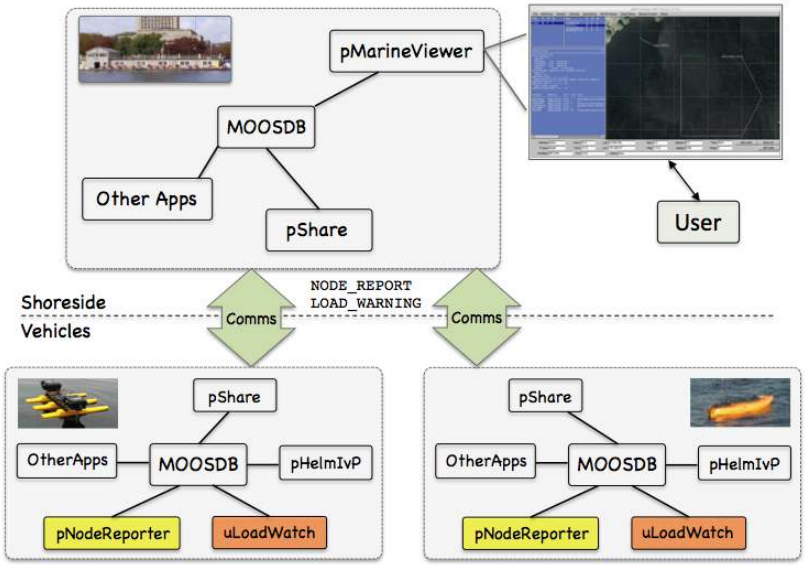

Figure 168: **Typical uLoadWatch Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports, and AppCast reports and warnings. If a load warning is produced, it will (a) show up in any shoreside appcast viewer, (b) show up in node reports sent to the shoreside, and (c) show up in local vehicle alog files.

When running an offline simulation, the output of uLoadWatch is primarily found in the log files.

51.5 Terminal and AppCast Output

The uLoadWatch application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 75 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. See the document for uMac for more on appcasting and viewing appcasts. The counter on the end of line 2 is incremented on each iteration of uLoadWatch, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 51.75: Example terminal or appcast output for uLoadWatch.

```
1  ============================================================
2  uLoadWatch henry                                      0/0(129)
3  ============================================================
4  Configured Thresholds:
```

```
 5    ANY: 2
 6    PHELMIVP: 1.5
 7
 8  Application       AvgGap  MaxGap  AvgLen  MaxLen
 9  ----------------  ------  ------  ------  ------
10  PBASICCONTACTMGR  1.01    1.02    0.00    0.02
11  PHELMIVP          1.01    1.04    0.00    0.01
12  PHOSTINFO         1.01    1.01    0.00    0.58
13  PNODEREPORTER     1.01    1.02    0.00    0.01
14  UFLDNODEBROKER    1.01    1.01    0.00    0.00
15  ULOADWATCH        1.01    1.01    0.00    0.00
16  UPROCESSWATCH     1.00    1.02    0.00    0.00
17  USIMMARINE        1.07    1.15    0.02    0.17
```

The first few lines indicate any user-configured thresholds being applied to incoming reports. Here a LOAD_WARNING will be issued if the pHelmIvP has an iter gap above 1.5, and if any application has a gap over 2.0. The remainder of the report (lines 8-17 here) provide a live update of the average and maximum iter gap and iter length reported for each application reporting.

51.6 How to Modify a MOOS App to Produce Load Information

All the applications shown in Listing 75 above produce the *_ITER_GAP and *_ITER_LEN information by virtue of being an AppCastingMOOSApp. If you have an application that does not subclass this superclass, it is still pretty easy to add this to your class.

To add the ITER_LEN, and ITER_GAP information, follow the example of the pseudocode below. The code requires some memory between iterations of the timestamp of the prior iteration. So the code requires the additional member variable, added to the YourApp class definition, m_last_iterate_time, initialized to zero in the constructor.

Listing 51.76: Adding appcast output for uLoadWatch.

```
1  YourApp::Iterate()
2  {
3    // Note the time the iterate loop started
4    double start_time = MOOSTime();
5
6
7    (The prior guts of your Iterate loop)
8
9
10   // Note the time the iterate loop ended
11   double end_time = MOOSTime();
12
13   // Only report iter_gap, iter_len if app frequency is > zero
14   double app_freq = GetAppFreq();
15   if(app_freq > 0) {
16      double interval = 1 / app_freq;
17      string app_name = MOOSToUpper(GetAppName());89
18
19      double iter_len = (MOOSTime() - start_time) / interval;
20      Notify(app_name + "_ITER_LEN", iter_len);
21
22      double iter_gap = (start_time - m_last_iterate_time) / interval;
```

```
23       if(m_last_iterate_time != 0)
24         Notify(app_name + "_ITER_GAP", iter_gap);
25       m_last_iterate_time = start_time;
26     }
27   }
```

52 pNodeReporter: Summarizing a Node's Position and Status

52.1 Overview

The pNodeReporter app runs on each vehicle (real or simulated) and generates node reports for sharing between vehicles, depicted in Figure 169. They can be regarded as a proxy for AIS reports. The app serves one primary function - it repeatedly gathers local platform information and navigation data and creates an AIS like report in the form of the MOOS variable NODE_REPORT_LOCAL. The NODE_REPORT messages are communicated between the vehicles and the shore or shipside command and control through an inter-MOOSDB communications process such as pShare or via acoustic modem. Since a node or platform may both generate and receive reports, the locally generated reports are labeled with the _LOCAL suffix and transmitted to the outside communities without the suffix. This is to ensure that processes running locally may easily distinguish between locally generated and externally generated reports.

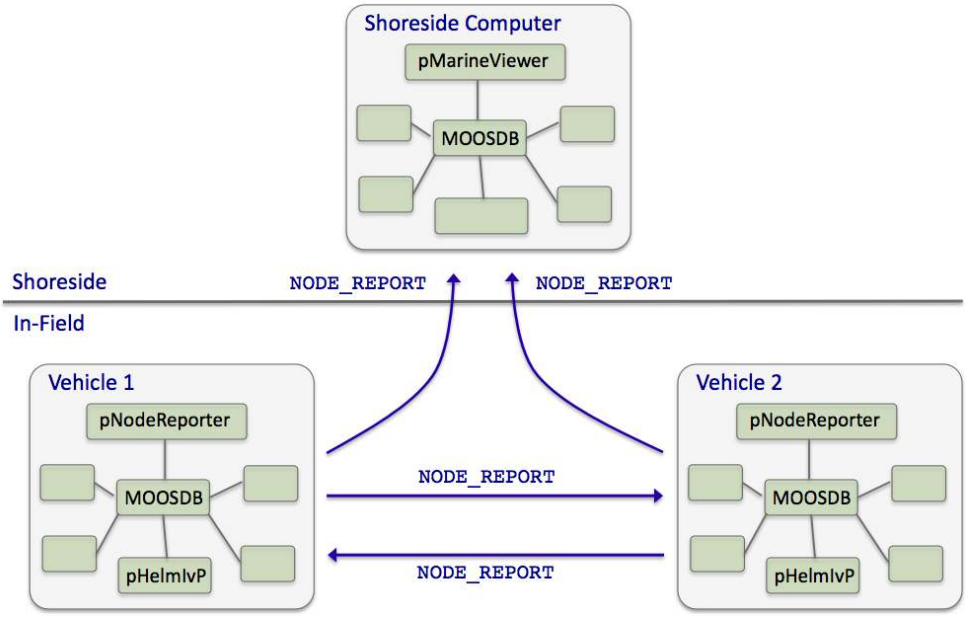

Figure 169: **Typical pNodeReporter usage:** The pNodeReporter application is typically used with pShare or acoustic modems to share node summaries between vehicles and to a shoreside command-and-control GUI.

To generate the local report, pNodeReporter registers for the local NAV_* vehicle navigation data and creates a report in the form of a single string posted to the variable NODE_REPORT_LOCAL. An example of this variable is given in below in Section 52.2.1. The pMarineViewer and pHelmIvP applications are two modules that consume and parse the incoming NODE_REPORT messages.

The pNodeReporter utility may also publish a second report, the PLATFORM_REPORT. While the NODE_REPORT summary consists of an immutable set of data fields described later in this section, the PLATFORM_REPORT consists of data fields configured by the user and may therefore vary widely across applications. The user may also configure the frequency in which components of the PLATFORM_REPORT are posted within the report.

52.2 Using pNodeReporter

52.2.1 Overview Node Report Components

The primary output of pNodeReporter is the node report string. It is a comma-separated list of key-value pairs. The order of the pairs is not significant. The following is an example report:

```
NODE_REPORT_LOCAL = "NAME=alpha,TYPE=UUV,TIME=1252348077.59,X=51.71,Y=-35.50,
                     LAT=43.824981,LON=-70.329755,SPD=2.00,HDG=118.85,YAW=118.84754,
                     DEP=4.63,LENGTH=3.8,MODE=MODE@ACTIVE:LOITERING"
```

The TIME reflects the Coordinated Universal Time as indicated by the system clock running on the machine where the MOOSDB is running. Speed is given in meters per second, heading is in degrees in the range $[0, 360)$, depth is in meters, and the local x-y coordinates are also in meters. The source of information for these fields is the NAV_* navigation MOOS variables such as NAV_SPEED. The report also contains several components describing characteristics of the physical platform, and the state of the IvP Helm, described next.

If desired, pNodeReporter may be configured to use a different variable than NODE_REPORT_LOCAL for its node reports, by setting the configuration parameter node_report_output to MY_REPORT for example. Most applications that subscribe to node reports, subscribe to two variables, NODE_REPORT_LOCAL and NODE_REPORT. This is because node reports are meant to be bridged to other MOOS communities (typically with pShare but not necessarily). A node report should be broadcast only from the community that generated the report. In practice, to ensure that node reports that arrive in one community are not then sent out to other communities, the node reports generated locally have the _LOCAL suffix, and when they are sent to other communities they are sent to arrive with the new variable name, minus the suffix.

52.2.2 Helm Characteristics

The node report contains one field regarding the current mode of the helm, MODE. Typically the pNodeReporter and pHelmIvP applications are running on the same platform, connected to the same MOOSDB. When the helm is running, but disengaged, i.e., in manual override mode, the MODE field in the node report simply reads "MODE=DISENGAGED". When or if the helm is detected to be not running, the field reads "MODE=NOHELM-SECS", where SECS is the number of seconds since the last time pNodeReporter detected the presence of the helm, or "MODE=NOHELM-EVER" if no helm presence has ever been detected since pNodeReporter has been launched.

How does pNodeReporter know about the health or status of the helm? It subscribes to two MOOS variables published by the helm, IVPHELM_STATE and IVPHELM_SUMMARY. These are described more fully in helm documentation, but below are typical example values:

```
IVPHELM_STATE   = "DRIVE"

IVPHELM_SUMMARY = "iter=72,ofnum=1,warnings=0,time=127349406.22,solve_time=0.00,
                   create_time=0.00,loop_time=0.00,var=course:209.0,var=speed:1.2,
                   halted=false,running_bhvs=none,modes=MODE@ACTIVE:LOITERING,
                   active_bhvs=loiter$17.8$100.00$9$0.04$0/0,completed_bhvs=none
                   idle_bhvs=waypt_return$17.8$0/0:station-keep$17.8$n/a
```

The IVPHELM_STATE variable is published on each iteration of the pHelmIvP process regardless of whether the helm is in manual override ("PARK") mode or not, and regardless of whether the value of this variable has changed between iterations. It is considered the "heartbeat" of the helm. This is the variable monitored by pNodeReporter to determine whether a "NOHELM" message is warranted. By default, a period of five seconds is used as a threshold for triggering a "NOHELM" warning. This value may be changed by setting the nohelm_threshold configuration parameter.

When the helm is indeed engaged, i.e., not in manual override mode, the value of IVPHELM_STATE posting simply reads "DRIVE", but the helm further publishes the IVPHELM_SUMMARY variable similar to the above example. If the user has chosen to configure the helm using hierarchical mode declarations (as described in helm documentation), the IVPHELM_SUMMARY posting will include a component such as "modes=MODE@ACTIVE:LOITERING" as above. This value is then included in the node report by pNodeReporter. If the helm is not configured with hierarchical mode declarations, the node report simply reports "MODE=DRIVE".

52.2.3 Custom Rider Augmentation of the Node Report

A node report typically is limited to the components shown in the Section 52.2.1, e.g., vehicle heading, speed, and position. These components are derived from the MOOS variables NAV_HEADING, NAV_SPEED, NAV_LAT and NAV_LONG.

A rider is a custom augmentation of the node report, resulting in additional param=value pairs in the outgoing node report. There are *static* riders, configured in the mission file, and *dynamic* riders with values filled in at run time, where the source is a named MOOS variable, configured in the mission file.

Static Riders A *static* rider is set in the mission file, typically for values that are known prior to launch time, and do not change during the course of the mission. Post Release 24.8, this is possible with the platform_aspect configuration parameter: For example:

```
platform_aspect = ID=439
platform_aspect = FUEL_TYPE=diesel
```

This will result in the string "ID=439,FUEL_TYPE=diesel" added to all outgoing node reports.

Of course the following is possible too:

```
platform_aspect = ID=$(ID)
```

In this case presumably the $(ID) macro is filled in at launch time with a utility like nsplug.

Note: if two platform_aspect configurations are made with the same lefthand component, the second line will overwrite the first, and there will be no warning.

Dynamic Riders A *dynamic* rider is a param=value component where the value changes dynamically at run time. For example, if there are MOOS variables ENGINE_TEMP, or FUEL_LEVEL, a

user may wish to include these in a node report. The last part of the node report may look like `ETMP=145.32,FUEL=84.5`.

Post Release 24.8, this can be done with the `rider` configuration parameter. Each rider contains (a) a MOOS variable, (b) the field name in the node report, (c) the frequency at which it is include in node reports, and (d) the precision, if the value is numerical. Continuing with the above two examples, the configuration would like:

```
rider = var=ENGINE_TEMP, rfld=ETMP, policy=always, prec=2
rider = var=FUEL_LEVEL, rfld=FUEL, policy=always, prec=1
```

The `var` component names the MOOS variable to watch. The `rfld` component names the "rider field", i.e., the field in the node report. The policy component indicates how often it should be included in the node report. If set to *always*, it will be in every report. If set to *updated*, it will be include in reports only when the value changes. If set to *15*, it will be included only once every 15 seconds. If set to *updated+15*, it will be included immediately if the value is updated, or otherwise once every 15 seconds. The `prec` component names the precision if the value is numerical. A value of say 3, will be posted with three digits after the decimal.

52.2.4 Enabling Sparse Node Report Intervals

By default, a node report is generated on each iteration of `pNodeReporter`. In certain situations a reduction in the frequency may be appropriate. Each node report contributes both to the local alog file size, and consumes inter-vehicle communications bandwidth. If the vehicle is traversing a long straight leg, with little to no deviation in heading or speed, the vehicle position can be inferred by extrapolation. The node report frequency can perhaps be reduced from say 4Hz, to 0.5Hz.

Extrapolation is used in two parts:

- The sender of reports, `pNodeReporter`, extrapolates from the last posted report, for the time since the last posted report, and compares the difference. If the difference between the extrapolated position and the actual current position exceeds a configured threshold, then a node report with the current position is posted.
- The receiver of reports, e.g., other vehicles, or `pMarineViewer`, or similar, use extrapolation from the last received node report to update local vehicle position state.

The following configuration parameters may be used to reduce the frequency of node reports:

The `extrap_enabled` parameter must set to `true` to enable reduced posting frequency through extrapolation. By default it is set to `false`.

The `extrap_pos_thresh` parameter is a value given in meters. The current vehicle position is compared against the position extrapolated from the last reported position assuming the heading in the last reported position, and the time since the last reported position. If the distance between these two positions is greater than this threshold, a node report is generated regardless of other configuration parameters.

The `extrap_hdg_thresh` parameter is a value given in degrees. The current vehicle heading is

compared to the heading in the most recently posted node report. If the difference in heading exceeds the threshold, then a node report is generated regardless of other configuration parameters.

The extrap_max_gap parameter is a value given in seconds. The current time is compared to the time of the most recently posted node report. If the difference in time exceeds the threshold, then a node report is generated regardless of other configuration parameters.

The default values for these configuration parameters are:

```
extrap_enabled    = false
extrap_pos_thresh = 0.25    // meters
extrap_hdg_thresh = 1       // degrees
extrap_max_gap    = 5       // seconds
```

52.2.5 Platform Characteristics

The node report contains four fields regarding the platform characteristics, NAME, TYPE, LENGTH, and BEAM. The name of the platform is equivalent to the name of the MOOS community within which pNodeReporter is running. The MOOS community is declared as a global MOOS parameter (outside any given process' configuration block) in the .moos mission file. The TYPE, LENGTH, and BEAM parameters are set in the pNodeReporter configuration block.

Example:

```
platform_length = 16              // meters
platform_beam   = 15              // meters
platform_type   = kayak
platform_color  = dodger_blue
```

If the platform type is known, but no information about the platform length is known, certain rough default values may be used if the platform type matches one of the following: "kayak" maps to 4 meters, "mokai" maps to 4 meters, "uuv" maps to 4 meters, "auv" maps to 4 meters, "ship" maps to 18 meters, "glider" maps to 3 meters.

52.2.6 Dealing with Local versus Global Coordinates

A primary component of the node report is the current position of the vehicle. The pNodeReporter application subscribes for the following MOOS variables to garner this information: NAV_X, NAV_Y in local coordinates, and the pair NAV_LAT, NAV_LONG in global coordinates. These two pairs should be consistent, but what if they are not? And what if pNodeReporter is receiving mail for one pair but not the other? Four distinct policy choices are supported, set in the configuration parameter crossfill_policy:

- The default policy: node reports include exactly what is given. If NAV_X and NAV_Y are being received only, then there will be no entry in the node report for global coordinates, and vice versa. If both pairs are being received, then both pairs are reported. No attempt is made

to check or ensure that they are consistent. This is the default policy, equivalent to the configuration crossfill_policy=literal.

- If one of the two pairs is not being received, pNodeReporter will fill in the missing pair from the other. This policy can be chosen with the configuration cross_fill_policy=fill-empty.
- If NAV_LAT and NAV_LONG are being received, pNodeReporter will use these in its report, *and* convert the global coordinates to local coordinates, and use these converted coordinates in its report, essentially disregarding NAV_X and NAV_Y if they are also being received. This policy can be chosen with the configuration cross_fill_policy=global. This is available in the next release after Release 19.8.x.
- If one of the two pairs has been received more recently, the older pair is updated by converting from the other pair. The older pair may also be in a state where it has never been received. This policy can be chosen with the configuration cross_fill_policy=fill-latest.

An additional configuration param, coord_global_policy, accepts a Boolean value, by default `false`. When it is set to true, it will mask out the x/y portion of the node report, leaving only the lat/lon portion. The manner in which the lat/lon is set is determined by the crossfill_policy configuration parameter.

52.2.7 Processing Alternate Navigation Solutions

Under normal circumstances, node reports are generated reflecting the current navigation solution as defined by the incoming NAV_* variables. The pNodeReporter application can handle the case where the vehicle also publishes an alternate navigation solution, as defined by a sister set of incoming MOOS variables separate from the NAV_* variables. In this case pNodeReporter will monitor both sets of variables and may generate *two* node reports on each iteration. The following two configuration parameters are needed to activate this capability:

```
alt_nav_prefix = <prefix>      // example: NAV_GT_
alt_nav_name   = <node-name>   // example: _GT
alt_nav_group  = <group-name>  // example: ground_truth (Post R.19.8.x)
```

The configuration parameter, alt_nav_prefix, names a prefix for the alternate incoming navigation variables. For example, alt_nav_prefix=NAV_GT_ would result in pNodeReporter subscribing for NAV_GT_X, NAV_GT_Y and so on. A separate vehicle state would be maintained internally based on this alternate set of navigation information and a second node report would be generated. The alt_nav_group allows the alt nav node report to have a different group name than the baseline node report. If not specified, the group names will match. This feature was introduced after Release 19.8.x.

A second node report would be published under the same MOOS variable, NODE_REPORT_LOCAL, but the NAME component of the report would be distinct based on the value provided in the alt_nav_name parameter. If a name is provided that does not begin with an underscore character, that name is used. If the name does begin with an underscore, the name used in the report is the otherwise configured name of the vehicle plus the suffix.

52.2.8 Publishing Node Reports in JSON Format

Post Release 24.8, node reports may optionally be published in JSON format. Either (a) the NODE_REPORT_LOCAL message can be made in JSON format, or (b) a second MOOS variable can be specified to be additionally published in JSON format. The deserialization function, converting node report strings into a C++ record, will auto-detect the format. All apps in MOOS-IvP ingesting node reports use this function.

To publish NODE_REPORT_LOCAL in JSON format, simply use:

```
  json_report = true
```

If the value of the argument is any string other than `true`, or `false`, case insensitive, the value is interpreted to mean a MOOS variable to which a second node report will be published in JSON format. For example, consider the below configuration:

```
  json_report = NODE_REPORT_JSON
```

Both the comma-separated-pair (CSP) format, and JSON format will be published:

```
NODE_REPORT_JSON = {"NAME":"abe","SPD":1.9,"HDG":190.45,"DEP":0,"LAT":43.82472523,
    "LON":-70.33059595,"TYPE":"kayak","COLOR":"yellow",
    "MODE":"MODE@ACTIVE:LOITERING","ALLSTOP":"clear",
    "INDEX":113,"YAW":-1.753119,"TIME":17398343683.8,"LENGTH":4}

NODE_REPORT_LOCAL = NAME=abe,SPD=1.9,HDG=190.45,DEP=0,LAT=43.82472523,
    LON=-70.33059595,TYPE=kayak,COLOR=yellow,
    MODE=MODE@ACTIVE:LOITERING,ALLSTOP=clear,
    INDEX=113,YAW=-1.753119,TIME=17398343683.8,LENGTH=4
```

52.3 The Optional Blackout Interval Option

Under normal circumstances, the pNodeReporter application will post a node report once per iteration, the gap between postings being determined solely by the app_tick parameter (Figure 170). However, there are times when it is desirable to add an artificial delay between postings. Node reports are typically only useful as information sent to another node, or to a shoreside computer rendering fielded vehicles, and there are often dropped node report messages due to the uncertain nature of communications in the field, whether it be acoustic communications, WiFi, or satellite link.

Applications receiving node reports usually implement provisions that take dropped messages into account. A collision-avoidance or formation-following behavior, or a contact manager, may extrapolate a contact position from its last received position and trajectory. A shoreside command-and-control GUI such as pMarineViewer may render an interpolation of vehicle positions between node reports. To test the robustness of applications needing to deal with dropped messages, a way of simulating the dropped messages is desired. One way is to add this to the simulation version of whatever communications medium is being used. For example, there is an acoustic communications

simulator where the dropping of messages may be simulated, where the probability of a drop may even be tied to the range between vehicles. Another way is to simply simulate the dropped message at the source, by adding delay to the posting of reports by pNodeReporter.

By setting the blackout_interval parameter, pNodeReporter may be configured to ensure that a node report is not posted until *at least* the duration specified by this parameter has elapsed, as shown in Figure 171.

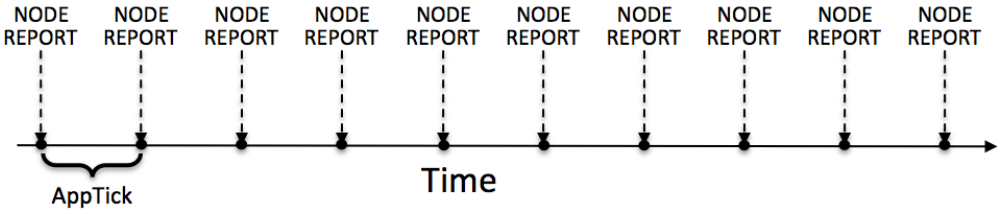

Figure 170: **Normal schedule of node report postings**: The pNodeReporter application will post node reports once per application iteration. The duration of time between postings is directly tied to the frequency at which pNodeReporter is configured to run, as set by the standard MOOS AppTick parameter.

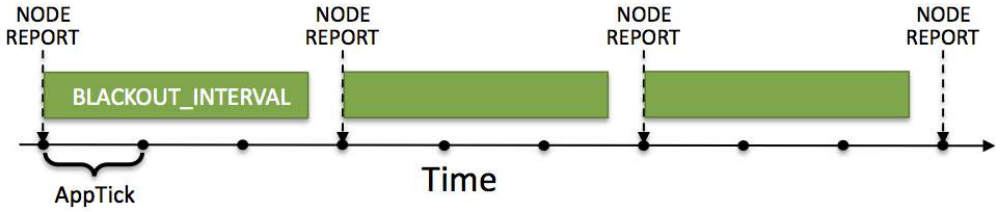

Figure 171: **The optional blackout interval parameter:** The schedule of node report postings may be altered by the setting the BLACKOUT_INTERVAL parameter. Reports will not be posted until at least the time specified by the blackout interval has elapsed since the previous posting.

An element of unpredictability may be added by specifying a value for the blackout_variance parameter. This parameter is given in seconds and defines an interval $[-t, t]$ from which a value is chosen with uniform probability, to be added to the duration of the blackout interval. This variation is re-calculated after each interval determination. The idea is depicted in Figure 172.

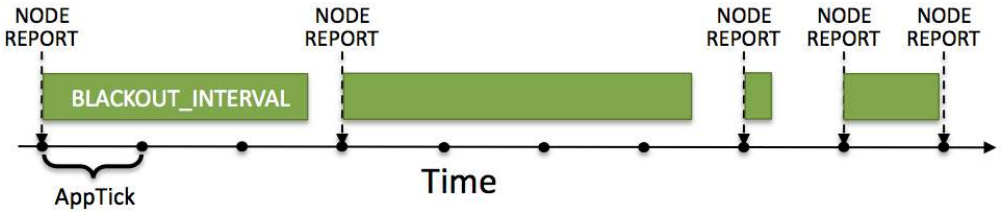

Figure 172: **Blackout intervals with varying duration:** The duration of a blackout interval may be configured to vary randomly within a user-specified range, specified in the blackout_variance parameter.

Message dropping is typically tied semi-predictably to characteristics of the environment, such as range between nodes, water temperature or platform depth, an so on. This method of simulating dropped messages captures none of that. It is however simple and allows for easily proceeding with the testing of applications that need to deal with the dropped messages.

52.4 Configuration Parameters for pNodeReporter

The following parameters are defined for pNodeReporter. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so in parentheses.

Listing 52.77: Configuration Parameters for pNodeReporter.

alt_nav_prefix:	Source for processing alternate nav reports. Section 52.2.7.
alt_nav_group:	Group associated with node reports generated for alternate nav node reports. Introduced after Release 19.8.x. Section 52.2.7.
alt_nav_name:	Node name in posting alternate nav reports. Section 52.2.7.
blackout_interval:	Minimum duration, in seconds, between reports (0). Section 52.3.
blackout_variance:	Variance in uniformly random blackout duration. Legal values: any non-negative value. The default is zero. Section 52.3.
coord_policy_global:	If true (default is false), then only lat/lon coordinates are published in the node report; no local x/y coordinates. Section 52.2.6.
crossfill_policy:	Policy for handling local versus global nav reports (`"literal"`). Section 52.2.6.
extrap_enabled:	If `true` (the default is `false`), then a node report is not generated on every iteration, but can instead be held back for some time, assuming the consumers of the node reports are capable of extrapolating the vehicle position. This can reduce network traffic and log file size, but must be used with care and when sure that consumers are capable of extrapolation. Section 52.2.4.
extrap_hdg_thresh:	If extrapolation is enabled, this parameter, in degrees, determines a threshold for posting a new node report. If the vehicle heading has changed by degrees set here, a new node report must be posted. The default is 1 degree. Section 52.2.4.
extrap_max_gap:	If extrapolation is enabled, this parameter, in seconds, determines a threshold for posting a new node report. If node report has not been published in the time set here, a new node report must be posted. The default is 5 seconds. Section 52.2.4.
extrap_pos_thresh:	If extrapolation is enabled, this parameter, in meters, determines a threshold for posting a new node report. If the vehicle position has changed by this distance, a new node report must be posted. The default is `0.25` meters. Section 52.2.4.
json_report:	If `true`, the node report will be published in JSON format. If set to something other than true, e.g., `FOOBAR`, a second node report will be published, in JSON format, to `FOOBAR`, and the normal node report in CSP format will be published as normal. Section 52.2.8.
node_report_output:	MOOS variable used for the node report (`NODE_REPORT_LOCAL`). Section 52.2.1.

nav_grace_period: A duration, in seconds, before a run warning is posted due to mission navigation information, e.g, NAV_LAT or NAV_LONG.

nohelm_threshold: Seconds after which a quiet helm is reported as AWOL. Legal values: any non-negative value. The default is 5 seconds. Section 52.2.2.

platform_aspect: A custom param=value pair to be include in all node reports. Section 52.2.3.

platform_beam: The reported beam length (from side-to-side at the widest point) of the platform in meters. Legal values: any non-negative value. The default is zero. Section 52.2.5.

platform_length: The reported length of the platform in meters. Legal values: any non-negative value. The default is zero. Section 52.2.5.

plat_report_output: The Platform report MOOS variable. Legal values: conventions for MOOS variable names. The default is PLATFORM_REPORT_LOCAL. Section 52.6.

plat_report_input: A component of the optional platform report. Section 52.6.

platform_color: A hint on how to render the vehicle in a GUI application like pMarineViewer or alogview. Introduced in Release 16.5. Section 52.2.5.

platform_type: The reported type of the platform. Legal values: any string. The default is "unknown". Section 52.2.5.

paused: If true, posting of reports is suspended. The default is "false". With release 15.3.

rider: A custom augmentation of the node report can be configured, by registering for one or more MOOS variables and reformatting the output and tagging onto (riding) the end of a node report. Post 24.8. See Section 52.2.3.

52.4.1 An Example MOOS Configuration Block

An example MOOS configuration block is provided in Listing 78 below. To see an example MOOS configuration block from the console, enter the following:

```
$ pNodeReporter --example or -e
```

This will show the output shown in Listing 78 below.

Listing 52.78: Example configuration of pNodeReporter.

```
1  ============================================================
2  pNodeReporter Example MOOS Configuration
3  ============================================================
4  Blue lines: Default configuration
5
6  ProcessConfig = pNodeReporter
7  {
8    AppTick   = 4
9    CommsTick = 4
10
11   // Configure key aspects of the node
12   platform_type      = glider   // or {uuv,auv,ship,kayak}
13   platform_length    = 8        // meters.  Range [0,inf)
14
15   // Configure optional blackout functionality
```

```
16    blackout_interval  = 0        // seconds. Range [0,inf)
17
18     // Configure the optional platform report summary
19    plat_report_input  = COMPASS_HEADING, gap=1
20    plat_report_input  = GPS_SAT, gap=5
21    plat_report_input  = WIFI_QUALITY, gap=1
22    plat_report_output = PLATFORM_REPORT_LOCAL
23
24    // Configure the MOOS variable containg the node report
25    node_report_output = NODE_REPORT_LOCAL
26
27    // Threshold for conveying an absense of the helm
28    nohelm_threshold   = 5        // seconds
39
30    // Policy for filling in missing lat/lon from x/y or v.versa
31    // Valid policies: [literal], fill-empty, use-latest, global
32    crossfill_policy   = literal
33
34    // Configure monitor/reporting of dual nav solution
35    alt_nav_prefix     = NAV_GT
36    alt_nav_name       = _GT
37  }
```

52.5 Publications and Subscriptions for pNodeReporter

The interface for pNodeReporter, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pNodeReporter --interface or -i
```

52.5.1 Variables Published by pNodeReporter

The primary output of pNodeReporter to the MOOSDB is the node report and the optional platform report:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- NODE_REPORT_LOCAL: Primary summary of the node's navigation and helm status. Section 52.2.1.
- PLATFORM_REPORT_LOCAL: Optional summary of certain platform characteristics. Section 52.2.5.

52.5.2 Variables Subscribed for by pNodeReporter

Variables subscribed for by pNodeReporter are summarized below. A more detailed description of each variable follows. In addition to these variables, any MOOS variable that the user requests to be included in the optional PLATFORM_REPORT will also be automatically subscribed for.

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- IVPHELM_STATE: A indicator of the helm state produced by pHelmIvP, e.g., "PARK", "DRIVE" "DISABLED", or "STANDBY".

- IVPHELM_ALLSTOP: A indicator of the helm allstop produced by pHelmIvP, e.g., "clear", or a reason why the vehicle is at zero speed.
- IVPHELM_SUMMARY: A summary report produced by the IvP Helm (pHelmIvP).
- NAV_X: The ownship vehicle position on the x axis of local coordinates.
- NAV_Y: The ownship vehicle position on the y axis of local coordinates.
- NAV_LAT: The ownship vehicle position on the y axis of global coordinates.
- NAV_LONG: The ownship vehicle position on the x axis of global coordinates.
- NAV_HEADING: The ownship vehicle heading in degrees.
- NAV_YAW: The ownship vehicle yaw in radians.
- NAV_SPEED: The ownship vehicle speed in meters per second.
- NAV_DEPTH: The ownship vehicle depth in meters.
- PNR_PAUSE: Allows the paused state to be set or toggled. Acceptable values are "true" "false" "toggle". With release 15.3.

If pNodeReporter is configured to handle a second navigation solution as described in Section 52.2.7, the corresponding additional variables described in that section will also be automatically subscribed for.

52.5.3 Command Line Usage of pNodeReporter

The pNodeReporter application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. The basic command line usage for the pNodeReporter application is the following:

Listing 52.79: Command line usage for the pNodeReporter application.

```
Usage: pNodeReporter file.moos [OPTIONS]

Options:
  --alias=<ProcessName>
      Launch pNodeReporter with the given process name
      rather than pNodeReporter.
  --example, -e
      Display example MOOS configuration block.
  --help, -h
      Display this help message.
  --version,-v
      Display the release version of pNodeReporter.
```

52.6 The Optional Platform Report Feature

The pNodeReporter application allows for the optional reporting of another user-specified list of information. This report is made by posting the PLATFORM_REPORT_LOCAL variable. An alternative variable name may be used by setting the PLAT_REPORT_SUMMARY configuration parameter. This report may be configured by specifying one or more components in the pNodeReporter configuration block, of the following form:

```
plat_report_input = <variable>, gap=<duration>, alias=<variable>
```

If no component is specified, then no platform report will be posted. The <variable> element specifies the name of a MOOS variable. This variable will be automatically subscribed for by pNodeReporter and included in (not necessarily all) postings of the platform report. If the variable BODY_TEMP is specified, a component of the report may contain "BODY_TEMP=98.6". An alias for a MOOS variable may be specified. For example, alias=T, for the BODY_TEMP component would result in "T=98.6" in the platform report instead.

How often is the platform report posted? Certainly it will not be posted any more often than the apptick parameter allows, but it may be posted far more infrequently depending on the user configuration and how often the values of its components are changing. The platform report is posted only when one or more of its components requires a re-posting. A component requires a re-posting only if (a) its value has changed, *and* (b) the time specified by its gap setting has elapsed since the last platform report that included that component. When a PLATFORM_REPORT_LOCAL posting is made, only components that required a posting will be included in the report.

The wide variation in configurations of the platform report allow for reporting information about the node that may be very specific to the platform, not suitable for a general-purpose node report. As an example, consider a situation where a shoreside application is running to monitor the platform's battery level and whether or not the payload compartment has suffered a breach, i.e., the presence of water is detected inside. A platform report could be configured as follows:

```
plat_report_input = ACME_BATT_LEVEL, gap=300, alias=BATTERY_LEVEL
plat_report_input = PAYLOAD_BREACH
```

This would result in an initial posting of:

```
PLATFORM_REPORT_LOCAL = "platform=alpha,utc_time=1273510720.99,BATTERY_LEVEL=97.3,
                        PAYLOAD_BREACH=false"
```

In this case, the platform uses batteries made by the ACME Battery Company and the interface to the battery monitor happens to publish its value in the variable ACME_BATT_LEVEL, and the software on the shoreside that monitors all vehicles in the field accepts the generic variable BATTERY_LEVEL, so the alias is used. It is also known that the ACME battery monitor output tends to fluctuate a percentage point or two on each posting, so the platform report is configured to include a battery level component no more than once every five minutes, (gap=300). The MOOS process monitoring the indication of a payload breach is known to have few false alarms and to publish its findings in the variable PAYLOAD_BREACH. Unlike the battery level which has frequent minor fluctuations and degrades slowly, the detection of a payload breach amounts to the flipping of a Boolean value and needs to be conveyed to the shoreside as quickly as possible. Setting gap=0, the default, ensures that a platform report is posted on the very next iteration of pNodeReporter, presumably to be read by a MOOS process controlling the platform's outgoing communication mechanism.

52.7 An Example Platform Report Configuration Block for pNodeReporter

Listing 80 below shows an example configuration block for pNodeReporter where an extensive platform report is configured to report information about the autonomous kayak platform to

support a "kayak dashboard" display running on a shoreside computer. Most of the components in the platform report are specific to the autonomous kayak platform, which is precisely why this information is included in the platform report, and not the node report.

Listing 52.80: An example pNodeReporter configuration block.

```
//------------------------------------------------------------
// pNodeReporter config block

ProcessConfig = pNodeReporter
{
  AppTick  = 2
  CommsTick = 2

  platform_type       = KAYAK
  platform_length     = 3.5     // Units in meters
  nohelm_thresh       = 5       // The default
  blackout_interval   = 0       // The default
  blackout_variance   = 0       // The default

  node_report_output  = NODE_REPORT_LOCAL        // The default
  plat_report_output  = PLATFORM_REPORT_LOCAL    // The default

  plat_report_input = COMPASS_PITCH, gap=1
  plat_report_input = COMPASS_HEADING, gap=1
  plat_report_input = COMPASS_ROLL, gap=1
  plat_report_input = DB_UPTIME, gap=1
  plat_report_input = COMPASS_TEMPERATURE, gap=1, alias=COMPASS_TEMP
  plat_report_input = GPS_MAGNETIC_DECLINATION, gap=10, alias=MAG_DECL
  plat_report_input = GPS_SAT, gap=5
  plat_report_input = DESIRED_RUDDER, gap=0.5
  plat_report_input = DESIRED_HEADING, gap=0.5
  plat_report_input = DESIRED_THRUST, gap=0.5
  plat_report_input = GPS_SPEED, gap=0.5
  plat_report_input = DESIRED_SPEED, gap=0.5
  plat_report_input = WIFI_QUALITY, gap=0.5
  plat_report_input = WIFI_QUALITY, gap=1.0
  plat_report_input = MOOS_MANUAL_OVERRIDE, gap=1.0
}
```

52.8 Measuring the Odometry Extent Per Mission Hash

A new feature, *after* Release 22.8, is introduced to enable pNodeReporter to post useful information to be logged with the vehicle alog file. This coincides with the introduction of a *mission hash*. The mission hash is published by the shoreside and shared to all vehicles and logged. It has the form:

```
MISSION_HASH = mhash=221119-1255K-NEAR-MILK,utc=1668880538.69
```

The posting contains both the hash itself, `221119-1255K-NEAR-MILK`, and the UTC timestamp when the hash was created on the shoreside. The first part of the hash reflects the time and date, `241119-1255`, Nov 19th, 2024, 12:55. The second part of the hash is comprised of randomly configured words that are easier to remember. The final two-word part of the hash can be used as an alias in some applications.

The shoreside generates the hash, and it cannot be ruled out that the shoreside generates multiple hashes for a single vehicle alog file. This can happen for example if the shoreside app, e.g., pMarineViewer, generating the mission hash, is re-started during the vehicle deployment. In this case the vehicle alog file may have multiple mission hashes, leading to confusion. This is where pNodeReporter can help.

pNodeReporter will register for MISSION_HASH and use an odometer to track the maximum vehicle extent associated with this hash. The *extent* is the maximum distance from the odometer starting point, i.e., the farthest linear distance from the starting point at any time in the mission. The node reporter will publish to PNR_MHASH whenever this maximum extent has reached a new max value. If the MISSION_HASH value should change mid-mission, the PNR_MHASH value will also change.

If the mission hash changes mid-mission, we may see a set of alog entries from pMarineViewer along the lines of:

```
14.853   PNR_MHASH  mhash=221119-1412M-GOLD-EPIC,ext=2.6
15.874   PNR_MHASH  mhash=221119-1412M-GOLD-EPIC,ext=3.9
16.897   PNR_MHASH  mhash=221119-1412M-GOLD-EPIC,ext=5.4
22.538   PNR_MHASH  mhash=221119-1412M-GOLD-EPIC,ext=6.9
23.560   PNR_MHASH  mhash=221119-1412M-GOLD-EPIC,ext=8.3
24.587   PNR_MHASH  mhash=221119-1412M-GOLD-EPIC,ext=9.5
26.119   PNR_MHASH  mhash=221119-1412M-GOLD-EPIC,ext=10.6
177.068  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=12.1
178.093  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=13.5
179.117  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=14.9
180.129  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=16.3
181.159  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=17.8
182.172  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=19.2
183.207  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=20.4
184.228  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=21.8
218.643  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=23.2
219.682  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=24.3
220.693  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=25.4
222.219  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=26.9
223.762  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=28.3
225.317  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=29.8
226.876  PNR_MHASH  mhash=221119-1413E-MAIN-KHAN,ext=31.1
```

In this case the mission hash was changed sometime between the offset times of 26.119 and 177.068. When pNodeReporter noticed that the max extent distance accumulated while the mission hash was "MAIN-KHAN" had exceeded 10.6, a new PNR_MHASH posting was made.

In a nutshell, if there was unfortunately more than one mission hash received by a vehicle and logged in a single alog file, this information can help determine which mission hash was the "real" one. It should always be the hash associated with the final posting to PNR_MHASH. And this information can be obtained using a few aloggrep switches:

```
$ aloggrep file.alog --final --no_report PNR_MHASH
226.876  PNR_MHASH    pNodeReporter   mhash=221119-1413E-MAIN-KHAN,ext=31.1
```

Or, to isolate the hash itself:

```
$ aloggrep file.alog --final --format=val --subpat=mhash PNR_MHASH
221119-1413E-MAIN-KHAN
```

The latter example is essentially the whole point of this feature: to enable aloggrep to determine the most likely single mission hash for a give alog file, even if multiple hashes exist. This tool can then be used as a component of other scripts that are performing post-mission log file archiving tasks.

53 uXMS: Scoping the MOOSDB from the Console

53.1 Overview

uXMS is a terminal based MOOS app for scoping the MOOSDB. It has no graphics library build dependencies and is easily launched from the command line to scope on just what the user wants, from everything to just one important variable. For example, typing the following on the command line after say the alpha example mission is launched, will result in Figure 173.

```
$ uXMS alpha.moos NAV_X NAV_Y DEPLOY RETURN IVPHELM_STATE
```

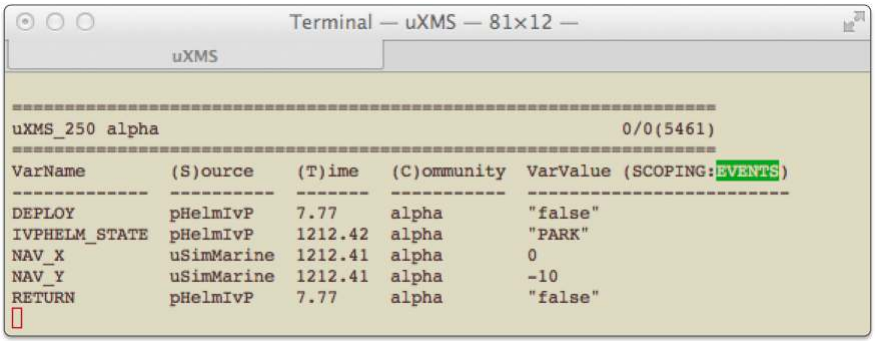

Figure 173: **A simple scope on five variables with uXMS:** A line is used for each variable showing the variable name, the time of the most recent posting, the source, and the current value.

Scoping on the MOOSDB is a very important tool in the process of development and debugging. The uXMS tool has a substantial set of configuration choices for making this job easier by bringing just the right data to the user's attention. The default usage, as shown above, is fairly simple, but there are other options discussed in this section that are worth exploiting by the more experienced user.

- *Use with appcasting:* uXMS is appcast enabled meaning its terminal reports may be viewed with tools other than the terminal window. It is possible to configure multilple uXMS scopes to automatically launch with a mission and viewer each of them in remote appcast viewing tools. More on this topic in Section 53.9.
- *Scoping on history:* uXMS may be configured to scope on the history of a variable to view not just its current state but recent values.
- *Remote low-bandwidth scoping:* uXMS can be launched and connected to a remote MOOSDB over a low-bandwidth link, with refresh requests made only on the user's request. uXMS can also be on a remote vehicle via an ssh session.
- *Dynamic changes to the scope list:* The set of scoped variables may be altered dynamically by selecting MOOS apps to include or exclude from the scope list.

At any time the user may hit the 'h' key to see a list of help commands.

53.2 The uXMS Refresh Modes

Reports such as the one shown in Figure 173 are generated either automatically or specifically when the user asks for it. The latter is important in situations where bandwidth is low. This feature was the original motivation for developing uXMS. When a new report is sent to the terminal is determined by the *refresh mode*. The three refresh modes are shown in Figure 174 along with the key strokes for switching between modes.

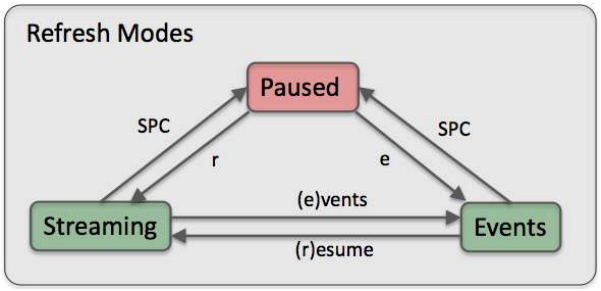

Figure 174: **Refresh Modes:** The `uXMS` refresh mode determines when a new report is written to the screen. The user may switch between modes with the shown keystrokes.

The refresh mode may be changed by the user as uXMS is running, or it may be given an initial mode value on startup from the command line with `--mode=paused`, `--mode=streaming`, or `--mode=events`. The latter is the default. It may also be set in the uXMS configuration block in the mission file with the `refresh_mode` parameter. The current refresh mode is shown in parentheses in the report header as shown in Figure 173 where it is in the *events* refresh mode.

53.2.1 The Streaming Refresh Mode

In the *streaming* refresh mode, a new report is generated and written to stdout on every iteration of the uXMS application. The frequency is limited from above by the apptick setting in the MOOS configuration block. It is also limited from above by the parameter `term_report_interval`, which is by default 0.6 seconds. Each report written to the terminal will show an incremented counter at the end of the first line, in parentheses. This counter represents the uXMS iteration counter. This mode may be entered by hitting the `'r'` key, or chosen as the initial refresh mode at startup from the command line with the `--mode=streaming` option.

53.2.2 The Events Refresh Mode

In the *events* refresh mode, the default refresh mode, a new report is generated only when new mail is received for one of the scoped variables. Note this does not necessarily mean that the value of the variable has changed, only that it has been written to again by some process. As with the *streaming* mode, the report frequency is limited by the apptick and the `term_report_interval` setting. This mode is useful in low-bandwidth situations where a user cannot afford the streaming refresh mode, but may be monitoring changes to one or two variables. This mode may be entered by hitting the `'e'` key, or chosen as the initial refresh mode at startup from the command line with the `--mode=events` option.

The Paused Refresh Mode In the *paused* refresh mode, the report will not be updated until the user specifically requests a new update by hitting the spacebar key. This mode is the preferred mode in low bandwidth situations, and simply as a way of stabilzing the rapid refreshing output of the other modes so one can actually read the output. This mode is entered by the spacebar key and subsequent hits refresh the output once. To launch uXMS in the paused mode, use the `--mode=paused` command line switch.

53.3 The uXMS Content Modes

The contents of the uXMS report vary between one of a few modes. In the *scoping* mode, a snapshot of a subset of MOOS variables is generated, similar to what is shown in Figure 173. In the *history* mode the recent history of changes to a single MOOS variable is reported.

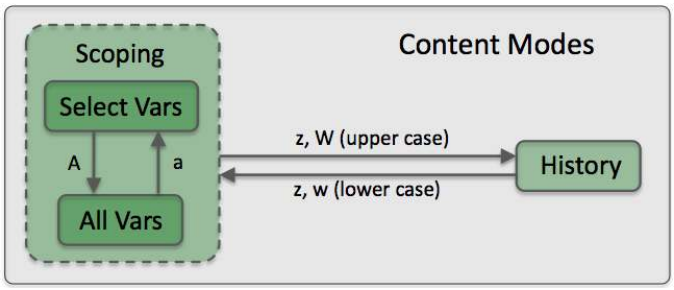

Figure 175: **Content Modes:** The `uXMS` content mode determines what data is included in each new report. The two major modes are the *scoping* and *history* modes. In the former, snapshots of one or more MOOS variables are reported. In the latter, the recent history of a single variable is reported.

53.3.1 The Scoping Content Mode

The *scoping* mode has two sub-modes as shown in Figure 175. In the first sub-mode, the *SelectVars* sub-mode, the only variables shown are the ones the user requested. They are requested on the command-line upon start-up (Section 53.5), or in the uXMS configuration block in the `.moos` file provided on startup, or both. One may also select variables for viewing by specifying one or MOOS processes with the command line option `--src=<process>,<process>,...`. All variables from these processes will then be included in the scope list.

In the *AllVars* sub-mode, all MOOS variables in the MOOSDB are displayed, unless explicitly filtered out. The most common way of filtering out variables in the *AllVars* sub-mode is to provide a filter string interactively by typing the `'/'` key and entering a filter. Only lines that contain this string as a substring in the variable name will then be shown. The filter may also be provided on startup with the `--filter=pattern` command line option.

In both sub-modes, variables that would otherwise be included in the report may be masked out with two further options. Variables that have never been written to by any MOOS process are referred to as virgin variables, and by default are shown with the string `"n/a"` in their value column. These may be shut off from the command line with `--mask=virgin`, or in the MOOS configuration block by including the line `display_virgins=false`. Similarly, variables with an empty string value may be masked out from the command line with `--mask=empty`, or with the line `display_empty_strings=false` in the MOOS configuration block of the `.moos` file.

53.3.2 The History Content Mode

In the *history* content mode, the recent values for a single MOOS variable are reported. Contrast this with the *scoping* mode where a snapshot of a variable value is displayed, and that value may have changed several times between successive reports. The output generated in this mode may look like that in Figure 176 which shows the desired heading of a vehicle going into the first turn of the alpha mission. This `uXMS` session can by launched from the command line with:

```
$ uXMS alpha.moos --history=DESIRED_HEADING
```

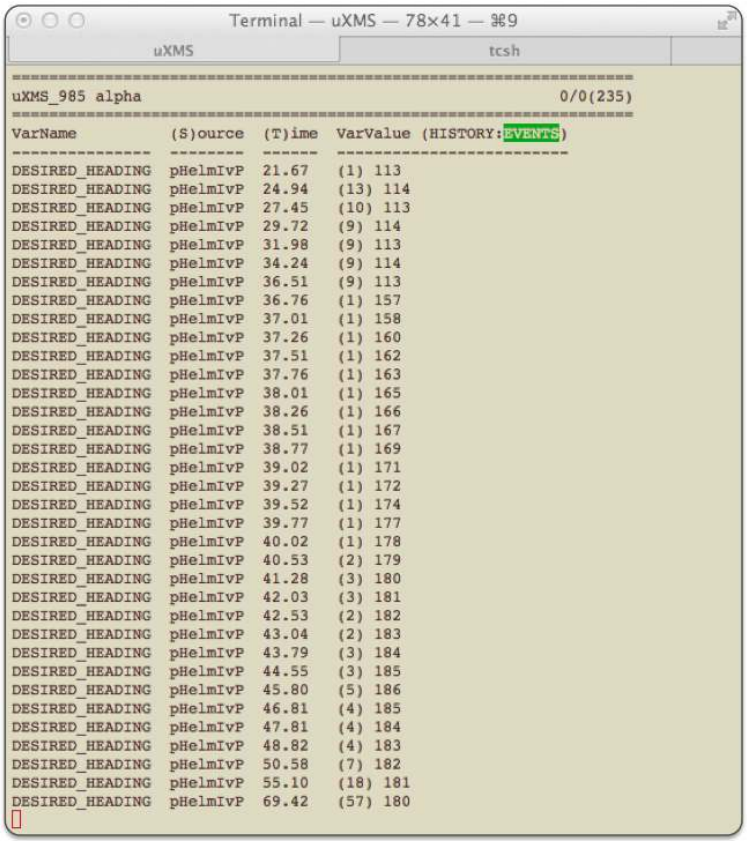

Figure 176: **A uXMS scope on a single variable history:** A vehicle's desired heading is monitored as it goes into the first turn of the alpha mission. Values in parentheses indicate the number of successive postings without a change of value.

The output structure in the *history* mode is the same as in the *scoping* mode in terms of what data is in the columns and header lines. Each line however is dedicated to the same variable and shows the progression of values through time. To save screen real estate, successive mail received for with identical source and value will consolidated on one line, and the number in parentheses is merely incremented for each such identical mail. For example, the last line shown in Figure 176, the value of `DESIRED_HEADING` has remained the same for 57 consecutives posts to the MOOSDB.

The output in the *history* mode may be adjusted in a few ways:

- *Modifying the number of history lines:* The number of lines of history may be increased or descreased by hitting the '>' or '<' keys respectively. A maximum of 100 and minimum of 5 lines is allowed. The default is 40.
- *Setting the history variable:* The history variable may be set on the command line with `--history=VAR`, or set in the the mission file with `history_var=<MOOSVar>`. If set in both, the command line setting takes precedent.
- *Hiding the history variable:* To increase the available real estate on each line, the variable name column may be supressed or restored by toggling the 'j' key. The history variable is shown by default but may be configured to be off upon startup by setting `display_history_var=false` in the mission file.

Presently there is no way to dynamically change the history variable, or scope on more than one variable's history. (But you can open more than one uXMS session to scope on more than one variable's history.)

53.3.3 The Processes Content Mode

In the *processes* content mode running processes may be monitored and selected for either including or excluding variables from the selected processes. This mode may be toggled with the 'p' key. For example, launching the alpha mission and then uXMS from the command line:

```
$ uXMS alpha.moos DESIRED_HEADING
```

After uXMS is launched, toggle into the procesess content mode with the 'p'. This should present something similar to Figure 177.

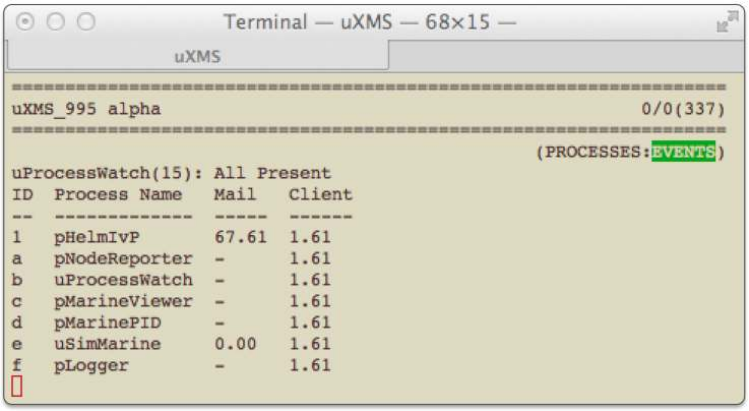

Figure 177: **A uXMS processes content mode:** All processes known to the scope (via the DB_CLIENTS variable) are shown. The Mail column shows the time since mail has been received from the client. The Client column shows the time since the client has shown up on the DB_CLIENTS list.

The first two columns show the processes known to uXMS and a process ID randomly assigned to each process. These IDs may be used select a process as explaine shortly. The last two columns

show the time since mail was last received and the time since the process last appeared on the DB_CLIENTS list. Try killing one of the processes and see what happens.

By default uXMS tries to make use of information produced by uProcessWatch. The first line in the body of the report in Figure 177 shows the contents of the PROC_WATCH_SUMMARY variable. In this example, mail has only been received from pHelmIvP and uProcessWatch. The former because uXMS was launched from the command line scoping on DESIRED_HEADING, and the latter because uXMS is automatically configured to received the PROC_WATCH_SUMMARY mail from uProcessWatch. If uProcessWatch is not running the report will simply state so.

Perhaps the most useful feature of the *processes* content mode is the ability to select a process to either include or exclude variables published by that process on the watch list. To *include* variables from a process, type the '+' key, and a menu and prompt like that shown in Figure 178

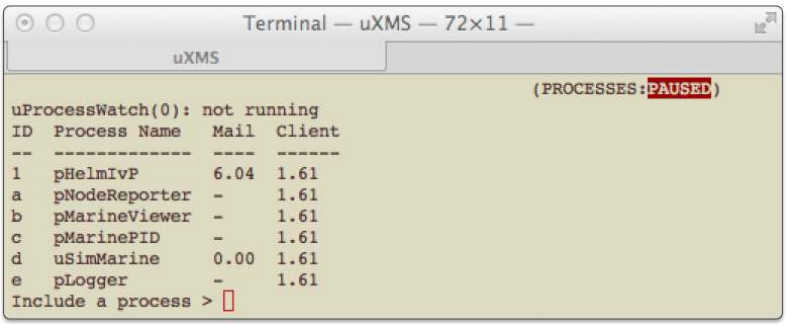

Figure 178: **Adding watch-list variables based process inclusion:** All processes known to the scope (via the DB_CLIENTS variable) are shown. Each process has a single-character ID which may be entered at the prompt to select the process for inclusion.

Once a process has been selected, uXMS will subscribe for all variables published by the selected process. Note this is different from subscribing for mail solely produced by the selected MOOS app since the same variable(s) may be also published by other MOOS applications. The example in Figure 178 is also from the alpha mission. If the pMarineViewer application were selected, a scoping report something like that below in Figure 179 would result.

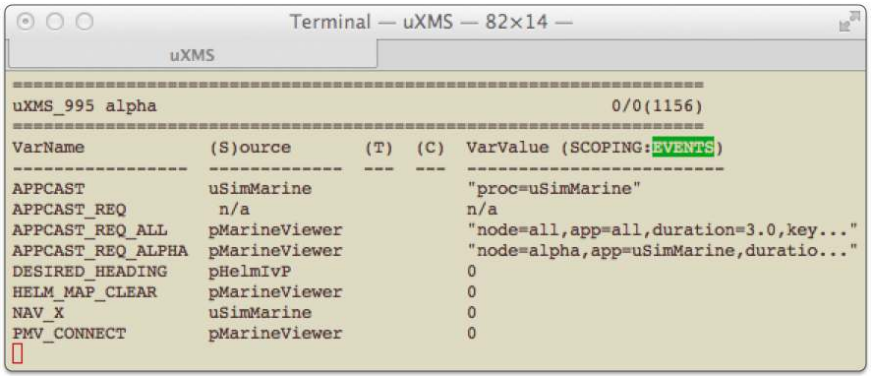

Figure 179: **Augmented scoping report:** The variables published by pMarineViewer are now included in the scoping report after this process was selected for inclusion.

Once a process has been added or excluded, its status will be indicated next time the processes mode is entered, with either a '+' or '-' next to the process ID. For example, the report shown in Figure 180 below is generated after hitting the '-' key to select a process for exclusion. The plus sign next to the pMarineViewer indicates that it has been selected for inclusion previously.

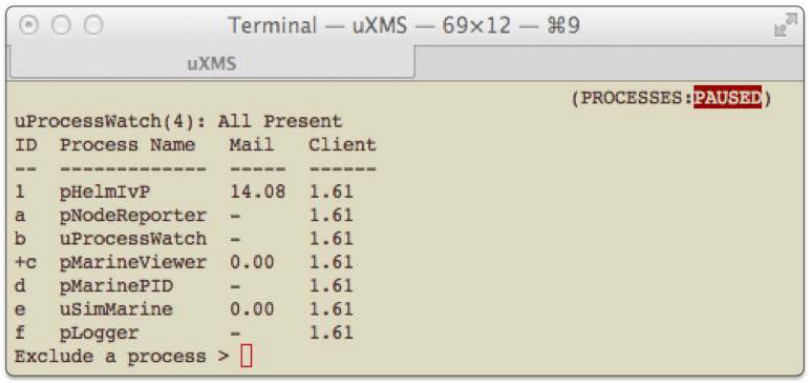

Figure 180: **Excluding watch-list variables based on process origin:** The process list includes an indicator to the left of the ID showing whether the process is presently included ('+') or excluded ('-'). In this case the pMarineViewer application has been included but no action has been taken regarding any other processes.

When a process or application has been selected for *exclusion*, this is handled in the following way. When a scoping report is being generated, if a particular variable has most recently been set by an excluded process, it is not include in the scoping report. Note, it may have also been published by another application not on the exclusion list.

53.4 Configuration File Parameters for uXMS

Configuraton of uXMS may be done from a configuration file (.moos file), from the command line, or both. Generally the parameter settings given on the command line override the settings from the .moos file, but using the configuration file is a convenient way of ensuring certain settings are in effect on repeated command line invocations. The following is short description of the parameters:

Listing 53.81: Configuration Parameters for uXMS.

colormap:	Associates a color for the line of text reporting the given variable.
content_mode:	Set content mode to either `scoping`, `history`, `procs`, or `help`.
display_all:	If `true`, all variables are reported in the *scoping* content mode.
display_aux_source:	If `true`, non-null auxilliary source is shown in place of source.
display_community:	If `true`, the Community column is rendered.
display_history_var:	If `false`, history var not shown in history mode.
display_source:	If `true`, the Source column is rendered.
display_time:	If `true`, the Time column is rendered.
display_virgins:	If `false`, variables never written to the MOOSDB are not reported.
history_var:	Names the MOOS variable reported in the *history* mode.
refresh_mode:	Determines when new reports are written to the screen.
source:	Names a MOOS app for which all variables will be scoped.
term_report_interval:	Time (secs) between report updates (default 0.6).
trunc_data:	If `true`, variable string values are truncated.
wrap_data:	If `true`, variable string values are multi-line wrapped.
wrap_data_len:	Content line length if wrapping string value output. Default is 45.
var:	A comma-separated list of variables to scope on in the *scoping* mode.

53.4.1 An Example uXMS Configuration Block

An example configuration is given in Listing 82. This may also be elicited from the command line:

```
$ uXMS --example or -e
```

Listing 53.82: An example uXMS configuration block.

```
ProcessConfig = uXMS
{
  AppTick   = 4
  CommsTick = 4

  var    = NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING
  var    = PROC_WATCH_SUMMARY
  var    = PROC_WATCH_EVENT
  source = pHelmIvP, pMarineViewer

  history_var           = DB_CLIENTS

  display_virgins       = true    // default
  display_source        = false   // default
  display_aux_source    = false   // default
  display_time          = false   // default
  display_community     = false   // default
  display_all           = false   // default
```

```
19    trunc_data            = 40      // default is no trucation.
20
21    wrap_data             = false   // default
22    wrap_data_len         = 45      // default
23
24    term_report_interval  = 0.6     // default (seconds)
25
26    color_map    = pHelmIvP, red   // All postings by pHelmIvP red
27    color_map    = NAV_SPEED, blue // Only var NAV_SPEED is blue
28
29    refresh_mode = events     // default (or streaming/paused)
30    content_mode = scoping    // default (or history,procs)
31  }
```

53.4.2 The colormap Configuration Parameter

Most of the the configurable options deal with content and layout of the information in the terminal window, but color can also be used to faciliate monitoring one or more variables. The parameter

```
colormap = <variable/app>, <color>
```

is used to request that a line the report containing the given variable or produced by the given MOOS application (source) is rendered in the given color. The choices for color are limited to red, green, blue, cyan, and magenta.

53.4.3 The content_mode Configuration Parameter

The content mode determines what information is generated in each report to the terminal output (Section 53.3). This mode is set with the following parameter:

```
content_mode = <mode-type>   // Default is "scoping"
```

The default setting is `"scoping"` to select the scoping content mode described in Section 53.3.1. It may also be set to `"history"` to select the history mode described in Section 53.3.2, or set to `"procs"` to select the processes mode described in Section 53.3.3.

53.4.4 The display* Configuration Parameters

In the scoping and history content modes, the uXMS report has columns of data that may be optionally turned off to conserve real estate, the *Time*, *Source* and *Community* columns as shown in Figures 173, 176, and 179. By default they are turned off, and they may be toggled on and off by the user at run time. Their initial state may also be configured with the following three parameters:

```
display_community  = <Boolean> // Default is false
display_source     = <Boolean> // Default is false
display_time       = <Boolean> // Default is false
display_aux_source = <Boolean> // Default is false
```

The display_aux_source parameter, when true, not only activates this column, but also indicates that the auxilliary source is to be shown instead of the source. Not all MOOS variable postings have the auxilliary source field filled in. In the case of variables posted by the helm, however, this field contains the both the helm iteration and name of the behavior. If the auxilliary source is empty for a particular variable, the primary source is shown instead. To be clear what is being shown, the auxilliary source is always contained in brackets. For example, `[241:waypoint_return]`, may indicate the variable was posted by the helm on iteration 241 by the waypoint return behavior.

The display_all parameter determines whether the scope list contains only those variables specified by the user, or all MOOS variables pubished by any MOOS process. The latter is useful at times when you can't quite remember the variable name you're looking for or who publishes it. When displaying all variables, certain variables may be masked out by selecting a process (MOOS app) to exclude, as described in Section 53.3.3. It may also be enabled with the `'A'` key, and disabled with the `'a'` key at the terminal at run time. It may also be enabled from the command line with the `--all` or `-a` switches.

```
display_all = <Boolean> // Default is false
```

Using the dipplay_virgins parameter, the report content may be further modified to mask out lines containing variables that have never been written to, and variables with an empty-string value. This is done with the below configuration line. It may also be toggled with the `'v'` key at the terminal at run time, and it may also be specified from the command line with the `--novirgins` or `-g` switches.

```
display_virgins = <Boolean> // Default is true
```

In the history mode, the name of the variable is the same on each line. This makes it clear what variable is being shown, but takes up screen real estate and is redundant. It may be suppressed by setting display_history_var to false in the mission file. It may also be toggled with the `'j'` key at the terminal at run time.

```
display_history_var = <Boolean> // Default is true
```

53.4.5 The history_var Configuration Parameter

The variable reported in the *history* mode is set with the below configuration line:

```
history_var = <MOOSVar>
```

The history report only allows for one variable, and multiple instances of the above line will simply honor the last line provided. The history variable is also automatically added to the watch list used in the scoping mode. The history variable may also be set on the command line when uXMS is launched from the terminal, with `--history=<MOOS-variable>`. If set in both places, the command line choice overrides the choice in the mission file.

53.4.6 The refresh_mode Configuration Parameter

The *refresh* mode determines when new reports are generated to the screen, as discussed in Section 53.2. It is set with the below configuration line:

```
refresh_mode = <mode>  // Valid modes are "paused", "streaming", "events"
```

The initial refresh mode is set to `"events"` by default. The refresh mode set in the configuration file may be overridden from the command line with `--mode=paused|events|streaming`, or chosen interactively at run time with the `'e'` key for *events*, the spacebar key for *paused*, or the `'r'` key for *streaming*.

53.4.7 The source Configuration Parameter

The variable scope list may be set or augmented by naming a particular MOOS app source with the below parameter:

```
source = <MOOSApp>, <MOOSApp>, ...
```

With this, uXMS will subscribe for any MOOS variable published by the named application(s). Since variables may be published by multiple applications, don't be surprised to see postings made by other applications. This is *not* a request to receive mail only from the named source(s). Sources may also be chosen from the command line with the `--src=<MOOSApp>,<MOOSApp>,..K` command line switch. Sources may also be included or excluded dynamically in the `processes` content mode as described in Section 53.3.3, with the '+' and '-' keys.

53.4.8 The term_report_interval Configuration Parameter

The term_report_interval is a parameter defined for all AppCasting MOOS applications. It specifies the amount of time between success updates to the terminal. Report updates are not based on the application's apptick since this may be considerably faster than the human may absorb and wastes CPU resources. The default refresh rate is 0.6 seconds between refreshes. This may be overridden with:

```
term_report_interval = <Non-Zero Value> // Default is 0.6 seconds
```

The interval may also be specified on the command line with the `--termint=<Non-Zero Value>` switch. The accepted range, in seconds, is $[0, 10]$. Keep in mind that report frequency cannot be any faster than the actual apptick set for uXMS.

53.4.9 The trunc_data Configuration Parameter

The current value of a MOOS variable is shown in the `VarValue` column in both the scoping and history content modes. This value may be quite long and overwrap several lines and make things hard to read. The user can choose to truncate the content by setting trunc_data parameter:

```
trunc_data = <unsigned int> // Default is zero (no truncating)
```

Values are accepted in the range $[10, 1000]$. Truncated string output will be further indicated by adding a trailing "..." to the end of the output. Truncation may be toggled on/off at run time by hitting the ''' (back-tick) key. The default truncation length is 40 characters. The length of truncated output may also be adjusted at run time with the '{' and '}' keys.

53.4.10 The wrap_data and wrap_data_len Configuration Parameters

The current value of a MOOS variable is shown in the `VarValue` column in both the scoping and history content modes. This value may be quite long and overwrap several lines and make things hard to read. When view uXMS output in the appcasting window of pMarineViewer, it is really hard to see long output. The user can choose to wrap the content *within the value column of the output* by setting wrap_data parameter:

```
wrap_data = true
wrap_data_len = 30  // default is 45
```

Values for wrap_data_data are accepted in the range $[10, 120]$. The default value is 45. The value of this parameter is set only in the configuration file and cannot be changed at run time. If uXMS is run from the terminal (rather than viewing content in an appcasting window), the wrapping can be toggled on and off with the `'w'` character.

53.4.11 The var Configuration Parameter

The variables reported on in the *scoping* mode, the scope list, are declared with configuration lines of the form:

```
var = <MOOSVar>, <MOOSVar>, ...
```

Multiple such lines, each perhaps with multiple variables, are accommodated. The scope list may be *augmented* on the command line by simply naming variables as command line arguments. The scope list provided on the command line may *replace* the list given in the configuration file if the `--clean` command line option is also invoked.

53.5 Command Line Usage of uXMS

Many of the parameters available for setting the `.moos` file configuration block can also be affected from the command line. The command line configurations always trump any configurations in the `.moos` file. As with the uPokeDB application, the server host and server port information can be specified from the command line too to make it easy to open a uXMS window from anywhere within the directory tree without needing to know where the `.moos` file resides. A uXMS session can be launched to connect to the MOOSDB of a remote vehicle on the network, if the IP address and port number are known, with:

```
$ uXMS --serverhost=10.25.0.191 --serverport=9000 --src=pHelmIvP
```

The basic command line usage for the uXMS application is the following:

```
$ uXMS --help or -h
```

Listing 53.83: Command line usage for the uXMS tool.

```
Usage: uXMS [file.moos] [OPTIONS]
Options:
  --alias=<ProcessName>
      Launch uXMS with the given process name rather than uXMS.
  --all,-a
    Show ALL MOOS variables in the MOOSDB
  --clean,-c
    Ignore scope variables in file.moos
  --colormap=<MOOSVar>,<color>
    Display all entries where the variable, source, or community
    has VAR as substring. Allowable olors: blue, red, magenta,
    cyan, or green.
  --colorany=<MOOSVar>,<MOOSVar>,...
    Display all entries where the variable, community, or source
    has VAR as substring. Color auto-chosen from unused colors.
  --example, -e
      Display example MOOS configuration block.
  --help,-h
      Display this help message.
  --history=<MOOSVar>
    Allow history-scoping on variable
  --interface,-i
      Display MOOS publications and subscriptions.
  --novirgins,-g
    Don't display virgin variables
  --mode=[paused,EVENTS,streaming]
    Determine display mode. Paused: scope updated only on user
    request. Events: data updated only on change to a scoped
    variable. Streaming: updates continuously on each app-tick.
  --serverhost=<IPAddress>
  --mooshost=<IPAddress>
    Connect to MOOSDB at IP=value, not from the .moos file.
  --serverport=<PortNumber>
  --moosport=<PortNumber>
    Connect to MOOSDB at port=value, not from the .moos file.
  --show=[source,time,community,aux]
    Turn on data display in the named column, source, time, or
    community. All off by default enabling aux shows the
    auxilliary source in the souce column.
  --src=<MOOSApp>,<MOOSApp>, ...
    Scope only on vars posted by the given MOOS processes
  --trunc=value [10,1000]
    Truncate the output in the data column.
  --termint=value [0,10]  (default is 0.6)
    Minimum real-time seconds between terminal reports.
  --version,-v
    Display the release version of uXMS.

```

```
49  Shortcuts
50
51    -t  Short for --trunc=25
52    -p  Short for --mode=paused
53    -s  Short for --show=source
55    -st Short for --show=source,time
```

Using the `--clean` switch will cause uXMS to ignore the variables or sources specified in the `.moos` file configuration block and only scope on the variables specified on the command line (otherwise the union of the two sets of variables is used). Typically this is done when a user wants to quickly scope on a couple variables and doesn't want to be distracted with the longer list specified in the `.moos` file. Arguments on the command line other than the ones described above are treated as variable requests.

If the `serverhost` or the `serverport` arguments are not provided on the command line, and a MOOS file is also not provided, the user will be prompted for the two values. Since the most common scenario is when the MOOSDB is running on the local machine ("localhost") with port 9000, these are the default values and the user can simply hit the return key.

```
$ uXMS
$ Enter Server: [localhost] <return>
    The server is set to "localhost"
$ Enter Port: [9000] <return>
$    The port is set to "9000"
```

53.6 Console Interaction with uXMS at Run Time

Many of the launch-time configuration parameters may be altered at run time through interaction with the console window. For example, the displaying of the Source column is configured to be false by default, may be configured in the mission file with `display_source=true`, and may be configured on the command line with `--show=source`. It may also be toggled at run time by typing the `'s'` character at the console window. A list of run-time key mappings may be shown at any time by typing the `'h'` key for help. Re-hitting this key resumes prior scoping. Listing 84 shows the contents of the help menu.

Listing 53.84: The help-menu on the uXMS console.

```
 1  KeyStroke    Function         (HELP)
 2  ---------    --------------------------
 3      s        Toggle show source of variables
 4      t        Toggle show time of variables
 5      c        Toggle show community of variables
 6      v        Toggle show virgin variables
 7      x        Toggle show Auxilliary Src if non-empty
 8      d        Content Mode: Scoping Normal
 9      h        Content Mode: Help. Hit 'R' to resume
10      p        Content Mode: Processes Info
11      z        Content Mode: Variable History
12    > or <     Show More or Less Variable History
13    } or {     Show More or Less Truncated VarValue
14      /        Begin entering a filter string
15      `        Toggle Data Field truncation
```

```
16      ?        Clear current filter
17      a        Revert to variables shown at startup
18      A        Display all variables in the database
19    u/SPC      Refresh Mode: Update then Pause
20      r        Refresh Mode: Streaming
21      e        Refresh Mode: Event-driven refresh
```

53.7 Running uXMS Locally or Remotely

The choice of uXMS as a scoping tool was designed in part to support situations where the target MOOSDB is running on a vehicle with low bandwidth communications, such as an AUV sitting on the surface with only a weak RF link back to the ship. There are two distinct ways one can run uXMS in this situation and its worth noting the difference. One way is to run uXMS locally on one's own machine, and connect remotely to the MOOSDB on the vehicle. The other way is to log onto the vehicle through a terminal, run uXMS remotely, but in effect connecting locally to the MOOSDB also running on the vehicle.

The difference is seen when considering that uXMS is running three separate threads. One accepts mail delivered by the MOOSDB, one executes the iterate loop of uXMS where reports are written to the terminal, and one monitors the keyboard for user input. If running uXMS locally, connected remotely, even though the user may be in paused mode with no keyboard interaction or reports written to the terminal, the first thread still may have a communication requirement perhaps larger than the bandwidth will support. If running remotely, connected locally, the first thread is easily supported since the mail is communicated locally. Bandwidth is consumed in the second two threads, but the user controls this by being in paused mode and requesting new reports judiciously.

53.8 Connecting multiple uXMS processes to a single MOOSDB

Multiple versions of uXMS may be connected to a single MOOSDB. This is to simultaneously allow several people a scope onto a vehicle. Although MOOS disallows two processes of the same name to connect to MOOSDB, uXMS generates a random number between 0-999 and adds it as a suffix to the uXMS name when connected. Thus it may show up as something like `uXMS_871` if you scope on the variable DB_CLIENTS. In the unlikely event of a name collision, the user can just try again.

53.9 Using uXMS with Appcasting

Appcasting allows a really useful way of using uXMS, especially in the case of multiple deployed vehicles. Prior to appcasting, the only way to use uXMS is through a terminal window. With appcasting the uXMS report may also be published to the MOOSDB for remote viewing via a uMAC utility or with pMarineViewer.

For example, consider a case where some number of vehicles are deployed, each with an interface to their batteries, compass and GPS. The interfaces may be named iBatteryMonitor, iCompass, and iGPS. Each interface publishes several MOOS variables including health status messages. A mission could be configurd with three uXMS processes launched at mission startup with something similar to:

Listing 53.85: Launching several uXMS processes with appcasting.

```
1  ProcessConfig = ANTLER
2  {
```

```
3    MSBetweenLaunches = 200
4
5    Run = MOOSDB          @ NewConsole = false
6    // Other MOOS Apps
7    Run = iGPS            @ NewConsole = false
8    Run = iBatteryMonitor @ NewConsole = false
9    Run = iCompas         @ NewConsole = false
10   Run = uXMS            @ NewConsole = false ~ uXMS_GPS
11   Run = uXMS            @ NewConsole = false ~ uXMS_BATTERY_MONITOR
12   Run = uXMS            @ NewConsole = false ~ uXMS_COMPASS
13 }
14
15 ProcessConfig = uXMS_GPS
16 {
17   SOURCE = iGPS
18 }
19 ProcessConfig = uXMS_BATTERY_MONITOR
20 {
21   SOURCE = iBatteryMonitor
22 }
23 ProcessConfig = uXMS_COMPASS
24 {
25   SOURCE = iCompass
26 }
```

With this configuration the three uXMS processes will launch, each *without* a terminal window open, and each with a different descriptive name. The uXMS reports are accessible with any of the appcast viewing tools, uMAC, uMACView, and pMarineViewer. In the latter two tools for example, the uXMS reports may appear in a menu selection like that shown in Figure 181.

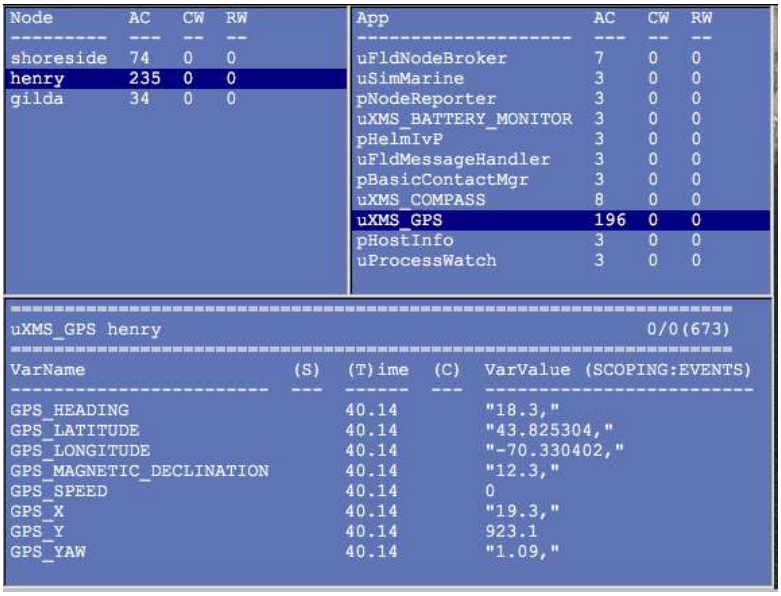

Figure 181: **Appcasting and uXMS:** Multiple vehicles are each configured with three dedicated uXMS processes to scope on variables particular to a given device or sensor. The uMAC viewer interface allows the user to select any vehicle and select a uXMS report to see the desired information for that vehicle and device.

In this way the user may monitor the health of these three instruments across all fielded vehicles with a single GUI without having to write any special code for these devices.

53.10 Publications and Subscriptions for uXMS

The interface for uXMS, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uXMS --interface or -i
```

53.10.1 Variables Published by uXMS

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 53.9.

53.10.2 Variables Subscribed for by uXMS

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 53.9.
- DB_CLIENTS: To handle requests to scope on all variables.
- DB_UPTIME: To determine the MOOSDB start time. All uXMS times reported are times since MOOSDB started.
- PROC_WATCH_SUMMARY: As a convenience this summary is displayed in the *processes* content mode. It is posted by uProcessWatch.
- USER-DEFINED: The variables subscribed for are those on the *scope list*, augmented with the var and source parameters described in Sections 53.4.7 and 53.4.11.

54 uSimMarineV22: Vehicle Simulation

54.1 Overview

The uSimMarineV22 application is a 3D vehicle simulator that updates vehicle state, position and trajectory, based on the present actuator values and prior vehicle state. This app is the successor to the uSimMarine app. Released in 2022, uSimMarineV22 is a backward compatible superset of uSimMarine, with the following notable changes or additions:

- Embeded PID controller: The user has the option to embed a PID controller within the simulator for near-zero latency between PID output and simulator position updates. This allows for substantial increases in maximum time warp. Where previous missions had a maximum time warp of say 50x realtime, 200x to 300x realtime has been observed.
- Worm holes: A worm hole a region in the operation area that will consume an vehicle, and instantly shift its position to another region in the operating area. This facilitates missions where traffic patterns of several contacts are used for testing collision avoidance.
- Sailing: The simulator will apply basic sailing effects on the vehicle given (a) a polar plot describing vehicle speed relative to a given wind direction and magnitude, (b) incoming updates to the current wind direction and magnitude.

The typical usage scenario has a single instance of uSimMarineV22 associated with each simulated vehicle, as shown in Figure 182.

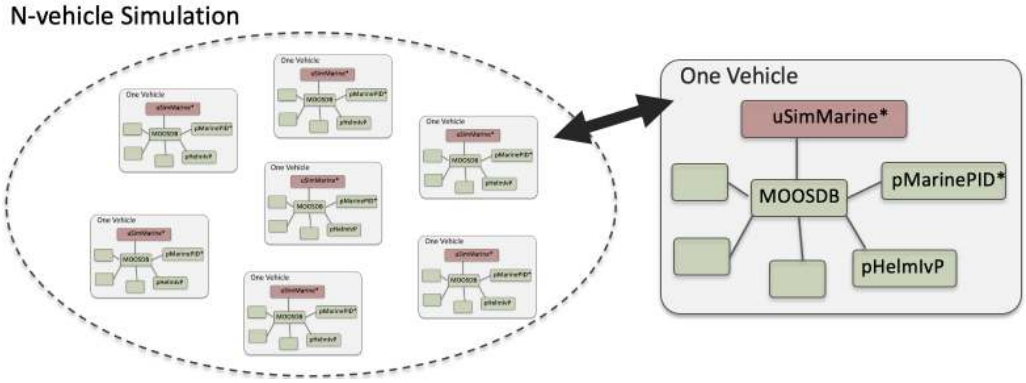

Figure 182: **Typical** uSimMarineV22 **Usage:** In an N-vehicle simulation, an instance if uSimMarineV22 is used for each vehicle. Each simulated vehicle typically has its own dedicated MOOS community. The IvP Helm (pHelmIvP) publishes high-level control decisions. The PID controller (pMarinePID) converts the high-level control decisions to low-level actuator decisions. Finally the simulator (uSimMarineV22) reads the low-level actuator postings to produce a new vehicle position.

This style of simulation can be contrasted with simulators that simulate a comprehensive set of aspects of the simulation, including multiple vehicles, and aspects of the environment and communications. The uSimMarineV22 simulator simply focuses on a single vehicle. It subscribes for the vehicle navigation state variables:

- NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING, and NAV_DEPTH

as well as the actuator values

- DESIRED_RUDDER, DESIRED_THRUST, and DESIRED_ELEVATOR.

The simulator accommodates a notion of external drifts applied to the vehicle to crudely simulate current or wind. These drifts may be set statically or may be changing dynamically by other MOOS processes. The simulator also may be configured with a simple geo-referenced data structure representing a field of water currents.

Under typical UUV payload autonomy operation, the simulator and pMarinePID MOOS modules would not be present. The vehicle's native controller would handle the role of pMarinePID, and the vehicle's native navigation system (and the vehicle itself) would handle the role of the simulator.

54.2 Configuration Parameters for uSimMarineV22

The following parameters are defined for uSimMarineV22. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so in parentheses below.

Listing 54.86: Configuration Parameters for uSimMarineV22.

buoyancy_rate:	Rate, in meters per second, at which vehicle floats to surface at zero speed. The default is zero. Section 54.5.4.
current_field:	A file containing the specification of a current field.
current_field_active:	If true, simulator uses the current field if specified.
default_water_depth:	Default value for local water depth for calculating altitude (0).
drift_vector:	A pair of external drift values, direction and magnitude.
rotate_speed:	An external rotational speed in degrees per second (0).
drift_x:	An external drift value applied in the x direction (0).
drift_y:	An external drift value applied in the y direction (0).
max_acceleration:	Maximum rate of vehicle acceleration in m/s^2 (0.5).
max_deceleration:	Maximum rate of vehicle deceleration in m/s^2 (0.5).
max_depth_rate:	Maximum rate of vehicle depth change, meters per second. The defauls is 0.5. Section 54.5.4.
max_depth_rate_speed:	Vehicle speed at which max depth rate is achievable (2.5). Section 54.5.4.
prefix:	Prefix of MOOS variables published. The default is USM_.
sim_pause:	If true, the simulation is paused. The default is false.
start_depth:	Initial vehicle depth in meters. The default is zero. Section 54.4.
start_heading:	Initial vehicle heading in degrees. The default is zero. Section 54.4.
start_pos:	A full starting position and trajectory specification. Section 54.4.
start_speed:	Initial vehicle speed in meters per second. The default is zero. Section 54.4.
start_x:	Initial vehicle x position in local coordinates. The default is zero. Section 54.4.

start_y:	Initial vehicle y position in local coordinates. The default is zero. Section 54.4.
thrust_factor:	A scalar correlation between thrust and speed. The default is 20.
thrust_map:	A mapping between thrust and speed values. Section 54.8.
thrust_reflect:	If true, negative thrust is simply opposite positive thrust. The default is false. Section 54.8.
turn_loss:	A range $[0, 1]$ affecting speed lost during a turn. The default is 0.85.
turn_rate:	A range $[0, 100]$ affecting vehicle turn radius, e.g., 0 is an infinite turn radius. The default is 70.

54.2.1 An Example MOOS Configuration Block

An example MOOS configuration block is provided in Listing 87 below. This can also be obtained from a terminal window with:

```
$ uSimMarineV22 --example or -e
```

Listing 54.87: Example configuration of the uSimMarineV22 application.

```
1  ========================================================================
2  uSimMarineV22 Example MOOS Configuration
3  ========================================================================
4
5  ProcessConfig = uSimMarineV22
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   start_x       = 0
11   start_y       = 0
12   start_heading = 0
13   start_speed   = 0
14   start_depth   = 0
15   start_pos     = x=0, y=0, speed=0, heading=0, depth=0
16
17   drift_x       = 0
18   drift_y       = 0
19   rotate_speed  = 0
20   drift_vector  = 0,0      // heading, magnitude
21
22   buoyancy_rate        = 0.025 // meters/sec
23   max_acceleration     = 0     // meters/sec^2
24   max_deceleration     = 0.5   // meters/sec^2
25   max_depth_rate       = 0.5   // meters/sec
26   max_depth_rate_speed = 2.0   // meters/sec
27
28   sim_pause            = false // or {true}
29   dual_state           = false // or {true}
30   thrust_reflect       = false // or {true}
31   thrust_factor        = 20    // range [0,inf)
```

```
32    turn_rate            = 70    // range [0,100]
33    thrust_map           = 0:0, 20:1, 40:2, 60:3, 80:5, 100:5
34  }
```

54.3 Publications and Subscriptions for uSimMarineV22

The interface for uSimMarineV22, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ usimMarine --interface or -i
```

54.3.1 Variables Published by uSimMarineV22

The primary output of uSimMarineV22 to the MOOSDB is the full specification of the updated vehicle position and trajectory, along with a few other pieces of information:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- BUOYANCY_REPORT:
- TRIM_REPORT:
- USM_ALTITUDE: The updated vehicle altitude in meters if water depth known.
- USM_DEPTH: The updated vehicle depth in meters. Section 54.5.4.
- USM_DRIFT_SUMMARY: A summary of the current total external drift.
- USM_HEADING: The updated vehicle heading in degrees.
- USM_HEADING_OVER_GROUND: The updated vehicle heading over ground.
- USM_LAT: The updated vehicle latitude position.
- USM_LONG: The updated vehicle longitude position.
- USM_RESET_COUNT: The number of time the simulator has been reset.
- USM_SPEED: The updated vehicle speed in meters per second.
- USM_SPEED_OVER_GROUND: The updated speed over ground.
- USM_X: The updated vehicle x position in local coordinates.
- USM_Y: The updated vehicle y position in local coordinates.
- USM_YAW: The updated vehicle yaw in radians.

An example USM_DRIFT_SUMMARY string: `"ang=90, mag=1.5, xmag=90, ymag=0"`.

54.3.2 Variables Subscribed for by uSimMarineV22

The uSimMarineV22 application will subscribe for the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration. See the documentation on Appcasting.
- DESIRED_THRUST: The thruster actuator setting, $[-100, 100]$.
- DESIRED_RUDDER: The rudder actuator setting, $[-100, 100]$.
- DESIRED_ELEVATOR: The depth elevator setting, $[-100, 100]$.

- USM_SIM_PAUSED: Simulation pause request, either `true` or `false`.
- USM_CURRENT_FIELD: If `true`, a configured current field is active.
- USM_BUOYANCY_RATE: Dynamically set the zero-speed float rate.
- ROTATE_SPEED: Dynamically set the external rotational speed.
- DRIFT_X: Dynamically set the external drift in the x direction.
- DRIFT_Y: Dynamically set the external drift in the y direction.
- DRIFT_VECTOR: Dynamically set the external drift direction and magnitude.
- DRIFT_VECTOR_ADD: Dynamically modify the external drift vector.
- DRIFT_VECTOR_MULT: Dynamically modify the external drift vector magnitude.
- USM_RESET: Reset the simulator with a new position, heading, speed and depth.
- WATER_DEPTH: Water depth at the present vehicle position.

Each iteration, after noting the changes in the navigation and actuator values, it posts a new set of navigation state variables in the form of USM_X, USM_Y, USM_SPEED, USM_HEADING, USM_DEPTH.

54.3.3 Command Line Usage of uSimMarineV22

The uSimMarineV22 application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. The basic command line usage for the uSimMarineV22 application is the following:

Listing 54.88: Command line usage for the uSimMarineV22 application.

```
Usage: uSimMarineV22 file.moos [OPTIONS]

Options:
  --alias=<ProcessName>
      Launch uSimMarineV22 with the given process name
      rather than uSimMarineV22.
  --example, -e
      Display example MOOS configuration block.
  --help, -h
      Display this help message.
  --version,-v
      Display the release version of uSimMarineV22.
```

54.4 Setting the Initial Vehicle Position, Pose and Trajectory

The simulator is typically configured with a vehicle starting position, pose and trajectory given by the following five configuration parameters:

- start_x
- start_y
- start_heading
- start_speed
- start_depth

The position is specified in local coordinates in relation to a local datum, or $(0, 0)$ position. This datum is specified in the `.moos` file at the global level. The heading is specified in degrees and

corresponds to the direction the vehicle is pointing. The initial speed and depth by default are zero, and are often left unspecified in configuration. Alternatively, the same five parameters may be set with the start_pos parameter as follows:

```
start_pos = x=100, y=150, speed=0, heading=45, depth=0
```

The simulator can also be reset at any point during its operation, by posting to the MOOS variable USM_RESET. A posting of the following form will reset the same five parameters as above:

```
USM_RESET = x=200, y=250, speed=0.4, heading=135, depth=10
```

This has been useful in cases where the objective is to observe the behavior of a vehicle from several different starting positions, and an external MOOS script, e.g., uTimerScript, is used to reset the simulator from each of the desired starting states.

54.5 Propagating the Vehicle Speed, Heading, Position and Depth

The vehicle position is updated on each iteration of the uSimMarineV22 application, based on (a) the previous vehicle state, (b) the elapsed time since the last update, ΔT, (c) the current actuator values, DESIRED_RUDDER, DESIRED_THRUST, and DESIRED_ELEVATOR, and (d) several parameter settings describing the vehicle model.

For simplicity, this simulator updates the vehicle speed, heading, position and depth in sequence, in this order. For example, the position is updated after the heading is updated, and the new position update is made as if the new heading were the vehicle heading for the entire ΔT. The error introduced by this simplification is mitigated by running uSimMarineV22 with a fairly high MOOS AppTick value keeping the value of ΔT sufficiently small.

54.5.1 Propagating the Vehicle Speed

The vehicle speed is propagated primarily based on the current value of thrust, which is presumably refreshed upon each iteration by reading the incoming mail on the MOOS variable DESIRED_THRUST. To simulate a small speed penalty when the vehicle is conducting a turn through the water, the new thrust value may also be affected by the current rudder value, referenced by the incoming MOOS variable DESIRED_RUDDER. The newly calculated speed is also dependent on the previously noted speed noted by the incoming MOOS variable NAV_SPEED, and the settings to the two configuration parameters MAX_ACCELERATION and MAX_DECELERATION.

The algorithm for updating the vehicle speed proceeds as:

1. Calculate $v_{i(\texttt{RAW})}$, the new raw speed based on the thrust.
2. Calculate $v_{i(\texttt{TURN})}$, an adjusted and potentially lower speed, based on the raw speed, $v_{i(\texttt{RAW})}$, and the current rudder angle, DESIRED_RUDDER.
3. Calculate $v_{i(\texttt{FINAL})}$, an adjusted and potentially lower speed based on $v_{i(\texttt{TURN})}$, compared to the prior speed. If the magnitude of change violates either the max acceleration or max deceleration settings, then the new speed is clipped appropriately.
4. Set the new speed to be $v_{i(\texttt{FINAL})}$, and use this new speed in the later updates on heading, position and depth.

Step 1: In the first step, the new speed is calculated by the current value of thrust. In this case the *thrust map* is consulted, which is a mapping from possible thrust values to speed values. The thrust map is configured with the THRUST_MAP configuration parameter, and is described in detail in Section 54.8.

$$v_{i_{(\texttt{RAW})}} = \texttt{THRUST\_MAP}(\texttt{DESIRED\_THRUST})$$

Step 2: In the second step, the calculated speed is potentially reduced depending on the degree to which the vehicle is turning, as indicated by the current value of the MOOS variable DESIRED_RUDDER. If it is not turning, it is not diminished at all. The adjusted speed value is set according to:

$$v_{i_{(\texttt{TURN})}} = v_{i_{(\texttt{RAW})}} * (1 - (\frac{|\texttt{RUDDER}|}{100} * \texttt{TURN\_LOSS}))$$

The configuration parameter turn_loss is a value in the range of $[0, 1]$. When set to zero, there is no speed lost in any turn. When set to 1, there is a 100% speed loss when there is a maximum rudder. The default value is 0.85.

Step 3: In the last step, the candidate new speed, $v_{i_{(\texttt{TURN})}}$, is compared with the incoming vehicle speed, v_{i-1}. The elapsed time since the previous simulator iteration, ΔT, is used to calculate the acceleration or deceleration implied by the new speed. If the change in speed violates either the min_acceleration, or max_acceleration parameters, the speed is adjusted as follows:

$$v_{i_{(\texttt{FINAL})}} = \begin{cases} v_{i-1} + (\texttt{MAX\_ACCELERATION} * \Delta T) & \frac{(v_{i_{(\texttt{TURN})}} - v_{i-1})}{\Delta T} > \texttt{MAX\_ACCELERATION}, \\ v_{i-1} - (\texttt{MAX\_DECELERATION} * \Delta T) & \frac{(v_{i-1} - v_{i_{(\texttt{TURN})}})}{\Delta T} > \texttt{MAX\_DECELERATION}, \\ v_{i_{(\texttt{TURN})}} & \text{otherwise.} \end{cases}$$

Step 4: The final speed from the previous step is posted by the simulator as USM_SPEED, and is used the calculations of position and depth, described next.

54.5.2 Propagating the Vehicle Heading

The vehicle heading is propagated primarily based on the current RUDDER value which is refreshed upon each iteration by reading the incoming mail on the MOOS variable DESIRED_RUDDER, and the elapsed time since the simulator previously updated the vehicle state, ΔT. The change in heading my also be influenced by the THRUST value from the MOOS variable DESIRED_THRUST, and may also factor an external rotational speed.

The algorithm for updating the new vehicle heading proceeds as:

1. Calculate $\Delta\theta_{i(\texttt{RAW})}$, the new raw change in heading influenced only by the current rudder value.
2. Calculate $\Delta\theta_{i(\texttt{THRUST})}$, an adjusted change in heading, based on the raw change in heading, $\Delta\theta_{i(\texttt{RAW})}$, and the current THRUST value.
3. Calculate $\Delta\theta_{i(\texttt{EXTERNAL})}$, an adjusted change in heading considering external rotational speed.
4. Calculate θ_i, the final new heading based on the calculated change in heading and the previous heading, and converted to the range of $[0, 359]$.

Step 1: In the first step, the new heading is calculated by the current RUDDER value:

$$\Delta\theta_{i(\texttt{RAW})} = \texttt{RUDDER} * \frac{\texttt{TURN\_RATE}}{100} * \Delta T$$

The TURN_RATE is an uSimMarineV22 configuration parameter with the allowable range of $[0, 100]$. The default value of this parameter is 70, chosen in part to be consistent with the performance of the simulator prior to this parameter being exposed to configuration. A value of 0 would result in the vehicle never turning, regardless of the rudder value.

Step 2: In the second step the influence of the current vehicle thrust (from the MOOS variable DESIRED_THRUST) may be applied to the change in heading. The magnitude of the change of heading is adjusted to be greater when the thrust is greater than 50% and less when the thrust is less than 50%.

$$\Delta\theta_{i(\texttt{THRUST})} = \theta_{i(\texttt{RAW})} * (1 + \frac{|\texttt{THRUST}| - 50}{50})$$

The direction in heading change is then potentially altered based on the sign of the THRUST:

$$\Delta\theta_{i(\texttt{THRUST})} = \begin{cases} -\Delta\theta_{i(\texttt{THRUST})} & \texttt{THRUST} < 0, \\ \Delta\theta_{i(\texttt{THRUST})} & \text{otherwise.} \end{cases}$$

Step 3: In the third step, the change in heading may be further influenced by an external rotational speed. This speed, if present, would be read at the outset of the simulator iteration from either the configuration parameter rotate_speed, or dynamically from the MOOS variable ROTATE_SPEED. The updated value is calculated as follows:

$$\Delta\theta_{i(\texttt{EXTERNAL})} = \theta_{i(\texttt{THRUST})} + (\texttt{ROTATE\_SPEED} * \Delta T)$$

Step 4: In final step, the final new heading is set based on the previous heading and the change in heading calculated in the previous three steps. If needed, the value of the new heading is converted to its equivalent heading in the range $[0, 359]$.

$$\theta_i = \texttt{heading360}(\theta_{i-1} + \Delta\theta_{i(\texttt{EXTERNAL})})$$

The simulator then posts this value to the MOOSDB as USM_HEADING.

54.5.3 Propagating the Vehicle Position

The vehicle position is propagated primarily based on the newly calculated vehicle heading and speed, the previous vehicle position, and the elapsed time since updating the previous vehicle position, ΔT.

The algorithm for updating the new vehicle position proceeds as:

1. Calculate the vehicle heading and speed used for updating the new vehicle position, with the heading converted into radians.
2. Calculate the new positions, x_i and y_i, based on the heading, speed and elapsed time.
3. Calculate a possibly revised new position, factoring in any external drift.

Step 1: In the first step, the heading value, $\bar{\theta}$, and speed value, $\bar{v}$ used for calculating the new vehicle position is set averaging the newly calculated values with their prior values:

$$\bar{v} = \frac{(v_i + v_{i-1})}{2} \tag{8}$$

$$\bar{\theta} = \text{atan2}(s, c)$$

where s and c are given by:

$$s = sin(\theta_{i-1}\pi/180) + sin(\theta_i\pi/180)$$
$$c = cos(\theta_{i-1}\pi/180) + cos(\theta_i\pi/180)$$

The above calculation of the heading average handles the issue of angle wrap, i.e., the average of 359 and 1 is zero, not 180.

Step 2: The vehicle x and y position is updated by the following two equations:

$$x_i = x_{i-1} + sin(\bar{\theta}) * \bar{v} * \Delta\mathtt{T}$$
$$y_i = y_{i-1} + cos(\bar{\theta}) * \bar{v} * \Delta\mathtt{T}$$

The above is calculated keeping in mind the difference in convention used in marine navigation where zero degrees is due North and 90 degrees is due East. That is, the mapping is as follows from marine to traditional trigonometric convention: $0° \rightarrow 90°$, $90° \rightarrow 0°$, $180° \rightarrow 270°$, $270° \rightarrow 180°$.

Step 3: The final step adjusts the x, and y position from above, taking into consideration any external drift that may be present. This drift includes both the drift that may be directed from the incoming MOOS variables as described in Section 54.7. The drift components below are also a misnomer since they are provided in units of meters per second.

$$x_i = x_i + \texttt{EXTERNAL\_DRIFT\_X} * \Delta\mathtt{T} \tag{9}$$
$$y_i = y_i + \texttt{EXTERNAL\_DRIFT\_Y} * \Delta\mathtt{T} \tag{10}$$

54.5.4 Propagating the Vehicle Depth

Depth change in uSimMarineV22 is simulated based on a few input parameters. The primary parameter that changes from one iteration to the next is the ELEVATOR actuator value, from the MOOS variable DESIRED_ELEVATOR. On any given iteration the new vehicle depth, z_i, is determined by:

$$z_i = z_{i-1} + (\dot{z}_i * \Delta t)$$

The new vehicle depth is altered by the *depth change rate*, $\dot{z}_i$, applied to the elapsed time, Δt, which is roughly equivalent to the apptick interval set in the uSimMarineV22 configuration block. The depth change rate on the current iteration is determined by the vehicle speed and the ELEVATOR actuator value, and by the following three vehicle-specific simulator configuration parameters that allow for some variation in simulating the physical properties of the vehicle. The buoyancy_rate, for simplicity, is given in meters per second where positive values represent a positively buoyant vehicle. The max_depth_rate, and max_depth_rate_speed parameters determine the function(s) shown in Figure 183. The vehicle will have a higher depth change rate at higher speeds, up to some maximum speed where the speed no longer affects the depth change rate. The actual depth change rate then depends on the elevator and vehicle speed.

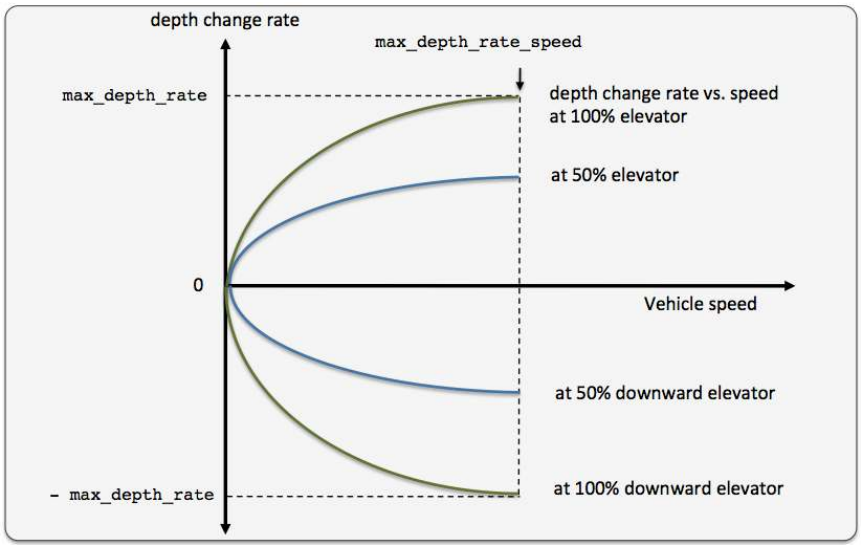

Figure 183: The relationship between the rate of depth change rate, given a current vehicle speed. Different elevator settings determine unique curves as shown.

The value of the depth change rate, $\dot{v}_i$, is determined as follows:

$$\dot{z}_i = \left(\frac{\bar{v}}{\texttt{MAX\_DEPTH\_RATE\_SPEED}}\right)^2 * \frac{\texttt{ELEVATOR}}{100} * \texttt{MAX\_DEPTH\_RATE} + \texttt{BUOYANCY\_RATE} \tag{11}$$

Both fraction components in the above equation are clipped to $[-1, 1]$. When the vehicle is in reverse thrust and has a negative speed, this equation still holds. However, a vehicle would likely not have a depth change rate curve symmetric between positive and negative vehicle speeds. By default the value of buoyancy_rate is set to 0.025, slightly positively buoyant, max_depth_rate is set to 0.5, and max_depth_rate_speed is set to 2.0. The prevailing buoyancy rate may be dynamically adjusted by a separate MOOS application publishing to the variable BUOYANCY_RATE.

54.6 Propagating the Vehicle Altitude

The vehicle altitude is base solely on the current vehicle depth and the depth of the water at the current vehicle position. If nothing is known about the water depth, then USM_ALTITUDE is not published. The simulator may be configured with a default water depth:

```
default_water_depth  = 100
```

This will allow the simulator to produce some altitude information if needed for testing consumers of USM_ALTITUDE information. Furthermore, the simulator subscribes for water depth information in the variable USM_WATER_DEPTH which could conceivably be produced by another MOOS application with access to bathymetry data and the vehicle's navigation position.

54.7 Simulation of External Drift

When the simulator updates the vehicle position as in equations provided in Section 54.5.3, it factors a possible external drift in the x and y directions, in the term EXTERNAL_DRIFT_X, and EXTERNAL_DRIFT_X respectively. The external drift may have two distinct components; a drift applied generally, and a drift applied due to a current field configured with an external file correlating drift vectors to local x and y positions. These drifts may be set in one of three ways discussed next.

54.7.1 External X-Y Drift from Initial Simulator Configuration

An external drift may be configured upon startup by either specifying explicitly the drift in the x and y direction, or by specifying a drift magnitude and direction. Figure 184 shows two external drifts each with the appropriate configuration using either the drift_x and drift_y parameters or the single drift_vector parameter:

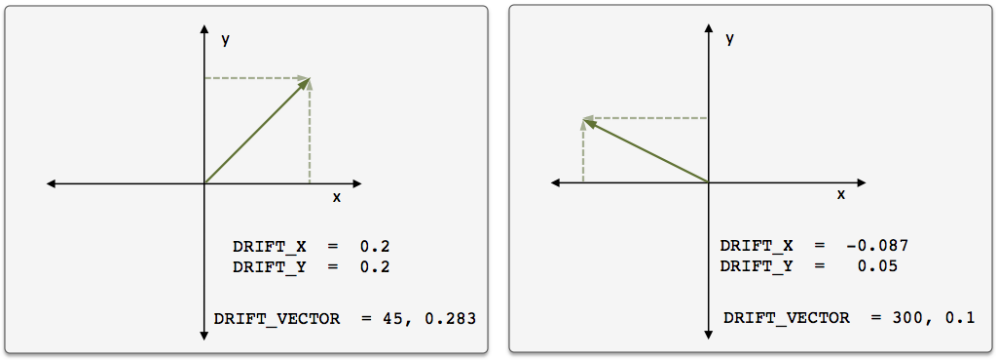

Figure 184: **External Drift Vectors**: Two drift vectors each configured with either the drift_x and drift_y configuration parameters or their equivalent single drift_vector parameter.

If, for some reason, the user mistakenly configures the simulator with both configuration styles, the configuration appearing last in the configuration block will be the prevailing configuration. If uSimMarineV22 is configured with these parameters, these external drifts will be applied on the very first iteration and all later iterations unless changed dynamically, as discussed next.

54.7.2 External X-Y Drift Received from Other MOOS Applications

External drifts may be adjusted dynamically by other MOOS applications based on any criteria wished by the user and developer. The uSimMarineV22 application registers for the following MOOS variables in this regard: DRIFT_X, DRIFT_Y, DRIFT_VECTOR, DRIFT_VECTOR_ADD, DRIFT_VECTOR_MULT. The first three variables simply override the previously prevailing drift, set by either the initial configuration or the last received mail concerning the drift.

By posting to the USM_DRIFT_VECTOR_MULT variable the *magnitude* of the prevailing vector may be modified with a single multiplier such as:

```
DRIFT_VECTOR_MULT = 2
DRIFT_VECTOR_MULT = -1
```

The first MOOS posting above would double the size of the prevailing drift vector, and the second example would reverse the direction of the vector. The DRIFT_VECTOR_ADD variable describes a drift vector to be *added* to the prevailing drift vector. For example, consider the prevailing drift vector shown on the left in Figure 184, with the following MOOS mail received by the simulator:

```
DRIFT_VECTOR_ADD = "262.47, 15.796"
```

The resulting drift vector would be the vector shown on the right in Figure 184. This interface opens the door for the scripting changes to the drift vector like the one below, that crudely simulate a gust of wind in a given direction that builds up to a certain magnitude and dies back down to a net zero drift.

```
DRIFT_VECTOR_ADD = 137, 0.25
DRIFT_VECTOR_ADD = 137, 0.25
DRIFT_VECTOR_ADD = 137, 0.25
DRIFT_VECTOR_ADD = 137, 0.25
DRIFT_VECTOR_ADD = 137, 0.25
DRIFT_VECTOR_ADD = 137, -0.25
DRIFT_VECTOR_ADD = 137, -0.25
DRIFT_VECTOR_ADD = 137, -0.25
DRIFT_VECTOR_ADD = 137, -0.25
DRIFT_VECTOR_ADD = 137, -0.25
```

The above style script was described in the documentation for uTimerScript, where the uTimerScript utility was used to simulate wind gusts in random directions with random magnitude. The DRIFT_* interface may also be used by any third party MOOS application simulating things such as ocean or wind currents.

54.8 The ThrustMap Data Structure

A *thrust map* is a data structure that may be used to simulate a non-linear relationship between thrust and speed. This is configured in the uSimMarineV22 configuration block with the thrust_map parameter containing a comma-separated list of colon-separated pairs. Each element in the comma-separated list is a single mapping component. In each component, the value to the left of the colon is a thrust value, and the other value is a corresponding speed. The following is an example mapping given in string form, and rendered in Figure 185.

```
thrust_map = "-100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:5"
```

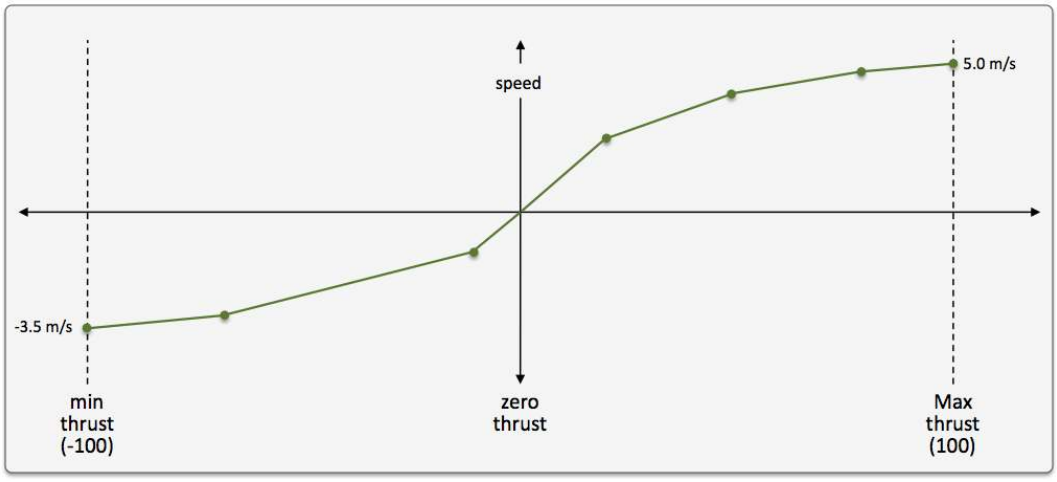

Figure 185: **A Thrust Map**: The example thrust map was defined by seven mapping points in the string "-100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:5".

54.8.1 Automatic Pruning of Invalid Configuration Pairs

The thrust map has an immutable domain of $[-100, 100]$, indicating 100% forward and reverse thrust. Mapping pairs given outside this domain will simply be ignored. The thrust mapping must also be monotonically increasing. This follows the intuition that more positive thrust will not result in the vehicle going slower, and likewise for negative thrust. Since the map is configured with a sequence of pairs as above, a pair that would result in a non-monotonic map is discarded. All maps are created as if they had the pair `0:0` given explicitly. Any pair provided in configuration with zero as the thrust value will ignored; zero thrust always means zero speed. Therefore, the following map configurations would all be equivalent to the map configuration above and shown in Figure 185:

```
thrust_map = -120:-5, -100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:5.0, 120:6
thrust_map = -100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 90:4, 100:5.0
thrust_map = -100:-3.5, -75:-3.2, -10:-2, 0:0, 20:2.4, 50:4.2, 80:4.8, 100:5.0
thrust_map = -100:-3.5, -75:-3.2, -10:-2, 0:1, 20:2.4, 50:4.2, 80:4.8, 100:5.0
```

In the first case, the pairs `"-120:-5"` and `"120:6"` would be ignored since they are outside the $[-100, 100]$ domain. In the second case, the pair `"90:4"` would be ignored since its inclusion would entail a non-monotonic mapping given the previous pair of `"80:4.8"`. In the third case, the pair `"0:0"` would be effectively ignored since it is implied in all map configurations anyway. In the fourth case, the pair `"0:1"` would be ignored since a mapping from a non-zero speed to zero thrust is not permitted.

54.8.2 Automatic Inclusion of Implied Configuration Pairs

Since the domain $[-100, 100]$ is immutable, the thrust map is altered a bit automatically when or if the user provides a configuration without explicit mappings for the thrust values of -100 or 100. In this case, the missing mapping becomes an implied mapping. The mapping `100:v` is added where `v` is the speed value of the closest point. For example, the following two configurations are equivalent:

```
thrust_map = -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8
thrust_map = -100:-3.2, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:4.8
```

54.8.3 A Shortcut for Specifying the Negative Thrust Mapping

For convenience, the mapping of positive thrust values to speed values can be used in reverse for negative thrust values. This is done by configuring `uSimMarineV22` with `thrust_reflect=true`, which is false by default. If `thrust_reflect` is false, then a speed of zero is mapped to all negative thrust values. If `thrust_reflect` is true, but the user nevertheless provides a mapping for a negative thrust in a thrust map, then the `thrust_reflect` directive is simply ignored and the thrust map is used instead. For example, the following two configurations are equivalent:

```
thrust_map = -100:-5, -80:-4.8, -50:-4.2, -20:-2.4, 20:2.4, 50:4.2, 80:4.8, 100:5
```

and

```
thrust_map = 20:2.4, 50:4.2, 80:4.8, 100:5
thrust_reflect = true
```

54.8.4 The Inverse Mapping - From Speed To Thrust

Since a thrust map only permits configurations resulting in a non-monotonic function, the inverse also holds (almost) as a valid mapping from speed to thrust. We say ”almost” because there is ambiguity in cases where there is one or more plateau in the thrust map as in:

```
thrust_map = -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8
```

In this case a speed of 4.8 maps to any thrust in the range $[80, 100]$. To remove such ambiguity, the thrust map, as implemented in a C++ class with methods, returns the lowest magnitude thrust in such cases. A speed of 4.8 (or 5 for that matter), would return a thrust value of 80. A speed of

−3.2 would return a thrust value of −75. The motivation for this way of disambiguation is that if a thrust value of 80 and 100, both result in the same speed, one would always choose the setting that conserves less energy. Reverse mappings are not used by the uSimMarineV22 application, but may be of use in applications responsible for posting a desired thrust given a desired speed, as with the pMarinePID application.

54.8.5 Default Behavior of an Empty or Unspecified ThrustMap

If uSimMarineV22 is configured without an explicit thrust_map or thrust_reflect configuration, the default behavior is governed as if the following two lines were actually included in the uSimMarineV22 configuration block:

```
thrust_map     = 100:5
thrust_reflect = false
```

The default thrust map is rendered in Figure 186.

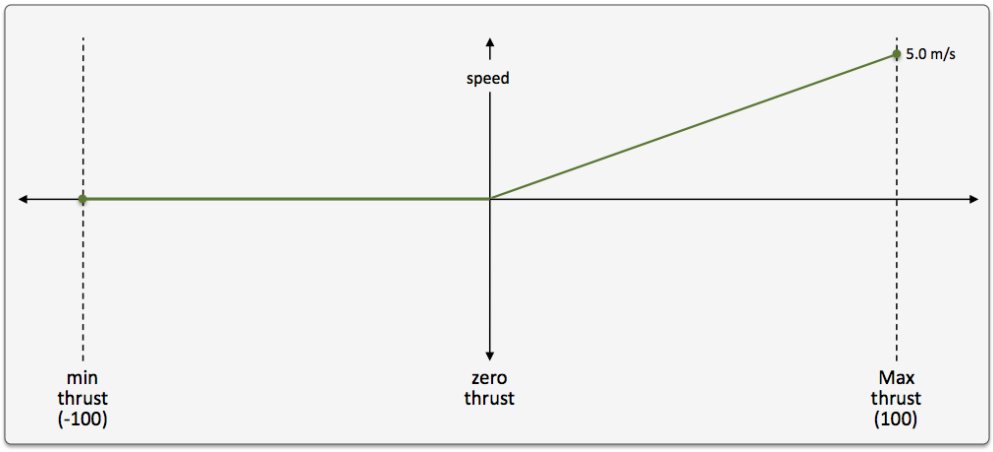

Figure 186: **The Default Thrust Map**: This thrust map is used if no explicit configuration is provided.

This default configuration was chosen for its reasonableness, and to be consistent with the behavior of prior versions of uSimMarineV22 where the user did not have the ability to configure a thrust map.

54.9 Wormholes

A *wormhole* connects two points in space whereby a vehicle entering from one direction immediately emerges at the opposite end. While the existence in nature is just a theory, their existence in uSimMarineV22 is a feature now available as of July 2022.

54.9.1 Wormhole Motivation

A goal of certain simulutions is to test vehicle performance, given some initial starting conditions. If the goal is to test performace over say thousands of variations of the mission, the user has roughly two options. (a) Repeatedly launch the mission with the right conditions, shut down the mission, note the results, and repeat. (b) Within a single mission, repeatedly re-create the starting conditions, let the event of interest unfold again, note the results and repeat.

The advantage of (a) is that the mission is short and "clean". There are no phases of the simulation where the vehicles are re-setting or driving back to their starting positions. The drawback comes with the overhead of repeatedly starting, killing, post-processing a simulation for each test of the event of interest. The advantage of (b) is that many events are observed for a single mission start, kill, post-process cycle. The drawbacks of (b) are that, between events, vehicles need to drive to re-set the conditions for the event of interest. Consider the mission in Figure 187 below. A single vehicle traverse a travel lane, where several contacts are crossing the lane. After the initial traversals, each vehicle drives back to re-set the starting conditions.

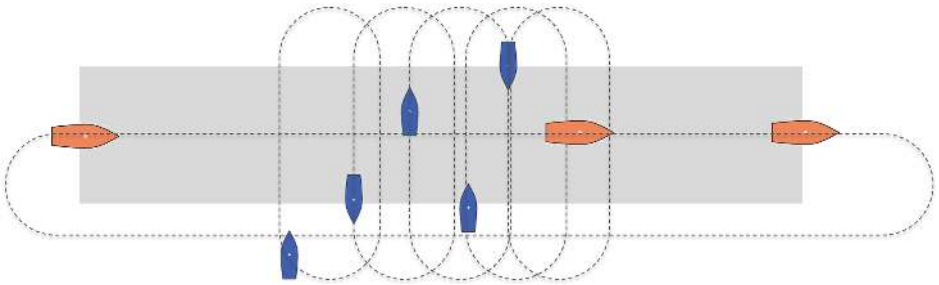

Figure 187: **Simulation with No Wormhole**: A single vehicle, ownship, traverses a travel lane, avoiding collisions with several vehicles which may be crossing the lane. When ownship reaches the end of the lane it traverses back to the beginning of the lane, perhaps encountering contacts outside the intended context for testing. Likewise, the contacts also repeatedly turn around to re-enter the travel lane as foils for testing ownship collision avoidance.

Wormholes can be used in simulations like the one shown above in Figure 187, to create a variation like the one shown in Figure 188. The orange test vehicle, when it reaches the end of the travel lane, will enter the wormhole to instantly re-appear at the same end of the travel lane it came from. The contacts in this example also do not need to turn around (and potentially take evasive action between each other during the turns).

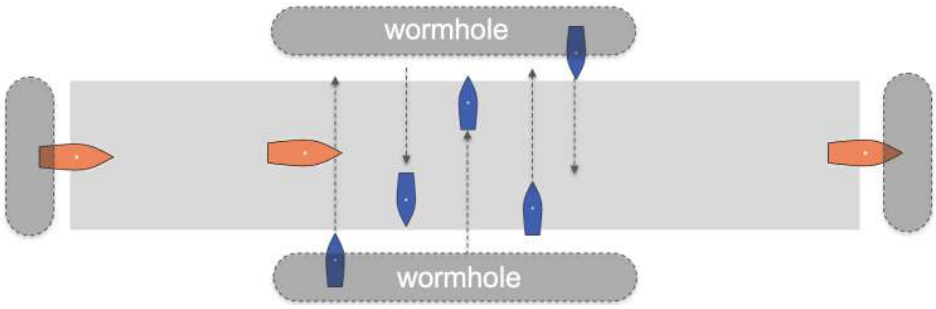

Figure 188: **Simulation with Wormhole**: The single orange vehicle, ownship, traverses a travel lane, entering a wormhole at the end of the lane, and re-emerging at the start of the lane. Contacts crossing the travel lane each enter a wormhole on the destination side, to then immediately re-emerge at the originating side. All vehicles continuously progress in one direction with no need for turning around.

54.9.2 Wormhole Configuration

A wormhole is configured with a pair of polygons, one polygon is called the `madrid_poly`, and the other is the `weber_poly`. The cities of Madrid (Spain) and Weber (New Zealand), are Antipode cities, lying exactly or nearly exactly at opposite coorindates on the globe.

The below three configuration lines declare the entry and exit polygons and the direction in which the wormhole is configure. The `tag` component allows multiple wormholes to be confgured, each with unique tag.

```
wormhole = tag=one, madrid_poly={format=ellipse,x=80,y=-180,degs=0,major=160,minor=20,pts=20}
wormhole = tag=one, weber_poly={format=ellipse,x=80,y=0,degs=0,major=160,minor=20,pts=20}
wormhole = tag=one, connection=from_madrid
wormhole = tag=one, id_change=true
wormhole = tag=one, delay=0
```

The last two components, `id_change=true` and `delay=3`, are related to features not yet implement. Three features are yet to be implemented:

- `connection`: Currently the wormhole only works in one declared direction. Bi-directional wormholes will be implemented to support, for example, the bi-directional traffic shown in Figure 188.
- `id_change`: Currently the vehicle name does not change as it passes through the wormhole. At times it may be useful if the name does change, as a manner of testing the operation of the contact manager which needs to handle memory management over many ephemeral contacts over a long mission.
- `delay`: Currently a vehicle entering a wormhole will instantly appear on the other

55 pHostInfo: Detecting and Sharing Host Info

55.1 Overview

The pHostInfo application is a tool with a simple objective - determine the IP address of the machine on which it is running and post it to the MOOSDB. Although this information is available in a number of ways to a user at the keyboard, it may not be readily available for reasoning about within a MOOS community. Often, from an application's perspective, the host name is simply known and configured as *localhost*. This is fine for most purposes, but in situations where a user is on a machine where the IP address changes frequently, and the user is launching MOOS processes that talk to other machines, it may be very convenient to auto-determine the prevailing IP address and publish it to the MOOSDB. The typical usage scenario for pHostInfo is shown in Figure 189.

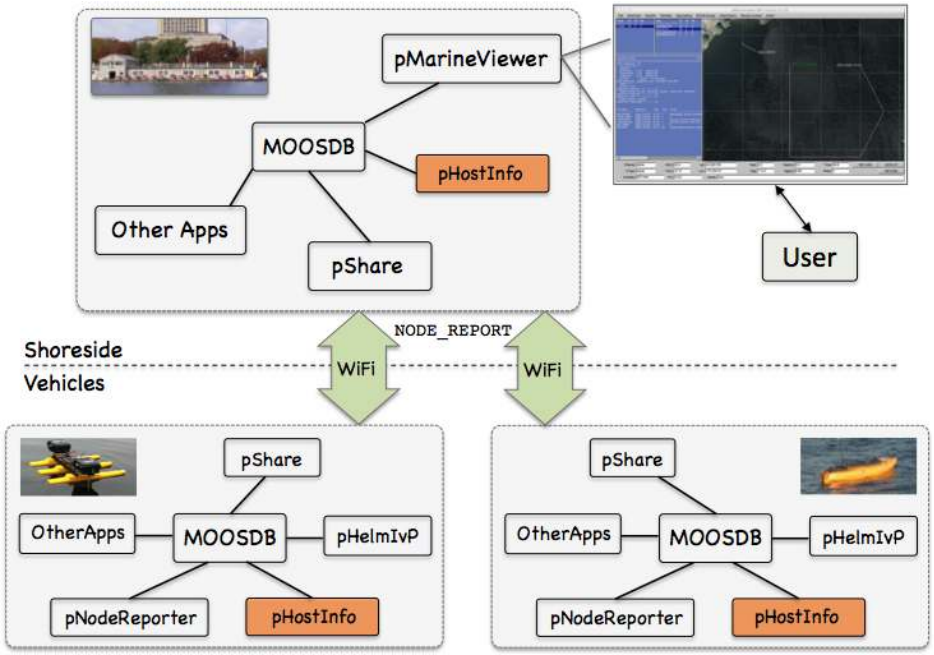

Figure 189: **Typical pHostInfo Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

There are two scenarios where this is currently envisioned to be useful. The first is when the fielded vehicles are on the network with their IP addresses set via DHCP. For example if on the network via a cellular phone connection. The second is if the "vehicle" is a simulated vehicle running on a user's laptop also with its IP address set via DHCP. In both cases there may be another MOOS community with a known IP address, e.g., a shoreside community, to which the local vehicle wishes to inform it of its current IP address. This simple process however does not get involved in any activity regarding the communication to other MOOS communities, but simply tries to determine and post, the IP address, e.g., `PHI_HOST_IP ="192.168.0.1"` for other applications to do as they see fit.

55.2 Configuration Parameters for pHostInfo

The pHostInfo application may be configured with a configuration block within a MOOS mission file, typically with a `.moos` file suffix. The following parameters are defined for pHostInfo.

Listing 55.89: Configuration Parameters for pHostInto.

temp_file_dir:	Directory where temporary files are written. Default is `"~/"`.
default_hostip:	IP address used if no IP address can otherwise be determined.
default_hostip_force:	This IP address will override any IP address from an auto-discovered network interface. Useful for debugging.
prefer_inerface_:	If multiple interfaces have a valid IP address then a preference can be specified. Valid options are `wlan0`, `wifi`, `eth0`, `eth1`, `usb0`, `usb1`,`usb2`.

55.2.1 An Example MOOS Configuration Block

An example MOOS configuration block may be obtained from the command line with the following:

```
$ pHostInfo --example or -e
```

Listing 55.90: Example configuration of the pHostInfo application.

```
1  ============================================================
2  pHostInfo Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pHostInfo
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   temp_file_dir  = ./
11   default_hostip = 192.168.0.55    // default is "localhost"
12
13   default_hostip_force = 192.168.0.99
14 }
```

55.3 Publications and Subscriptions for pHostInfo

The interface for pHostInfo, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pHostInfo --interface or -i
```

55.3.1 Variables Published by pHostInfo

The primary output of pHostInfo to the MOOSDB are the following five PHI_* variables. Once these variables are published, pHostInfo does not publish them again unless requested by receiving mail

HOST_INFO_REQUEST. Thus pHostInfo is mostly idle once the below five variables are posted.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- PHI_HOST_IP: The single best guess of the Host's IP address.
- PHI_HOST_IP_ALL: A comma-separated list of IP addresses if multiple addresses detected.
- PHI_HOST_IP_VERBOSE: A comma-separated list of IP addresses, with source information, if multiple addresses detected.
- PHI_HOST_PORT_DB: The port number of the MOOSDB for this community.
- PHI_HOST_PORT_INFO: A comma-separated list of parameter-value pairs describing all relevant aspects of the host.

55.3.2 Variables Subscribed for by pHostInfo

The pHostInfo application subscribes to the following MOOS variable:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 92.5.
- HOST_INFO_REQUEST: A request to re-determine and re-post the platform's host information.
- PSHARE_INPUT_SUMMARY: The input routes used by pShare for listening for incoming messages from other MOOSDBs.

55.3.3 Command Line Usage of pHostInfo

The pHostInfo application is typically launched with pAntler, along with a group of other modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ pHostInfo --help or -h
```

Listing 55.91: Command line usage for the pHostInfo tool.

```
1  Usage: pHostInfo file.moos [OPTIONS]
2
3  Options:
4    --alias=<ProcessName>
5        Launch pHostInfo with the given process
6        name rather than pHostInfo.
7    --example, -e
8        Display example MOOS configuration block
9    --help, -h
10       Display this help message.
11   --HOSTIP=<HostIP>
12       Force the use of the given IP address as the reported IP
13       address ignoring any other auto-discovered IP address.
14   --interface, -i
15       Display MOOS publications and subscriptions.
16   --version,-v
17       Display the release version of pHostInfo.
```

```
18
19  Note: If argv[2] is not of one of the above formats
20        this will be interpreted as a run alias. This
21        is to support pAntler launching conventions.
```

55.4 Usage Scenarios for the pHostInfo Utility

55.4.1 Handling Multiple IP Addresses

It is possible that a machine has more than one valid IP address at any given time, e.g., if its ethernet cable is plugged in, and it has a wireless connection. In this case, pHostInfo will make a guess that the ethernet connection takes precedent, and it will report this in the variable PHI_HOST_IP. The full set of IP addresses can be found in the other postings. For example it may not be uncommon to see something like the following three postings at one time:

```
PHI_HOST_IP          = 118.10.24.23
PHI_HOST_IP_ALL      = 118.10.24.23,169.224.126.40
PHI_HOST_IP_VERBOSE = OSX_ETHERNET2=118.10.24.23,OSX_AIRPORT=169.224.126.40
```

55.5 A Peek Under the Hood

The pHostInfo application currently only works for GNU/Linux and Apple OS X. It determines the IP information by making a system call within C++. A system call when generated will act as if the argument were typed on the command line. In this case the system call is generated and the output is redirected to a file. In a second step, pHostInfo then tries to read the IP address information from those files.

In GNU/Linux, the system call is based on the `ifconfig` command. In OS X, the system call is based on the `networksetup` command. Rather than determining in pHostInfo whether the user is running in a GNU/Linux or OS X environment, the system calls for both are invoked. Presumably a system call on a command not found in the user's shell path will not generate something that is confusable with a valid IP address.

55.5.1 Temporary Files

The temporary files are written to the user's home directory by default. This may be changed with the temp_file_dir cofiguration parameter, for example, temp_file_dir=/tmp. The set of temporary files are put into a folder named `.phostinfo/`. The set of temporary files may look like:

```
  $ cd ~/.phostinfo
  $ ls -a .ipinfo*
.ipinfo_linux_ethernet.txt    .ipinfo_osx_airport.txt     .ipinfo_osx_ethernet2.txt
.ipinfo_osx_wifi.txt          .ipinfo_linux_wifi.txt      .ipinfo_osx_ethernet1.txt
.ipinfo_osx_ethernet.txt
```

Some of these files may be empty, or some may contain error output if one of the system commands was not found, or was given an improper argument. The pHostInfo app will try to parse all of them to find a valid IP address. If more than one IP address is found, then this handled in the manner described previously in Section 55.4.1.

55.5.2 Possible Gotchas

The system calls invoked by `pHostInfo` need to be in the users shell path. A typical user default environment would have these in their shell path anyway, but it may be worth checking if things aren't working properly. Below is a list of commands that are run under the hood, and their probable locations on your system.

```
For Linux:
  /sbin/ifconfig
  /bin/grep
  /usr/bin/cut
  /usr/bin/awk
  /usr/bin/print
For OS X:
  /usr/sbin/networksetup
```

56 uPokeDB: Poking the MOOSDB from the Command Line

56.1 Overview

The uPokeDB application is a lightweight process that runs without any user interaction for writing to (poking) a running MOOSDB with one or more variable-value pairs. It is run from a console window with no GUI. For example, the alpha example mission is normally kicked off by hitting the DEPLOY button. The same could be accomplished from the terminal with:

```
$ uPokeDB alpha.moos DEPLOY=true, MOOS_MANUAL_OVERIDE=false
```

After accepting variable-value pairs from the command line, uPokeDB connects to the MOOSDB, displays the variable values prior to poking, performs the poke, displays the variable values after poking, and then disconnects from the MOOSDB and terminates. It also accepts a .moos file as a command line argument to grab the IP and port information to find the MOOSDB for connecting. Other than that, it does not read a uPokeDB configuration block from the .moos file.

56.1.1 Other Methods for Poking a MOOSDB

There are few other MOOS applications capable of poking a MOOSDB. The uMS (MOOS Scope) is an application for both monitoring and poking a MOOSDB. It is substantially more feature rich than uPokeDB, and depends on the FLTK library. The iRemote application can poke the MOOSDB by using the CustomKey parameter, but is limited to the free unmapped keyboard keys, and is good when used with some planning ahead. The latest versions of uMS and iRemote are maintained on the Oxford MOOS website. The uTermCommand application is a tool primarily for poking the MOOSDB with a pre-defined list of variable-value pairs configured in its .moos file configuration block. The user initiates each poke by entering a keyword at a terminal window. Unlike iRemote it associates a variable-value pair with a key *word* rather than a keyboard key. The uTimerScript application is another tool for poking the MOOSDB with a pre-defined list of variable-value pairs configured in its .moos file configuration block. Unlike uTermCommand, uTimerScript will poke the MOOSDB without requiring further user action, but instead executes its pokes based on a timed script. The uMOOSPoke application, written by Matt Grund, is similar in intent to uPokeDB in that it accepts a command line variable-value pair. uPokeDB has a few additional features described below, namely multiple command-line pokes, accepting a .moos file on the command-line, and a MOOSDB summary prior and after the poke.

56.2 Command-line Arguments of uPokeDB

The command-line invocation of uPokeDB accepts two types of arguments - a .moos file, and one or more variable-value pairs. The former is optional, and if left unspecified, will infer that the machine and port number to find a running MOOSDB process is localhost and port 9000. The uPokeDB process does not otherwise look for a uPokeDB configuration block in this file. The variable-value pairs are delimited by the '=' character as in the following example:

```
$ uPokeDB alpha.moos FOO=bar TEMP=98.6 MOTTO="such is life" TEMP_STRING:=98.6
```

Since white-space characters on a command line delineate arguments, the use of double-quotes must be used if one wants to refer to a string value with white-space as in the third variable-value pair above. The value *type* in the variable-value pair is assumed to be a double if the value is numerical, and assumed to be a string type otherwise. If one really wants to poke with a string type that happens to be numerical, i.e., the string "98.6", then the ":=" separator must be used as in the last argument in the example above. If uPokeDB is invoked with a variable type different than that already associated with a variable in the MOOSDB, the attempted poke simply has no effect.

56.3 MOOS Poke Macro Expansion

The uPokeDB utility supports macro expansion for timestamps. This may be used to generate a proxy posting from another application that uses timestamps as part of it posting. The macro for timestamps is `@MOOSTIME`. This will expand to the value returned by the MOOS function call `MOOSTime()`. This function call is implemented to return UTC time. This following is an example:

```
$ uPokeDB file.moos FOOBAR=color=red,temp=blue,timestamp=@MOOSTIME
```

The above poke would result in a posting similar to:

```
FOOBAR = color=red,temp=blue,timestamp=10376674605.24
```

As with other pokes, if the macro is part of a posting of type double, the timestamp is treated as a double. The posting

```
$ uPokeDB file.moos TIME_OF_START=@MOOSTIME
```

would result in the posting of type double for the variable TIME_OF_START, assuming it has not been posted previously as a different type.

56.4 Providing the ServerHost and ServerPort on the Command Line

The specification of a MOOS file on the command line is optional. The only two pieces of information uPokeDB needs from this file are (a) the `server_host` IP address, and (b) the `server_port` number of the running MOOSDB to poke. These values can instead be provided on the command line:

```
$ uPokeDB FOO=bar --host=18.38.2.158  --port=9000
```

If the `host` or the `port` are not provided on the command line, and a MOOS file is also not provided, the user will be prompted for the two values. Since the most common scenario by convention has the MOOSDB running on the local machine ("localhost") with port 9000, these are the default values and the user can simply hit the return key.

```
$ uPokeDB FOO=bar                      // User launches with no server host/port info
$ Enter Server: [localhost]            // User accepts default by hitting Return key
$    The server is set to "localhost"  // Server host confirmed to be set to "localhost"
$ Enter Port: [9000] 9123              // User overrides the default 9000 port with 9123
$    The port is set to "9123"         // Server port confirmed to be set to "9123"
```

56.5 Session Output from uPokeDB

The output in Listing 92 shows an example session when a running MOOSDB is poked with the following invocation:

```
$ uPokeDB alpha.moos DEPLOY=true RETURN=true
```

Lines 1-16 are standard output of a MOOS application that has successfully connected to a running MOOSDB. Lines 19-23 indicate the value of the variables prior to being poked, along with their source, i.e., the MOOS process responsible for publishing the current value to the MOOSDB, and the time at which it was last written. The time is given in seconds elapsed since the MOOSDB was started. Lines 26-30 show the new state of the poked variables in the MOOSDB after uPokeDB has done its thing.

Listing 56.92: An example uPokeDB session output.

```
 1  -----------------------------------------------------
 2  |       This is an Asynchronous MOOS Client         |
 3  |       c. P. Newman U. Oxford 2001-2012            |
 4  -----------------------------------------------------
 5
 6  ---------------MOOS CONNECT-----------------------
 7    contacting a MOOS server localhost:9000 -  try 00001
 8    Contact Made
 9    Handshaking as "uPokeDB"........ [OK]
10  -------------------------------------------------
11
12  uPokeDB is Running:
13   +Baseline AppTick   @ 5.0 Hz
14   +Comms is Full Duplex and Asynchronous
15   +Iterate Mode 0 :
16     -Regular iterate and message delivery at 5 Hz
17
18
19  PRIOR to Poking the MOOSDB
20    VarName               (S)ource    (T)ime      VarValue
21    ----------------      ----------  ----------   -------------
22    DEPLOY                 uTimerScript1.92        "true"
23    RETURN                 pHelmIvP    1           "false"
24
25
26  AFTER Poking the MOOSDB
27    VarName               (S)ource    (T)ime      VarValue
28    ----------------      ----------  ----------   -------------
29    DEPLOY                 uPokeDB     22.58       "true"
30    RETURN                 uPokeDB     22.58       "false"
```

56.6 Publications and Subscriptions for uPokeDB

56.6.1 Variables published by the uPokeDB application

- USER-DEFINED: The only variables published are those that are poked. These variables are provided on the command line. See Section 56.2.

56.6.2 Variables subscribed for by the uPokeDB application

- `USER-DEFINED`: Since uPokeDB provides two reports as described in the above Section 56.5, it subscribes for the same variables it is asked to poke, so it can generate its before-and-after reports.

57 pEchoVar: Re-publishing Variables Under a Different Name

57.1 Overview

The pEchoVar application is a lightweight process that runs without any user interaction for "echoing" the posting of specified variable-value pairs with a follow-on posting having different variable name. For example the posting of FOO=5.5 could be echoed such that BAR=5.5 immediately follows the first posting. The motivation for this tool was to convert, for example, a posting such as GPS_X to become NAV_X. The former is the output of a particular device, and the latter is a de facto standard for representing the vehicle's longitudinal position in local coordinates.

57.2 Using pEchoVar

Configuring pEchoVar minimally involves the specification of one or more `echo` or `flip` mapping events. It may also optionally involve specifying one or more logic conditions that must be met before mapping events are posted.

57.2.1 Configuring Echo Mapping Events

An *echo event mapping* maps one MOOS variable to another. Each mapping requires one line using the `echo` configuration parameter of the form:

```
echo = <MOOSVar> -> <MOOSVar>
```

The source `<MOOSVar>` and target `<MOOSVar>` components are case sensitive since they are MOOS variables. A source variable can be echoed to more than one target variable. If the set of lines forms a cycle, this will be detected and pEchoVar post a configuration and run warning and cease to perform any function whatsoever. An example configuration is given in Listing 1.

Listing 57.1: An example pEchoVar configuration block.

```
1  //---------------------------------------------------------
2  // pEchoVar configuration block
3
4  ProcessConfig = pEchoVar
5  {
6    AppTick   = 20
7    CommsTick = 20
8
9    echo = GPS_X           ->  NAV_X
10   echo = GPS_Y           ->  NAV_Y
11   echo = COMPASS_HEADING ->  NAV_HEADING
12   echo = GPS_SPEED       ->  NAV_SPEED
13 }
```

57.2.2 Configuring Flip Mapping Events

The pEchoVar application can be used to "flip" a variable rather than doing a simple echo. A flipped variable, like an echoed variable, is one that is republished under a different name, but a flipped variable parses the contents of a string comprised of a series of `variable=value` comma-separated

pairs, and republishes a portion of the series under the new variable name. For example, the following string,

```
ALPHA = "xpos=23, ypos=-49, depth=20, age=19.3, certainty=1"
```

may be flipped to publish the below new string, with the fields `xpos`, `ypos`, and `depth` replaced with `x`, `y`, `vehicle_depth` respectively.

```
BRAVO = "x=23, y=-49, vehicle_depth=20"
```

The above "flip relationship" is configured with the `flip` configuration parameter with the following form:

```
flip:<key>   = source_variable  = <variable>
flip:<key>   = dest_variable    = <variable>
flip:<key>   = source_separator = <separator>
flip:<key>   = dest_separator   = <separator>
flip:<key>   = filter           = <variable>=<value>
flip:<key>   = component        = <old-field> -> <new-field>
flip:<key>   = component        = <old-field> -> <new-field>
```

The relationship is distinguished with a `<key>`, and several components. The `source_variable` and `dest_variable` components are mandatory and must be different. The `source_separator` and `dest_separator` components are optional with default values being the string `","`. Fields in the source variable will only be included in the destination variable if they are specified in a component mapping `<old-vield> -> <new-field>`. The example configuration in Listing 2 implements the above described example flip mapping. In this case only postings that satisfy the further filter, `certainty=1`, will be posted.

Listing 57.2: An example pEchoVar configuration block with flip mappings.

```
1  //--------------------------------------------------------------
2  // pEchoVar configuration block
3
4  ProcessConfig = pEchoVar
5  {
6    AppTick   = 10
7    CommsTick = 10
8
9    flip:1   = source_variable  = ALPHA
10   flip:1   = dest_variable    = BRAVO
11   flip:1   = source_separator = ,
12   flip:1   = dest_separator   = ,
13   flip:1   = filter           = certainty=1
14   flip:1   = component        = ypos -> y
15   flip:1   = component        = xpos -> x
16 }
```

Some caution should be noted with flip mappings - the current implementation does not check for cycles, as is done with echo mappings.

57.2.3 Applying Conditions to the Echo and Flip Operation

The execution of the mappings configured in pEchoVar may be configured to depend on one or more logic conditions. If conditions are specified in the configuration block, all specified logic conditions must be met or else the posting of echo and flip mappings will be suspended. The logic conditions are configured with the condition parameter as follows:

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

The <logic-expression> syntax is described in the Appendix document on logic utilities, and may involve the simple comparison of MOOS variables to specified literal values, or the comparison of MOOS variables to one another. If a condition parameter is specified, pEchoVar will automatically subscribe to all MOOS variables used in the condition expressions.

57.2.4 Holding Outgoing Messages Until Conditions are Met

If the conditions are not met, all incoming mail messages that would otherwise result in an echo or flip posting, are held. When or if the conditions are met at some point later, those mail messages are processed in the order received and echo and flip mappings may be posted en masse. However, if several mail messages for a given MOOS variable are received and stored while conditions are unmet, only the latest received mail message for that variable will be processed. As an example, consider pEchoVar configured with the below two lines:

```
echo      = FOO -> BAR
condition = DEGREES <= 32
```

If the condition is not met for some period of time, and the following mail were received during this interval: FOO="apples", FOO="pears", FOO="grapes", followed by DEGREES=30, then pEchoVar would post BAR="grapes" immediately on the very iteration that the DEGREES=30 message was received. Note that BAR="apples" and BAR="pears" would never be posted. This is to help ensure that the pEchoVar memory doesn't grow unbounded by holding onto all mail while conditions are unmet.

The user may alternatively configure pEchoVar to *not* hold incoming mail messages when or if it is in a state where its logic conditions are not met. This can be done with the hold_messages parameter:

```
hold_messages = false   // The default is true
```

When configured this way, upon meeting the specified logic conditions, pEchoVar will begin processing echo and flip mappings when or if new mail messages are received relevant to the mappings. In the above example, once DEGREES=30 is received by pEchoVar, nothing would be posted until new incoming mail on the variable FOO is received (not even BAR="grapes").

57.2.5 Limiting the Echo Posting Frequency to the AppTick Setting

By default, when the conditions are met, an echo posting is made once for each incoming piece of mail related that echo mapping. If FOO is echo mapped to BAR, and if 40 pieces of incoming mail for

FOO are received on one iteration, 40 postings are made to BAR on that iteration. Instead one may wish that, on each iteration where there is posting ready for BAR, that only the latest value for BAR be made. This may be arranged with the echo_latest_only parameter:

```
echo_latest_only = true    // The default is false
```

In this case, the frequency of postings to BAR and all other echo mappings will occur at most at a frequency equal to the AppTick setting.

57.3 Configuring for Vehicle Simulation with pEchoVar

When in simulation mode with uSimMarine, the navigation information is generated by the simulator and not the sensors such as GPS or compass as indicated in lines 9-12 in Listing 1. The simulator instead produces USM_* values which can be echoed as NAV_* values as shown in Listing 3.

Listing 57.3: An example pEchoVar configuration block during simulation.

```
1  //------------------------------------------------------------
2  // pEchoVar configuration block (for simulation mode)
3
4  ProcessConfig = pEchoVar
5  {
6    AppTick   = 20
7    CommsTick = 20
8
9    echo = USM_X       ->  NAV_X
10   echo = USM_Y       ->  NAV_Y
11   echo = USM_HEADING ->  NAV_HEADING
12   echo = USM_SPEED   ->  NAV_SPEED
13 }
```

Note in more recent versions of uSimMarine the simulator output may be changed to have a NAV_ prefix by setting prefix = NAV_, obviating the use of pEchoVar configured as above.

57.4 Configuration Parameters for pEchoVar

The following parameters are defined for pEchoVar. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 57.4: Configuration Parameters for pEchoVar.

echo:	A mapping from one MOOS variable to another constituting an echo. Section 57.2.1.
echo_latest_only:	If true, only the latest value of variable will be echoed on each iteration, even if several pieces of incoming mail have been received since the last posting. Legal values: true, false. The default is false. Section 57.2.1.
condition:	A logic condition that must be met or all echo and flip publications are held. Section 57.2.3.
flip:	A description of how components from one variable are re-posted under another MOOS variable. Section 57.2.2.

hold_messages: If true, messages are held when conditions are not met for later processing when logic conditions are indeed met. Legal values: true, false. The default is true. Section 57.2.3.

57.5 Publications and Subscriptions for pEchoVar

The interface for pEchoVar, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pEchoVar --interface or -i
```

57.5.1 Variables Posted by pEchoVar

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 57.6.
- <USER-DEFINED>: Any MOOS variable specified in either the echo or flip config parameters.

57.5.2 Variables Subscribed for by pEchoVar

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- <USER-DEFINED>: Any MOOS variables found in the antecedent of either the echo or flip mappings. It will also subscribe for any MOOS variable found in any of its logic conditions.

57.6 Terminal and AppCast Output

The pEchoVar application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 5 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. See the documentation on uMAC viewing utilities for more on appcasting and viewing appcasts. The counter on the end of line 2 in parentheses is incremented on each iteration of pEchoVar, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 57.5: Example terminal or appcast output for pEchoVar.

```
1  ============================================================
2  pEchoVar alpha                                        0/0(81)
3  ============================================================
4  conditions_met:   true
5  hold_messages:    true
6  echo_latest_only: false
7
8  ================================================
9  Echoes: (5)
10 ================================================
11
12 Source          Dest          Hits  Posts
```

```
13  ---------  ---  -------------  ----  -----
14  NAV_X      -->  NAV_XX         324   324
15  NAV_X      -->  NAV_XPOS       324   324
16  NAV_Y      -->  NAV_YY         324   324
17  NAV_Y      -->  NAV_YPOS       324   324
18  NAV_SPEED  -->  NAV_SPEED_ALT  324   324
19
20  ================================================
21  Flips: (1)
22  ================================================
23
24                                       Src  Dest              Old    New
25  Key  Hits  Source             Dest     Sep  Sep   Filter      Field  Field
26  ---  ----  -----------------  ------   ---  ----  ----------  -----  -----
27  1    162   NODE_REPORT_LOCAL  FOOBAR   ,    #     type:kayak  X      xpos
28  1    162   NODE_REPORT_LOCAL  FOOBAR   ,    #     type:kayak  Y      ypos
```

Lines 4 indicates whether or not any specified logic conditions have been met. This line will also read `true` even if no logic conditions were provided. Lines 5 and 6 simply confirm the user's settings for the `hold_messages` and `echo_latest_only` parameters discussed in Sections 57.2.4 and 57.2.5 respectively.

Lines 12-18 convey the configured echo mappings, one for each line. At the end of each line, the *Hits* column shows the number of incoming mails received for that variable. The number of times it is echoed, or re-posted is shown under the *Posts* column. When `echo_latest_only` is false, these numbers should match. Lines 20-28 convey the configure flips. A single flip configuration, identified by its key, may have several lines, as in this example. Here the only difference between lines 27 and 28 are the flip components. One maps `X` to `xpos`, and the other maps `Y` to `ypos`.

The terminal or appcast output shown in Listing 5 above may be seen first hand by running the Alpha example mission. The below configuration block in Listing 6 corresponds to the above appcast output. The user just needs to add `pEchoVar` to the Antler launch list.

Listing 57.6: Example pEchoVar configuration from the Alpha example mission.

```
1  ProcessConfig = pEchoVar
2  {
3    AppTick = 1
4    CommsTick = 1
5
6    echo = NAV_X     -> NAV_XX
7    echo = NAV_X     -> NAV_XPOS
8    echo = NAV_Y     -> NAV_YY
9    echo = NAV_Y     -> NAV_YPOS
10   //echo = NAV_YY    -> FOOBAR
11   //echo = FOOBAR    -> NAV_Y
12   echo = NAV_SPEED -> NAV_SPEED_ALT
13
14   FLIP:1    = source_variable  = NODE_REPORT_LOCAL
15   FLIP:1    = dest_variable    = FOOBAR
16   FLIP:1    = source_separator = ,
17   FLIP:1    = dest_separator   = #
18   FLIP:1    = filter = type == kayak
19   FLIP:1    = component = X -> xpos
```

```
20    FLIP:1    = component = Y -> ypos
21  }
```

The two echo mappings in lines 10 and 11 may be commented out to demonstrate the detection of echo mapping cycles. The pairs of mappings in Lines 6-7 and 8-9 demonstrate that a single incoming variable may be mapped to multiple destinations.

58 pObstacleMgr: Managing Vehicle Belief State of Obstacles

58.1 Overview

The pObstacleMgr is designed to reason about obstacles and provide coordinated alerts and updates to the helm for spawning and adjusting obstacle avoidance behaviors. The obstacle manager reasons about *given* obstacles, with prior known locations, loaded at launch time and may include buoys, rocky outcrops, bridge pylons and so on. It also reasons about *sensed* objects that may be derived from LIDAR point clouds, or other sensor sources. The obstacle manager will manage both the mission-loaded and dynamic incoming data to maintain a single list of obstacles. Depending on the robot range to the obstacle, the obstacle manager will produce alerts for coordination with obstacle avoidance behaviors. And in the case of dynamically sensed obstacles, it will continually update the position and shape of the obstacle to previously spawned obstacle avoidance behaviors.

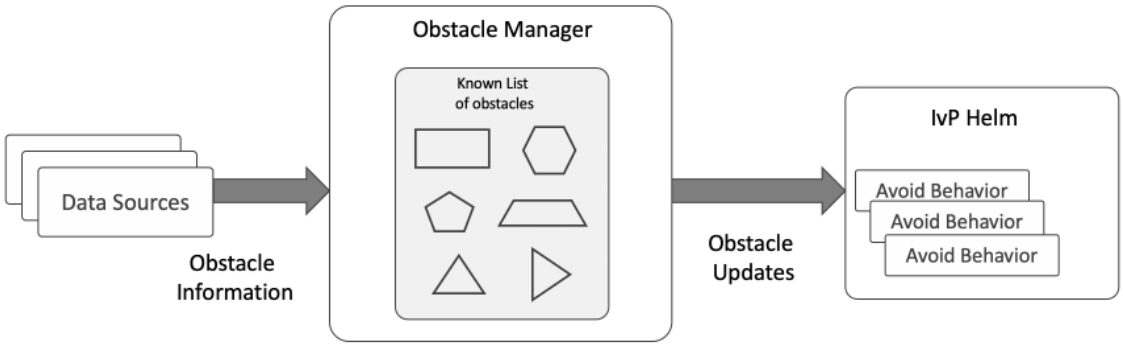

Figure 190: **The Obstacle Manager**: Obstacle information from a variety of sources is received by the obstacle manager, which maintains a list of known obstacles. This is modified with time both in terms of updating the obstacle locations as well as deleting stale obstacles. Continuous updates are posted and consumed by the helm to enable the spawning of avoidance behaviors.

The obstacle manager exists and sits between the sensor stream and the helm for two reasons. (1) This arrangement allows the helm to remain passively postured with respect to behavior spawning of obstacle avoidance behaviors. The helm will spawn behaviors based on incoming events from the obstacle manager. This is implemented in the helm in a manner that is consistent with any other type of behavior or sensor stream. Nothing special was implemented in the helm to handle obstacle information, other than the `AvoidObstacle` behavior. (2) The nature of the obstacle manager may change over time as different sensors, information sources, or outlier rejection algorithms are available. None of these changes will require a change to the helm or its behaviors.

58.2 Using the Obstacle Manager

To use the obstacle manager there are a few steps and considerations.

- The pObstacleMgr app must be run on the vehicle, by adding it to the Antler block for the vehicle. An example configuration block is given in Section 70.4.1.
- Obstacle information must be fed to the obstacle manager from either one of three sources. Either it (a) arrives from a sensor-based source as labeled points as decribed in Section 70.2.1,

or (b) arrives as convex polygon obstacle mail message in the variable GIVEN_OBSTACLE, or (c) the obstacle size and position information is provided at launch time from a set of obstacles known a priori, listed in the configuration file with the parameter given_obstacle, as described in Section 70.2.1.

- The helm is configured to use an obstacle avoidance behavior in a templating mode, allowing new behaviors to spawn based on output from the obstacle manager. See Section 70.2.2.

58.2.1 Obstacle Manager Input Sources

The obstacle manager may be populated with obstacle information in one of three methods as shown below. Regardless of the source, all obstacles are maintained as a list of known obstacles by the obstacle manager. And each time there is a change in shape of a known obstacle, an update is posted to the obstacle manager consumers, e.g., the helm.

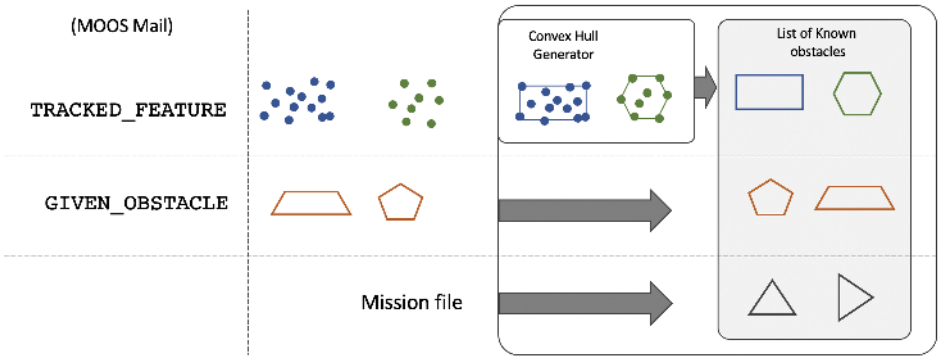

Figure 191: **Obstacle Manager Input**: Currently supported obstacle manager input comes from one of three sources. (1) As individual LIDAR points with grouping information, (2) as polygon obstacles via incoming mail, typically followed by updates for each obstacle, or (3) as polygon obstacles read from a mission configuration file.

In this section, the obstacle manager sources are described in more detail, along with how the obstacle manager processess incoming information.

Tracked Feature Inputs The obstacle manager is designed to work with one or more other applications producing tracked features. These tracked features may be points generated by a LIDAR, or some other sensor. The obstacle simulator, uFldObstacleSim, has a mode that supports generation of simulated LIDAR points. The obstacle manager subscribes for the MOOS variable TRACKED_FEATURE, of the form:

```
TRACKED_FEATURE  = x=23.2,y=19.8,label=47
TRACKED_FEATURE  = x=22.9,y=18.2,label=47
```

It is also assumed that the input will arrive with some grouping or clustering algorithm applied to each feature, reflected in the label field in the examples above. The obstacle manager maintains a database in the form of a mapping, keyed on the obstacle label, to a list of features. For each obstacle only the N most recent features (points) are held. The obstacle manager may be configured

to ignore incoming features beyond a certain range to the robot. This helps ensure bounded memory growth of the application as longer missions unfold.

For each cluster of points, the obstacle manager maintains a single convex polygon representing each cluster. By default a convex hull polygon of each cluster is maintained as shown in Figure 214.

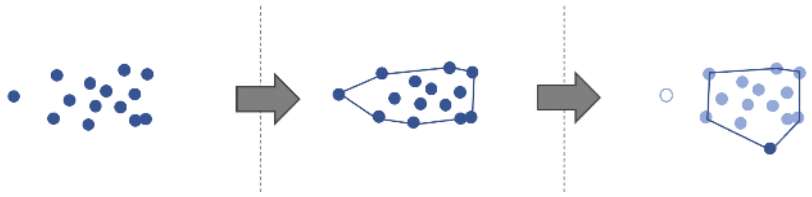

Figure 192: **The Convex Hull Generator**: Each stream of like-labeled points is a cluster from which the generator will maintain a current convex hull. As points age-out, the convex hull may shrink or shift through space. This process naturally accommodates both outlier points, and points associated with a slowly moving obstacle. In the right-hand panel of the figure, the dark blue point shows a newly received point. The white point has aged-out. The lighter blue points are not as old as the new point, but have not yet aged-out.

The stream of points, typically originating from a LIDAR, is presumed to be a constant update stream of points as new information is generated from the LIDAR. The obstacle manager holds enough points in memory to periodically generate a convex hull, but it also guards against unbounded memory by allowing points to drop. A point will be dropped based on one of two criteria. The first is a maximum total number of points held, with oldest points dropped as new points come in, i.e., first-in-first-out. The max amount is applied per-cluster. Currently there is no limit on the number of clusters. This max limit is by default 20 points, but can be set with the `max_pts_per_cluster` parameter as shown below.

```
max_pts_per_cluster = 40   // default is 20
max_age_per_point   = 10   // default is 20 seconds
```

The obstacle manager also will remove points from memory based on the age of the incoming point. By default, the point will be dropped after 20 seconds. The `TRACKED_FEATURE` message does not contain a timestamp. The age of the point is based on the timestamp the obstacle manager applies upon receipt. Dropping points based on age not only serves the purpose of bounding memory growth, it also serves as a kind of outlier rejection. A spurious LIDAR point, that may have come from a wave or some other non-obstacle phenomena, may cause a temporary growth of the convex hull, but it will not last long. Dropping points based on age also allows moving obstacles to have a convex hull that shifts with the motion of the obstacle. Each time the convex hull for an obstacle changes shape or location, an update message is produced by the obstacle manager. If there is an obstacle avoidance behavior currently in existence in the helm, tied to this obstacle, the behavior will be immediately updated, likely resulting in a slight adjustment for the output on that particular behavior.

An Alternative to Convex Hull Clustering The convex hull is general and preferred when the object is not of a known size or geometry. In certain cases when information about the object size is known a priori, the user may configure the obstacle manager to associate a regular polygon

of fixed size. The polygon is centered on the center of mass for all points that have not aged-out, as shown in Figure 215. As before, the polygon will shift with a moving obstacle as new points arrive and older points age-out.

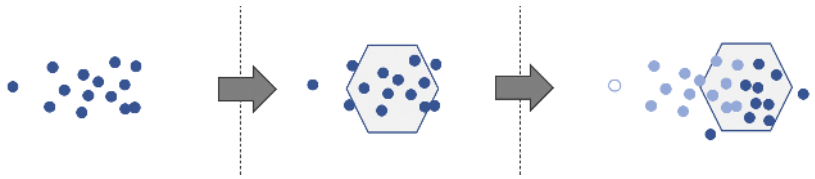

Figure 193: The obstacle manager may also maintain a regular polygon of user configurable radius and number of vertices, centered on the average of points for a given cluster.

To generate obstacle polygons in this manner the `lasso` parameter is used with the following options:

```
lasso = true                // default is false
lasso_points = 6            // default is 6
lasso_radius = 5            // (meters) default is 5
```

The first parameter turns on the lasso option, and the following two set the radius and number of vertices used in the regular polygon.

Configuring Obstacles of Given Location and Size The obstacle manager may also be configured with obstacles, in polygon form, at given shape and location. This can be done with a configuration parameter in the mission (`.moos` file), or through incoming mail. In either case, the format is the same.

To configure given obstacles through incoming mail, the format is:

```
GIVEN_OBSTACLE = pts={109,-72:113,-76:113,-82:109,-86:103,-86:99},label=23
```

Providing given obstacles through MOOS messages is convenient when using another MOOS app to generate obstacles, perhaps from different sensor input, or perhaps through simulation to stress test the autonomy system. An obstacle is distinct by its label. For any given obstacle, the message may simply arrive once, or it may be steadily updated depending on the source application. Updates are applied immediately and passed on from the obstacle manager with a posting for any consumer of information about this obstacle, e.g., an obstacle avoidance behavior in the helm.

To configure given obstacles in the mission file, the format is:

```
given_obstacle = pts={109,-72:113,-76:113,-82:109,-86:103,-86:99},label=23
```

Providing given obstacles in the mission file may be convenient if there are known obstacles such as buoys, or other items that are consistent with an operation area. Such data can simply be just loaded at mission time.

Obstacle Duration and Memory Management in the Obstacle Manager The obstacle manager holds information about all known obstacles. A policy for obstacle deletion is needed to guard against unbounded memory growth. In the case of obstacles tied to LIDAR points, there is already a policy for points to age-out after a certain duration. The obstacle manager will remove the obstacle from its list of obstacles when there are no longer any LIDAR points associated with the obstacle. This will occur naturally as the vehicle moves away from an obstacle, beyond sensor range. And if the LIDAR points were a false detection due to a sensor anomaly, or wave, the flow of points related to this false detection will typically soon cease and the obstacle is deleted from the obstacle manager memory.

The case of obstacles arriving from GIVEN_OBSTACLE messages is different. The source of these obstacles is not known by the obstacle manager, by design. By default, once the obstacle manager has received information about an obstacle, it is held by the obstacle manager forever. When the obstacle manager is consuming information in this manner, a *duration* is required as part of the mail message. For example:

```
GIVEN_OBSTACLE = pts={109,-72:113,-76:113,-82:109,-86:103,-86:99},label=23,duration=5
```

By default, duration components are mandatory and cannot be any greater than 60 seconds. This amount may be changed with the configuration parameter:

```
given_max_duration=30  // Default is 60 seconds
```

Or it may be disabled completely with:

```
given_max_duration = off
```

This compels the application generating the GIVEN_OBSTACLE information to be mindful of obstacle duration, typically updating the information for each obstacle at a rate that is faster than the posted duration. This allows the obstacle manager to delete an obstacle in the *absence* of a recent update.

Currently the only application producing GIVEN_OBSTACLE mail messages is the uFldObstacleSim application. This method of feeding the obstacle manager may also be used by a future app that has a better method of clustering LIDAR points than the simple convex hull algorithms described earlier that are currently in use by the obstacle manager

Obstacle Manager Actions Upon Deletion of an Obstacle The intended consumer of obstacle manager output is the helm. When a new obstacle is detected, if it is within the configured range of the vehicle, a new obstacle avoidance behavior is spawned. When the vehicle later goes beyond this range, the avoidance behavior will be deleted by the helm.

Normally, by the time the obstacle manager is about to delete a known obstacle, the helm has probably already deleted the corresponding obstacle avoidance behavior. However, it is possible that the obstacle became known to the obstacle manager due to spurious sensor/LIDAR data. Perhaps

this obstacle is very near the vehicle and caused the helm to spawn an obstacle avoidance behavior for this phantom obstacle. In this case the flow of sensor information related to this false obstacle may cease a very short time later, and the obstacle manager will quickly delete the obstacle from its list of known obstacls. Yet the helm may still have the avoidance behavior associated with the false obstacle, and would otherwise continue to have this behavior until the vehicle moves far enough away. This is clearly not desirable since the avoidance behavior for that short-lived false obstacle may constrain the vehicle motion in adverse ways. Instead, we want the vehicle behavior to be removed when the obstacle manager removes the obstacle. For this reason, the obstacle manager will publish the below posting whenever an obstacle is removed from its memory:

```
OBM_RESOLVED = 428
```

The helm obstacle avoidance behavior `BHV_AvoidObstacleV21` or newer, monitors for the above postings. If it notes that the obstacle id for which it is associated has been resolved, the behavior will put in motion the steps to self-delete immediately.

58.2.2 Configuring the Helm to Handle Obstacle Manager Output

The obstacle manager exists primarily to serve the helm, by posting notifications to the helm that allow the helm to (a) spawn an obstacle avoidance behavior for a new obstacle, and (b) update a previously spawned obstacle avoidance behavior with an updated location or shape of the obstacle.

Configuration on the helm side is straight-forward, requiring only a configuration block for the obstacle avoidance behavior, similar to:

```
//------------------------------------------
Behavior=BHV_AvoidObstacleV21
{
  name       = avd_obstacles_
  pwt        = 500
  condition  = DEPLOY = true
  templating = spawn
  updates    = OBSTACLE_ALERT

     allowable_ttc = 5
       buffer_dist = 3
    pwt_outer_dist = 20
    pwt_inner_dist = 10
    completed_dist = 25
}
```

See the documentation for the obstacle avoidance behavior for more details on the above parameters. The important point here is that upon startup of the helm, no obstacle avoidance behavior will be spawned until the helm receives an alert about an obstacle. This alert comes via a posting to the variable `OBSTACLE_ALERT`. Until an alert arrives, this behavior exists in the helm as a *template*, capable of spawning any number of behaviors, one for each obstacle.

When the helm starts, on behalf of the obstacle avoidance behavior template, the helm posts an alert request:

```
OBM_ALERT_REQUEST = alert_range=20, update_var=OBSTACLE_ALERT
```

The alert request uses the value of the `updates` parameter and the `pwt_outer_dist` parameter to construct the alert request. The `alert_range` component of the alert request is automatically set to match the `pwt_outer_dist` configuration parameter of the behavior. In the behavior, when the obstacle is at, or beyond this range, the priority weight of the behavior becomes zero. This same range value will inform the obstacle manager that (1) until the obstacle is closer than this distance, postings to the `OBSTACLE_ALERT` should not be made for this obstacle id, and (2) after the obstacle has become farther than this distance, the same said postings should cease. After the vehicle has opened range to the obstacle beyond the `completed_dist`, the behavior will complete and will be removed from them helm.

Note that if there are say ten instances of this behavior, for ten separate obstacles, they will all receive their updates through the same `OBSTACLE_ALERT` MOOS variable. Each posting, however, will contain the name of the behavior. For example:

```
OBSTACLE_ALERT = name=avd_obstacles_ob_08#poly=pts={52.2,-32.2:53,-33.02:53,...
OBSTACLE_ALERT = name=avd_obstacles_ob_03#poly=pts={52.82,-115.86:50.64,...
OBSTACLE_ALERT = name=avd_obstacles_ob_02#poly=pts={72.07,-68.93:75.15,...
```

The helm will ensure the updates are only applied to the behavior that matches the name in the update.

58.3 Implementation of the Obstacle Manager

58.3.1 Obstacles versus Contacts

The obstacle manager was designed to handle stationary obstacles like buoys or bridge pylons. To handle moving obstacles such as other marine vessels, the IvP Helm uses a similar MOOS application called a contact manager and a collision avoidance behavior based on COLREGS protocol. This is outside the scope of this paper. Our convention is to use the term *obstacle* for objects that are stationary or at best slowly drifting. Obstacles are handled by the obstacle manager and the Avoid Obstacle behavior. We use the term *contact* for moving vessels. Contacts are handled by the contact manager, and the Collision Avoidance behaviors.

58.3.2 Drifting Obstacles

The obstacle manager is equipped with the ability to handle *drifting* obstacles, e.g., a drifting buoy. In effect, there is not much difference between a truly drifting object and an object with slightly shifting sensor readings. Proper functioning of the obstacle manager depends on proper configuration of the *decay* of sensed points. As the drifting obstacle moves, sensed points at the new location will appear, and sensed points at older locations will become stale and disappear from the obstacle manager memory. By default the number of sensed points is limited in size, per cluster key, to 20 points. This can be changed with the parameter `max_pts_per_cluster`. By default, the points will decay and be removed from memory after 20 seconds. This can be changed wit the parameter `max_age_per_point`.

58.3.3 Obstacle and Alert Management

The obstacle manager by default will not generate any alerts unless another app has indicated that it would like to receive alerts. This alert request comes in the form of a message:

```
OBM_ALERT_REQUEST = update_var=OBSTACLE_ALERT, alert_range=40, name=avd_obstacle
```

For the current release of the obstacle manager, an alert of a single configuration is supported. Subsequent publications to OBM_ALERT_REQUEST, with different values for the alert variable or range, will simply overwrite the previous setting.

Alert Generation and The Alert Range The obstacle manager will generate alerts about an obstacle to the variable requested in the OBM_ALERT_REQUEST message. The first time this alert variable is published, it can be regarded as alert in the sense that the obstacle's existence is new information for whomever is subscribing for these alerts (typically the helm). A couple example postings are shown below.

```
OBSTACLE_ALERT = name=ob_4#poly=pts={48.7,-77.2:52.3,-80.8:52.3,-86:48.7,-89.6:43.5,\
                 -89.6:39.9,-86:39.9,-80.8:43.5,-77.2},label=ob_4

OBSTACLE_ALERT = name=ob_2#poly=pts={62.9,-48.7:67.1,-52.9:67.1,-58.9:62.9,-63.1:56.9,\
                 -63.1:52.7,-58.9:52.7,-52.9:56.9,-48.7},label=ob_2
```

Subsequent alert publications are essentially updates on the obstacle. These subsequent publications are only made if the size or the position of the obstacle changes. For sensed obstacles derived from point data or other dynamic data, alert updates are fairly frequent. For given static obstacles, subsequent alert updates may never happen after the initial alert.
When the robot is beyond the *alert range* to an obstacle, the obstacle manager will no longer generate alerts. Alert publications will resume as soon as the robot returns to within the alert range. For the relatively static given obstacles, the obstacle manager takes care to re-publish an alert for the obstacle when the robot returns within the alert range, even if the size or position of the obstacle has not changed. A vehicle returning within the alert range always needs to be re-alerted as if encountering this obstacle for the very first time.

Resolution of Alerts The obstacle manager will generate a single alert for each obstacle, thereafter assuming the entity that needed to know about the obstacle has been properly notified. For dynamic obstacles, alerts will continue as the position or shape of the obstacle changes.

The helm may at some point want to delete the, e.g., obstacle avoidance, behavior that registered for the alert, typically when the obstacle has been passed and has reached a range where it is no longer a concern. In the case of the `AvoidObstacle` behavior, the behavior will be deleted when the obstacle is beyond the completed_dist range, and this range is tied to be equivalent to the requested alert range.

The Ignore Range For dynamic obstacles, receiving tracked feature point information, normally all such points are received and processed, regardless of range. The ignore_range parameter sets

a distance, in meters, beyond which an incoming point will be ignored. This can help reduce management of spurious obstacles at obviously harmless ranges to ownship. By default this value is −1, meaning all points are received and managed. In future releases this range may be automatically tied to the alert range, and ignore regions will also be supported, to reject points on land or outside of the vehicle operation area.

58.4 Configuration Parameters for pObstacleMgr

The following parameters are defined for pObstacleMgr. A more detailed description is provided in other parts of this section. Parameters having default values are indicated.

Listing 58.7: Configuration Parameters for pObstacleMgr.

alert_range:	The range in meters, between ownship and an obstacle, that an alert will be triggered. Section 70.3.3.
ignore_range:	The range in meters, between an incoming point (tracked feature) and ownship, beyond which the point will be ignored.
lasso:	If `true`, the polygon associated with each cluster will be firmly set to a circular polygon of a set radius and set number of vertices. The default is `false`. Section 70.2.1.
lasso_points:	The number of points used in the regular polygon representing the obstacle, if the lasso parameter is set to `true`. The default is 6, a hexagon. Section 70.2.1.
lasso_radius:	The radius (distance to any vertex) used in the regular polygon representing the obstacle, if the lasso parameter is set to `true`. The default is 5 meters. Section 70.2.1.
max_age_per_point:	The maximum number of seconds that a point (tracked feature) will be retained in memory. Beyond this number, points will be dropped. The default is 20 seconds. Section 70.3.2.
max_pts_per_cluster:	The maximum number of points (tracked features) retained in memory per cluster (points having the same label). Beyond this number, oldest points will be dropped. The default is 20 points. Section 70.3.2.
point_var:	The name of the MOOS variable to looked for tracked features. The default is TRACKED_FEATURE. Section 70.2.1.
obstacles_color:	When the obstacle manager is rendering obstacles (post_view_polys is `true`), this parameter will set the obstacle color. The default value is `blue`.
poly_label_thresh:	When the number of obstacle polygons is greater than this parameter, the view polylgons will be published with a label color of `invisible`, resulting in less work for the pMarineViewer. The default is 25. This is solely to help boost performance of pMarineViewer in extreme uses cases of very large numbers of obstacles and high time warp, and possibly slower machines. It does not affect the function of the obstacle manager in any other way.

poly_shade_thresh: When the number of obstacle polygons is greater than this parameter, the view polylgons will be published with a shade (fill) color of `invisible`, resulting in less work for the pMarineViewer. The default is 100. This is solely to help boost performance of pMarineViewer in extreme uses cases of very large numbers of obstacles and high time warp, and possibly slower machines. It does not affect the function of the obstacle manager in any other way.

poly_vertex_thresh: When the number of obstacle polygons is greater than this parameter, the view polylgons will be published with a vertex size of zero, resulting in less work for the pMarineViewer by only rendering the polygon edges, without the vertices. The default is 150. This is solely to help boost performance of pMarineViewer in extreme uses cases of very large numbers of obstacles and high time warp, and possibly slower machines. It does not affect the function of the obstacle manager in any other way.

post_dist_to_polys: Legal values are `true`, `false`, or `close`. If `true`, the distance from ownship to an obstacle is published for each obstacle, to the variable OBM_DIST_TO_OBJ. When set to `close`, these publications only occur when the obstacle is closer than alert_range. When `false`, these postings are turned of completely. The default is `close`.

post_view_polys: When `true`, the obstacle polygons are published published by the obstacle manager. Normally this is redundant since the obstacles are also published by the obstacle avoidance behavior and the obstacle simulator. Legal values are `true` and `false`. The default is `false`.

58.4.1 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ pBasicContactMgr --example or -e
```

This will show the output shown in Listing 39 below.

Listing 58.8: A Simple pObstacleMgr Example.

```
1  ============================================================
2  pObstacleMgr Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pObstacleMgr
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   point_var = TRACKED_FEATURE  // default is TRACKED_FEATURE
11
12   given_obstacle = pts={90.2,-80.4:....:85.4,-80.4},label=ob_23
13
14   post_dist_to_polys = true  // true, false or (close)
```

```
15    post_view_polys = true     // (true) or false or
16
17    max_pts_per_cluster = 20   // default is 20
18    max_age_per_point   = 20   // (secs)  default is 20
19
20    alert_range  = 20          // (meters) default is 20
21    ignore_range = -1          // (meters) default is -1, (off)
22
23    lasso = true               // default is false
24    lasso_points = 6           // default is 6
25    lasso_radius = 5           // (meters) default is 5
26
27    obstacles_color = color    // default is blue
28
29    // To squeeze more viewer effic when large # of obstacles:
30    poly_label_thresh = 25     // Set label color=off if amt>25
31    poly_shade_thresh = 100    // Set shade color=off if amt>100
32    poly_vertex_thresh = 150   // Set vertex size=0 if amt>150
33  }
```

58.5 Publications and Subscriptions of pObstacleMgr

The interface for pObstacleMgr, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pObstacleMgr --interface or -i
```

58.5.1 Variables Published by pObstacleMgr

The output of pObstacleMgr is:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- [ALERT_VAR]: The obstacle manager will publish alerts through a variable specified in an alert configuration, via the incoming OBM_ALERT_REQUEST message. See Section 70.3.3.
- VIEW_POLYGON: The polygon representing the obstacle is posted in this variable. It is re-published when/if the shape or location changes, or upon ownship approaching within range of the obstacle.
- OBM_DIST_TO_OBJ: For each object retained in the object manager's memory, object manager will continually post the distance from ownship to the obstacle, in meters. This posting can disabled by setting the post_dist_to_polys to `false`. By default it is enabled.
- OBM_CONNECT: Upon successful launch of the obstacle manager, this variable is posted. It helps coordinate with the obstacle simulator which may be running on the shoreside and may have already published obstacle information. Upon receipt of this posting, the obstacle simulator will refresh the postings.
- OBM_MIN_DIST_EVER: The obstacle manager uses its knowlege of all obstacle locations and ownship location and keeps track of the closest that an obstacle has ever come to ownship. This minimum distance, and the id of the obstacle, are posted to this variable.

- OBM_RESOLVED: When an obstacle is removed from the obstacle manager, a notice is posted containing just the id of the removed obstacle. This may be used by the obstacle avoidance behavior in the helm to let it know that this obstacle no longer exists. Section **??**.

The obstacle manager will also publish to whatever MOOS variables are specified in the obstacle alerts. See Section 70.3.3.

58.5.2 Variables Subscribed for by pObstacleMgr

The pObstacleMgr application will subscribe for the following four MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- GIVEN_OBSTACLE: One of the obstacle manager input options is to receive the obstacle information in the form of a convex polygon with a unique label. See Section 70.2.1.
- NAV_X: Ownship's current position in x coordinates.
- NAV_Y: Ownship's current position in y coordinates.
- OBM_ALERT_REQUEST: A message, typically from the obstacle avoidance behavior of the helm, to configure the criteria and format for posting obstacle manager alerts. See Section 70.3.3.
- TRACKED_FEATURE: One of the obstacle manager's input options is to received simulated LIDAR points. Each point is received as a message of this variable. See Section 70.2.1.

58.6 Terminal and AppCast Output

The pObstacleMgr application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 40 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. The counter on the end of line 2 is incremented on each iteration of pObstacleMgr, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

The output in the below example comes from the `s1_alpha_obstacles` mission.

Listing 58.9: Example terminal or appcast output for pObstacleMgr.

```
 1  ============================================================================
 2  pObstacleMgr alpha                                              0/0(238)
 3  ============================================================================
 4  Configuration (point handling):
 5    point_var:    TRACKED_FEATURE
 6    max_pts_per_cluster: 50
 7    max_age_per_point:   60
 8    ignore_range:        40
 9  Configuration (alerts):
10    alert_var:   OBSTACLE_ALERT
11    alert_name:  avoid_obstacle_
12    alert_range: 19
13  Configuration (lasso option):
14    lasso:        true
15    lasso_points: 8
```

```
16    lasso_radius: 6
17  ============================================
18  State:
19    Nav Position:      (61,-129.7)
20    Points Received:   58
21    Points Invalid:    0
22    Points Ignored:    147
23    Polygon obstacles: 4
24    Clusters:          4
25    Clusters released: 0
26
27  ObstacleKey  Points  HullSize  Updates
28  -----------  ------  --------  -------
29  b            14      8         n/a
30  c            16      8         55
31  d            13      8         23
32  e            11      8         46
```

The first group of lines (4-16) show the configuration settings for pObstacleMgr. The status of pObstacleMgr is shown in Lines 18-32.

58.7 Simple Example Missions

As of Release 19.8, there are two example missions using the obstacle manager.

- s1_alpha_obstaclemgr: A single vehicle mission with obstacles generate by a stream of points.
- m2_berta_obstacles: A two vehicle mission with obstacles given at fixed locations.

Differences between the obstacle manager and the contact manager:
- obstacles don't have type or group
- Only one global alert range for all obstacles. Set in the config block but may be overridden by a behavior when it registers wit the obstacle manager

59 pDeadManPost: Arranging Posts in the Absence of Events

59.1 Overview

The pDeadManPost application allows the user to arrange configured posts to the MOOSDB in the *absence* of another event. There are many conceivable uses for this, but here are a few of the ideas that motivated this app:

- On a shoreside community, a dead-man post can be made to trigger an alert when a deployed vehicle is out of contact after some period of time. The alert could be a posting to trigger an alarm .wav file or a spoken alert message through iSay.
- On a surface vehicle, a dead-man post can be made to put the vehicle in a station-keeping mode if comms to the shoreside command and control goes silent for some period of time.
- On an underwater vehicle, a dead-man post can be made to put the vehicle in a return-to-home mode if it loses updates from navigation beacons for some period of time.

59.2 Configuration Parameters for pDeadManPost

The following parameters are defined for pDeadManPost. A more detailed description is provided in other parts of this section. Parameters having default values are indicated.

Listing 59.1: Configuration Parameters for pDeadManPost.

heartbeat_var:	Names the MOOS variable serving as the heartbeat. Section 59.4.
active_at_start:	if true then the hearbeat condition (timer) is active upon startup of the app. Otherwise the heartbeat condition is not active until the first heartbeat is received. The default is heartbeat true. Section 59.4.
deadflag:	A MOOS variable-value pair to be posted when the heartbeat disappears. Section 59.5.
max_noheart:	The maximum amount of time, in seconds, a missing heartbeat is tolerated before the deadflags are posted. Default is 10. Section 59.4.
post_policy:	Either "once", "repeat", or "reset" determines how the deadflags are posted once a heartbeat condition has failed. Section 59.5.

59.3 Publications and Subscriptions of pDeadManPost

The interface for pDeadManPost, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pDeadManPost --interface or -i
```

59.3.1 Variables Published by pDeadManPost

The only output of pDeadManPost is:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted

only after an appcast request is received from an appcast viewing utility. Section 59.6.

The pDeadManPost app will also publish whatever MOOS variables are identified in the deadflag parameter.

59.3.2 Variables Subscribed for by pDeadManPost

The pDeadManPost application will subscribe for the following four MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration. See the documentation on appcasting.

It will also subscribe for whatever MOOS variable is identified as serving as the heartbeat variable in the heartbeat_var parameter.

59.4 Configuring the Heartbeat Condition

The *hearbeat condition* stipulates that a named MOOS variable must be continually written to by some other application, with a specified frequency. If the heartbeat condition fails, deadflags are posted. The heartbeat is monitored simply by monitory the mailbox in pDeadManPost for incoming mail on the named MOOS variable. No check is made as to the value or source of the incoming mail, only that mail has been received. The monitored variable is set by heartbeat_var parameter. For example:

```
heartbeat_var = CONTINUE_MISSION
```

The frequency at which the heartbeat must be received is determined by max_noheart parameter, given in seconds. The default for this parameter is 10 seconds. It can be set, for example, with:

```
max_noheart = 120
```

The user may further specify whether the heartbeat condition is active immediately upon startup of the pDeadManPost app, or if it instead becomes active only after receiving the first heartbeat message. This is determined with the active_on_start parameter. For example:

```
active_on_start = false     // The default is true
```

The status of the heartbeat condition can be monitored in the appcasting output of the app, by noting the line "Time Remaining:". If this is zero, then the heartbeat condition has failed. If it reads "N/A", it means the heartbeat condition is not in effect, likely because the app is configured to be not active on start and no heartbeat message has yet been received. Otherwise, it should show the time remaining in seconds before the heartbeat condition fails unless another heartbeat message is received in the meanwhile.

59.5 Specifying DeadFlags

A deadflag is a MOOS variable-value pair to be posted when or if the heartbeat condition times out. Multiple dead flags may used. An example:

```
deadflag = RETURN=true
deadflag = DEAD_MAN_POST_INTERRUPT=true
```

When the heartbeat condition fails, and one or more deadman posts are made, by default, these posts are made only `once`, regardless of whether further heartbeat messages are received. The user has two further options: the posts can be made to `repeat` on each iteration of the pDeadManPost app by setting:

```
post_policy = repeat    // The default is "once"
```

The second option is to configure pDeadManPost to `reset` after a posting is made. In this case the heartbeat timer is reset and if another period of time elapses without a heartbeat, another posting is made. This is configured with:

```
post_policy = reset    // The default is "once"
```

When the post_policy parameter is set to `repeat`, the postings will cease when a new heartbeat message is received.

59.6 Terminal and AppCast Output

The pDeadManPost application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 2 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. See the documentation on uMac or uMacView for more on appcasting and viewing appcasts. The counter on the end of line 2 is incremented on each iteration of pDeadManPost, and serves a bit as a heartbeat indicator. The `"0/0"` also on line 2 indicates there are no configuration or run warnings detected.

The output in the below example comes from the example described in Section 59.7.

Listing 59.2: Example terminal or appcast output for pDeadManPost.

```
1  ============================================================
2  pDeadManPost alpha                                   0/0(362)
3  ============================================================
4  Configuration:
5  ------------------------------------
6        heart_var: CONTINUE
7     max_noheart: 60
```

```
 8  active_at_start: false
 9      post_policy: repeat
10        deadflags: (1)
11                   [1] RETURN=true
12
13  State:
14  ------------------------------------
15    Elapsed Time: 23.0875
16  Time Remaining: 36.9125
17      Heartbeats: 2
18  Total Postings: 0
```

The first few lines (4-11) show the configuration settings for pDeadManPost. The status of pDeadManPost is shown in Lines 13-18. In this case, the most recent heartbeat message was received 23 seconds prior, as shown in line 15, and no deadman posts have been made as shown on line 18.

59.7 A Simple Example

The s1_alpha_deadman example mission distributed with moos-ivp provides a simple working example. More explanation to come...

60 pSearchGrid: Using a 2D Grid Model for Track History

60.1 Overview

The pSearchGrid application is a module for storing a history of vehicle positions in a 2D grid defined over a region of operation. This module may have utility as-is, to help guide a vehicle to complete or uniform coverage of a given area, but also exists as an example of how to use and visualize the XYConvexGrid data structure. This data structure may be used in similar modules to store a wide variety of user specified data for later use by other modules or simply for visualization. Here the structure is used in a MOOS application, but it could also be used within a behavior. The pSearchGrid module begins with a user-defined grid, defined in the MOOS configuration file. As a vehicle moves and generates node reports, pSearchGrid simply notes the vehicle's current position increments the cell containing vehicle's current position. After time, the grid shows a cumulative history of the most commonly traveled positions in the local operating area.

The pSearchGrid module is an optional part of the Charlie example mission discussed later in this section. When running and grid viewing is enabled in pMarineViewer, the viewer may show something similar to Figure 194.

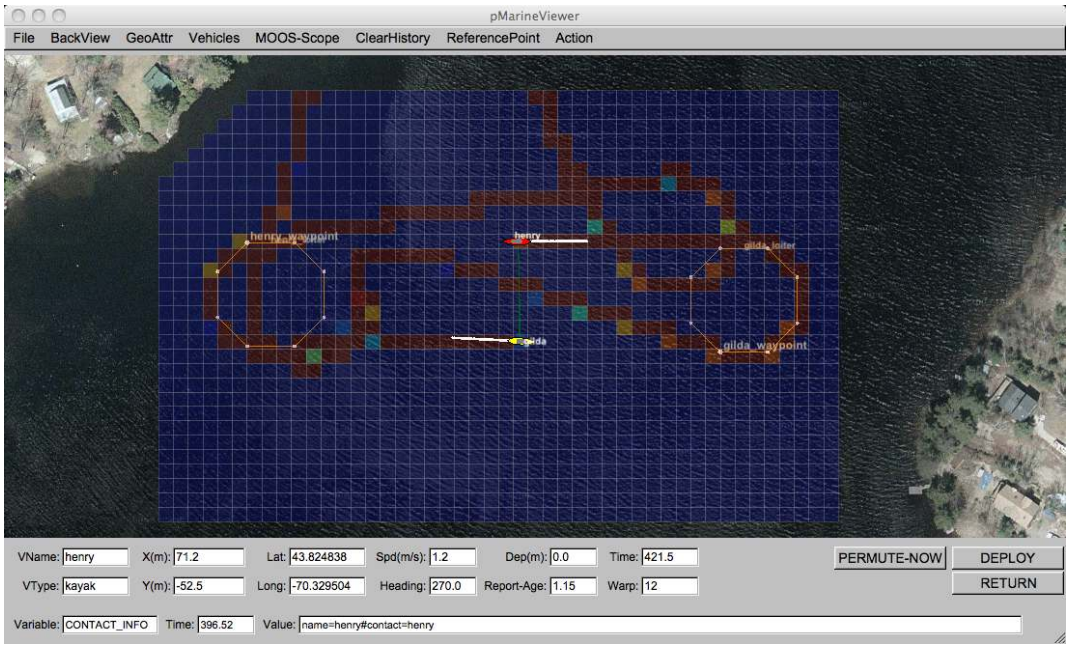

Figure 194: **An Example pSearchGrid Scenario:** A grid is configured around the operation area of the Berta example mission. Higher values in a given grid cell represents a longer noted time of any vehicle passing through that cell.

60.2 Using pSearchGrid

The present configuration of pSearchGrid will store values in its grid cells proportional to the time a vehicle is noted to be within a grid cell. One may use this application as written or regard this as

a template for writing a new application that stores some other value with each grid cell, such as bathymetry data, water velocity data, or likelihood of there being an object of interest in the region of the cell for further investigation. The below discusses the general usage of managing the grid data within a MOOS application.

60.2.1 Basic Configuration of Grid Cells

Basic grid configuration consists of specifying (a) a convex polygon, and (b) the size of the grid squares. From this the construction of a grid proceeds by calculating a bounding rectangle containing the polygon, generating a set of squares covering the rectangle, and then removing the squares not intersecting the original polygon. The result is a non-rectilinear grid as shown in Figure 194. The basic configuration of grid cells is done with the `grid_config` parameter specifying the points and the cell size as the following example:

```
grid_config = pts={-50,-40:-10,0:180,0:180,-150:-50,-150}, cell_size=5
```

60.2.2 Cell Variables

The grid is primary used for associating information with each grid cell. In `pSearchGrid`, the grid is configured to store a single numerical value (a C++ double) with each cell. The grid structure may be configured in another application to store multiple numerical values with each cell. In `pSearchGrid`, the cell variables are declared upon startup in the MOOS configuration block, similar to:

```
grid_config = cell_vars=x:0y:100
grid_config = cell_min=x:0
grid_config = cell_min=x:100
```

This configuration associates two cell variables, x and y, which each cell. The cell variable x is initialized to 0 for each cell, and the cell variable y is initialized to 100. The first variable has a minimum and maximum constraint of 0 and 100, and the second variable is unconstrained. The above configuration could also have been made on one line, separating each with a comma.

60.2.3 Serializing and De-serializing the Grid Structure

The grid structure maintained by `pSearchGrid` is periodically published to the MOOSDB for consumption by other applications, or conceivably by a behavior with the IvP Helm. The structure may be serialized into a string by calling the `get_spec()` method on an instance of the `XYConvexGrid` class. Serializing a grid instance and posting it to the MOOSDB may look something like:

```
#include "XYConvexGrid.h"
....
XYConvexGrid mygrid;
....
string str = mygrid.get_spec();
m_Comms.Notify(VIEW_GRID, str);
```

Likewise a string representation of the grid may be de-serialized to a grid instance by calling a function provided in the same library where the grid is defined. De-serializing a string read from the MOOSDB into a local grid instance may look something like:

```
#include "XYFormatUtilsConvexGrid.h"
#include "XYConvexGrid.h"
....
string grid_string_spec;
....
XYConvexGrid mygrid = string2ConvexGrid(grid_string_spec);
```

The same function used to de-serialize a string is used by pSearchGrid to configure the initial grid. In other words, the components from the various GRID configuration parameters are concatenated into a single comma-separated string and passed to the de-serialization function, string2ConvexGrid, to form the initial instance used by pSearchGrid.

60.2.4 Resetting the Grid

The grid may be reset at any point in the operation when pSearchGrid receives mail on the variable PSG_GRID_RESET. If this variable's string value is the empty string, it will reset all cell variables for all cell elements to their given initial values. If the string value is non-empty, it will interpret this as an attempt to reset the values for a named cell variable. Thus PSG_GRID_RESET="x" will reset the x cell variable and no other cell variables. If "x" is not a cell variable, no action will be taken.

60.2.5 Viewing Grids in pMarineViewer

The pMarineViewer will display grids by subscribing for postings to the variable VIEW_GRID. As with other viewable objects such as polygons, points, etc., the viewer keeps a local cache of grid instances, one for each named grid, based on the grid label. For example if two successive grids are received with different labels, the viewer will store and render both of them. If they have the same label, the second will replace the first and the viewer will just render the one.

The rendering of grids may be toggled on/off by hitting the 'g' key, and may be made more transparent with the CTRL-g key, and less transparent with the ALT-g key. The color map used for the grid is taken from the typical MATLAB color map; red is the highest value, blue is the lowest.

60.2.6 Examples

Example usage of pSearchGrid may be found in both the Charlie and Berta example missions distributed with the moos-ivp tree. For example, in the Charlie example mission, edit charlie.moos and uncomment the line beginning with Run = pSearchGrid. Likewise for the Berta mission and the file meta_shoreside.moos.

60.3 Configuration Parameters of pSearchGrid

The following parameters are defined for pSearchGrid. A more detailed description is provided in other parts of this section. Parameters having default values indicate so.

Listing 60.3: Configuration Parameters for pSearchGrid.

grid_config: A portion or all of a grid configuration description. See Listing 4.

60.3.1 An Example MOOS Configuration Block

An example pSearchGrid configuration block is given in Listing 4 below, and may also be seen from the command line invocation of:

```
$ pSearchGrid --example or -e
```

Listing 60.4: Example configuration of the pSearchGrid application.

```
1  ============================================================
2  pSearchGrid Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pSearchGrid
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   GRID_CONFIG = pts={-50,-40: -10,0: 180,0: 180,-150: -50,-150}
11   GRID_CONFIG = cell_size=5
12   GRID_CONFIG = cell_vars=x:0:y:0:z:0
13   GRID_CONFIG = cell_min=x:0
14   GRID_CONFIG = cell_max=x:10
15   GRID_CONFIG = cell_min=y:0
16   GRID_CONFIG = cell_max=y:1000
17 }
```

60.4 Publications and Subscriptions for pSearchGrid

The interface for pSearchGrid, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pSearchGrid --interface or -i
```

60.4.1 Variables Published by pSearchGrid

The pSearchGrid application publishes the following variables:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- VIEW_GRID: A full description of a grid format and contents.

A typical string may be:

```
VIEW_GRID = pts={-50,-40:-10,0:100,0:100,-100:50,-150:-50,-100},cell_size=5,
            cell_vars=x:0:y:0:z:0,cell_min=x:0,cell_max=x:50,cell=211:x:50,
            cell=212:x:50,cell=237:x:50,cell=238:x:50,label=psg
```

60.4.2 Variables Subscribed for by pSearchGrid

The pSearchGrid application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- NODE_REPORT: A node report for a given vehicle from pNodeReporter.
- NODE_REPORT_LOCAL: A node report for a given vehicle from pNodeReporter.
- PSG_GRID_RESET: A request to reset the grid to its original configuration.

60.4.3 Command Line Usage of pSearchGrid

The pSearchGrid application is typically launched with pAntler, along with a group of other modules. However, it may be launched separately from the command line. The command line options may be shown by typing "pSearchGrid --help":

Listing 60.5: Command line usage for the pSearchGrid tool.

```
Usage: pSearchGrid file.moos [OPTIONS]

Options:
  --alias=<ProcessName>
      Launch pSearchGrid with the given process
      name rather than pSearchGrid.
  --example, -e
      Display example MOOS configuration block
  --help, -h
      Display this help message.
  --version,-v
      Display the release version of pSearchGrid.
Note: If argv[2] is not of one of the above formats
      this will be interpreted as a run alias. This
      is to support pAntler launching conventions.
```

61 uFldObstacleSim: Simulating Obstacles

61.1 Overview

The uFldObstacleSim application is a tool for simulating obstacles, and obstacle sensor output. Ground truth position and size of obstacles are read from a pre-generated *obstacle file*. The simulation generates information in one of two modes, in one of two manners:

- Obstacle polygons are simply published and shared to all vehicles, or
- Simulated sensor data, similar to Lidar points on the obstacle, are published and shared to the vehicles. Downstream apps are then left with the job of inferring the obstacle from the sensor data.

When this simulator is operating in the first mode, it publishes information to the variable GIVEN_OBSTACLE. In the second mode, it publishes the information to the variable TRACKED_FEATURE.

The simulator also supports two different modes of obstacle generation:

- By default, the obstacles are simply read in from the obstacle file and never change.
- A second mode is supported where the obstacles are periodically regenerated with the same configuration parameters used for creating the original obstacle file.

A key parameter in regeneration is the obstacle region, a polygon within which all randomly generated obstacles are guaranteed to reside within. During a simulation, we want to take care that obstacle regeneration does not take place while any vehicles are in the obstacle region. Otherwise an obstacle could be generated in a position currently occupied by, or just in front of a vehicle. The ensuing unavoidable collision would skew the test results. The obstacle simulator therefore will monitor vehicle positions and only regenerate obstacles when all vehicles are a safe distance from the obstacle region. The vehicle locations are known to the simulator from the NODE_REPORT messages received from the vehicles.

The setup overview is depicted in Figure 195:

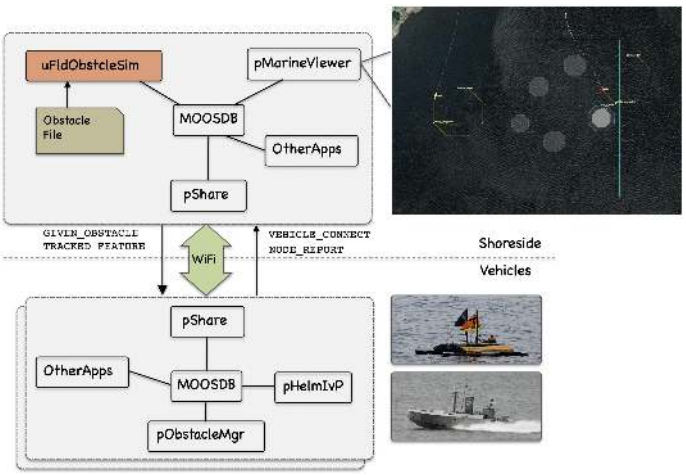

Figure 195: **The Obstacle Simulator:** The obstacle simulator resides on the shoreside and ingests and maintains ground truth obstacle state. Information is fed to each connected vehicle. Typically the primary consumer on the vehicle side is the obstacle manager, pObstacleMgr.

61.2 A Quick Start Guide to Using uFldObstacleSim

To get started, we (a) point you to an example mission using uFldObstacleSim, (b) lay out the minimum uFldObstacleSim configuration components, i.e., those which do not have default values, and (c) discuss the structure of a simple obstacle file representing the ground-truth set of obstacles used by the simulator.

61.2.1 A Working Example Mission - the Bo Alpha Mission

The example mission is referred to as the Bo Alpha example mission and may be found and launched in the moos-ivp distribution with:

```
$ cd moos-ivp/ivp/missions/m34_bo_alpha
$ ./launch.sh 10
```

This launches the simulation with time warp 10. The time warp may be adjusted to suit your preference and is bounded above by your computer's processing capability. After launching and hitting the deploy button, you should see something similar to Figure 196.

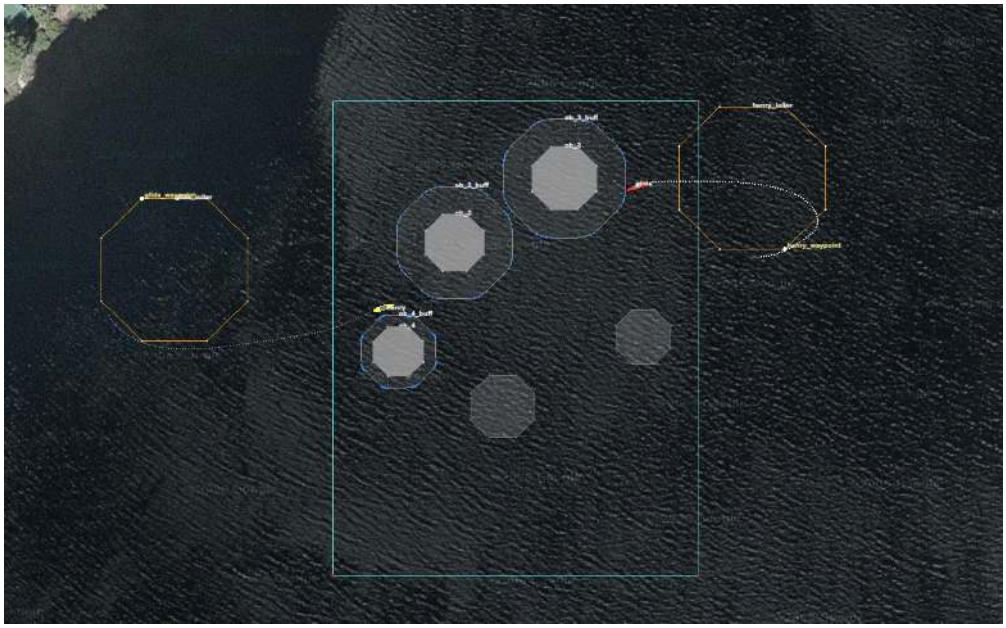

Figure 196: **The Bo Alpha Mission:** The Bo Alpha example mission involves a two vehicles each traversing East-West through an obstacle field created by the simulator. The helm on both vehicles is performing obstacle avoidance and COLREGS based collision avoidance with the other vehicle. The true obstacle is the inner polygon. When the vehicle is actively avoiding the obstacle, the true obstacle is rendered more opaque, and the buffer region around the obstacle is rendered.
Video:(1:01): https://vimeo.com/424523746

There are a few notable components of this simulation that comprise the nature of the mission:

- The uFldObstacleSim: Running on the shoreside, distributes knowledge of the obstacles to each vehicle.
- The `obstacles.txt` file read in by the uFldObstacleSim obstacle simulator.
- The gen_obstacles command-line app that generated the `obstacles.txt` file, used by the simulator. This app chooses random obstacle locations within a polygon and ensures a given minimum spacing between obstacles.
- pObstacleMgr app running on each vehicle that receives the obstacle knowledge from the obstacle simulator and maintains a set of known obstacles. It will also generate obstacle alerts which result in spawned obstacle avoidance behaviors in the helm.

It may also be worth considering, in Figure 196, where all the visual artifacts shown in the pMarineViewer snapshot are coming from:

- The uFldObstacleSim app is publishing the large square rectangle, which is the region from which the random obstacles were originally chosen from. It is also publishing all five ground-truth obstacles. The three north-west obstacles have other things being rendered over them, but the two south-east obstacles are the ones published by uFldObstacleSim.
- The Avoid Obstacle behavior in the helm also publishes the obstacle polygon when the behavior is active, but it publishes a request to render the polygon with more opacity. Thus the three

north-west polygons show that at least one of the vehicles is actively avoiding it. The behavior also published the buffer polygon around the ground truth obstacle, with less opacity. And of course the behavior is also publishing the loiter polygon to the far east and west to which each is transiting through the obstacle field.

- Not shown in the figure, but when the two vehicles are close to each other and also actively engaging in collision avoidance, each vehicle's collision avoidance behavior will render a line between vehicles, with color varying with the behavior's priority weight.

61.2.2 A Bare-Bones Example uFldObstacleSim Configuration

Listing 6 below shows a bare-bones configuration. Line 3 names a obstacle file, containing the ground truth description of objects in the field. This format is described in Section 61.2.3, and an example obstacle file is in the same directory as the Bo Alpha example mission. There are several more configuration parameters supported by uFldObstacleSim, but they all have default values, accepted in this simple simulation. The other parameter values are discussed later.

Listing 61.6: Example bare-bones configuration of uFldObstacleSim*.*

```
1  ProcessConfig = uFldObstacleSim
2  {
3    obstacle_file = obstacles.txt
4  }
```

61.2.3 A Simple Obstacle File

In the Bo Alpha example mission, two vehicles traverse an obstacle field comprised of five obstacles. These obstacles were chosen randomly by the gen_obstacles command line utility. The results were stored in the `obstacles.txt` file read in by the obstacle simulator. This file was named in the obstacle simulator config block in Listing 6 above. The gen_obstacles utility will choose obstacles while ensuring (a) each obstacle is within a given polygon area, and (b) a minimum configurable separation distance between obstacles. This command-line tool is discussed in Section 61.2.4.

An obstacle file is comprised of:

- The first line, a comment, showing the exact command that generated the file,
- The command line parameters on a separate line,
- The polygon for each obstacle, with a unique label.

The obstacle file for the Bo Alpha mission is given below.

```
# gen_obstacles --poly=30,-20:30,-140:120,-140:120,-20 --amt=5 --min_size=1 --max_size=6 \
   --min_range=20 --meter
region    = pts={30,-20:30,-140:120,-140:120,-20}
min_range = 20
min_size  = 6
max_size  = 10
poly = pts={107,-101:112,-106:112,-113:107,-118:100,-118:95,-113:95,-106:100,-101},label=ob_0
poly = pts={50,-30:53,-33:53,-38:50,-42:45,-42:41,-38:41,-33:45,-30},label=ob_1
poly = pts={82,-65:87,-70:87,-76:82,-81:76,-81:71,-76:71,-70:76,-65},label=ob_2
poly = pts={59,-101:64,-105:64,-112:59,-116:53,-116:48,-112:48,-105:53,-101},label=ob_3
poly = pts={107,-38:112,-43:112,-49:107,-54:100,-54:96,-49:96,-43:100,-38},label=ob_4
```

61.2.4 Generating an Obstacle File

The gen_obstacles command line tool will generate a list of randomly generated obstacle polygons, given a polygon region. The tool will guarantee that the obstacle will completely reside in the region, and will guarantee a minimal separation between obstacles.

For example, the below command will generate the obstacle file shown in the above section, with the initial five obstacles of the Bo Alpha mission. The `--poly` parameter indicates the obstacle *region*. This region must be convex, and the utility will produces as many obstacles as possible up to the amount specified with the `--amt` parameter.

```
$ gen_obstacles --poly=30,-20:30,-140:120,-140:120,-20 --amt=5
  --min_size=1 --max_size=6 --min_range=10 --meter
```

Each obstacle is an octagon with a random radius no smaller than that given by the `--min_size` parameter, and no greater than that given by the `--max_size` parameter. The obstacles will be placed with a guaranteed minimum separation to the other obstacles given by the `--min_range` parameter. Finally, the `--meter` parameter, will round all obstacle vertices to the nearest meter if desired, instead of the default value of `0.1` meters.

61.3 Dynamic Resetting of Ground Truth Obstacles

The obstacle manager supports an option to periodically reset the ground truth size and location of obstacles. This allows simulations to test against a wider variety of obstacle situations without the need to re-start the simulation. The dynamic reset capability ensures that the reset only occurs only when all known vehicles are outside the obstacle field, to avoid generating a new obstacle too close to (or on) a vehicle and thus creating an unavoidable collision.

61.3.1 Parameters for Enabling Dynamic Resetting

The dynamic reset occurs on a schedule. After an elapsed period of time, given by the reset_interval parameter, a reset will occur as soon as all the vehicles are outside the obstacle region. All vehicles must be not only outside the obstacle region but also a given distance away from the obstacle region given by the reset_range parameter. Of course, if all vehicles never happen to be outside the obstacle region simultaneously, then the reset will never occur. The mission must be otherwise constructed, as with the Bo Alpha mission, to ensure the vehicles periodically all exit the obstacle

region simultaneously.

Here is an example setting:

```
reset_interval = 300
reset_range    = 20
```

The reset_interval units are in seconds, and the reset_range units are in meters. The default value for reset_interval is -1, indicating that resets are disabled. The default value for reset_range is 10 meters.

61.3.2 MOOS Variable for Enabling Dynamic Resetting

If you would like to reset the obstacle field on demand, by poking a MOOS variable, this can be done. By publishing UFOS_RESET = now, the obstacle simulator will schedule an obstacle reset to occur as soon as possible. As discussed above, the reset will only actually occur as soon as all vehicles have exited the obstacle region, sufficiently far away from the obstacle region given by the reset_range parameter.

61.3.3 Enabling Dynamic Resetting in the Bo Alpha Mission

Dynamic obstacle resetting is not enabled by default in the Bo Alpha mission. It can be enabled by launching with the -d option on the command line:

```
$ cd moos-ivp/ivp/missions/m34_bo_alpha
$ ./launch.sh 10 -d
```

This additional flag is supported in the launch.sh script to add the following two lines to the uFldObstacleSim configuration block:

```
reset_interval = 100
reset_range    = 10
```

Every 100 seconds, when the two vehicles are at least 10 meters outside the obstacle region, the obstacles will be reset. Figure 197 shows two such random obstacle configurations, and the associated video link shows the dynamic mission in action.

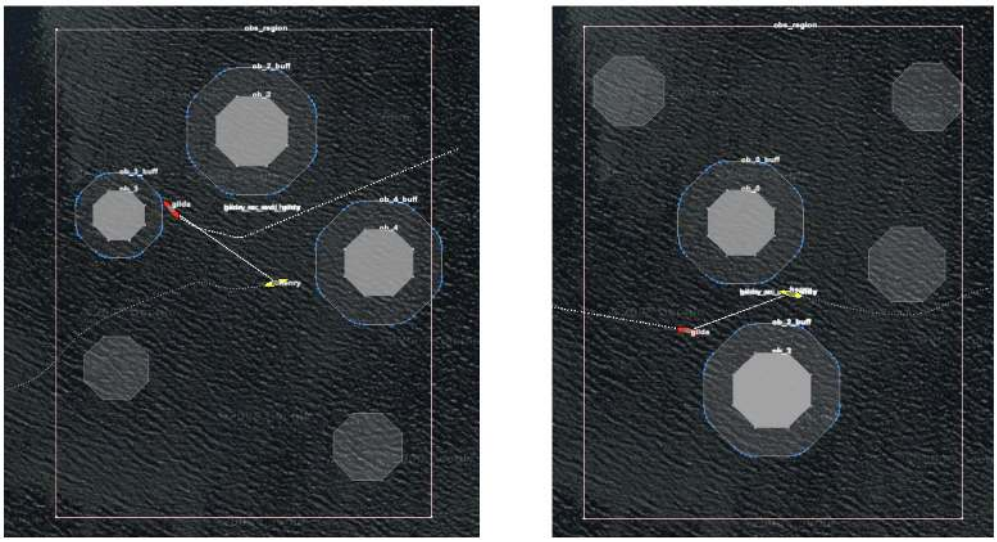

Figure 197: **The Dynamic Bo Alpha Mission:** The Bo Alpha is launched with dynamic obstacle reset enabled. Roughly each time the vehicles reach their east west loiter positions, the obstacle simulator reset the obstacles. This produces a different challenge each time the vehicles traverse through the obstacle field.
Video:(0:54): https://vimeo.com/425156397

61.4 Simulating Sensor Data from Ground Truth Obstacles

The obstacle simulator supports a second mode of distributing obstacle information to vehicles. In the default mode discussed so far, the ground truth obstacle position and location are sent directly to the vehicles in the GIVEN_OBSTACLE MOOS variable. For the vehicle there is no guesswork, and obstacle avoidance is conducted with perfect knowledge of the obstacles.

In the second mode, the *points* mode, the obstacle simulator generates a steady stream of random points inside the ground truth obstacles. It then sends these points to the vehicle in the MOOS variable TRACKED_FEATURE. Here is an example publication:

```
TRACKED_FEATURE = x=100,y=-49,key=ob_4
```

The message contains both the point location, and a unique ID associated with the ground truth obstacle. So this simulatad sensor data is still pretty artificially simplistic - there are no false points because all points generated by the simulator do reside within the ground truth obstacle polygon. And by including a key, clustering of points is already done and perfect.

61.4.1 Enabling the Points Sensor Data Mode

The *points* mode is disabled by default. It is enabled by setting the post_points configuration parameter to `true`:

```
post_points = true
rate_points = 5
```

The rate_points parameter sets the number of points generated, per obstacle, per iteration of the simulator. The default value is 5. When the points mode is enabled, the obstacle simulator does

not post the ground truth obstacle information to GIVEN_OBSTACLE. However, it is still posted to KNOWN_OBSTACLE. Typically uField sharing is configured to *not* share KNOWN_OBSTACLE to the vehicles. However it may still be useful for other apps in the shoreside community to have access to ground truth obstacle information. For example the uFldCollObDetect app runs in the shoreside and monitors vehicles for collisions with obstacles, so it needs access to ground truth obstacle information.

61.4.2 Generation of Simulated Sensor Points

Simulated sensor points are generated by calculating random line-of-sight points on the edge of the obstacle polygon from the direction of the vehicle. Sensor points are generated per vehicle and shared only with the relevant vehicle as in Figure 198.

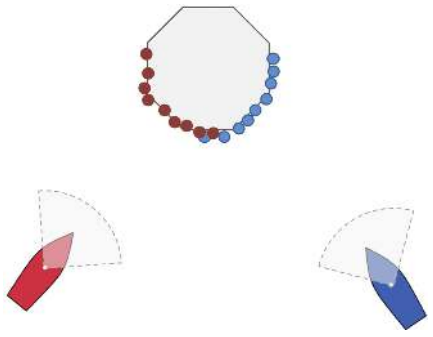

Figure 198: **Vehicle-centered line-of-site sensor:** Sensor points are based on line of site, with different sensor points generated for each vehicle depending on the bearing of the obstacle to the vehicle. This is not currently implemented in the uFldObstacleSim app.

Recall that each point corresponds to two publications. One publication, to the variable TRACKED_FEATURE_ABE, is generated and shared only to the vehicle abe. The other publication is to the variable VIEW_POINT which published locally on the shoreside for the benefit of rendering in a GUI app such as pMarineViewer. The color of the points is based on the color of the vehicle, derived from NODE_REPORT messsages received on the shoreside from each vehicle.

In the event that uFldObstacleSim is run in a headless simulation, i.e., no need for visuals, the publication of VIEW_POINT can be disabled with the configuration post_visuals=false.

61.5 Obstacle Expiration

There are three notions or reasons why we may want to consider the *expiration* of obstacles.

- The robot or vehicle has moved far away from the obstacle and we no longer care about it, or
- The obstacle never existed to begin with, but perhaps briefly it appeared to exist due to sensor noise, or
- The obstacle is still close and is real, but perhaps a new identifier was mistakenly created for the same obstacle.

Each of these things happen in practice, and the downstream apps that manage and reason about

obstacles need to deal with these issues. So the obstacle simulator has the capability to replicate the expiration of an obstacle to enable testing of the downstream apps.

The expiration policy of the two primary obstacle modes of the simulator are discussed here:

- The *points* mode, where the simulator publishes sensor points in the form of the `TRACKED_FEATURE` variable, and
- The *ground-truth* mode where the simulator publishes the ground truth obstacles in the form of the `GIVEN_OBSTACLE` variable.

61.5.1 Case 1 - Expiration of Sensor Points

The simplest of the two modes is the *points* mode, perhaps because elements in a sensor data stream are usually considered to be ephemeral - as things change, so does the data. In the *points* mode, data is published to the `TRACKED_FEATURE` variable. These postings are regarded as similar to LIDAR points. Although the simulator generates them randomly on or in the obstacle, they at some point will either cease, or evolve position, due to:

- The obstacle goes out of range from the vehicle
- The obstacle moves or drifts, and thus so do the points
- The obstacle stops producing points because the simulator deletes the obstacle
- The obstacle stops producing points because in the real world the obstacle actually didn't exist but was perhaps produced by a wave or some other noise

Figure 199 conveys the lifespan of sensor points on an obstacle as a vehicle approaches, passes and leaves behind an obstacle.

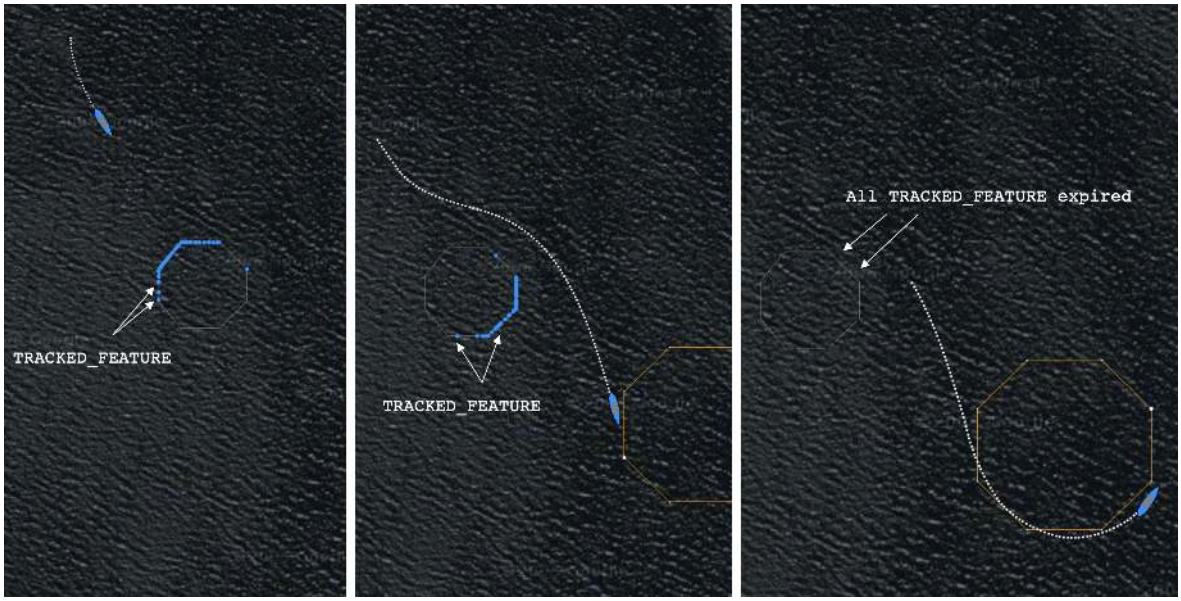

Figure 199: **Expiration of Sensor Points:** Sensor points are generated on the edge of the obstacle as the vehicle comes in range (left). As long as the vehicle remains in range, regardless of the relative bearing from the vehicle to the obstacle, sensor points on the line of site to the obstacle will continue to be generated (middle). As the vehicle open range to the obstacle the sensor points will cease to be generated (right).

61.5.2 Case 2 - Expiration of Ground Truth Obstacles

In *ground-truth* mode, where the simulator is publishing the actual obstacle polygon rather than simulated sensor points, the simulator publishes the polygon information in the form of:

- VIEW_POLYGON: for consumption by a GUI app like pMarineViewer.
- KNOWN_OBSTACLE: for consumption by other shoreside apps that need access to ground truth obstacles such as uFldCollObDetect.
- GIVEN_OBSTACLE: for sharing to the vehicles and consumption by the obstacle manager pObstacleMgr.

In each of these cases, there is a need to "forget" about an obstacle after time. So each obstacle message contains a `duration` field, in seconds. While the obstacle is relevant, the obstacle simulator will periodically publish to these variables, each time with same duration. The consumers presumably reset their duration clocks each time a new message is received. The obstacle simulator will publish only periodically, but often enough such that the obstacle will endure indefinitely if the periodic refresh publications continue.

The refresh_interval determines how often uFldObstacleSim re-publishes the current ground-truth obstacles. A duration is associated with each obstacle, so typically the refresh should occur before an obstacle expires (if the goal is persistence). The duration is set with two parameters, min_duration and max_duration. A random duration will be chosen between these two values.

The default settings for refresh_interval, min_duration, and max_duration are all -1. When refresh_interval is not set, the obstacle simulator will never refresh (republish) the obstacle

postings unless one or more vertices change. When the obstacle duration is not set, they we never expire in any of the consumer apps unless they are cleared or erased via other methods.

61.6 Configuration Parameters of uFldObstacleSim

The following parameters are defined for uFldObstacleSim. For some parameters, more detailed description are provided in other sections. Parameters having default values are indicated so.

Listing 61.7: Configuration Parameters for uFldObstacleSim.

obstacle_file:	A file with obstacle position and size location. Sections 61.2.3 and 61.2.4.
poly_vert_color:	Color of obstacle polygon vertices. The default is "gray50".
poly_edge_color:	Color of obstacle polygon edges. The default is "gray50".
poly_fill_color:	Color of obstacle polygon interior. The default is "white".
poly_label_color:	Color of obstacle polygon labels. The default is "invisible".
poly_vert_size:	Size of rendered obstacle polygon vertices. The default is 1.
poly_edge_size:	Size of obstacle polygon edges. The default is 1.
poly_transparency:	Transparency of rendered obstacle polygons. The default is 0.15.
draw_region:	If true, draw the obstacle region. The default is true.
region_edge_color:	Color of obstacle polygon edges. The default is "gray50".
region_edge_color:	Color of obstacle region edges. The default is "white".
post_points:	If true, sensor points are generated rather than ground truth obstacle polygons. The value false. Section 61.4.1.
rate_points:	When post_points is true, this parameter sets the rate of point generation. The default is 5 points, per obstacle, per iteration. Section 61.4.1.
min_duration:	If non-negative, set a random duration for each obstacle no lower than this value. The default is -1. Section 61.5.2.
post_visuals:	If true, visual posts to VIEW_POINT and VIEW_POLYGON are generated. The default is true. Section 61.4.2.
refresh_interval:	If non-negative, publications to GIVEN_OBSTACLE will be reposted evern N seconds where N is the value of this parameter. The default is -1. Section 61.5.2.
reset_interval:	Time, in seconds, between automatic reseting of obstacle locations. The default is -1, indicating disabled resetting. Section 61.3.1.
reset_range:	Distance, in meters, that all vehicles need to be outside the obstacle region in order for an obstacle reset to be allowed. The default is 10 meters. Section 61.3.1.
reuse_ids:	If false, each time the obstacle field is reset, a unique set of obstacle IDs, i.e., labels, will be generated for the newly generated obstacles. The default is true. Section 61.3.1.
sensor_range:	The range to an obstacle at which simulated LIDAR point will be generated. The default is 50. Section 61.4.2.

61.6.1 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldObstacleSim --example or -e
```

This will show the output shown in Listing 8 below.

Listing 61.8: Example configuration for uFldObstacleSim.

```
 1  ============================================================
 2  uFldObstacleSim Example MOOS Configuration
 3  ============================================================
 4
 5  ProcessConfig = uFldObstacleSim
 6  {
 7    AppTick   = 4
 8    CommsTick = 4
 9
10    obstacle_file    = obstacles.txt
11    poly_vert_color  = color    (default is gray50)
12    poly_edge_color  = color    (default is gray50)
13    poly_fill_color  = color    (default is white)
14    poly_label_color = color    (default is invisible)
15
16    poly_vert_size    = 1       (default is 1)
17    poly_edge_size    = 1       (default is 1)
18    poly_transparency = 0.15    (default is 0.15)
19
20    region_edge_color = color   (default is gray50)
21    region_vert_color = color   (default is white )
22
23   draw_region       = true    (default is true)
24   region_edge_color = color   (default is gray50)
25   region_vert_color = color   (default is white)
26
27   post_points      = true     (default is false)
28   rate_points      = 5        (default is 5)
29   point_size       = 5        (default is 2)
30
31   min_duration     = 10       (default is -1)
32   max_duration     = 15       (default is -1)
33   refresh_interval = 8        (default is -1)
34
35   reset_interval   = -1       (default is -1)
36   reset_range      = 10       (default is 10)
37
38   reuse_ids        = true     (default is true)
39   sensor_range     = 50       (default is 50)
40
41   app_logging  = true  // {true or file} By default disabled
42
43   post_visuals = true  // {true or false} By default true
44 }
```

61.7 Publications and Subscriptions for uFldObstacleSim

The interface for uFldObstacleSim, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldObstacleSim --interface or -i
```

61.7.1 Variables Published by uFldObstacleSim

The primary output of uFldObstcleSim to the MOOSDB is posting of sensor reports, visual cues for the sensor reports, and visual cues for the hazard objects themselves.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 61.8
- GIVEN_OBSTACLE: A ground truth obstacle, same as, KNOWN_OBSTACLE, but published separately. Typically this variable will be shared to the the vehicles.
- KNOWN_OBSTACLE: A ground truth obstacle, same as, GIVEN_OBSTACLE, but published separately. Typically shoreside apps will register for this variable for knowledge of ground truth obstacles.
- TRACKED_FEATURE: A single point related to a sensor measurement related to a ground truth obstacle, and key associated with that obstacle.
- VIEW_POLYGON: A visual artifact for rendering a a ground truth obstacle polygon.

Example postings:

```
KNOWN_OBSTACLE  = pts={48,-77:52,-80:52,-86:48,-89:43,-89:39,-86:39,-80:43,-77},  \
                  label=ob_4,duration=5
GIVEN_OBSTACLE  = pts={48,-77:52,-80:52,-86:48,-89:43,-89:39,-86:39,-80:43,-77},  \
                  label=ob_4,duration=5
TRACKED_FEATURE = x=100,y=-49,key=ob_4
VIEW_POLYGON    = pts={48,-77:52,-80:52,-86:48,-89:43,-89:39,-86:39,-80:43,-77},  \
                  label=ob_4,label_color=invisible,edge_color=gray50,vertex_color=gray50, \
                  fill_color=white,vertex_size=1,edge_size=1,fill_transparency=0.15
```

61.7.2 Variables Subscribed for by uFldObstcleSim

The uFldObstcleSim application will subscribe for the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- VEHICLE_CONNECT: A message from the vehicle indicating its presence in the simulation. This will prompt a (re)publication and sharing of ground truth obstacles to the vehicle.
- UFOS_RESET: A request to the simulator to reset the obstacle field immediately, or as soon as all vehicles have safely cleared the obstacle field. 61.3.2.
- NODE_REPORT: A report on a vehicle location and status.

Example postings:

```
UFOS_RESET = true
VEHICLE_CONNECT = true
```

61.7.3 Command Line Usage of uFldObstcleSim

The uFldObstcleSim application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. To see command-line options enter the following from the command-line:

```
$ uFldObstacleSim --help or -h
```

This will show the output shown in Listing 9 below.

Listing 61.9: Command line usage for the uFldObstacleSim tool.

```
1  ============================================================
2  Usage: uFldObstcleSim file.moos [OPTIONS]
3  ============================================================
4
5  Options:
6    --alias=<ProcessName>
7        Launch uFldObstcleSim with the given process
8        name rather than uFldObstcleSim.
9    --example, -e
10       Display example MOOS configuration block.
11   --help, -h
12       Display this help message.
13   --interface, -i
14       Display MOOS publications and subscriptions.
15   --version,-v
16       Display release version of uFldObstcleSim.
17   --verbose=<setting>
18       Set verbosity. true or false (default)
19
20 Note: If argv[2] does not otherwise match a known option,
21       then it will be interpreted as a run alias. This is
22       to support pAntler launching conventions.
```

61.8 Terminal and AppCast Output

The uFldObstacleSim application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 10 below. On line 2, the name of the local community, typically the shoreside community, is listed on the left. On the right, "0/0(204) indicates there are no configuration or run warnings, and the current iteration of uFldObstacleSim is 204. Lines 5-9 show the obstacle file configuration. Lines 10-12 indicate whether ground truth obstacles or simulate sensor points will be generated. Lines 13-15 indicate the range of random durations associated with each obstacle. Lines 16-18 indicate whether the simulator will periodically reset the obstacle locations and if so, how often.

Lines 20-28 reveal the current state of the simulator. Lines 21-22 show how many times the polygon obstacles have been posted as viewable polygons and how many times as given polygons respectively. Lines 23-28 show the state with respect to possibly resetting the obstacles. Line 24 shows the distance between the obstacle region to the closest vehicle. The zero in this case indicates that one or more vehicles are inside the obstacle region. Lines 25-28 show the progress toward potentially resetting the obstacles.

Lines 31-38 show key information related to each obstacle. Column 3 will show total simulated sensor points published per obstacle, if in the *points* mode. Column 4 will be used when given obstacles are published, and incremented each time they are published.

Listing 61.10: Example `UFldObstacleSim` *console output.*

```
 1  ============================================================================
 2  uFldObstacleSim shoreside                                        0/0(204)
 3  ============================================================================
 4  ================================
 5  Config (Obstacles)
 6    Obstacles: 5
 7    MinRange:  20
 8    MinSize:   6
 9    MaxSize:   10
10  Config (Points)
11    Post Points:   false
12    Rate Points:   5
13  Config (Duration)
14    Min Duration: 400.0
15    Max Duration: 500.0
16  Config (Reset)
17    Reset Range:  10
18    Reset_Interv: -1
19  ================================
20  State
21    Viewables Posted: 1
22    Obstacles Posted: 2
23  State (resetting)
24    Min Poly Range: 0
25    Reset Pending:  false
26    Newly Exited :  false
27    Reset Tstamp :  23867167818
28    Reset Total  :  0
29
30
31  Obs   Obs       Points     Given
32  Key   Duration  Published  Published
33  ----  --------  ---------  ---------
34  ob_0  468.1     0          2
35  ob_1  470.1     0          2
36  ob_2  478.2     0          2
37  ob_3  451.7     0          2
38  ob_4  445.3     0          2
```

62 uTermCommand: Poking the MOOSDB with Pre-Set Values

62.1 Overview

The uTermCommand application is a terminal based tool for poking the MOOS database with pre-defined variable-value pairs. This can be used for command and control for example by setting variables in the MOOSDB that affect the behavior conditions running in the helm. There are a few other ways of doing this:

- *pMarineViewer*: The action pull-down menu and on-screen buttons may be used for posting the MOOSDB.
- *uPokeDB*: A command-line tool for poking the MOOSDB. This may be used in conjunction with shell aliases or shell scripts for further convenience.
- *iRemote*: The custom keys feature may be used to bind variable-value pairs to the numeric keys. The primary drawback is the limitation to ten mappings.

62.2 Configuration Parameters for uTermCommand

The variable-value mappings are set in the uTermCommand configuration block of the MOOS file. Each mapping requires one line of the form:

```
cmd

cmd = cue --> variable --> value
```

The *cue* and *variable* fields are case sensitive, and the *value* field may also be case sensitive depending on how the subscribing MOOS process(es) handle the value. An example configuration is given in Listing 11.

Listing 62.11: An example uTermCommand configuration block.

```
1  //------------------------------------------------------------
2  // uTermCommand configuration block
4
5  ProcessConfig = uTermCommand
6  {
7    cmd = override_true  -->  MOOS_MANUAL_OVERRIDE --> true
8    cmd = override_false -->  MOOS_MANUAL_OVERRIDE --> false
9    cmd = deploy_true    -->  DEPLOY       --> true
10   cmd = deploy_false   -->  DEPLOY       --> false
11   cmd = return_true    -->  RETURN       --> true
12   cmd = return_false   -->  RETURN       --> false
13 }
```

Recall the type of a MOOS variable is either a string, double or binary data. If a variable has yet to be posted to the MOOSDB, it accepts whatever type is first written, otherwise postings of the wrong type are ignored. If quotes surround the entry in the value field, it is interpreted to be a string. If not, the value is inspected as to whether it represents a numerical value. If so, it is posted as a double. For example `true` and `"true"` are the same type (no such thing as a Boolean type), `25` is a double and `"25"` is a string.

62.2.1 Run Time Console Interaction

When uTermCommand is launched, a separate thread accepts user input at the console window. When first launched the entire list of cues and the associated variable-value pairs are listed. Listing 12 shows what the console output would look like given the configuration parameters of Listing 11. This configuration block is in the Alpha example mission. You can launch that mission and then launch uTermCommand:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ ./launch.sh
```

Then in a separate terminal:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ uTermCommand alpha.moos
```

The output in the terminal window should look similar to Listing 12. Note that even though quotes were not necessary in the configuration file to clarify that `true` was to be posted as a string, the quotes are always placed around string values in the terminal output.

Listing 62.12: Console output at start-up.

```
1    Cue                  VarName                VarValue
2    -----------          ---------------        --------------
3    override_true        MOOS_MANUAL_OVERRIDE   "true"
4    override_false       MOOS_MANUAL_OVERRIDE   "false"
5    deploy_true          DEPLOY                 "true"
6    deploy_false         DEPLOY                 "false"
7    return_true          RETURN                 "true"
8    return_false         RETURN                 "false"
9
10  >
```

In the Alpha mission, the `DEPLOY` button in the lower right corner of pMarineViewer is configured to post:

```
MOOS_MANUAL_OVERRIDE = false
DEPLOY = true
```

This will be handled instead by uTermCommand in our simple example. A prompt is shown on the last line where user key strokes will be displayed. As the user types characters, the list of choices is narrowed based on matches to the cue. After typing a single 'o' character, only the `override_true` and `override_false` cues match and the list of choices shown are reduced as shown in Listing 13. At this point, hitting the TAB key will complete the input field out to `override_`, much like tab-completion works at a Linux shell prompt.

Listing 62.13: Console output after typing a single character 'r'.

```
1    Cue                  VarName                VarValue
2    -----------          ---------------        --------------
```

```
3     override_true          MOOS_MANUAL_OVERRIDE          "true"
4     override_false         MOOS_MANUAL_OVERRIDE          "false"
5
6   > o
```

When the user has typed out a valid cue that matches a single entry, only the one line is displayed, with the tag <-- SELECT at the end of the line, as shown in Listing 14.

Listing 62.14: Console output when a single command is identified.

```
1     Cue                    VarName                    VarValue
2     -----------            ---------------            --------------
3     override_false         MOOS_MANUAL_OVERRIDE       "false" <-- SELECT
4
5   > override_false
```

At this point hitting the ENTER key will execute the posting of that variable-value pair to the MOOSDB, and the console output will return to its original start-up output. A local history is augmented after each entry is made, and the up- and down-arrow keys can be used to select and re-execute postings on subsequent iterations. To finish the launch in the Alpha mission, use uTermCommand to post DEPLOY=true.

62.3 Connecting uTermCommand to the MOOSDB Under an Alias

A convention of MOOS is that each application connected to the MOOSDB must register with a unique name. Typically the name used by a process to register with the MOOSDB is the process name, e.g., uTermCommand. One may want to run multiple instances of uTermCommand all connected to the same MOOSDB. To support this, an optional command line argument may be provided when launching uTermCommand:

```
$ uTermCommand file.moos --alias=uTermCommandAlpha
```

The command line argument may also be invoked from within pAntler to launch multiple uTermCommand instances simultaneously. The configuration block in the mission file needs to have the same name as the launch alias.

62.4 Publications and Subscriptions for uTermCommand

The only variables published by uTermCommand are those configured and selected by the user at run-time, and uTermCommand does not subscribe for any variables.

63 uSimCurrent: Simulating Drift Effects

63.1 Overview

The uSimCurrent MOOS application is a newcomer in the toolbox and documentation is thin. Nevertheless it has been tested and used quite a bit and is worth a quick introduction here for those with a need for some ability to simulate water current on unmanned vehicles.

uSimCurrent is intended to be used with the uSimMarine simulator, by generating drift vectors and publishing them to the MOOSDB. The uSimMarine simulator has a generic interface to accept externally published drift vectors regardless of the source, written to the variables DRIFT_X, DRIFT_Y, and DRIFT_VECTOR. The uSimCurrent application reads a provided *current field file* containing an association of water current to positions in the water. On iteration of uSimCurrent, the vehicle's current position is noted, looked up in the current-field data structure, and a new drift vector is posted. The idea is shown in Figure 200.

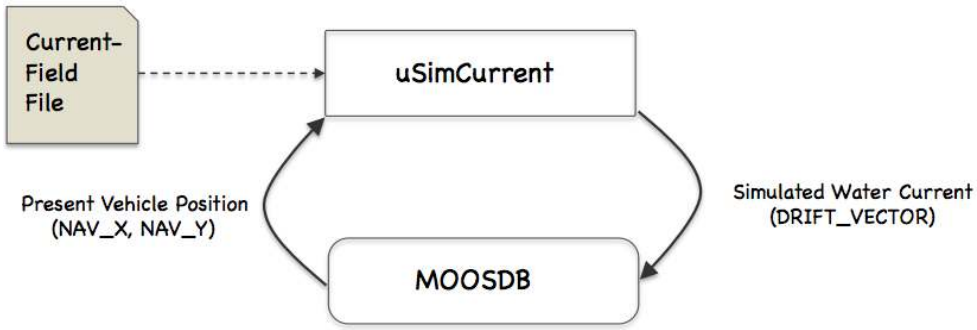

Figure 200: **The uSimCurrent utility:** The simulator is initialized with a data file describing currents and locations. The simulator then repeatedly publishes a current vector based on the present vehicle position.

63.2 Configuration Parameters for uSimCurrent

The following configuration parameters are defined for uSimCurrent. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 63.15: Configuration Parameters for uSimCurrent.

current_field:	Name of a file describing a current field.
current_field_active:	Boolean indicating whether the simulator is active.

63.3 Publications and Subscriptions for uSimCurrent

The interface for uSimCurrent, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uSimCurrent --interface or -i
```

63.3.1 MOOS Variables Published by uSimCurrent

The primary output of uSimCurrent to the MOOSDB is the drift vector to be consumed by the uSimMarine application.

- DRIFT_VECTOR: drift vector representing the prevailing current. See the uSimMarine documentation.
- USC_CFIELD_SUMMARY: Summary of configured current field.
- VIEW_VECTOR: Vector objects suitable for rendering in GUI applications.

63.3.2 MOOS Variables Subscribed for by uSimCurrent

Variables subscribed for by uSimCurrent are summarized below.

- NAV_X: The ownship vehicle position on the x axis of local coordinates.
- NAV_Y: The ownship vehicle position on the y axis of local coordinates.

64 iSay: Invoking the Linux or OSX Speech Generation Commands from MOOS

64.1 Overview

The iSay application is a way to invoke the native speech generation utilities of OSX (the say command) and GNU/Linux (the espeak command). It may also invoke the afplay commands available in both OSX and GNU/Linux to play a given wav or mp3 file. Normally, without MOOS running, one can generate voice from the command line with these built in commands:

```
$ say hello
```

in OSX, or in GNU/Linux with:

```
$ espeak hello
```

When iSay is running, the same can be accomplished by poking the MOOSDB:

```
$ uPokeDB SAY_MOOS=hello
```

The advantage being of course that the speech may be generated by an event in MOOS, e.g. a vehicle returning. The iSay application also makes provisions for choosing the voice, the time in between utterances or in between utterance beginnings, and an utterance priority if a queue of utterances begins to form due to a backlog of several utterance requests. The utterance may be either a specified text of speech, or it may be a sound provide by a named `.wav` file. Presently, if a named `.wav` file is not found or if a named voice is not recognized, no sound will be generated. Future versions of iSay may make provisions for fallback sounds in these cases.

64.2 Configuration Parameters for iSay

The following parameters are defined for iSay. A more detailed description is provided in other parts of this section. Parameters having default values are indicated.

Listing 64.16: Configuration Parameters for iSay.

audio_dir:	Names a system directory to be added to path of directories to search for audio files named in a particular utterance request.
default_rate:	The speech rate to be used, in words per minute, if left unspecified for a particular utterance request (200).
default_voice:	The default voice to be used if left unspecified for a particular utterance request.
interval_policy:	Either `"from_start"` or `"from_end"`, indicating whether the interval of time between utterances is marked from the start of the previous utterance or the end of the previous utterance (`"from_end"`).
os_mode:	Either `"osx` or `"linux"` or `"both"`. The default is `"both"`

`min_utter_interval`: The minimum time (in seconds) between utterances before a next queued utterance will commence (1.0).

64.3 Publications and Subscriptions of iSay

The interface for `iSay`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ iSay --interface or -i
```

64.3.1 Variables Published by iSay

The only output of `iSay` is:

- `APPCAST`: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 64.7.

64.3.2 Variables Subscribed for by iSay

The `iSay` application will subscribe for the following four MOOS variables:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 92.5.
- `SAY_MOOS`: A description of an utterance to be made. Section 64.4.
- `SAY_FILTER`: Either `"none"`, `"hold"`, or `"ignore"` (`"none"`). Section 64.6.
- `SAY_VOLUME`: Either `"mute"`, `"vsoft"`, `"soft"`, `"normal"`, `"loud"`, `"vlou"`, or a number in the range `"[0,2]"`. This will only affect the volume of wav files only on MacOS, relative to the prevailing output volume on the computer. Text to speech (MacOS,Linux) and wav files on Linux, are not affected.

64.4 Specifying an Utterance

The `iSay` app works by receiving utterance requests through incoming MOOS mail via the `SAY_MOOS` variable. This mail has the following format by a couple examples:

```
SAY_MOOS = "say={hello world}, priority=100, voice=Albert, rate=250, source=henry"
SAY_MOOS = "file=file.wav"
```

The `source` component is optional and by default will be set to a string comprised of the MOOS community and the MOOS application that generated the utterance request, separated by a colon. See, for example, line 24 in Listing 17. This component provides an opportunity to be more specific about the source, as in line 25 in the same example.

The `rate` component specifies the speech rate in words per second. The default rate is 200. The `priority` component specifies the priority of the utterence request when the request is pushed onto the priority queue. See Section 64.5 for more on this.

The voice component allows one to request a voice different from the default voice. The options for this component depend on the options supported by the OSX say command and the GNU/Linux espeak command. These options may be found by typing "say -v ?" on the command line or "espeak --voices" on the command line in GNU/Linux.

64.5 The iSay Priority Queue

The iSay priority queue holds all received utterance requests before they are eventually popped and processed. The priority queue is popped (if non-empty) once on each iteration of iSay, and no more frequently than the allowed by the setting of min_utter_interval and the time it took to process the system command of the previous utterance.

By default, all uttterances have a priority of zero, and are simply processed first-come first-served. An utterance request may instead be specified with a higher (or lower) priority to affect the order in which it is popped from the queue. Requests of equal priority are processed first-come first-served. A request may also be made with "top" priority. These requests go immediatedly to the top of the priority queue. If multiple top priority requests are received, the latest one received will be handled first.

64.6 The iSay Filter

The iSay filter may be set to one of three values. The default is "none" and all utterances are received and processed normally. When the filter is set to "ignore", it's as if the mute button is hit. Any received utterances will be immediately put on the queue and any utterances popped from the queue will be handled according to the normal interval specifications. But the actual system command to generate the utterance will not be made. When the filter is set to "hold", all received utterances will be put in the queue and nothing will be popped from the queue until the hold is released. At some point the maximume queue size (100) may be exceeded.

64.7 Terminal and AppCast Output

The iSay application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 17 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. See Section 90.1 for more on appcasting and viewing appcasts. The counter on the end of line 2 is incremented on each iteration of iSay, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

The output in the below example comes from the example described in Section 64.8.

Listing 64.17: Example terminal or appcast output for iSay.

```
1  ============================================================
2  iSay alpha                                           0/0(96)
3  ============================================================
4  Configuration Parameters:
5  -------------------------
6     Default Voice: alex
7      Default Rate: 200
8   Max Utter Queue: 100
```

```
 9   Min Utter Inter: 1
10   Interval Policy: from_end
11
12  Status:
13  -------------------------
14    Utter Queue Size: 36
15    Unhandled Audios: 0
16               Filter: none
17
18  Source              Time  Time  Utterance
19                      Recd  Post
20  ------------------  ----  ----  ---------
21  alpha:uTimerScript  12.8  13.7  five
22  alpha:uTimerScript  1.4   12.1  four
23  alpha:uTimerScript  10.0  10.4  five
24  alpha:uTimerScript  7.7   8.7   five
25  james               1.2   7.1   three
26  alpha:uTimerScript  4.4   5.4   five
27  alpha:uTimerScript  1.2   3.9   two
28  alpha:uTimerScript  1.6   2.2   five
29  alpha:uTimerScript  0.7   0.7   one
30
31  ================================================================
32  Most Recent Events (8):
33  ================================================================
34  [14.33]: Utter Rec'd:say={nine}
35  [13.66]: Say:five
36  [13.46]: Utter Rec'd:say={eight}
37  [13.26]: Utter Rec'd:say={seven}
38  [12.96]: Utter Rec'd:say={six}
39  [12.76]: Utter Rec'd:say={five},priority=100
40  [12.66]: Utter Rec'd:say={four}
41  [12.66]: Utter Rec'd:say={three},source=james
```

The first few lines (4-10) show the configuration settings for iSay. The status of iSay is shown in Lines 12-16. In this case the queue is growing fairly quickly (line 14) since the utterance requests are being received about four times per second, but the minimum time between utterances is set to be one second (line 9). Processed utterances are shown in reverse order starting on line 21. This block has a maximum length of ten items, to keep the size of the AppCast report in check.

Recent events are shown in the block beginning on line 34. Events may be either a received or processed utterance. This block has a maximum length of eight items, to keep the size of the AppCast report in check.

64.8 A Simple Example

The below is a simple example, using the uTimerScript application to generate inputs to iSay. You can try this example by adding these to blocks to an existing example mission and adding iSay and uTimerScript to the Antler block. The script should generate spoken utterances of the numbers one through nine pretty much in sequence except that fives are handled with a high priority and spoken almost immediately. Since the min_utterance_interval is set to one second, fives are spoke almost half the time, and the utterance queue also grows fairly quickly. Try changing this parameter to 0.1

seconds instead and note the change in behavior.

Listing 64.18: A Simple `iSay` *Example.*

```
//------------------------------------------
ProcessConfig = iSay
{
  AppTick   = 4
  CommsTick = 4

  min_utter_interval = 1
  interval_policy    = from_end
  volume             = soft
}

//------------------------------------------
ProcessConfig = uTimerScript
{
  AppTick   = 4
  CommsTick = 4

  paused       = false
  reset_max    = nolimit
  reset_time   = all-posted
  delay_reset  = 0.25

  event        = var=SAY_MOOS, val="say={one}",   time=0.25
  event        = var=SAY_MOOS, val="say={two}",   time=0.5
  event        = var=SAY_MOOS, val="say={three},source=james", time=0.75
  event        = var=SAY_MOOS, val="say={four}",  time=1
  event        = var=SAY_MOOS, val="say={five},priority=100",  time=1.25
  event        = var=SAY_MOOS, val="say={six}",   time=1.5
  event        = var=SAY_MOOS, val="say={seven}", time=1.75
  event        = var=SAY_MOOS, val="say={eight}", time=2
  event        = var=SAY_MOOS, val="say={nine}",  time=2.25
}
```

65 pMissionHash: A Lightweight App for Generating a Mission Hash

65.1 Overview

The pMissionHash app was meant to run and generate a mission hash when (a) pMarineViewer is *not* being run, and (b) mission hash generation is still desired. A mission hash is normally generated on the shoreside MOOS community in the form of a posting to the variable MISSION_HASH. Normally the hash is generated and posted by pMarineViewer. However, in headless missions, where there is no GUI and no pMarineViewer, a mission hash posting can be generated instead with pMissionHash. Headless mission configurations are common in automated missions, e.g., for validation, automated competitions, or Monte Carlo testing.

Why are mission hashes important? The mission hash is created and logged on the shoreside and shared to and logged on all the vehicles. The presence of identical mission hashes is the definitive confirmation that the individual log files are all part of the same instance of a particular mission, either in simulation or in the physical world. As of this writing, a mission hash is generated either with pMarineViewer or pMissionHash. The command-line tool mhash_gen is also distributed with MOOS-IvP but is not generally part of a mission.

65.2 Typical Application Topology

The pMissionHash app typically runs in the shoreside community, in a headless mission, when pMarineViewer is not running.

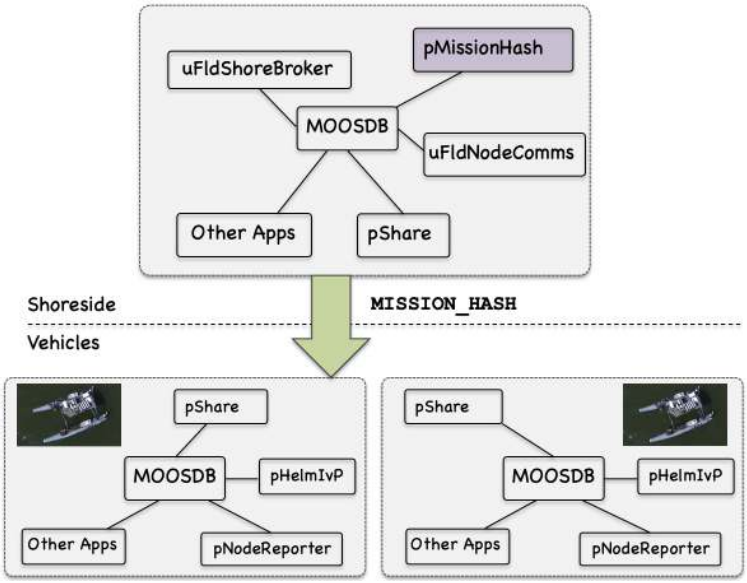

Figure 201: **Typical pMissionHash Topology:** A unique randomly created mission hash is generated by pMissionHash at startup, and shared to the MOOS communities of all vehicles. This unique has is thus logged in all log files for each vehicle and shoreside, thus allowing for confirmation that the log files were all part of the same mission.

65.3 Configuration Parameters of pMissionHash

The following parameters are defined for pMissionHash. If a configuration block is not provided for pMissionHash in the mission file, this is acceptable and no warning will be posted.

Listing 65.19: Configuration parameters for pMissionHash.

mission_hash_var:	The variable to which the mission hash is posted. The default is MISSION_HASH.
mhash_short_var:	The variable to which the short version of the mission hash is posted. If this parameter is not specified, this information will not be published. This cannot be the same variable as the mission_hash_var.

65.3.1 An Example MOOS Configuration Block

Listing 20 shows an example MOOS configuration block produced from the following command line invocation:

```
$ pMissionHash --example or -e
```

Listing 65.20: Example configuration of the pMissionHash application.

```
1  ============================================================
2  pMissionHash Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pMissionHash
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   mission_hash_var = MISSION_HASH  // default
11   mhash_short_var  = MHASH_SHORT   // default is disabled
12
13   // Note: If a config block is NOT provided in the mission
14   //       file, NO config warning will be generated.
15 }
```

65.3.2 Variables Published

The primary output of pMissionHash to the MOOSDB is the mission hash message, or the short mission hash publication if this is enabled.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 65.5.
- MISSION_HASH: The full mission hash. This may be published to another variable name, depending on the user configuration.
- MHASH_SHORT: The short mission hash. This will only be published if enabled through configuration.

For examples of a full and short mission hash, see Section 65.5.

Both variables are published immediately upon startup, and once every 30 seconds thereafter.

65.3.3 Variables Subscriptions

The pMissionHash application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- DB_CLIENTS: The list of running MOOS apps is monitored, checking for pMarineViewer, and posting a warning if present. Both apps should not be running and generating conflicting mission hash values.
- RESET_MHASH: A request to reset the mission hash.

65.4 Command Line Usage of pMissionHash

The pMissionHash application is typically launched with pAntler, along with a group of other vehicle modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uFldMessageHandler --help or -h
```

Listing 65.21: Command line usage for the uFldMessageHandler tool.

```
1  ============================================================
2  Usage: pMissionHash file.moos [OPTIONS]
3  ============================================================
4
5  SYNOPSIS:
6  ----------------------------------
7    The pMissionHash app is used for generating a MISSION_HASH
8    posting. Normally this is produced by pMarineViewer. In
9    headless mission not running pMarineViewer, this app can be
10   used instead. They should not be both run unless the mission
11   feature is configured to be disabled in pMarineViewer.
12
13 Options:
14   --alias=<ProcessName>
15       Launch pMissionHash with the given process name
16       rather than pMissionHash.
17   --example, -e
18       Display example MOOS configuration block.
19   --help, -h
20       Display this help message.
21   --interface, -i
22       Display MOOS publications and subscriptions.
23   --version,-v
24       Display the release version of pMissionHash.
25   --web,-w
26       Open browser to:
27       https://oceanai.mit.edu/ivpman/apps/pMissionHash
```

```
28
29  Note: If argv[2] does not otherwise match a known option,
30        then it will be interpreted as a run alias. This is
31        to support pAntler launching conventions.
```

65.5 Terminal and AppCast Output

The pMissionHash application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 22 below. On line 2, the name of the local community or vehicle name is listed on the left. On the right, "0/0(95) indicates there are no configuration or run warnings, and the current iteration of pMissionHash is 95.

On line 4, the full mission hash is shown and the variable it is posted to, by default MISSION_HASH. On line 6, the short version of the the mission hash is shown, presumably do to the configuration of mhash_short=MHASH.

Listing 65.22: Example appcast and terminal output of pMissionHash.

```
1  ============================================================================
2  pMissionHash shoreside                                              0/0(95)
3  ============================================================================
4  MISSION_HASH=mhash=250127-1826S-LUSH-ARMY,utc=1738020397.43
5  MHASH=LUSH-ARMY
```

66 pAutoPoke: Automated Pokes for Headless Missions

66.1 Overview

The pAutoPoke application is a tool to enable an automatic poke to the MOOSDB with one or more configured publications. Typically this is in service of conducting headless, auto-tested missions, where there is no user to kick off a mission by hitting a "DEPLOY" button on a command and control GUI, such as apppMarineViewer

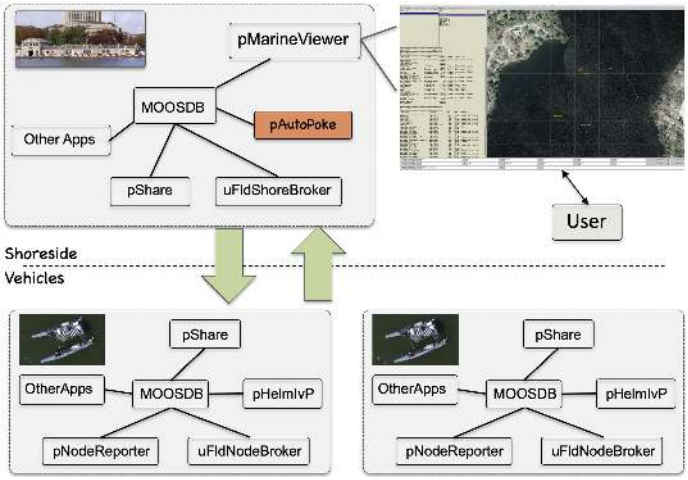

Figure 202: **Typical pAutoPoke Topology:** The flags configured for pAutoPoke are posted immediately after connecting to the MOOSDB. If configured to wait for vehicle connections, the pShoreBroker app will publish the number of connected vehicles in the variable UFSB_NODE_COUNT.

66.2 Configuration Parameters for pAutoPoke

The pAutoPoke application may be configured with a configuration block within a MOOS mission file, typically with a `.moos` file suffix. The following parameters are defined for pAutoPoke.

Listing 66.23: Configuration Parameters for pAutoPoke.

flag:	A MOOS variable and value to be published.
required_nodes:	The number of nodes required to be detected, before flags are published. By default this is zero.

66.2.1 An Example MOOS Configuration Block

An example MOOS configuration block may be obtained from the command line with the following:

```
$ pAutoPoke --example or -e
```

Listing 66.24: Example configuration of the pAutoPoke application.

```
1  ============================================================
2  pAutoPoke Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pAutoPoke
6  {
7    AppTick   = 2
8    CommsTick = 2
9
10   flag = MOOS_MANUAL_OVERRIDE_ALL=false
11   flag = DEPLOY_ALL=false
12
13   required_nodes = 2
14 }
```

66.3 Publications and Subscriptions for pAutoPoke

The interface for pAutoPoke, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pAutoPoke --interface or -i
```

66.3.1 Variables Published by pAutoPoke

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.

66.3.2 Variables Subscribed for by pAutoPoke

The pAutoPoke application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- DB_RWSUMMARY: A report generated by the MOOSDB listing, for each app connected to the MOOSDB, the set of variables subscribed for by each app, and the set of variables observed to be published by each app.

Node that pAutoPoke will also subscribe for all variables discovered from the contents of DB_RWSUMMARY publications.

66.3.3 Command Line Usage of pAutoPoke

The pAutoPoke application is typically launched with pAntler, along with a group of other modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ pAutoPoke --help or -h
```

Listing 66.25: Command line usage for the pAutoPoke tool.

```
1  Usage: pAutoPoke file.moos [OPTIONS]
2
3  Options:
4    --alias=<ProcessName>
5        Launch pAutoPoke with the given process
6        name rather than pAutoPoke
7    --example, -e
8        Display example MOOS configuration block
9    --help, -h
10       Display this help message.
14   --interface, -i
15       Display MOOS publications and subscriptions.
16   --version,-v
17       Display the release version of pAutoPoke.
18   --web,-w
19       Open browser to: https://oceanai.mit.edu/ivpman/apps/pAutoPoke
20
21 Note: If argv[2] is not of one of the above formats
22       this will be interpreted as a run alias. This
23       is to support pAntler launching conventions.
```

66.4 Terminal and AppCast Output

Some useful information is published by pAutoPoke to the terminal on every iteration. An example is shown in Listing 26 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. The counter on the end of line 2 is incremented on each iteration of pAutoPoke, and also serves as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 66.26: Example terminal or appcast output for pAutoPoke*.*

```
1  ============================================================
2  pAutoPoke shoreside                                  0/0(44)
3  ============================================================
4  Required Nodes:2
5  Flags:
6  [0]: MOOS_MANUAL_OVERRIDE_ALL=false
7  [1]: DEPLOY_ALL=true
8
9  Node Count:   2
10 flags_posted: true
```

Lines 4-7 of the output show the user configuration. Lines 9 and 10 show the current state of the app.

67 pMissionEval: Evaluating User-Defined Mission Success

67.1 Overview

The pMissionEval application is a tool primarily used in a mission that is part of an automated test. An automated test is typically run headlessly, i.e., no GUI, no human interaction. It is meant to at times complement other MOOS apps that serve to evaluate a mission. For example, in an obstacle avoidance mission, there may be other another app that detects collisions, and another app that randomizes aspects of the obstacle placement. The role of pMissionEval is to work in conjunction with the apps of the mission to (a) define when a mission has completed, (b) define what constitutes the result to be reported, and (c) write the mission salient mission configuration aspects and results to a file in a user defined format.

The typical topology is shown below, although it is possible to run pMissionEval in a vehicle commmunity, or in a simple Alpha-like mission with a single MOOS community.

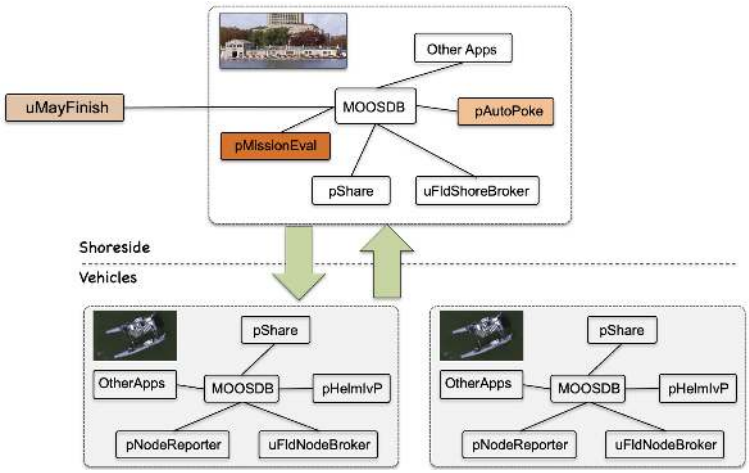

Figure 203: **Typical pMissionEval Topology:** Typically run in the shoreside community, pMissionEval, performs a series of evaluations at events configured by the user, with results posted either to the MOOSDB or to a file, or both.

Note in the figure that two other apps are typically operated in coordination:

- pAutoPoke: When the mission is launched, this app will poke the MOODB with the commands normally provided by a human command to kick off a mission, e.g., `DEPLOY=true`.
- uMayFinish: After the mission is launched, from within a script, this app is launched from with in the script, essentially blocking the script until the mission is finished. This app connect to the MOOSDB and monitor for a posting such as `MISSION_EVALUATED=true`. Then disconnect from the MOOSDB and exit, presumably to allow the remainder of the blocked script to complete bring down the mission.

The primary action produced by pMissionEval is the logging of a single line of text in a named file, e.g., `results.txt`. This file is named in a configuration parameter for pMissionEval. After a mission has been completed, there may be a line produced like:

```
form=alpha, mmod=cycles, grade=pass, mhash=K-POSH-TIME, date=250126, time=1508
```

When pMissionEval determines that evaluation has completed, it will (a) write a result like like the one above to the results file, and (b) post something like MISSION_EVALUATED=true, to match the configuration of uMayFinish to trigger the shut down of the simulation.

67.2 Four General Mission Types

Consider the following mision types:

- A simple pass/fail sanity check on certain defined features of an app or a behavior. Sometimes referred to as a unit test.
- A performance pass/fail test that requires a lengthy mission duration, with randomization introduced during the mission. Internally the pass/fail determination may be derived from a non Boolean score, e.g., the number of vehicle near-misses over 100 obstacle encounters, but the mission is still a pass/fail test, with the test designer deciding whether success is say 0 near misses or some tolerable number.
- A test that produces a performance score, as in the example above, also possibly involving randomized starting conditions, e.g., obstacle locations. The difference is that this test also has several configuration variations, such as vehicle speed, turning capability, sensor distance, communications drop-out. Rather than this mission being a single component of a CI pipeline, it is part of set of sensitivity test, find the performance of algorithms relative to configuration and environment assumptions. A user may later wish to identify a few of the variations later to be part of a CI pipeline.
- A competition mission that pits two teams, producing a resulting winner and score.

Each of these four mission types are fleshed about a bit more in the following examples.

67.2.1 The Alpha Mission Pass/Fail Verifying Event Flags

The Alpha mission is the first hello-world mission to where new users of MOOS-IvP are directed on an initial download. A vehicle starts and proceeds through five waypoints in a home-plate pattern. It may traverse the patterntwice before finishing and returning home.

The waypoint behavior has certain event flags defined. The endflag is defined for all behaviors and is published when the behavior "completes". For the waypoint behavior, completion is when it has traversed all waypoints. If the waypoint specific parameter repeat is set to 1, then the behavior will not complete until it has traverse all five points twice.

The waypoint behavior has two additional event flags unique to this behavior: the cycle_flag and wpt_flag. The former is published each time the waypoint behavior completes a set of waypoints, and the latter is published each time the behavior arrives at any waypoint. So, for a waypoint behavior configured with five waypoints and repeat=1, the cycle_flag should be posted twice, and the wpt_flag should be posted 10 times.

For such a mission, something like the below result line would be expected.

```
form=alpha, mmod=eflags, grade=pass, mhash=J-BARE-LEAF, date=250126, time=1601
```

The `form=alpha` indicates this is a result of the alpha mission. The `mmod=eflags` is a "mission modification" tag. This is discussed further in Section **??**. A mission may have multiple mods, each to test something slightly different, thus allowing one mission folder to serve multiple purposes.

67.2.2 The ObAvoid Mission Pass/Fail Verifying Obstacle Avoidance

In the ObAvoid mission, a vehicle is driven around a race-track like pattern, through a rectangular region situated between the two ends of the racetrack pattern. The region is populated with a randomly generated obstacle field. A shoreside MOOS app, `uFldCollObDetect`, runs and montitors the position of vehicles relative to the obstacles in the field. For each obstacle encounter, the closest point of the encounter is noted, and evaluated. An encounter with an obstacle, with a closest point of approach (CPA), wwithin a certain distance, is considered a collision. A near-miss might be defined as an encounter with a CPA distance within a slightly larger distance.

The automated version of this mission can be configured to run the vehicle for a given number of times around the racetrack, or for a given amount of obstacle encounters. After a certain amount of runs, the mission receives a passing grade if no collisions are detected.

For such a mission, something like the below result line would be expected.

```
form=obavoid, mmod=ob10, grade=pass, mhash=K-ICED-DINO, date=250126, time=1614
```

The `form=obavoid` indicates this is a result from the obavoid mission. The `mmod=ob10` is a modifier that may indicate in this mission there were ten randomly generated obstacles.

67.2.3 The ObAvoid Mission with Sensitivity Analysis

In the same ObAvoid mission as above, the mission could be graded as a `fail` if a collision with an obstacle is detected. I could also be given a score, from 0 to 100, based on the percentage of near-misses noted for each encounter with an obstacle.

The important distinction is that mission can be varied in several ways. (a) the density of the obstacle field, (b) the minimal guaranteed separation distance between randomly generate obstacles in the field, (c) the range at which the vehicle can detect the obstacle, (d) the turn rate of the vehicle.

An experiment would involve potentially large numbers automated simulations. The result of each simulation would be a single pass/fail grade or score. The goal is plot the relationship between the above mentioned conditions, versus performance.

For such a mission, something like the below result lines would be expected.

```
form=obavoid, mmod=ob6, hits=0, nmisses=0, mhash=A-WEAK-BEER, date=250126, time=1614
form=obavoid, mmod=ob6, hits=0, nmisses=0, mhash=K-LAZY-WEST, date=250126, time=1615
form=obavoid, mmod=ob6, hits=0, nmisses=0, mhash=P-DEAD-ENZO, date=250126, time=1617
...
form=obavoid, mmod=ob10, hits=0, nmisses=2, mhash=M-INKY-SIZE, date=250126, time=1623
form=obavoid, mmod=ob10, hits=0, nmisses=0, mhash=D-COOL-REED, date=250126, time=1624
form=obavoid, mmod=ob10, hits=0, nmisses=1, mhash=M-EASY-VICE, date=250126, time=1637
...
form=obavoid, mmod=ob14, hits=0, nmisses=5, mhash=D-DANK-PHIL, date=250126, time=1639
form=obavoid, mmod=ob14, hits=1, nmisses=8, mhash=G-COLD-KERR, date=250126, time=1641
form=obavoid, mmod=ob14, hits=1, nmisses=2, mhash=C-MAIN-KNOX, date=250126, time=1644
```

The `form=obavoid` indicates this is a result from the obavoid mission. The `mmod=ob10` is a modifier that may indicate in this mission there were ten randomly generated obstacles. The overall test involves several instances of each configuration. In this example, the number of obstacles presumably increases the difficulty in obstacle avoidance and the data should enable a plot relating obstacle density with to the frequency of hits or near misses.

67.2.4 The Rescue Mission Head-to-Head Competion

The Rescue mission is a head-to-head competition mission from the MIT 2.680 Marine Autonomy class. This mission is essentially in the style of an Easter egg hunt. Two vehicles are deployed to rescue N swimmers, by visiting each swimmer location. The vehicle to visit the majority of the N swimmers wins. There is an game manager app running in the shoreside community called `uFldRescueMgr`. This app will publish something like

```
WINNER   = blue
SCORE    = blue=9 # green=7
SWIMMERS = 17

COMPETITION_COMPLETE = true
```

When launched with a GUI, the participants could watch the mission play out, and check the scoreboard in the appcasting output of `uFldRescueMgr` to see the winner and indication of completeness. In an automated headless mode, `pMissionEval` would wait for `COMPETITION_COMPLETED` to be true, and publish a line to the results file looking something like:

```
form=rescue, winner=blue, score=blue=9 # green=7, mhash=K-PLUS-MOTH, date=250126, time=1644
```

Since the competition involves randomly generated starting positions and locations of swimmers, an experiment may involve many such automated competition missions to get a good feel for the team with the better strategy.

67.3 Basic Operation

In any mission, `pMissionEval` will do essentially the following three things:

- Wait until the proper point in the mission
- Evaluate the results of the mission
- Post the results to either a file or to the MOOSDB, or both

In an automated, headless mission, there is always the concern about having an absolute timeout, in case for some reason the first stage above fails to happen. However, pMissionEval is typically not concerned about this. The uMayFinish app is normally configured with an absolute for aborting a mission.

67.3.1 Conditions and Mission Evaluation

pMissionEval can be configured with one or more logic conditions confiured to convey a mission stage or mission completeness. A simple example is

```
lead_condition = ARRIVED = true
```

When configured this way, pMissionEval will register for the MOOS variable ARRIVED. When it receives mail with a value of `true`, the mission will be evaluated and results posted. Evaluation in terms of pass/fail is determined by an additional two kinds of conditions, *pass conditions* and *fail conditions*. A result of *pass* will be award if (a) all pass conditions hold and (b) no fail condition holds. For example, the above line may be further configured with:

```
pass_condition = ODOMETRY < 1500
pass_condition = FOUND_OBJECT = true
fail_condition = COLLISIONS > 0
```

The above three lines will be evaluated only when the lead condition is satisfied, ARRIVED=true. If both pass conditions are satisfied, and none of the fail conditions are satisfied, then the mission is (a) considered to be evalutated, and (b) the pass/fail result is set to pass.

67.3.2 Results of a Mission Evaluation

When a mission is considered to be evaluated (lead conditions satisfied), either or both of the above actions are taken:

- User configured result flags are posted to the MOOSDB
- User configures result lines are written to a file.

Result flags come in one of three flavors: *pass* flag, *fail* flags, and simply *result* flags. The first is published only when the pass/fail result passes, the second only when it fails, and the result flag is published either way. Here are some examples:

```
pass_flag = RESULT=success
fail_flag = RESULT=failed
result_flag = MISSION_EVALUATED=true
```

The other action taken on evaluation is the publication of a result line to a result file. For example, if configured:

```
result_file = result.txt
result_col = result=$[GRADE]
```

The action on mission evaluation will be to write a single line to the file `results.txt`:

```
result=pass
```

Continuing with the above example, we know that if the result passes then the two conditions involving ODOMETRY and FOUND_OBJECT must have passed. But way may want to also know the actual odometry value. Using another column

```
result_col = odometry=$[ODOMETRY]
```

then the line written to the results file would be:

```
result=pass, odometry=1451
```

Any number of result file columns can be configured.

67.3.3 Multi-Part Tests

A multi-part test may be used for evaluating a mission at different stages. In a single mission, perhaps a vehicle is designed to transit to a point, then loiter for some period of time, and then return home, finally setting ARRIVED=true. Rather than just checking to see if that post was eventually made, the intermediate stages can be verified too with a *multi-aspect logic sequence*. Each aspect is comprised of one or more lead conditions and one or more pass/fail conditions.

```
lead_condition = ARRIVED_MID = true
pass_condition = ODOMETRY < 500

lead_condition = LOITER_FINISH = true
pass_condition = ODOMETRY < 1000

lead_condition = ARRIVED = true
pass_condition = ODOMETRY < 1500
```

Each pair above is *aspect* and the group is called a *logic sequence*. The overall mission will not be considered evaluated and results will not be posted or written to a file, until the full sequence is evaluated. There are no limits to the number of lead, pass and fail conditions per aspect, and there are no limits on the number of aspects. The appcasting output of pMissionEval will show the current progress of a multi-aspect logic sequence.

Note: Unlike configuration parameters in nearly all other MOOS apps, the *order* of the configuration lines is significant.

67.3.4 Structuring the Results File

The results file may have as many columns as the user desires, and there are no minimums or limits on the columns. Columns are by default separated by white space, but they can be configured to be separated by a comma (csp, for comma-separated-pairs) with:

```
report_line_format = csp
```

The results file will likely contain results from multiple missions, or mission types. Some consideration should be made to add information to report lines with an eye toward how the results file will be later processed, as part of a CI pipeline or a file for generating plotted data. With this in mind, there are a number of provisions for macros available for configuring report lines, described next.

67.3.5 Report Macros

Any of the flags (result, pass, or fail flags) may include a *macro*. Likewise, any report column may also contain a macro. Macros are of the form `$[MACRO]`. A macro may be something built into the app, such as `$[DATE]`, or others listed below. If a `$[MACRO]` is not one of the built-in macros, the pMissionEval will interpret this to mean that the macro refers to a MOOS variable.

For example, consider a mission with an odometry app running and continuously, posting to the variable ODOMETRY. We may want to construct an evaluation that uses either (a) a result flag, or (b) a report column, that uses the odometry information:

```
result_flag = DIST = $[ODOMETRY]
```

or:

```
report_column = dist=$[ODOMETRY]
```

Built-in macros:

- GRADE: This is the mission result value. It will be either ”pending”, ”pass”, or ”fail”.
- MHASH_SHORT: The short version of a mission hash string. An example long version would be `"mhash=250117-1614B-MEEK-EROS,utc=1737148477.26"`. The short version would simply be `"MEEK-EROS"`.
- MHASH_LSHORT: A longer version of the short version, keeping the single letter preceding it. For example, `"B-MEEK-EROS"`, instead of `"MEEK-EROS"`.
- MHASH: The full version of a mission hash, e.g., `"mhash=250117-1614B-MEEK-EROS,utc=1737148477.26"`.
- MISSION_HASH: Same as `$[MHASH]`.
- MHASH_UTC: The timestamp component of a mission hash string. An example long version would be `"mhash=250117-1614B-MEEK-EROS,utc=1737148477.26"`. The timestamp component would simply be `"1737148477.26"`.
- MISSION_FORM: This is the value set in the `mission_form` configuration parameter. It should be set to the colloquial name of the mission, e.g., the ”Alpha” mission.
- MISSION_MOD: This is the value set in the `mission_mod` configuration parameter. If a mission has several mods, then this is the place to distinguish that.

- WALL_TIME: The wall time is the DB_UPTIME value, divided by the time warp value.

Built-in Time and Date macros:

- DATE: A string in the format of YYYY:MM:DD.
- TIME: A string in the format of HH:MM:SS.
- YEAR: A string in the format of YYYY.
- MONTH: A string in the format of MM.
- DAY: A string in the format of DD.
- HOUR: A string in the format of HH.
- MIN: A string in the format of MM.
- SEC: A string in the format of SS.

67.4 Four Example Missions

67.4.1 The Alpha Mission Pass/Fail Verifying Event Flags

Returning to the Alpha mission Section 67.2.1. The goal is to verify the end flag, waypoint flag and cycle flag features. First, the waypoint behavior is configured to increment a counter on each posting of the flags as fallows.

```
wptflag   = WPT_FLAG=$[CTR1]
cycleflag = CYCLE_FLAG=$[CTR2]
endflag   = ARRIVED=true
```

Two tests are made in two logic aspects:

```
lead_condition = CYCLE_FLAG = 1
pass_condition = WPT_FLAG = 5

lead_condition = ARRIVED = true
pass_condition = WPT_FLAG = 10
pass_condition = CYCLE_FLAG = 2

mission_form  = alpha_ufld
mission_mod   = mod1
report_file   = results.txt
report_column = grade=$[GRADE]
report_column = form=$[MISSION_FORM]
report_column = mmod=$[MMOD]
report_column = mhash_short=$[MHASH_SHORT]
report_column = lshort_short=$[MHASH_LSHORT]

result_flag = MISSION_EVALUATED = true
```

67.4.2 The ObAvoid Mission Pass/Fail Verifying Obstacle Avoidance

Returning to the obstacle avoidance mission described in Section 67.2.2. This is misison `02-obavoid` in the `missions-auto` repository. A vehicle is driven through an obstacle field with a dedicated MOOS app monitoring each obstacle encounter. This app is called `uFldCollObDetect`. It publishes three key variables:

- `OB_TOTAL_ENCOUNTERS`: Incremented each time a vehicle passes an obstacle.
- `OB_TOTAL_COLLISIONS`: Incremented each time a vehicle collides with an obstacle.
- `OB_TOTAL_NEAR_MISSES`: Incremented each time a vehicle encounter has a near miss with an obstacle.

For each of the above, the app is configured with range threshold, e.g, a 20 meter CPA to an obstacle counts as an encounter, a 5 meter CPA is a near miss, and 1 meter CPA is considered a collision.

A simple pass/fail test can be constructed with the below `pMissionEval` configuration:

```
lead_condition = (OB_TOTAL_ENCOUNTERS > $(TEST_ENCOUNTERS))

pass_flag   = OB_TOTAL_COLLISIONS = 0
result_flag = MISSION_EVALUATED = true

report_column = min_sep=$(SEP)
report_column = grade=$[GRADE]
report_column = encounters=$[OB_TOTAL_ENCOUNTERS]
report_file = results.log
```

This test can be conducted by running, using time warp 10:

```
$ xlaunch.sh 10
```

This should produce a line a output in file `results.log` similar to:

```
min_sep=10  grade=pass  encounters=25
```

Note in this mission, the launch script accepts command line arguments setting the (a) the minimum separation between obstacles, and (b) the total number of encounters tested. The defaults are 10 and 25 respectively as shown in the above result line. The values are expanded in the `nsplug` macros `$(SEP)` and `$(TEST_ENCOUNTERS)`. The mission could be launched instead with:

```
$ xlaunch.sh 10 --sep=12 --enc=40
```

This should produce a line a output in file `results.log` similar to:

```
min_sep=12  grade=pass  encounters=40
```

67.4.3 The ObAvoid Mission with Sensitivity Analysis

Staying with the same obstacle avoidance mission, a sequence of tests is constructed. The end result will be a `results.log` file with many lines, each representing the results of distict tests, with perhaps different configuration parameters. For this reason, each line in the results file is configured with more information:

```
lead_condition = (OB_TOTAL_ENCOUNTERS > $(TEST_ENCOUNTERS))

pass_flag   = OB_TOTAL_COLLISIONS = 0
result_flag = MISSION_EVALUATED = true

report_column = form=obavoid
report_column = min_sep=$(SEP)
report_column = grade=$[GRADE]
report_column = collisions=$[OB_TOTAL_COLLISIONS]
report_column = near_misses=$[OB_TOTAL_NEAR_MISSES]
report_column = encounters=$[OB_TOTAL_ENCOUNTERS]
report_file = results.log
```

Now a set of tests can be launched, each varying the minimum separation distance between obstacles in the randomly generated obstacle field:

```
$ xlaunch.sh --sep=14 10
$ xlaunch.sh --sep=12 10
$ xlaunch.sh --sep=10 10
$ xlaunch.sh --sep=8 10
$ xlaunch.sh --sep=6 10
$ xlaunch.sh --sep=4 10
```

This should produce several lines of output in the file `results.log` similar to:

```
form=obavoid  min_sep=14  grade=pass collisions=0  near_misses=0  encounters=25
form=obavoid  min_sep=12  grade=pass collisions=0  near_misses=0  encounters=25
form=obavoid  min_sep=10  grade=pass collisions=0  near_misses=1  encounters=25
form=obavoid  min_sep=8  grade=pass collisions=0  near_misses=0  encounters=25
form=obavoid  min_sep=6  grade=pass collisions=0  near_misses=2  encounters=25
form=obavoid  min_sep=4  grade=pass collisions=0  near_misses=4  encounters=25
```

67.5 Configuration Parameters for pMissionEval

The following parameters are defined for pMissionEval. A more detailed description is provided in other parts of this section. Parameters having default values are indicated.

Listing 67.27: Configuration Parameters for pMissionEval.

fail_condition: A logic condition that must *not* be satisfied or otherwise the overall mission will be considered successful. Section 67.3.1.

fail_flag:	A flag to be publish once the result is known and is mission has been deemed to be not successful. Section 67.3.2.
lead_condition:	A logic condition that must be satisfied before the mission can be evaluated. Section 67.3.1.
pass_condition:	A logic condition that must be satisfied for the overall mission to be considered successful. Section 67.3.1.
pass_flag:	A flag to be publish once the result is known and is mission has been deemed to be successful. Section 67.3.2.
mission_form:	Set the name of the mission, e.g., "alpha", or "berta" etc., so the MISSION_FORM macro can be expanded in any reporting.
report_column:	Define a column of output in the in the report file. See Section 67.3.4.
report_file:	Name the report file to receive output of results. See Section 67.3.4.
report_line_format:	Determines the separator between report columns. By default it is white space, "white", but can also be set to us a comma, "csp".
result_flag:	A flag to be publish once the result is known. Section 67.3.2.

67.6 Publications and Subscriptions of pMissionEval

The interface for pMissionEval, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pMissionEval --interface or -i
```

67.6.1 Variables Published by pMissionEval

The publications of pMissionEval are mostly configurable in the user specified pass, fail and result flags. There are a few hard-coded publications:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 67.7.
- MISSION_LCHECK_STAT: See Section **??**.
- MISSION_RESULT: See Section **??**.
- MISSION_EVALUATED: See Section **??**.

67.6.2 Variables Subscribed for by pMissionEval

The subscriptions for pMissionEval mostly are dictated by the user-specified test conditions, i.e., the lead_condition, pass_condition and fail_condition parameters. Users may also specified any MOOS variable to be echoed in any user specified event flags, and in publications to a results file. There are a few hard-coded subscriptions:

- APPCAST_REQ: A request to generate and post a new appcast report, with reporting criteria, and expiration.
- DB_UPTIME: The amount of seconds that the local MOOSDB has been running.

- MISSION_HASH: The mission hash posting contains a few fields that need to be know to support a couple mission hash macros. See Section 67.3.5.
- APPNAME_PID: "ignore" ("none"). Section **??**.

67.7 Terminal and AppCast Output

The pMissionEval application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 28 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. The counter on the end of line 2 is incremented on each iteration of pMissionEval, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

The output in the below example comes from the example described in Section **??**.

Listing 67.28: Example terminal or appcast output for pMissionEval.

```
1  ================================================================
2  pMissionEval shoreside                                    0/0(25)
3  ================================================================
4  Overall State: pending
5  ============================================
6   logic_tests_stat: unmet_lead_cond: [ARRIVED=true]
7       result flags: 1
8         pass flags: 0
9         fail flags: 0
10        curr_index: 0
11       report_file: results.log
12
13 ============================================
14 Supported Macros:
15 ============================================
16
17 Macro         CurrVal
18 ------------  -------------
19 ENCOUNTER     0
20 MHASH_SHORT   GOLD-BABY
21 MHASH_UTC     1737549326.28
22 MISSION_FORM  01-colavd
23 WALL_TIME     0
24
25 ============================================
26 Logic Sequence:
27 ============================================
28
29 Index  Status   Cond  PFlag  FFLag
30 -----  -------  ----  -----  -----
31 0      current  1     0      0
32
33 ============================================
34 Reporting:
35 ============================================
36   report_file: results.log
37    report_col: m_form=$[MISSION_FORM]
```

```
38     report_col: 5
39     report_col: $[ENCOUNTER]
40     report_col: mhash=$[MHASH_SHORT]
41     report_col: utc=$[MHASH_UTC]
42     report_col: wall=$[WALL_TIME]
43    latest_line:
44
45  ================================================================
46  Most Recent Events (4):
47  ================================================================
48  [4.80]: Mail:DB_UPTIME
49  [2.78]: Mail:DB_UPTIME
50  [2.27]: Mail:MISSION_HASH
51  [0.76]: Mail:DB_UPTIME
```

The first few lines (4-11) provide a high-level snapshot of the current state. Line 4 shows the overall state of the test. Lines 13-23 show all macros available in posting flags or generating column output in a report file. Lines 25-31 show the prevailing logic sequence. A sequence may have several aspects (one per line in this table). Lines 33-43 show the configuration of the report file output. Line 43 shows the latest line written to this file. Line 45 is the start of the most recent events. For pMissionEval, for now, the events are just the received mail registered for this app.

All flags for all behaviors also have certain macros defined for event postings. Two such macros are `$[CTR1]` and `$[CTR2]`. They are counters that are incremented each time they are used in a posting. So if the cyle_flag and wpt_flag parameters are set in the waypoint behavior of the Alpha mission:

```
wpt_flag = WFLAG=$[CTR1]
cycle_flag = CFLAG=$[CTR1]
repeat = 1
end_flag = ARRIVED=true
```

The vehicle will traverse the five points twice, and the MOOS variables WFLAG and CFLAG should have the values 10 and 2 respectively, and ARRIVED should be set to `true`.

- It encapsulates mission evaluation to be defined by the pMissionEval configuration block. This allows automated scripts to treat a mission generically: launch it, monitor for its ending, and shut it down.
- It can facilitate multiple component evaluations in a single mission, distilling it down to a single pass/fail grade.
- It allows for custom configuration of a report file that facilitates easier post-processing for plotting result data.
-

67.7.1 Relation to uMayFinish

In an automated mission, some entity is monitoring one or more of the MOOS communities involved in the situation, typically the shoreside, for an event that indicates that the mission is complete and evaluations are completed and registered somewhere. After this point, the mission (all MOOS apps) are automatically shut down.

The `uMayFinish` application ma

68 pRealm: Integrated Scoping of the MOOSDB

68.1 Overview

The pRealm application is a tool to enable a shoreside multi-vehicle scope of the MOOSDB for all connected vehicles, as well as the shoreside MOOSDB. An instance of pRealm in each MOOS community, one per MOOSDB. It creates a shadow copy of the MOOSDB by registering for all known variables, i.e., those listed in the DB_RWSUMMARY published by the MOOSDB. The pRealm app only *listens* and keeps a copy of all the mail processed by the MOOSDB. Occasionally pRealm will receive and process a request to produce a report, a REALMCAST, showing the current values of a subset of known MOOS variables. Typically this subset is correlated to the subscriptions and publications of a particular app connected to the local MOOSDB. The typical usage scenario for pRealm is shown in Figure 204.

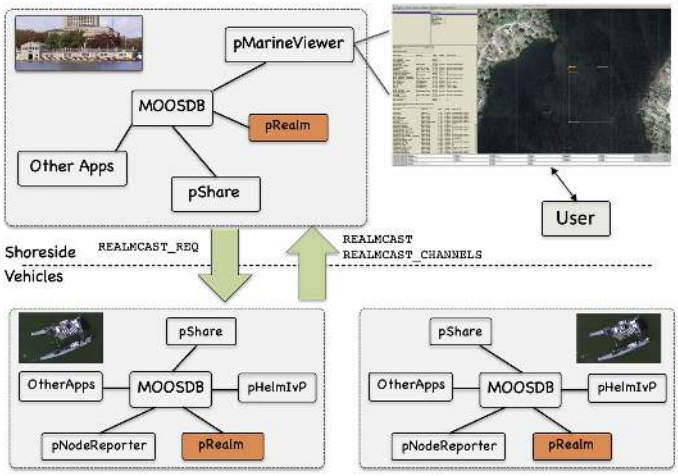

Figure 204: **Typical pRealm Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

The goal of this application is to enable the user, situated at a shoreside console, to select any deployed vehicle and scope the MOOSDB from a variety of vantage points. Initially this means selecting a vehicle and process and seeing the publications and subscriptions of that process. This is especially powerful for confirming the flow of information between vehicles or between vehicles and the shoreside. The pMarineViewer app has been substantially augmented to capitalize on a set of MOOS communities that *realm enabled*. If pRealm is not enabled, pMarineViewer and all other apps will be unaffected.

The implementation of pRealm is similar to and based on *appcasting* which precedes pRealm by roughly seven years. It is similar to appcasting in that a realmcast is only generated when there is a client, e.g. pMarineViewer, with an open realmcast window, pointed to a particular vehicle and particular channel. In this case, pRealm on the chosen vehicle will only respond with a generated report solely for the selected channel. Like appcasting, the realmcast rate generation is immune time the MOOS Time Warp and will be generated roughly once per second. This regime, like with

appcasting, minimizes bandwidth which may be limited in field operations.

Unlike appcasting, pRealm requires no augmentation to the participating apps. The only requirement is that an instance of pRealm is launched alongside all the apps otherwise launched in each MOOS community. No configuration is required. Although there are configuration parameters the user may adjust, the default parameters are fine in most cases.

68.2 Configuration Parameters for pRealm

The pRealm application may be configured with a configuration block within a MOOS mission file, typically with a `.moos` file suffix. Unlike most other MOOS apps, there will no warning generated if a configuration block is not provided. This is partly because the default values of parameters rarely need changing, and partly because not requiring a configuration block makes it even easier to add pRealm to existing missions. The following parameters are defined for pRealm.

Listing 68.29: Configuration Parameters for pRealm.

relcast_interval:	Time duration, in seconds, that must occur between the previously generated realmcast and a new one. The allowable range is [0.4, 15]. Values outside this range will be clipped to the range. The default value is 0.8 seconds.
summary_interval:	Time duration, in seconds, between auto-generated postings of REALMCAST_CHANNELS. The allowable range is [1, 10]. Values outside this range will be clipped to the range. The default value is 2 seconds.
wrap_length:	The number of characters, per line, when the user has opted to have the variable value field wrapped. Legal values are any integer value greater than zero. The default is 90.
trunc_length:	The number of characters that the value field of an output line will be truncated to, when the user has opted to enable truncated output. Legal values are any integer value greater than zero. It is recommended to choose a number that is a multiple of the wrap length. The default is 270.
msg_max_hist:	The number of MOOS mail messages held per variable. Currently multiple messages are only held for string messages. Legal values are any integer value greater than zero. The default is 10.

68.2.1 An Example MOOS Configuration Block

An example MOOS configuration block may be obtained from the command line with the following:

```
$ pRealm --example or -e
```

Listing 68.30: Example configuration of the pRealm application.

```
1  ============================================================
2  pHostInfo Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pRealm
```

```
 6  {
 7    AppTick   = 4
 8    CommsTick = 4
 9
10    relcast_interval = 0.8       //  [0.4, 15] Default is 0.8
11    summary_interval = 2.0       //  [1, 10] Default is 2
12    wrap_length = 90             //  [1,inf] Default is 90
13    trunc_length = 270           //  [1,inf] Default is 270
14    msg_max_history = 10         //  [1,inf] Default is 10
15  }
```

68.3 Publications and Subscriptions for pRealm

The interface for pRealm, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pHostInfo --interface or -i
```

68.3.1 Variables Published by pRealm

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- REALMCAST: Contains a realmcast report, showing the current values of a set of MOOS variables. Typically the set of variables is related to a particular MOOS app running in the same community as pRealm. This is a multi-line text report that has been serialized into a single string. It will be unpacked as a multi-line text report by the receiving client.
- REALMCAST_CHANNELS: A list channels associated with this instance of pRealm. Typically this list contains the names of MOOS apps running in this MOOS community, but may contain other user-configured channels.
- WATCHCAST: Contains a watchcast report, containing the the information on the most recent post for a particular single MOOS variable.

68.3.2 Variables Subscribed for by pRealm

The pRealm application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- REALMCAST_REQ: A request to generate publications to REALMCAST for a specified channel, for specified duration.
- DB_RWSUMMARY: A report generated by the MOOSDB listing, for each app connected to the MOOSDB, the set of variables subscribed for by each app, and the set of variables observed to be published by each app.

Node that pRealm will also subscribe for all variables discovered from the contents of DB_RWSUMMARY publications.

68.3.3 Command Line Usage of pRealm

The pRealm application is typically launched with pAntler, along with a group of other modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ pRealm --help or -h
```

Listing 68.31: Command line usage for the pRealm tool.

```
Usage: pRealm file.moos [OPTIONS]

Options:
  --alias=<ProcessName>
      Launch pRealm with the given process
      name rather than pRealm.
  --example, -e
      Display example MOOS configuration block
  --help, -h
      Display this help message.
  --HOSTIP=<HostIP>
      Force the use of the given IP address as the reported IP
      address ignoring any other auto-discovered IP address.
  --interface, -i
      Display MOOS publications and subscriptions.
  --version,-v
      Display the release version of pRealm.

Note: If argv[2] is not of one of the above formats
      this will be interpreted as a run alias. This
      is to support pAntler launching conventions.
```

68.4 Structure of RealmCast Reports

The primary output of pRealm is a realmcast, a posting to the MOOS variable REALMCAST. If pRealm is running in a vehicle MOOS community, then the realmcast is typically shared to the shoreside community. A realmcast is only generated on demand, to ensure minimal bandwidth, keeping in mind scenarios with large numbers of vehicles and limited physical communications bandwidth. In this section the structure of the realmcast message is described, how this message is serialized, and how the content of the realmcast message may be customized at run time to suit user preferences.

68.4.1 The RealmCast Structure

The structure of realmcast is simply a list of strings. While there is a distinct formatting of the report, the end result is the list of strings with inserted white space. The assumption is that whatever app is rendering the report will be using a fixed-width character output, which will preserve the columns and spacing of the report. An example is shown below.

```
=====================================
pProxonoi                      ben (33)
=====================================
Subscriptions
=====================================
Variable        Source          Time    Comm       Variable
--------------  -------------   ------  ---------  ------------------------------------------------------
APPCAST_REQ     uMAC_5533       111.44  shoreside  node=all,app=all,duration=3.0,key=uMAC_5533:app,thresh=
NAV_X           uSimMarineX     111.69  ben        5408
NAV_Y           uSimMarineX     111.69  ben        -1725
NODE_REPORT     uFldNodeComms   39.22   shoreside  NAME=abe,X=5327,Y=-1769,SPD=0,HDG=287,TYPE=kayak,COLOR=
PROX_CLEAR      -               never   -          -
PROX_POLY_VIEW  -               never   -          -

=====================================
Publications
=====================================
Variable             Source           Time    Comm  Variable
-------------------  ---------------  ------  ----  ------------------------------------------------------
APPCAST              uFldNodeBroker   26.28   ben   formatted report
PPROXONOI_ITER_GAP   pProxonoi        111.43  ben   1.011477
PPROXONOI_ITER_LEN   pProxonoi        111.43  ben   0.000436
PPROXONOI_STATUS     pProxonoi        110.93  ben   AppErrorFlag=false,Uptime=112.009,cpuload=0.1298,memory
PROXONOI_PID         pProxonoi        -0.42   ben   63498
PROXONOI_POLY        pProxonoi        111.43  ben   active=false,label=vpoly_ben
PROXONOI_REGION      pProxonoi        110.43  ben   pts={-500,2300:-3500,-1000:-3500,-4600:3100,-4600:8400,
```

The above example output is typical of the default format used when pRealm is generating a report for a particular application, the pProxonoi app in this case. On the top line, the app name is listed on the left, with the vehicle name on the right. The number in the parentheses is a counter showning how many reports for this app have been generated.

The body of the report shows the set of variables subscribed for by this application, followed by the variables published by this application. For each variable, the variable name, source, time of the publication, community and variable contents are shown, similar to other scoping utilities like uXMS.

68.4.2 Serialization of the RealmCast Structure

As mentioned above in Section 68.4.1, a realmcast message is comprised primarily of a list of strings. The entire message is serialized into a single string for posting to the MOOSDB. The structure of this message is:

```
node=<nodename>!Q!proc=<procname>!Q!count=<num>!Q!line!Z!line!Z!line!Z!....
```

It is comprised of four main components, each separated by `"!Q!"`:

- The *node* is typically the name of the vehicle, or `"shoreside"` if this instance of pRealm is running in the shoreside community.
- The *proc* is typically the name of the application related to this report. A realmcast may be custom configured to contain information not strictly correlated with an app. In such a case the *proc* name is the name used for this custom configuration.
- The *count* represents the number of realmcast reports generated for this particular *node* and *proc* combination.
- The *line* components represent each line in the report, separated by the `"!Z!"` separator.

The serialization methods are defined on the C++ class `RealmCast`. There is a function defined to convert from the RealmCast class to a string:

```
RealmCast relcast;
... structure populated

string realmcast_report = relcast.get_spec();
```

And a function convert from a string to a RealmCast class instance.

```
string realmcast_report;
... string otherwise populated

RealmCast relcast = string2Realmast(realmcast_report);
```

Serialization is handled internally by pRealm and deserialization is handled internally by the client application, e.g., pMarineViewer.

68.4.3 When RealmCast Reports are Generated

The default state of pRealm is to be not generating any realmcast messages. Messages are only generated if there is a client requesting this information. A request comes in the form of the MOOS variable REALMCAST_REQ. Minimally this message specifies the pRealm channel of interest and the duration of time into the future during which it hopes to receive responding realmcast messages. The basic idea is shown in Figure 205.

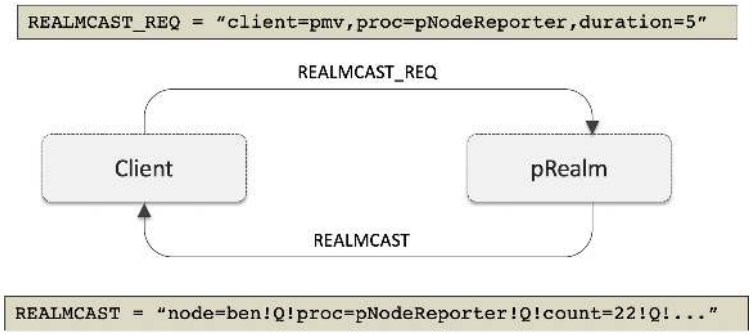

Figure 205: **RealmCast Request:** A request is generated by a client, specifying a proc (i.e., channel or app), and a duration. For the next period in time specified by the duration, pRealm will produce realmcast reports at a fixed interval.

Note that if the client suddenly quits, or turns its attention elsewhere, after the short duration, pRealm, resumes being quiet. Note also that the rate of publications to REALMCAST by pRealm is determined by the pRealm configuration parameter relcast_interval. This interval by default is 0.8 seconds. So, in the above example, a *single* request with a duration of five seconds should result in six realmcast publications. A typical client, if interested in receiving a steady stream of realmcast reports over a period longer than five seconds, would simply send requests at a steady rate somewhat faster than the duration specified in each request. Each newly received request will

reset the duration.

68.5 Altering the RealmCast Format to Suit the User

The default format for a typical realmcast report is shown in Section 68.4.1. There are seven simple ways to modify the output to further accommodate the preferences of a user. Each modification is a simple on/off state that can be toggled by the client application. As `pMarineViewer` is such a client, it has seven on-screen buttons for toggling the preferences. The seven methods are:

- Omission of the variable Source column
- Omission of the variable Community column
- Omission of the Subscriptions (top) block
- Masking out certain variables (described in Section 68.5.2)
- Wrapping the output of the variable Value column
- Truncating the output of the variabe Value column
- Showing time stamps in UTC format

Each of the above seven modifications come as part of the client's request for a realmcast report. A keyword is associated with each modification, and the absence of the keywork indicates that the modification is not requested. For example, a realmcast request with the Source and Community columns suppressed would look something like:

```
REALMCAST_REQ = client=pmv,duration=5,nosrc,nocomm,proc=pProxonoi
```

The keywords for the seven modifications are: `nosrc`, `nocom`, `nosub`, `mask`, `wrap`, `trunc` and `utc` respectively. Each incoming request sets the state for each of these mods. For example if the next request after the above one were:

```
REALMCAST_REQ = client=pmv,duration=5,nocomm,proc=pProxonoi
```

then the next outoing realmcast would no longer suppress the Source column for each variable.

68.5.1 Modifying the Column Output Based on Source and Community

By omitting the Source and Community columns the original output from Section 68.4.1 would look like that below, with both columns removed in both the Subscriptions block (top) and the Publications block (bottom):

```
=====================================
pProxonoi                      ben (33)
=====================================
Subscriptions
=====================================
Variable        Time    Value
--------------  ------  ------------------------------------------------------
APPCAST_REQ     111.44  node=all,app=all,duration=3.0,key=uMAC_5533:app,thresh=run_warning
NAV_X           111.69  5408
NAV_Y           111.69  -1725
NODE_REPORT      39.22  NAME=abe,X=5327,Y=-1769,SPD=0,HDG=287,TYPE=kayak,COLOR=dodgerblue,MODE=PARK,ALLSTO
PROX_CLEAR      never   -
PROX_POLY_VIEW  never   -

=====================================
Publications
=====================================
Variable            Time    Value
------------------  ------  ------------------------------------------------------
APPCAST             26.28   formatted report
PPROXONOI_ITER_GAP  111.43  1.011477
PPROXONOI_ITER_LEN  111.43  0.000436
PPROXONOI_STATUS    110.93  AppErrorFlag=false,Uptime=112.009,cpuload=0.1298,memory_kb=1768,memory_max_kb=
PROXONOI_PID        -0.42   63498
PROXONOI_POLY       111.43  active=false,label=vpoly_ben
PROXONOI_REGION     110.43  pts={-500,2300:-3500,-1000:-3500,-4600:3100,-4600:8400,2300},label=prox_opregi
```

The modified output simply allows more of the righthand Value columm to be more visible to the user.

68.5.2 Modifying the Row Output By Masking and Dropping Subscriptions

Notice the last two lines of Subscriptions section of the above output. These two variables have never been written to. They are included in the output because pRealm noticed in the DB_RWSUMMARY content that this particular app subscribes to those two variables. And at times it is helpful to see exactly the list of variables subscribed to for a given app. For some apps, however, these virgin variables can take big chunk of the report space, and it may be better to mask them out. This can be achieved by attaching the `mask` keyword to the realmcast request. The resulting output is shown below, with the two lines dropped.

```
=======================================
pProxonoi                      ben (33)
=======================================
Subscriptions
=======================================
Variable        Time    Value
--------------  ------  ------------------------------------------------------
APPCAST_REQ     111.44  node=all,app=all,duration=3.0,key=uMAC_5533:app,thresh=run_warning
NAV_X           111.69  5408
NAV_Y           111.69  -1725
NODE_REPORT      39.22  NAME=abe,X=5327,Y=-1769,SPD=0,HDG=287,TYPE=kayak,COLOR=dodgerblue,MODE=PARK,ALLSTO

=======================================
Publications
=======================================
Variable            Time    Value
------------------  ------  ------------------------------------------------------
APPCAST             26.28   formatted report
PPROXONOI_ITER_GAP  111.43  1.011477
PPROXONOI_ITER_LEN  111.43  0.000436
PPROXONOI_STATUS    110.93  AppErrorFlag=false,Uptime=112.009,cpuload=0.1298,memory_kb=1768,memory_max
PROXONOI_PID        -0.42   63498
PROXONOI_POLY       111.43  active=false,label=vpoly_ben
PROXONOI_REGION     110.43  pts={-500,2300:-3500,-1000:-3500,-4600:3100,-4600:8400,2300},label=prox_o
```

Furthermore, for certain apps, the number of variable subscriptions can be huge, and the user may be solely interested in monitoring one or more variable publications. The Subscriptions section can be omitted entirely by attaching the `nosubs` keyword to the realmcast request, resulting in the above output being further reduced to:

```
=======================================
pProxonoi                      ben (33)
=======================================
Subscriptions
=======================================
Variable        Time    Variable
--------------  ------  ------------------------------------------------------
APPCAST_REQ     111.44  node=all,app=all,duration=3.0,key=uMAC_5533:app,thresh=run_warning
NAV_X           111.69  5408
NAV_Y           111.69  -1725
NODE_REPORT      39.22  NAME=abe,X=5327,Y=-1769,SPD=0,HDG=287,TYPE=kayak,COLOR=dodgerblue,MODE=PARK,ALLSTO
PROX_CLEAR      never   -
PROX_POLY_VIEW  never   -
```

68.5.3 Modifying the Content of Very Large String Publications

Occasionally an app will publish very long string messages. In many cases, just seeing the first part of the message is sufficient for a user monitoring the system. As with the NODE_REPORT posting in the example above, the last part of the variable value is simply cut off in the output presented to the user. Rather than cutting off the line, it can be wrapped instead, by including the `wrap` tag in the realmcast request. Wrapping is shown in the example below.

```
=====================================
pProxonoi                      ben (33)
=====================================
Subscriptions
=====================================
Variable        Time    Value
--------------  ------  ------------------------------------------------------
APPCAST_REQ     111.44  node=all,app=all,duration=3.0,key=uMAC_5533:app,thresh=run_warning
NAV_X           111.69  5408
NAV_Y           111.69  -1725
NODE_REPORT      39.22  NAME=abe,X=5327,Y=-1769,SPD=0,HDG=287,TYPE=kayak,COLOR=dodgerblue,MODE=PARK,
                        ALLSTOP=ManualOverride,INDEX=81,TIME=3216098849.96,LENGTH=4
PROX_CLEAR      never   -
PROX_POLY_VIEW  never   -

=====================================
Publications
=====================================
Variable            Time    Value
------------------  ------  ------------------------------------------------------
APPCAST             26.28   formatted report
PPROXONOI_ITER_GAP  111.43  1.011477
PPROXONOI_ITER_LEN  111.43  0.000436
PPROXONOI_STATUS    110.93  AppErrorFlag=false,Uptime=112.009,cpuload=0.1298,memory_kb=1768,memory_max
                            _kb=1768,
PROXONOI_PID        -0.42   63498
PROXONOI_POLY       111.43  active=false,label=vpoly_ben
PROXONOI_REGION     110.43  pts={-500,2300:-3500,-1000:-3500,-4600:3100,-4600:8400,2300},label=prox_o
                            pregion
```

By default, the wrapping is done in 90 character increments. This can be changed with the pRealm configuration parameter, wrap_length.

Finally, in some cases wrapping may not be enough. If the string is thousands of characters long, it will dominate the realmcast output and make it hard to read anything else. The user has the option of also truncating the variable string value, by default, to 270 characters. This can be modified with the trunc_length parameter. To make the most of each line of output, it is recommended to set the trunc_length to be a multiple of the wrap_length.

68.5.4 Modifying the Time Format to UTC

Normally the posted time stamp is relative to the "start of the mission". There is no consensus on then the start of the mission occurs, although one could argue that it is the very instance the MOOSDB starts. However, in multi-vehicle missions, each MOOSDB starts at a slightly different time. In the case of pRealm, the start time used for calculating timestamps is simply the UTC time when pRealm started. This vagary is usually tolerated because we are simply trying to get a feel for the rough time or age of a variable post, and the nice small numbers help get a quick sense. However, at times, more precision may be preferred. The user has the option of requesting timestamps to be posted in absolute UTC time, i.e., the number of seconds since January 1st, 1970. In this way, all postings from all vehicles will be referencing the same start time, regardless of when the MOOSDB or other processes began.

A client application simply needs to add the `utc` component to the end of a realmcast request as such:

```
REALMCAST_REQ = client=pmv,duration=5,utc,proc=pProxonoi
```

68.5.5 How Does the User Actually Modify Output?

As described above, modification to the realmcast report content is achieved by adding tags to the incoming REALMCAST_REQ messages. For example, the Source and Community columns are removed when the request contains the `nosrc` and `nocom` tags:

```
REALMCAST_REQ = client=pmv,duration=5,nosrc,nocomm,proc=pProxonoi
```

And virgin variables and string content can be wrapped with the `mask` and `wrap` tags:

```
REALMCAST_REQ = client=pmv,duration=5,mask,wrap,proc=pProxonoi
```

In general the user does not need to be involved in formatting string messages with these tags. The pMarineViewer app is a client that interacts with the user and contains toggle buttons for the above content modifiers. Figure 206 shows the lefthand bottom corner of the pMarineViewer window when realmcast output is being viewed. (Also see Figure 210.) Each button is a toggle button.

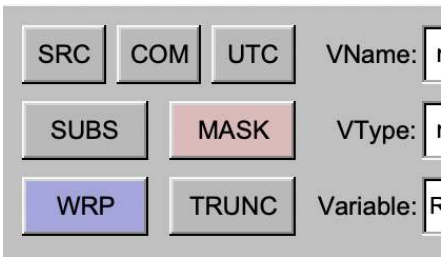

Figure 206: **Content Modifier Buttons:** The pMarineViewer app acts as a client requesting and receiving realmcast reports. The modifier buttons will toggle the seven modifiers affecting realmast formatting.

When pMarineViewer is in the mode to present realmcast content, it will generate a steady stream of REALMCAST_REQ messages. As the above seven content buttons are toggled, the next outgoing request message will be modified with the proper content tags.

68.6 WatchCasting: A Special Type of RealmCast

In addition to the outgoing REALMCAST message, pRealm supports a second type of message called a watchcast, posted in the variable WATCHCAST. This message contains the information about the most recent post to *one* MOOS variable. Watchcasts are generated when the most recent realmcast request names a set of MOOS variables. Here is a normal realmcast request discussed above, requesting the pub/sub information for pHelmIvP:

```
REALMCAST_REQ = client=pmv,duration=5,proc=pHelmIvP
```

And here is a realmcast request that specifies a particular set of MOOS variables:

```
REALMCAST_REQ = client=pmv,duration=5,vars=DEPLOY:RETURN:STATION_KEEP
```

When pRealm has been asked to produce this kind of content, it produces it in a watchcast message, one per variable. The idea is shown in Figure 207.

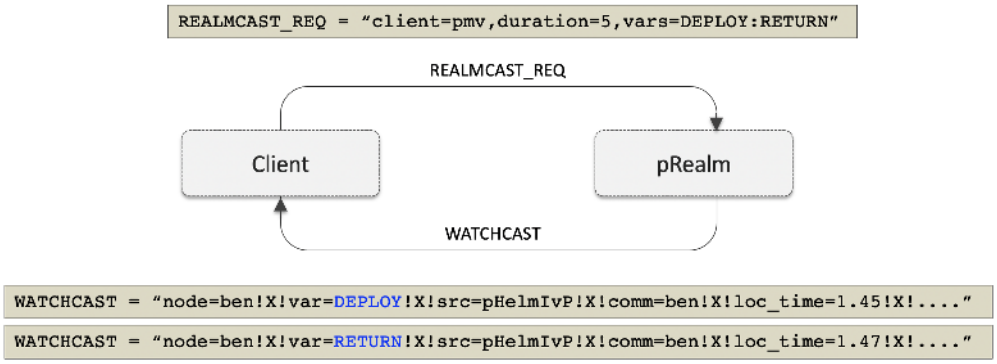

Figure 207: **WatchCast Request:** A request is generated by a client, specifying a set of variables, and a duration. For the next period in time specified by the duration, pRealm will produce a watchcast report for each variable, each time the variable changes, but no more frequent than a fixed interval.

Watchcasting allows the client to configure a cluster of variables that may be important to the user. These variables may be involved in several different MOOS applications and thus hard to visualize and monitor. Clients of pRealm like pMarineViewer and uMACView are able to specify watch clusters and view a single report similar to:

```
Node    DEPLOY     RETURN    STATION_KEEP
----    ------     ------    ------------
abe     true       false     false
ben     true       false     false
cal     true       false     false
deb     true       false     false
eve     true       false     false
fin     true       false     false
gus     true       false     false
hal     true       false     false
```

These may be the key variables in the autonomy of a particular mission, or they could be health montitoring variables for deployed platforms:

Node	BATT_VOLTAGE	CPU_TEMP	GPS_SATELLITES
abe	17.2	111.2	11
ben	16.9	112.0	8
cal	17.4	111.9	9
deb	17.4	108.7	10
eve	17.0	101.2	11
fin	15.8	105.5	11
gus	16.4	114.2	9
hal	16.1	105.6	10

Since multiple pRealm clients may be run simultaneously, e.g., pMarineViewer and one or more instances of uMACView, then the user could have several such tables viewable. In multi-robot vehicle deployments, this may be extremely useful. Unlike appcast content, the above content is customizable by the user without any adjustments to code. Watch clusters are configured in pMarineViewer or uMACView configuration files. See the documentation for these app for more information, [?], [?].

68.7 How pRealm Informs Potential Clients

We have established how clients of pRealm may request reports on a given channel, Figure 205. But how does a client like pMarineViewer know there exists a pRealm to query, and which channels may be queried? To accomplish this, pRealm publishes its key information periodically for clients to discover. At a rate of once per five seconds, pRealm publishes to the MOOS variable REALMCAST_CHANNELS a report that identifies the name of the node/vehicle, and all available channels for querying. This idea is conveyed below.

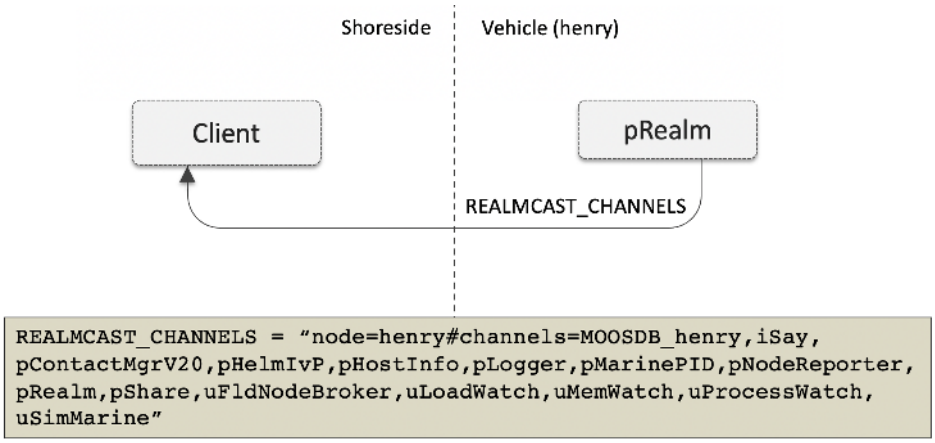

Figure 208: **RealmCast Channels:** The pRealm app periodically posts information about itself. This information is used later by clients to request realmcast reports from specific nodes and specific channels.

By default, these messages are posted once every 5 seconds. This value can be changed with the summary_interval parameter. There are two exceptions to this interval. For the first minute after launch, the summary will be posted every half second. The second exception is applied when a new app has been detected. The set of apps participating in the local MOOS community normally doesn't change after the initial startup. But if a new app is later detected by pRealm, the REALMCAST_CHANNELS

variable will be published immediately with this new information.

A final note: the time interval between summaries is real time, not time warp time.

68.8 Terminal and AppCast Output

The pRealm application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 32 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. The counter on the end of line 2 is incremented on each iteration of pRealm, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 68.32: Example terminal or appcast output for pRealm.

```
 1  ===============================================================
 2  pRealm gilda                                          0/0(3074)
 3  ===============================================================
 4  Configuration:
 5  ------------------------------------------
 6    RelCast Interval: 0.55
 7    Summary Interval: 7.60
 8    Wrap Length:  90
 9    Trunc Length: 270
10    Max Msg Hist: 10
11
12  MOOS Community State:
13  ------------------------------------------
14    MOOSDB Name: MOOSDB_gilda
15     Known Apps: 15
16     Known Apps: MOOSDB_gilda,iSay,pContactMgrV20,pHelmIvP,
17                 pHostInfo,pLogger,pMarinePID,pNodeReporter,
18                 pRealm,pShare,uFldNodeBroker,uLoadWatch,
19                 uMemWatch,uProcessWatch,uSimMarine
20    Total SVars: 196
21    Total PVars: 137
22    Unique Vars: 206
23    UTC Time:    12874301098.43
24    Local Time:  1605.84
25    Summaries:   40
26
27  Recent RealmCasts or WatchCasts:
28  ------------------------------------------
29    Total RealmCasts: 238
30    Total WatchCasts: 6
31  ------------------------------------------
32  Count     Time  Client  Content
33  -----  -------  ------  ----------------------------------------
34  (165)  1604.28  pmv     pHelmIvP
35  (47)    829.04  pmv     iSay
36  (1)     424.30  pmv     vars=DEPLOY
37  (1)     137.69  pmv     vars=AVOID,DEPLOY,LOITER,RETURN,STATION_KEE
38  (26)    132.98  pmv     iSay
39
```

```
40  Currently Active Clients:
41  -----------------------------------------
42  Client  Active  Content
43  ------  ------  --------
44  pmv     21.37   pHelmIvP
45
46  ================================================================
47  Most Recent Events (8):
48  ================================================================
49  [831.62]: Client pmv, new pipe: pHelmIvP
50  [533.97]: Client pmv, new pipe: iSay
51  [386.22]: Client pmv, new pipe: vars=DEPLOY
52  [336.66]: Client pmv, new pipe: vars=AVOID
```

Lines 6-10 of the output show the current values of the five pRealm configuration parameters listed in Section 68.2.

Lines 12-22 relate state of the MOOSDB, including the name of the MOOSDB which is always the community name at the end of the MOOSDB_ prefix. The number of and list of know apps connected to the MOOSDB are shown in lines 15 and 16. Line 20 shows the number of MOOS variables involved in a subscription by at least one app. Line 21 shows the number of unique MOOS variable publications. Line 22 shows the number of unique variables known to the MOOSDB involved in either a subscription or publication. Line 33 shows the current UTC time, with the time warp multiplier applied. Line 24 shows the local time, i.e., the time since pRealm was started. Line 25 shows the total number of realmcast summaries posted, to the variable REALMCAST_CHANNELS.

Lines 27-38 indicate the most recent realmcasts or watchcasts. First the total of each is shown on lines 29 and 30. Lines 34-38 show the five most recent content settings. The first column, Count, shows the number of successive outgoing messages. The Time column shows the local time of the most recent message in that group. The Client column shows which app is requesting the content. This will show as "pmv" when pMarineViewer is the client. Finally, the Client column indicates the content of the outgoing message. When the content begins with "vars=", this indicates that it is a watchcast.

Lines 40-44 show the status of active clients. A client is active if pRealm has recently received a realmcast request. As discussed earlier, each request has a duration. The duration implies a time remaining before the request expires. This time is shown in column two. Column three shows the content being requested in the realmcast request.

Lines 46-56 contain the typical recent events block for all appcasting apps. In the case of pRealm, an event is posted each time a realmcast request has been received that contains a change in requested content.

68.9 A Preview of Using pRealm Information within pMarineViewer

In most cases the user won't give another thought about pRealm beyond simply including it in the pAntler launch lists. The user experience is through pMarineViewer. Initially this is the only app that is able to handle realmcasts and present them to a user. So here is a quick preview of the pMarineViewer experience.

Normally when the a mission is launched with the pMarineViewer the window shows the three "infocast" panes shown on the left. The infocast panes are either showing appcast content, or realmcast content. The infocast panes show appcast content by default. Unless the user changes the default color schemes, appcasting is distinguished by an indigo background as in Figure 209.

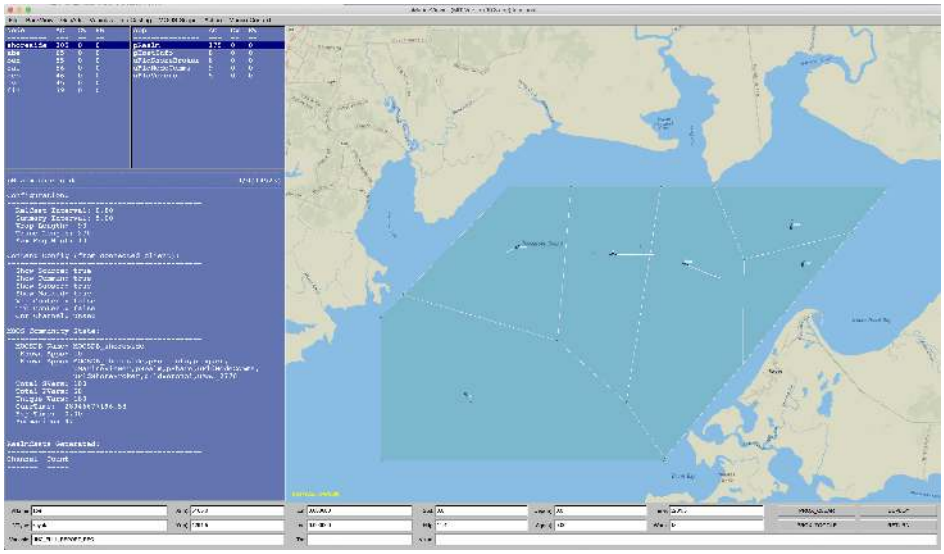

Figure 209: **The pMarineViewer AppCasting Mode:** When pMarineViewer starts, normally it is in appcasting mode, with the left three appcasting panes rendered in indigo showing appcast content in the lower pane.

The appcasting panes let the user select the node in the upper left pane, e.g., the shoreside or one of several vehicles. The upper right pane lets the user select the MOOS app running on the selected node. The bottom pane shows the appcast content for the node and app selected in the top two panes.

The realmcast mode is very similar. To toggle between modes, the 'a' key is used. Unless the user changes the default color schemes, the realmcasting is distinguished by a beige background as in Figure 210. As in the appcasting mode, the top two panes offer virtually the same options based on nodes and processes. The bottom middle pane switches however to show realmcast content.

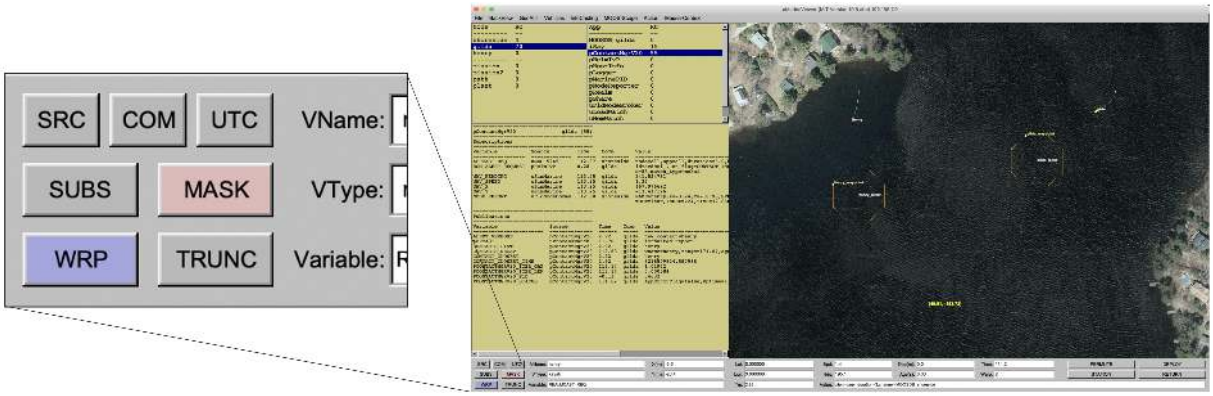

Figure 210: **The pMarineViewer RealmCasting Mode:** The pMarineViewer user may toggle into realmcasting mode using the 'a' key, with the left three panes rendered in beige.

When in realmcast mode, pMarineViewer will display the seven buttons shown above. These buttons allow the user to modify the content modes discussed in Section 68.5.

68.10 Additional Notes

Some additions planned for the future:

- General masking: Currently masking is only done on virgin variables. The plan is to also allow the user to mask out entire apps, or a list of variables for an app, or list of variables across all apps.
- Configurable channels: Currently the only channels are connected apps, plus an artficial channel for the MOOSDB. Like uXMS, the plan is to allow the user to define their own channel with a custom list of variables to watch.
- Global channel: Currently there is no channel that simply lists all variables.
- Messaging channel: A channel specifically for monitoring inter-vehicle messaging.

69 uMayFinish: A command-line MOOS Tool for Waiting for Mission Completion

69.1 Overview

uMayFinish is typically a terminal-launched MOOS app launched within a shell script, e.g., xlaunch.sh. It will connect to a MOOS community and monitor for a completion event or timeout based on DB_UPTIME. When completed, it simply exits. Presumably to allow the executing script to proceed to a next phase. For example a script could proceed to bring down the MOOS community for the mission it was monitoring.

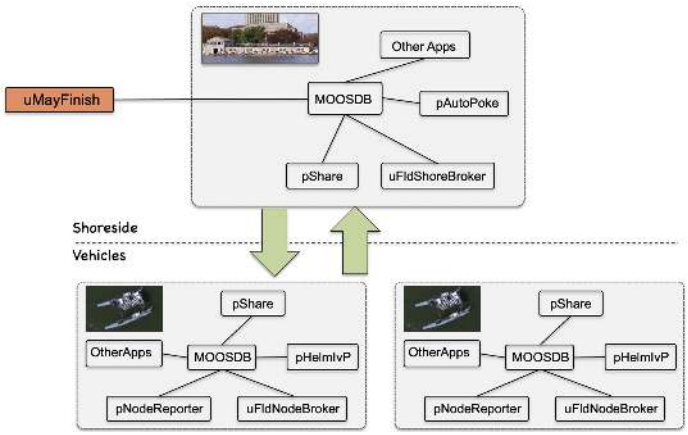

Figure 211: **Typical uMayFinish Topology:** The uMayFinish app is typically not part of the initial MOOS community of either the shoreside or vehicles. Typically, shortly after everything is launched, the uMayFinish app is launched and connects to the shoreside MOOSDB. It waits for a either a time limit to be reached, or a specified MOOS variable to have a specified value. Then the app disconnects. The disconnection may be followed by an action to shut down all MOOS communities.

Consider the following pseudo-code snippet:

```
pAntler mission.moos &
uMayFinish --max_time=600 mission.moos
ktm
```

In the first line, the mission is launched. It is launched in the background, so control proceeds immediately to the next line. Then uMayFinish launches and connects to the same MOOSDB since it is invoked with the same mission file. It is *not* launched in the background, so control will stop here until the uMayFinish completes. After 10 minutes (600 seconds), it will disconnect from the MOOSDB, and exit. Control passes to the next line which invokes ktm (kill-the-moos), to bring down the entire mission.

Something similar is used in xlaunch.sh, which is meant to be a general way for auto-launching missions with provisions for time-conditioned or event-conditioned exiting. In the above snippet, the maximum time was passed on the command line. The maximum time can also be specified in the uMayFinish configuration file in the mission file, as described in Section 69.2.

69.2 Configuration Parameters for uMayFinish

The uMayFinish application may be configured with a configuration block within a MOOS mission file, typically with a `.moos` file suffix. The following parameters are defined for uMayFinish.

Listing 69.33: Configuration Parameters for uMayFinish.

finish_var:	A MOOS variable.
finish_val:	A prescribed value for the MOOS variable.
max_db_uptime:	The number of seconds, found in the MOOSDB published variable `DB_UPTIME`, after which this app will exit.

69.2.1 An Example MOOS Configuration Block

An example MOOS configuration block may be obtained from the command line with the following:

```
$ uMayFinish --example or -e
```

Listing 69.34: Example configuration of the uMayFinish application.

```
1  ============================================================
2  uMayFinish Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = uMayFinish
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   finish_var = MISSION_EVALUATED  // Default
11   finish_val = true               // Default
12
13   max_db_uptime = 3600            // Default is -1 (off)
14 }
```

69.3 Publications and Subscriptions for uMayFinish

The interface for uMayFinish, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uMayFinish --interface or -i
```

69.3.1 Variables Published by uMayFinish

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.

69.3.2 Variables Subscribed for by uMayFinish

The uMayFinish application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- DB_UPTIME: The number of seconds since the MOOSDB has been launched. If a mission has been auto-started shortly after launch (e.g., using pAutoPoke), then this value is a good approximation for measuring how long the mission has been running. Note, this subscription will only be made if the configuration parameter max_db_uptime is set. This param is set to -1 by default, meaning there is no maximum time.

Node that uMayFinish will also subscribe for the variable named in the configuration parameter finish_var, if provided.

69.3.3 Command Line Usage of uMayFinish

The uMayFinish application is typically launched with pAntler, along with a group of other modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uMayFinish --help or -h
```

Listing 69.35: Command line usage for the uMayFinish tool.

```
1  Usage: uMayFinish file.moos [OPTIONS]
2
3  Options:
4    --alias=<ProcessName>
5        Launch uMayFinish with the given process
6        name rather than uMayFinish.
7    --example, -e
8        Display example MOOS configuration block
9    --help, -h
10       Display this help message.
14   --interface, -i
15       Display MOOS publications and subscriptions.
16   --version,-v
17       Display the release version of uMayFinish.
18   --max_time=1200
19       Maximum DB_UPTIME before exiting.
20   --web,-w
21       Open browser to: https://oceanai.mit.edu/ivpman/apps/uMayFinish
22
23 Note: If argv[2] is not of one of the above formats
24       this will be interpreted as a run alias. This
25       is to support pAntler launching conventions.
```

69.4 Terminal and AppCast Output

Some useful information is published by uMayFinish to the terminal on every iteration. An example is shown in Listing 36 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. The counter on the end of line 2 is incremented on each iteration of uMayFinish, and also serves as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 69.36: Example terminal or appcast output for uMayFinish.

```
 1  ============================================================
 2  uMayFinish shoreside                                  0/0(67)
 3  ============================================================
 4  Config:
 5    Finish Var: MISSION_EVALUATED
 6    Finish Val: true
 7    Max DBTime: -1
 8
 9  State:
10    DB_UPTIME: n/a
11    MISSION_EVALUATED:
```

Lines 4-7 of the output show the user configuration. Lines 10 and 11 show the current state of the app. Note the DB_UPTIME output on line 10 indicates "n/a" since no max time was set for this launch (line 7).

When the exit criteria is met, the appcast output may change as below. Note the only change is on line 11, where the empty value for the current value of MISSION_EVALUATED in the listing above, is replaced with MISSION_EVALUATED: true in the listing below. Since this satisfies the exit criteria, this will be the very last appcasting output to either the terminal or seen in pMarineViewer.

Listing 69.37: Example output as uMayFinish *finishes..*

```
 1  ============================================================
 2  uMayFinish shoreside                                0/0(341)
 3  ============================================================
 4  Config:
 5    Finish Var: MISSION_EVALUATED
 6    Finish Val: true
 7    Max DBTime: -1
 8
 9  State:
10    DB_UPTIME: n/a
11    MISSION_EVALUATED: true
```

When uMayFinishApp exits due to satisfying the finish variable criteria, it will return with a value of 0. When uMayFinishApp exits due to exceeding a maximum time limit, the exit value is 1 instead.

70 pObstacleMgr: Managing Vehicle Belief State of Obstacles

70.1 Overview

The pObstacleMgr is designed to reason about obstacles and provide coordinated alerts and updates to the helm for spawning and adjusting obstacle avoidance behaviors. The obstacle manager reasons about *given* obstacles, with prior known locations, loaded at launch time and may include buoys, rocky outcrops, bridge pylons and so on. It also reasons about *sensed* objects that may be derived from LIDAR point clouds, or other sensor sources. The obstacle manager will manage both the mission-loaded and dynamic incoming data to maintain a single list of obstacles. Depending on the robot range to the obstacle, the obstacle manager will produce alerts for coordination with obstacle avoidance behaviors. And in the case of dynamically sensed obstacles, it will continually update the position and shape of the obstacle to previously spawned obstacle avoidance behaviors.

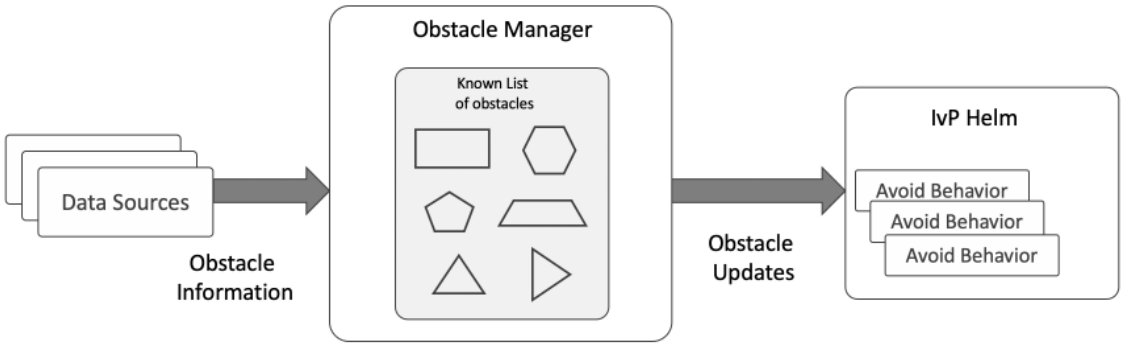

Figure 212: **The Obstacle Manager**: Obstacle information from a variety of sources is received by the obstacle manager, which maintains a list of known obstacles. This is modified with time both in terms of updating the obstacle locations as well as deleting stale obstacles. Continuous updates are posted and consumed by the helm to enable the spawning of avoidance behaviors.

The obstacle manager exists and sits between the sensor stream and the helm for two reasons. (1) This arrangement allows the helm to remain passively postured with respect to behavior spawning of obstacle avoidance behaviors. The helm will spawn behaviors based on incoming events from the obstacle manager. This is implemented in the helm in a manner that is consistent with any other type of behavior or sensor stream. Nothing special was implemented in the helm to handle obstacle information, other than the `AvoidObstacle` behavior. (2) The nature of the obstacle manager may change over time as different sensors, information sources, or outlier rejection algorithms are available. None of these changes will require a change to the helm or its behaviors.

70.2 Using the Obstacle Manager

To use the obstacle manager there are a few steps and considerations.

- The pObstacleMgr app must be run on the vehicle, by adding it to the Antler block for the vehicle. An example configuration block is given in Section 70.4.1.
- Obstacle information must be fed to the obstacle manager from either one of three sources. Either it (a) arrives from a sensor-based source as labeled points as decribed in Section 70.2.1,

or (b) arrives as convex polygon obstacle mail message in the variable GIVEN_OBSTACLE, or (c) the obstacle size and position information is provided at launch time from a set of obstacles known a priori, listed in the configuration file with the parameter given_obstacle, as described in Section 70.2.1.

- The helm is configured to use an obstacle avoidance behavior in a templating mode, allowing new behaviors to spawn based on output from the obstacle manager. See Section 70.2.2.

70.2.1 Obstacle Manager Input Sources

The obstacle manager may be populated with obstacle information in one of three methods as shown below. Regardless of the source, all obstacles are maintained as a list of known obstacles by the obstacle manager. And each time there is a change in shape of a known obstacle, an update is posted to the obstacle manager consumers, e.g., the helm.

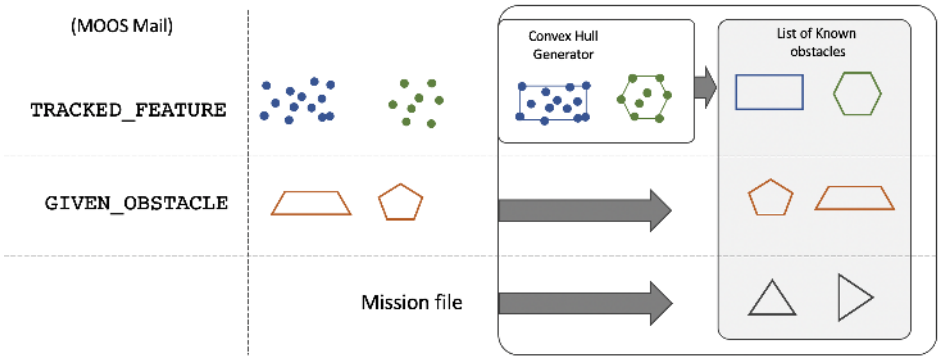

Figure 213: **Obstacle Manager Input**: Currently supported obstacle manager input comes from one of three sources. (1) As individual LIDAR points with grouping information, (2) as polygon obstacles via incoming mail, typically followed by updates for each obstacle, or (3) as polygon obstacles read from a mission configuration file.

In this section, the obstacle manager sources are described in more detail, along with how the obstacle manager processess incoming information.

Tracked Feature Inputs The obstacle manager is designed to work with one or more other applications producing tracked features. These tracked features may be points generated by a LIDAR, or some other sensor. The obstacle simulator, uFldObstacleSim, has a mode that supports generation of simulated LIDAR points. The obstacle manager subscribes for the MOOS variable TRACKED_FEATURE, of the form:

```
TRACKED_FEATURE  = x=23.2,y=19.8,label=47
TRACKED_FEATURE  = x=22.9,y=18.2,label=47
```

It is also assumed that the input will arrive with some grouping or clustering algorithm applied to each feature, reflected in the label field in the examples above. The obstacle manager maintains a database in the form of a mapping, keyed on the obstacle label, to a list of features. For each obstacle only the N most recent features (points) are held. The obstacle manager may be configured

to ignore incoming features beyond a certain range to the robot. This helps ensure bounded memory growth of the application as longer missions unfold.

For each cluster of points, the obstacle manager maintains a single convex polygon representing each cluster. By default a convex hull polygon of each cluster is maintained as shown in Figure 214.

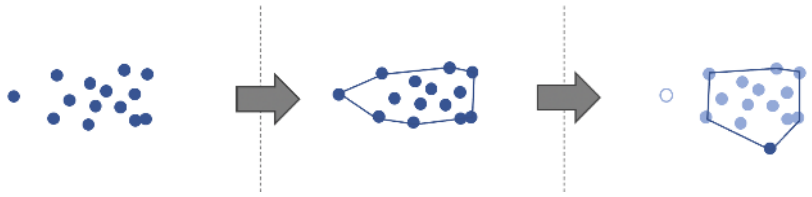

Figure 214: **The Convex Hull Generator**: Each stream of like-labeled points is a cluster from which the generator will maintain a current convex hull. As points age-out, the convex hull may shrink or shift through space. This process naturally accommodates both outlier points, and points associated with a slowly moving obstacle. In the right-hand panel of the figure, the dark blue point shows a newly received point. The white point has aged-out. The lighter blue points are not as old as the new point, but have not yet aged-out.

The stream of points, typically originating from a LIDAR, is presumed to be a constant update stream of points as new information is generated from the LIDAR. The obstacle manager holds enough points in memory to periodically generate a convex hull, but it also guards against unbounded memory by allowing points to drop. A point will be dropped based on one of two criteria. The first is a maximum total number of points held, with oldest points dropped as new points come in, i.e., first-in-first-out. The max amount is applied per-cluster. Currently there is no limit on the number of clusters. This max limit is by default 20 points, but can be set with the `max_pts_per_cluster` parameter as shown below.

```
max_pts_per_cluster = 40   // default is 20
max_age_per_point   = 10   // default is 20 seconds
```

The obstacle manager also will remove points from memory based on the age of the incoming point. By default, the point will be dropped after 20 seconds. The `TRACKED_FEATURE` message does not contain a timestamp. The age of the point is based on the timestamp the obstacle manager applies upon receipt. Dropping points based on age not only serves the purpose of bounding memory growth, it also serves as a kind of outlier rejection. A spurious LIDAR point, that may have come from a wave or some other non-obstacle phenomena, may cause a temporary growth of the convex hull, but it will not last long. Dropping points based on age also allows moving obstacles to have a convex hull that shifts with the motion of the obstacle. Each time the convex hull for an obstacle changes shape or location, an update message is produced by the obstacle manager. If there is an obstacle avoidance behavior currently in existence in the helm, tied to this obstacle, the behavior will be immediately updated, likely resulting in a slight adjustment for the output on that particular behavior.

An Alternative to Convex Hull Clustering The convex hull is general and preferred when the object is not of a known size or geometry. In certain cases when information about the object size is known a priori, the user may configure the obstacle manager to associate a regular polygon

of fixed size. The polygon is centered on the center of mass for all points that have not aged-out, as shown in Figure 215. As before, the polygon will shift with a moving obstacle as new points arrive and older points age-out.

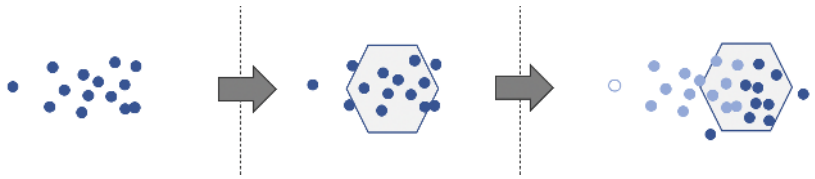

Figure 215: The obstacle manager may also maintain a regular polygon of user configurable radius and number of vertices, centered on the average of points for a given cluster.

To generate obstacle polygons in this manner the `lasso` parameter is used with the following options:

```
lasso = true                // default is false
lasso_points = 6            // default is 6
lasso_radius = 5            // (meters) default is 5
```

The first parameter turns on the lasso option, and the following two set the radius and number of vertices used in the regular polygon.

Configuring Obstacles of Given Location and Size The obstacle manager may also be configured with obstacles, in polygon form, at given shape and location. This can be done with a configuration parameter in the mission (`.moos` file), or through incoming mail. In either case, the format is the same.

To configure given obstacles through incoming mail, the format is:

```
GIVEN_OBSTACLE = pts={109,-72:113,-76:113,-82:109,-86:103,-86:99},label=23
```

Providing given obstacles through MOOS messages is convenient when using another MOOS app to generate obstacles, perhaps from different sensor input, or perhaps through simulation to stress test the autonomy system. An obstacle is distinct by its label. For any given obstacle, the message may simply arrive once, or it may be steadily updated depending on the source application. Updates are applied immediately and passed on from the obstacle manager with a posting for any consumer of information about this obstacle, e.g., an obstacle avoidance behavior in the helm.

To configure given obstacles in the mission file, the format is:

```
given_obstacle = pts={109,-72:113,-76:113,-82:109,-86:103,-86:99},label=23
```

Providing given obstacles in the mission file may be convenient if there are known obstacles such as buoys, or other items that are consistent with an operation area. Such data can simply be just loaded at mission time.

Obstacle Duration and Memory Management in the Obstacle Manager The obstacle manager holds information about all known obstacles. A policy for obstacle deletion is needed to guard against unbounded memory growth. In the case of obstacles tied to LIDAR points, there is already a policy for points to age-out after a certain duration. The obstacle manager will remove the obstacle from its list of obstacles when there are no longer any LIDAR points associated with the obstacle. This will occur naturally as the vehicle moves away from an obstacle, beyond sensor range. And if the LIDAR points were a false detection due to a sensor anomaly, or wave, the flow of points related to this false detection will typically soon cease and the obstacle is deleted from the obstacle manager memory.

The case of obstacles arriving from GIVEN_OBSTACLE messages is different. The source of these obstacles is not known by the obstacle manager, by design. By default, once the obstacle manager has received information about an obstacle, it is held by the obstacle manager forever. When the obstacle manager is consuming information in this manner, a *duration* is required as part of the mail message. For example:

```
GIVEN_OBSTACLE = pts={109,-72:113,-76:113,-82:109,-86:103,-86:99},label=23,duration=5
```

By default, duration components are mandatory and cannot be any greater than 60 seconds. This amount may be changed with the configuration parameter:

```
given_max_duration=30  // Default is 60 seconds
```

Or it may be disabled completely with:

```
given_max_duration = off
```

This compels the application generating the GIVEN_OBSTACLE information to be mindful of obstacle duration, typically updating the information for each obstacle at a rate that is faster than the posted duration. This allows the obstacle manager to delete an obstacle in the *absence* of a recent update.

Currently the only application producing GIVEN_OBSTACLE mail messages is the uFldObstacleSim application. This method of feeding the obstacle manager may also be used by a future app that has a better method of clustering LIDAR points than the simple convex hull algorithms described earlier that are currently in use by the obstacle manager

Obstacle Manager Actions Upon Deletion of an Obstacle The intended consumer of obstacle manager output is the helm. When a new obstacle is detected, if it is within the configured range of the vehicle, a new obstacle avoidance behavior is spawned. When the vehicle later goes beyond this range, the avoidance behavior will be deleted by the helm.

Normally, by the time the obstacle manager is about to delete a known obstacle, the helm has probably already deleted the corresponding obstacle avoidance behavior. However, it is possible that the obstacle became known to the obstacle manager due to spurious sensor/LIDAR data. Perhaps

this obstacle is very near the vehicle and caused the helm to spawn an obstacle avoidance behavior for this phantom obstacle. In this case the flow of sensor information related to this false obstacle may cease a very short time later, and the obstacle manager will quickly delete the obstacle from its list of known obstacls. Yet the helm may still have the avoidance behavior associated with the false obstacle, and would otherwise continue to have this behavior until the vehicle moves far enough away. This is clearly not desirable since the avoidance behavior for that short-lived false obstacle may constrain the vehicle motion in adverse ways. Instead, we want the vehicle behavior to be removed when the obstacle manager removes the obstacle. For this reason, the obstacle manager will publish the below posting whenever an obstacle is removed from its memory:

```
OBM_RESOLVED = 428
```

The helm obstacle avoidance behavior `BHV_AvoidObstacleV21` or newer, monitors for the above postings. If it notes that the obstacle id for which it is associated has been resolved, the behavior will put in motion the steps to self-delete immediately.

70.2.2 Configuring the Helm to Handle Obstacle Manager Output

The obstacle manager exists primarily to serve the helm, by posting notifications to the helm that allow the helm to (a) spawn an obstacle avoidance behavior for a new obstacle, and (b) update a previously spawned obstacle avoidance behavior with an updated location or shape of the obstacle.

Configuration on the helm side is straight-forward, requiring only a configuration block for the obstacle avoidance behavior, similar to:

```
//------------------------------------------
Behavior=BHV_AvoidObstacleV21
{
  name       = avd_obstacles_
  pwt        = 500
  condition  = DEPLOY = true
  templating = spawn
  updates    = OBSTACLE_ALERT

     allowable_ttc = 5
       buffer_dist = 3
    pwt_outer_dist = 20
    pwt_inner_dist = 10
    completed_dist = 25
}
```

See the documentation for the obstacle avoidance behavior for more details on the above parameters. The important point here is that upon startup of the helm, no obstacle avoidance behavior will be spawned until the helm receives an alert about an obstacle. This alert comes via a posting to the variable `OBSTACLE_ALERT`. Until an alert arrives, this behavior exists in the helm as a *template*, capable of spawning any number of behaviors, one for each obstacle.

When the helm starts, on behalf of the obstacle avoidance behavior template, the helm posts an alert request:

```
OBM_ALERT_REQUEST = alert_range=20, update_var=OBSTACLE_ALERT
```

The alert request uses the value of the updates parameter and the pwt_outer_dist parameter to construct the alert request. The alert_range component of the alert request is automatically set to match the pwt_outer_dist configuration parameter of the behavior. In the behavior, when the obstacle is at, or beyond this range, the priority weight of the behavior becomes zero. This same range value will inform the obstacle manager that (1) until the obstacle is closer than this distance, postings to the OBSTACLE_ALERT should not be made for this obstacle id, and (2) after the obstacle has become farther than this distance, the same said postings should cease. After the vehicle has opened range to the obstacle beyond the completed_dist, the behavior will complete and will be removed from them helm.

Note that if there are say ten instances of this behavior, for ten separate obstacles, they will all receive their updates through the same OBSTACLE_ALERT MOOS variable. Each posting, however, will contain the name of the behavior. For example:

```
OBSTACLE_ALERT = name=avd_obstacles_ob_08#poly=pts={52.2,-32.2:53,-33.02:53,...
OBSTACLE_ALERT = name=avd_obstacles_ob_03#poly=pts={52.82,-115.86:50.64,...
OBSTACLE_ALERT = name=avd_obstacles_ob_02#poly=pts={72.07,-68.93:75.15,...
```

The helm will ensure the updates are only applied to the behavior that matches the name in the update.

70.3 Implementation of the Obstacle Manager

70.3.1 Obstacles versus Contacts

The obstacle manager was designed to handle stationary obstacles like buoys or bridge pylons. To handle moving obstacles such as other marine vessels, the IvP Helm uses a similar MOOS application called a contact manager and a collision avoidance behavior based on COLREGS protocol. This is outside the scope of this paper. Our convention is to use the term *obstacle* for objects that are stationary or at best slowly drifting. Obstacles are handled by the obstacle manager and the Avoid Obstacle behavior. We use the term *contact* for moving vessels. Contacts are handled by the contact manager, and the Collision Avoidance behaviors.

70.3.2 Drifting Obstacles

The obstacle manager is equipped with the ability to handle *drifting* obstacles, e.g., a drifting buoy. In effect, there is not much difference between a truly drifting object and an object with slightly shifting sensor readings. Proper functioning of the obstacle manager depends on proper configuration of the *decay* of sensed points. As the drifting obstacle moves, sensed points at the new location will appear, and sensed points at older locations will become stale and disappear from the obstacle manager memory. By default the number of sensed points is limited in size, per cluster key, to 20 points. This can be changed with the parameter max_pts_per_cluster. By default, the points will decay and be removed from memory after 20 seconds. This can be changed wit the parameter max_age_per_point.

70.3.3 Obstacle and Alert Management

The obstacle manager by default will not generate any alerts unless another app has indicated that it would like to receive alerts. This alert request comes in the form of a message:

```
OBM_ALERT_REQUEST = update_var=OBSTACLE_ALERT, alert_range=40, name=avd_obstacle
```

For the current release of the obstacle manager, an alert of a single configuration is supported. Subsequent publications to OBM_ALERT_REQUEST, with different values for the alert variable or range, will simply overwrite the previous setting.

Alert Generation and The Alert Range The obstacle manager will generate alerts about an obstacle to the variable requested in the OBM_ALERT_REQUEST message. The first time this alert variable is published, it can be regarded as alert in the sense that the obstacle's existence is new information for whomever is subscribing for these alerts (typically the helm). A couple example postings are shown below.

```
OBSTACLE_ALERT = name=ob_4#poly=pts={48.7,-77.2:52.3,-80.8:52.3,-86:48.7,-89.6:43.5,\
                 -89.6:39.9,-86:39.9,-80.8:43.5,-77.2},label=ob_4

OBSTACLE_ALERT = name=ob_2#poly=pts={62.9,-48.7:67.1,-52.9:67.1,-58.9:62.9,-63.1:56.9,\
                 -63.1:52.7,-58.9:52.7,-52.9:56.9,-48.7},label=ob_2
```

Subsequent alert publications are essentially updates on the obstacle. These subsequent publications are only made if the size or the position of the obstacle changes. For sensed obstacles derived from point data or other dynamic data, alert updates are fairly frequent. For given static obstacles, subsequent alert updates may never happen after the initial alert.
When the robot is beyond the *alert range* to an obstacle, the obstacle manager will no longer generate alerts. Alert publications will resume as soon as the robot returns to within the alert range. For the relatively static given obstacles, the obstacle manager takes care to re-publish an alert for the obstacle when the robot returns within the alert range, even if the size or position of the obstacle has not changed. A vehicle returning within the alert range always needs to be re-alerted as if encountering this obstacle for the very first time.

Resolution of Alerts The obstacle manager will generate a single alert for each obstacle, thereafter assuming the entity that needed to know about the obstacle has been properly notified. For dynamic obstacles, alerts will continue as the position or shape of the obstacle changes.

The helm may at some point want to delete the, e.g., obstacle avoidance, behavior that registered for the alert, typically when the obstacle has been passed and has reached a range where it is no longer a concern. In the case of the `AvoidObstacle` behavior, the behavior will be deleted when the obstacle is beyond the completed_dist range, and this range is tied to be equivalent to the requested alert range.

The Ignore Range For dynamic obstacles, receiving tracked feature point information, normally all such points are received and processed, regardless of range. The ignore_range parameter sets

a distance, in meters, beyond which an incoming point will be ignored. This can help reduce management of spurious obstacles at obviously harmless ranges to ownship. By default this value is −1, meaning all points are received and managed. In future releases this range may be automatically tied to the alert range, and ignore regions will also be supported, to reject points on land or outside of the vehicle operation area.

70.4 Configuration Parameters for pObstacleMgr

The following parameters are defined for pObstacleMgr. A more detailed description is provided in other parts of this section. Parameters having default values are indicated.

Listing 70.38: Configuration Parameters for pObstacleMgr.

alert_range:	The range in meters, between ownship and an obstacle, that an alert will be triggered. Section 70.3.3.
ignore_range:	The range in meters, between an incoming point (tracked feature) and ownship, beyond which the point will be ignored.
lasso:	If `true`, the polygon associated with each cluster will be firmly set to a circular polygon of a set radius and set number of vertices. The default is `false`. Section 70.2.1.
lasso_points:	The number of points used in the regular polygon representing the obstacle, if the lasso parameter is set to `true`. The default is 6, a hexagon. Section 70.2.1.
lasso_radius:	The radius (distance to any vertex) used in the regular polygon representing the obstacle, if the lasso parameter is set to `true`. The default is 5 meters. Section 70.2.1.
max_age_per_point:	The maximum number of seconds that a point (tracked feature) will be retained in memory. Beyond this number, points will be dropped. The default is 20 seconds. Section 70.3.2.
max_pts_per_cluster:	The maximum number of points (tracked features) retained in memory per cluster (points having the same label). Beyond this number, oldest points will be dropped. The default is 20 points. Section 70.3.2.
point_var:	The name of the MOOS variable to looked for tracked features. The default is TRACKED_FEATURE. Section 70.2.1.
obstacles_color:	When the obstacle manager is rendering obstacles (post_view_polys is `true`), this parameter will set the obstacle color. The default value is `blue`.
poly_label_thresh:	When the number of obstacle polygons is greater than this parameter, the view polylgons will be published with a label color of `invisible`, resulting in less work for the pMarineViewer. The default is 25. This is solely to help boost performance of pMarineViewer in extreme uses cases of very large numbers of obstacles and high time warp, and possibly slower machines. It does not affect the function of the obstacle manager in any other way.

poly_shade_thresh: When the number of obstacle polygons is greater than this parameter, the view polylgons will be published with a shade (fill) color of `invisible`, resulting in less work for the pMarineViewer. The default is 100. This is solely to help boost performance of pMarineViewer in extreme uses cases of very large numbers of obstacles and high time warp, and possibly slower machines. It does not affect the function of the obstacle manager in any other way.

poly_vertex_thresh: When the number of obstacle polygons is greater than this parameter, the view polylgons will be published with a vertex size of zero, resulting in less work for the pMarineViewer by only rendering the polygon edges, without the vertices. The default is 150. This is solely to help boost performance of pMarineViewer in extreme uses cases of very large numbers of obstacles and high time warp, and possibly slower machines. It does not affect the function of the obstacle manager in any other way.

post_dist_to_polys: Legal values are `true`, `false`, or `close`. If `true`, the distance from ownship to an obstacle is published for each obstacle, to the variable OBM_DIST_TO_OBJ. When set to `close`, these publications only occur when the obstacle is closer than alert_range. When `false`, these postings are turned of completely. The default is `close`.

post_view_polys: When `true`, the obstacle polygons are published published by the obstacle manager. Normally this is redundant since the obstacles are also published by the obstacle avoidance behavior and the obstacle simulator. Legal values are `true` and `false`. The default is `false`.

70.4.1 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ pBasicContactMgr --example or -e
```

This will show the output shown in Listing 39 below.

Listing 70.39: A Simple pObstacleMgr Example.

```
1  ============================================================
2  pObstacleMgr Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = pObstacleMgr
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   point_var = TRACKED_FEATURE  // default is TRACKED_FEATURE
11
12   given_obstacle = pts={90.2,-80.4:....:85.4,-80.4},label=ob_23
13
14   post_dist_to_polys = true  // true, false or (close)
```

```
15    post_view_polys = true     // (true) or false or
16
17    max_pts_per_cluster = 20   // default is 20
18    max_age_per_point   = 20   // (secs)  default is 20
19
20    alert_range  = 20          // (meters) default is 20
21    ignore_range = -1          // (meters) default is -1, (off)
22
23    lasso = true               // default is false
24    lasso_points = 6           // default is 6
25    lasso_radius = 5           // (meters) default is 5
26
27    obstacles_color = color    // default is blue
28
29    // To squeeze more viewer effic when large # of obstacles:
30    poly_label_thresh = 25     // Set label color=off if amt>25
31    poly_shade_thresh = 100    // Set shade color=off if amt>100
32    poly_vertex_thresh = 150   // Set vertex size=0 if amt>150
33  }
```

70.5 Publications and Subscriptions of pObstacleMgr

The interface for pObstacleMgr, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pObstacleMgr --interface or -i
```

70.5.1 Variables Published by pObstacleMgr

The output of pObstacleMgr is:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- [ALERT_VAR]: The obstacle manager will publish alerts through a variable specified in an alert configuration, via the incoming OBM_ALERT_REQUEST message. See Section 70.3.3.
- VIEW_POLYGON: The polygon representing the obstacle is posted in this variable. It is re-published when/if the shape or location changes, or upon ownship approaching within range of the obstacle.
- OBM_DIST_TO_OBJ: For each object retained in the object manager's memory, object manager will continually post the distance from ownship to the obstacle, in meters. This posting can disabled by setting the post_dist_to_polys to `false`. By default it is enabled.
- OBM_CONNECT: Upon successful launch of the obstacle manager, this variable is posted. It helps coordinate with the obstacle simulator which may be running on the shoreside and may have already published obstacle information. Upon receipt of this posting, the obstacle simulator will refresh the postings.
- OBM_MIN_DIST_EVER: The obstacle manager uses its knowlege of all obstacle locations and ownship location and keeps track of the closest that an obstacle has ever come to ownship. This minimum distance, and the id of the obstacle, are posted to this variable.

- OBM_RESOLVED: When an obstacle is removed from the obstacle manager, a notice is posted containing just the id of the removed obstacle. This may be used by the obstacle avoidance behavior in the helm to let it know that this obstacle no longer exists. Section **??**.

The obstacle manager will also publish to whatever MOOS variables are specified in the obstacle alerts. See Section 70.3.3.

70.5.2 Variables Subscribed for by pObstacleMgr

The pObstacleMgr application will subscribe for the following four MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- GIVEN_OBSTACLE: One of the obstacle manager input options is to receive the obstacle information in the form of a convex polygon with a unique label. See Section 70.2.1.
- NAV_X: Ownship's current position in x coordinates.
- NAV_Y: Ownship's current position in y coordinates.
- OBM_ALERT_REQUEST: A message, typically from the obstacle avoidance behavior of the helm, to configure the criteria and format for posting obstacle manager alerts. See Section 70.3.3.
- TRACKED_FEATURE: One of the obstacle manager's input options is to received simulated LIDAR points. Each point is received as a message of this variable. See Section 70.2.1.

70.6 Terminal and AppCast Output

The pObstacleMgr application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 40 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. The counter on the end of line 2 is incremented on each iteration of pObstacleMgr, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

The output in the below example comes from the s1_alpha_obstacles mission.

Listing 70.40: Example terminal or appcast output for pObstacleMgr.

```
 1  ============================================================================
 2  pObstacleMgr alpha                                              0/0(238)
 3  ============================================================================
 4  Configuration (point handling):
 5    point_var:    TRACKED_FEATURE
 6    max_pts_per_cluster: 50
 7    max_age_per_point:   60
 8    ignore_range:        40
 9  Configuration (alerts):
10    alert_var:   OBSTACLE_ALERT
11    alert_name:  avoid_obstacle_
12    alert_range: 19
13  Configuration (lasso option):
14    lasso:        true
15    lasso_points: 8
```

```
16    lasso_radius: 6
17  ============================================
18  State:
19    Nav Position:      (61,-129.7)
20    Points Received:   58
21    Points Invalid:    0
22    Points Ignored:    147
23    Polygon obstacles: 4
24    Clusters:          4
25    Clusters released: 0
26
27  ObstacleKey  Points  HullSize  Updates
28  -----------  ------  --------  -------
29  b            14      8         n/a
30  c            16      8         55
31  d            13      8         23
32  e            11      8         46
```

The first group of lines (4-16) show the configuration settings for pObstacleMgr. The status of pObstacleMgr is shown in Lines 18-32.

70.7 Simple Example Missions

As of Release 19.8, there are two example missions using the obstacle manager.

- s1_alpha_obstaclemgr: A single vehicle mission with obstacles generate by a stream of points.
- m2_berta_obstacles: A two vehicle mission with obstacles given at fixed locations.

Differences between the obstacle manager and the contact manager:
- obstacles don't have type or group
- Only one global alert range for all obstacles. Set in the config block but may be overridden by a behavior when it registers wit the obstacle manager

71 Geometry Utilities

71.1 Overview

This section discusses a few geometry data structures often used by the helm and the pMarineViewer application - convex polygons, lists of line segments, points, seglrs and vectors. These data structures are implemented by the classes XYPolygon, XYSegList, XYPoint, XYSeglr and XYVector respectively in the lib_geometry module distributed with the MOOS-IvP software bundle. The implementation of these class definitions is somewhat shielded from the helm user's perspective, but they are often involved in parameter settings of for behaviors. So the issue of how to specify a given geometric structure with a formatted string is discussed here.

Furthermore, the pMarineViewer application accepts these data structures for rendering by subscribing to three MOOS variables VIEW_POLYGON, VIEW_SEGLIST, mVIEW_POINT, anv VIEW_VECTOR. These variables contain a string format representation of the structure, often with further visual hints on the color or size of the edges and vertices for rendering. These variables may originate from any MOOS application, but are also often posted by helm behaviors to provide visual clues about what is going on in the vehicle. In the Alpha mission, for example, the waypoint behavior posted a seglist representing the set of waypoints for which it was configured, as well as posting a point indicating the next point on the behavior's list to traverse.

71.2 General Geometric Object Properties

Each of the four geometric objects, XYPolygon, XYSegList, XYPoint, and XYVector, are a subclass of the general XYObject class, and share certain properties discussed here.

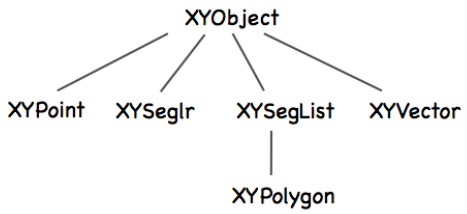

Figure 216: **Class hierarchy:** All geometric objects are subclasses of the XYObject class.

71.2.1 Common Properties

The following properties are defined at the XYObject level. All properties may be optionally left undefined by the user.

- *label*: A string that is often rendered in a GUI alongside the object rendering. In the pMarineViewer application, the label is also used as a unique identifier and successive receipt of geometric objects with the same label will result in the new one replacing" the older one. Thus the visual effect of "moving" objects is rendered in this way.

- *msg*: A string typically regarded as an alternative to the label string when rendering an object. In the pMarineViewer application, if an object has a non-null *msg* field, this will be used for rendering the label instead of the *label* field.
- *type*: A string conveying some information on the type of the object from the source application. For example a point may be of type "waypoint" or "rendez-vous", and so on.
- *time*: The time field is a double that may optionally be set to indicate when the point was generated, or how long it should exist before "expiring", or however an application may wish to interpret it.
- *source*: A string representing the source that generated the object. This may distinguish MOOS applications or helm behaviors for example.

71.2.2 Rendering Hint Properties

The following optional properties are defined at the `XYObject` level for providing hints to applications that may be using them for rendering.

- *active*: This is a Boolean that is regarded as `true` if left unspecified, and is used to indicate whether or not the object should be rendered. By setting this value to `false` for a given object, it would effectively be erased in the pMarineViewer application for example.
- *vertex_size*: This non-negative floating point value is a hint for how large to draw a vertex point for the given object. If left unspecified in the pMarineViewer application for example, the size of the vertex would be determined by the global setting for vertices set in the GUI.
- *edge_size*: This non-negative floating point value is a hint for how wide to draw an edge for the given object (if it has edges). If left unspecified in the pMarineViewer application for example, the width of the edge would be determined by the global setting for edges set in the GUI.
- *vertex_color*: This parameter specifies a hint or request to draw the vertices in this object in the given color specification. Colors may be specified by any defined color string or RGB string value as described in the Colors Appendix. In the pMarineViewer application, if this hint is not provided for received objects, the vertex color would be determined by the global setting for vertices set in the GUI. If the color is set to `"invisible"`, this is effectively a request that the vertex not be rendered.
- *edge_color*: This parameter specifies a hint or request to draw the edgees in this object in the given color specification. Colors may be specified by any defined color string or RGB string value as described in the Colors Appendix. In the pMarineViewer application, if this hint is not provided for received objects, the edge color would be determined by the global setting for edges set in the GUI. If the color is set to `"invisible"`, this is effectively a request that the edge not be rendered.
- *label_color*: This parameter specifies a hint or request to draw the label for this object in the given color specification. Colors may be specified by any defined color string or RGB string value as described in the Colors Appendix. In the pMarineViewer application, if this hint is

not provided for received objects, the label color would be determined by the global setting for labels set in the GUI. If the color is set to `"invisible"`, this is effectively a request that label not be rendered.

71.3 Points

Points are implemented in the `XYPoint` class, and minimally represent a point in the x-y plane. These objects are used internally for applications and behaviors, and may also be involved in rendering in a GUI and therefore may have additional fields to support this as described in Section 71.2.

71.3.1 String Representations for Points

The only required information for a point specification is its position in the x-y plane. A third value may optionally be specified in the z-plane. If z is left unspecified, it will be set to zero. The standard string representation is a comma-separated list of param=value pairs like the following examples:

```
point   = x=60, y=-40
point   = x=60, y=-40, z=12
point   = z=19, y=5, x=23
```

Partly for backward compatibility, a very simplified string representation of a point is also supported:

```
point   = 60,-40
point   = 60, -40, 0
```

An example of string representation of point with all the optional parameters described in Section 71.2 might look something like:

```
point   = x=60, y=-40, label=home, label_color=red, source=henry, type=waypoint,
          time=30, active=false, vertex_color=white, vertex_size=5, msg=bingo
```

Note that although a few different string formats are supported for *specifying* a point, only a single format is used when a `XYPoint` object is serialized into a string representation.

71.4 Seglists

Seglists are implemented in the `XYSegList` class and are comprised of an ordered set of vertices, implying line segments between each vertex. Seglist instances may be used for many things, but perhaps most often used to represent a set of vehicle waypoints. The geometry library has a number of basic operations defined for this class such as intersection testing with other objects, rotation, proximity testing and so on.

71.4.1 Standard String Representation for Seglists

The only requirement for a seglist specification is one or more vertex locations in the x, y plane. Values may optionally be specified in the z plane. If z is left unspecified it will default to zero. The standard string representation is a comma-separated list of param=value pairs like the following examples:

```
points   = pts={60,-40:60,-160:150,-160:180,-100:150,-40},label=foxtrot,type=top
```

By "standard" format we mean it is not only accepted as initialization input, but is also the format produced when an existing instance is serialized using the object's `string XYSeglist::get_spec()` function. If z values are used, an example may look like the following which is the same as the above but associates $z = 2$ with each point:

```
points = pts={60,-40,2:60,-160,2:150,-160,2:180,-100,2:150,-40,2},label=foxtrot
```

An example of a string representation with all the optional parameters described in Section 71.2 might look something like:

```
points = {pts=60,-40,2:60,-160,2:150,-160,2:180,-100,2:150,-40,2},label=foxtrot
         label_color=green, source=henry, type=return_path, time=30, active=false,
         vertex_color=white, vertex_size=5, edge_size=2, edge_color=red, msg=bingo
```

71.4.2 The Lawnmower String Representation for Seglists

Seglists may also be built using the `lawnmower` format. The following is an example:

```
points = format=lawnmower, label=foxtrot, x=0, y=40, height=60, width=180,
         lane_width=15, rows=north-south, startx=20, starty=-300, degs=45
```

The rotation of the pattern can optionally be specified in radians. For example, `degs=45` is equivalent to `rads=0.785398`. If, for some reason, both are specified, the seglist will be built using the `rads` parameter.

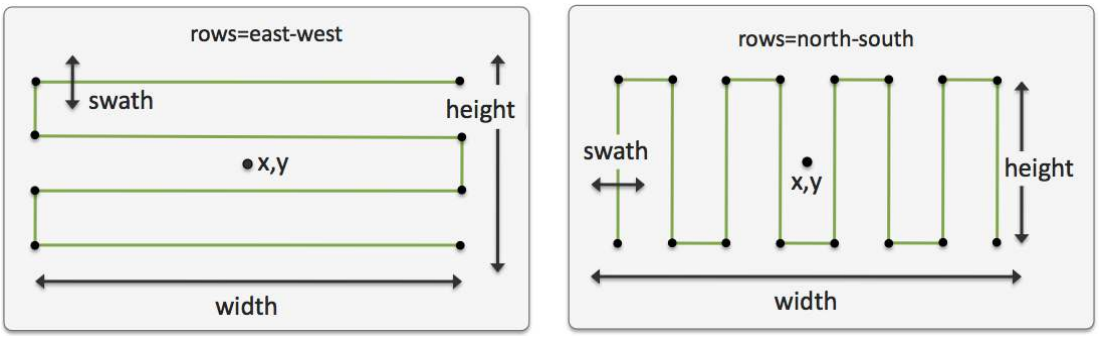

Figure 217: **Seglists built with the `lawnmower` format:** The pattern is specified by (a) the location of the center of the pattern, (b) the height and width of the pattern, (c) the lane width which determines the number of rows, (d) whether the pattern rows proceed north-south or east-west, and (e) an optional rotation of the pattern.

The `startx` and `starty` parameters name a point in the xy-plane to influence the starting position of the pattern. The seglist has four possible starting positions, and chosen starting position will be the position closest to the (`startx`, `starty`) position. The `startx` and `starty` parameters are optional and default to zero. Users sometimes may which to use a vehicle's present x-y position for these parameters.

71.4.3 Seglists in the pMarineViewer Application

The pMarineViewer application registers for the MOOS variable VIEW_SEGLIST. The viewer maintains a list of seglists keyed on the label field of each incoming seglist. A seglist received with a thus far unique label will be added to the list of seglists rendered in the viewer. A seglist received with a non-unique label will replace the seglist with the same label in the memory of the viewer. This has the effect of erasing the old seglist since each iteration of the viewer redraws everything, the background and all objects, from scratch several times per second.

The label is the only text rendered with the seglist in pMarineViewer. Since the label is also used as the key, if the user trys to "update" the label, or use the label to convey changing information, this may inadvertently result in accumulating multiple seglists in the viewer, each drawn over one another. Instead, the VIEW_SEGLIST may be posted with the message to be posted in the `msg=value` component. When this component is non-null, pMarineViewer renders it instead of the contents in the label component.

71.5 Polygons

Polygons are implemented in the `XYPolygon` class. This implementation accepts as a valid construction only specifications that build a convex polygon. Common operations used internally by behaviors and other applications, such as intersection tests, distance calculations etc, are greatly simplified and more efficient when dealing with convex polygons.

71.5.1 Supported String Representation for Polygons

The only requirement for a polygon specification is three or more vertex locations in the x, y plane constituting a convex polygon. Values may optionally be specified in the z plane. If z is left unspecified it will default to zero. The standard string representation is a comma-separated list of param=value pairs like the following examples:

```
points   = pts={60,-40:60,-160:150,-160:150,-40},label=foxtrot,type=one
```

By "standard" format we mean it is not only accepted as initialization input, but is also the format produced when an existing instance is serialized using the object's `string XYPolygon::get_spec()` function. If z values are used, an example may look like the following which is the same as the above but associates $z = 4$ with each point:

```
points   = pts={60,-40,4:60,-160,4:150,-160,4:150,-40,4},label=foxtrot,type=one
```

Polygons are defined by a set of vertices and the simplest way to specify the points is with a line comprised of a sequence of colon-separated pairs of comma-separated x-y points in local coordinates such as:

```
polygon    = 60,-40:60,-160:150,-160:180,-100:150,-40:label,foxtrot
```

If one of the pairs, such as the last one above, contains the keyword `label` on the left, then the value on the right, e.g., `foxtrot` as above, is the label associated with the polygon. An alternative notation for the same polygon is given by the following:

```
polygon    = label=foxtrot, pts={60,-40:60,-160:150,-160:180,-100:150,-40}
```

This is an comma-separated list of equals-separated pairs. The ordering of the comma-separated components is insignificant. The points describing the polygon are provided in braces to signify to the parser that everything in quotes is the right-hand side of the `pts=` component. Both formats are acceptable specifications of a polygon in a behavior for which there is a `polygon` parameter.

71.5.2 A Polygon String Representation using the Radial Format

Polygons may also be specified by their shape and the shape parameters. For example, a commonly used polygon is formed by points of an equal radial distance around a center point. The following is an example:

```
polygon = format=radial, label=foxtrot, x=0, y=40, radius=60, pts=6, snap=1
```

The `snap` component in the above example signifies that the vertices should be rounded to the nearest 1-meter value. The `x`, `y` parameters specify the middle of the polygon, and `radius` parameters specify the distance from the center for each vertex. The `pts` parameters specifies the number of vertices used, as shown in Figure 218.

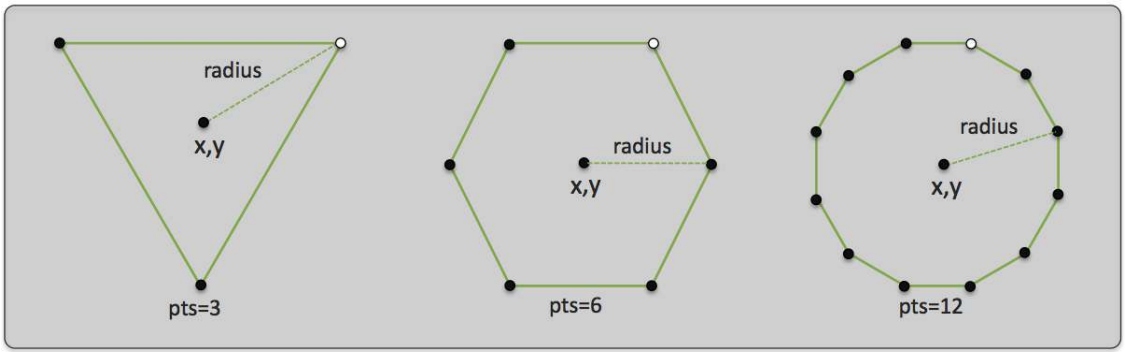

Figure 218: **Polygons built with the `radial` format:** Radial polygons are specified by (a) the location of their center, (b) the number of vertices, and (c) the radial distance from the center to each vertex. The lighter vertex in each polygon indicates the first vertex if traversing in sequence, proceeding clockwise.

71.5.3 A Polygon String Representation using the Ellipse Format

Polygons may also be built using the `ellipse` format. The following is an example:

```
polygon    = label=golf, format=ellipse, x=0, y=40, degs=45, pts=14, snap=1,
             major=100, minor=70
```

The x, y parameters specify the middle of the polygon, the major and minor parameters specify the radial distance of the major and minor axes. The pts parameters specifies the number of vertices used, as shown in Figure 219.

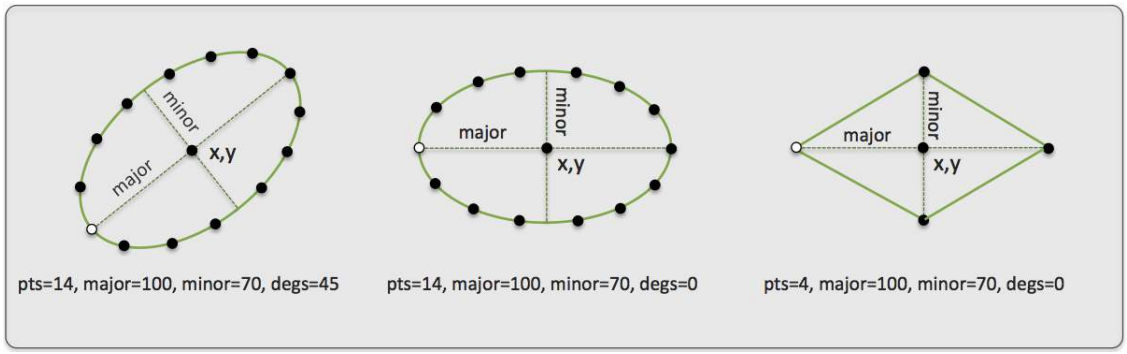

Figure 219: **Polygons built with the ellipse format:** Ellipse polygons are specified by (a) the location of their center, (b) the number of vertices, (c) the length of their major axis, (d) the length of their minor axis, and (e) the rotation of the ellipse.The lighter vertex in each polygon indicates the first vertex if traversing in sequence, proceeding clockwise.

The rotation of the ellipse can optionally be specified in radians. For example, degs=45 is equivalent to rads=0.785398. If, for some reason, both are specified, the polygon will be built using the rads parameter. When using the ellipse format, a minimum pts=4 must be specified.

71.5.4 Optional Polygon Parameters

Polygons also may have several optional fields associated with them. The label field is string that is often rendered with a polygon in MOOS GUI applications such as the pMarineViewer. The label_color field represents a color preference for the label rendering. The type and source fields are additional string fields for further distinguishing a polygon in applications that handle them. The active field is a Boolean that is used in the pMarineViewer application to indicate whether the the polygon should be rendered. The time field is a double that may optionally be set to indicate when the polygon was generated, or how long it should exist before "expiring", or however an application may wish to interpret it. The vertex_color, edge_color, and vertex_size fields represent further rendering preferences. The following are two equivalent further string representations:

```
polygon = format=radial, x=60, y=-40, radius=60, pts=8, snap=1, label=home,
          label_color=red, source=henry, type=survey, time=30, active=true,
          vertex_color=white, vertex_size=5, edge_size=2
polygon = format,radial:60,-40:radius,60:pts,8:snap,1:label,home:
          label_color,red:source,henry:type,survey:time,30:active,true:
          vertex_color,white:vertex_size,5:edge_size,2
```

The former is a more user-friendly format for specifying a polygon, perhaps found in a configuration file for example. The latter is the string representation passed around internally when XYPolygon objects are automatically converted to strings and back again in the code. This format is more likely to be found in log files or seen when scoping on variables with one of the MOOS scoping tools.

71.6 Seglrs

Seglrs are implemented in the `XYSeglr` class and are essentially comprised of list of line segments followed by a ray. They may be used in certain applications to indicate a ship maneuver. Minimally each seglr consists of at least one vertext in the x, y plane (but typically more, indicating a series of line segments), and the direction of the ray. They may also be configured with all relevant drawing hints discussed in Section 71.2.

71.6.1 String Representations for Seglrs

Seglr objects may be initialized from a string representation, and converted to a string representation from an existing object. The following is an example:

```
seglr = pts={5,5:30,10:90,-20},ray=45
```

The following is a string example using many of the general object fields and drawing hints available:

```
seglr = pts={5,5:30,10:90,-20},ray=45,            <-- defines the seglr
        ray_len=10,head_size=3,                   <-- seglr drawing hints
        label=alpha,edge_color=red,edge_size=2,   <-- general object
        vertex_size=3,vertex_color=green          <--   drawing hints
```

The parameters, `ray_len=10` and `"head_size=3"`, are drawing hints unique to the seglr object and refers to how big the arrow head should be rendered and how long the ray should be rendered. They are given in meters, and the default is `ray_len=10` and `head_size=3`.

71.6.2 Seglrs in the pMarineViewer Application

The `pMarineViewer` application registers for the MOOS variable `VIEW_SEGLR`. The viewer maintains a list of seglrs keyed on the label field of each incoming seglr. A seglr received with a thus far unique label will be added to the list of seglrs rendered in the viewer. A seglr received with a non-unique label will replace the seglr with the same label in the memory of the viewer. This has the effect of erasing the old seglr since each iteration of the viewer redraws everything, the background and all objects, from scratch several times per second.

The `VIEW_VECTOR` may be posted with the message label, to be posted just beyond the arrow head. This is done with the `msg` component of the posting. For example:

```
VIEW_SEGLR = pts={5,5:30,10:90,-20},ray=45,ray_len=10,head_size=3,  \
             label=alpha,edge_color=red,edge_size=2,msg=alpha
```

71.7 Vectors

Vectors are implemented in the `XYVector` class and are essentially comprised of a location, direction and magnitude. They may be used in certain applications dealing with simulated or sensed forces, or simply used in a GUI application to render current fields or an instantaneous vehicle pose and trajectory. Minimally each vector consists of a location in the x, y plane and a direction and magnitude. They may also be configured with all relevant drawing hints discussed in Section 71.2.

71.7.1 String Representations for Vectors

Vector objects may be initialized from a string representation, and converted to a string representation from an existing object. The following is an example:

```
vector = x=5,y=10,ang=45,mag=20
```

Alternatively, instead of expressing the vector in terms of its direction and magnitude, it may also be given in terms of its magnitude in both the x and y direction. The following vector is nearly identical, modulo rounding errors, to the above configured vector:

```
vector = x=5,y=10,xdot=12.142,ydot=12.142
```

When a vector object is serialized to a string by invoking the object's native serialization function, the first of the two above formats will be used. The following is a string example using many of the general object fields and drawing hints available:

```
vector = x=5,y=10,ang=45,mag=20,label=pingu,source=simulator,type=wind,
         vertex_size=2,vertex_color=red,edge_color=red,edge_size=2,head_size=12
```

The last parameter, `"head_size=12"`, is a drawing hint unique to the vector object and refers to how big the arrow head should be rendered. If left unspecified, it may simply be rendered at whatever size the GUI application uses by default.

71.7.2 Vectors in the pMarineViewer Application

The pMarineViewer application registers for the MOOS variable VIEW_VECTOR. The viewer maintains a list of vectors keyed on the label field of each incoming vector. A vector received with a thus far unique label will be added to the list of vectors rendered in the viewer. A vector received with a non-unique label will replace the vector with the same label in the memory of the viewer. This has the effect of erasing the old vector since each iteration of the viewer redraws everything, the background and all objects, from scratch several times per second.

The label is the only text rendered with the vector in pMarineViewer. Since the label is also used as the key, if the user trys to ”update” the label, or use the label to convey changing information, this may inadvertently result in accumulating multiple vectors in the viewer, each drawn over one another. Instead, the VIEW_VECTOR may be posted with the message to be posted in the `msg=value` component. When this component is non-null, pMarineViewer renders it instead of the contents in the label component.

72 Introduction to the Alog Toolbox

72.1 Overview

The Alog Toolbox provides a number of utilities for post-mission analysis and/or editing of mission log files. They all assume the `.alog` file format produced by `pLogger`. There are two groups of tools:

72.2 Graphical

- `alogview`: A graphical tool for rendering multiple vehicle trajectories, time-series plots for any logged numerical data, objective functions produced from helm behaviors, helm summary reports for any iteration in any vehicle, and interleaved variable histories for any logged MOOS variable. Figure 220 shows the primary data windows on the left and a few optional pop-up windows on the right.

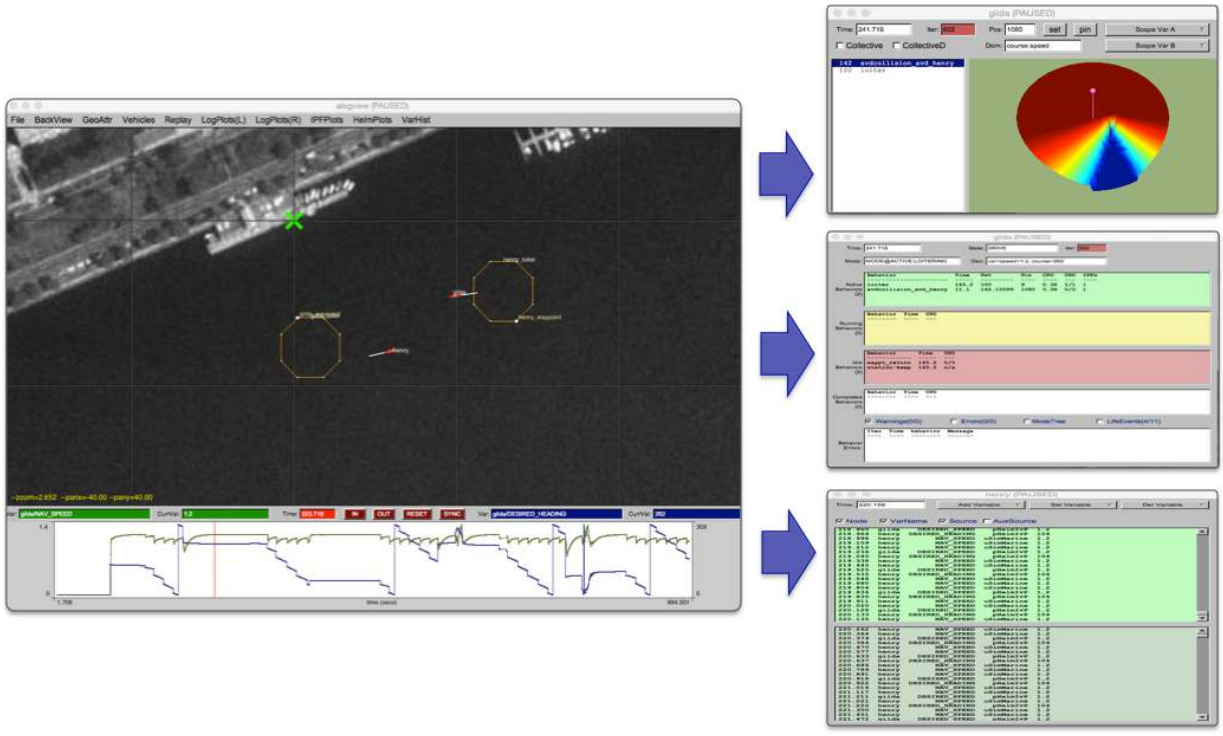

Figure 220: **The `alogview` tool**: used for post-mission rendering of alog files from one or more vehicles and stepping op-area and time.

72.3 Command Line

- `alogscan`: Generates a summary report on the contents of a given .alog file. The report lists each logged MOOS variable, which app(s) publish it, the min/max publish time and total number of character and lines for the variable.

- aloggrep: Create a new MOOS .alog file by retaining only the given MOOS variables or sources, named on the command line, from a given .alog file.
- alogrm: Remove the entries matching the given MOOS variables or sources from the given .alog file and generate a new .alog file.
- alogclip: Create a new MOOS .alog file from a given .alog file by removing entries outside a given time window.
- aloghelm: Perform one of several optional helm reports based on helm output logged in the given .alog file.
- alogiter: Analyzes the ITER_GAP and ITER_LEN information provided by all applications recorded in the given alog file.
- alogsort: Create a new MOOS .alog file by sorting the alog entries based on their timestamps.
- alogsplit: Split the given alog file into a directory, within which each MOOS variable is split into its own file containing only that variable. The split will also create a summary.klog file with summary information. This is essentially the operation done at the outset of launching the alogview applicaton.

73 The Alog Toolbox Command Line Utilities

73.1 Overview

The Alog Toolbox, in addition to the alogview GUI based utility, also contains a set of command line post-mission analysis utility applications:

- alogclip
- aloggrep
- alogrm
- alogscan
- aloghelm
- alogiter
- alogsplit
- alogpare
- alogcd
- alogcat
- alogavg
- alogmhash
- alognpos

Each application manipulates or renderings `.alog` files generated by the pLogger application. Four of the applications, alogclip, aloggrep, alogpare, and alogrm are command-line tools for filtering a given `.alog` file to a reduced size. Reduction of a log file size may facilitate the time to load a file in a post-processing application, may facilitate its transmission over slow transmission links when analyzing data between remote users, or may simply ease in the storing and back-up procedures. The alogscan tool provides statistics on a given `.alog` file that may indicate how to best reduce file size by eliminating variable entries not used in post-processing. It also generates other information that may be handy in debugging a mission. The alogsplit tool will split a single alog file into a folder containing a dedicated alog file for each logged variable. This operation is also done automatically upon the launch of appalogview when launched on an alog file for the first time. The alogview tool is a GUI-based tool that accepts one or more `.alog` files and renders a vehicle positions over time on an operation area, provides time-correlated plots of any logged numerical MOOS variables, and renders helm autonomy mode data with plots of generated objective functions.

73.2 An Example .alog File

The `.alog` file used in the examples below was generated from the Alpha example mission. This file, `alpha.alog`, is found in the missions distributed with the MOOS-IvP tree. The `alpha.alog` file was created by simply running the mission as described, and can be found in:

```
moos-ivp/trunk/ivp/missions/alpha/alpha.alog
```

74 The alogscan Tool

The alogscan tool is a command-line application for providing statistics relating to a given .alog file. It reports, for each unique MOOS variable in the log file, (a) the number of lines in which the variable appears, i.e., the number of times the variable was posted by a MOOS application, (b) the total number of characters comprising the variable value for all entries of a variable, (c) the timestamp of the first recorded posting of the variable, (d) the timestamp of the last recorded posting of the variable, (e) the list of MOOS applications the posted the variable.

74.1 Command Line Usage for the alogscan Tool

The alogscan tool is run from the command line with a given .alog file and a number of options. The usage options are listed when the tool is launched with the -h switch:

```
$ alogscan --help or -h
```

```
 1  Usage:
 2    alogscan file.alog [OPTIONS]
 3
 4  Synopsis:
 5    Generate a report on the contents of a given
 6    MOOS .alog file.
 7
 8  Options:
 9  --sort=type   Sort by one of SIX criteria:
10                start: sort by first post of a var
11                stop:  sort by last post of a var
12     (Default)  vars:  sort by variable name
13                proc:  sort by process/source name
14                chars: sort by total chars for a var
15                lines: sort by total lines for a var
16
17  --appstat     Output application statistics
18  -r,--reverse  Reverse the sorting output
19  -n,--nocolors Turn off process/source color coding
20  -h,--help     Displays this help message
21  -v,--version  Displays the current release version
22  --rate_only   Only report the data rate
23  --noaux       Ignore auxilliary source info
24
25  See also: aloggrp, alogrm, alogclip, alogview
```

The order of the arguments passed to alogscan do not matter. The lines of output are sorted by grouping variables posted by the same MOOS process or source. The sorting criteria can instead be done by alphabetical order on the variable name (--sort=vars), the total characters in the file due to a variable (--sort=chars), the total lines in the file due to a variable (--sort=lines), the time of the first posting of the variable (--sort=start), or the time of the last posting of the variable (--sort=stop). The order of the output may be reversed (-r, --reverse). By default, the entries are color-coded by the variable source, using the few available terminal colors (there are not many). When unique colors are exhausted, the color reverts back to the default terminal color in effect at

the time.

74.2 Example Output from the alogscan Tool

The output shown below was generated from the `alpha.alog` file generated by the Alpha example mission.

```
$ alogscan file.alog
```

```
Variable Name           Lines   Chars   Start    Stop  Sources
-------------           -----   -----  ------  ------  -------
DB_CLIENTS                282   22252   -0.38  566.42  MOOSDB_alpha
DB_TIME                   556    7132    1.21  566.18  MOOSDB_alpha
DB_UPTIME                 556    7173    1.21  566.18  MOOSDB_alpha
USIMMARINE_STATUS         276   92705    0.39  565.82  uSimMarine
NAV_DEPTH                6011    6011    1.43  566.38  uSimMarine
NAV_HEADING              6011   75312    1.43  566.38  uSimMarine
NAV_LAT                  6011   74799    1.43  566.38  uSimMarine
NAV_LONG                 6011   80377    1.43  566.38  uSimMarine
NAV_SPEED                6011    8352    1.43  566.38  uSimMarine
NAV_STATE                6011   18033    1.43  566.38  uSimMarine
NAV_X                    6011   72244    1.43  566.38  uSimMarine
NAV_Y                    6011   77568    1.43  566.38  uSimMarine
NAV_YAW                  6011   80273    1.43  566.38  uSimMarine
BHV_IPF                  2009  564165   46.26  542.85  pHelmIvP
CREATE_CPU               2108    2348   46.26  566.33  pHelmIvP
CYCLE_INDEX                 5       5   44.98  543.09  pHelmIvP
DEPLOY                      3      14    3.84  543.09  pHelmIvP,pMarineViewer
DESIRED_HEADING          2017    5445    3.85  543.09  pHelmIvP
DESIRED_SPEED            2017    2017    3.85  543.09  pHelmIvP
HELM_IPF_COUNT           2108    2108   46.26  566.32  pHelmIvP
HSLINE                      1       3    3.84    3.84  pHelmIvP
IVPHELM_DOMAIN              1      29    3.84    3.84  pHelmIvP
IVPHELM_ENGAGED           462    3342    3.85  566.32  pHelmIvP
IVPHELM_MODESET             1       0    3.84    3.84  pHelmIvP
IVPHELM_POSTINGS         2014  236320   46.26  543.33  pHelmIvP
IVPHELM_STATEVARS           1      20   44.98   44.98  pHelmIvP
IVPHELM_SUMMARY          2113  612685   44.98  566.33  pHelmIvP
LOOP_CPU                 2108    2348   46.26  566.33  pHelmIvP
PC_hsline                   1       9   44.98   44.98  pHelmIvP
PC_waypt_return             3      14   44.98  543.33  pHelmIvP
PC_waypt_survey             3      14   44.98  543.33  pHelmIvP
PHELMIVP_STATUS           255  198957    3.85  565.12  pHelmIvP
PLOGGER_CMD                 1      17    3.84    3.84  pHelmIvP
PWT_BHV_HSLINE              1       1   44.98   44.98  pHelmIvP
PWT_BHV_WAYPT_RETURN        3       5   44.98  543.09  pHelmIvP
PWT_BHV_WAYPT_SURVEY        2       4   44.98  462.90  pHelmIvP
RETURN                      4      19    3.84  543.09  pHelmIvP,pMarineViewer
STATE_BHV_HSLINE            1       1   44.98   44.98  pHelmIvP
STATE_BHV_WAYPT_RETURN      4       4   44.98  543.33  pHelmIvP
STATE_BHV_WAYPT_SURVEY      3       3   44.98  463.15  pHelmIvP
SURVEY_INDEX               10      10   44.98  429.70  pHelmIvP
SURVEY_STATUS            1116   77929   45.97  462.90  pHelmIvP
VIEW_POINT               4034  101662   44.98  543.33  pHelmIvP
VIEW_SEGLIST                4     273   44.98  543.33  pHelmIvP
WPT_INDEX                   1       1  463.15  463.15  pHelmIvP
WPT_STAT                  223   15626  463.15  543.09  pHelmIvP
LOGGER_DIRECTORY           56    1792    1.07  559.19  pLogger
PLOGGER_STATUS            263  331114    1.07  566.40  pLogger
DESIRED_RUDDER          10185  150449   -9.28  545.18  pMarinePID
DESIRED_THRUST          10637   20774   -9.28  566.52  pMarinePID
MOOS_DEBUG                  5      39   -9.31  545.23  pMarinePID,pHelmIvP
PMARINEPID_STATUS         279   81990    0.95  566.28  pMarinePID
HELM_MAP_CLEAR              1       1   -1.56   -1.56  pMarineViewer
MOOS_MANUAL_OVERIDE         1       5   44.65   44.65  pMarineViewer
PMARINEVIEWER_STATUS      270   95560   -0.95  564.89  pMarineViewer
NODE_REPORT_LOCAL        1159  207535    1.15  565.91  pNodeReporter
PNODEREPORTER_STATUS      233   50534   -0.37  563.93  pNodeReporter
-------------------------------------------------------------------
Total variables: 57
Start/Stop Time: -9.31 / 566.52
```

When the -appstat command line option is included, a second report is generated, after the above report, that provides statistics keyed by application, rather than by variable. For each application that has posted a variable recorded in the given .alog file, the number of lines and characters are recorded, as well as the percentage of total lines and characters. An example of this report:

```
MOOS Application     Total Lines  Total Chars  Lines/Total  Chars/Total
---------------      -----------  -----------  -----------  -----------
MOOSDB_alpha                1394        36557         1.37         1.08
uSimMarine                 54375       585674        53.57        17.29
pHelmIvP                   22642      1825437        22.31        53.89
pLogger                      319       332906         0.31         9.83
pMarinePID                 21106       253252        20.80         7.48
pMarineViewer                279        95599         0.27         2.82
pNodeReporter               1392       258069         1.37         7.62
```

Further Tips

- If a small number of variables are responsible for a relatively large portion of the file size, and are expendable in terms of how data is being analyzed, the variables may be removed to ease the handling, transmission, or storage of the data. To remove variables from existing files, the alogrm tool described in Section 77 may be used. To remove the variable from future files, the pLogger configuration may be edited by either removing the variable from the list of variables explicitly requested for logging, or if WildCardLogging is used, mask out the variable with the WildCardOmitPattern parameter setting. See the pLogger documentation.
- The output of alogscan can be further distilled using common tools such as grep. For example, if one only wants a report on variables published by the pHelmIvP application, one could type:

```
$ alogscan alpha.alog | grep pHelmIvP
```

75 The alogclip Tool

The alogclip tool will prune a given .alog file based on a given beginning and end timestamp. This is particularly useful when a log file contains a sizeable stretch of data logged after mission completion, such as data being recorded while the vehicle is being recovered or sitting idle topside after recovery.

75.1 Command Line Usage for the alogclip Tool

The alogclip tool is run from the command line with a given .alog file, a start time, end time, and the name of a new .alog file. By default, if the named output file exists, the user will be prompted before overwriting it. The user prompt can be bypassed with the -f,--force option. The usage options are listed when the tool is launched with the -h switch:

```
$ alogclip --help or -h
```

```
Usage:
  alogclip in.alog mintime maxtime [out.alog] [OPTIONS]

Synopsis:
  Create a new MOOS .alog file from a given .alog file
  by removing entries outside a given time window.

Standard Arguments:
  in.alog  - The input logfile.
  mintime  - Log entries with timestamps below mintime
             will be excluded from the output file.
  maxtime  - Log entries with timestamps above mintime
             will be excluded from the output file.
  out.alog - The newly generated output logfile. If no
             file provided, output goes to stdout.

Options:
  -h,--help     Display this usage/help message.
  -v,--version  Display version information.
  -f,--force    Overwrite an existing output file
  -q,--quiet    Verbose report suppressed at conclusion.

Further Notes:
  (1) The order of arguments may vary. The first alog
      file is treated as the input file, and the first
      numerical value is treated as the mintime.
  (2) Two numerical values, in order, must be given.
  (3) See also: alogscan, alogrm, aloggrep, alogview
```

75.2 Example Output from the alogclip Tool

The output shown below was generated from the `alpha.alog` file generated by the Alpha example mission.

```
$ alogclip alpha.alog new.alog 50 350
```

```
Processing input file alpha.alog...

Total lines clipped:      44,988  (44.32 pct)
  Front lines clipped:     5,474
  Back  lines clipped:    39,514
Total chars clipped:   4,200,260  (43.09 pct)
  Front chars clipped:   432,409
  Back  chars clipped: 3,767,851
```

76 The aloggrep Tool

The aloggrep tool has two primary purposes. The first is to prune a given `.alog` file into a smaller, yet syntactically valid `.alog` file. It this mode it can also be used for enforcing strict ordering of entries, and removing duplicate entries, since the pLogger app does not guarantee perfect ordering or files without the occasional duplicate entry. The second primary purpose is to extract data for

plotting in a tool like matlab or similar, or extracting a single value that may be ingested in a shell script. These use cases are described below.

A concise form of this documentation is available with `aloggrep --help`, and the web version of this content can be quickly opened with `aloggrep --web`.

76.1 Using aloggrep to Produce a Reduced and/or Ordered Output

The aloggrep tool will prune a given `.alog` file by retaining lines of the original file that contain log entries for a user-specified list of MOOS variables or MOOS processes (sources). As the name implies it is motivated by the Unix `grep` command, but `grep` will return a matched line *regardless* of where the pattern appears in the line. Since MOOS variables also often appear in the string content of other MOOS variables, `grep` often returns much more than one is looking for. The aloggrep tool will only pattern-match on the second column of data (the MOOS variable name), or the third column of data (the MOOS source), of any given entry in a given `.alog` file.

For example, try running the alpha mission and let the vehicle traverse the waypoints for a little while. Quit the mission and find the `.alog` file.

```
$ cd alpha_4_1_2022_____18_33_46/
$ ls
alpha._bhv                          alpha_4_1_2022_____18_33_46.blog
alpha_4_1_2022_____18_33_46._moos   alpha_4_1_2022_____18_33_46.xlog
alpha_4_1_2022_____18_33_46.alog    alpha_4_1_2022_____18_33_46.ylog
```

Then we can use aloggrep to see what is happening with the DEPLOY and WFLAG variable. The former is used for starting the mission, while the latter is incremented each time the vehicle hits a waypoint:

```
$ aloggrep DEPLOY WFLAGalpha_4_1_2022_____18_33_46.alog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOG FILE:       ./alpha_4_1_2022_____18_33_46/alpha_4_1_2022_____18_33_46.alog
%% FILE OPENED ON  Wed Dec 31 19:00:00 1969
%% LOGSTART           41033480654.29403
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
26.28008        DEPLOY               pHelmIvP:HELM_VAR_INIT false
26.28008        DEPLOY               pHelmIvP:HELM_VAR_INIT false
26.28008        DEPLOY               pHelmIvP:HELM_VAR_INIT false
26.28008        DEPLOY               pHelmIvP:HELM_VAR_INIT false
26.28008        DEPLOY               pHelmIvP:HELM_VAR_INIT false
26.28008        DEPLOY               pHelmIvP:HELM_VAR_INIT false
74.65635        DEPLOY               pMarineViewer   true
74.65635        DEPLOY               pMarineViewer   true
74.65635        DEPLOY               pMarineViewer   true
85.23170        WFLAG                pHelmIvP:38:waypt_survey waypoints=1
116.06017       WFLAG                pHelmIvP:148:waypt_survey waypoints=2
74.65635        DEPLOY               pMarineViewer   true
116.06017       WFLAG                pHelmIvP:148:waypt_survey waypoints=2
140.11575       WFLAG                pHelmIvP:234:waypt_survey waypoints=3
158.09377       WFLAG                pHelmIvP:298:waypt_survey waypoints=4
176.15134       WFLAG                pHelmIvP:362:waypt_survey waypoints=5
74.65635        DEPLOY               pMarineViewer   true
176.15134       WFLAG                pHelmIvP:362:waypt_survey waypoints=5
199.22812       WFLAG                pHelmIvP:444:waypt_survey waypoints=6
230.22727       WFLAG                pHelmIvP:554:waypt_survey waypoints=7
254.00302       WFLAG                pHelmIvP:640:waypt_survey waypoints=8
  Total lines retained: 21 (0.05%)
  Total lines excluded: 44238 (99.95%)
  Total chars retained: 1435 (0.04%)
  Total chars excluded: 3873898 (99.96%)
    Variables retained: (2) DEPLOY, WFLAG
```

The last five lines provide a summary contained retained and excluded. All lines above that are essentially a valid `.alog` file. Note however that there are some duplicate lines, and a couple lines out of order. To order the output and remove duplicates, try running with the `--sort,-s` and `--duplicates,-d` options, or simply the `-sd` option to do both:

```
$ aloggrep DEPLOY WFLAG alpha_4_1_2022_____18_33_46.alog -sd
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOG FILE:       ./alpha_4_1_2022_____18_33_46/alpha_4_1_2022_____18_33_46.alog
%% FILE OPENED ON  Wed Dec 31 19:00:00 1969
%% LOGSTART           41033480654.29403
26.28008        DEPLOY               pHelmIvP:HELM_VAR_INIT false
74.65635        DEPLOY               pMarineViewer   true
85.23170        WFLAG                pHelmIvP:38:waypt_survey waypoints=1
116.06017       WFLAG                pHelmIvP:148:waypt_survey waypoints=2
140.11575       WFLAG                pHelmIvP:234:waypt_survey waypoints=3
158.09377       WFLAG                pHelmIvP:298:waypt_survey waypoints=4
176.15134       WFLAG                pHelmIvP:362:waypt_survey waypoints=5
199.22812       WFLAG                pHelmIvP:444:waypt_survey waypoints=6
230.22727       WFLAG                pHelmIvP:554:waypt_survey waypoints=7
254.00302       WFLAG                pHelmIvP:640:waypt_survey waypoints=8
  Total re-sorts: 2
  Total lines retained: 10 (0.02%)
  Total lines excluded: 44238 (99.98%)
  Total chars retained: 723 (0.02%)
  Total chars excluded: 3873898 (99.98%)
    Variables retained: (2) DEPLOY, WFLAG
```

Notice the duplicates are removed, and the entries are ordered. Note also that, with alog tools, the order of the arguments on the command line do not matter. In the above example, the report section on the bottom now also indicates how many lines needed to be re-sorted, in this case 2.

76.1.1 Creating a New ALog File

Notice that the output above is not quite a syntactically valid `.alog` file since the report lines at the end are not log lines. So if output is simply re-directed to a file, the following would produce an `.alog` file with essentially garbage at the end:

```
$ aloggrep DEPLOY WFLAG alpha_4_1_2022_____18_33_46.alog -sd > newfile.alog
```

To create a new logfile either suppress the report lines at the end with:

```
$ aloggrep DEPLOY WFLAG oldfile.alog -sd --no_report > newfile.alog
```

Or simply provide a second (new) file name and the new file will not include the report lines:

```
$ aloggrep DEPLOY WFLAG alpha_4_1_2022_____18_33_46.alog -sd  newfile.alog
  Total re-sorts: 2
  Total lines retained: 10 (0.02%)
  Total lines excluded: 44238 (99.98%)
  Total chars retained: 723 (0.02%)
  Total chars excluded: 3873898 (99.98%)
    Variables retained: (2) DEPLOY, WFLAG
```

Note the report lines still go to the terminal, but the log files are written to the new `.alog` file.

76.1.2 Filtering based on the Data Source (Publishing App)

Rather than naming variables to keep, `aloggrep` can be provided with the name of one or more apps, and all entries published by these apps will be retained.

```
$ aloggrep pHelmIvP alpha_4_1_2022_____18_33_46.alog
```

76.1.3 Wildcard Matching

Limited simple wildcard matching is supported:

```
$ aloggrep NAV_*  file.alog
```

The above will grab all variables beginning with `NAV_`. Placing the asterisk in any other position will have no effect, e.g., `*_SPEED` will not grap `NAV_SPEED`.

76.2 Using aloggrep to Extract Plottable Data

An additional use case for `aloggrep` involves getting data for plotting. Using the same example as above, suppose we want to plot the value of `WPT_ODO` versus time. This variable represents vehicle odometry data, published by the waypoint behavior, and reset to zero on each waypoint. Using the above arguments we get:

```
$ aloggrep alpha_4_1_2022_____18_33_46.alog WPT_ODO
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOG FILE:       ./alpha_4_1_2022_____18_33_46/alpha_4_1_2022_____18_33_46.alog
%% FILE OPENED ON  Wed Dec 31 19:00:00 1969
%% LOGSTART           41033480654.29403
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
74.83427        WPT_ODO        pHelmIvP:1:waypt_survey   0.00000
75.90277        WPT_ODO        pHelmIvP:5:waypt_survey   0.34869
76.20800        WPT_ODO        pHelmIvP:6:waypt_survey   0.57995
76.49590        WPT_ODO        pHelmIvP:7:waypt_survey   0.88998
76.77897        WPT_ODO        pHelmIvP:8:waypt_survey   1.42170
77.08220        WPT_ODO        pHelmIvP:9:waypt_survey   2.79976
77.39445        WPT_ODO        pHelmIvP:10:waypt_survey  3.53386
77.67854        WPT_ODO        pHelmIvP:11:waypt_survey  4.38950
77.95012        WPT_ODO        pHelmIvP:12:waypt_survey  5.40420

...

  Total lines retained: 571 (1.29%)
  Total lines excluded: 43688 (98.71%)
  Total chars retained: 43305 (1.12%)
  Total chars excluded: 3832028 (98.88%)
    Variables retained: (1) WPT_ODO
```

The above output does grab the relevant lines, but we'd like to drop the header lines and report at

the end. Using the `--format=time:val` of `--tv` option, the results are stripped down to data lines only and just the time and value columns are retained:

```
$ aloggrep alpha_4_1_2022_____18_33_46.alog WPT_ODO --format=time:val
74.83427 0.00000
75.90277 0.34869
76.20800 0.57995
76.49590 0.88998
76.77897 1.42170
77.08220 2.79976
77.39445 3.53386
77.67854 4.38950
77.95012 5.40420
...
```

76.2.1 Extracting Plottable Data from a Complex Posting

Not all posted data is in simple numerical format as in case above with the variable `WPT_ODO`. Suppose for example, instead of `WPT_ODO`, our objective is to plot the value of the waypoint index versus time. This value, in the Alpha mission, is published as part of a string:

```
$ aloggrep alpha_4_1_2022_____18_33_46.alog WFLAG --format=time:val
85.23170 waypoints=1
116.06017 waypoints=2
140.11575 waypoints=3
158.09377 waypoints=4
176.15134 waypoints=5
199.22812 waypoints=6
230.22727 waypoints=7
254.00302 waypoints=8
```

In this case, we would like to isolate the numerical value from the string. Using the `--subpat=PATTERN` option we can isolate numerical values from variable postings made in the common format of:

```
field1=value1, field2=value2, ..., fieldN=valueN
```

```
$ aloggrep alpha_4_1_2022_____18_33_46.alog WFLAG --format=time:val --subpat=waypoints
85.23170 1
116.06017 2
140.11575 3
158.09377 4
176.15134 5
199.22812 6
230.22727 7
254.00302 8
```

Note in the above example, the column separator is single white space character (ASCII 32). If a colon separator is desired, use the `--cso` flag. If a comma separator is desired, use the `--csc` flag.

76.3 Extracting a First or Final Posting

In some cases, aloggrep may be used as a tool within a script to determine mission outcome. If the mission outcome is represented in a single MOOS variable, e.g., MISSION_RESULT or MISSION_SCORE, then the goal is to isolate just this value. Using the --first or --final option, we can reduce the output to just one line. Using the example above:

```
$ aloggrep alpha_4_1_2022_____18_33_46.alog WFLAG --format=time:val --final
254.00302 8
```

To isolate just the value column, use the --format=val, or --v option:

```
$ aloggrep alpha_4_1_2022_____18_33_46.alog WFLAG --v --final
8
```

The style of output can then be used within a shell script to take action based on the result, stored in a shell variable:

```
$ FOO=(‘aloggrep alpha_4_1_2022_____18_33_46.alog WFLAG --v --final‘)
$ echo $FOO
8
```

```
$ FOO=(‘aloggrep alpha_10_5_2022_____14_31_06.alog MISSION_HASH --v --first‘)
$ echo $MISSION_HASH
2211-0528V-SOUR-CLUB
```

77 The alogrm Tool

The alogrm tool will prune a given .alog file by removing lines of the original file that contain log entries for a user-specified list of MOOS variables or MOOS processes (sources). It may be fairly viewed as the complement of the aloggrep tool.

77.1 Command Line Usage for the alogrm Tool

```
 $ alogrm --help or -h
```

```
 Usage:
  alogrm in.alog [VAR] [SRC] [out.alog] [OPTIONS]

Synopsis:
   Remove the entries matching the given MOOS variables or sources
   from the given .alog file and generate a new .alog file.

Standard Arguments:
  in.alog  - The input logfile.
  out.alog - The newly generated output logfile. If no
             file provided, output goes to stdout.
  VAR      - The name of a MOOS variable
  SRC      - The name of a MOOS process (source)

Options:
  -h,--help     Displays this help message
  -v,--version  Displays the current release version
  -f,--force    Force overwrite of existing file
  -q,--quiet    Verbose report suppressed at conclusion
  --nostr       Remove lines with string data values
  --nonum       Remove lines with double data values
  --clean       Remove lines that have a timestamp that is
                non-numerical or lines w/ no 4th column

Further Notes:
  (1) The second alog is the output file. Otherwise the
      order of arguments is irrelevent.
  (2) VAR* matches any MOOS variable starting with VAR
  (3) See also: alogscan, aloggrep, alogclip, alogview
```

Note that, in specifying items to be filtered out, there is no distinction made on the command line that a given item refers to a entry's variable name or an entry's source, i.e., MOOS process name.

77.2 Example Output from the alogrm Tool

The output shown below was generated from the `alpha.alog` file generated by the Alpha example mission.

```
$ alogrm alpha.alog NAV_* new.alog
```

```
Processing on file : alpha.alog
  Total lines retained: 47396 (46.70%)
  Total lines excluded: 54099 (53.30%)
  Total chars retained: 6453494 (66.21%)
  Total chars excluded: 3293774 (33.79%)
  Variables retained: (48) BHV_IPF, CREATE_CPU, CYCLE_INDEX, DB_CLIENTS,
DB_TIME, DB_UPTIME, DEPLOY, DESIRED_HEADING, DESIRED_RUDDER, DESIRED_SPEED,
DESIRED_THRUST, HELM_IPF_COUNT, HELM_MAP_CLEAR, HSLINE, USIMMARINE_STATUS,
IVPHELM_DOMAIN, IVPHELM_ENGAGED, IVPHELM_MODESET, IVPHELM_POSTINGS,
IVPHELM_STATEVARS, IVPHELM_SUMMARY, LOGGER_DIRECTORY, LOOP_CPU, MOOS_DEBUG,
MOOS_MANUAL_OVERIDE, NODE_REPORT_LOCAL, PC_hsline, PC_waypt_return,
PC_waypt_survey, PHELMIVP_STATUS, PLOGGER_CMD, PLOGGER_STATUS,
PMARINEPID_STATUS, PMARINEVIEWER_STATUS, PNODEREPORTER_STATUS,
PWT_BHV_HSLINE, PWT_BHV_WAYPT_RETURN, PWT_BHV_WAYPT_SURVEY, RETURN,
STATE_BHV_HSLINE, STATE_BHV_WAYPT_RETURN, STATE_BHV_WAYPT_SURVEY,
SURVEY_INDEX, SURVEY_STATUS, VIEW_POINT, VIEW_SEGLIST, WPT_INDEX, WPT_STAT
```

78 The aloghelm Tool

The `aloghelm` tool provides a few handy ways of looking at helm activity over the course of a given single alog file. This includes:

- *Life Events*: Using the `--life/-l` option, every spawning or death of a behavior is sorted into a list of life events. Section 78.1.
- *Mode Changes*: Using the `--modes/-m` option, every helm mode change is sorted into a list of chronological entries. Section 78.2.
- *Behavior States*: Using the `--bhvs/-b` option, every instance where a behavior changes states is recorded and sorted into a list of chronological entries. Section 78.3.

In each mode, the user may additionally specify one or more MOOS variables to be interleaved in the report as they occur chronologically

78.1 The Life Events (–life) Option in the aloghelm Tool

The *life events* option in `aloghelm` will scan the given alog file for all life events, defined by the spawning or destruction of a behavior instance. This information is posted by the helm in the `IVPHELM_LIFE_EVENT` variable. Example output is show below:

```
$ aloghelm file.alog --life
```

```
Processing on file : henry.alog
++++++++++ (100,000) lines
++++++++++ (200,000) lines
+++
233,736 lines total.
10 life events.


      ************************************************
      *        Summary of Behavior Life Events        *
      ************************************************

Time    Iter  Event  Behavior      Behavior Type       Spawning Seed
------  ----  -----  ------------  ------------------  ------------------------------
 41.27     1  spawn  loiter        BHV_Loiter          helm_startup
 41.27     1  spawn  waypt_return  BHV_Waypoint        helm_startup
 41.27     1  spawn  station-keep  BHV_StationKeep     helm_startup
316.78   995  spawn  ac_avd_gilda  BHV_AvoidCollision  name=avd_gilda # contact=gilda
369.72  1191  death  ac_avd_gilda  BHV_AvoidCollision
482.92  1601  spawn  ac_avd_gilda  BHV_AvoidCollision  name=avd_gilda # contact=gilda
545.18  1833  death  ac_avd_gilda  BHV_AvoidCollision
654.70  2228  spawn  ac_avd_gilda  BHV_AvoidCollision  name=avd_gilda # contact=gilda
751.87  2591  death  ac_avd_gilda  BHV_AvoidCollision
809.85  2799  spawn  ac_avd_gilda  BHV_AvoidCollision  name=avd_gilda # contact=gilda
```

The actual output, by default, is color-code green for all spawnings and black for all deaths. The color-coding can be turned off with the additional command line argument `--nocolor`.

78.2 The Modes (–modes) Option in the aloghelm Tool

The *modes* option in `aloghelm` will scan the given alog and report all instances of a helm mode change.

```
$ aloghelm file.alog --modes
```

```
Processing on file : /Users/mikerb/henry.alog
================================================
45.221 Mode: ACTIVE:LOITERING
================================================
92.687 Mode: ACTIVE:STATION-KEEPING
================================================
120.919 Mode: ACTIVE:LOITERING
================================================
386.632 Mode: ACTIVE:RETURNING
================================================
413.980 Mode: ACTIVE:LOITERING
================================================
558.254 Mode: ACTIVE:RETURNING
================================================
584.283 Mode: ACTIVE:LOITERING
================================================
663.162 Mode: ACTIVE:STATION-KEEPING
================================================
703.517 Mode: ACTIVE:LOITERING
================================================
766.938 Mode: ACTIVE:RETURNING

233,736 lines total.
```

Using the `--mode` option, it is sometimes helpful to augment the output to include certain other variable postings, by simply naming the MOOS variable on the command line. The variables and mode changes will be presented on the screen in their chronological order. For example:

```
$ aloghelm file.alog --modes CONTACT_RESOLVED
```

```
Processing on file : /Users/mikerb/henry.alog
=================================================
45.221 Mode: ACTIVE:LOITERING
=================================================
92.687 Mode: ACTIVE:STATION-KEEPING
=================================================
120.919 Mode: ACTIVE:LOITERING
373.392         CONTACT_RESOLVED     pHelmIvP:1190:ac_avd_gilda GILDA
=================================================
386.632 Mode: ACTIVE:RETURNING
=================================================
413.980 Mode: ACTIVE:LOITERING
548.838         CONTACT_RESOLVED     pHelmIvP:1832:ac_avd_gilda GILDA
=================================================
558.254 Mode: ACTIVE:RETURNING
=================================================
584.283 Mode: ACTIVE:LOITERING
=================================================
663.162 Mode: ACTIVE:STATION-KEEPING
=================================================
703.517 Mode: ACTIVE:LOITERING
755.509         CONTACT_RESOLVED     pHelmIvP:2590:ac_avd_gilda GILDA
=================================================
766.938 Mode: ACTIVE:RETURNING

233,736 lines total.
```

78.3 The Behaviors Option in the aloghelm Tool

The *behaviors* option in `aloghelm` will scan the given alog file taking note of all helm iterations where there is a change to one or more of the four groups of (a) active, (b) running, (c) idle, or (d) completed behaviors. Example output is show below:

```
$ aloghelm file.alog --bhvs
```

```
Processing on file : henry.alog
================================================
45.221 Mode: ACTIVE:LOITERING
 - - - - - - - - - - - - - - - - - - - - - - - -
45.225    (1) Active:    loiter
45.225    (1) Running:
45.225    (1) Idle:      waypt_return,station-keep
================================================
92.687 Mode: ACTIVE:STATION-KEEPING
 - - - - - - - - - - - - - - - - - - - - - - - -
92.689    (172) Active:    station-keep
92.689    (172) Running:
92.689    (172) Idle:      loiter,waypt_return
================================================
120.919 Mode: ACTIVE:LOITERING
 - - - - - - - - - - - - - - - - - - - - - - - -
120.921    (274) Active:    loiter
120.921    (274) Running:
120.921    (274) Idle:      waypt_return,station-keep
 - - - - - - - - - - - - - - - - - - - - - - - -
320.786    (995) Active:    loiter,ac_avd_gilda
320.786    (995) Running:
320.786    (995) Idle:      waypt_return,station-keep
 - - - - - - - - - - - - - - - - - - - - - - - -
345.778    (1090) Active:    loiter
345.778    (1090) Running:   ac_avd_gilda
345.778    (1090) Idle:      waypt_return,station-keep
 - - - - - - - - - - - - - - - - - - - - - - - -
373.642    (1191) Active:    loiter
373.642    (1191) Running:
373.642    (1191) Idle:      waypt_return,station-keep
373.642    (1191) Completed: ac_avd_gilda
================================================
386.632 Mode: ACTIVE:RETURNING
 - - - - - - - - - - - - - - - - - - - - - - - -
386.636    (1238) Active:    waypt_return
386.636    (1238) Running:
386.636    (1238) Idle:      loiter,station-keep
386.636    (1238) Completed: ac_avd_gilda
================================================
```

In some cases, there is interest in a particular behavior in the this kind of output. To make it easier to visually parse, the `--watch=BHV` option can be used to draw attention to each the particular behavior changes state. Example output is shown below. The primary difference is the `CHANGE` tag for each instance of a state change. In the terminal, such lines are also rendered in a different color.

```
 $ aloghelm file.alog --bhvs --watch=loiter
```

```
Processing on file : henry.alog
================================================
45.221 Mode: ACTIVE:LOITERING
 - - - - - - - - - - - - - - - - - - - - - - - - -
45.225    (1) Active:    loiter             CHANGE
45.225    (1) Running:
45.225    (1) Idle:      waypt_return,station-keep
================================================
92.687 Mode: ACTIVE:STATION-KEEPING
 - - - - - - - - - - - - - - - - - - - - - - - - -
92.689    (172) Active:    station-keep
92.689    (172) Running:
92.689    (172) Idle:      loiter,waypt_return            CHANGE
================================================
120.919 Mode: ACTIVE:LOITERING
 - - - - - - - - - - - - - - - - - - - - - - - - -
120.921    (274) Active:    loiter             CHANGE
120.921    (274) Running:
120.921    (274) Idle:      waypt_return,station-keep
 - - - - - - - - - - - - - - - - - - - - - - - - -
320.786    (995) Active:    loiter,ac_avd_gilda
320.786    (995) Running:
320.786    (995) Idle:      waypt_return,station-keep
 - - - - - - - - - - - - - - - - - - - - - - - - -
345.778    (1090) Active:    loiter
345.778    (1090) Running:   ac_avd_gilda
345.778    (1090) Idle:      waypt_return,station-keep
 - - - - - - - - - - - - - - - - - - - - - - - - -
373.642    (1191) Active:    loiter
373.642    (1191) Running:
373.642    (1191) Idle:      waypt_return,station-keep
373.642    (1191) Completed: ac_avd_gilda
================================================
386.632 Mode: ACTIVE:RETURNING
 - - - - - - - - - - - - - - - - - - - - - - - - -
386.636    (1238) Active:    waypt_return
386.636    (1238) Running:
386.636    (1238) Idle:      loiter,station-keep            CHANGE
386.636    (1238) Completed: ac_avd_gilda
================================================
```

78.4 Command Line Usage for the aloghelm Tool

```
$ aloghelm --help or -h
```

Listing 78.41: Command line usage for the aloghelm tool.

```
1  Usage:
2    aloghelm file.alog [OPTIONS] [MOOSVARS]
3
4  Synopsis:
5    Perform one of several optional helm reports based on
6    helm output logged in the given .alog file.
7
8  Options:
```

```
 9    -h,--help     Displays this help message
10    -v,--version  Displays the current release version
11    -l,--life     Show report on IvP Helm Life Events
12    -b,--bhvs     Show helm behavior state changes
13    -m,--modes    Show helm mode changes
14    --watch=bhv   Watch a particular behavior for state change
15    --nocolor     Turn off use of color coding
16    --notrunc     Don't truncate MOOSVAR output (on by default)
17
18  Further Notes:
19    (1) The order of arguments is irrelevent.
20    (2) Only the first specified .alog file is reported on.
21    (3) Arguments that are not one of the above options or an
22        alog file, are interpreted as MOOS variables on which
23        to report as encountered.
```

79 The alogiter Tool

The alogiter tool will analyze the ITER_GAP and ITER_LEN information produced by any appcasting MOOS app. These variables indicate the ability of an application to keep up with the requested apptick frequency. For example PHELMIVP_ITER_GAP will be close to 1.0 when configured with an apptick of 4, and the observed apptick is also 4. The gap value will be around 2 if the observed apptick is around 2. The PHELMIVP_ITER_LEN is the elapsed time between the start and end of the helm iterate loop.

79.1 Command Line Usage for the alogiter Tool

```
$ alogiter --help or -h
```

Listing 79.42: Command line usage for the alogiter tool.

```
 1  $ alogrm -h
 2
 3  Usage:
 4    alogiter in.alog [OPTIONS]
 5
 6  Synopsis:
 7    Analyze the ITER_GAP and ITER_LEN information provided by
 8    all applications recorded in the given alog file.
 9
10  Standard Arguments:
11    file.alog - The input logfile.
12
13  Options:
14    -h,--help     Displays this help message
15    -v,--version  Displays the current release version
16
17  Further Notes:
18    See also: alogscan, alogrm, alogclip, alogview, aloggrep
```

79.2 Example Output from the alogiter Tool

The output shown in Listing 43 was generated from the alpha.alog file generated by the Alpha example mission, at time warp 20.

Listing 79.43: Example alogiter output applied to the alpha.alog file.

```
$ alogiter alpha.alog

Processing on file : MOOSLog_22_4_2015_____13_25_19.alog
                 GAP     GAP       PCT     PCT     PCT
AppName          MAX     AVG       >1.25   >1.50   >2.0
-------------  -----   -----      -----   -----   -----
PHELMIVP         1.26    1.11      0.005   0.000   0.000
PMARINEVIEWER    1.10    1.07      0.000   0.000   0.000
PNODEREPORTER    1.25    1.15      0.008   0.000   0.000
UPROCESSWATCH    1.26    1.12      0.014   0.000   0.000
USIMMARINE       1.27    1.12      0.009   0.000   0.000

                 LEN     LEN       PCT     PCT     PCT     PCT
AppName          MAX     AVG       >0.25   >0.50   >0.75   >1.0
-------------  -----   -----      -----   -----   -----   -----
PHELMIVP         0.08    0.04      0.000   0.000   0.000   0.000
PMARINEVIEWER    0.00    0.00      0.000   0.000   0.000   0.000
PNODEREPORTER    0.01    0.00      0.000   0.000   0.000   0.000
UPROCESSWATCH    0.01    0.00      0.000   0.000   0.000   0.000
USIMMARINE       0.02    0.00      0.000   0.000   0.000   0.000

Mission Summmary
--------------------------
 Collective APP_GAP: 1.11
 Collective APP_LEN: 0.01
```

80 The alogsplit Tool

The alogsplit tool will split a given .alog file into a directory containing a file for each MOOS variable found in the .alog file. This is essentially the first stage of pre-processing done at the outset of launching the alogview tool. It is implement here as a stand-alone app to be used for purposes other than alogview. It may also be useful as a command-line tool for preparing multiple .alog files from a shell script well before the first time they are used in alogview.

This tool in was introduced in Release 15.4 coinciding with the major re-write of the alogview tool also released in 15.4.

80.1 Naming and Cleaning the Auto-Generated Split Directories

The name of the *split directory* created by alogsplit is determined automatically from the .alog filename. For a file name alpha.alog, the directory created will be alpha_alvtmp/ by default. This can be overridden with the command line switch --dir=my_dirname. The fairly distinctive _alvtmp suffix was chosen to facilitate cleaning these auto-generated temporary directories with a simple shell script, alvrm.sh:

```
#!/bin/bash
find . -name '*_alvtmp'  -print -exec rm -rfv {} \;
```

The above script is found in the moos-ivp/scripts directory and will remove (without prompting for confirmation) all split directories in the current directory and sub-directories.

```
 $ alvrm.sh -d -v
```

80.2 Command Line Usage for the alogsplit Tool

```
$ alogsplit --help or -h
```

Listing 80.44: Command line usage for the `alogsplit` *tool.*

```
$ alogsplit -h

Usage:
  alogsplit in.alog [OPTIONS]

Synopsis:
  Split the given alog file into a directory, within which
  each MOOS variable is split into it's own (klog) file
  containing only that variable. The split will also create
  a summary.klog file with summary information.

  Given file.alog, file_alvtmp/ directory will be created.
  Will not overwrite directory if previously created.
  This is essentially the operation done at the outset of
  launching the alogview applicaton.

Standard Arguments:
  in.alog  - The input logfile.

Options:
 -h,--help      Displays this help message
 -v,--version   Displays the current release version
 --verbose      Show output for successful operation
 --dir=DIR      Override the default dir with given dir.

 --max_fptrs=N  Set max number of OS file pointers allowed
                to be open during splitting. Default 125.

 --detached=var:key  Split out key from complext var post

 --web,-w   Open browser to:
            https://oceanai.mit.edu/ivpman/apps/alogsplit

Further Notes:
 (1) The order of arguments is irrelevant.
 (2) See also: alogscan, alogrm, aloggrep, alogclip,
X137       alogview
```

80.3 Example Output from the alogsplit Tool

The output shown in Listing 45 was generated from the `alpha.alog` file generated by the Alpha example mission.

Listing 80.45: Example `alogsplit` *directory applied to the* `alpha.alog` *file.*

```
$ alogsplit alpha.alog

APPCAST.klog                 IVPHELM_CPU.klog             NODE_REPORT_LOCAL.klog
APPCAST_REQ.klog             IVPHELM_CREATE_CPU.klog      PHELMIVP_ITER_GAP.klog
APPCAST_REQ_ALL.klog         IVPHELM_DOMAIN.klog          PHELMIVP_ITER_LEN.klog
APPCAST_REQ_ALPHA.klog       IVPHELM_IPF_CNT.klog         PLOGGER_CMD.klog
BHV_IPF_waypt_return.klog    IVPHELM_ITER.klog            PMARINEVIEWER_ITER_GAP.klog
BHV_IPF_waypt_survey.klog    IVPHELM_LIFE_EVENT.klog      PMARINEVIEWER_ITER_LEN.klog
CYCLE_INDEXd.klog             IVPHELM_LOOP_CPU.klog        PMV_CONNECT.klog
CYCLE_INDEX_SURVEYING.klog   IVPHELM_MODESET.klog         PNODEREPORTER_ITER_GAP.klog
```

```
11  DB_CLIENTS.klog                IVPHELM_REGISTER.klog          PNODEREPORTER_ITER_LEN.klog
12  DB_EVENT.klog                  IVPHELM_STATE.klog             PROC_WATCH_EVENT.klog
13  DB_QOS.klog                    IVPHELM_STATEVARS.klog         PROC_WATCH_FULL_SUMMARY.klog
14  DB_RWSUMMARY.klog              IVPHELM_SUMMARY.klog           PROC_WATCH_SUMMARY.klog
15  DB_TIME.klog                   LOGGER_DIRECTORY.klog          PROC_WATCH_TIME_WARP.klog
16  DB_UPTIME.klog                 MOOS_DEBUG.klog                RETURN.klog
17  DEPLOY.klog                    MOOS_MANUAL_OVERRIDE.klog      SIMULATION_MODE.klog
18  DESIRED_HEADING.klog           NAV_DEPTH.klog                 TRUE_X.klog
19  DESIRED_RUDDER.klog            NAV_HEADING.klog               TRUE_Y.klog
20  DESIRED_SPEED.klog             NAV_HEADING_OVER_GROUND.klog   UPROCESSWATCH_ITER_GAP.klog
21  DESIRED_THRUST.klog            NAV_LAT.klog                   UPROCESSWATCH_ITER_LEN.klog
22  HELM_MAP_CLEAR.klog            NAV_LONG.klog                  USIMMARINE_ITER_GAP.klog
23  IVPHELM_ALLSTOP.klog           NAV_PITCH.klog                 USIMMARINE_ITER_LEN.klog
24  IVPHELM_ALLSTOP_DEBUG.klog     NAV_SPEED.klog                 USM_DRIFT_SUMMARY.klog
25  IVPHELM_BHV_ACTIVE.klog        NAV_SPEED_OVER_GROUND.klog     USM_FSUMMARY.klog
26  IVPHELM_BHV_CNT.klog           NAV_X.klog                     VISUALS.klog
27  IVPHELM_BHV_CNT_EVER.klog      NAV_Y.klog                     summary.klog
28  IVPHELM_BHV_IDLE.klog          NAV_YAW.klog
29  IVPHELM_BHV_RUNNING.klog       NAV_Z.klog
```

Notice the `summary.klog` file on line 27. It contains some meta information gathered during the split process that is useful for `alogview` in fetching information at run time.

80.4 Using the Detached Variable Option

Certain string log entries may be in the format of multi-part comma-separated pairs (CSP). For example:

```
REPORT =  x=77, y=99, speed=3.2, depth=4.5, heading=93, battery_level=16.4
```

Normally `alogsplit` would create a single `.klog` file, `REPORT.klog`. However, one or more fields may be desired for splitting out into their own `.klog` file. With the optional `--detached=VARNAME` command line argument, the variable `VARNAME` will be parsed and separate `.klog` files created for sub-components of the contents of the string. For example, the following argument:

```
$ alogsplit --detached=REPORT file.alog
```

would create the following `.klog` files:

```
REPORT:x.klog
REPORT:y.klog
REPORT:speed.klog
REPORT:depth.klog
REPORT:heading.klog
REPORT:batter_level.klog
```

If the run-time speed of `alogsplit` is a concern, especially on very large files, the above action can be quickened by isolating just the sub-component of interest:

```
$ alogsplit --detached=REPORT:speed file.alog
```

would create just the following `.klog` file:

```
REPORT:speed.klog
```

If two or more sub-components are desired, the following two equivalent invocations could be used:

```
$ alogsplit --detached=REPORT:speed --detached=REPORT:depth file.alog
$ alogsplit --detached=REPORT:speed:depth file.alog
```

would create just the following two `.klog` files:

```
REPORT:speed.klog
REPORT:depth.klog
```

Although `alogsplit` is its own stand-alone command-line utility, the functionality of splitting is largely used as a pre-cursor step when `alogview` is launched. As such, all the above command-line options are also relevant when launching `alogview`. For example:

```
$ alogview --detached=REPORT file.alog
```

As `alogview` is launched, the first step after launch will be the creation of the `_alvtmp` directories. If a detached variable is named, e.g., a file like `REPORT:depth.klog` is created, then `alogview` will now have this as a pull-down menu option for plotting.

Note, if the string is in JSON format, e.g.

```
REPORT =  {"x":77, "y":99, "speed":3.2, "depth":4.5, "heading":93}
```

then `alogview` will internally convert this into CSP format, and will handle accordingly.

81 The alogpare Tool

The `alogpare` tool is a utility for pruning alog files by removing certain alog entries outside certain time windows. The time windows are defined by a user-defined time duration around the entries of further user-defined variables in the log file. The idea is that some robot missions have events of interest, e.g., a near collision event, where retaining all data just before and after the event is critical to analyzing what may have gone wrong. Perhaps certain high data rate log entries outside these critical event windows may be removed without any loss in utility to the users. In some cases this reduction in logged data may dramatically ease the acrhiving of these log files.

This was a new tool in Release 17.7 but was not documented until the following release.

81.1 Mark Variables Define Events of Interest

A *mark variable* is a MOOS variable provided to `alogpare` on the command line to indicate an event of interest. From the perspective of `alogpare`, the value of the mark variable does not matter. One or variables may be provided. For example:

```
$ alogpare --markvars=ENCOUNTER,NEAR_MISS
```

The alogpare utility will make an initial pass through the alog file and make not of each instance of a mark variable. A window of time, given by the command line parameter `--pare_window`, will will be associated around each instance of a mark variable. If windows overlap, that's fine. The during of the pare window is 30 seconds by default, even split in time before and after the mark event. This may be adjusted on the command line. For example:

```
$ alogpare --markvars=ENCOUNTER --pare_window=60
```

The alogpare utility will make a second pass through the alog file pruning log entries *outside* the pare windows, based on variables on the pare list.

81.2 The Pare List of Variables to be Pared

Variables on the *pare list* indicate which lines of an alog file are to be removed, outside of pare windows. The pare list is defined on the command line with:

```
$ alogpare --markvars=ENCOUNTER --pare_window=60 --parevars=BHV_IPF,BIG_ENTRY
```

Typically these variables constitute relatively large portions of an alog file, and provide little value outside the pare windows.

81.3 The Hit List of Variables to be Removed Completely

The alogpare utility also provides the means for removing named variables outright, regardless of where they occur relative to a pare window. These variables are on the *hit list*. For example:

```
$ alogpare --varkvars=ENCOUNTER --parevars=BHV_IPF --hitvars=ITER_GAP
```

This functionality is also achieved with the alogrm utility, and is provided in this tool just as a convenience.

81.4 Command Line Usage for the alogpare Tool

```
 $ alogpare --help or -h
```

Listing 81.46: Command line usage for the alogpare tool.

```
1  $ alogpare -h
2
3    Usage:
4    alogpare .alog [out.alog] [OPTIONS]
5
6  Synopsis:
7    Pare back the given alog file in a two-pass manner.
8    First pass detects events defined by given mark vars.
9    The second pass removes lines with vars on the pare
```

```
10    list if they are not within pare_window seconds of
11    an event line. It also removes lines with vars on the
13    hitlist unconditionally. Latter could also be done
15    with alogrm.
16    The original alog file is not altered.
17
18  Options:
19    -h,--help         Displays this help message
20    -v,--version      Display current release version
21    --verbose         Enable verbose output
22    --markvars=<L>    Comma-separated list of mark vars
23    --hitvars=<L>     Comma-separated list of hit vars
24    --parevars=<L>    Comma-separated list of pare vars
25    --pare_window=<N> Set window to N seconds (default 30)
26
27  Examples:
28    alogpare --markvars=ENCOUNTER --parevars=BHV_IPF
29             original.alog smaller.alog
30    alogpare --markvars=ENCOUNTER
31             --parevars=BHV_IPF,VIEW_*
32             --hitvars=*ITER_GAP,*ITER_LEN,DB_QOS
33             --pare_window=10
34             original.alog smaller.alog
35
36  Further Notes:
37    (1) The order of alogfile args IS significant.
38    (2) The order of non alogfile args is not significant.
```

81.5 Planned additions to the alogpare Utility

- Soft parevars: removing perhaps every other entry outside pare window. Or remove success entries with identical values outside the pare window.
- Separate specification for pare_window time. Currently the window is split evenly around the mark event. Some user may want more control.
- Pattern matching: Add support for specifying sets of variables with simple wildcard prefix or suffix, e.g., `NAV_*` or `*_REPORT`.

82 The alogcd Tool

The `alogcd` tool is a utility for scanning a given alog file and tallying the number of encounters, near misses, and collisions. This utility works under the assumption that another utility had been running during the mission, and monitoring encounters, near misses and collisions. It assumes that these three events were separately noted with MOOS variables that also indicate the closest point of approach (CPA) range for each event. And it also assumes that these three variables were logged in the alog file.

The `alogcd` utility uses the MOOS variables `ENCOUNTER`, `NEAR_MISS`, and `COLLISION`. For now, these three variables are hard-coded in this utility. The `uFldCollisionDetect` utility is one utility capable of generating this kind of output. If there is collision to report, the report will also show the CPA value for the worst collision encounter.

An example run may produce output similar to:

```
$ alogcd file.alog

7,686 total alog file lines.

========================================
Collision Report:
========================================
Encounters:  27  (avg 16.93 m)
Near Misses: 6  (avg 10.18 m)
Collisions:  3  (avg 5.55 m)
Collision Worst: 3.87
```

82.1 Producing a Time-Stamped file of Collisions and Near Misses

The near misses and collisions are the real events of interests, and if the standard summary report is not enough, a time stamped list of each near miss and collision may be written to a file, if the `--tfile=filename` parameter is provided. For example, the six near misses and three collisions reported above could be written to file with:

```
$ alogcd file.alog --tfile=myfile
$ cat myfile
69.149,COLLISION,5.17
231.826,NEAR_MISS,9.41
351.374,NEAR_MISS,10.33
556.815,NEAR_MISS,10.35
592.976,NEAR_MISS,9.69
792.884,COLLISION,3.87
1065.484,NEAR_MISS,11.51
1129.862,COLLISION,7.61
1275.018,NEAR_MISS,9.82
```

The first column is the timestamp from the alog file, the second column is the type of encounter (near miss or collision), and the third column is the CPA distance for that encounter.

82.2 The Terse Output Option

For a super terse, one line report, use the following, which produces the below output for the same alog file as in the example above:

```
$ alogcd file.alog
27/6/3
```

27 encounters, 6 near misses, 3 collisions.

82.3 Command Line Return Values

The ultimate terse output is none at all! In this case we're only interested in the return value of `alogcd`. This can be used for example in a shell script to launch a series of simulations, altering the

configuration parameters until no collisions are detected. The following (integer) return values are implemented:

- `[0]`: The alog file was found and readable, encounters were indeed reported, and no collisions were reported. The success condition.
- `[1]`: The alog file was not found or it was not readable.
- `[2]`: The alog file was indeed found and was readable, but sadly, collisions were reported.
- `[3]`: The alog file was indeed found and was readable, and no collisions were reported, but no encounters were reported either. Something is amiss. Either the vehicles never even got close enough to each other to constitute an encounter, or a monitoring app like `uFldCollisionDetect` was not even running.

82.4 Command Line Usage for the alogcd Tool

```
$ alogcd --help or -h
```

Listing 82.47: Command line usage for the `alogcd` tool.

```
$ alogcd -h

Usage:
  alogcd .alog [OPTIONS]

Synopsis:
  Scan an alog file for collision detection reports.
  Tally the totals and averages, and optionally create
  a file holding all the timestamps of events.

  By default, it scans for events defined by postings
  to the following three MOOS variables:

  (1) COLLISION
  (2) NEAR_MISS
  (3) ENCOUNTER

Options:
  -h,--help       Displays this help message
  -v,--version    Display current release version
  -t,--terse      Write terse output.

Returns:
  0 if alog file ok, encounters detected, no collisions.
  1 if alog file not ok, unable to open.
  2 if alog ok, but collisions were detected
  3 if alog ok, no collisions or encounters detected
```

82.5 Planned additions to the alogcd Utility

- Allow the key MOOS variables to be provided as parameters, rather then fixed to `ENCOUNTER`, `NEAR_MISS`, and `COLLISION`.
- Support cmd line option like `--collision_count` which produces the integer value of collision counts as the command line return value. Perhaps the same for `--near_miss_count` or `encounter_count`.

83 The alogcat Tool

A concise form of this documentation is available with `alogcat --help, -h`, and the web version of this content can be quickly opened with `alogcat --web, -w`.

The alogcat tool is a utility for concatenating a given set of alog files into a new single alog file. Recall that each alog file has a header at the beginning of the file with meta information. This includes the starting timestamp which allows all further timestamps to be relative to the starting time. So to concatenate alog files, we cannot simply just append one file onto the end another. If that were done, there would be multiple header blocks in the file and different blocks of data with timestamps relative to different start times.

Why would we need this tool? In certain field exercises, occasionally an operator may decide to stop the mission (killing the MOOS community), and restart with perhaps a slight modification in an important parameter. In such cases, two sets of log files will be produced, one from before the restart and one from after. The two of them may constitute a valid mission log file, but they are now split into two. The alogcat utility can be used for merging them back into one.

The alogcat tool performs a proper concatenation. First, it determines the chronological ordering of the provided alog files. It will use the header block and starting time of the earliest file. For the remaning files, (a) the relative time to the first file is calculated, (b) the header block of the later file(s) is removed, (c) the log entries of the later file(s) are appended to the end of the earlier file with timestamps appropriately adjusted along the way.

As an example, consider two alog files created from the same mission, `fil1.alog` and `file2.alog`. They can be joined with:

```
$ alogcatfile1.alog file2.alog --new=final.alog --verbose
Performing a pre-check on the list of alog files...
Processing file: file1.alog
Processing file: file2.alog
Created new alog file: final.alog
```

Overlapping files will produce an error and no new file will be created. If a second log file is created by re-starting the logger after it has been stopped during a mission, it is possible that initial postings will have negative time stamps. This is due to the MOOSDB delivering the mail with the latest values. These postings may have occurred before the logger re-started, and in fact may represent duplicate postings found in the earlier log file. These postings are ignore. Keep in mind that any posting in subsequent log files that have a negative timestamp will be ignored with the files are concatenated.

If the new file to be created already exists, no action will be taken. This can be overridden with the `--force, -f` parameter.

84 The alogavg Tool

The post-mission analysis pipeline starts with raw alog files and ends with plotted data. The raw data in alog files cannot be directly plotted without some filtering and analysis steps. The alogavg

tool is one of the tools for performing these steps. These steps could also be done within the plotting language such as Matlab or Python, but our preference is to have general tools in the ALog Toolbox for these steps to allow users to choose between plotting tools without having to re-write the data preparation steps in each plotting language.

A concise form of this documentation is available on the command line:

```
$ alogavg --help    (or -h)
```

The web version of this content can be quickly summoned from the command line with:

```
$ alogavg --web    (or -w)
```

84.1 Input Format for alogavg

The `alogavg` tool ingests a single file, using the `--file=file.txt` argument. This file will have two columns of data, the domain (x) in the first column, and the range (y) in the second column. A file of this type may have been produced by `aloggrep` directly from the raw alog file. No ordering is presumed. A simple example:

```
2  289.09
1  357.331
2  287.762
1  358.505
2  289.645
1  358.952
```

For now it is assumed that the column separator is white space (ASCII 32). White space at either the beginning or end of each line is ignored, and it does not matter how much white space is between the two columns.

84.2 Output Format for alogavg

The output of `alogavg` will be five columns:

- The domain (x values), ordered
- The average of the range (y values)
- The minimum range value for the given domain value
- The maximum range value for the given domain value
- The standard deviation of range values for the given domain value

A simple example, using the data above, stored in `file.log`:

```
$ alogavg file.log
1 358.262667 357.331 358.952 0.683596
2 288.832333 287.762 289.645 0.790028
```

By default the columns of the output are padded to line up all columns in a more human readable format. With the `--noformat` option, the columns will all be separated by a single white space.

85 The alogmhash Tool

The `alogmhash` command-line tool will examine a given single alog file and discern key pieces of information related to the mission hash. It's primary intended use is to be invoked within a script as step in the automated archiving of log files. But it can also be run on the command line to provide some useful information about the mission hash of an alog file:

```
$ alogmhash LOG_ABE_26_6_2023_____22_00_52.alog
Analyzing odometry in file : LOG_ABE_26_6_2023_____22_00_52.alog

6,697 total alog file lines.
-------------------------------------
MHash:     230626-2200N-MALT-PAIR
Odometry:  0
Duration:  73.2
Node Name: abe
UTC:       1687831252.73
Full List of MHashes noted:
 [230626-2200N-MALT-PAIR]: 0.00 secs
```

The condensed output version of this tool is the primary use case. It will produce a slingle line of ouput that will be saved in an `.mhash` file. This information is critical to the archiving pipeline

```
$ alogmhash --all LOG_ABE_26_6_2023_____22_00_52.alog
mhash=230626-2200N-MALT-PAIR,odo=0.0,duration=73.2,name=abe,utc=1687831252,xhash=ABE-073S-000M
```

85.1 Output Components

The `mhash` utility will scour the log file for the following pieces of information.

- The mission hash (mhash)
- The UTC start time of the mission
- The odometry, e.g., distance travelled
- The duration of the log file
- The node name

The mission hash is obtained by examining the alog file for entries to the variable `MISSION_HASH`. An example is:

`mhash=230626-2200N-MALT-PAIR,utc=1687831252.73`

If there is only one value for this variable in the alog file, then identifying the mission hash and the UTC start time is pretty easy. However, if the alog file is from a vehicle, it's possible that the shoreside mission may have been restarted one or more times in the duration of the vehicle log file. This would result in the vehicle log file having multiple values for the `MISSION_HASH`. For this

reason alogmhash tallies the odometry distance accumulated while each distinct MISSION_HASH is the prevailing value. In the end, the one with the longest odometry distance is the one chosen. The assumption is that, when or if a shoreside mission is re-started, it is usually while the vehicle is sitting in a pre-launch mode, in which case the mission hash values from earlier starts should be disregarded.

The odometry distance calculated by alogmhash is derived simply from the logged NAV_X and NAV_Y values.

The duration value calculated by alogmhash is derived simply from timestamp of the last logged entry in the alog file against the UTC start time recorded at the top of the alog file.

The node name is either the vehicle name or "shoreside". It is derived by examining a publication to the MOOS variable DB_UPTIME and noting the source (MOOS App) name that published it, which is always the MOOSDB with the format MOOSDB_node. The component to the right of the underscore is regarded as the node name. The node name of "shoreside" is always shortened to "shore"

85.2 Command Line Usage for the alogmhash Tool

```
$ alogmhash --help or -h
```

Listing 85.48: Command line usage for the alogmhash tool.

```
Usage:
  alogmhash file.alog [OPTIONS]

Synopsis:
  alogmhash will read the one given alog file and
  parse for key values used in a .mhash cache file.
  Value Examples:
    MHASH:    230513-1453B-ICED-OWEN
    ODO:      423.2
    DURATION: 874.3
    NAME:     abe
    UTC:      33678848885.977
  If multiple MISSION_HASH values are present in
  the given log file, the posting value with the
  highest odemetry distance is the one chosen. This
  guards against the situation where a shoreside
  re-starts after the logging has started and thus
  generates a new mission hash. Rule of thumb is to
  regard the MISSION_HASH that prevailed for the
  highest odometry as the one true MISSION_HASH.

Options:
  -h,--help       Displays this help message
  -v,--version    Display current release version
  -t,--terse      terse output (No newline/CRLF)

  -m,--mhash      Report MHASH w/ longest odometry
  -o,--odo        Report total odometry
  -d,--duration   Report total duration
  -n,--name       Report node/vehicle name
  -u,--utc        Report UTC start time
  -a,--all        Report mhash,odo,dur,name

  --web,-w
```

```
    Open browser to:
    https://oceanai.mit.edu/ivpman/apps/alogmhash

Returns:
  0 if alog file ok.
  1 if alog file not ok, unable to open.

Further Notes:
  (1) The order of arguments is irrelevent.
  (2) Only last given .alog file is reported on.
  (3) Intended to be used in mhash_tag.sh script.

(1) Example:
  $ alogmhash *.alog --all
    mhash=230512-2147I-ICED-OWEN,odo=1507.5,     \
    duration=451.3,name=alpha,utc=33678848885.977
(2) Example:
  $ alogmhash *.alog --mhash
    mhash=230512-2147I-ICED-OWEN
```

86 The alogeval Tool

The `alogeval` tool will scan a given alog file and apply test criteria found in a separate input file. The test criteria should be similar to that used by `pMissionEval`, specifying criteria for *when* the test should be applied, and then the criteria for passing or failing the test. The advantage of `alogeval` over `pMissionEval` is that the user can repeatedly adjust the evaluation criteria and re-run the evaluations. With `pMissionEval`, the test criteria loaded at run-time will produce mission evaluations that are based on the run-time criteria only.

A concise form of this documentation is available on the command line:

```
$ alogeval --help    (or -h)
```

The web version of this content can be quickly summoned from the command line with:

```
$ alogeval --web    (or -w)
```

86.1 Test Criteria Input File

The test criteria is provided in a separate input file. If the file has the suffix `.txt`, it can simply be provided, otherwise use the `--testfile=filename` argument. The following two are equivalent:

```
$ alogeval mission.alog criteria.txt
$ alogeval mission.alog --testfile=criteria.txt
```

86.2 Test Criteria Logic Test Sequence

The primary configuration structure is a *logic test sequence*, comprised of one or more *logic tests*. The sequence of tests may be ordered by time, but may also be time invariant. The overall test

evaluation is completed when either (a) all logic tests in the sequence have been evaluated and passed, or (b) when any one of the logic tests in the sequence has failed.

A logic single test is comprised of:

- lead conditions
- pass conditions
- fail conditions

One or more lead conditions may be provided. As soon as all lead conditions have been satisfied, the pass/fail conditions are evaluated. If one or more pass conditions have been provided, then *all* must satisfied or else the test fails. If one or more fail conditions are provided, then if *any* fail condition is satisfied, the test fails.

A test sequence is comprised one or more tests. Each test has its own conditions. In this case *the order of lines in the configuration file is important* as it determines the grouping. For example, in the Alpha End Flag mission, the sole test is whether the end flag has been set after the fifth waypoint has been hit. This does not test for the possible error case where the endflag is posted after the third or fourth waypoint has been hit. The single logic test could be replace with the following test sequence:

```
lead_condition = WC_FLAG = 3
pass_condition = EFLAG = 0

lead_condition = WC_FLAG = 4
pass_condition = EFLAG = 0
```

Note in this case there are three tests, all tests need to pass if the `pass_flag` is to be posted. Note also that the test `EFLAG=0` will not pass unless it is explicitly published with this value. And the condition `EFLAG!=1` will also not pass unless `EFLAG` has been published. In this example, in the Alpha End Flag mission, the helm initializes the `EFLAG` variable to 0 upon helm launch.

The order of the tasks in the configuration does matter. The app at any given moment will address the next element in the sequence. Only when the last element has been addressed will the result, pass, and fail flags be published. Consider what would happen if the first two of the above tests were reversed? In this case the mission would never finish evaluation. Why?

86.3 Test Output and Return Values

If `alogeval` succeeds, with valid alog file, valid criteria file, and passing all logic tests, it returns with a value of 0. Other possible values:

- 0: Success
- 1: All valid files, but test criteria failed
- 2: Unhandled command line argument
- 3: Missing alog file or could not be opened
- 4: Missing criteria file or could not be opened
- 5: ALog or criteria file could not be parsed

The above return values are returned to the shell on the command line. By default, alogeval produces no output to the terminal. Output can be enabled with the --verbose flag, for example:

```
$ alogeval file.alog criteria.txt --verbose
BEGIN ALogEvaluator: Handling ALog File:
    14.72909      EFLAG  0   [false,false] (0)
   121.82700    WC_FLAG  1   [false,false] (0)
   162.14515    WC_FLAG  2   [false,false] (0)
   192.43813    WC_FLAG  3   [false,false] (1)
   215.20145    WC_FLAG  4   [true,true]   (2)
EVALUATION COMPLETE
END  ALogEvaluator: Handling ALog File
Result: pass=true
$ echo $?
0
```

In the above output, several lines with timestamps are shown. Variables involved in the criteria logic conditions are noted by alogeval. Each time a log entry is encountered with one of these variables, the test criteria is applied, and a line of output is produced. The second column is the MOOS variable name. The third column is the posted value of the MOOS variable. The fourth column contains two Booleans. The first is whether the entire test sequence has been evaluated, and the second Boolean is whether the evaluation passed. If the first is false, the second will false. The fifth column is index of the current logic test. Since there are only two logic tests in this example, with the index is 2, then the evaluations should be complete.

In the line with timestamp at 192 seconds, the WC_FLAG posting is 3, which is the first lead_condition. Since EFLAG=0 is the pass condition, this first logic test passes and the logic test index increments from 0 to 1. In the line with timestamp at 215 seconds, the WC_FLAG posting is 4, which is the second lead_condition. Since EFLAG=0 is the pass condition, this second logic test passes and the logic test index increments from 0 to 2. Since all logic tests have been evaluated, and all passed, the fourth column on this final line reads [true,true], and the evaluation is complete.

86.4 Examining the Test Sequence Structure

The normal verbose output can be augmented with the --show_seq or -ss flag. This will show the structure of test criteria before the evaluation, and again after the evaluations. For example:

```
$ alogeval file.alog criteria.txt --verbose -ss
BEGIN ALogEvaluator LogicTestSequence =========
LogicTestSequence: Total Aspects:2
Enabled:   true
Evaluated: false
Satisfied: false
Status:    pending
++++++++++ aspect[0] ++++++++++++
Lead Conditions: (1)  WC_FLAG = 3
Pass Conditions: (1)  EFLAG = 0
Fail Conditions: (none)
Enabled(true), Evaluated(false), Satisfied(false), Status(pending)
++++++++++ aspect[1] ++++++++++++
Lead Conditions: (1)  WC_FLAG = 4
Pass Conditions: (1)  EFLAG = 0
Fail Conditions: (none)
Enabled(true), Evaluated(false), Satisfied(false), Status(pending)
END   ALogEvaluator LogicTestSequence =========

BEGIN ALogEvaluator: Handling ALog File:
    14.72909      EFLAG  0   [false,false] (0)
   121.82700    WC_FLAG  1   [false,false] (0)
   162.14515    WC_FLAG  2   [false,false] (0)
   192.43813    WC_FLAG  3   [false,false] (1)
   215.20145    WC_FLAG  4   [true,true]   (2)
EVALUATION COMPLETE
END  ALogEvaluator: Handling ALog File

BEGIN ALogEvaluator LogicTestSequence =========
LogicTestSequence: Total Aspects:2
Enabled:   true
Evaluated: true
Satisfied: true
Status:    pass
++++++++++ aspect[0] ++++++++++++
Lead Conditions: (1)  WC_FLAG = 3
Pass Conditions: (1)  EFLAG = 0
Fail Conditions: (none)
Enabled(true), Evaluated(true), Satisfied(true), Status(pass)
++++++++++ aspect[1] ++++++++++++
Lead Conditions: (1)  WC_FLAG = 4
Pass Conditions: (1)  EFLAG = 0
Fail Conditions: (none)
Enabled(true), Evaluated(true), Satisfied(true), Status(pass)
END   ALogEvaluator LogicTestSequence =========
Result: pass=true
```

87 The alogview Tool for Analyzing Mission Log Files

87.1 Overview

The alogview application is a post-mission analysis utility loaded with one or more alog files, typically one per vehicle. It is more than a log file playback tool. A full state is maintained across all vehicles for both playing back sequentially and rewinding or jumping to any arbitrary point in time. The main window may look a bit like the run-time pMarineViewer window, but many of the GUI components in alogview are available solely for post-mission analysis. A snapshot of the tool, with no pop-up sub-windows open, is shown in Figure 221.

Figure 221: **The alogview tool**: used for post-mission rendering of alog files from one or more vehicles and stepping through time to analyze helm status, IvP objective functions, and other logged numerical data correlated to vehicle position in the op-area and time.

Quick Start: Launch alogview with one or more alog files on the command line, for example the two vehicle alog files from the m2_berta example mission:

```
$ alogview henry.alog gilda.alog
```

Upon launch, the viewer should look similar to Figure 221. Step forward or backward through time with the '[' or ']' keys, or click anywhere in the time-series in the bottom part of the screen. Replay can also be automated by accelerating or decelerating replay with the 'a or 'z' characters. Many other replay and visualization options are available via the pull-down menus and are introduced in

later sections.

87.2 Recent History of Development with Releases

- Pre-2011: This tool existed as `logview`, was not documented and had limited support for rendering objective functions and helm information. Mostly used in-house by developers.
- 2011: Better support for IvP function rendering. Initial thin documentation. Support added for geometric log entries. Still very buggy but useful enough to share with other adventurous users. Biggest limitations: (a) initial full-alog load into memory too huge for anything but small missions. (b) no general scoping tool available for non-numerical data, (c) many little bugs or incomplete features.
- 2015: First real attempt to support users generally outside core developers. Better memory management for large and/or multi-vehicle missions. New scope for non-numerical data. Pop-up implementation of sub-windows. Unlimited sub-windows supported for multi-vehicle missions. Overall much less buggy and more feature-rich.

87.3 The Five Primary Viewing Windows

The `alogview` tool can be described by its five GUI components, listed below. The first two are always open upon startup, and the remaining three are upon user request via the pull-down menus:

- The *NavPlot* window: This is the main vehicle viewing area shown at the top in Figure 221. It primarily shows the current vehicle position(s) and perhaps recent trajectory and other geometric artifacts produced by the vehicle apps or behaviors, such as waypoints or sensor artifacts. More detail in Section 87.4.
- The *LogPlot* window: This is the time-series window shown at the bottom in Figure 221. The variable's rendered are selected by the user from the list any logged numerical data from any vehicle. More detail in Section 88.9.
- The *IPFPlot* window: (pronounced "i-p-f-plot") This is a pop-up window launched from the `IPFPlots` pull-down menu, for a chosen vehicle. It will render the prevailing IvP objective functions responsible for latest helm decision. More detail in Section 88.10.
- The *HelmPlot* window: This is a pop-up window launched from the `HelmPlots` pull-down menu, for a chosen vehicle. It shows the helm state in terms of active, running, idle behaviors, as well as behavior warnings, errors and life events. More detail in Section 88.11.
- The *VarPlot* window: This is a pop-up window launched from the `VarPlots` pull-down menu, for a chosen vehicle, and chosen logged MOOS variable (string or double). It shows an ordered list of recent and pending postings for the given variable. Additional variables can be added resulting in an interleaved sequence. More detail in Section 88.12.

In all windows, the current time can be altered with the same keyboard hot keys, found in the `Replay` pull-down menu and described in Section 87.4.

87.4 The NavPlot Window and Controlling Replay

The *NavPlot* window is the main viewing space, shown upon startup, with the current and recent vehicle positions rendered for each vehicle alog file. It is the top part shown in Figure 221.

87.4.1 Controlling the Current Time in Paused Mode

By default alogview is paused upon startup. The options for stepping forward and backward through time may be seen from the Replay pull-down menu. There are three pairs of options:

- The '[' and ']' keys for stepping forward and backward by 1 second.
- The '' and '' keys for stepping forward and backward by 5 seconds.
- The 'ctrl-[' and 'ctrl-]' keys for stepping forward and backward by 0.1 seconds.

87.4.2 Controlling the Current Time in Streaming Mode

Replay may also be put into a streaming mode at any time by accelerating or decelerating the streaming speed. Streaming is controlled by the following three keys:

- The 'a' key will accelerate the time warp, doubling the warp upon each successive hit, up to 64Hz.
- The 'z' key will decelerate the time warp, halving the warp upon each successive hit, down to 1/8Hz.
- The 'SPACEBAR' key will pause or unpause the streaming mode at any time.

The time warp is always shown in the very top of the alogview window. When streaming at say 16Hz, you should see (Time Warp = 16) in the window title bar. When not streaming, i.e., paused, this is indicated instead of the time warp, as in Figure 221.

Note that in streaming, between each update, there is work going on under the hood in addition to the rendering tasks. This includes the update of the current state of what is being rendered in the main and pop-up windows. Often it is not possible to refresh at the time warp requested. In this situation, alogview, and your machine, will do the best it can to keep up. Instead of seeing (Time Warp = 16) in the title bar, you may see something like (Time Warp = 16)(Actual:8.73). The latter number will vary, but represents the best warping rate possible given what is being updated between refreshes.

A faster observed time warp can be achieved by altering the *streaming redraw interval*. By default this interval is 0.5 seconds. So, for example, if the replay time warp is 4, then 8 re-draws will need to be handled per second. This may or may not be achievable. If the redraw interval is instead 1.0 seconds, then half as many re-draws are required for the same effective time warp. The drawback will be that the motion of the vehicle and objects will appear more "jumpy". The *streaming redraw interval* may be altered in the Replay pull-down menu.

88 Background Region Images

In both pMarineViewer and alogview, typical use involves a background region image upon which vehicles and other geometric objects are overlayed. Both utilies can be configured to use images of their own choosing. See examples in Figure 222. The user provides these images from whatever

source they wish. Options for obtaining images and proper format are discussed below in the section *Obtaining Image Files*.

Figure 222: **Example Background Region Images**: The example images downloaded from freely available tile servers that may be utilized in either pMarineViewer or alogview.

88.1 Default Packaged Images

Image files can be large, and of course there are endless possibilities depending on where you are operarating and which kind of background images you prefer. That being said, a couple images are distributed with MOOS-IvP. This allows example missions distributed with the MOOS-IvP code to having working images out-of-the-box without requiring the new user to fetch images. These two images are for (a) the MIT Sailing Pavilion, and (b) Forest Lake in Gray Maine, the site of some of the earliest in-water experiments circa 2004. This files are:

- `MIT_SP.tif`
- `forrest19.tif`

Both are distributed with MOOS-IvP and can be found in `moos-ivp/ivp/data`. When pMarineViewer or alogview is launched, this directory will be examined for the specified image file. Instructions for loading user-provided images is are given below in section *Loading Images at Run Time*.

88.2 Image File Format and Meta Data (Info Files)

Images loaded into pMarineViewer and alogview are in the format of `"Tiff"` files. These files have the suffix `".tif"`. Tiff files have been around since the eighties, use lossless compression, are typically high quality, but are not as common as formats such as jpeg. There are many freely available tools for converting back and forth between tiff and jpeg and other formats. Tiff files are used in pMarineViewer and alogview primarily due to the availability of the `libtiff` library, readily available through package managers on both the MacOS and Linux platforms.

Each `.tif` file used in pMarineViewer and alogview has a corresponding `.info` file, containing (a) the latitude and longitude coordinates of the image edges, and (b) the datum, or `(0,0)` coordinate, on the image. For example, the `MIT_SP.tif` file has a corresonding `MIT_SP.info` file found in the same directory.

```
lat_north = 42.360264
lat_south = 42.35415
lon_east  = -71.080058
lon_west  = -71.094214
datum_lat = 42.358436
datum_lon = -71.087448
```

88.3 Obtaining Image Files

Image files may be obtained and used in pMarineViewer and alogview from any source convenient to the user. This includes opening an image in say Google Maps on a web browser and performing a screen grab. The drawback of this method however is that it may be hard to precisely determine the lat/lon coordinates of the edges used in the corresponding `.info` file.

There are several open *tile servers* which allow a user to download an image tile, or set of tiles, provided with a range of lat/lon coordinates. These tiles can then be stitched together to make a single image. Although this sounds cumbersome, this process can be automated in a script. One such script is Anaxi Map, written by Conlan Cesar:

https://github.com/HeroCC/AnaxiMap

This utility is capable of using one of several tile servers with various background styles, such as Google Maps, maps with terrain or bathymetry information, or maps with street data. See Figure 222.

AnaxiMap, or similar utilities, may produce images in jpeg or png format. MacOS and Linux provide native utilities for converting the format, or "exporting" the file, to tiff format. On MacOS or Linux, if the free ImageMagick package is installed, you can use the "convert" utility:

```
$ convert region.jpg region.tif
```

88.4 Loading Images at Run Time

Post Release 22.8, both pMarineViewer and alogview support operation with multiple background images. Toggling between images at run time is done by selecting the `BackView` pull-down menu and selcting `tiff_type toggle`, or by simply hitting the ‘ (back-tick) key.

In pMarineViewer, the multiple background images may be specified with multiple configuration lines, for example:

```
tiff_file  = MIT_SP.tif
tiff_file  = mit_sp_osm18.tif
```

In alogview, the multiple background images may be specified on the command line:

```
$ alogview --bg=MIT_SP.tif --bg=mit_sp_osm18.tif file.alog
```

88.5 Automatic alogview Detection of Background Image

When launching alogview typically the user wants to use the same background region image used by pMarineViewer during the course of the mission that generated the alog file. In a new feature, post Release 22.8, alogview will automatically attempt to detect the image file used by pMarineViewer. The name name of region image is now published by pMarineViewer during the execution of the mission. This information is contained in the variable REGION_INFO. For example:

```
REGION_INFO = lat_datum=42.358436, lon_datum=-71.087448, img_file=MIT_SP.tif,\
              zoom=2.5, pan_x=129, pan_y=-364
```

The region info contains the name of the image (tiff) file used during the course of the mission, as well as the pan and zoom information as hints for alogview for use upon startup. The images found in REGION_INFO in the alog file will be loaded, as well as any images specified on the command line with the --bg options.

88.6 Background Image Path

Support for the IVP_IMAGE_DIRS shell path is a new feature, post Release 22.8, relevant to both pMarineViewer and alogview. This is explained below.

Image files are named in either the pMarineViewer config block of the .moos mission file, or named on the command line when launching alogview, as specified in Section *Loading Images at Run Time*. For alogview, the file may also be named in the REGION_INFO logged variable as discussed above.

Both apps need to *find* the named .tif file. When found, it looks for the corresponding .info file in the same directory. There are four options for making this work:

- The image file is in the same directory as the mission file.
- The image file is in the special directory moos-ivp/ivp/data.
- The full or relative path name of the file is specified.
- The file exists in a directory on your IVP_IMAGE_DIRS path.

The first option has the drawback of duplicating the image file in potentially many places. The second option has the drawback that the directory moos-ivp/ivp/data is part of the MOOS-IvP code distribution which users otherwise consider to be read-only. A fresh check out of MOOS-IvP would reset this directory and users would need to take care to migrate files to a new checkout. The third option is that full or relative path name may not be the same between different users or machines. The fourth option is the newest option and arguably has the least downside.

The IVP_IMAGE_DIRS is shell (e.g., bash) environment variable. It contains a list of one or more directories on your local computer where pMarineViewer or alogview will look when attempting to load image files. Shell environment variables are already common settings that users will customize on their particular machine.

The recommended way for users to use a set of custom image files is to (a) organize them in one or more directories, preferably under version control, (b) install them at a convenient location on your

local machine, (c) configure the IVP_IMAGE_DIRS shell variable to contain the one or more directories where your image files reside.

For example, if you have a folder of image files with the following structure:

```
my_images/
   napa_bay/
      napa_bay_gmaps.tif
      napa_bay_gmaps.info
      napa_bay_osm.tif
      napa_bay_osm.info
   happy_harbor/
      happy_harbor_gmaps.tif
      happy_harbor_gmaps.info
      happy_harbor_osm.tif
      happy_harbor_osm.info
```

If this folder is installed on your machine in the home dirctory folder call "project", then you would set your IVP_IMAGE_DIRS path in your shell configuration file, e.g., .bashrc, as follows:

```
IVP_IMAGE_DIRS=~/project/napa_bay
IVP_IMAGE_DIRS+=:~/project/happy_harbor
```

To verify which file has been loaded, the appcasing output of pMarineViewer shows the full path name of the loaded file(s). And when alogview is launched, the terminal output indicates which directories are being searched, in order, for the image files. This information may be obscured however when the alogview window pops up, but you can find it if you go back to it and perhaps scroll up. Note: It is not sufficient, in the example above, to simply set IVP_IMG_DIR= /project, the parent directory of all image folders. Each image folder must be named.

88.7 Troubleshooting

88.7.1 pMarineViewer fails to load the image (see only gray screen)

1. Check the appcasting output of pMarineViewer. The top few lines should show which image file is loaded. Is this a file you recognize?
2. Does this file exist on your computer? Verify it is where you think it is.
3. How are you specifying this file in your pMarineViewer config block? If it is specified as a relative path, e.g., ../my_images/napa_bay.tif, make sure that relative path location is correct.
4. If you are specifying the image file with just the file name (no path information), then check you have your IVP_IMAGE_DIRS variable set properly. Run echo $IVP_IMAGE_DIRS on the command line.
5. Simplest but most common: Make sure your image file name (configuration and actual name) end in the suffix .tif and not .tiff.

88.7.2 pMarineViewer or alogview image is fine but no vehicles

1. Check the `.info` file. Make sure the lat/lon values sanity check, e.g., rough magnitude, relative values.
2. Make sure the `datum_lat` and `datum_lon` values are in the range of the image.
3. Make sure the `datum_lat` and `datum_lon` values match the datum set at the top of the vehicle and shoreside mission files.

88.7.3 alogview fails to load the image (see only gray screen)

In the newer version of `alogview`, when launching it will attempt to read the name of the image file used by `pMarineViewer`. So if `pMarineViewer` launched ok, chanches are good `alogview` will also launch with the same image. However, it could be the case that (a) the mission was run on some other computer that contained the image file and your current computer does not. The image file is not logged. (b) the mission was named

1. Does this file exist on your computer? Verify it is where you think it is. It is possible that you are trying to run `alogview` on your computer with alog files generated on someone else's computer. Make sure you have the image file.
2. Check the terminal output of `alogview` as it is loading. To demonstrate the below output from an `alogview` launch purposely use the the file `dforrest19.tif` instead of `forrest19.tif`. Note the sequences of folders searched in the attempt to find the image file. The first attempt is the in the `moos-ivp/ivp/data` directory. The next attempts are based on the value of `IVP_IMAGE_DIRS`. The final attempt is in the current working directory where `alogview` was launched. Does this match up with your expectations?

```
TIFF FILES COUNT:1
[1] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/data]
    Not found.
[2] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/pavlab_map_images/popolopen]
[3] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/pavlab_map_images/mit]
[4] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/datax]
[5] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/data-local]
[6] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [/Users/james/moos-ivp/ivp/data]
[*] Looking for dforrest19.tif and dforrest19.info in:
    Dir: [./]
    Not found.
Could not find the pair of files:
dforrest19.tif and dforrest19.info
Opening Tiff:
TIFFOpen: : No such file or directory.
  Failed!!!!!!!!!
```

3. Simplest but most common: Make sure your image file name (configuration and actual name) end in the suffix `.tif` and not `.tiff`.

88.8 Video Capture

Video capture is no longer natively supported in `alogview` due to the common availability of third party video capturing software. Just launch `alogview`, put it into streaming mode (Section 87.4.2) at the desired rate, and launch your favorite video capture program.

88.9 The LogPlot Window For Viewing Time-Series Data

The *LogPlot* window allows the rendering of any numerical data from any loaded alog file to be plotted against time, as shown in Figure 221 and up close in Figure 223 below. The user may zoom in to see finer resolution in the postings, by clicking either the `IN` or `OUT` buttons, or by the `'i'` or `'o'` keys while the mouse is in the *LogPlot* window. To return to the original full apperature zoom, hit the `RESET` button. The user may left-click at any point in the time series to adjust the current time (shown by the red bar in the viewer). This time adjustment is propagated not only to the *NavPlot* viewer but to all open pop-up windows.

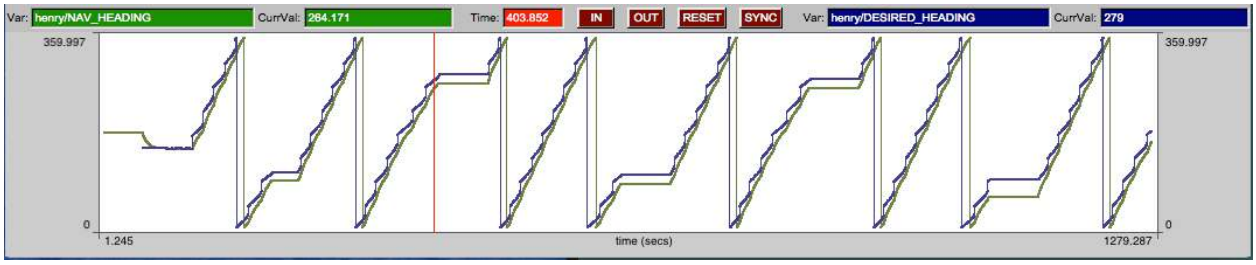

Figure 223: **The `logplot` window**: is used for viewing any numerical data found in any of the alog files loaded into `alogview`. Here the desired and actual vehicle headings are logged side-by-side.

Two variables, and two y-axes are available for selection. By default they are drawn on independent scales to allow variables with starkly different ranges, such as speed and heading, to be meaningfully rendered. The user may choose to synchronize the two variable ranges rendered to allow for more meaningful comparisons when the two variables have the same range, such as desired and observed speed. The synchronization is done by hitting the `SYNC` button at the top of the logplot window. The user chooses which two variables are rendered from the pull-down menu as shown in Figure 224.

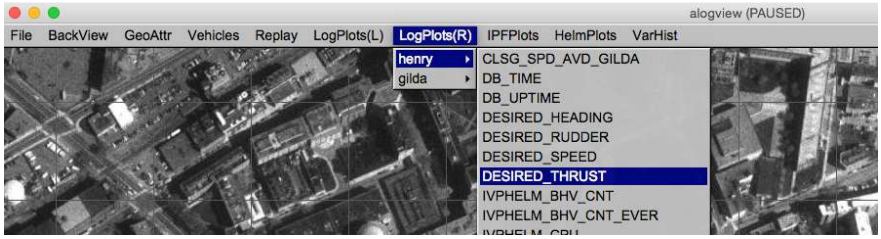

Figure 224: **The `logplot` pull-down menu**: is used for first selecting the vehicle and then the variable to be plotted.

If you find yourself wanting to repeatedly open an alog file with same one or two variables chosen for the *LogPlot* window, these variables can be chosen and at launch time passed as command line arguments:

```
$ alogview --lp=henry:NAV_SPEED --rp=gilda:NAV_SPEED
```

88.9.1 Using the Detached Variable Option

Certain string log entries may be in the format of multi-part comma-separated pairs (CSP). For example:

```
REPORT =  x=77, y=99, speed=3.2, depth=4.5, heading=93, battery_level=16.4
```

Normally, during the split/cache stage at launch time, would create a single `.klog` file, `REPORT.klog`. However, one or more fields may be desired for plotting in alogview With the optional `--detached=VARNAME` command line argument, the variable `VARNAME` will be parsed and separate `.klog` files created for sub-components of the contents of the string. For example, the following argument:

```
$ alogview --detached=REPORT file.alog
```

would result in the creation of `.klog` files for each of the sub-components:

```
REPORT:x.klog
REPORT:y.klog
REPORT:speed.klog
REPORT:depth.klog
REPORT:heading.klog
REPORT:batter_level.klog
```

If the run-time speed of launching `alogview` is a concern, especially on very large files, the above action can be quickened by isolating just the sub-component of interest:

```
$ alogview --detached=REPORT:speed file.alog
```

would create just the following `.klog` file:

```
REPORT:speed.klog
```

If two or more sub-components are desired, the following two equivalent invocations could be used:

```
$ alogview --detached=REPORT:speed --detached=REPORT:depth file.alog
$ alogview --detached=REPORT:speed:depth file.alog
```

would create just the following two `.klog` files:

```
REPORT:speed.klog
REPORT:depth.klog
```

If a detached variable is named, e.g., a file like `REPORT:depth.klog` is created, then `alogview` will now have this as a pull-down menu option for plotting.

Note, if the string is in JSON format, e.g.

```
REPORT =  {"x":77, "y":99, "speed":3.2, "depth":4.5, "heading":93}
```

then `alogview` will internally convert this into CSP format, and will handle accordingly.

88.9.2 Additional Buried but Useful Hot-Keys in the LogPlot Window

There are a few hot-key functions available (only when the main window is active and the mouse is over the *LogPlot* window, that may not be apparent from simply interacting with the window:

- The `'i'` key: will zoom in the resolution on the plot. This can also be done by hitting the `IN` button.
- The `'o'` key: will zoom out the resolution on the plot. This can also be done by hitting the `OUT` button.
- The `'LEFT'` and `'RIGHT'` keys: will step forward and backward through time by 1 second.
- The `'UP'` and `'DOWN'` keys: will zoom in and out on the plot, essentially the same as the `'i'` and `'o'` keys.

The above hot-keys are in addition to the time control hot-keys listed in Section `sec_alv_time_replay` and Section 87.4.1.

88.10 The IPFPlot Window For Viewing Helm Objective Functions

The *IPFPlot* window allows the rendering of an IvP function (IPF) from any vehicle and any behavior found in any of the alog files loaded into `alogview`.

88.10.1 Launching an IPFPlot Window

An *IPFPlot* pop-up window is launched from the `alogview` main pull-down window as shown in Figure 225, by choosing a vehicle name. Separate windows can be opened for different vehicles, or even for the same vehicle.

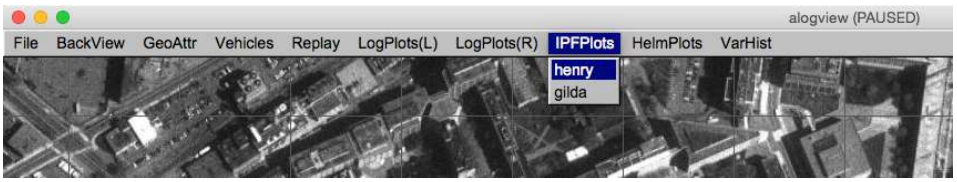

Figure 225: **The `ipfplot` pull-down menu**: is used for first selecting the vehicle and then the variable to be plotted.

A newly launched pop-up window loads the relevant information from disk into the `alogview` memory space. This includes all IvP functions for all behaviors at all times, for the given vehicle. This can be large. For some (larger) missions and some computer configurations, the larger memory footprint will begin to be noticed in terms of application responsiveness. Likewise, the internal data structure updates, each time the current time is changed, increases for each pop-up window. This may result in a slower streaming rate when in replay. Closing the pop-up window releases the memory and should immediately boost responsiveness and playback rate.

88.10.2 Stepping Through Time and Replay Control in the IPFPlot Window

The current time is shown in the upper left corner and is synchronized with the main viewing window and any other pop-up windows that may be open. The same time keys are functional when this window is active, e.g., `'['`, `']'`, `''`, `''`, for stepping forward and backward in time. The replay control keys, e.g., `'a'`, `'z'`, are also functional, and the replay state and rate is also shown in the title bar of this window.

88.10.3 Selecting the IvP Function in the IPFPlot Window

The user may choose any single objective function at a time from the menu on the left side of the window shown in Figure 226. The current behavior weight is shown alongside the behavior name in the menu. If a behavior presently has a weight of zero, no function will be rendered when that behavior is selected. Multiple objective functions may be viewed simultaneously by launching more pop-up windows.

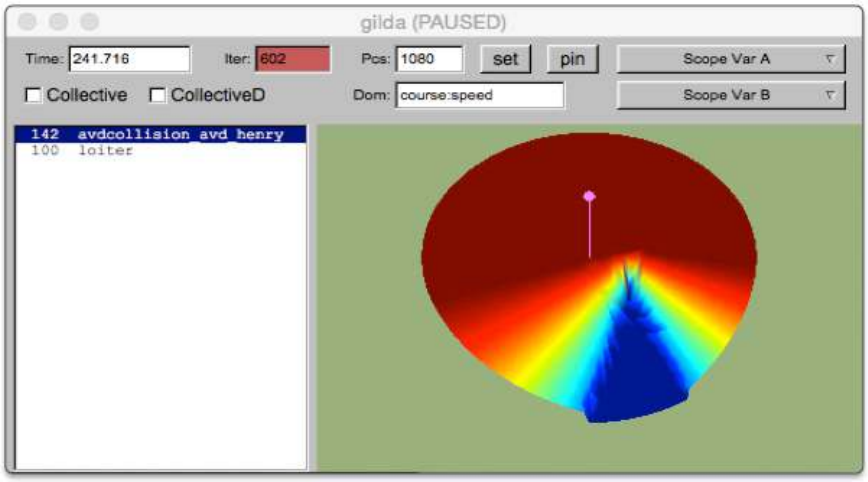

Figure 226: **The `ipfplot` window**: used for examining IvP functions for the chosen vehicle at the current time.

Since each window is associated with a particular vehicle, the helm iteration may be shown unambiguously in the redish window in the upper left. The rendering of the IvP function may be rotated or tilted for different perspectives by using the arrow keys. The perspective may be returned to the bird's eye view by hitting the `SET` button.

88.10.4 Rendering the Collective IvP Function in the IPFPlot Window

IvP functions of two forms are renderable - those defined over course and speed, and those defined over depth. In both cases, a rendering of the collective function is possible by clicking on the `Collective` or `CollectiveD` check-boxes on the left. In Figure 227 below, the collective objective function is shown, showing the weighted sum of the two behaviors The purplish pin rendered in Figures 226 and 227 shows the actual chosen decision for the present helm iteration. When rendering the collective function as in Figure 227 the pin should be visually consistent with the peak of the function. When rendering a single objective function, as in Figure 226, the pin may not visually correlate to the peak of the function since other functions may be in play. For example in Figure 226 the pin identifies one of many points on the red plateau. The pin can be toggled on and off by hitting the `PIN` button.

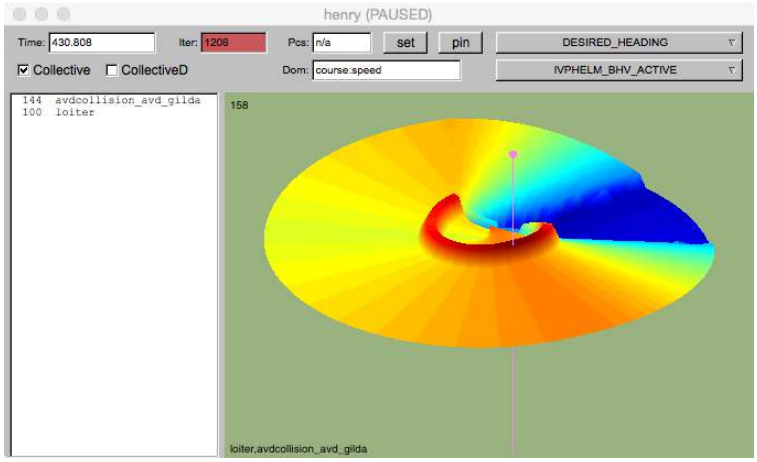

Figure 227: **The `ipfplot` window**: showing here the collective objective function representing the weighted sum of the two active behaviors.

88.10.5 Variable Scoping Within the IPFPlot Window

Often it is useful to scope on a couple key variables related to the rendered objective function, to observe how they change and perhaps influence the function as time evolves. Although a general scoping ability exists via the *VarPlot* window described below in Section 88.12, it can be handy to see the variable value(s) right in the *IPFPlot* window. The two pull-down menu buttons in the upper right in Figure 227 allow the selection of two variables whose values are output in the upper and lower left hand corners of the window as shown. The choice of these two variables is specific to the chosen behavior, or collective function. A different set of variables my be associated with each behavior, and they will be refreshed accordingly as the user switches between behaviors.

88.10.6 Additional Buried but Useful Hot-Keys in the IPFPlot Window

There are a few hot-key functions available that may not be apparent from simply interacting with the window:

- The '+' key: will increase the font size in the browser pane, and the font size of any scoped variable.
- The '-' key: will decrease the font size in the browser panes, and the font size of any scoped variable.
- The 'c' key: will toggle the IvP function rendering from an individual selected function to the collective objective function.
- The 'f' key: will put the window in a "full-window" mode showing only the rendered objective function, without the browser pane.
- The 'UP' and 'DOWN' keys: will tilt the rendered objective functions.
- The 'LEFT' and 'RIGHT' keys: will rotate the rendered objective functions.

The above hot-keys are in addition to the time control hot-keys listed in Section sec_alv_time_replay and Section 87.4.1.

88.11 The HelmPlot Window For Viewing Helm State

The *HelmPlot* window renders information regarding the present helm state, including which behaviors are in which state and for how long, summary of update attempts for each behavior, behavior warnings, behavior errors, a history of helm mode changes, and a history of helm life events.

88.11.1 Launching the HelmPlot Window

The *HelmPlot* pop-up window is launched from the alogview main pull-down window as shown in Figure 228, by choosing a vehicle name. Separate windows can be opened for different vehicles, or even for the same vehicle.

Figure 228: **The helmplot pull-down menu**: is used for first selecting the vehicle and launching the *HelmPlot* window.

A newly launched pop-up window loads the relevant information from disk into the alogview memory space. For some (larger) missions and some computer configurations, the larger memory footprint will begin to be noticed in terms of application responsiveness. Likewise, the internal data structure updates, each time the current time is changed, increases for each pop-up window. This may result in a slower streaming rate when in replay.

88.11.2 Stepping Through Time and Replay Control in the HelmPlot Window

The current time is shown in the upper left corner and is synchronized with the main viewing window and any other pop-up windows that may be open. The same time keys are functional when

this window is active, e.g., '[', ']', '', '', for stepping forward and backward in time. The replay control keys, e.g., 'a', 'z', are also functional, and the replay state and rate is also shown in the title bar of this window.

88.11.3 Behavior States in the Helm Plot Window

Each behavior known to the helm up to the present time is grouped into a list of either active, running, idle or completed modes as shown in Figure 229.

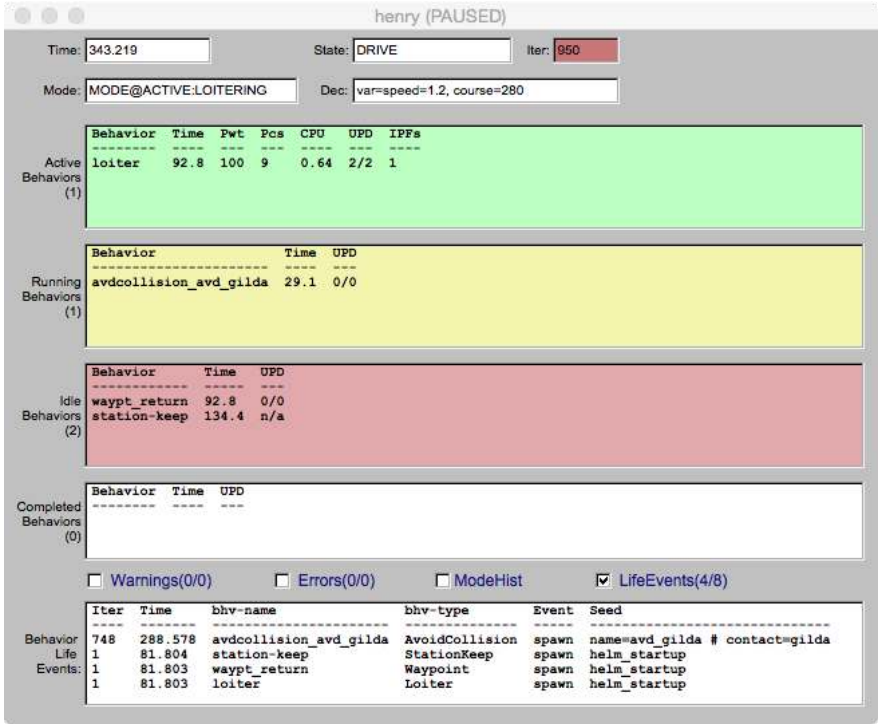

Figure 229: **The `helmplot` window**: used for analyzing the helm state and behavior modes.

For each behavior, the behavior name is listed, along with the elapsed time since the behavior has been in that mode. If the behavior is configured with the `updates` parameter, then the number of successful vs attempted updates are also listed. If the behavior does not have the `updates` parameter set, then `n/a` will instead be listed in that column as with the station-keep behavior in the figure below. For active, behaviors, the present priority weight and number of pieces is also shown.

88.11.4 Behavior Warnings, Errors, Mode History and Life Events

The bottom pane of the *HelmPlot* window may have one of four content modes, selectable with the check-box shown.

- *Behavior Warnings*: In this mode, all behavior warnings, in the form of messages written to `BHV_WARNING`, will be listed in chronological order with the most recent at the top. One handy

feature is that the existence of possible future warnings can be see in the check-box label. In parentheses, (a/b) indicates that a warnings are shown in the list out of b warnings total for the mission.

- *Behavior Errors*: In this mode, all behavior errors, in the form of messages written to BHV_ERROR, will be listed in chronological order with the most recent at the top. In the check-box label, in parentheses, (a/b) indicates that a errors are shown in the list out of b errors total for the mission.
- *Mode History*: In this viewing mode, all helm mode changes will be listed in chronological order with the most recent at the top.
- *Life Events*: In this viewing mode, all helm life events (behavior spawning and deaths) will be listed in chronological order with the most recent at the top.

88.11.5 Additional Buried but Useful Hot-Keys in the HelmPlot Window

There are a few hot-key functions available that may not be apparent from simply interacting with the window:

- The '+' key: will increase the font size in the browser pane, and the font size of any scoped variable.
- The '-' key: will decrease the font size in the browser panes, and the font size of any scoped variable.
- The 'UP' and 'DOWN' keys: will tilt the rendered objective functions.
- The 'LEFT' and 'RIGHT' keys: will rotate the rendered objective functions.

The above hot-keys are in addition to the time control hot-keys listed in Section sec_alv_time_replay and Section 87.4.1.

88.12 The VarPlot Window For Viewing Variable Histories

The *VarPlot* window renders the full history of one or more chosen MOOS variables, allowing the user to see the present value, recent postings, and upcoming pending postings. This holds for both numerical and string variables.

88.12.1 Launching the VarPlot Window

The *VarPlot* pop-up window is launched from the alogview main pull-down window as shown in Figure 230, by choosing a vehicle name, and MOOS variable name. Separate windows can be opened for different vehicles, variables, or even for the same vehicle and variable. Although one variable is chosen upon launch, any number of variables can be inserted later from the VarPlot pull-down buttons

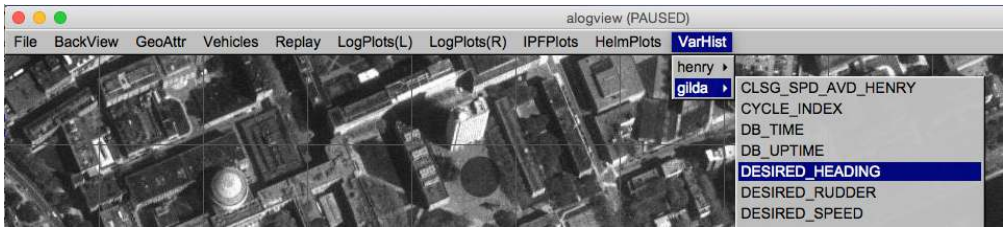

Figure 230: **The `VarPlot` pull-down menu**: is used for first selecting a vehicle and variable name, and launching the *VarPlot* window.

A newly launched pop-up window loads the variable history information from disk into the `alogview` memory space. For some (larger) missions and some computer configurations, the larger memory footprint will begin to be noticed in terms of application responsiveness. Likewise, the internal data structure updates, conducted each time the current time is changed, increases for each pop-up window. This may result in a slower streaming rate when in replay. Closing the pop-up window releases the memory and should immediately boost responsiveness and playback rate.

88.12.2 Viewing the Variable History in the VarPlot Window

The *VarPlot* window contains two primary panes. The top pane shows everything in the past, up to the most recent post. The bottom pane shows everything in the future, starting with the next pending post. The ”current time” can be regarded as being in between the two panes.

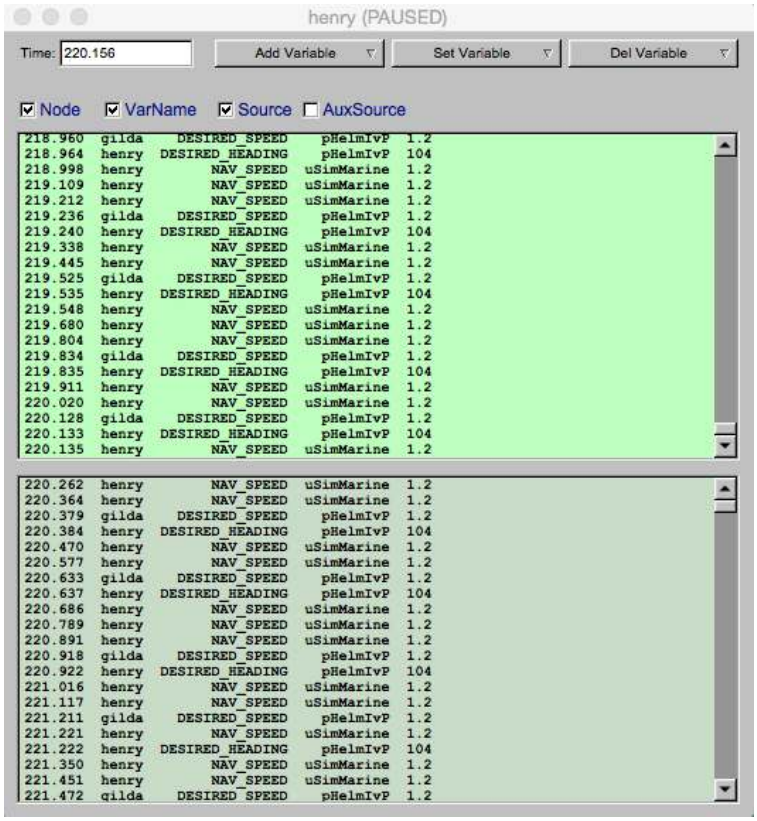

Figure 231: **The `VarPlot` window**: used for scoping on a set of MOOS variables into the past and upcoming in the future.

Each variable posting has six components, separated into six columns in both panes:

- *Time*: The timestamp of the posting. This is always shown in the first column.
- *Node*: The node, i.e., vehicle name is shown in the second column. Since the scope may contain values across nodes, this can be useful to see. If only scoping on one variable, or if all variables are from the same node, then this column can be hidden by toggling the `Node` check-box.
- *Variable Name*: If scoping on only one variable, this third column may be hidden to provide more visibility to the other columns.
- *Source*: The source, i.e., MOOS App that generated the posting, is shown in the fourth column. This may be toggled on or off.
- *Auxiliary Source*: The auxiliary source is shown in the fifth column. Most MOOS apps do not make postings with an auxiliary source. The helm does however, and this typically indicates the particular behavior that produced the posting.
- *Variable Value*: The last column is just the value of the posting. This column cannot be hidden.

88.12.3 Adding and Removing Variables from the History List

Upon launching the *VarPlot* window, a single variable is chosen. After launching, additional variables may be added (from any vehicle) from the `Add Variable` pull-down menu button as shown in Figure 232. The `Set Variable` button presents the same set of choices, but will essentially remove all previously selected variables and replace it with the chosen one. The `Del Variable` button will remove the selected variable from the current set of variables.

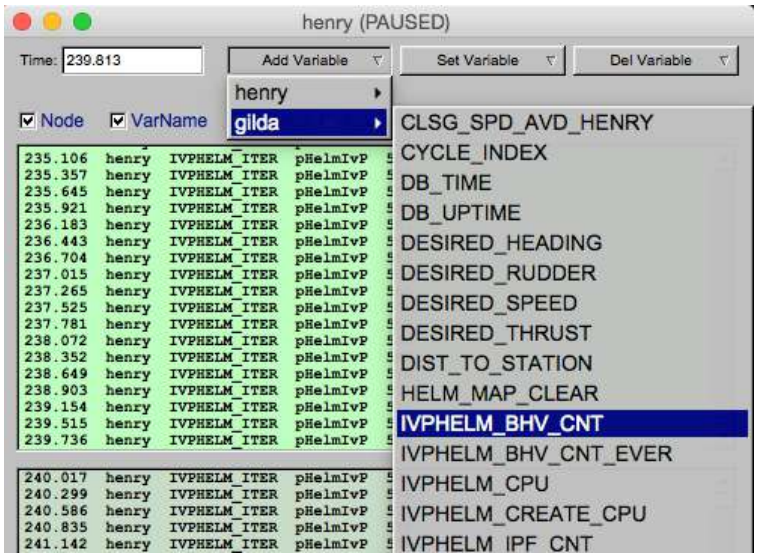

Figure 232: **The `VarPlot` window**: Any variable from any vehicle may be added any time after the window is opened.

88.12.4 Additional Buried but Useful Hot-Keys in the VarPlot Window

There are a few hot-key functions available that may not be apparent from simply interacting with the window:

- The '+' key: will increase the font size in the browser panes.
- The '-' key: will decrease the font size in the browser panes.

The above hot-keys are in addition to the time control hot-keys listed in Section `sec_alv_time_replay` and Section 87.4.1.

88.13 Command Line Usage for the alogview Tool

The `alogview` tool is run from the command line with one or more given `.alog` files and a number of options. The usage options are listed when the tool is launched with the `-h` switch:

```
$ alogview --help or -h
```

Listing 88.49: Command line usage for the `alogview` tool.

```
1  Usage:
```

```
  alogview file.alog [another_file.alog] [OPTIONS]

Synopsis:
  Renders vehicle paths from multiple MOOS .alog files. Replay
  logged data or step through manually. Supports several
  specialized pop-up windows for viewing helm state, objective
  functions, any logged variable across vehicles. If multiple
  alog files are given, they will synchronized. Upon launch,
  the original alog files are split into dedicated directories
  to cache data base on the MOOS variable name.

Standard Arguments:
  file.alog - The input logfile.

Options:
  -h,--help       Displays this help message
  -v,--version    Displays the current release version
  -vb, --verbose  Verbose output

  --bg=file.tiff  Specify an alternate background image.

  --lp=VEH:VAR    Specify starting left log plot.
  --rp=VEH:VAR    Specify starting right log plot.
                  Example: --lp=henry:NAV_X
                  Example: --rp=NAV_SPEED

  --bv=BHV:VAR    Assoc scope var w/ behavior for IPF viewer.

--nowtime=val   Set the initial startup time
--mintime=val   Clip all times/vals below this time
--maxtime=val   Clip all times/vals above this time

--quick,-q      Quick start (no geo shapes, logplots)
--altnav=PREF   Alt nav solution prefix, e.g., NAV_GT_

--zoom=val      Set initial zoom value (default: 1)
--panx=val      Set initial panx value (default: 0)
--pany=val      Set initial pany value (default: 0)

--grep=str1     Set grep pattern 1 in AppLog viewer
--grep=str2     Set grep pattern 2 in AppLog viewer

--alc=FILE      Read config params from file named FILE.
                FILE may contain any param except this one.

--geometry=xsmall  Open GUI with dimensions 770x605
--geometry=small   Open GUI with dimensions 980x770
--geometry=medium  Open GUI with dimensions 1190x935
--geometry=large   Open GUI with dimensions 1400x1100
--geometry=WxH     Open GUI with dimensions WxH

--detached=var          Detached var for plotting
--detached=var:key      Detached var for plotting

--seglist_viewable_all=true/false
--seglist_viewable_labels=true/false
--seglr_viewable_all=true/false
--seglr_viewable_labels=true/false
--point_viewable_all=true/false
--point_viewable_labels=true/false
--vector_viewable_all=true/false
--vector_viewable_labels=true/false
--circle_viewable_all=true/false
--circle_viewable_labels=true/false
```

```
--grid_viewable_all=true/false
--grid_viewable_labels=true/false
--range_pulse_viewable_all=true/false
--trails_color=COLOR
--hash_vieweble=true/false
--tiff_viewable=true/false
--hash_delta={10,50,100,200,500,1000}
--vehicles_shape_scale=(currently unsupported)
--vehicles_viewable=true/false
--trails_point_size=(currently unsupported)
--trails_viewable=true/false
--trails_color=COLOR
--marker_viewable_all=true/false
--marker_viewable_labels=true/false
--polygon_viewable_all=true/false
--polygon_viewable_labels=true/false

--web,-w   Open browser to:
           https://oceanai.mit.edu/ivpman/apps/alogview

Further Notes:
  (1) Multiple .alog files ok - typically one per vehicle
  (2) See also: alogscan, alogrm, alogclip, aloggrep
      alogsort, alogiter, aloghelm
  (3) Config params may also be read from .alogview hidden
      file if present in the directory where launched.
```

The order of the arguments passed to `alogview` do not matter. The `--mintime` and `--maxtime` arguments allow the user to effectively clip the alog files to reduce the amount of data loaded into RAM by `alogview` during a session. The `--geometry` argument allows the user to custom set the size of the display window. A few shortcuts, `"large"`, `"medium"`, `"small"`, and `"xsmall"` are allowed.

89 Introduction to AppCasting

89.1 Overview

AppCasting provides an *optional* and powerful new way of delivering application status output beyond the traditional means of writing to standard I/O in a terminal window. It is motivated by a few common recurring observations:

- The biggest headache of users new to MOOS (e.g., students in MIT 2.680) was the derailment of a mission due to an unnoticed configuration or runtime error.
- Debugging typically involves re-launching with app terminal windows open and analyzing expected vs. observed output.
- When deploying multiple simulated vehicles, each with multiple MOOS apps, the number of open terminal windows may be unmanageable.
- When deploying a vehicle in the field, one cannot ssh in and see any application terminal output at all.
- Since terminal output is rarely viewable for the above practical reasons, apps are rarely designed with much thought put into their status output.

AppCasting is designed to make it easier to see application terminal output. This includes app-specific status messages, configuration and runtime warnings, and notable events. It is designed to allow appcast viewing tools to render this information and alerts on a single screen across multiple vehicles, each with several running apps, whether in simulation or the field. Having this form of information easier at hand, application developers will find it more rewarding to add thoughtful status reports to their application. Most applications in the MOOS-IvP tree have been converted to support appcasting. It's worth repeating that this is an opt-in feature of MOOS. All existing MOOS apps work just fine without appcasting. Appcasting apps and non-appcasting apps work side-by-side seamlessly.

89.2 MOOS Applications and Terminal Output

A MOOS application typically interacts with the world primarily through its subscriptions and publications to the MOOSDB. It may also interact through a terminal interface by generating status or debugging output, or in some rare cases, accepting terminal input.

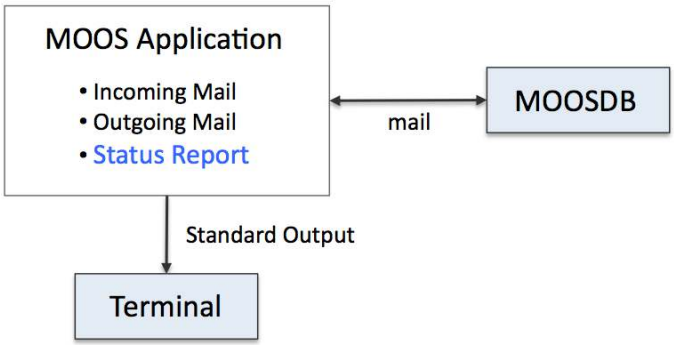

Figure 233: A typical MOOS application interacts with the world by publishing and receiving mail through the MOOSDB and by writing status messages to a terminal window, if one is open.

Even MOOS GUI applications have the option of opening a terminal interface in addition to the GUI. The terminal interface option is invoked by setting `NewConsole=true` for the given app in the `ANTLER` process configuration block in the mission file, or simply by just launching the app manually from the command line. With AppCasting, an application still generates terminal output, but does so by creating a report, in the form of an appcast data structure. The data structure may be converted to a list of strings, sent to the terminal, or a single long string, published to the MOOSDB.

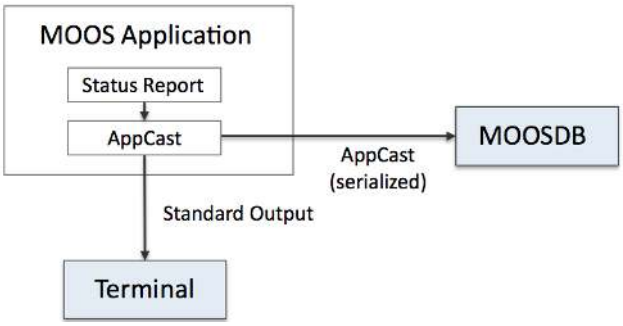

Figure 234: An appcasting MOOS app produces terminal by repeatedly generating and sending an *appcast* to the terminal. The appcast is a also serialized and published to the MOOSDB for other MOOS applications to consume.

By sending the appcast to the MOOSDB this makes a few things possible. First, even if the application was launched long ago without a terminal, it is now possible to launch a separate MOOS application (an appcast viewer tool discussed in Section 89.5) to start looking at the status output. Second, the same appcast data structure may now also be bridged to a separate off-board MOOS community, using something like `pShare`, so remote users may be alerted to or debug problems. Third, the appcast data structure may contain configuration and run-time *alerts* to bring issues to the attention of operators quickly. Fourth, the appcast structure may be logged like any other MOOS variable, making it possible to review terminal output during the post-mission analysis phase.

89.3 Viewing AppCasts and Navigating AppCast Collections

A primary motivation for appcasting is the ability to view appcasts over several applications with a single viewer:

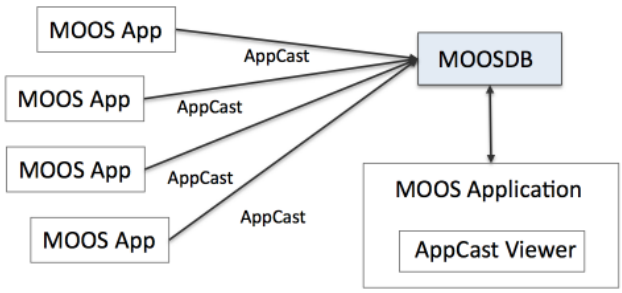

Figure 235: An AppCast Viewer is a separate MOOS application for gathering and navigating through appcasts from several other MOOS applications.

In addition to being able to see application output from a single window, it is possible to view application output for a particular app regardless of how that application was launched. In comparison, if the status output were only viewable from a terminal window, that app would have had to be launched with a terminal window from the outset. Furthermore, when the AppCast Viewer has a terminal interface, a user may log onto a remotely deployed vehicle and launch the viewer and see application output that would not be viewable otherwise since applications on fielded platforms are never run with terminal windows open.

An AppCast Viewer may also handle appcasts from several vehicles as conveyed in Figure 236. The viewer lets the user navigate between different vehicles and appcasts within a vehicle. The interface is discussed in a later section, but the idea is shown on the right in Figure 238.

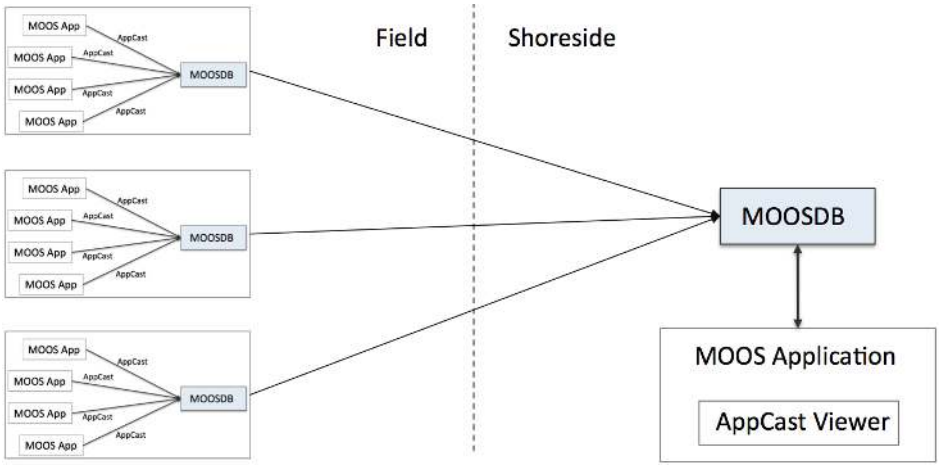

Figure 236: An AppCast Viewer may also be used for sorting and navigating through appcasts from several applications *over several vehicles* or nodes. The appcasts may be bridged from one community to a single shoreside community using a tool such as `pShare`.

The above arrangement assumes that appcasts are sent from each MOOS community to a single *shoreside* MOOS community, to which the appcast viewer is connected. This may be done with the `pShare` utility. The uField Toolbox also provides a set of MOOS utilities to facilitate this kind of arrangement.

89.4 The AppCast Data Structure

An example appcast is shown in Figure 237. It has four distinct parts:

- **config warnings:** Configuration warnings are typically generated during the application's OnStartUp() routine.
- **run warnings:** Run warnings may be generated any time during the execution of the application.
- **general messages:** A list of app developer-formatted strings having whatever the application developer thought would best constitute a succinct meaningful status report.
- **events:** A set of time-stamped events. Exactly what constitutes an event is determined by the application developer.

Figure 237: An appcast consists of four main parts: (1) configuration warnings, (2) run-time warnings, (3) general status report messages, and (4) run-time events. The bar at the top of the figure is rendered in red due to the presence of a run-time warning. The `"1/1"` on this line indicates one configuration warning and one run-time warning. The `"(150)"` on this line indicates that it is iteration #150 for this application. This image is a screen shot take from the `uMACView` utility described later.

For any given appcast, all fields are optional. Indeed, often an appcast will be devoid of any configuration or run warnings. It is also not uncommon for an application not to have a notion of an event.

For reasons explained later, an appcast instance is typically created once upon application startup. The block of general messages is cleared and overwritten each time an appcast is generated. Run warnings and events are added any time during the application operation, but are limited in amount (first-in-first-out/FIFO). This is done to ensure against unbounded growth of the appcast message, and relieve the app developer from addressing the logic of bounded message growth. Configuration warnings are unbounded however since they are only generated at startup time and are typically bounded from above by the number of application configuration lines.

89.5 A Preview of AppCast Viewing Utilities

An appcast viewer is primarily a utility for viewing appcasts, rendering an appcast to look something like that shown in Figure 237. It also does a couple other important things. First, it provides a mechanism to allow the user to navigate between incoming appcasts from multiple vehicles, each with multiple applications. Second, it implements, under the hood, a protocol between the appcast viewer utility and the applications, to ensure on-demand appcasting.

The uMAC utility shown on the left in Figure 238 is run from a terminal window. The uMACView utility shown on the right is a GUI with a bit more capable interface, allowing the user to see all vehicles, all apps for a chosen vehicle, and the appcast for a single chosen application. The advantage of the uMAC utility however is that it may be launched remotely after logging in to a vehicle that may otherwise be unresponsive and in need of some debugging.

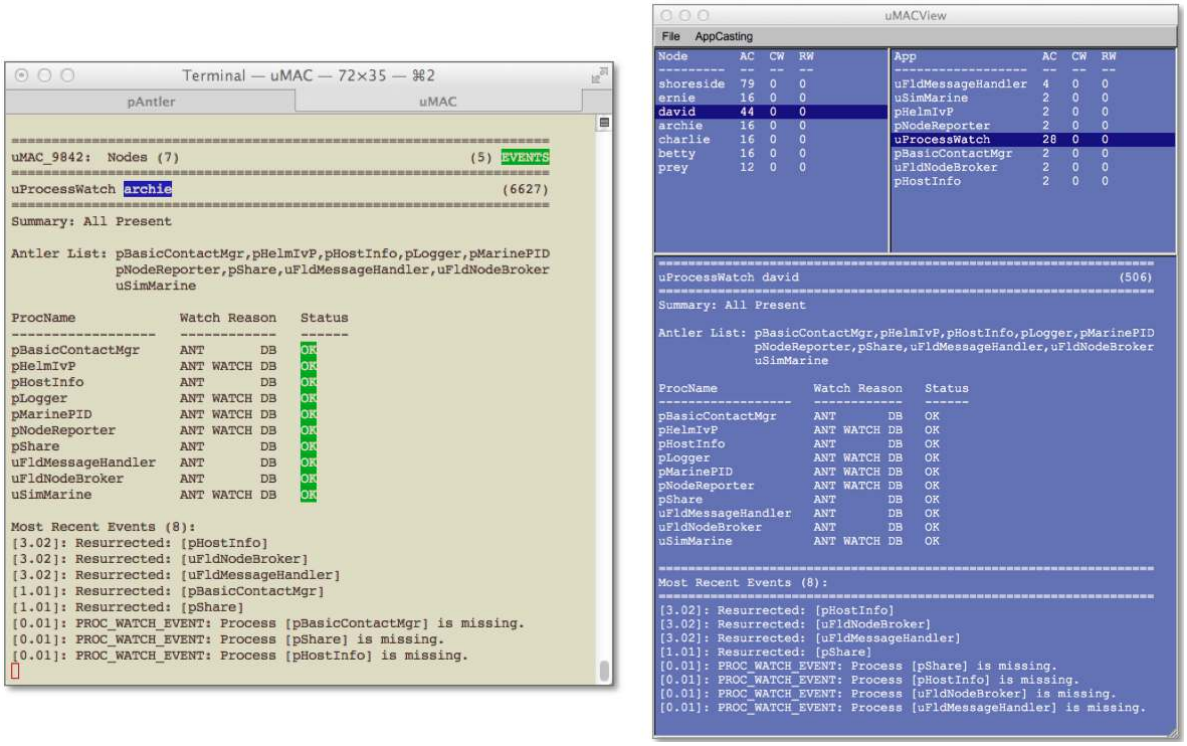

Figure 238: Two appcast viewer utilities: On the left is the terminal-based uMAC utility. On the right is the GUI-based **uMACView** utility. Both provide access to the same information. While the latter has a bit nicer user interface, the former may be run remotely while ssh'ing into a fielded vehicle.

While the uMAC and uMACView utilities are completely stand-alone and do not assume the use of any other tool, a third option exists for users of the pMarineViewer tool. This tool has been augmented to support an integrated uMACView style interface into the same single window as shown in Figure 239. The appcasting interface may be toggled on and off by simply hitting the `'a'` key. The user may also specify the mode upon startup.

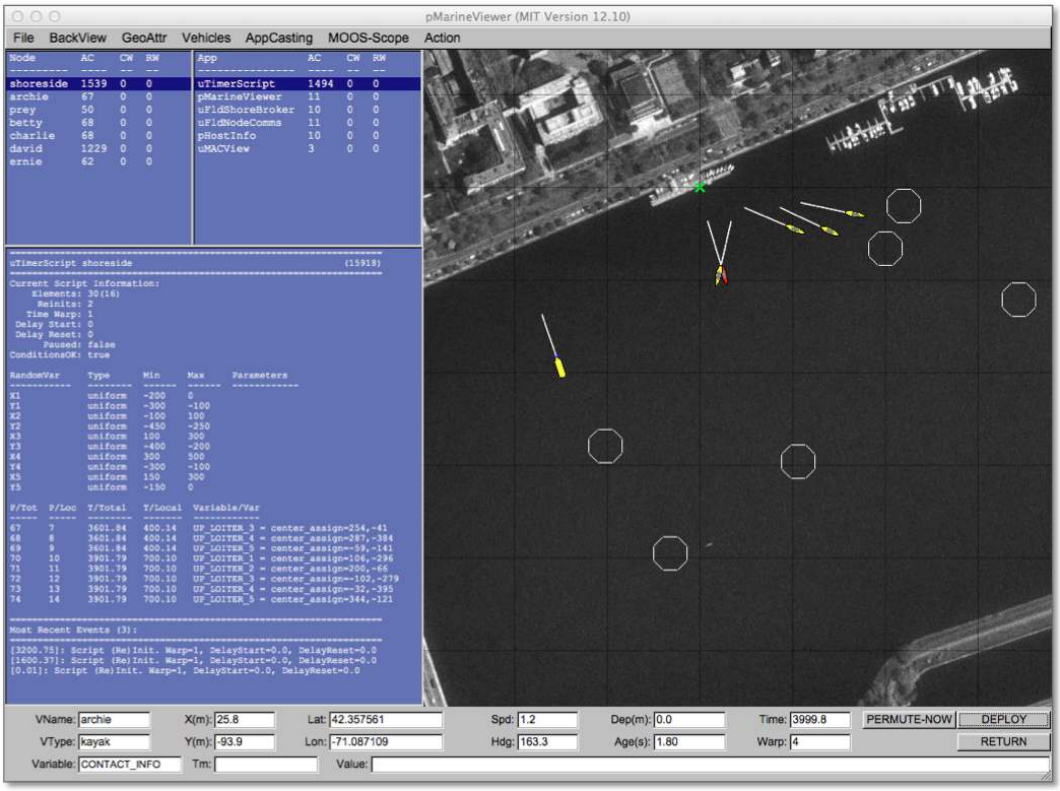

Figure 239: The pMarineViewer application has been augmented to support appcast viewing in a separate window pane. The interface is nearly identical to that of the uMACView application. The integration is for better convenience to existing pMarineViewer users. The appcasting information may be toggled on and off with the 'a' key.

90 The uMAC Utilities

90.1 Overview

In this section the utilities for viewing appcasts and/or realmcasts are discussed. They include:

- The uMACView utility: A GUI tool containing navigation tools for viewing appcasts or realmcasts across multiple vehicles or nodes. A snapshot is shown in Figure 240.
- The uMAC utility: A utility implement in the terminal window. Simpler in the interface, but notable in that it allows a user to remotely launch the tool on deployed vehicle.
- uMACView integrated with pMarineViewer: An interface nearly identical with uMACView fully integrated within pMarineViewer, convenient for those already using pMarineViewer.

Note: Starting with the first release after 2019's Release 19.8, uMACView and pMarineViewer were augmented to include realmcasting in addition to appcasting. The term *infocasting* is the general term referring to either appcasting or realmcasting. To use realmcasting, an additional app, pRealm needs to be running in each MOOS community, typically on the shoreside and in each vehicle community. More information on pRealm can be found in [**?**]

The uMAC utility currently only support appcasting output.

90.2 The uMACView Utility

The uMACView utility is an appcast or realmcast viewer with a graphical user interface using the FLTK library. It contains three panes as shown in Figure 240. The bottom pane renders incoming appcasts or realmcasts. The top right pane lists possible applications for selecting appcast or realmcast viewing for the present node. The upper left pane shows all presently known nodes from which appcasts or realmcasts have been received. In realmcasting, the upper left pane may also offer a number of variable clusters to select from.

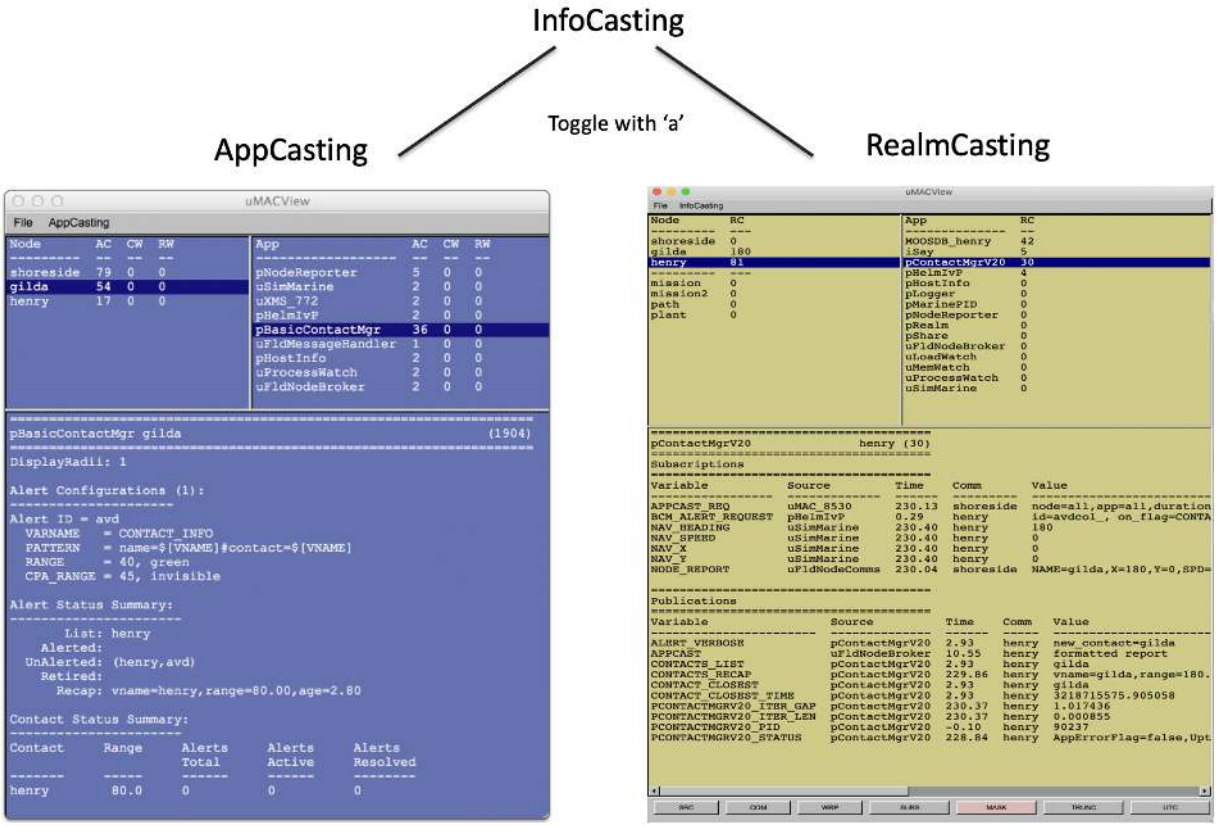

Figure 240: The uMACView utility receives appcasts or realmcasts from perhaps several different nodes and applications on each node. The interface allows the user to select a node and application for viewing the selected application's latest appcast or realmcast. The with appcasting, the viewer will raise alerts from non-selected nodes and applications when or if a run warning occurs. To toggle between appcasting and realmcasting, the `'a'` key is used.

The content mode is either *appcasting* or *realmcasting*. To toggle between the two modes, the `'a'` key is used. Alternatively the startup mode can be set with the `content_mode` configuration parameter, which is set to either `appcast` or `realmcast`, with the former being the default.

90.2.1 AppCasting

The content of the *appcast pane* is solely determined by the appcast content itself. From the viewer's perspective it is simply pushing a list of strings to a browser pane. Even the formatting of the header lines showing the application name, node name and application iteration, are formatted by a method defined over the AppCast class.

The content of the `application pane` lists the applications known thus far to the viewer from incoming appcasts for a particular node. The order is shown by the order in which they were received. The three columns, `"AC"`, `"CW"`, and `"RW"`, show the number of appcasts received from the application, and the number of run warnings and configuration warnings in the latest appcast.

The content of the *node pane* in the upper left lists the nodes known thus far to the viewer from incoming appcasts. The order is shown by the order in which they were received. The three

columns, "AC", "CW", and "RW", show the number of appcasts received from a given node for *all* applications from that node, as well as the sum of all run warnings and configuration warnings for all applications received from the given node.

90.2.2 RealmCasting

The content of the *realmcast pane* is fed from an app called pRealm running on the node selected in the node pane, and related to the application selected in the upper right app pane. A typical realmcast report is shown in Figure 241:

```
=====================================
pContactMgrV20              gilda (140)
=====================================
Subscriptions
=====================================
Variable          Source         Time    Comm       Value
----------------  -------------  ------  ---------  ----------------------------------------------------------------------------------------
APPCAST_REQ       uMAC_220       241.52  shoreside  node=all,app=all,duration=3.0,key=uMAC_220:app,thresh=run_warning
BCM_ALERT_REQUEST pHelmIvP       0.28    gilda      id=avdcol_, on_flag=CONTACT_INFO=name=$[VNAME] # contact=$[VNAME],alert_range=80, cpa_rang
                                                    e=85,match_type=mokai
NAV_HEADING       uSimMarine     242.21  gilda      180
NAV_SPEED         uSimMarine     242.21  gilda      0
NAV_X             uSimMarine     242.21  gilda      180
NAV_Y             uSimMarine     242.21  gilda      0
NODE_REPORT       uFldNodeComms  241.73  shoreside  NAME=henry,X=0,Y=0,SPD=0,HDG=180,TYPE=kayak,MODE=PARK,ALLSTOP=ManualOverride,INDEX=479,TIM
                                                    E=3218502913.32,LENGTH=4

=====================================
Publications
=====================================
Variable                 Source           Time    Comm   Value
-----------------------  ---------------  ------  -----  ------------------------------------------------------------------------------------
ALERT_VERBOSE            pContactMgrV20   2.91    gilda  new_contact=henry
APPCAST                  uProcessWatch    7.68    gilda  formatted report
CONTACTS_LIST            pContactMgrV20   2.91    gilda  henry
CONTACTS_RECAP           pContactMgrV20   242.05  gilda  vname=henry,range=180.00,age=0.69
CONTACT_CLOSEST          pContactMgrV20   2.91    gilda  henry
CONTACT_CLOSEST_TIME     pContactMgrV20   2.91    gilda  3218502674.874138
PCONTACTMGRV20_ITER_GAP  pContactMgrV20   242.04  gilda  1.004112
PCONTACTMGRV20_ITER_LEN  pContactMgrV20   242.05  gilda  0.000828
PCONTACTMGRV20_PID       pContactMgrV20   -0.12   gilda  42130
PCONTACTMGRV20_STATUS    pContactMgrV20   241.04  gilda  AppErrorFlag=false,Uptime=241.812,cpuload=0.2981,memory_kb=3224,memory_max_kb=3224,
```

Figure 241: **Example RealmCast Report:** A realmcast message is expanded into a multi-line report, consisting of a report on subscribed variables and published variables for a given application. In this case the report is for the pContactMgrV20 application on vehicle gilda.

The content of the report can be adjusted by the client, e.g., uMACViewer user, to help visualize the important information. There are seven options:

- Source: The Source column of the report may be suppressed.
- Community: The Community column of the report may be suppressed.
- UTC Time: The time format may be shown in absolute UTC time, or relative to the start of the local MOOSDB.
- Subscriptions: The subscriptions portion of the report may be suppressed.
- Mask: In the subscriptions portion, virgin variables may be suppressed.
- Wrap: Long string content may be wrapped over several lines, e.g., as in Figure 241.
- Truncate: Long string content may be truncated.

Realmcast content may be adjusted through the seven buttons on the lower part of uMACViewer. These buttons are only present when realmcasting is enabled.

The start-up value of these settings may also be set to the user's liking in the uMACViewer configuration block:

```
realmcast_show_source        = true/false   // Default is true
realmcast_show_community     = true/false   // Default is true
realmcast_show_subscriptions = true/false   // Default is true
realmcast_show_masked        = true/false   // Default is true
realmcast_wrap_content       = true/false   // Default is false
realmcast_trunc_content      = true/false   // Default is false
realmcast_time_format_utc    = true/false   // Default is false
```

90.2.3 Publications and Subscriptions

Variables Published by uMACViewer

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility.
- APPCAST_REQ_<COMMUNITY>: As an appcast viewer, pMarineViewer also generates outgoing appcast requests to MOOS communities it is aware of, including its own MOOS community. These postings are typically bridged to the other named MOOS community with the variable renamed simply to APPCAST_REQ when it arrives in the other community.

Variables Subscribed for by uMACViewer

- APPCAST: As an appcast enabled *viewer*, pMarineViewer also subscribes for appcasts from other other applications and communities to provide the content for its own viewing capability.
- REALMCAST: As an realmcast enabled *viewer*, pMarineViewer also subscribes for realmcasts from MOOS communities running pRealm, and renders them in the infocast panes described in Section 46.10.5.
- WATCHCAST: As an realmcast enabled *viewer*, uMACViewer also subscribes for watchcasts from MOOS communities running pRealm, and renders them in the infocast panes.

90.2.4 Configuration File Parameters

A few configuration parameters are exposed to facilitate visual preference settings that would otherwise need to be done each time the application is launched via pull-down menus. These parameters are listed below, but may be recalled anytime by typing on the command line: `"uMACView -e"`.

Note: uMACView was released in 2012. Starting with the first release after 2019's Release 19.8, this app was augmented to include realmcasting in addition to appcasting. The term *infocasting* is the general term referring to either appcasting or realmcasting. As such, some of the earlier configuration parameters, e.g., appcast_font_size, have been replace with a term like infocast_font_size. The older configuration parameters are deprecated but still supported foreseeable releases.

Listing 90.50: Configuration Parameters for uMACView.

appcast_color_scheme:	Possible settings, `default`, `beige`, `indigo` or `white`. The default is `indigo`.

content_mode:	Sets the on startup content, which can be changed by the user at any time. Either `appcast` or `realmcast`. The default is `appcast`.
infocast_font_size:	Possible settings, `xsmall`, `small`, `medium`,`large`,`xlarge`. The default is `medium`.
infocast_height:	Possible settings, [`30,35,40,..., 85,90`]. The default is `70`.
nodes_font_size:	The font size used in the *nodes* pane of the set of infocasting panes. Possible settings, `xsmall`, `small`, `medium`, `large`, or `xlarge`. The default is `large`.
procs_font_size:	Possible settings, `xsmall`, `small`, `medium`, `large` or `xlarge`. The default is `large`.
realmcast_color_scheme:	Either `indigo`, `beige`, `hillside`, `white`, or `default`. The default is `hillside`.
realmcast_show_source:	If `true`, the Source column is shown on realmcast output. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`.
realmcast_show_community:	If `true`, the Community column is shown on realmcast output. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`.
realmcast_show_subscriptions:	If `true`, the Subscriptions block is shown on realmcast output. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`.
realmcast_show_masked:	If `true`, certain variables are excluded in realmcast output, e.g., virgin variables. Setting to `false` conserves screen space, possibly enhancing readability. The default is `true`.
realmcast_wrap_content:	If `true`, the variable Value column in realmcast output will wrap onto several lines. Setting to `true` possibly enhances readability for very long output. The default is `false`.
realmcast_trunc_content:	If `true`, the variable Value column in realmcast output will be truncated. Setting to `true` possibly enhances readability for very long output. The default is `false`.
realmcast_time_format_utc:	If `true` the Time column in realmcast output will be shown in UTC time instead of local time since app startup. The default is `false`.
refresh_mode:	Possible settings, `paused`, `events`, `streaming`. The default is `events`.

The `infocast_height` refers to the relative height of the infocast pane to the window. The default of `70` means the pane will be `70%` of the overall window height.

The `refresh_mode` refers to the policy of refreshing appcasts via appcast requests sent to the nodes and their applications. This is discussed below in Section 90.2.6

90.2.5 Command Line Arguments and Options

A few command line arguments are available. They are similar to most other MOOS-IvP applications. These arguments are listed below, but may be recalled anytime by typing on the command line:

```
$ MACView --help or -h
```

- `--alias=<ProcessName>`: Launch with the given process name rather than uMACView.
- `--example, -e`: Display example MOOS configuration block.
- `--help, -h`: Display command line usage.
- `--interface, -i`: Display MOOS publications and subscriptions.
- `--version, -v`: Display release version information.

90.2.6 Refresh Modes

The uMACView utility, operates in one of three *refresh modes*. This mode is always shown on the upper right in the title bar in reverse color. The three modes are the *streaming*, *events*, and *paused* modes.

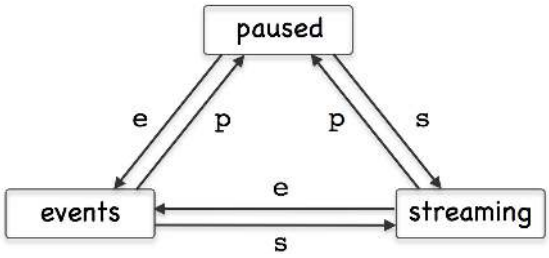

Figure 242: The uMAC utility is in one of three *refresh modes*, determining the manner in which appcast requests are conveyed to known applications and nodes. Reserved keyboard keys are used to transition between modes.

The Paused Refresh Mode In the *paused* refresh mode, no appcasts are solicited by the uMAC utility. The information being rendered should remain constant, virtually paused. Since appcast requests have a duration associated with them, prior appcast requests received by an application may need some time before expiring. Therefore the uMAC user may still see a trickle of updates after entering the paused mode. Furthermore, if there is another uMAC utility open, generating appcast requests, updates might still be seen after pausing.

The Events Refresh Mode In the *events* refresh mode, appcast requests of two types are sent to all known nodes and applications. The first type of request is sent only to the selected node and application. This type requests a continual update of appcasts, unconditionally. This application is, after all, the selected application for viewing. The second type of request is sent to all other nodes and applications requesting an appcast *only if a new run warning is generated*. The *events* refresh mode is the default mode upon launch.

The Streaming Refresh Mode In the *streaming* refresh mode, appcast requests are sent to all nodes and all applications, all the time. Furthermore, the request type is unconditional, meaning the application is requested to post a new appcast regardless of whether anything has changed in the application status. This mode is normally used for brief debugging, perhaps to check whether a node or application fails to respond. It creates a lot of appcast messaging traffic. It is not recommended to operate in this mode normally.

90.3 The uMAC Utility

The `uMAC` utility is an appcast viewer implemented in the terminal console. It acts the same as `uMACView` in terms of soliciting appcasts by publishing `APPCAST_REQ` messages and receiving `APPCAST` mail. The advantage over the GUI tool is the ability to remotely log into a vehicle and launch `uMAC` in a terminal window. This is espcially useful if the vehicle is otherwise not behaving in its communication with a shoreside MOOS community. Like `uMACView`, multiple versions of the utility may be running at the same time without interference with one another.

90.3.1 Content Modes

The viewable information in the `uMAC` tool is rendered in one of four *content modes*. Besides a *help* mode, the other three modes correlate to one of the three panes of the `uMACView` window in Figure 240. The primary content mode is the *appcast content mode*, where the output is an appcast of a particular application as shown in Figure 243.

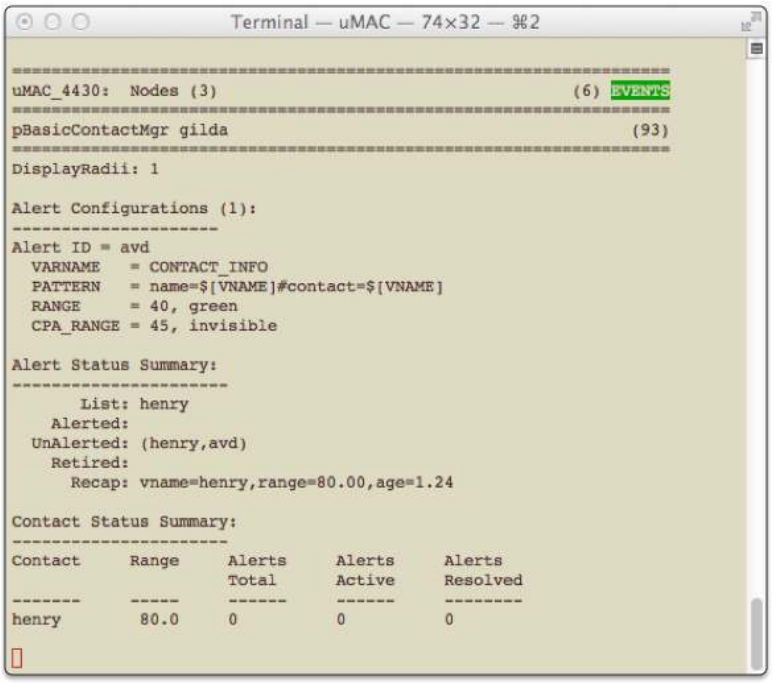

Figure 243: The `uMAC` utility monitors appcasts from a terminal window. The user may switch between appcasting sources by navigating with a keyboard menu. The primary advantage or `uMAC` is the ability to run it on a remotely deployed vehicle with a network connection.

The content of this window should look very similar to the bottom pane of the uMACView window in Figure 240. Two other content modes are supported, the *nodes content mode*, and the *procs content mode*. The former allows the user to select between different nodes, i.e., vehicles. The latter allows selection between different processes (MOOS applications) for the selected node. These modes correlate to the top two panes in the uMACView tool shown in Figure 240. A fourth, help, content mode is also supported. Transitioning between modes usually is done by selecting one of the choices presented, or popping back up a mode to allow a higher level choice. This done with three reserved keyboard keys, `p, n, h`, implementing the transitions shown in Figure 244.

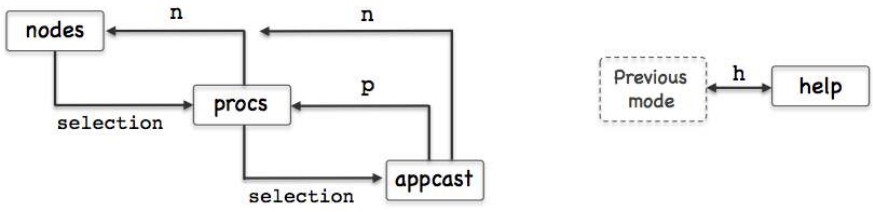

Figure 244: The uMAC utility monitors appcasts from a terminal window. The

An example rendering of uMAC in the *nodes content mode* is shown on the left in Figure 245. Each discovered node is assigned a character ID in the left-most column for selecting the node. The ID's are assigned as the nodes are discovered from incoming APPCAST messages.

The title line at the top of the report shows, on the left, the name of the uMAC application as it is known to the MOOSDB, and the number of nodes discovered. When uMAC is launched it gives itself a random suffix, such as uMAC_5991 in this example, to allow multiple uMAC sessions connect with the same MOOSDB. Recall the MOOSDB requires unique names across applications. The righthand side of the title line shows the number of iterations of the uMAC in parentheses, and the *refresh mode* shown in reverse color.

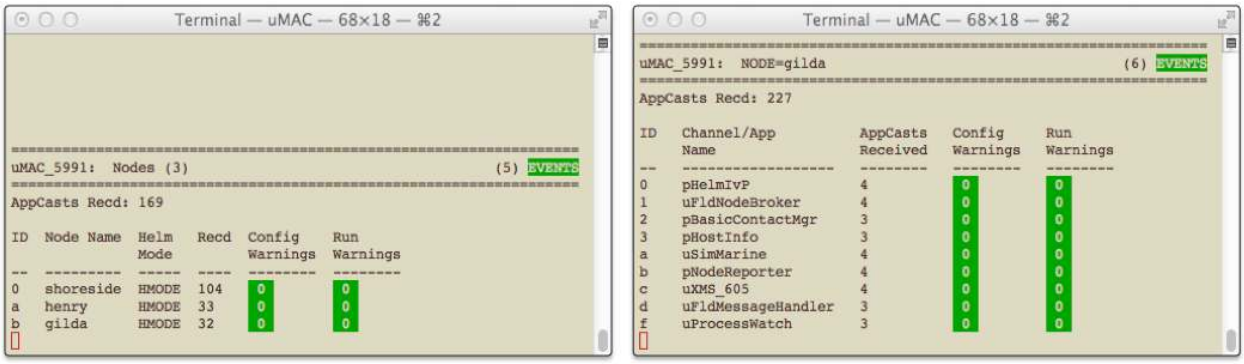

Figure 245: The uMAC utility monitors appcasts from a terminal window. The user may switch between appcasting sources by navigating with a keyboard menu. The primary advantage or uMAC is the ability to run it on a remotely deployed vehicle with a network connection.

An example from the *procs content mode* is shown on the right in Figure 245. Each discovered process (MOOS application) is assigned an ID in the left-most column for selecting the process.

The ID's are assigned as the apps are discovered from incoming APPCAST messages. The title line at the top is nearly the same format as in the *nodes* content mode, except that the selected node is shown rather than the total nodes. The appcast counter just below the title line indicates the total number of appcasts received over all nodes, not just the selected node.

90.3.2 Refresh Modes

The uMAC utility, like the uMacView utility, operates in one of three *refresh modes*. This mode is always shown on the upper right in the title bar in reverse color. The three modes are the *streaming*, *events*, and *paused* modes. The desciption of these modes, given in Section 90.2.6, is also applicable to the operation for the uMAC utility.

90.3.3 A Tip Regarding Process Monitoring and uMAC Sessions

The uProcessWatch application is a utility for monitoring the presence of applications connected to the MOOSDB. It is appcast enabled, and will post a run warning when it detects the disappearance of a prior noted process. If a uMAC is launched and then exited, the uProcessWatch utility may interpret this as a problem and post a run warning. The effect may be that real run warnings are then later ignored. Tip: Add the following configuration line to uProcessWatch to ignore the exit of a uMAC session: `nowatcth = uMAC*`.

90.3.4 Publications and Subscriptions

The sole subscription for uMACView is the appcast message in the MOOS variable APPCAST. The sole publication is the appcast request, in the variable APPCAST_REQ.

90.3.5 Configuration File Parameters

There are no configuration file parameters specific to uMAC.

Command Line Arguments and Options A few command line arguments are available. They are similar to most other MOOS-IvP applications. These arguments are listed below, but may be recalled anytime by typing on the command line:

```
$ uMAC --help or -h
```

- `--alias=<ProcessName>`: Launch with the given process name rather than uMACView.
- `--example, -e`: Display example MOOS configuration block.
- `--help, -h`: Display command line usage.
- `--interface, -i`: Display MOOS publications and subscriptions.
- `--version, -v`: Display release version information.

90.4 The uMACView Utility Integrated with pMarineViewer

The final uMAC tool is essentially uMACView embedded in the pMarineViewer application as shown in Figure 246. This is mostly just a convenience for users already using pMarineViewer. The appcasting mode may be toggled with the 'a' key to return to the traditional viewing layout. The AppCasting pull-down menu offers the same set of selections as uMACView with the exception of the hot keys used to switch between appcasting refresh modes since those keys were already used for other things in pMarineViewer.

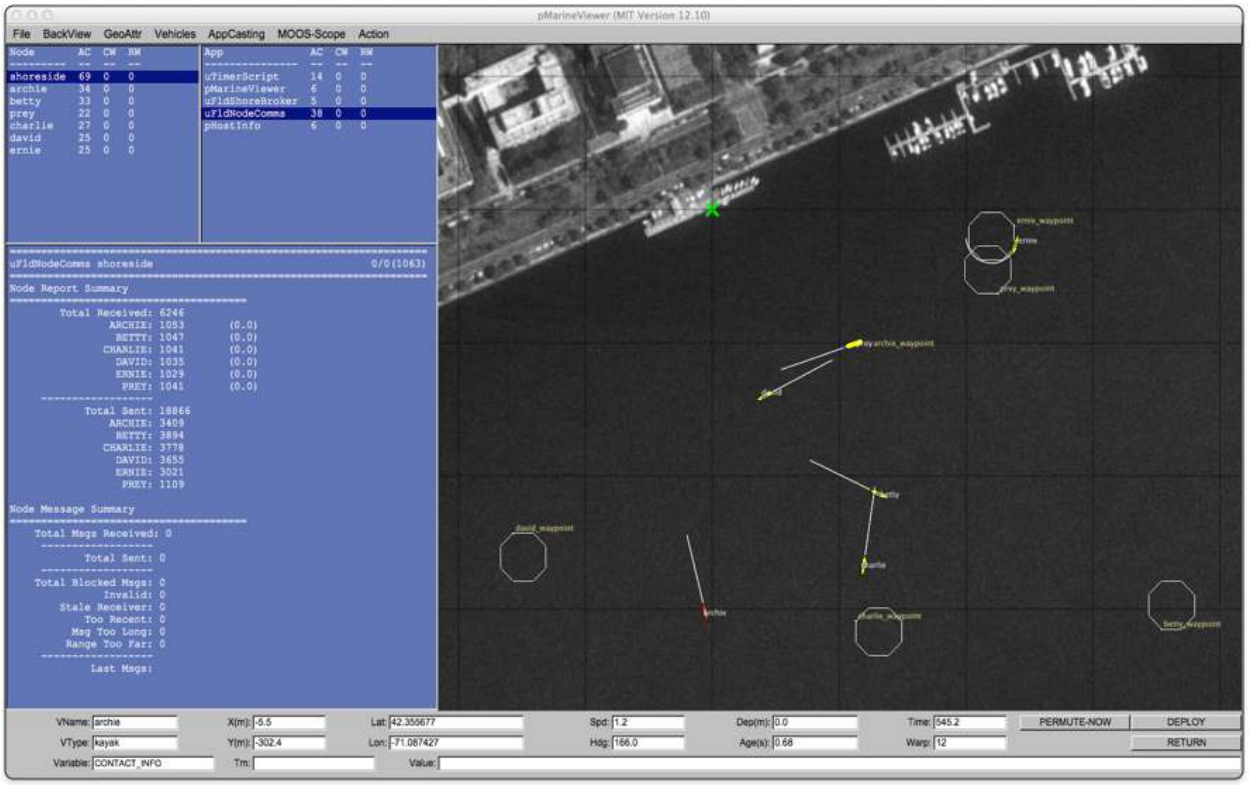

Figure 246: The pMarineViewer utility has an appcasting viewing capability very similar to uMACView embedded in the viewer. The rendering of the appcasting panes may be toggled on/off with the 'a' key.

In addition to toggling on/off the appcasting portion of the window, the width of the set of appcasting panes may be made wider or thinner using the CTRL-ALT-ARROW keys. By default the appcasting panes consume 30% of the width. The default height of the appcasting (bottom) portion of the appcasting panes consume 75% of the height of the set of appcasting panes. These startup extents may be changed with configuration the parameters:

```
appcasting_width  = 25    // legal values [20, 25,... ,65, 70]
appcasting_height = 80    // legal values [30, 35,..., 85, 90]
```

The prevailing value of these parameters can always be discovered by checking which radio button in the pull-down menu is presently selected.

91 Enabling a MOOS Application for AppCasting

91.1 Overview

In this section we discuss the steps for enabling a new or existing MOOS application to support appcasting. Much of the requisite appcasting source code is the same for any application. This is captured in a new `AppCastingMOOSApp` class to minimize appcasting boilerplate code. This class, as indicated in Figure 247, is a subclass of the common `CMOOSApp` class distributed with core MOOS.

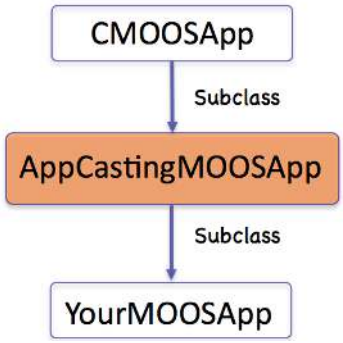

Figure 247: On-demand appcasting is implemented in a new CMOOSApp subclass called AppCastingMOOSApp.

The following is the complete list of required steps:

- Subclass the `AppCastingMOOSApp` superclass,
- Invoke a pair of superclass methods in the `Iterate()` function,
- Invoke a superclass method in the `OnNewMail()` function,
- Invoke a superclass method in the `OnStartUp()` function,
- Invoke a superclass method when registering for variables,
- Implement a `buildReport()` function where appcasts are formed.

In our discussions to follow, a hypothetical `YourMOOSApp` application and class definition is described. The above steps are the minimum requirements for appcasting, and they are fairly boilerplate, in most cases a single line of code. To make good use of the appcasting features, configuration warnings, run warnings and events may be placed in your application code. This is neither mandatory nor boilerplate, but really dependent on the application. Nevertheless, a few rules of thumb discussed:

- Posting events,
- Posting run warnings,
- Posting configuration warnings.

[2]

91.2 Sub-classing the AppCastingMOOSApp Superclass

The first step is to make `YourMOOSApp` a subclass of the `AppCastingMOOSApp`. This brings your application class everything from the traditional `CMOOSApp` class as well as the appcasting features

of the AppCastingMOOSApp class. The only additional thing besides declaring the superclass is to declare the buildReport() function. This virtual function is invoked when an appcast has been deemed warranted. Appcasts are not typically generated on each iteration. See the later discussion about *on-demand appcasting*. The contents of the buildReport() function are discussed in greater detail in Section 91.7.

Listing 91.51: Pseudocode for sub-classing the AppCastingMOOSApp superclass.

```
1  #include "MOOS/libMOOS/Thirdparty/AppCasting/AppCastingMOOSApp.h"
2
3  class YourMOOSApp : public AppCastingMOOSApp        // Instead of CMOOSApp
4  {
5    // All your normal class declaration stuff
6
7    bool  buildReport();                              // Add this line
8  };
```

91.3 Invoking Superclass Methods in the Iterate() Method

The next step is to implement YourMOOSApp::Iterate() to invoke two superclass functions; one at the very beginning and one at the very end. The first superclass function, on line 2 below, does certain common bookkeeping such as incrementing the counter representing the number of application iterations, and updating a variable holding the present MOOS time. The second superclass function, on line 6, invokes the on-demand appcasting logic discussed previously. If an appcast is deemed warranted, it will invoke the buildReport() function.

Listing 91.52: Pseudocode for invoking subclass methods in the Iterate() method.

```
1  bool YourMOOSApp::Iterate()
2  {
3    AppCastingMOOSApp::Iterate();                  // Add this line
4
5    // Do all your normal Iterate stuff
6
7    AppCastingMOOSApp::PostReport();               // Add this line
8    return(true);
9  }
```

91.4 Invoking a Superclass Method in the OnNewMail() Method

The next step is to implement YourMOOSApp::OnNewMail() to invoke a superclass function to have the first opportunity to handle incoming mail. For example, APPCAST_REQ mail is handled in the superclass. The list of mail messages is passed by reference to the superclass handler, allowing the AppCastingMOOSApp::OnNewMail() function to remove handled messages before returning to the mail handling implemented in YourMOOSApp::OnNewMail().

Listing 91.53: Pseudocode for invoking a superclass method in the OnNewMail() method.

```
1  bool YourMOOSApp::OnNewMail(MOOSMSG_LIST &NewMail)
2  {
3    AppCastingMOOSApp::OnNewMail(NewMail);        // Add this line
4
5    // Do all your other normal mail handling.
6  }
```

91.5 Invoking a Superclass Method in the OnStartUp() Method

The next step is to implement `YourMOOSApp::OnStartUp()` to invoke a superclass function to perform startup steps needed by the `AppCastingMOOSApp` superclass.

Listing 91.54: Pseudocode for invoking a superclass method in the `OnStartUp()` method.

```
1  void YourMOOSApp::OnStartUp()
2  {
3    AppCastingMOOSApp::OnStartUp();               // Add this line
4
5    // Do all your other startup stuff
6  }
```

91.6 Invoking a Superclass Method When Registering for Variables

The next step is to invoke the `AppCastingMOOSApp::RegisterVariables()` wherever variables are registered in `YourMOOSApp` implementation. Many application developers have, in practice, created a dedicated `registerVariables()` function, typically invoked at the conclusion of both `OnStartUp()` and `OnConnectToServer()`. The following example is one way to handle this.

Listing 91.55: Pseudocode for invoking a superclass method when registering for variables.

```
1  void YourMOOSApp::registerVariables()
2  {
3    AppCastingMOOSApp::RegisterVariables();      // Add this line
4
5    // Do all your other registrations
6  }
```

91.7 Implementing a buildReport Method for Generating AppCasts

The `buildReport()` function is where appcasts are made! The action that happens here is unique to the application. The form is designed by the application developer to reflect the most meaningful, concise snapshot of the application's present status. Recall that it is invoked automatically when or if the application deems an appcast is to be generated. For the purposes here though, a decision has indeed been made to generate an appcast, and `buildReport()` has been invoked to see that it happens. A simple example of `buildReport()` is shown below in Listing 56

Listing 91.56: Pseudocode for a very simple `buildReport()` example.

```
1  bool YourMOOSApp::buildReport()
2  {
3     m_msgs << "Number of good messages: " << m_good_message_count << endl;
4     m_msgs << "Number of bad  messages: " << m_bad_message_count  << endl;
5
6     return(true);
7  }
```

This simple example would generate something similar to the appcast rendered in Figure 248.

Figure 248: The rendering of a very simple appcast with just two message lines, no warnings, and no events. The header bar shows the name of the application, the originating MOOS community, the number of configuration and run warnings, and the application's current iteration counter.

Assume for the sake of the example that the two counter variables at the end of lines 2 and 3 are member variables for the fictitious `YourMOOSApp` class. The member variable `m_msgs` however is an STL stringstream also declared in the `AppCastingMOOSApp` class, for holding message output. The primary means of building an appcast is to add successive lines to the appcast via:

```
m_msgs <<   <element>   << endl;
```

where `<element>` may be a `string`, `double`, `int`, `unsigned int`, or any combination joined by the `"<<"` operator. A new line is indicated by tacking on the newline, `"\n"`, at the end. That's pretty much it, but there are a few other noteworthy points:

- Run warnings, configuration warnings, and events are not added during `buildReport()`, though not strictly prevented. They are more typically added as warnings are discovered, or events occur during the normal mail handling or iterate cycle.
- The header lines shown in Figure 248 is made automatically by the uMAC tool. They grab information in the appcast such as the application name and iteration number. These are filled by the boilerplate function calls such as `AppCastingMOOSApp::Iterate()` described earlier.
- The appcast is automatically cleared prior to each invocation of `buildReport()`. Warnings and events however are not cleared as discussed earlier.
- Returning true indicates that the appcast was indeed populated. Returning false would result in the appcast not being sent to the terminal or published to the MOOSDB. This is useful for some applications that may want to apply additional criteria before deciding to appcast.
- Although report formatting, e.g., columns or tables, is not natively supported somehow in the `buildReport()` routine, there are tools available that facilitate formatting that work easily with the `buildReport()` interface. They are discussed later in Section 99.

91.8 Posting Events

Events are messages (strings) that the application developer deems to be noteworthy enough to want to include in appcast output. An event may be posted anywhere in the application code with the `reportEvent()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportEvent("Good msg received: " + message);
```

When the appcast is rendered in either a uMAC tool or in the terminal, the result would look similar to that in Figure 249.

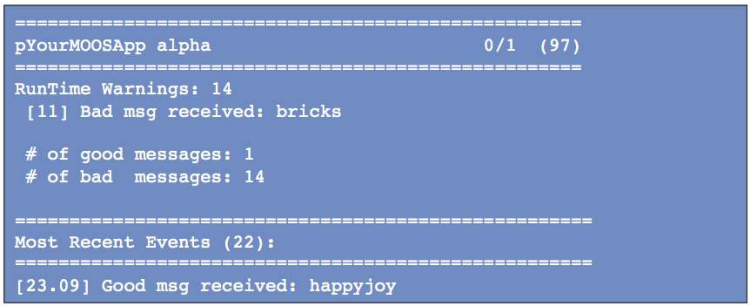

Figure 249: The rendering of a simple appcast with two message lines, a single run warning, and single event. The header bar shows the name of the application, the originating MOOS community, the number of configuration and run warnings, and the application's current iteration counter.

Recall that only a limited number of events are retained. Older events are dropped once a maximum amount is exceeded. The default event list size is eight, but this may be overridden for a particular application with the following parameter setting in the applications MOOS configuration block:

```
max_appcast_events = 25
```

The event list size may be set to at most 100.

91.9 Posting Run Warnings

Run warnings are similar to events, but they convey that something may have gone wrong. A run warning may be posted anywhere in the application code with the `reportRunWarning()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportRunWarning("Bad msg received: " + message);
```

The appcast structure and uMAC tools are implemented such that run warnings are more easily brought to the attention of the operator. This is due to the following reasons:

- When an appcast has a run warning, the uMAC utilities will indicate so by turning the text red, as in Figure 250. Even when the focus of the uMAC utility is not on the particular appcast containing the run warning, the menu items for the appcast and vehicle will be rendered red. In Figure 250 for example, the menu browser focus is on vehicle `archie` and the `uFldMessageHandler` application. The red highlights also indicate there is a run warning on another application on `archie`, and there is also a run warning on vehicle `charlie`.
- An appcast request to an application may specify the reporting threshold to be `"run_warning"`. In this case an application will repeatedly choose not to publish an appcast unless a new run warning has been generated.
- The uMAC tools also keep a running tally of run warnings for each vehicle and each application under the column labeled `"RW"` as shown in Figure 250.

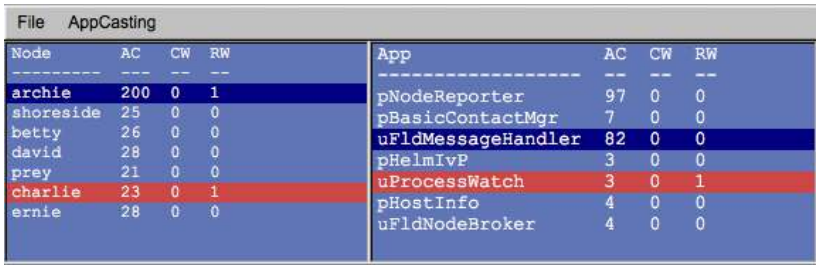

Figure 250: The uMAC tools will highlight an application that has produced an appcast with a run warning. If the run warning occurred on a node not currently in focus, e.g., charlie in the figure, the node itself is highlighted. The user can then select the other node to view the application generating the run warning.

Recall that only a limited number of run warnings are retained. Unlike events where the older ones are dropped once a maximum has been exceeded, old run warnings are never dropped. After the maximum has been reached, the generic warning `"Other Run Warnings"` is simply incremented. The default list size is ten, but this may be overridden for a particular application with the following parameter setting in the applications MOOS configuration block:

```
max_appcast_run_warnings = 50
```

The run warning list size may be set to at most 100.

91.10 Posting Configuration Warnings

Configuration warnings are similar to run warnings but they are typically only posted during the application startup, when the mission configuration file is read. A configuration warning may be posted with the `reportConfigWarning()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportConfigWarning("Problem configuring FOOBAR. Expected a number but got: " + str);
```

There is a second way to post a configuration warning. This second method takes as an argument the original full configuration parameter line found in the mission file. Before posting the configuration warning it checks to see if the parameter was something that likely was handled by the superclass. This prevents the application from reporting that `AppTick=10"` is an unknown parameter for example.

```
reportUnhandledConfigWarning(original_full_config_line);
```

The appcast structure and uMAC tools are implemented such that configuration warnings are more easily brought to the attention of the operator. This is due to the following reasons:

- When an appcast has a configuration warning, the uMAC utilities will indicate so by turning the text green, as in Figure 251. Even when the focus of the uMAC utility is not on the particular appcast containing the config warning, the menu items for the appcast and vehicle will be

rendered green. In Figure 251 for example, the menu browser focus is on the `shoreside` and the `uTimerScript` application application. The green highlights indicate there is a configuration warning in the `uFldShoreBroker` application and on other applications on `archie`, `charlie`, and `ernie`.

- The uMAC tools also keep a running tally of configuration warnings for each vehicle and each application under the column labeled `"CW"` as shown in Figure 251.

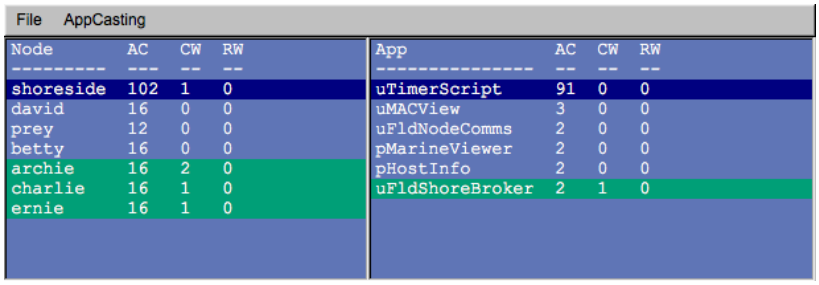

Figure 251: The uMAC tools include will highlight an application that has produced an appcast with a configuration warning. If the configuration warning occurred on a node not currently in focus, e.g., archie, charlie, or ernie in the figure, the node itself is highlighted. The user can then select the other node to view the application generating the warning.

Like run warnings and events, configuration warnings are limited in number to prevent runaway growth in the size of an appcast over time. The limit however is large, 100, and fixed. Presumably the number of configuration warnings is limited by the number of possible configuration parameters for an application, and large number of configuration warnings usually indicates that a mission should be halted and fixed before moving on.

The example code in Listing 57 below is an example `OnStartUp()` method showing the intended scenarios of reporting configuration warnings.

Listing 91.57: Pseudocode example for `OnStartUp()` configuration warning handling.

```
1  bool YourMOOSApp::OnStartUp()
2  {
3    AppCastingMOOSApp::OnStartUp();
4
5    STRING_LIST sParams;
6    if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
7      reportConfigWarning("No config block found for " + GetAppName());
8
9    STRING_LIST::iterator p;
10   for(p=sParams.begin(); p!=sParams.end(); p++) {
11     string orig  = *p;
12     string line  = *p;
13     string param = toupper(MOOSChomp(line, "="));
14     string value = line;
15
16     if(param == "FOO") {
17       bool handled = handleConfigFOO(value);
```

```
18        if(!handled)
19          reportConfigWarning("Problem with configuring FOO: " + value);
20      }
21      else if(param == "BAR")
22        bool handled = handleConfigBAR(value);
23        if(!handled)
24          reportConfigWarning("Problem with configuring BAR: " + value);
25      }
26
27      else
28        reportUnhandledConfigWarning(orig);
29    }
30    return(true);
31  }
```

There are a few issues worth noting in this example:

- A check is made that the application actually has a configuration block. This is done on lines 5-6, and catches a common bug with newly minted mission files.
- Checks are made that known parameters have legal values. This is done for the parameters `FOO` in lines 15-19 and `BAR` in lines 20-24 in Listing 56. In each case an external handler is invoked, e.g., `handleConfigFOO(value)` on line 16, which returns a Boolean indicating whether the parameter value was proper or not. If not, a configuration warning is reported as on lines 18 and 23.
- A final case is handled (lines 26-27) if the present parameter is not matched by any of the previous cases. This catches the common mistake of mis-spelling the parameter name. The `reportUnhandledConfigWarning()` function is used rather than the `reportConfigWarning()` function. The former function takes the whole original configuration line as input and checks to see if the parameter was a parameter handled at the superclass level. This prevents the generation of a warning for a line like `AppTick=5`.

92 Under the Hood of On-Demand AppCasting

92.1 Overview

On-demand appcasting refers to the goal of minimizing generated appcasts, ideally only when there is a reasonable chance that an appcast will be tended to (looked at) by a user. Users may be reading an appcast either by looking at terminal output or through a uMAC utility.

92.2 Motivation

Consider the following scenario:

- 20 simulated vehicles,
- 10 MOOS applications on each vehicle,
- Each application running with an apptick of 4Hz,
- MOOS time warp running at 25x real time.

In the above scenario (a conceivable scenario in our lab), $20,000$ appcasts would be generated *per second*. Consider a second scenario:

- One fielded underwater vehicle,
- 10 MOOS applications,
- Each application running with an apptick of 4Hz,
- Mission duration 6 months.

Although only 40 appcasts per second are generated, the vehicle is underwater and, limited to acoustic messages, likely no appcast will ever be viewed. With a six month mission, the CPU time and power budget needed to generate those 40 reports per second may come under scrutiny.

92.3 AppCast Generation Criteria

An appcast will be generated on any given iteration only if the contingencies and criteria depicted in Figure 252 are met. In short, an appcast is generated if either a terminal is open or a uMAC tool is requesting the appcast. Even when appcasts are being generated, they may be generated less frequently than each iteration, to be more in line with the frequency a user is able to process refreshed report information. These issues are discussed next.

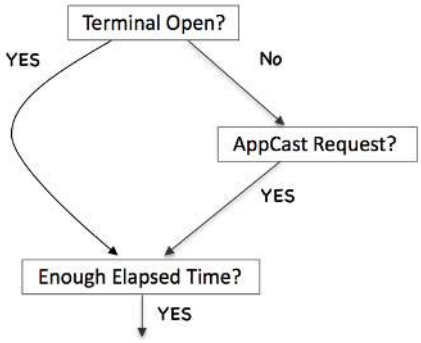

Figure 252: The appcast-generation decision is based on a few checks and contingencies. An appcast is generated only when tended to by a user, and only as often as a user may reasonably expect to process updated reports.

92.4 Terminal Switching

In the flow diagram of Figure 252, the first step in determining whether an appcast is to be generated involves the presence of a terminal. If an app is launched with a terminal window open, or launched within a terminal window, appcasts should be generated. In this regard, appcasting apps should behave like non-appcasting apps, although the former produce its output by different means. From the user's perspective, launching an application with or within a terminal window should result in the same familiar behavior regardless of the application.

Upon startup an AppCastingMOOSApp application will try to determine if information directed to `stdout` will make its way to an open terminal window. The following reasoning is applied:

- By default the application assumes information directed to `stdout` is indeed being rendered in a terminal window.
- During startup, when mission file parameters are examined, if the application detects the presence of a global parameter, `TERM_REPORTING`, and it is set to `"false"`, then the application assumes that a terminal window is not open to receive output written to `stdout`.
- During application startup, the following library utility function is invoked:
  `int isatty(int);`
  This function is defined in `"unistd.h"` and is able to detect whether `stdout` is receiving output.

The above ensures that the application will err on the side of always producing appcasts/output to the terminal. To ensure otherwise, make sure to include `TERM_REPORTING="false"` in the MOOS configuration file. The last check is just a convenience or extra fail-safe; if an application is launched with `pAntler` using `NewConsole=false` and `pAntler` itself is also launched with `stdout` redirected to `/dev/null/`, then the application will automatically detect this. This style of launching is actually a fairly common scenario in launching MOOS communities on vehicles in our lab. In detecting this situation, the application will not generate an appcast unless a uMAC tool is explicitly requesting it. This is the next step in the flow diagram of Figure 252, and discussed next.

92.5 AppCast Requests

An *appcast request* is message sent to an application to begin or continue generating appcasts for some specified period of time. The request may originate in the local MOOSDB community, or in a

remote MOOSDB community as the two cases suggest in Figure 253.

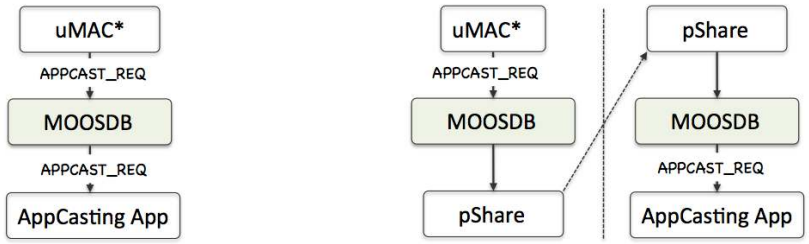

Figure 253: An appcast request requests that the receiving MOOS application begin or continue to generate appcasts for some specified period of time. The request may come from an app within the same MOOS community, or from an external community.

The content of an appcast request has the following fields:

- **node**: This must match the name of the MOOS community within which the application is running, otherwise the appcast request is ignored. If the `node` is `"any"`, the match is always granted.
- **app**: This must match the name of the MOOS application receiving the appcast request, otherwise the request is ignored. If the `app` is `"any"`, the match is always granted.
- **duration**: This number is given in seconds and represents the duration the appcast request would honored after time of receipt. The maximum value is 30.
- **threshold**: The threshold name indicates under what conditions the receiving application should generate an appcast. The two possible values are `"any"` and `"run_warning"`. Their meaning is discussed below.
- **key**: The key helps the receiving application discern appcast requests from different applications.

An example `APPCAST_REQ` message:

```
APPCAST_REQ = "node=henry,app=uProcessWatch,duration=3.0,key=pMarineViewer:shore,thresh=any"
```

92.5.1 Node and Application Name Matching

An appcast request will be ignored by a receiving application if the request does not *match*. The match must be made between both (a) the requested and actual node name, and (b) the requested and actual application name. A requested value `"any"` will match to anything. In most circumstances the requesting application, e.g., `uMAC`, `uMACView` or `pMarineViewer`, will publish requests naming nodes and applications explicitly. Upon startup however, requests may be sent broadly to all nodes and all applications just to learn about existing appcasting sources before beginning to make requests explicitly.

92.5.2 Duration Time

An appcast request comes with a duration. If the requesting application disconnects, the duration caveat ensures the original request will timeout before too long, avoiding the perpetual generation of now unwanted appcasts. Typically if a uMAC tool is monitoring the appcasts of a particular application, it repeatedly sends an appcast request to that application, each time refreshing the timeout criteria.

92.5.3 Request Threshold

The appcast request will specify one of two criteria to the application. First, if the criteria is `"any"`, the application is to publish an appcast on each possible occasion. This may not be the same as each iteration, since a further minimum reporting interval may be applied, as described next in Section 92.6. The second threshold type is `"run_warning"`. This indicates to the receiving application that it should only generate an appcast when a new run warning has been added since the last appcast. Typically a uMAC tool will run in a mode sending appcast requests with the latter threshold to virtually all nodes and apps, but appcast requests using the former threshold to a single node and app chosen by the user.

92.5.4 Request Key

AppCast requests specify a particular key, presumably a string that is unique to the originator. This allows the receiving application to do separate bookkeeping for each requesting application. As long as the appcast request threshold matches, hasn't timed out, and met the threshold criteria for *one* of its logged keys, then it indeed meets the criteria.

92.6 Limiting the AppCast Frequency

A third criteria is applied to the decision of generating an appcast on any given iteration. This criteria is depicted at the bottom of Figure 252. Even if a terminal window is open, or a valid appcast request has been received recently enough, the generation of an appcast may still be skipped if the previously generated appcast was generated too recently. AppCast content is meant to be human-readable, and humans can only read so fast. There is no sense in updating a terminal report say 50 times per second. Although most MOOS apps are typically not configured with an apptick of 50Hz, an apptick of 5 is fairly common, as well as a `MOOSTimeWarp` of say 10 or greater. By default, appcasting applications are limited in real-time frequency to once every 0.6 seconds. This number just reflects a number that, from experience, feels right for an update frequency. This can be overridden for any application with the following parameter in its configuration block:

```
term_report_interval = N
```

where N ranges from zero (appcast generation only limited by the iteration frequency) to at most 30 seconds.

92.7 Generating and AppCast vs. Publishing and AppCast

The flow chart shown in Figure 252 addresses the issue of whether or not an appcast is *generated*. Whether or not the appcast is *published* is a separate issue. Simply put, if a terminal window is open for the application, and it has not received any appcast requests, the appcast is generated (and rendered to the terminal `stdout`) but not published.

Below is informal logic shorthand for policy and conditions described over the last few pages. First, on the policy of whether an appcast is generated:

```
generate_appcast = (terminal || appcast_requested) && !recent_appcast
```

Next, on the question of whether an appcast is published:

```
publish_appcast  = generate_appcast && appcast_requested
```

Both of the above depend on the term `appcast_requested` from the following expression:

```
appcast_requested = unexpired_request && ((request_threshold == "any") || new_run_warning)
```

The terms in this last expression will hopefully be apparent from the preceding pages.

92.8 Monitoring AppCast Traffic Volume

The uMAC tools provide a means for verifying that on-demand appcasting is behaving. These same tools are also useful for catching other anomalies. The information in Figure 254 below focuses on the information at the top of the `uMACView` tool depicted in Figure 254.

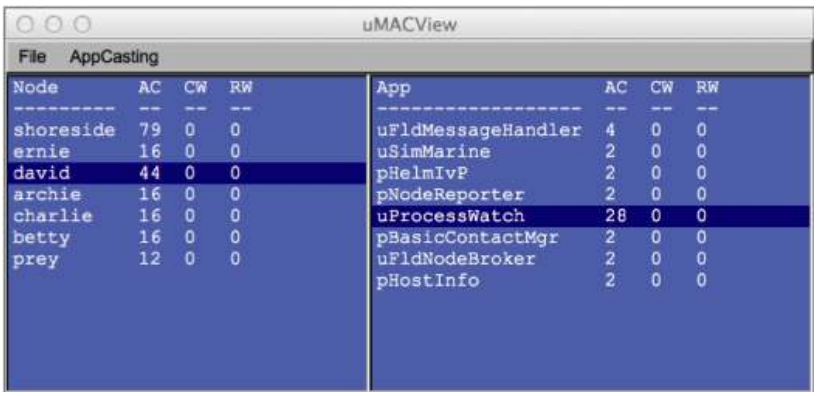

Figure 254: The uMAC tools include tallies of the number of appcasts received for each node (vehicle) and each application. The above is from the top of the `uMACView` and appcast tallies are shown under the `"AC"` column.

From the above figure, one might ask why so many appcasts have been received when the user is only focusing on one vehicle and one application. There are few answers to this question and each, listed below, are related to the need for a uMAC utility to *find* existing sources of appcasts. After all, how can it send an appcast request to a particular vehicle and particular application if it doesn't know that it exists?

- When an application starts up, it generates appcasts for the first few iterations. Even if no one is tending to this information, a few iterations worth of un-tended appcasts is not harmful. In practice this helps uMAC tools discover appcasting apps.
- When a uMAC tool starts up, it posts an appcast request to all known nodes and all known applications. The uMAC tool doesn't need to know or keep track of vehicles, but just simply posts `APPCAST_REQ="node=all,app=all, ..."` to its local MOOSDB. Other applications have the responsibility of bridging this variable to other vehicles as they are discovered.
- Each time a uMAC tool receives a new appcast from a vehicle/node it has never heard from before, it responds by posting an appcast request back to that vehicle for all applications, e.g.., `APPCAST_REQ="node=henry,app=all, ..."`. This request will time-out shortly, but is usually sufficient to learn about all applications on that node. Subsequent requests are then more selective.

93 uFldNodeBroker: Brokering Node Connections

93.1 Overview

The uFldNodeBroker application is a tool for brokering connections between a node (a simulated or real vehicle) and a shoreside community. It is used primarily in coordination with uFldShoreBroker to discover and share host IP and port information to automate the dynamic configurations of pShare. Inter-vehicle communications over the network are handled by pShare in both simulation with single or multiple machines as well as on fielded vehicles using Wi-Fi or cellphone connections. The pShare application simply needs to know the IP address and port number of connected machines. Often these aren't known at run-time and even if they were, maintaining that information in configuration files may be cumbersome, especially for large sets of vehicles. This tool is meant to automate the configuration by letting the nodes and shoreside community discover each other by giving the node (uFldNodeBroker) some initial hints on where to find the shoreside community on the network. The typical layout is shown in Figure 255.

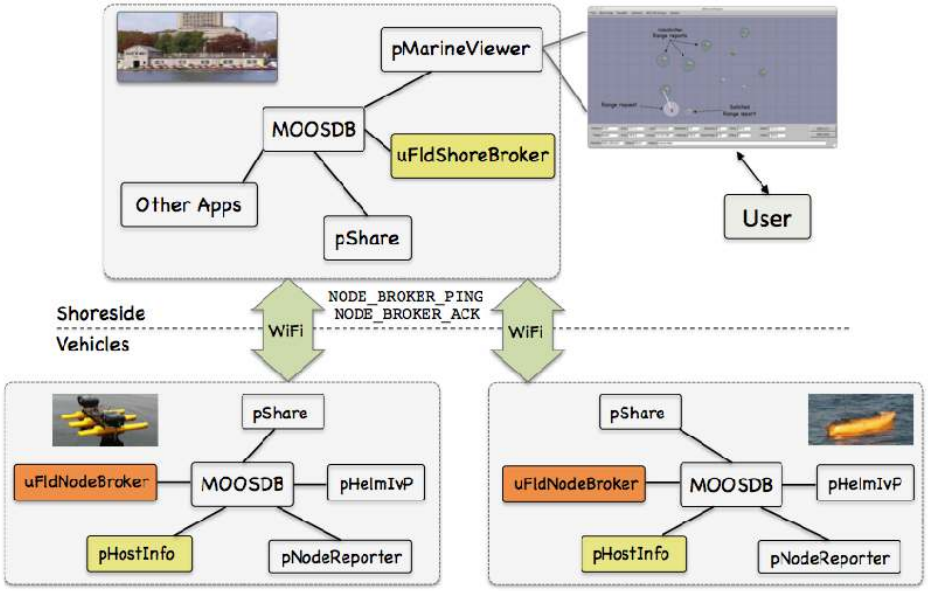

Figure 255: **Typical uFldNodeBroker Topology:** A vehicle (node) sends information about itself (IP address and port number) to possible shoreside locations. Once a connection is made, further sharing is established between the node and shoreside communities.

The functionality of uFldNodeBroker paraphrased:

- Discover the node's host information (typically from pHostInfo).
- For candidate shoreside hosts, request a new share configuration in pShare to each candidate for the variable NODE_BROKER_PING.
- Publish NODE_BROKER_PING with the node's host information.
- Await a reply in the form of incoming NODE_BROKER_ACK mail, presumably from the shoreside community running uFldShoreBroker.

- Now that a shoreside community is known, request new share configurations from the node's local pShare for all the info otherwise wanted for sharing to the shoreside.
- Keep sending pings periodically in case the shoreside community is re-started and needs to re-establish connections to nodes.

93.2 The uFldNodeBroker Interface and Configuration Options

The uFldNodeBroker application may be configured with a configuration block within a .moos file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section. If one has access to a command line where uFldNodeBroker has been built, interface information may also be seen by typing "uFldNodeBroker --interface", and configuration information by typing "uFldNodeBroker --example".

93.2.1 Configuration Parameters of uFldNodeBroker

The following parameters are defined for uFldNodeBroker.

Listing 93.58: Configuration Parameters for uFldNodeBroker.

auto_bridge_appcast:	Suppress the normal automatic bridging of the APPCAST variable. The default is false.
auto_bridge_realmcast:	Suppress the normal automatic bridging of the REALMCAST and REALMCAST_CHANNELS variables. The default is false.
bridge:	A variable to register with pShare for bridging to a shoreside MOOS community once a shoreside connection has been established.
try_shore_host:	A candidate route to send send initial pings in hopes of establishing a connection. The route specifies an input route for pShare running in a shoreside community.

Since appcasting and realmcasting are common and recommended practices for users of the uField Toolbox, key variables that are bridged are done automatically. Normally the APPCAST variable is auto bridged to support appcasting. And the REALMCAST and REALMCAST_CHANNELS variables are auto bridged to support realmcasting. The user has the option to opt out of this configuration by setting the auto_bridge_* configuration parameters described above. Note that realmcasting was first introduced to the uField Toolbox and MOOS-IvP code in the first release after 19.8.1.

93.2.2 An Example MOOS Configuration Block

An example MOOS configuration block can be obtained by entering the following from the command-line:

```
$ uFldNodeBroker --example or -e
```

Listing 93.59: Example configuration of the uFldNodeBroker application.

```
1  ============================================================
2  uFldNodeBroker Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = uFldNodeBroker
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   keyword        = lemon
11
12   try_shore_host = pshare_route=localhost:9200
13   try_shore_host = pshare_route=192.168.0.122:9301
14   try_shore_host = pshare_route=multicast_8
15
16   bridge = src=VIEW_POLYGON
17   bridge = src=VIEW_POINT
18   bridge = src=VIEW_SEGLIST
19
20   bridge = src=NODE_REPORT_LOCAL, alias=NODE_REPORT
21 }
```

93.3 Publications and Subscriptions for uFldNodeBroker

The interface for uFldNodeBroker, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldNodeBroker --interface or -i
```

93.3.1 Variables Published by uFldNodeBroker

The primary output of uFldNodeBroker to the MOOSDB are the requests to pShare for registrations, and the outgoing pings to candidate shoreside communities.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 93.4.
- NODE_BROKER_PING: A message written locally but bridged to a candidate shoreside MOOS community, containing IP address and pShare route information about the local community.
- PSHARE_CMD: message to pShare to add a new bridge for a given variable and given target MOOS community at a specified IP address and port number.

93.3.2 Variables Subscribed for by uFldNodeBroker

The uFldNodeBroker application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- NODE_BROKER_ACK: Information published presumably by uFldShoreBroker running in a separate shoreside community. Message has information about the shoreside host including the community name, IP address and port numbers for the MOOSDB and its local pShare process.

- PHI_HOST_INFO: Information about the local host IP address, the MOOS community name, the port on which the DB is running, and the port on which the local pShare is listening for UDP messages.

93.3.3 Command Line Usage of uFldNodeBroker

The uFldNodeBroker application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing

```
$ uFldNodeBroker --help or -h
```

Listing 93.60: Command line usage for the uFldNodeBroker tool.

```
1  ============================================================
2  Usage: uFldNodeBroker file.moos [OPTIONS]
3  ============================================================
4
5  Options:
6    --alias=<ProcessName>
7        Launch uFldNodeBroker with the given
8        process name rather than uFldNodeBroker.
9    --example, -e
10       Display example MOOS configuration block.
11   --help, -h
12       Display this help message.
13   --interface, -i
14       Display MOOS publications and subscriptions.
15   --version,-v
16       Display release version of uFldNodeBroker.
```

93.4 Terminal and AppCast Output

The uFldNodeBroker application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 61 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. The counter on the end of line 2 is incremented on each iteration of uFldNodeBroker, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 93.61: Example terminal or appcast output for uFldNodeBroker.

```
1  ====================================================================
2  uFldNodeBroker henry                                        0/0(129)
3  ====================================================================
4
5    Total OK  PHI_HOST_INFO    received: 13
6    Total BAD PHI_HOST_INFO    received: 0
7    Total HOST_INFO changes    received: 1
8    Total          PSHARE_CMD    posted: 7
9    Total BAD NODE_BROKER_ACK  received: 6
```

```
10
11  ============================================================
12               Vehicle Node Information:
13  ============================================================
14
15       Community: henry
16          HostIP: 10.0.0.5
17     Port MOOSDB: 9001
18       Time Warp: 4
19        IRoutes: 10.0.0.5:9301
20
21  ============================================================
22               Shoreside Node(s) Information:
23  ============================================================6
24
25  Community                   Pings  Pings  IP        Time
26  Name       Route            Sent   Acked  Address   Warp
27  ---------  --------------   -----  -----  --------  ----
28  shoreside  localhost:9300  128    120    10.0.0.5  4
29  Phase Completion Summary:
30  -----------------------------------
31   Phase 1: (Y) Valid Host information retrieved (iroutes).
32   Phase 2: (Y) Valid TryHosts (1) configured.
33   Phase 3: (Y) NODE_BROKER_PINGS are being sent to TryHosts.
34   Phase 4: (Y) A Valid NODE_BROKER_ACK has been received.
35   Phase 5: (Y) pShare requested to share user vars with shoreside.
36  All Phases complete. Things should be working as configured.
```

On line 5, the number of incoming mail messages for PHI_HOST_INFO is tallied where the host information bundle is deemed complete. It is complete if it contains the host community, IP address, time warp and pShare input route information. If an incomplete host information packet is received, it is tallied in line 6. The number on line 6 should always be zero. If the number on line 6 is not zero, or the number on line 5 never increments, you should check the operation of the pHostInfo.

The number on line 7 indicates the number of time the detected host information changes. This number should stabilize very quickly maxing out at 1 or 2 typically. The number on line 8 indicates the number of dynamic bridge requests posted to pShare. These are in the form of postings to the variable PSHARE_CMD, which occur first for outgoing pings to candidate shoreside hosts, and then for the user bridge configuration variables after a shoreside host has been connected. Invalid NODE_BROKER_ACK messages are tallied on line 9. An invalid ack may be due to a mismatch in time warp between node and shoreside, or a mismatch in keywords if keywords are being used.

Self node information is displayed in the next group, lines 11-19. The community name, MOOSDB port, and time warp are all read from the .moos mission configuration file. The node's IP address (line 16) and the local pShare input routes (line 19) are obtained from output received from pHostInfo.

Information about candidate and connected shoreside communities is shown next in lines 21-28. In this case there is only one entry, line 27. For each candidate entry, the community name, shoreside pShare input route, number of pings sent and acknowledged are in the first four columns. The last two columns indicate the IP address and time warp of the shoreside learned from NODE_BROKER_ACK messages received from the shoreside. When a candidate shoreside community has *not* been connected, the entry in this table will something like:

```
Community                   Pings  Pings  IP       Time
Name       Route            Sent   Acked  Address  Warp
---------  ---------------  -----  -----  -------  ----
           localhost:92003  385    0
```

The last block of information in the report is the Phase Completion Summary, lines 29-36. It lists the rough sequence of events typical to reach a shoreside community connection. If any one of these phases is incomplete, the output on line 36 will be replaced with a few hints on where to troubleshoot.

94 uFldShoreBroker: Brokering Shore Connections

94.1 Overview

The uFldShoreBroker application is a tool for brokering connections between a shoreside community and one or more nodes (simulated or real vehicles). A shoreside community is collection of MOOS processes typically running a GUI providing a situational display and managing messages to and from fielded vehicles. This is depicted in the notional rendering in Figure 256 below. The shoreside community in practice is often situated on a ship with UUVs below, and is more aptly referred to as the topside community. The user interacts with the GUI or perhaps other communication modules, to send high-level messages to the vehicles.

The uFldShoreBroker application is used primarily in coordination with uFldNodeBroker, running on the vehicles, to discover and share host IP and port information to automate the dynamic configurations of pShare. Inter-vehicle communications over the network are handled by pShare in both simulation with single or multiple machines as well as on fielded vehicles using Wi-Fi or cellphone connections. The pShare application simply needs to know the IP address and port number of connected machines. Often these aren't known at run-time and even if they were, maintaining that information in configuration files may be unduly cumbersome, especially for large sets of vehicles. This tool is meant to automate the configuration by letting the nodes and shoreside community discover each other by letting (uFldShoreBroker) respond to incoming pings, i.e., initialization messages, from nodes on the network. The typical layout is shown in Figure 256.

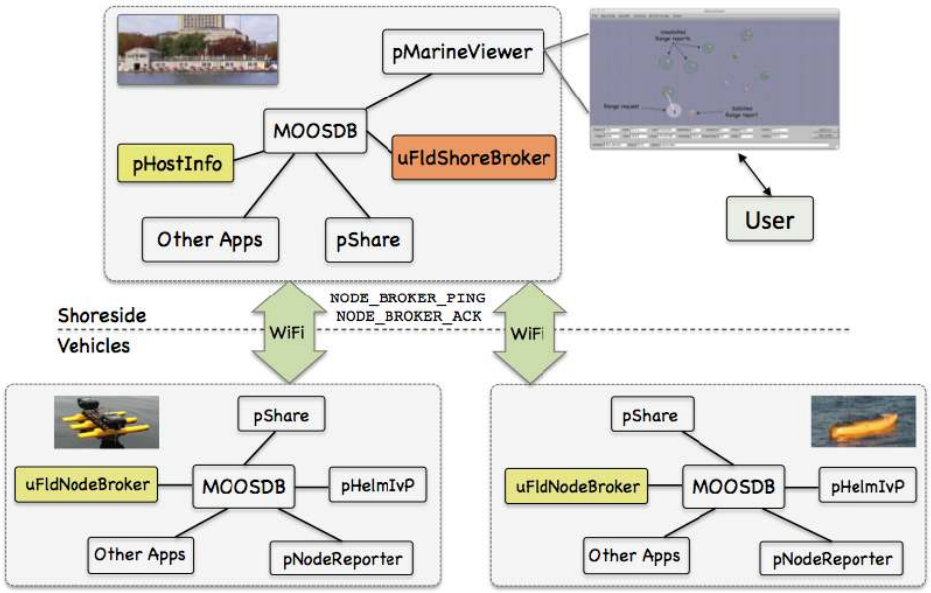

Figure 256: **Typical uFldShoreBroker Topology:** A vehicle (node) sends information about itself (IP address and port number) to the shoreside, received by uFldShoreBroker. It responds by (a) acknowledging the connection to the node, and (b) establishing user configured bridges of particular MOOS variables to the node.

The functionality of uFldShoreBroker paraphrased:

- Discover the shoreside's own host information (typically from pHostInfo).

- Await incoming NODE_BROKER_PING messages from non-local vehicles.
- Upon an incoming ping, respond to the nodes with a NODE_BROKER_ACK message to the location specified in the ping message.
- Establish new bridges to the nodes for variables specified previously by the user in the uFldShoreBroker configuration.
- Keep sending acknowledgments periodically to confirm to vehicles that they are still connected to the shoreside community.

94.2 Bridging Variables Upon Connection to Nodes

A primary function of uFldShoreBroker is to establish bridging relationships to a remote node community after it has received a ping from that community. These variables are specified in the configuration block with lines like `bridge = "src=DEPLOY_ALL, alias=DEPLOY"` as in Listing 64. This step is described next.

94.2.1 Inter-MOOSDB Bridging with pShare

Static bridging with pShare is done by specifying the desired route in the pShare configuration block with a line of the form:

```
output = src_name=VAR, dest_name=ALIAS, route=ROUTE
```

For example:

```
output = src_name=DEPLOY_HENRY, dest_name=DEPLOY, route=12.56.111.1:9200
```

The above connection may be used to send the vehicle Henry the deploy command from the shoreside community. The problem is that the shoreside may not know the IP address of Henry (or the port on which it's pShare is listening) until it presents itself to the shoreside at run time.

In this case, dynamic share registration needs to be used by sending pShare a message after it has been launched. For example, the above sharing relationship could be established by sending the following message:

```
PSHARE_CMD = "cmd=output, src_name=DEPLOY_ALL, dest_name=DEPLOY,
              route=2.56.111.1:9200
```

It is the job of uFldShoreBroker to post the above style dynamic requests once the node information becomes known to the shoreside community.

94.2.2 Handling a Valid Incoming Ping from a Remote Node

The basic job of uFldShoreBroker is to await incoming pings, and use the information in a ping message to (a) decide if the ping should be accepted, and (b) send the appropriate response back to the sender, and (c) set up new outgoing pShare relationships if the ping is indeed accepted. The contents of a ping may look something like:

```
NODE_BROKER_PING = "community=henry,host=192.168.1.22,port=9000,time_warp=10
                    pshare_iroutes=192.168.1.22:9200,time=1325178800.81"
```

There must be a match in the MOOS *time warp* used by the shoreside MOOS community and any node connected to the shore. This is always 1 when operating vehicles in the field, but may be set to a much larger number in simulation. The time warp is set with the parameter MOOSTimeWarp at the top of the .moos configuration file.

The ping consists of three key pieces of information:

- The community name of the node,
- The IP address of the node,
- The input routes on which the node is listening for messages with its own local pShare running.

Once this information is known by the shoreside broker, new bridges can be established for variables identified by the user. The only other information needed is (a) the name of the variable in the local MOOSDB, and (b) the name (alias) of the variable as it is to be known in the remote MOOSDB. Once a valid ping has been received and accepted, uFldShoreBroker is ready to establish bridge arrangements with its local pShare running.

94.2.3 Vanilla Bridge Arrangements

The simplest bridge arrangement specifies (a) the variable as it is known locally, and (b) the variable name as it is to be known remotely. This is done with a uFldShoreBroker configuration line similar to:

```
bridge = src=DEPLOY_ALL, alias=DEPLOY
```

For *each* unique incoming ping, a new bridge arrangement will be requested. By unique, we mean having a distinct community (remote vehicle) name. For example, if pings are received and accepted from *henry*, *james*, and *ike*, uFldShoreBroker would make three separate posts, perhaps looking like:

```
PSHARE_CMD = "src_name=DEPLOY_ALL, dest_name=DEPLOY, route=2.56.111.1:9200"
PSHARE_CMD = "src_name=DEPLOY_ALL, dest_name=DEPLOY, route=2.56.111.3:9200"
PSHARE_CMD = "src_name=DEPLOY_ALL, dest_name=DEPLOY, route=2.56.111.6:9200"
```

At this point the behavior of pShare on the shoreside would be functionally equivalent to the scenario where the following three lines were in the pShare configuration block:

```
output = src_name=DEPLOY_ALL, dest_name=DEPLOY, route=12.56.111.1:9200
output = src_name=DEPLOY_ALL, dest_name=DEPLOY, route=12.56.111.3:9200
output = src_name=DEPLOY_ALL, dest_name=DEPLOY, route=12.56.111.6:9200
```

94.2.4 Bridge Arrangements with Macros

The user may configure uFldShoreBroker with bridge arrangements containing a couple types of macros. For example, consider the configuration:

```
bridge = src=DEPLOY_$V, alias=DEPLOY
```

The $V macro will expand to the name of the vehicle when it comes time to request a new bridge. If the newly received ping is from the node named *gilda*, the bridge request from the above pattern may look like:

```
PSHARE_CMD = cmd=output, src_name=DEPLOY_GILDA, dest_name=DEPLOY,
             route=2.56.111.1:9200
```

Note the vehicle name in the MOOS variable macro was expanded to be upper case, even though the ping information referred to the vehicle as *gilda*. This is just to aid in the convention that MOOS variable are typically all upper case. If one really want a literal expansion with no case altering, the macro $v, lower-case v, may be used instead. The macro is only respected as part of src field. In other words, if the bridge were configured with alias=DEPLOY_$V, the macro would not be expanded.

The other type of macro implemented is the $N macro, as in:

```
bridge = src=LOITER_$N, alias=LOITER
```

The $N macro will expand to the integer value representing number of unique pings received thus far. For example, if three pings are received and accepted from *henry*, *james*, and *ike*, uFldShoreBroker would make three separate posts, perhaps looking like:

```
PSHARE_CMD = "src_name=LOITER_1, dest_name=LOITER, route=2.56.111.1:9200"
PSHARE_CMD = "src_name=LOITER_2, dest_name=LOITER, route=2.56.111.4:9200"
PSHARE_CMD = "src_name=LOITER_3, dest_name=LOITER, route=2.56.111.12:9200"
```

This may be useful when used in conjunction with another MOOS process generating output generically for N vehicles, without having to know the vehicle names in advance.

94.2.5 A Common Configuration Shortcut - the qbridge Parameter

A common usage pattern is to configure uFldShoreBroker to request two types of bridges for a given variable, for example:

```
bridge = src=DEPLOY_ALL, alias=DEPLOY
bridge = src=DEPLOY_$V,  alias=DEPLOY
bridge = src=RETURN_ALL, alias=RETURN
bridge = src=RETURN_$V,  alias=RETURN
```

This could be use in the shoreside community for easily commanding vehicles. When the user wishes to deploy all vehicles, a posting of DEPLOY_ALL="true" does the trick. If the user wishes only the vehicle *james* to return, a posting of RETURN_JAMES="true" may be be made. This pattern is so common that this shortcut is supported. This is done with the qbridge, "quick bridge", parameter. The above four configuration lines could be accomplished instead by:

```
qbridge = DEPLOY, RETURN
```

94.3 Usage Scenarios for the uFldShoreBroker Utility

The uFldShoreBroker was designed with a canonical command-and control scenario in mind. The idea is that the N deployed vehicles have a common autonomy protocol implemented. For example, a message to deploy or return a vehicle is the same message for each deployed vehicle. The idea is that two types of communication channels need to be established with pShare, (a) messages sent to all vehicles, and (b) messages sent to a particular named vehicle. The convention proposed here is to do this with the two types of bridging described in the discussion of the qbridge parameter, in Section 94.2.5. For a variable such as DEPLOY, a posting in the shoreside community to DEPLOY_ALL would go to all known vehicles, and a posting to DEPLOY_HENRY would only go to that particular vehicle.

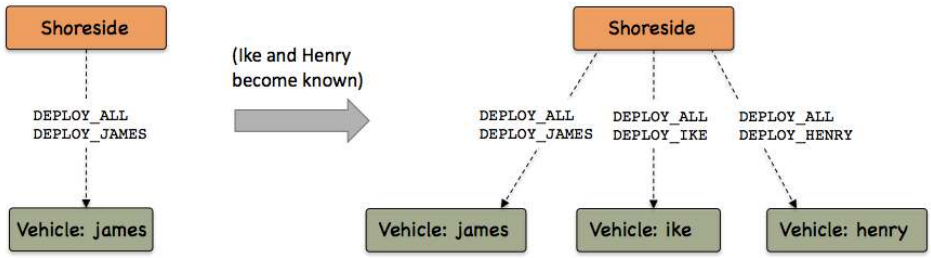

Figure 257: **Common uFldShoreBroker Usage Scenario:** As vehicles become known to the shoreside, each vehicle has two new bridges established. The first is the same for all vehicles to allow broadcasting, and the second bridge is unique to the particular vehicle, for individual command and control.

94.4 Terminal and AppCast Output

The uFldShoreBroker application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 62 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. See the appcasting documentation for more on appcasting and viewing appcasts.

On line 1, the application iteration is shown, more as a heartbeat indicator. In lines 4-7, the primary variables consumed and posted by uFldShoreBroker are summarized in terms of how many posts have been made and received for each variable.

Listing 94.62: Example terminal output of the uFldShoreBroker tool.

```
1  ============================================================
2  uFldShoreBroker_PS shoreside                          (109)
3  ============================================================
```

```
 4
 5   Total PHI_HOST_INFO    received: 11
 6   Total NODE_BROKER_PING received: 180
 7   Total NODE_BROKER_ACK    posted: 180
 8   Total PSHARE_CMD         posted: 34
 9
10
11  ============================================================
12              Shoreside Node(s) Information:
13  ============================================================
14
15       Community: shoreside
16          HostIP: 128.30.27.202
17     Port MOOSDB: 9000
18       Time Warp: 6
19         IRoutes: localhost:9200
20
21  ============================================================
22               Vehicle Node Information:
23  ============================================================
24
25  Node   IP                     Elap  pShare
26  Name   Address        Status  Time  Input Route(s)      Skew
27  -----  -------------  ------  ----  ------------------  ------
28  henry  128.30.27.202  ok      0.0   128.30.27.202:9301  1.6542
29  gilda  128.30.27.202  ok      0.0   128.30.27.202:9302  1.7572
30
31  Recent Events (2):
32  [22.06]: New node discovered: gilda
33  [16.04]: New node discovered: henry
```

In lines 11-19, the key shoreside properties are listed. Typically, but not always, the shoreside community is name "shoreside" as indicated on line 15. The shoreside IP address, determined by pHostInfo, is shown on line 16. The time warp, and MOOSDB port are read from the shoreside .moos file and listed on lines 17 and 18. The input routes used by pShare are listed on line 19. If lines 16 or 19 are blank, uFldShoreBroker will not make any connections and the first place to look is whether or not pHostInfo is running and producing valid information.

In lines 21-33, the status of each of the known vehicles is shown. The first two vehicles had their pings accepted. Their IP addresses are shown in the second column. Their status is shown in the third column. The elapsed time in the fourth column is the time since the last ping was received by the shoreside. The fifth column shows the input routes being used by pShare running on the vehicle node. If multiple routes are in use, this will be shown over multiple lines. The sixth column shows the time skew between the timestamp in the NODE_BROKER_PING message compared to the time it was received. Some of this is due to (a) latency in transmission, (b) latency due to App Ticks in brokers on both sides, and (c) clock discrepancy between the shoreside and the node computers. It's also worth mentioning that the skew will be magnified for higher time warps. Currently incoming ping connection requests are not denied due to a high clock skew, but this may be implemented in the future.

94.5 Configuration Parameters of uFldShoreBroker

The following parameters are defined for uFldShoreBroker.

Listing 94.63: Configuration Parameters for uFldShoreBroker.

auto_bridge_appcast:	Suppress the normal automatic bridging of the APPCAST_REQ variable. The default is false.
auto_bridge_mhash:	Suppress the normal automatic bridging of the MISSION_HASH variable. The default is false.
auto_bridge_realmcast:	Suppress the normal automatic bridging of the REALMCAST_REQ variable. The default is false.
bridge:	Names a MOOS variable to be bridged to a node community. Section 94.2.3.
keyword:	Optionally set a keyword. If set, incoming NODE_BROKER_PING messages must contain this keyword or else they ping will not get a response. Section TBD.
qbridge:	Shorthand notation for a common bridging pattern. Section 94.2.5.
try_vnode:	TBD
warning_on_stale:	If true, generate a warning if a previously known node has not been heard from in more than 10 seconds.

As an example, bridge = src =DEPLOY_ALL, alias=DEPLOY, will result in the bridging of variable DEPLOY_ALL from the local MOOSDB, to the variable DEPLOY in a remote MOOS community. Further examples are given in Section 94.2.

94.5.1 An Example MOOS Configuration Block

Listing 64 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uFldShoreBroker --example or -e
```

Listing 94.64: Example configuration of the uFldShoreBroker application.

```
1  ============================================================
2  uFldShoreBroker Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = uFldShoreBroker
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   warning_on_stale      = false (default)
11   auto_bridge_realmcast = true  (default)
```

```
12    auto_bridge_appcast   = true  (default)
13    auto_bridge_mhash     = true  (default)
15
15    bridge =  src=DEPLOY_ALL, alias=DEPLOY
16    bridge =  src=DEPLOY_$V,  alias=DEPLOY
17
18    try_vnode = 192.168.4.24:9200
19    try_vnode = 192.168.4.25:9200
20
21    qbridge = RETURN
22    qbridge = NODE_REPORT, STATION_KEEP
23
24    bridge  = src=UP_LOITER_$N, alias=UP_LOITER
25
26    // Note: [qbridge = FOO]  is shorthand for
27/   //       [bridge = src=FOO_$V,  alias=FOO] and
28    //       [bridge = src=FOO_ALL, alias=FOO]
29
30    app_logging = true  // false is default
31 }
```

94.6 Publications and Subscriptions for uFldShoreBroker

The interface for uFldShoreBroker, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldShoreBroker --interface or -i
```

94.6.1 Variables Published by uFldShoreBroker

The primary output of uFldShoreBroker to the MOOSDB are the requests to pShare for registrations, and the outgoing acknowledgment replies to remote node/vehicle communities.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 94.4.
- PMB_REGISTER: A message to pShare to add a new bridge for a given variable and given target MOOS community at a specified IP address and port number.
- NODE_BROKER_ACK: A message written locally but bridged to a remote vehicle MOOS community, containing IP address and port information about the local shoreside community.

94.6.2 MOOS Variables Subscribed for by uFldShoreBroker

The uFldShoreBroker application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- PHI_HOST_INFO: Information about the local host IP address, the MOOS community name, the port on which the DB is running, and the port on which the local pShare is listening for UDP messages.

- NODE_BROKER_PING: Information published presumably by uFldNodeBroker running in a remote vehicle community. Message has information about the node host including the community name, IP address, the port number for the MOOSDB, input route(s) for the local pShare process.

94.6.3 Command Line Usage of uFldShoreBroker

The uFldShoreBroker application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$       uFldShoreBroker --help or -h
```

Listing 94.65: Command line usage for the uFldShoreBroker tool.

```
1  ============================================================
2  Usage: uFldShoreBroker file.moos [OPTIONS]
3  ============================================================
4
5  Options:
6    --alias=<ProcessName>
7        Launch uFldShoreBroker with the given
8        process name rather than uFldShoreBroker.
9    --example, -e
10       Display example MOOS configuration block.
11   --help, -h
12       Display this help message.
13   --interface, -i
14       Display MOOS publications and subscriptions.
15   --version,-v
16       Display release version of uFldShoreBroker.
```

95 uFldNodeComms: Simulating Inter-vehicle Communications

95.1 Overview

The uFldNodeComms application is a tool for handling node reports and messages between vehicles. Rather than directly sending node reports and messages between vehicles, uFldNodeComms acts as an intermediary to conditionally pass a report or message on to another vehicle, where conditions may be the inter-vehicle range or other criteria. The assumption is that uFldNodeComms is running on a topside or shoreside computer, and receiving information about the present physical location of deployed vehicles through node reports. The typical layout is shown in Figure 258.

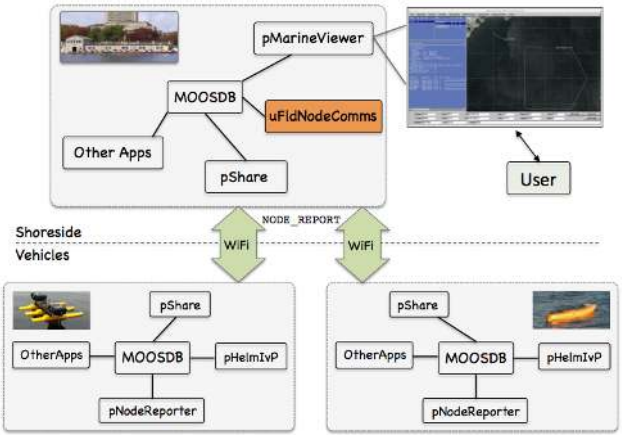

Figure 258: **Typical uFldNodeComms Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

In short, uFldNodeComms subscribes for incoming node reports for any number of vehicles, and keeps the latest node report for each vehicle. On each iteration, for a each vehicle, if the node report has been updated, the report is published to a specially created MOOS variable for the other n-1 vehicles. A user-configured criteria is applied before publishing the new information. Typically this criteria involves the range between vehicles, but the criteria may be further involved. The idea is conveyed for three vehicles *alpha*, *bravo*, and *charlie*, shown below in Figure 259.

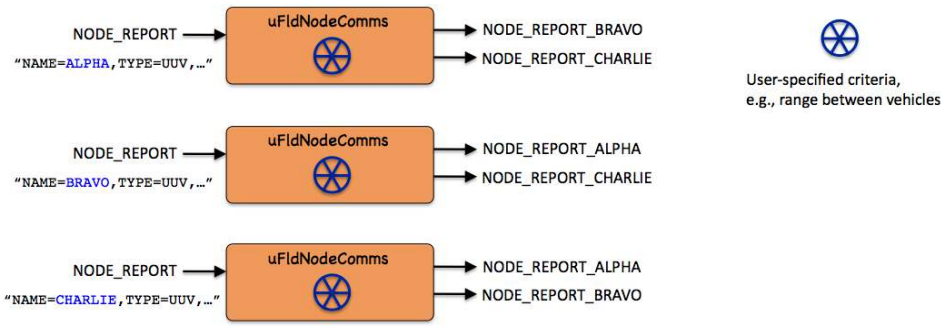

Figure 259: **Brokering by uFldNodeComms:** Each incoming node report is sent out to a specially named variable corresponding to one each of the other n-1 vehicles.

95.1.1 The Difference Between a Node Report and Node Message

There are two distinct message types in play with uFldNodeComms:

- Node reports - concise vehicle trajectory/state, in NODE_REPORT
- Node messages - open ended inter-vehicle content, in NODE_MESSAGE

A *node report* contains concise information regarding a vehicle's state, including its position, speed, heading and depth for underwater vehicles. It may also include information regarding the helm, including the current all-stop state and helm mode. It is typically published by pNodeReporter running on each vehicle, and generated by default at 4Hz. This may be faster for higher speed vehicles operating in close proximity, and lower for slower moving vehicles operating in expansive waters. A node report is typically contained in one of two variables, either NODE_REPORT_LOCAL or NODE_REPORT. The former is used by pNodeReporter to indicate the report has been generated locally about ownship. When this information is sent off-board, to the shoreside, uFldNodeComms, and eventually other vehicles, it converted to the variable NODE_REPORT.

A *node message* contains open ended inter-vehicle message content, in the form of (a) a destination vehicle, (b) a MOOS variable name, and (c) the value of the MOOS variable. Not all messages sent will arrive at their destination and uFldNodeComms is the arbiter of which messages get through. There is no single app the generates a node message. Any app, by the configuration of the user, may publish an outgoing message to NODE_MESSAGE_LOCAL. This indicates the message was generated on ownship to be sent elsewhere. In the uField simulation scheme, using uFldNodeComms, it will arrive on the shoreside as NODE_MESSAGE and, if allowed through, will arrive on the destination vehicle(s) as NODE_MESSAGE. The receiving vehicle will unpack the incoming node message using an app called uFldMessageHandler.

In short, uFldNodeComms reasons about node reports and node messages, but it uses node reports to have situational awareness of all vehicle positions. It uses this situational awareness to be the arbiter of which node messages are allowed to travel between which vehicles, depending on the restrictions configured by the user as uFldNodeComms parameters.

95.2 Handling Node Reports

Node reports may contain a bundle of useful information for sharing between vehicles. Perhaps the most important is the present vehicle position and trajectory. This may be used by the receiving vehicle for collision avoidance and formation keeping and so on. The vehicle position is also used by uFldNodeComms to determine if the node reports themselves are to be shared between vehicles. The inter-vehicle ranges derived from the NODE_REPORT messages are also used to determine if other more generic information contained in the NODE_MESSAGE variable is to be shared between vehicles.

95.2.1 The Criteria for Routing Node Reports

The basic criteria for sharing node reports is range. By default, the node report received for any one vehicle is passed on to *all* other known vehicles within comms range of the originating vehicle. The comms range is set by the parameter comms_range as on line 9 in Listing 68. Starting with this as a baseline, a few other factors may be involved in determining whether a node report is shared.

95.2.2 Using Vehicle Group Information

If uFldNodeComms is configured with the parameter groups set to true, as on line 21 in Listing 68, then node reports are only shared between vehicles having the same group name. The group name is a field contained in the node report itself, so the onus is on the vehicle to include this information as part of its report. The pNodeReporter application contains an optional configuration parameter group=<group-name> where the group information is declared for inclusion in all node reports. The motivation for the grouping option is to support multi-vehicle competitions where some vehicles want to convey positions to teammates, but not adversarial vehicles.

The groups feature only affects the passing of node reports between vehicles. It does not affect the passing of node *messages* discussed further below. Node messages contain their addressee information explicitly.

The msg_groups feature only affects the passing of node *messages* between vehicles. It does not affect the passing of node *reports* discussed above. This feature is introduced in the release following 19.8.2.

95.2.3 Establishing and Inter-Vehicle Critical Range

When the two vehicles are within a range deemed critical, as set by the critical_range configuration parameter as on line 10 in Listing 68, node reports are shared between vehicles regardless of the comms_range parameter and the groups parameter. The default for this parameter is 30 meters. The thought behind this feature is that, while it may be advantageous to not broadcast your own vehicle position to non group members for the purposes of a competition, it may be a good idea to share this information for the sake of collision avoidance.

95.2.4 Checking for Staleness

Since up-to-date inter-vehicle range information is used as part of the criteria in determining whether a vehicle receives a new node report from another, the position of the candidate recipient vehicle needs to reasonably up-to-date. The stale_time configuration parameter may be set as on line 11

in Listing 68, to determine the amount of elapsed time without receiving a node report from a vehicle before it is considered stale. If a recipient vehicle becomes stale, it will also not receive node messages.

95.2.5 Ignoring a Group

The user may specify a group name with the ignore_group parameter. All incoming node reports that match this group will simply be ignored. Introduced after Release 19.8.x.

95.2.6 Optional Limitation on the Node Report Share Rate

The min_report_interval, given in seconds, sets the minimum amount of time between node report messages from one vehicle to another. The default value is -1, disabling this feature.

Note that when the view_node_rpt_pulses parameter is set to true, then uFldNodeComms also publishes a visual artifact in a posting to VIEW_COMMS_PULSE. By limiting the node report share rate, the rate of this variable posting is limited too. This may also be a consideration in extreme missions of very high numbers of vehicles and very high time warp, for easing the burden on the viewer, e.g., pMarineViewer.

95.2.7 Node Report Transmissions and pShare

Node reports are communicated to recipient vehicles in coordination with the pShare application. For each unique vehicle name discovered by uFldNodeComms via received node reports, it will publish a new MOOS variable NODE_REPORT_NAME. This variable will be published with the contents of other vehicles' node reports.

As shown in Figure 259, if three vehicles, *alpha*, *bravo*, and *charlie* become known, three corresponding MOOS variables NODE_REPORT_ALPHA, NODE_REPORT_BRAVO, and NODE_REPORT_CHARLIE will be published. The variable NODE_REPORT_ALPHA will be published with reports from *bravo* and *charlie* and so on. To make this happen, uFldNodeComms needs the corresponding share relationships from, for example, NODE_REPORT_ALPHA in the shoreside community to the variable NODE_REPORT in the alpha community on the alpha vehicle. This sharing relationship is established separately by the uFldShoreBroker application running in the shoreside community.

95.3 Asymmetric Node Report Sharing

Under normal circumstances, if one vehicle is within range to send a node report or message to another, it should also be able to receive node reports and messages from the same vehicle. In this section we discuss ways to make this relationship asymmetric. This is done in one of two ways:

- Increasing a vehicle's *stealth*, i.e., requiring other receiving vehicles to be closer than normal to receive reports and messages, or
- Increasing a vehicle's *earange*, i.e., allowing other receiving vehicles to be farther than normal and still receive node reports and messages.

95.3.1 Bestowing a Vehicle with an Enhanced Stealth Property

By using the *stealth* parameter, one vehicle may be given a bit of an advantage. The stealth parameter is assigned to a particular vehicle and is a number in the range [0.5, 1]. A message from source to destination is sent if:

$$actual\_range(src, dest) < comms\_range * stealth(src) \tag{12}$$

By default, the stealth factor for each vehicle is 1. A stealthy vehicle with a factor of one half, means the receiving vehicle needs to be twice as close as it would otherwise to receive the stealthy vehicle's node report or node message. The `critical_range` parameter may be used to override a vehicle's stealthiness with respect to sending node reports. That is, a message from source to destination is sent if:

$$actual\_range(src, dest) < max((comms\_range * stealth(src)), critical\_range) \tag{13}$$

The `stealth` value for a particular vehicle may be set in the MOOS configuration file as shown on line 18 in Listing 68. It may also be set dynamically by receiving mail on the variable `UNC_STEALTH`. For example the posting

```
UNC_STEALTH = vname=alpha, stealth=0.75
```

would immediately reset the stealth factor for vehicle *alpha* on the next iteration. The motivation for exposing this parameter via incoming MOOS mail is that another shoreside application, monitoring vehicle speed for example, may reward a vehicle operating in the field with increased stealth based on any criteria. This can be useful in constructing vehicle competitions.

95.3.2 Bestowing a Vehicle with an Enhanced Listening Property

In addition to the stealth property, a complementary property may be bestowed upon a vehicle using the `earange` parameter. The earange parameter is assigned to a particular vehicle and is a number in the range [1, 2]. A message from source to destination is sent if:

$$actual\_range(src, dest) < comms\_range * earange(dest) \tag{14}$$

By default, the earange factor for each vehicle is 1. A vehicle with a factor of two may receive node reports of another vehicle at twice the range it may otherwise. In the end the stealth and earange properties may cancel each other out when their extreme values are factored together. Taken together a message from source to destination is sent if:

$$actual\_range(src, dest) < comms\_range * stealth(src) * earange(dest) \tag{15}$$

The `earange` value for a particular vehicle may be set in the MOOS configuration file as shown on line 19 in Listing 68. It may also be set dynamically by receiving mail on the variable `UNC_EARANGE`. For example the posting

```
UNC_EARANGE = vname=alpha, earange=1.5
```

would immediately reset the earange factor for vehicle *alpha* on the next iteration. The motivation for exposing this parameter via incoming MOOS mail is that another shoreside application, monitoring vehicle speed for example, may reward a vehicle operating in the field with increased earange based on any criteria. This can be useful in constructing vehicle competitions.

95.4 Handling Node Messages

Node messages are of a generic structure for sharing a MOOS variable-value pair between a source node and a destination node. A node message from vehicle *alpha* to vehicle *bravo* would be posted locally by *alpha* with something like:

```
NODE_MESSAGE = "src_node=alpha,dest_node=bravo,var_name=FOOBAR,string_val=hello"
```

The local NODE_MESSAGE posting is shared to the shoreside community running uFldNodeComms where it is considered for re-routing to the destination vehicle.

95.4.1 The Criteria for Routing Node Messages

The basic criteria for sharing node messages is (a) the message addressee, and (b) the range between the source and destination vehicles. Since uFldNodeComms is receiving and keeping track of incoming node reports from all vehicles, it has ready access to the inter-vehicle range between the source and destination nodes. The same criteria used for sending node *reports* is used for sending node messages. It must meet the range criteria as perhaps modified by the stealth and earange factors discussed earlier. The only difference is that the critical_range parameter is irrelevant for the issue of sending node messages. This parameter was only used for node reports in the interest of safety and collision avoidance. There are however two other factors that may affect node message transmission, discussed next, message frequency and message size.

95.4.2 Enforcing a Minimum Time Between Node Messages

A maximum send frequency is enforced by requiring a minimum wait time between successful sends from a given source node. This minimum time is given by the parameter min_msg_interval as on line 13 in Listing 68. The default is 30 seconds. This interval is defined by the time starting with a successful transmission of a node message from a source to any destination. A separate log is kept by uFldNodeComms for each known vehicle.

Further clarification: Suppose for example, the min_msg_interval is set to 60 seconds. If a vehicle attempts an outgoing message 58 second after the previous message, it will not go through. If attempted 4 seconds later or 62 seconds after the previous successful message, then it will go through. It's as if the unsuccessful attempt never happened.

Continuing this example, consider a vehicle that attempts a message 62 seconds after the previous message was sent. Suppose this new message fails due to the inter-vehicle range being to large, the message length being too long, or the destination node not being recognized. In this case the failure results in another full 60 seconds being needed before the next message will be handled.

95.4.3 Enforcing a Maximum Node Message Length

The length of node messages may be limited with the parameter max_msg_length, as on line 14 in Listing 68. The default maximum length is 1000 characters. The length of a message refers to the number of characters in the string_val field. For example, the length of the message

```
NODE_MESSAGE = "src_node=alpha,dest_node=bravo,var_name=FOOBAR,string_val=hello"
```

is five. Limiting the message length is a proxy for inter-vehicle communications where message packet length is constrained, as with acoustic communications for example.

95.4.4 Posting Messages to a Vehicle Group

A node message is typically addressed to another named vehicle. The sender may also address the message by group name. All vehicles in the group meeting the prevailing range criteria will receive the message. The group associated with the vehicle is declared in the node report sent by that vehicle. A node message using a group address is similar except for the use of the dest_group parameter

```
NODE_MESSAGE = "src_node=alpha,dest_group=red_team,var_name=FOOBAR,string_val=hello"
```

The sender has the option of indicating *both* a group name and vehicle name as the message destination. It may also specify more than one vehicle name explicitly. Thus the following is allowed:

```
NODE_MESSAGE = "src_node=alpha,dest_name=bravo:charlie:gilda,dest_group=red_team,
               var_name=FOOBAR,string_val=hello"
```

If a destination vehicle is specified twice in the list of destinations or implicitly in the named group, the message will be sent only once to the destination vehicle.

95.5 Visual Artifacts for Rendering Inter-Vehicle Communications

Each time a node report or message is sent to a vehicle by uFldNodeComms, another posting is made to the variable VIEW_COMMS_PULSE. This message may be subscribed for by another application using it to visually render the communications events. The screen shot in Figure 260 below shows four vehicles. The two bottom vehicles are sharing node reports indicated by the red and blue comms pulses.

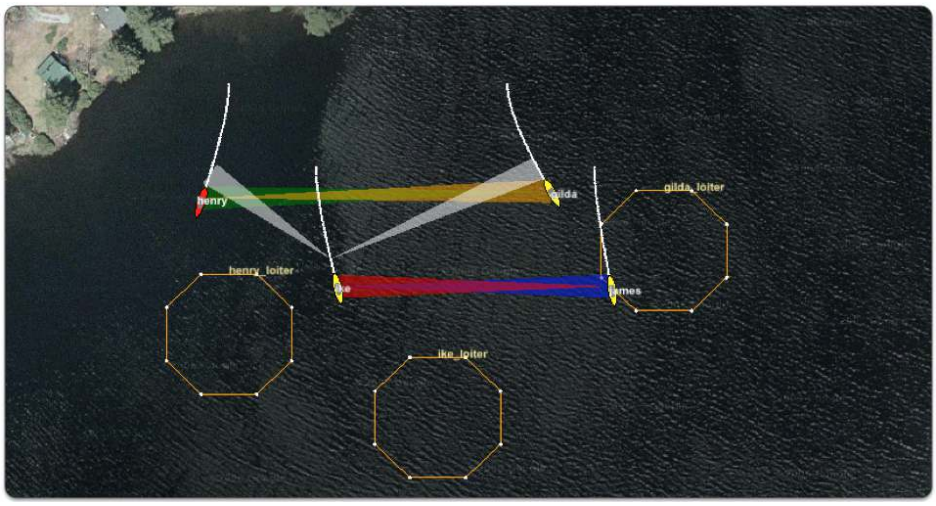

Figure 260: **Communication Pulses:** A visual artifact, a VIEW_COMMS_PULSE, is rendered between vehicles when either a node report or node message is generated. The white pulse indicates a node message, and the non-white pulses indicate a node report. The pulse widens from a point at the source vehicle to maximum width at the destination vehicle. A pulse will remain rendered in pMarineViewer for some number of seconds after initial post, unless it is replace by a new pulse with the same label.

The top two vehicles are also sharing node reports seen by the yellow and green pulses. The bottom left vehicle has also just sent a node message to the top two vehicles indicated by the two white pulse. Note the pair of white pulses were posted a few seconds prior to the present point in time since they point to vehicle positions just back in the vehicle history. The node reports are updated continuously, constantly replacing the pulse just previously posted.

The pulse colors are chosen automatically by uFldNodeComms. They may be toggled off in pMarineViewer with the '@' key. The comms pulse is conveyed in a posting to the variable VIEW_COMMS_PULSE. The format is shown with the following example.

```
VIEW_COMMS_PULSE = "sx=109.63,sy=-60.06,tx=30.96,ty=-60.22,beam_width=7,duration=10,
                    fill=0.35,label=JAMES2IKE,edge_color=green,fill_color=red,
                    time=2652414456.59,edge_size=1"
```

Comms pulses may be generated by other applications besides uFldNodeComms, and may be consumed and rendered by other applications besides pMarineViewer. The definition of the comms pulse object and the methods for serializing and de-serializing between object and string representation may be found in the lib_geometry library in the moos-ivp software tree.

95.6 Node Report Filtering and Sharing

As discussed above, a core input to uFldNodeComms are all the node reports arriving from each vehicle. Node reports are needed to:

- Send node reports back out to other vehicles, serving as proxy AIS system between vehicles
- Determine where vehicles are relative to each other, to decide whether inter-vehicle messages are sent from one vehicle to another.

The rate of node report arrivals may huge, especially when the shoreside is supporting a multi-vehicle mission of say 50 or more vehicles, and in simulations of say time warp 50 or higher. On each vehicle a node report is typically generated by pNodeReporter at 4Hz. Together this would amount to 10,000 node reports per second. The uFldNodeComms app can routinely handle this load without issue on any machine we have tested.

However, pMarineViewer also ingests this same stream of node reports, primarily for the purpose of updating and rendering vehicle trajectories. In a high time warp simulation resulting in a single vehicle generating a node report at 100-200 node reports per second, this rate is far more than what pMarineViewer needs to smoothly render a vehicle trajectory. Roughly 10-20 position updates per second is sufficient to give the human eye the feeling of a smoothly updated trajectory.

As an optional configuration, uFldNodeComms can be configured to re-post node reports under a different variable name, NODE_REPORT_UNC. The re-posting is generated at a rate that considers the prevailing time warp, and ensures a posting rate commensurate with what is needed for human visual updates, e.g. 10-20Hz. The idea is conveyed below in Figure 261.

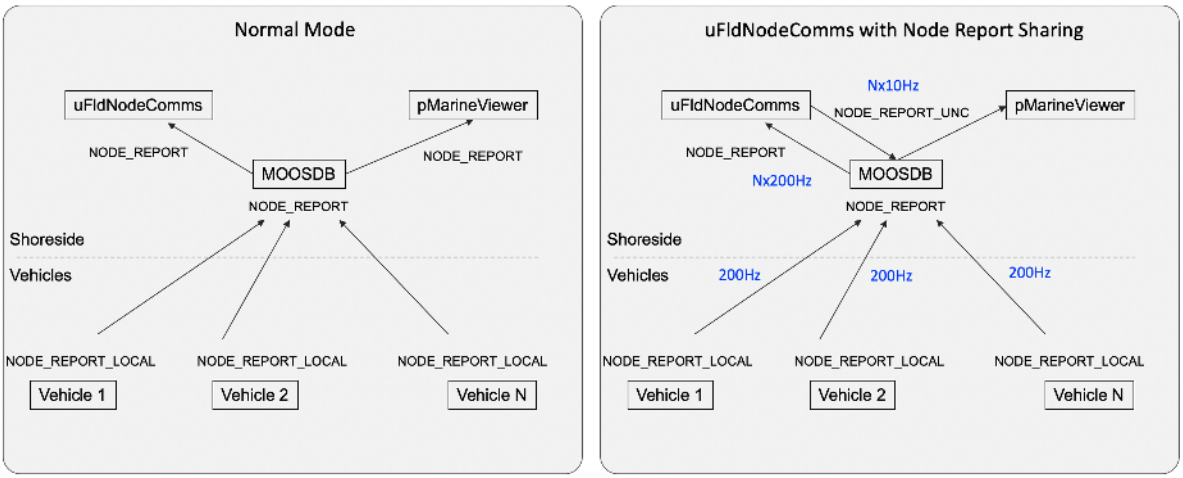

Figure 261: **Node Report Sharing:** With the optionally node report sharing configured, uFldNodeComms can act as a filter of node reports, re-posting node reports at a slower rate, in very high time warp situations.

An experienced MOOS user may note that pMarineViewer could be configured to simply register for NODE_REPORT at a slower rate, when the app registers for this variable. Of course the registration rate would have to also consider the prevailing time warp, but this is easy enough. This approach would be problematic however because using the MOOSDB to filter out say 9 of every 10 node reports may result in un-even filtering per vehicle. Since all vehicles publish to NODE_REPORT and not NODE_REPORT_ABE for example, the above approach may end up filtering out 99 of 100 reports from one vehicle and 4 of 5 reports from another, leading to jumpy rendering and false stale report warnings.

By shifting node report filtering to uFldNodeComms, under such extreme high contact, high time warp situations, uFldNodeComms shifts computational burden away from pMarineViewer. The viewer is probably the most computationally burdened app running in these kinds of missions and uFldNodeComms

is already processing the full set of node reports anyway as part of its basic job. Finally, most users are typically using a computer with multiple cores available, with uFldNodeComms and pMarineViewer running on different cores.

95.7 Terminal and AppCast Output

The uFldNodeComms application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 66 below. On line 2, the name of the local community (usually shoreside) is listed on the left. On the right, "0/0(1822) indicates there are no configuration or run warnings, and the current iteration of uFldNodeComms is 1822. Lines 4-13 show the configuration requested from the mission configuation file.

Lines 15-30 convey a summary of received and sent node reports from each vehicle. The number in parentheses at the end of lines 18-22 indicate the elapsed time since the last node report was received. The number in brackets indicates the rate of received node reports (total node reports received per vehicle over time since app startup). This number should roughly equal the AppTick for pNodeReporter running on each vehicle.

Listing 95.66: Example uFldNodeComms console and appcast output.

```
 1  ================================================================
 2  uFldNodeComms shoreside                                 0/0(1822)
 3  ================================================================
 4  Configuration:
 5         Comms Range: 1000
 6      Critical Range: 10
 7          Stale Drop: 5
 8         Stale Foget: 100
 9     Max Message Len: 1000
10     Min Message Int: 20
11     Min Report  Int: 60
12         Apply Group: false
13       Share Reports: true (0.1)
14
15  Node Report Summary
16  ====================================
17       Total Received: 9112      Elapsed      Rate
18                  ABE: 1828      (0.0)        [3.60]
19                  BEN: 1824      (0.0)        [3.59]
20                  CAL: 1820      (0.0)        [3.58]
21                  DEB: 1818      (0.0)        [3.58]
22                  EVE: 1822      (0.0)        [3.58]
23       ---------------
24           Total Sent: 90
25                  ABE: 27
26                  BEN: 18
27                  DEB: 18
28                  EVE: 27
29       ---------------
30       Stale Vehicles: 0
31
32  Node Message Summary
33  ====================================
```

```
34     Total Msgs Received: 3
35                      ABE: 3     (24.9)
36       ------------------
37              Total Sent: 3
38                      BEN: 3
39       ------------------
40      Total Blocked Msgs: 0
41                 Invalid: 0
42            Missing Info: 0
43           Stale Receiver: 0
44               Too Recent: 0
45             Msg Too Long: 0
46            Range Too Far: 0
47
48    ====================================================================
49    Most Recent Events (3):
50    ====================================================================
51    [96.22]: Msg rec'd: src_node=abe,dest_node=ben,var_name=UPDATE_LOITER,string_val=speed=2.4
52    [71.10]: Msg rec'd: src_node=abe,dest_node=ben,var_name=UPDATE_LOITER,string_val=speed=2.4
53    [56.46]: Msg rec'd: src_node=abe,dest_node=ben,var_name=UPDATE_LOITER,string_val=speed=2.4
```

The summary of node messages is shown in the second half of the report, in lines 32-46 in this case. The total messages received is shown on line 34, with a breakdown of where they have been received from in the following lines. Starting on line 37, a summary of sent node messages is given. First the total sent messages on line 37 and a breakdown of receivers in the following lines. A summary of blocked messages is given next, in this case in lines 40-46. The total number of blocked messages is given first, followed by the possible reasons for blocking in lines 41-46. Finally, the most recent messages are shown as events in the last lines of the report.

95.8 Configuration Parameters of uFldNodeComms

The following parameters are defined for uFldNodeComms.

Listing 95.67: Configuration Parameters for uFldNodeComms.

comms_range:	Max range outside which inter-vehicle node reports and node messages will not be sent. Legal values: any numerical value. The default is 100 meters. Section 95.2.1.
critical_range:	Range in meters within which inter-vehicle node reports will be shared even if group membership would otherwise disallow. Legal values: any numerical value. The default is 30 meters. Section 95.2.3.
debug:	If true, further debugging information is produced to the terminal output. Legal values: true, false. The default is false.
earange:	A parameter in the range of [1,10] for extending the range a vehicle may otherwise hear node reports from other vehicles. The default is 1. Section 95.3.2.
groups:	If true, inter-vehicle node reports are shared only if two vehicles are in the same group. May be overridden if the two vehicles are within the critical range. The vehicle group designation is typically declared in pNodeReporter. The default is false. Section 95.2.2.

ignore_group:	If specified, incoming node reports that match this group will be ignored. Introduced after Release 19.8.x.
min_msg_interval:	The number of seconds required between messaged sends for any one source vehicle. The default is 30 seconds. Section 95.4.2.
max_msg_length:	The total number of characters that may be sent in a string component of a node message. The default is 1000. Section 95.4.3.
min_rpt_interval:	The number of seconds required between report sends for any one source vehicle. The default is -1, meaning no dropping of reports. Section 95.2.6.
msg_groups:	If true, inter-vehicle node messages are shared only if the two vehicles are in the same group. The vehicle group designation is typically declared in pNodeReporter. The default is false. Section 95.2.2.
msg_color:	The color of inter-vehicle message comms pulses. The default is "white". Section 95.5.
msg_repeat_color:	The color of inter-vehicle message comms pulses when the message represents a repeat of a previously dropped message, under mediated message passing. The default is "light_green". Section 95.5.
stealth:	A parameter in the range [0,1] for reducing the range other vehicles may otherwise hear the node reports from a source vehicle. The default is 1. Section 95.3.1.
pulse_duration:	A duration, in seconds, that the comms or node pulse should be viewable. It is a field in VIEW_COMMS_PULSE message that is respected by at least pMarineViewer to render and slowly fade-out the rendering after this duration. The default is 10 seconds. Section 95.5.
shared_node_reports:	When enabled with true, node reports are re-posted to the shoreside community as NODE_REPORT_UNC. The default is false. This feature may also be more commonly enabled via in incoming message. See UNC_SHARED_NODE_REPORTS. Section 95.6.
stale_time:	Time in seconds after which a vehicle will not receive node reports or messages unless a node report has been received by that vehicle. The default is 5 seconds. Section 95.2.4.
verbose:	If true, status reports are displayed to the terminal during operation. The default is false.
view_node_rpt_pulses:	If true, comms pulses are rendered between vehicles whenever a node report successfully makes its way from one vehicle to another. The default is true. Section 95.5.

95.8.1 An Example MOOS Configuration Block

Listing 68 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uFldNodeComms --example or -e
```

Listing 95.68: Example configuration of the uFldNodeComms application.

```
1  ============================================================
2  uFldNodeComms Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = uFldNodeComms
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   comms_range    = 100       // default (in meters)
11   critical_range = 30        // default (in meters)
12   stale_time     = 5         // default (in seconds
13
14   max_msg_length   = 1000    // default (# of characters)
15   min_msg_interval = 30      // default (in seconds)
16   min_rpt_interval = -1      // default (in seconds)
17
18   verbose    = true          // default
19   groups     = false         // default
20   msg_groups = false         // default
21
22   stealth  = vname=alpha, stealth=0.8
23   earange  = vname=alpha, earange=4.5
24
25   shared_node_reports = true/false or any non-neg number
26
27   pulse_duration = 10;         // default (in seconds)
27   view_node_rpt_pulses = true  // default
28 }
```

95.9 Publications and Subscriptions for uFldNodeComms

The interface for uFldNodeComms, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldNodeComms --interface or -i
```

95.9.1 Variables Published by uFldNodeComms

The primary output of uFldNodeComms to the MOOSDB are the node reports and node messages out to the recipient vehicles, and visual artifacts to be used for rendering the inter-vehicle communications.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 95.7.
- NODE_MESSAGE_<VNAME>: Node messages destined to be sent to a destination vehicle <VNAME> indicated as the recipient in the node message.
- NODE_REPORT_<VNAME>: Node reports destined to be sent to a given vehicle <VNAME> about the vehicle named in the node report.

- NODE_REPORT_UNC: When node report sharing is enabled, a subset of all node reports will be re-posted to this variable. See Section 95.6.

- VIEW_COMMS_PULSE: A visual artifact for rendering the sending of a node report or node message between vehicles.

95.9.2 Variables Subscribed for by uFldNodeComms

The uFldNodeComms application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- NODE_MESSAGE: A node message from one vehicle to another.
- NODE_REPORT: A node report for a given vehicle from pNodeReporter.
- NODE_REPORT_LOCAL: Another name for a node report for a given vehicle from pNodeReporter.
- UNC_STEALTH: The extra stealth allowed a given vehicle to "hide" node reports to others.
- UNC_EARANGE: The extra range allowed a given vehicle to "hear" node reports of others.
- UNC_FULL_REPORT_REQ: For a given vehicle name, the next node report from each of the other vehicles will be received, regardless of all other criteria such as range, frequency, group. This only applies for the very next report only, then all criteria reverts. This is a convenient debugging tool.
- UNC_SHARED_NODE_REPORTS: If enabled, set to true, uFldNodeComms will re-post node reports from all vehicles to the variable NODE_REPORT_UNC but will clamp the publication rate, regardless of time warp, to a rate suitable for visual consumption, typically no more than 10 updates per second per vehicle. This feature is used in coordination with pMarineViewer to improve performance in high time warp missions with a high number of contacts. This variable is typically published by pMarineViewer requesting this service from uFldNodeComms. Section 95.6.

- UNC_VIEW_NODE_RPT_PULSES: A way for external apps to tell uFldNodeComms to turn off or on the rendering of comms pulses whenever a node report is sent from one vehicle to another. Section 95.5.

95.9.3 Command Line Usage of uFldNodeComms

The uFldNodeComms application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uFldNodeComms --help or -h
```

Listing 95.69: Command line usage for the uFldNodeComms tool.

```
1  ============================================================
2  Usage: uFldNodeComms file.moos [OPTIONS]
3  ============================================================
```

```
 4
 5  Options:
 6    --alias=<ProcessName>
 7        Launch uFldNodeComms with the given process
 8        name rather than uFldNodeComms.
 9    --example, -e
10        Display example MOOS configuration block.
11    --help, -h
12        Display this help message.
13    --interface, -i
14        Display MOOS publications and subscriptions.
15    --version,-v
16        Display the release version of uFldNodeComms.
17
18  Note: If argv[2] does not otherwise match a known option,
19        then it will be interpreted as a run alias. This is
20        to support pAntler launching conventions.
```

96 uFldMessageHandler: Handling Incoming Node Messages

96.1 Overview

The uFldMessageHandler application handles incoming messages from a remote MOOSDB. In MOOS, applications "talk" to each other through the normal publish-subscribe means of a common MOOSDB. Distinct robots or vehicles can also share information between vehicles and MOOSDBs using pShare. But often we want to simulate inter-vehicle messaging where messages are sometimes dropped, and messages are subject to inter-vehicle range limitations, band-width limitations, and limitations on message frequency. To simulate this we use the uFldNodeComms and uFldMessageHandler apps in concert. A message meant for another vehicle is packaged up in a format containing the message itself and a bit of routing information. The receiving vehicle, running uFldMessageHandler, unpacks the message and injects the message to the local vehicle by posting to the local MOOSDB. The job of routing and applying communications limitations is done by uFldNodeComms and is not the focus here. The sole job of uFldMessageHandler is to do the message unpacking and posting of information, while also keeping some stats for debugging if needed.

96.2 Typical Application Topology

In the uField Toolbox typical arrangement, messages arrive from a shoreside MOOS community running uFldNodeComms and pShare as shown below in Figure 262.

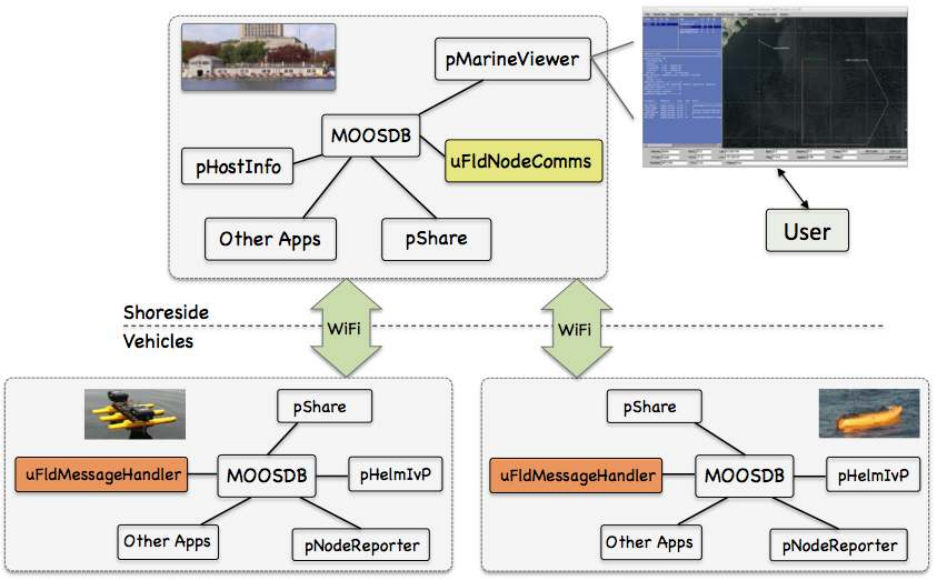

Figure 262: **Typical uFldMessageHandler Topology:** A vehicle (node) sends a message to another vehicle by wrapping the message content and addressee information in a single string sent to the shoreside. On the shoreside, the uFldNodeComms application redirects the message to the appropriate vehicle(s). The message is received on the vehicle by the uFldMessageHandler application which parses the MOOS variable and the variable value from the string and posts the variable-value pair to the local MOOSDB.

96.3 Inter-vehicle Messaging Sequence of Events

The functionality of uFldMessageHandler may be paraphrased:

- A source vehicle *alpha* wishes to send a message to vehicle *bravo* of the form SPEED=2.5.
- A local message is posted on vehicle *alpha* of the form:

```
NODE_MESSAGE_LOCAL = src_node=alpha,dest_node=bravo,var_name=SPEED,double_val=2.5
```

- The above message is shared from *alpha* to the shoreside community using pShare.
- The message is received in the shoreside community as the variable NODE_MESSAGE and handled by uFldNodeComms and republished as NODE_MESSAGE_BRAVO.
- The message is then shared out to *bravo* using pShare arriving in vehicle *bravo* as NODE_MESSAGE.
- On vehicle *bravo*, the NODE_MESSAGE is handled by uFldMessageHandler. The source variable and value are parsed and a post to the local MOOSDB on *bravo* is made, SPEED=2.5
- A scope on the MOOSDB on *bravo* would show the source of the SPEED=2.5 posting to be "uFldMessageHandler", and the auxiliary source would show "alpha"

96.4 Building an Outgoing Node Message

To construct an outgoing node message, there are two options. A message can be created through normal string construction, for example:

```
string m_hostname;      // previously set name of ownship
string m_dest_name;     // previously set name of vehicle to communicate
string m_moos_varname;  // previously set name of MOOS variable to send
string m_msg_contents;  // previously set contents of message;

string msg;
msg += "src_node="   + m_hostname;
msg += ",dest_node=" + m_dest_name;
msg += ",var_name="  + m_moos_varname;
msg += ",string_val=" + m_msg_contents;

Notify("NODE_MESSAGE_LOCAL", msg);
```

The above works, but may be prone to typos and may not be very ”future-proof” if the format for inter-vehicle messaging changes. Another way to accomplish this is to use the NodeMessage class which has the serializing and de-serializing steps implemented:

```
#include "NodeMessage.h"  // In the lib_ufield library

string m_hostname;     // previously set name of ownship
string m_dest_name;    // previously set name of vehicle to communicate
string m_moos_varname; // previously set name of MOOS variable to send
string m_msg_contents; // previously set contents of message;

NodeMessage node_message;

node_message.setSourceNode(m_hostname);
node_message.setDestNode(m_dest_name);
node_message.setVarName(m_moos_varname);
node_message.setStringVal(m_msg_contents);

string msg = node_message.getSpec();

Notify("NODE_MESSAGE_LOCAL", msg);
```

In the opposite direction, creating a `NodeMessage` instance from a string is done with:

```
#include "NodeMessageUtils.h"  // In the lib_ufield library

string node_message_str;       // previously set string containing a message

NodeMessage node_message = string2NodeMessage(node_message_str);
```

Note that the `NodeMessage` class is part of the `lib_ufield` library. The `uFldMessageHandler` app links to this library, but if the above class is being used in a new app, to create a message, the `lib_ufield` library needs to be linked as part of the build process of the new app.

96.5 Building an Outgoing Node Message - A Note of Caution

Note the structure of a node message, such as the one below, uses commas as component delimiters:

```
NODE_MESSAGE_LOCAL = src_node=alpha,dest_node=bravo,var_name=SPEED,double_val=2.5
```

Some care must be taken if the contents of the outgoing message is a string also containing commas, such as:

```
NODE_MESSAGE_LOCAL = src_node=alpha,dest_node=bravo,var_name=INFO,string_val=good,bad,ugly
```

In this case the string contents should be wrapped in double quotes:

```
NODE_MESSAGE_LOCAL = src_node=alpha,dest_node=bravo,var_name=INFO,string_val="good,bad,ugly"
```

If the message is built using the `NodeMessage` class as in the second example in Section 96.4 above, the double quotes are added automatically in the `setStringVal()` function if the argument is not quoted and it contains a comma.

96.6 Event Flags

With the optional msg_flag and bad_msg_flag, the user may configure events (postings) to be made whenever an incoming message results in a successful posting, and whenever an incoming message does not result in a posting.

The msg_flag is configured with variable-value pair, for example:

```
msg_flag = GOOD_POSTING=true
```

Likewise, the bad_msg_flag is configured with variable-value pair, for example:

```
bad_msg_flag = FAILED_POSTING=true
```

The value component of any posting may contain one or more of several supported macros. A macro is expanded at the time of posting. Supported macros include:

- $[CTR]: The total of all received messages, regardless of whether a message resulted in a posting.
- $[GOOD_CTR]: The total of all received messages that resulted in a posting.
- $[BAD_CTR]: The total of all received messages that did not result in a posting.

Of course multiple postings may be configured. For example:

```
msg_flag = GOOD_POSTING=true
msg_flag = POSTING_COUNT=$[CTR]
```

96.7 Configuration Parameters of uFldMessageHandler

The following parameters are defined for uFldMessageHandler.

Listing 96.70: Configuration parameters for uFldMessageHandler.

appcast_trunc_msg:	Number of characters allowed in the appcast report for each line reporting a successful message. For example, Lines 19-24 in Listing 73. The default is 75. Setting it to zero means no truncating will be applied.
strict_addressing:	If true, only messages with a destination specified by `dest_node`, matching the local community name are processed. Other messages with a destination specified by a group designation are ignored. The default is false.
msg_flag:	A variable-value pair posted upon receipt of a valid, un-rejected incoming message and posting to the MOOSDB. Section 96.6.
bad_msg_flag:	A variable-value pair posted upon receipt of a valid, but rejected incoming message. Section 96.6.
aux_info:	Adjust the content of the source auxilliary field. By default this is set to `"node"`. If set to `"node+app"`, the source auxialliary field of posted messages will contain both the sending node and the sending app.

96.7.1 An Example MOOS Configuration Block

Listing 71 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uFldMessageHandler --example or -e
```

Listing 96.71: Example configuration of the uFldMessageHandler *application.*

```
1  ============================================================
2  uFldMessageHandler Example MOOS Configuration
3  ============================================================
4
5  ProcessConfig = uFldMessageHandler
6  {
7    AppTick   = 4
8    CommsTick = 4
9
10   strict_addressing = false   // the default
11   appcast_trunc_msg = 75      // default: the number of chars per
12                               // line in the appcasting output
13
14   msg_flag     = RETURN=true
15   bad_msg_flag = TOTAL_BAD=$[BAD_CTR]
16
17   aux_info = node+app  // {node or node+app} Default is node
18
19   app_logging  = true  // {true or file} By default disabled
20 }
```

96.7.2 Variables Published

The primary output of uFldMessageHandler to the MOOSDB are the messages posted by parsing incoming NODE_MESSAGE postings. A summary is also posted periodically to recap message handling totals.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 96.9.
- UMH_SUMMARY_MSGS: A summary of total messages, valid messages and rejected messages handled thus far. An example: UMH_SUMMARY_MSGS=total=3,valid=3,rejected=0

Further publications will be made if the app is configure with any msg_flag or bad_msg_flag flags. For example if configured with msg_flag=POSTING=$[CTR], the variable POSTING will be posted each time uFldMessageHhandler posts an incoming message. The value of POSTING will be the total number of messages posted by uFldMessageHandler thus far.

96.7.3 Variables Subscriptions

The uFldMessageHandler application subscribes to the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- NODE_MESSAGE: Incoming node messages.

96.8 Command Line Usage of uFldMessageHandler

The uFldMessageHandler application is typically launched with pAntler, along with a group of other vehicle modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uFldMessageHandler --help or -h
```

Listing 96.72: Command line usage for the uFldMessageHandler tool.

```
1  ============================================================
2  Usage: uFldMessageHandler file.moos [OPTIONS]
3  ============================================================
4
5  SYNOPSIS:
6  ------------------------------------
7    The uFldMessageHandler tool is used for handling incoming
8    messages from other nodes. The message is a string that
9    contains the source and destination of the message as well as
0    the MOOS variable and value. This app simply posts to the
11   local MOOSDB the variable-value pair contents of the message.
12
13 Options:
14   --alias=<ProcessName>
15       Launch uFldMessageHandler with the given process name
16       rather than uFldMessageHandler.
17   --example, -e
18       Display example MOOS configuration block.
19   --help, -h
10       Display this help message.
21   --interface, -i
22       Display MOOS publications and subscriptions.
23   --version,-v
24       Display the release version of uFldMessageHandler.
25   --web,-w
26       Open browser to:
27       https://oceanai.mit.edu/ivpman/apps/uFldMessageHandler
28
29 Note: If argv[2] does not otherwise match a known option,
30       then it will be interpreted as a run alias. This is
31       to support pAntler launching conventions.
```

96.9 Terminal and AppCast Output

The uFldMessageHandler application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 73 below. On line 2, the name of the local community or vehicle name is listed on the left. On the right, "0/0(841) indicates there are no configuration or run warnings, and the current iteration of uFldMessageHandler is 841. In lines 4-9,

general tallies are shown of received, invalid, and rejected messages. In lines 11-15, the tallies for received messages sorted by source vehicle are shown. The variable-value columns reflect only the last received message.

Listing 96.73: Example appcast and terminal output of `uFldMessageHandler`*.*

```
 1  ================================================================
 2  uFldMessageHandler gilda                                 0/0(841)
 3  ================================================================
 4  Overall Totals Summary
 5  ======================================
 6     Total Received Valid: 5
 7                   Invalid: 0
 8                  Rejected: 0
 9      Time since last Msg: 101.3
10
11  Per Source Node Summary
12  ======================================
13  Source  Total  Elapsed  Variable  Value
14  ------  -----  -------  --------  -----
15  henry   5      101.3    RETURN    true
16
17  Last Few Messages: (oldest to newest)
18  ======================================
19  Valid Mgs:
20    src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
21    src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
22    src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
23    src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
24    src_node=henry,dest_node=gilda,var_name=RETURN,string_val=true
25  Invalid Mgs:
26    NONE
27  Rejected Mgs:
28    NONE
```

The information group starting on line 17 shows the last five received valid, invalid and rejected messages. Note that a rejected message may be rejected for being invalid, or if the destination field doesn't match, or if strict addressing is enabled and there is not a precise destination field match.

97 uFldCollisionDetect: Detecting Collisions

97.1 Overview

The uFldCollisionDetect application is run on the shoreside and monitors pairs of vehicles for encounters that come within a certain range. The closest point of approach (CPA) is noted when the range between two vehicles transitions from closing to opening. Depending on the CPA value, one of three events may be declared, either an *encounter*, a *near miss*, or a *collision*, depending on user configured range parameters.

97.2 Using uFldCollisionDetect

97.2.1 Setting the Range Thresholds for Events

There are three range threshold parameters that may be set. The first, encounter_range is the CPA range beyond which two vehicles are considered to be too far away to be regarded as having had an encounter. Outside this range it's a non-event.

The second parameter, near_miss_range, determines a CPA range within which an encounter is considered to be a near miss. Encounters even closer, with the range specified by collision_range, are categorized as collisions.

The default values are:

- encounter_range = 20
- near_miss_range = 6
- collision_range = 3

The following relationship between parameters will be enforced:

encounter_range ¿= near_miss_range ¿= collision_range

If this ordering is not respected by the configuration parameters, then the following happens upon startup: (1) If the near_miss_range is smaller than the collision_range, the former is adjusted upward to the latter. (2) If the encounter_range is less than the near_miss_range, then the former is adjusted upward to the latter. (3) A configuration warning is generated and shown in the appcasting output.

Upon each encounter less than or equal to the encounter_range, an encounter counter is internally incremented. Likewise for near misses and collisions. A collision encounter will not however also increment the near miss counter. These counters are maintained globally for all vehicles, as seen in lines 17-19 in the example appcasting output in Listing 76. They are also maintained *per vehicle*, as seen in lines 21-29 in Listing 76.

97.2.2 Setting Flags to be Posted Upon Events

Flags may be configured to be posted upon each event type - collision, near-miss or encounter. These flags are simply MOOS variable and value pairs like the flags in many other MOOS applications and helm behaviors. There are also macros available that can be filled in a run-time.

The following are the available macros:

- `$V1`: The name of vehicle 1.
- `$V2`: The name of vehicle 2.
- `$UP_V1`: The upper case name of vehicle 1.
- `$UP_V2`: The upper case name of vehicle 2.
- `$IDX`: The number of total encounters
- `$CPA`: The CPA range of the encounter

Note these macros are available for expansion in the contents of the published MOOS message. If the `$CPA` or `$IDX` macros are used in isolation in the message, the message will post as a double, not a string. All macros (except `$CPA`) may also be used in the naming of the MOOS variable. The following are all valid configurations:

```
near_miss_flag  = NEAR_MISS_$UP_V1 = $CPA
encounter_flag  = WARNING = Something happened between $V1 and $V2
collision_flag  = CRITICAL = collision between $UP_V1 and $UP_V2
```

97.2.3 Applying a Logic Condition

Logic conditions may also be used for disabling/enabling the reporting activities of `uFldCollisionDectect`. Variables involved in the conditions are automatically subscribed for by `uFldCollisionDectect`. For example

```
  condition = COLLISION_DETECT=true
  condition = DEPLOY_ALL=true
```

If multiple conditions are specified, they must *all* be true for application to actively report events.

97.2.4 Configuring Visual Range Pulses to be Generated

Range pulses can be configured to show the user a visual signal that an interaction has occurred. The range pulse can be configured to change both its range and duration using configuration parameters.

Figure 263: **Using Range Pulse with uFldCollisionDetect:** A range pulse is displayed in pMarineViewer with each detected interaction.

97.2.5 Ignoring Collisions Between Certain Contacts

In certain missions, it may be desirable to ignore encounters between certain contacts. Perhaps we want to focus on a particular vehicle. Or perhaps there are multiple contacts without collision avoidance enabled (hopefully only in simulation). In any event, there are two configuration parameters for `uFldCollisionDetect` that allow certain encounters to be disregarded. The `ignore_group` parameter and the `reject_group` parameter. Both parameters name one or more vehicle *groups*.

```
ignore_group = commercial
reject_group = sailing,rowing
```

For a vehicle in one of the *ignore* groups, encounters between this vehicle and any other vehicle also in an ignore group, the encounter will be disregarded. However, if the other vehicle is not in an ignore or reject group, the encounter will be included in the analysis. For a vehicle in one the *reject* groups, incoming node reports for this vehicle are rejected on arrival. Any encounter with this vehicle and any other vehicle will not be included in the analysis.

An example is given in Figure 97.2.5 below:

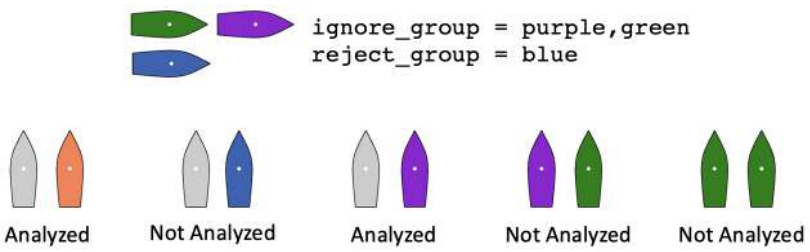

Figure 264: **Filtering out encounters base on group:** An encounter is only analyzed if at least one of vehicles is not in an ignore group *and* both vehicles are not members of a reject group.

The group name for a vehicle comes from the incoming node report, and the group name is set in the configuration of pNodeReporter for that vehicle.

97.2.6 Posting Range Status Messages

On each iteration of uFldCollisionDetect, the range between all pairs of vehicles is noted. For that instant in time, the minimum range of all ranges is also noted, and published in the variable:

UCD_CLOSEST_RANGE

This variable is only published when configured to do so:

post_closest_range=true

Additionally, the minimum range ever noted between any two vehicles is published in the variable:

UCD_CLOSEST_RANGE_EVER

97.3 Configuration Parameters for uFldCollisionDetect

The uFldCollisionDetect application may be configured with a configuration block within a MOOS mission file, typically with a `.moos` file suffix. The following parameters are defined for uFldCollisionDetect.

Listing 97.74: Configuration Parameters for uFldCollisionDetect.

collision_flag:	A MOOS variable and value posted upon the detection of a collision. Section 97.2.2.
collision_range:	The inter-vehicle range, within which the CPA is regarded as a collision. Section 97.2.1.
condition:	A logic condition which if not evaluated to be true, the monitoring of collisions will not be conducted. Section 97.2.3.
encounter_flag:	A MOOS variable and value posted upon the detection of an encounter. Section 97.2.2.
encounter_range:	The inter-vehicle range, within which the CPA is considered to be an encounter. Section 97.2.1.

ignore_group:	A comma-separated lists of group names. Encounters for vehicles in one of these groups will be ignored, if the other vehicle is also in one of these groups. Section 97.2.5.
near_miss_flag:	A MOOS variable and value posted upon the detection of a near miss. Section 97.2.2.
near_miss_range:	The inter-vehicle range, within which the CPA is regarded as a near miss. Section 97.2.1.
pulse_duration:	Sets the duration of the visual range pulse posted. Default is 10 seconds. 97.2.4.
pulse_range:	Sets the range of the visual range pulse posted. Default is 20 meters. 97.2.4.
pulse_render:	Determines if a visual range pulse will be posted on encounters. Default is true. Section 97.2.4.
reject_group:	A comma-separated lists of group names. Encounters for vehicles in one of these groups will be completely disregarded, regardless of the group of the other vehicle. Section 97.2.5.
report_all_encounters:	If true, all encounters will be published in the UCD_REPORT, not just near misses or collisions. The default is false.

97.3.1 An Example MOOS Configuration Block

An example MOOS configuration block is provided in Listing 75 below. This can also be obtained from a terminal window with:

```
$ uFldCollisionDetect --example or -e
```

Listing 97.75: Example configuration of the uFldCollisionDetect application.

```
================================================================
uFldCollisionDetect Example MOOS Configuration
================================================================

ProcessConfig = uFldCollisionDetect
{
  AppTick   = 4
  CommsTick = 4

  condition = DEPLOY_ALL = true

  encounter_flag = ENCOUNTER = $CPA
  collision_flag = COLLISION = $CPA
  near_miss_flag = NEAR_MISS = vname1=$V1,vname2=$V2,cpa=$CPA

  encounter_range = 10         // the default in meters
  near_miss_range = 6          // the default in meters
  collision_range = 3          // the default in meters
```

```
  ignore_group = alpha
  reject_group = bravo

  pulse_render   = true         // default true
  pulse_range    = 20           // default is 20 meters
  pulse_duration = 10           // default is 10 seconds

  report_all_encounters = true  // default is false
}
```

97.4 Publications and Subscriptions for uFldCollisionDetect

The interface for uFldCollisionDetect, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldCollisionDetect --interface or -i
```

97.4.1 Variables Published by uFldCollisionDetect

The user may configure any number of postings (flags) to be generated upon an encounter, near miss or collision.

- APPCAST: Contains an appcast report identical to the terminal output. Reports are posted only in response to an appcast request messages.
- COLLISION_DETECT_PARAMS: The values for encounter_range, near_miss_range, and collision_range are posted upon startup to other applications may use this information. For example: COLLISION_DETECT_PARAMS="collision_range=3,near_miss_range=6,encounter_range=20"
- ENCOUNTER_TOTAL: Total number of encounters, within encounter_range, regardless of CPA.
- UCD_REPORT: A report generated on each encounter showing which vehicles where involved and the CPA value. For example: UCD_REPORT="cpa=11.2,vname1=henry,vname2=gus,rank=near_miss"
- UCD_CLOSEST_RANGE: If the post_closest_range parameter is set to true, this variable is published. The value is the minimum range between all currently monitored vehicles.
- VIEW_RANGE_PULSE: A visual artifact generated upon an encounter centered at the mid-point between the two vehicles.

97.4.2 Variables Subscribed for by uFldCollisionDetect

The uFldCollisionDetect application will subscribe for the following variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration.
- NODE_REPORT: A report on a vehicle location and status.

Additionally, any variable used in a logic condition will also be subscribed for. See Section 97.2.3 for more on the use of logic conditions.

97.4.3 Command Line Usage of uFldCollisionDetect

The uFldCollisionDetect application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. To see command-line options enter the following from the command-line:

```
$ uFldCollisionDetect --help or -h
```

This will show the output shown below.

```
Usage: uFldCollisionDetect file.moos [OPTIONS]

Options:
  --alias=<ProcessName>
      Launch uFldCollisionDetect with the given process
      name rather than uFldCollisionDetect.
  --example, -e
      Display example MOOS configuration block
  --help, -h
      Display this help message.
  --interface, -i
     Display MOOS publications and subscriptions.
  --version,-v
      Display the release version of uFldCollisionDetect.
```

97.5 Terminal and AppCast Output

The uFldCollisionDetect application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 76 below. On line 2, the name of the local community, typically the shoreside community, is listed on the left. On the right, "0/0(2769) indicates there are no configuration or run warnings, and the current iteration of uFldCollisionDetect is 2769. Lines 4-11 show the how the application was configured at launch time.

Lines 13-19 convey the the overall application state, whether or not the application is actively assessing encounters (line 16), and the total number of encounters, near missions and collisions (lines 17-19).

Listing 97.76: Example terminal or appcast output for uFldCollisionDetect.

```
1  ============================================================================
2  uFldCollisionDetect shoreside                                  0/0(2769)
3  ============================================================================
4  Configuration:
5  ============================================
6          encounter_dist: 30.00
7          collision_dist: 8.00
8          near_miss_dist: 12.00
9     range_pulse_render: true
10   range_pulse_duration: 30.00
```

```
11      range_pulse_range: 20.00
12
13  ============================================
14  State Overall:
15  ============================================
16               Active: true
17     Total Encounters: 19
18    Total Near Misses: 0
19     Total Collisions: 0
20
21  ============================================
22  State By Vehicle:
23  ============================================
24  Vehicle  Encounters  Near Misses  Collisions
25  -------  ----------  -----------  ----------
26  abe      9           0            0
27  ben      9           0            0
28  cal      10          0            0
29  deb      10          0            0
30
31  ========================================================================
32  Most Recent Events (8):
33  ========================================================================
34  [806.84]: ben::deb, cpa=22.79
35  [783.85]: abe::ben, cpa=18.26
36  [757.90]: cal::deb, cpa=29.16
37  [638.36]: cal::deb, cpa=27.73
38  [629.30]: ben::deb, cpa=16.85
39  [611.36]: ben::cal, cpa=28.97
40  [583.23]: ben::deb, cpa=19.33
41  [578.27]: abe::ben, cpa=17.02
```

The state is further broken down by vehicle in lines 21-29. The total number of encounters, near misses and collisions per vehicle is shown. In the final portion of the the appcast output are the 8 most recent events. Each encounter is an event, and the cpa distance for is also shown.

References