Chapter 2
Autonomy for Unmanned Marine Vehicles with MOOS-IvP

Michael R. Benjamin, Henrik Schmidt, Paul M. Newman, and John J. Leonard

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

2.1 Introduction

The growing desire for autonomy in unmanned marine systems is driven by several trends, including increased complexity in mission objectives and duration, increased capability in on-board sensor processing and computing power, and an increase in the number of users and owners of unmanned vehicles. The MOOS-IvP project is an open-source project designed and developed in this context. It is an implementation of an autonomous helm and substantial support applications that aims to provide a capable autonomy system out of the box. It also has an architecture, software policy, documentation, and support network that allows this and the next generation

M.R. Benjamin (✉) • H. Schmidt • J.J. Leonard
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA
e-mail: benjamin@mit.edu; schmidt@mit.edu; leonard@mit.edu

P.M. Newman
Department of Engineering Science, University of Oxford, Oxford, UK
e-mail: pnewman@robots.ox.ac.uk

M.L. Seto (ed.), *Marine Robot Autonomy*, DOI 10.1007/978-1-4614-5659-9_2,
© Springer Science+Business Media New York 2013

of scientists, with newer vehicles and mission ambitions, to be nimble to build innovative autonomy algorithms to augment an existing set of capabilities. This chapter describes the MOOS-IvP autonomy architecture and software structure.

MOOS-IvP is comprised of two open-source software projects. The mission oriented operating suite (MOOS) is a product of the Mobile Robotics Group at the University of Oxford and provides middleware capabilities in a publish-subscribe architecture as well as several applications ubiquitous in unmanned marine robotic and land robotic applications using MOOS. Additional MOOS applications, including the IvP Helm, are available in the MOOS-IvP project. IvP stands for interval programming and refers to the multi-objective optimization method used by the IvP Helm for arbitrating between competing behaviors in a behavior-based architecture.

The MOOS-IvP software is available on the web via anonymous read-only access [5]. It consists of more than 130,000 lines of C++, comprising about 30 distinct applications and over a dozen vehicle behaviors. It represents about 25 work years of effort or more from individual contributors. Autonomy configurations and missions in this environment have been tested in several thousands of hours of simulation and several hundred hours of in-water experiments on platforms including the Bluefin 21-inch UUV, the Bluefin 9-inch UUV, the Hydroid REMUS-100 and REMUS-600 UUVs, the Ocean Server Iver2 UUV, the Ocean Explorer 21-inch UUV, autonomous kayaks from Robotic Marine Systems and SARA Inc, the Kingfisher USV from Clearpath Robotics, and two USVs at the NATO Undersea Research Centre in La Spezia Italy, an unmanned RHIB from H-Scientific, and the USV-2600 from SeaRobotics.

2.1.1 Trends in Unmanned Marine Vehicles Relating to Autonomy

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

Advances in sensor technologies include greater capabilities, at lower cost, lower size, and lower power consumption. The same is true for the on-board computing components needed to process sensor data. Increasingly underwater

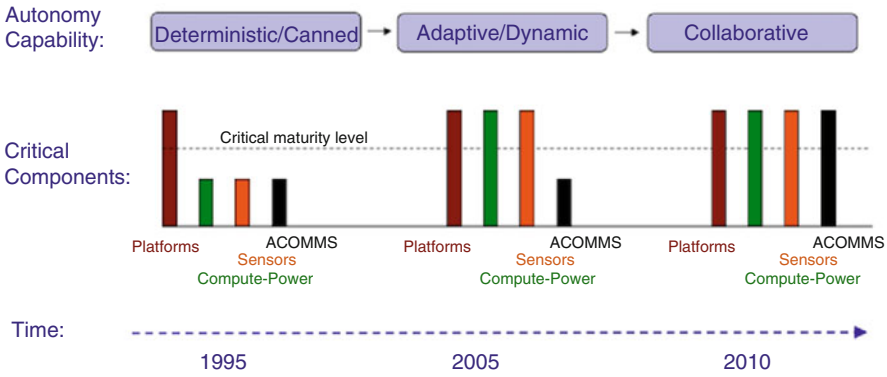

Fig. 2.1 Technologies and autonomy: A rough timeline and relationship between UUV autonomy and other critical UUV technologies. Critical components include (**a**) the platform itself in terms of reliability, cost, and endurance, (**b**) on-board computing power and sensor processing, (**c**) on-board sensors in terms of resolution, size, and cost, and (**d**) acoustic communications (ACOMMS). Each of these maturing technology trends affects what is possible and desired from the on-board autonomy system. The corresponding trend in autonomy is from deterministic vehicles acting independently, toward adaptive vehicles acting in collaboration

vehicles are able to see, hear, and localize objects and other vehicles in their environment and quickly analyze an array of qualities in water samples taken while underway. Likewise, the available mission duration at depth has grown longer due to improvements in inertial navigation systems, which have become cheaper, smaller, and more accurate and due to improvements in platform battery life. Each of these trends has contributed to making a UUV owner less satisfied with simply collecting the data and analyzing the results in a post-mission analysis phase. The notion of *adaptive autonomy* reflects the goal of using available in-stride information and analysis to the advantage of the mission objectives.

The chart in Fig. 2.1 conveys a rough timeline and relationship between the evolution of UUV autonomy capabilities and the evolution of other critical UUV technologies. The notion of *adaptive* in adaptive autonomy is a sliding scale and refers to the ability to allow increasing degrees of sensor information to affect in-stride autonomy decisions.

The notion of *collaboration* in collaborative autonomy may be viewed as a sliding scale as well. At one end of the spectrum are vehicles deployed alongside each other, executing a mission independently but each contributing to a joint mission. In this case, the collaboration occurs in the pre-deployment mission planning process. When at least periodic communication between deployed vehicles is feasible, real-time collaboration is possible, especially when each vehicle is able to adapt components of its mission to both its sensed environment *and* incoming communications from other vehicles. Advances in underwater acoustic communications (ACOMMS) in terms of reliability, range, flexibility in defining message sets, and bandwidth have enabled the development of adaptive, collaborative autonomy.

This trend also occurs in the context of declining cost and size of commercially available UUVs, making it possible for smaller organizations to operate several vehicles.

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

2.1.2 The Backseat Driver Design

The main idea in the backseat driver paradigm is the separation of vehicle *control* and *autonomy*. The control system runs on a platform's main vehicle computer and the autonomy system runs on a separate payload computer. A primary benefit is the decoupling of the platform autonomy from the vehicle hardware. The manufacturer provides a navigation and control system streaming vehicle position and trajectory information to the payload computer. In return, the main vehicle computer accepts a stream of autonomy decisions such as heading, speed, and depth. Exactly how the vehicle navigates and implements control is largely unspecified to the autonomy system running in the payload. The relationship is shown in Fig. 2.2.

The payload autonomy system consists of distinct processes communicating through a publish-subscribe database called the MOOSDB (mission oriented operating suite—database). One 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 "autonomy system," since MOOS itself is middleware, and actual autonomous decision making, sensor processing, contact management, etc., are implemented as individual MOOS processes.

2.1.3 The Publish-Subscribe Middleware Design 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 (Fig. 2.3). 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

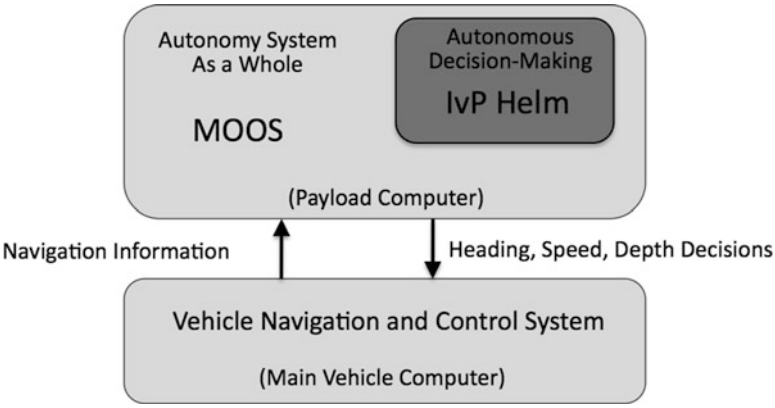

Fig. 2.2 The backseat driver paradigm: The key idea is the separation of vehicle autonomy from vehicle control. The autonomy system provides heading, speed, and depth commands to the vehicle control system. The vehicle control system executes the control and passes navigation information, e.g., position, 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

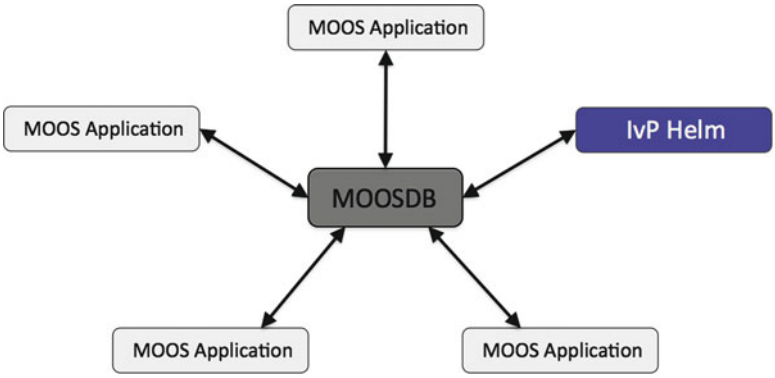

Fig. 2.3 A MOOS community: is a collection of MOOS applications typically running on a single machine each with a separate process ID. Each process communicates through a single MOOS database process (the MOOSDB) in a publish-subscribe manner. Each process may be executing its inner loop at a frequency independent from one another and set by the user. Processes may be all run on the same computer or distributed across a network

the values are either string or numerical values such as (STATE, "DEPLOY"), or (NAVSPEED, 2.2), or arbitrarily large binary packets. Allowing simple message types reduces the compile dependencies between modules and facilitates debugging since all messages are human readable.

The key idea with respect to facilitating code reuse is that applications are largely independent, defined only by their interface, and any application is easily replaceable with an improved version with a matching interface. 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. 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 code-base constant between MOOS developers enables the plug-and-play nature of applications.

2.1.4 The Behavior-Based Control Design and IvP Helm

The IvP Helm runs as a single MOOS application using a behavior-based architecture for implementing autonomy. Behaviors are distinct modules describable as self-contained mini expert systems dedicated to a particular aspect of overall vehicle autonomy. The helm implementation and each behavior implementation expose an interface for configuration by the user for a particular set of missions. This configuration often contains particulars such as a certain set of waypoints, search area, and vehicle speed. It also contains a specification of mission modes that determine which behaviors are active under 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 (Fig. 2.4).

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

The concept of a behavior-based architecture is often attributed to [7]. Various solutions to the problem 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 such that the highest priority behavior locks out all others as in the subsumption architecture in [7]. Another approach, referred to as the potential fields, or vector summation approach (see [1, 10]) regards the average action between multiple behaviors to be a good compromise. These action-selection approaches have been used effectively on a variety of platforms, including indoor robots, e.g., [1, 2, 13, 14], land vehicles, e.g., [15], and marine vehicles, e.g., [6, 8, 11, 16, 17]. However, action selection via the identification of a single highest priority behavior and via vector summation have well-known shortcomings later described in [13, 14] and [15] 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.

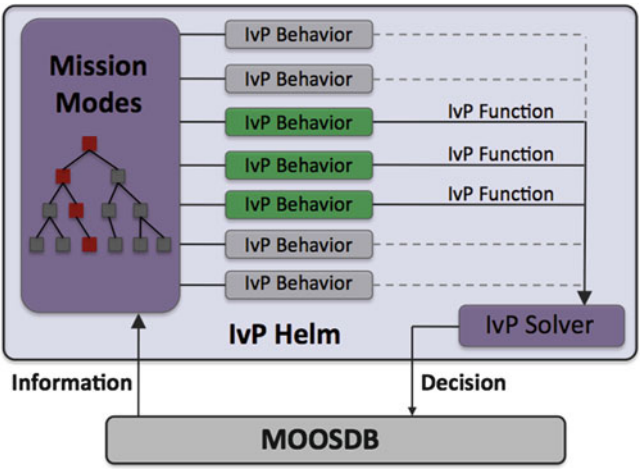

Fig. 2.4 The IvP Helm: The helm is a single MOOS application running as the process pHelmIvP. It is a behavior-based architecture where the primary output of a behavior on each iteration is an IvP objective function. The IvP solver performs multi-objective optimization on the set of functions to find 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

2.1.5 The Nested Autonomy Paradigm

For large-scale subsurface/surface ocean monitoring and observation systems (100+ square kilometers), no single unmanned platform has the ability in terms of sensing, endurance, and communications to achieve large-scale, long-endurance (several days to several weeks) system objectives. Even if multiple platforms are applied to the problem, effectiveness may be substantially diminished if limited to a single platform *type*. The *nested autonomy* paradigm, depicted in Fig. 2.5, is an approach to implementing a system of unmanned platforms for large-scale autonomous sensing applications. It is based in part on the objective of making seamless use of heterogeneous platform types using a uniform platform-independent autonomy architecture. It also assumes the platforms will have varying communications bandwidth, connectivity, and latency.

The *vertical* connectivity allows information to pass from sensors to the on-board sensor processing and autonomy modules, or from each node to other nodes in the cluster, or up to the field operator, and thus forms the basis for the autonomous *adaptive control* which is a key to the capability in compensating for the smaller sensor aperture of the distributed nodes. Similarly, the *horizontal* connectivity forms the basis for *collaboration* between sensors on a node (sensor fusion) or between nodes (collaborative processing and control).

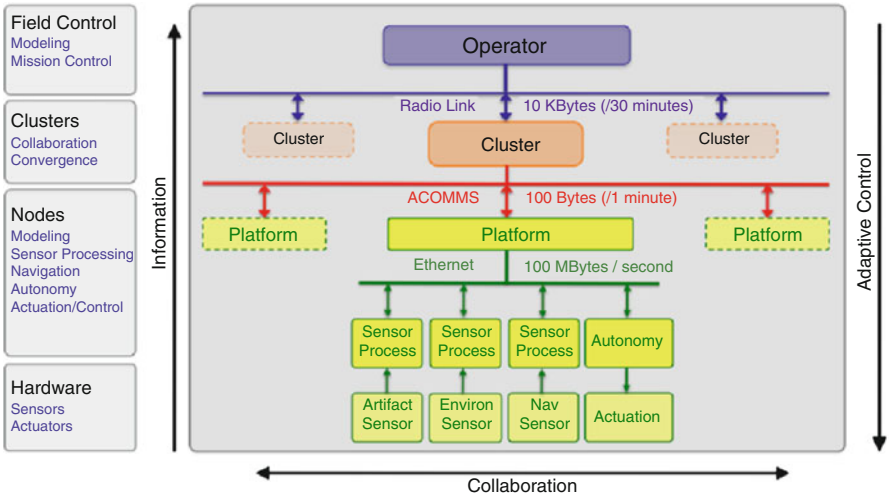

Fig. 2.5 The nested autonomy paradigm: Field-control operators receive intermittent information from field nodes as connectivity and bandwidth allow. Elements of clusters may serve a heterogeneous role as a gateway communications agent. Likewise, nodes receive intermittent commands and cues from field operators. Node autonomy compensates for and complements the sporadic connectivity to field control and other nodes in a cluster or network of clusters

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

The MOOS-IvP autonomy implementation is situated primary at the node level in the nested autonomy structure depicted in Fig. 2.5. However, aspects of the MOOS-IvP architecture are relevant to the larger picture as well. A key enabling factor to the nested autonomy paradigm is the platform independence of the node level autonomy system. The backseat driver design allows the decoupling of the vehicle platform from the autonomy system to achieve platform independence. The MOOS middleware architecture and the IvP Helm behavior-based architecture also contribute to platform independence by allowing an autonomy system to be comprised of modules that are swappable across platform types. Furthermore, collaborative and nested autonomy between nodes is facilitated by the simple modal interface to the on-board autonomy missions to control behavior activations.

2.2 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 standalone 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 in many extended field exercises with substantial robotic resources at stake. To frame the later discussion of the IvP Helm, the basic issues regarding MOOS applications are introduced here. For further information on the original design of MOOS, see [12].

2.2.1 Inter-Process Communication with Publish/Subscribe

MOOS has a star-like topology as depicted in Fig. 2.3. Application within a MOOS community (a MOOSApp) has 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 an unsolicited attempt to contact a MOOSApp from out of the blue.
- Each client has a unique name.
- The client need not have knowledge.

Table 2.1 The contents of MOOS message

Variable	Meaning
Name	The name of the data
String value	Data in string format
Double value	Numeric double float data
Source	Name of client that sent 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

- 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.

2.2.2 Message Content

The communications API in MOOS allow 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 of convenience MOOS does offer bespoke support for small "double" and string payloads.[1] Data is packed into messages which contain other salient information shown in Table 2.1.

Often it is convenient to send data in string format, for example the string `"Type=EST,Name=AUV,Pos=[3x1]3.4,6.3,0.2"` might describe the position estimate of a vehicle called "AUV" as a 3×1 column vector. This is human readable and does not require the sharing and synchronizing of headerfiles to ensure both sender and recipient understand how to interpret data (as is the case with binary data).

[1]Note that very early versions of MOOS only allowed data to be sent as strings or doubles—but this restriction is now long gone.

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 40 Hz in full color. At this point the need to send binary data is clear and of course MOOS supports it transparently (and via 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 *forwards* 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.

2.2.3 Mail Handling—Publish/Subscribe—in MOOS

Each MOOS application is a client having a connection to the MOOSDB. This connection is made on the client side and the client manages a 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

Publishing Data: Data is published as a pair—a variable and value—that constitutes the heart of a MOOS message described in Table 2.1. 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 2.1 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.

Registering for Notifications: Assume that a list of names of data published has been provided by the author of a particular MOOS application. For example, an application that interfaces to a GPS sensor may publish data called GPS_X and GPS_Y. A different application may register its interest in 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. Recent versions of MOOS also support "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."

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 2.1. 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.

2.2.4 Overloaded Functions in MOOS Applications

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

The roles of the three virtual functions in Fig. 2.6 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()) discussed in [12]. *The Iterate() Method:* By overriding the CMOOSApp::Iterate() function in a new derived class, the author

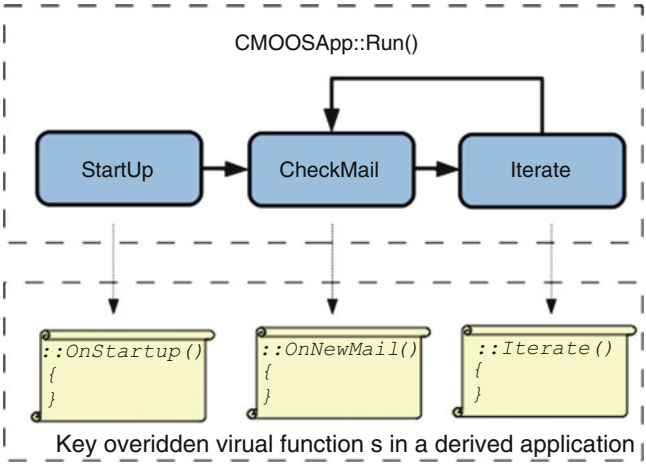

Fig. 2.6 Key virtual functions of the MOOS application base class: The flow of execution once `Run()` has been called on a class derived from `CMOOSApp`. The *scrolls* indicate where users of the functionality of CMOOSApp will be writing new code that implements whatever that is wanted from the new applications. 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

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 s to complete there is no way that this method can be called at more than 1 Hz. If you want to call `Iterate()` as fast as is possible simply request a frequency of zero—but you may want to reconsider why you need such a greedy application. *The OnNewMail() Method:* Just before `Iterate()` is called, the `CMOOSApp` base class determines whether new mail is present, i.e., whether some other process has posted data for which the client has previously registered, as described above. If new mail is waiting, the `CMOOSApp` base class calls the `OnNewMail()` virtual function, typically overloaded by the application. The mail arrives in the form of a list of `CMOOSMsg` objects (see Table 2.1). 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 backend 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. *The OnStartup() Method:* This function is called just before the application enters into its own forever-loop depicted in Fig. 2.6. This is the application that implements the application's initialization code and in particular reads configuration parameters (including those that modify the default behavior of the CMOOSApp base class) from a file.

2.3 IvP Helm Autonomy

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 competing behaviors, by performing 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 configuration given a set of behaviors and a set of mission objectives.

2.3.1 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 [7]. 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 predated the arrival of open-source software and the Free Software Foundation founded by Richard Stallman. Open-source software is not a prerequisite for building autonomy systems incrementally, but it has the capability of greatly accelerating that objective. The development of complex autonomy systems stands to significantly benefit if the set of developers at the table is large and diverse. Even more so if they can be from different organizations with perhaps even the loosest of overlap in interest regarding how to use the collective end product.

The interval programming solution algorithm, as well as the term itself, was motivated by the mathematical programming model, linear programming, developed by George Dantzig [9]. 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 expressive flexibility for handling current and future behavior and due to the opportunity to develop solution algorithms that exploit the piecewise linear constructs.

2.3.2 Nontraditional Aspects of the IvP Behavior-Based Helm

While IvP Helm takes its motivation from early notions of the behavior-based architecture, it is also quite different in many regards. The notion of behavior independence to temper the growth of complexity in progressively larger systems is still a principle closely followed in the IvP Helm. Behaviors may certainly influence one another from one iteration to the next. However, within a single iteration, the output generated by a single behavior is not affected at all by what is generated by other behaviors in the same iteration. The only 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.

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. Behavior input can be anything represented by a MOOS variable and perhaps generated by other MOOS processes outside the helm. It is allowable to have one or more planning engines running on the vehicle generating output consumed by one or more behaviors.
- Several instances of the same behavior are allowed. Behaviors generally accept a set of configuration parameters that allow them to be configured for quite different tasks or roles in the same helm and mission. Different waypoint behaviors, e.g., can be configured for different components of a transit mission. Or different collision avoidance behaviors can be instantiated for different contacts.
- Behaviors can be run in a configurable sequence. The `condition` and `endflag` parameters defined for all behaviors allow for a sequence of behaviors to be readily configured into a larger mission plan.
- Behaviors rate actions over a coupled decision space. IvP functions are defined over the Cartesian product of the set of vehicle decision variables. Objective functions for certain behaviors may only be adequately expressed over such a decision space. See, e.g., the function produced by the AvoidCollision behavior later in this chapter. This is distinct from the decoupled decision making

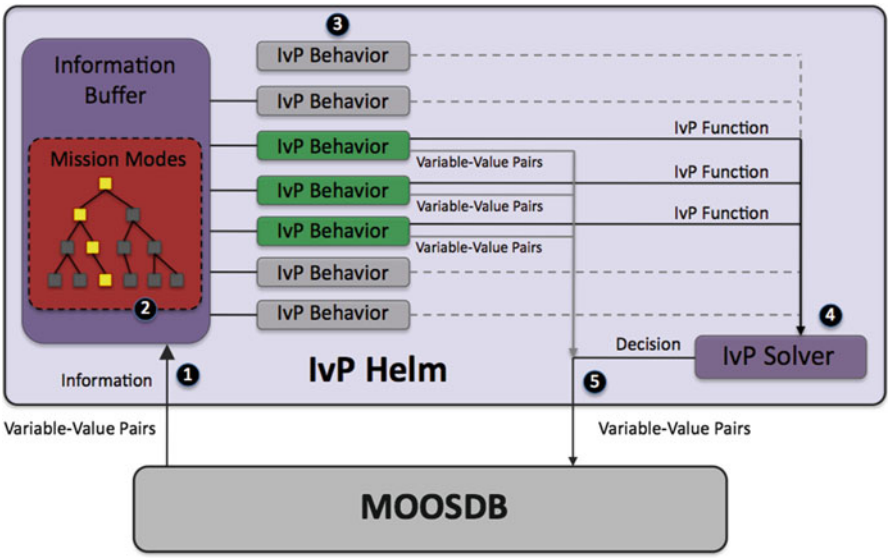

Fig. 2.7 The pHelmIvP iterate loop: (**a**) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors. (**b**) If there were any mode declarations in the mission behavior file they are evaluated at this step. (**c**) Each behavior is queried for its contribution 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. (**d**) The objective functions are resolved to produce an action, expressible as a set of variable–value pairs. (**e**) All variable–value pairs are published to the MOOSDB for other MOOS processes to consume

style proposed in [13, 15]—early advocates of multi-objective optimization in behavior-based action selection.

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 construction of mission behavior files.

2.3.3 Inside the IvP Helm: A Look at the Helm Iterate Loop

As with 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 Fig. 2.7.

Step 1. Reading Mail and Populating the Information Buffer: The first step of a helm iteration occurs outside the `Iterate()` loop. As depicted in Fig. 2.6, a

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

Step 2. Evaluation of Mode Declarations: Next, the helm evaluates any mode declarations specified in the behavior file. Mode declarations are discussed in Sect. 2.3.4. 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, e.g., evaluates its `condition` parameters. A condition for behavior participating in the current iteration could therefore read something like `condition = (MODE==SURVEYING)`. The exact value of the variable `MODE` is set during this step of the `Iterate()` loop.

Step 3. Behavior Participation: In the third step much of the work of the helm is realized by giving each behavior a chance to participate. Each behavior is queried sequentially—the helm contains no separate threads in this regard. The order in which behaviors are queried does not affect the output. This step contains two distinct parts for each behavior: (1) determination of whether the behavior will participate and (2) production of output if it is indeed participating on this iteration. Each behavior may produce two types of information as Fig. 2.7 indicates. The first is an objective function (or "utility" function) in the form of an IvP function. The second kind of behavior output is a list of variable–value pairs to be posted by the helm to the MOOSDB at the end of the `Iterate()` loop. A behavior may produce both kinds of information, neither, or one or the other, on any given iteration.

Step 4. Behavior Reconciliation: In the fourth step, 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. IvP functions are piecewise linearly defined, where 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.

Step 5. Publishing the Results to the MOOSDB: In the last step, the helm publishes all variable–value pairs to the MOOSDB, some of which were produced directly by behavior, and some of which were generated by the IvP Solver.

2.3.4 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.

2.3.4.1 Scripted Versus Adaptive Missions

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 being a sequence task to be instead 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, perhaps in sequences unknown at launch time, depending on what the vehicle senses and how they are commanded in the field.

2.3.4.2 Syntax of Hierarchical Mode Declarations

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

The hierarchy in Fig. 2.8 is formed by the mode declaration constructs on the left-hand side, taken as an excerpt from the `bravo.bhv` file. After the mode declarations

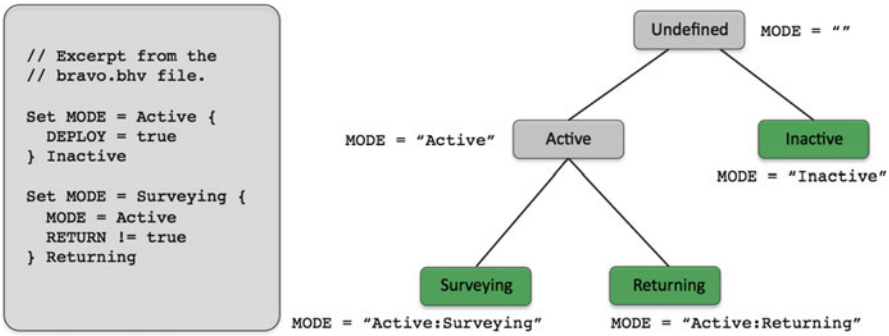

Fig. 2.8 Hierarchical modes for the Bravo mission: The vehicle will always be in one of the modes represented by a leaf node. A behavior may be associated with any node in the tree. If a behavior is associated with an internal node, it is also associated with all its children

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 Fig. 2.8 the variable is the concatenation of the names of all the nodes. The mode evaluation begins sequentially through each of the blocks. At the outset the value of the variable `MODE` is reset to the empty string. After the first block in Fig. 2.8, `MODE` will be set to either `"Active"` or `"Inactive"`. When the second block is evaluated, the condition `"MODE=Active"` is evaluated based on how `MODE` was set in the first block. For this reason, mode declarations of children need to be listed after the declarations of parents in the behavior file.

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

2.3.5 Behavior Participation in the IvP Helm

The primary work of the helm comes when the behaviors participate at each round of the helm `Iterate()` loop, by producing IvP functions, posting variable–value pairs to MOOS, or both. As depicted in Fig. 2.7, once the mode has been reevaluated

Fig. 2.9 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

taking into consideration newly received mail, it is time for the relevant behaviors to participate.

2.3.5.1 Behavior Conditions

On any single iteration a behavior may participate by generating an objective function to influence the helm's output over its decision space. Not all behaviors participate in this regard, and the primary 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>
```

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 such as:

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

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

2.3.5.2 Behavior Run States

On any given helm iteration a behavior may be in one of four states depicted in Fig. 2.9:

- Idle: A behavior is `idle` if it is not `complete` and it has not met its run conditions as described above in Sect. 2.3.5.1. The helm will invoke an idle behavior's `onIdleState()` function.
- Running: A behavior is `running` if it has met its run conditions and it is not `complete`. The helm will invoke a running behavior's `onRunState()` function thereby giving the behavior an opportunity to contribute an objective function.

- Active: A behavior is `active` if it is running and it did indeed produce an objective function when prompted. There are a number of reasons why a running behavior may not be active. For example, a collision avoidance behavior may opt to not produce an objective function when the contact is sufficiently far away.
- Complete: A behavior is `complete` when the behavior itself determines it is complete. It is up to the behavior author to implement this, and some behaviors may never complete. The function `setComplete()` is defined generally at the behavior superclass level, for calling by a behavior author. This provides some standard steps to be taken upon completion, such as posting of `endflags`, described below in Sect. 2.3.5.3. Once a behavior is in the `complete` state, it remains in that state permanently. All behaviors have a `DURATION` parameter defined to allow the behavior to be configured to time-out if desired. When a time-out occurs the behavior state will be set to `complete`.

2.3.5.3 Behavior Flags and Behavior Messages

Behaviors may produce a set of messages, i.e., variable–value pairs, on any given iteration (see Fig. 2.7). These messages can be critical for coordinating behaviors with each other and to other MOOS processes. This can also be invaluable for monitoring and debugging behaviors configured for particular missions. Behaviors do not post messages to the `MOOSDB`, they request the helm to post messages on their behalf. The helm collects these requests and publishes them to the `MOOSDB` at the end of the `Iterate()` loop. Variable–value pairs corresponding to state flags are set in the behavior file for each behavior. The following five flags are available:

- An `endflag` is posted once when or if the behavior enters the `complete` state. Multiple endflags may be configured for a behavior. By default, when a behavior is completed and has posted its endflag, it does not participate further. Its instance is destroyed and removed from the helm.
- An `idleflag` is posted on each iteration of the helm when the behavior is determined to be in the `idle` state. The variable–value pair representing the idleflag is given in the `idleflag` parameter in the behavior file.
- A `runflag` is posted on each iteration of the helm when the behavior is determined to be in the `running` state, regardless of whether it is further determined to be active or not. A `runflag` is posted exactly when an `idleflag` is not.
- An `activeflag` is posted on each iteration of the helm when the behavior is determined to be in the `active` state.
- An `inactiveflag` is posted on each iteration of the helm when the behavior is determined to be not in the `active` state.

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 Fig. 2.10. Behavior authors may implement their behaviors to post other messages as they see fit. For example, the waypoint

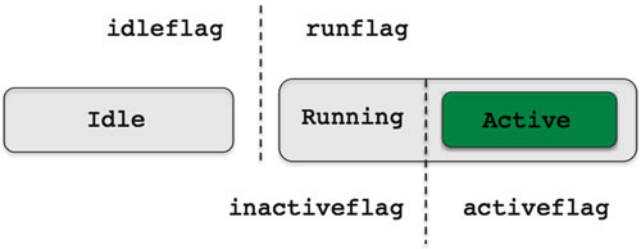

Fig. 2.10 Behavior flags: The four behavior flags `idleflag`, `runflag`, `activeflag`, and `inactiveflag` are posted depending on the behavior state and can be considered complementary in the manner indicated

behavior publishes a status variable, `WPT_STATUS`, with a status message similar to `"vname=alpha,index=0,dist=124,eta=62"` indicating the name of the vehicle, the index of the next point in the list of waypoints, the distance to that waypoint, and the estimated time of arrival.

2.3.6 Behavior Reconciliation Using Multi-objective Optimization

A unique aspect of the IvP Helm is the manner in which behaviors are reconciled when there are two or more behaviors competing for influence of the helm decision.

2.3.6.1 IvP Functions

IvP functions are produced by behaviors to influence the decision produced by the helm on the current iteration (see Fig. 2.7). The decision is typically comprised of the desired *heading*, *speed*, and *depth*, but the helm decision space could be comprised of any arbitrary configuration. 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 Fig. 2.12).
- 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.

- Pieces in IvP functions need not be of uniform size. More pieces can be dedicated to parts of the domain that are harder to approximate with linear functions.
- IvP functions need only be defined over a subset of the domain. Behaviors are not affected if the helm is configured for additional variables that a behavior may not care about. It is allowable, e.g., to have a behavior that produces IvP functions solely over the vehicle *depth*, even though the helm may be producing decisions over *heading*, *speed*, and *depth*.

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

2.3.6.2 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 in Fig. 2.11:

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 Fig. 2.12a, b. The difference in these two figures is only the size of the piece. More pieces (Fig. 2.12a) result in a more accurate approximation of the underlying function but takes longer to generate and creates further work for the IvP solver when the functions are combined. IvP functions need not use uniformly sized pieces. By using the *directed refinement* option in the IvP Build Toolbox, an initially uniform IvP function can be further refined with more pieces over a sub-domain directed by the caller, with smaller uniform pieces of the caller's choosing. This is rendered in Fig. 2.12c. 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.

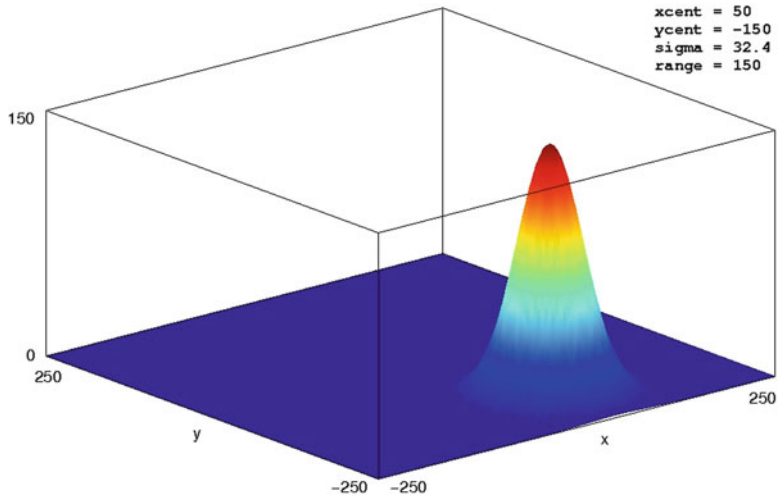

Fig. 2.11 A rendering of the function $f(x,y) = Ae^{-\left(\frac{(x=x_0)^2+(y=y_0)^2}{2\sigma^2}\right)}$ where A = `range` = 150, σ = `sigma` = 32.4, x_0 = `xcent` = 50, and y_0 = `ycent` = -150. The domain here for x and y ranges from -250 to 250

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 rendered in Fig. 2.12d.

The reflector tools work similarly in N dimensions and on multimodal functions. The only requirement for using the reflector tool is to provide it with access to the underlying function. 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 longest part of the process.

2.3.6.3 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

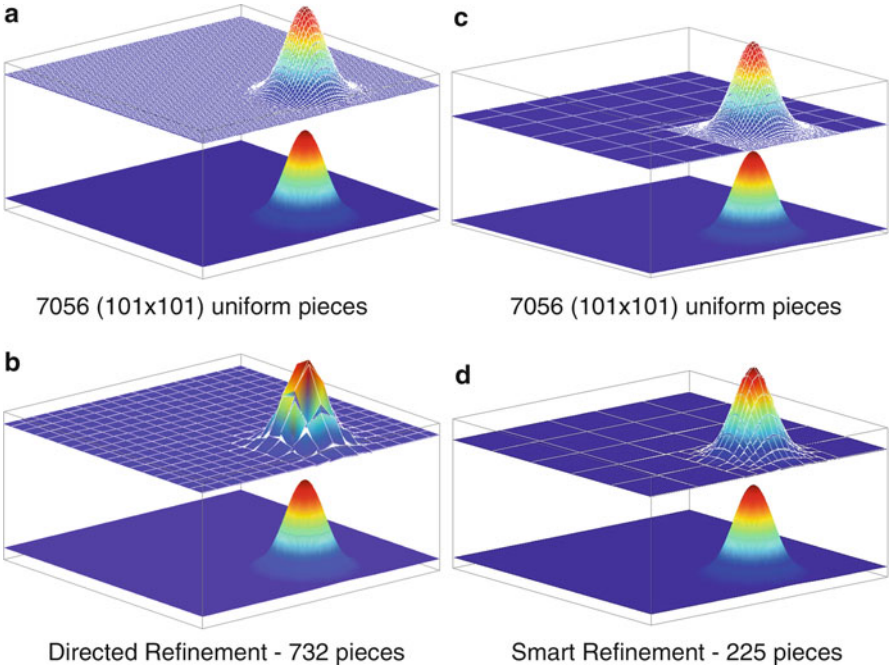

Fig. 2.12 A rendering of four different IvP functions approximating the same underlying function: The function in (**a**) uses a uniform distribution of 7,056 pieces. The function in (**b**) uses a uniform distribution of 1,024 pieces. The function in (**c**) was created by first building a uniform distribution 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

of each function is given by w_i, the solution to a problem with k functions is given by

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

The algorithm is described in detail in [3], 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 in the configuration of the IvP Helm, which always finds the global optimum, since this stage of computation is very fast in practice.
- *Initial Solution*: A key factor of an effective branch-and-bound algorithm is seeding the search with a decent initial solution. In the IvP Helm, the initial solution used is the solution (typically `heading`, `speed`, `depth`) generated on the previous helm iteration. In practice this appears to provide a speedup by about a factor of two.

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

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

2.4 IvP Helm Behaviors

Helm behaviors derive part of their function from inheriting properties from a base class behavior implementation, and their unique capabilities are an extension of the base capability. The uniform base capability allows for mission configurations to be constructed in a simple predictable manner. Here we discuss (a) the base capabilities

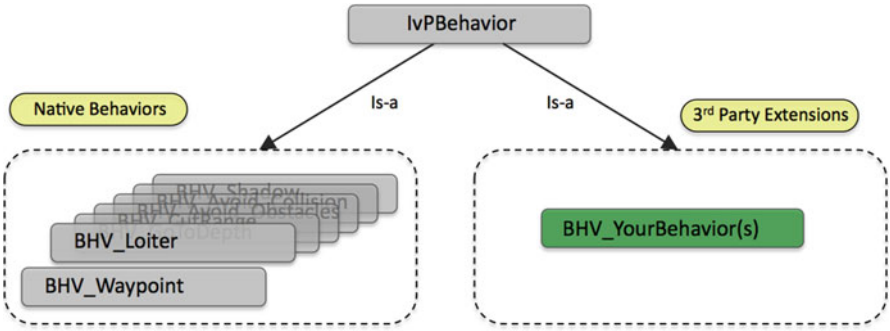

Fig. 2.13 Behavior inheritance: Behaviors are derived from the `IvPBehavior` superclass. The native behaviors are the behaviors distributed with the helm. New behaviors also need to be a subclass of the IvPBehavior class to work with the helm. Certain virtual functions invoked by the helm may be optionally but typically overloaded in all new behaviors. Other private functions may be invoked within a behavior function as a way of facilitating common tasks involved in implementing a behavior

of IvP behaviors, (b) how behaviors are handled generally by the helm in each iteration, (c) the hooks for creating a new third-party behavior, (d) an overview of standard behaviors that are distributed with the helm in the open domain, and (e) a more detailed look at a few representative behaviors.

2.4.1 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 Fig. 2.13. The is-a relationship of a derived class provides a form of code reuse as well as a common interface for constructing mission files with behaviors.

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

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

2.4.2 Parameters Common to All IvP Behaviors

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

```
parameter = value.
```

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

2.4.2.1 A Summary of General Behavior Parameters

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

NAME The name of the behavior—should be unique between all behaviors.

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.

DURATION The time in seconds that the behavior will remain running before declaring completion. If unspecified, the behavior will never time-out.

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

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

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

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

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

ENDFLAG This parameter specifies a variable and a value to be posted when the behavior completes. 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.

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 start-up time. Any parameter and value pair that would have been legal at start-up time is legal at runtime. This is one of the primary hooks to the helm for mission control—the other being the behavior conditions described above.

NOSTARVE The `NOSTARVE` parameter allows a behavior to assert a maximum staleness for one or more MOOS variables, i.e., the time since the variable was last updated.

PERPETUAL Setting the `perpetual` parameter to `true` allows the behavior to continue to run even after it has completed and posted its endflags.

2.4.2.2 Altering Behaviors 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 1. The return point is the (0,0) point in local coordinates, and return speed is 2.0 m/s. When the conditions are met, this is what will be executed.

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

```
Behavior = BHV_Waypoint
{
  name      = WAYPT_RETURN
  priority  = 100
  speed     = 2.0
  radius    = 8.0
  points    = 0,0
  UPDATES   = RETURN_UPDATES
  condition = RETURN = true
  condition = DEPLOY = true
}
```

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

```
parameter=value # parameter=value # ... # 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 start-up. 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."
```

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

2.4.3 IvP Helm Behaviors in the Public Domain

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

1. The *AvoidCollision* behavior will maneuver the vehicle to avoid a collision with a given contact. The generated objective function is based on the calculated closest point of approach (CPA) for a given contact and candidate maneuvers. The safe-distance tolerance and policy for priority based on range is provided in mission configuration.
2. The *AvoidObstacles* behavior will maneuver the vehicle to avoid obstacles at known locations, each expressed as a convex polygons. The safe-distance tolerance and policy for priority based on range are configuration parameters.
3. The *CutRange* behavior will maneuver the vehicle to reduce the range between itself and a given contact. The generated objective function is based either on the calculated CPA for a given contact and candidate maneuver or based purely on a greedy approach of heading toward the contact's present position. The policies may be combined (weighted) by the user in mission configuration. This

behavior also has the ability to extrapolate the other vehicle's position from prior reports when contact reports are received intermittently.

4. The *GoToDepth* behavior will drive the vehicle to a sequence of specified depths and duration at each depth. The duration is specified in seconds and reflects the time at depth after the vehicle has first achieved that depth, where achieving depth is defined by the user-specified tolerance parameter. If the current depth is within the tolerance, 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 tolerance setting.
5. The *Loiter* behavior is used for transiting to and repeatedly traversing a set of waypoints. A similar effect can be achieved with the waypoint behavior, but this behavior assumes a set of waypoints forming a convex polygon. It also robustly handles dynamic exit and reentry modes when or if the vehicle diverges from the loiter region due to external events. It is dynamically reconfigurable to allow a mission control module to repeatedly reassign the vehicle to different loiter regions by using a single persistent instance of the behavior.
6. The *MemoryTurnLimit* behavior is used to avoid vehicle turns that may cross back on its own path and risk damage to towed equipment. It is configured with two parameters combined to set a vehicle turn radius pseudo-limit. This behavior is described more fully in Sect. 2.4.3.2.
7. The *OpRegion* behavior provides four different types of safety functionality: (a) a boundary box given by a convex polygon in the *x*-*y* or lat-lon plane, (b) an overall time-out, (c) a depth limit, and (d) an altitude limit. The behavior does not produce an objective function to influence the vehicle to avoid violating these safety constraints. This behavior merely monitors the constraints and may post an error resulting in the posting of an all-stop command.
8. The *PeriodicSpeed* behavior will periodically influence the speed of the vehicle while remaining neutral at other times. The timing is specified by a given period in which the influence is on and a period specifying when the influence if off. The motivation was to provide an ability to periodically reduce self-noise to allow for a window of acoustic communications.
9. The *PeriodicSurface* 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.
10. The *Shadow* behavior will mimic the observed heading and speed of another given vehicle, regardless of its position relative to the vehicle.
11. The *StationKeep* 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.
12. The *Trail* behavior will attempt to keep the vehicle at a given range and bearing to another specified vehicle. It may serve the purpose of formation-keeping.

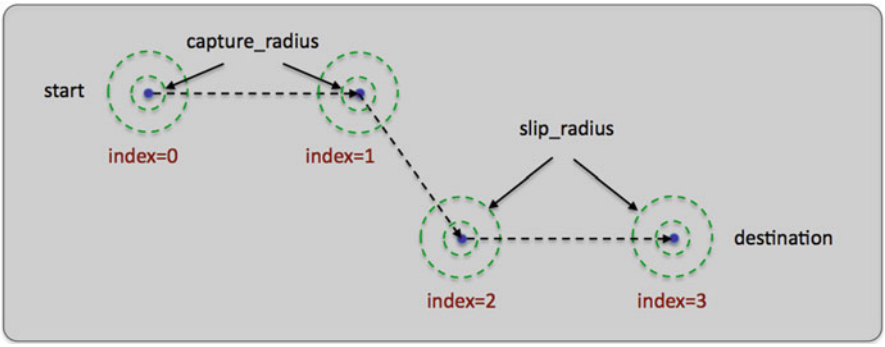

Fig. 2.14 The waypoint behavior: The waypoint behavior traverses a set of waypoints. A capture radius defines what it means to reach a waypoint, and a slip radius is specified to define what it means to be "close enough" should progress toward the waypoint be noted to degrade

It has the ability to extrapolate the other vehicle's position from prior reports when contact reports are received intermittently.

13. 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.

We now explore three of the above behaviors in detail.

2.4.3.1 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. The basic idea is shown in Fig. 2.14.

The behavior may also be configured to perform a degree of track-line following, i.e., 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 and lawnmower pattern.

The Waypoint Behavior Configuration Parameters

The configuration parameters and variables published collectively define the interface for the behavior. The following are the parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Sect. 2.4.2.

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

Variables Published by the Waypoint Behavior

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

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

```
      WPT_STAT = vname=alpha,behavior-name=waypt_survey,
                 index=1,hits=1/1,cycles=0,dist=30,eta=15
     WPT_INDEX = 3
   CYCLE_INDEX = 1
    VIEW_POINT = x=0,y=0,active=false,label=alpha's next waypoint,
                 type=waypoint,source=alpha_waypt_return
  VIEW_SEGLIST = pts={60,-40:60,-160:150,-160:180,-100:150,-40},
                 label=alpha_waypt_survey,vertex_color=yellow
```

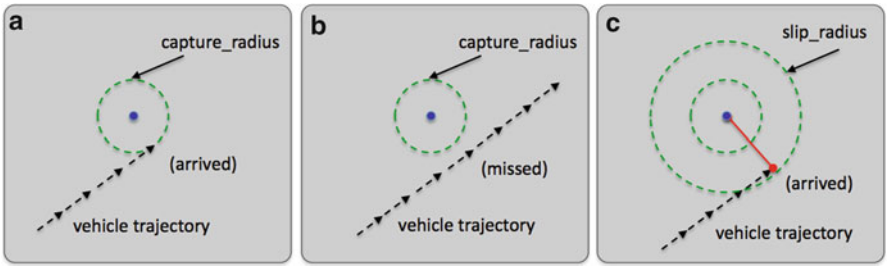

Fig. 2.15 The capture radius and slip radius: (**a**) A successful waypoint arrival by achieving proximity less than the capture radius. (**b**) A missed waypoint likely resulting in the vehicle looping back to try again. (**c**) A missed waypoint but arrival declared anyway when the distance to the waypoint begins to increase and the vehicle is within the slip radius

Specifying Waypoints: The `Points`, `Order`, and `Repeat` Parameters

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

```
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 not continue to launch. If the syntactic error is passed as part of a dynamic update (see Sect. 2.4.2.2), the change in waypoints will be ignored and the a warning posted to the `BHV_WARNING` variable. The behavior can be set to repeat its waypoints indefinitely by setting `repeat="forever"`.

The `Capture_Radius` and `Slip_Radius` Parameters

The `capture_radius` parameter specifies the distance to a given waypoint 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 Fig. 2.14. The slip radius parameter specifies an alternative criteria for achieving a waypoint.

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 `nm_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 Fig. 2.15. The

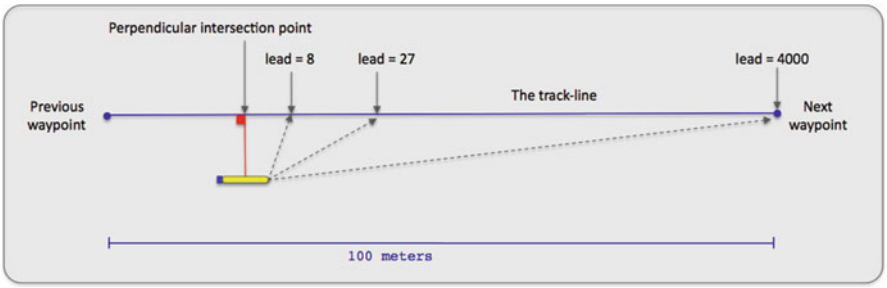

Fig. 2.16 The track-line mode: When in track-line mode, the vehicle steers toward a point on the track-line rather than simply toward the next waypoint. The steering point is determined by the `lead` parameter. This is the distance from the perpendicular intersection point toward the next waypoint

behavior keeps a running tally of hits achieved with the capture radius and those achieved with the slip radius. These tallies are reported in a status message.

Track-Line Following Using the `lead` Parameter

By default the waypoint behavior will output a preference for the heading that is directly toward the next waypoint. By setting the `lead` parameter, the behavior will instead output a preference for the heading that keeps the vehicle closer to the track-line or the line between the previous waypoint and the waypoint currently being driven to (Fig. 2.16).

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.

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

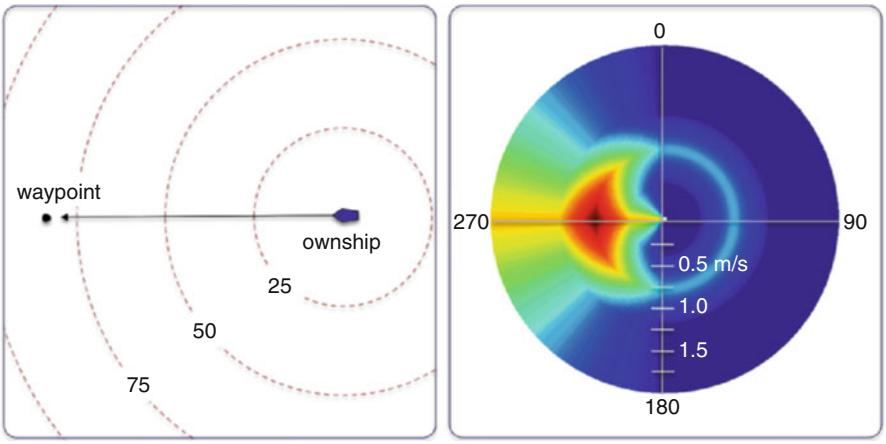

Fig. 2.17 A waypoint objective function: The objective function produced by the waypoint behavior is defined over possible heading and speed values. Depicted here is an objective function favoring maneuvers to a waypoint 270° from the current vehicle position and favoring speeds closer to the mid-range of capable vehicle speeds. Higher speeds are represented farther radially out from the center

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 Fig. 2.17 are built in the first manner.

2.4.3.2 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, 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:

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

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

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

$$\text{heading_avg} = \text{atan2}(s, c) \cdot 180/\pi, \tag{2.1}$$

where s and c are given by

$$s = \sum_{k=0}^{n-1} \sin(h_k \pi/180)), \qquad c = \sum_{k=0}^{n-1} \cos(h_k \pi/180)).$$

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

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

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

The objective function produced by this behavior looks effectively like a constraint on the value of the heading. Some typical functions are rendered from simulation in Fig. 2.18. This figure depicts the MemoryTurnLimit IvP function alongside the Waypoint IvP function as the vehicle traverses a set of waypoints, where the final waypoint (waypoint #3 in the figure) represents nearly a 180° turn with respect to the prior waypoint. The first frame in the figure depicts the vehicle trajectory *without* the influence of the MemoryTurnLimit behavior and shows a very sharp turn between waypoints #2 and #3. The MemoryTurnLimit behavior, shown in the last five frames of Fig. 2.18, is used to avoid the sharp turn in the first frame. This behavior was configured with TURN_RANGE=45 and MEMORY_TIME=20. The turn between waypoints #1 and #2 is not affected by the MemoryTurnLimit behavior because the new desired heading (180°) is within the tolerance of the heading history recorded up to the turning point. The same cannot be said when the vehicle reaches waypoint #2 and begins to traverse to waypoint #3. The recent heading history as it arrives at waypoint #2 reflects the time spent traversing from waypoint #1 and #2. The heading average given by (2.1) at this point (time = 98) is 180°, and the IvP function produced by the MemoryTurnLimit behavior restricts the vehicle heading to be $180 \pm 45°$. As the vehicle continues its turn toward waypoint #3, the heading average of the MemoryTurnLimit behavior and its IvP function evolve to eventually allow a desired heading that is consistent with the optimal point of the Waypoint IvP function as shown in the last frame at time = 170. The resulting turn between

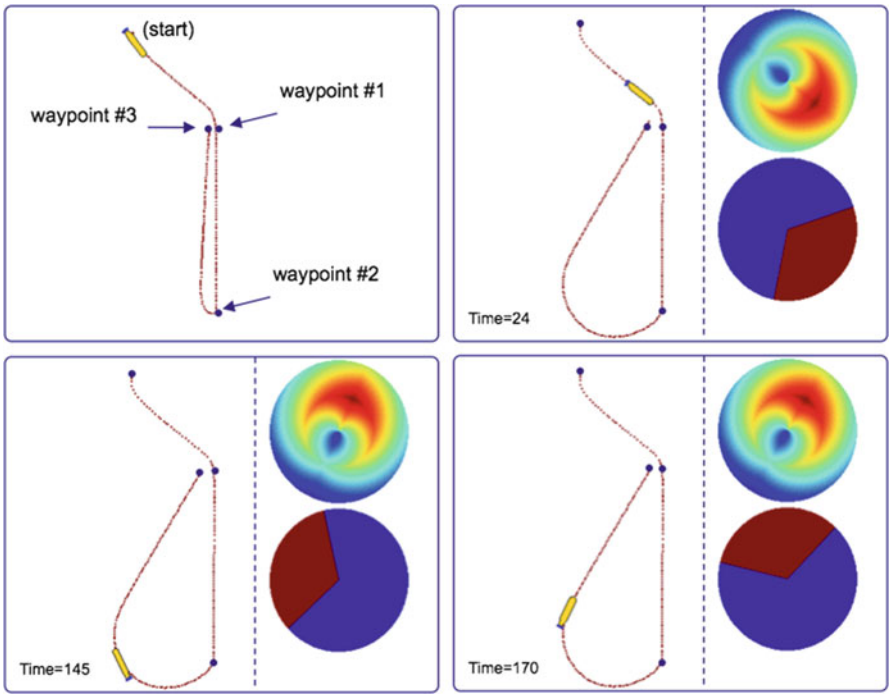

Fig. 2.18 The MemoryTurnLimit and waypoint behaviors: The MemoryTurnLimit behavior is used to avoid sharp vehicle turns between waypoints as shown, from simulation, in the first, *upper-left frame*. The MemoryTurnLimit behavior is configured with `TURN_RANGE=45` and `MEMORY_TIME=20`. In each frame, the IvP function of the waypoint behavior is shown in the *upper right* and that of the MemoryTurnLimit behavior is in the *lower right*. At time = 24, the vehicle is approaching waypoint #1 and its recent heading is 140°. After the vehicle reaches waypoint #2, the desired heading from the waypoint behavior is not in the range of allowed headings from the MemoryTurnLimit behavior. As the vehicle turns, e.g., time = 145, the heading history of the MemoryTurnLimit behavior evolves to eventually allow the peak desired heading of the waypoint behavior, time = 170

waypoints #2 and #3 is much wider than that shown in the first frame. Discussion of this behavior used in field experiments with a UUV towing a sensor array can be found in [4].

2.4.3.3 The AvoidCollision Behavior

The AvoidCollision behavior will produce IvP objective functions designed to avoid collisions (and near collisions) with another specified vehicle. The IvP functions produced by this behavior are defined over the domain of possible heading and speed choices. The utility assigned to a point in this domain (a heading–speed pair) depends in part on the calculated CPA between the candidate maneuver leg and the

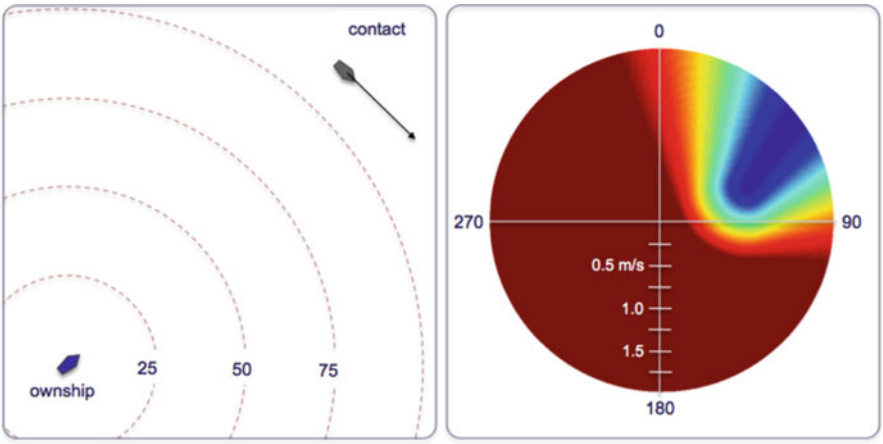

Fig. 2.19 The closest point of approach mapping: The function on the *right* indicates the relative change in calculated closest point of approach between ownship and contact position and trajectory shown on the *left*

contact leg formed from the contact's position and trajectory. Figure 2.19 shows the relationship $\text{cpa}(\theta, v)$ between CPA and candidate maneuvers (θ, v), where θ=heading and v=speed, for a given relative position between ownship and a given contact vehicle and trajectory. The IvP function generated by the AvoidCollision behavior applies a further user-defined utility function to the CPA calculation for a candidate maneuver, $f(\text{cpa}(\theta, v))$. The form of $f()$ is determined by behavior configuration parameters described below.

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

The AvoidCollision Configuration Parameters

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

Spawning and Killing New Behavior Instances Dynamically

The AvoidCollision behavior may be configured as a *template* that spawns a new behavior for each new contact, using the below two configuration lines:

```
templating = spawn
updates    = COLL_AVOID_INFO
```

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

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

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

Configuring the AvoidCollision Behavior Utility Function

The IvP function generated by this behavior is defined over the range of possible heading and speed decisions. Its form is derived in part from the calculation of the CPA calculated for a candidate maneuver leg, of a duration given by the `TIME_ON_LEG` parameter, which is set to 60 seconds by default. The *utility* of a given CPA value is determined further by a pair of configuration parameters. Distances less than or equal to `MIN_UTIL_CPA_DIST` are given the lowest utility, equivalent to an actual collision. Distances greater than or equal to `MAX_UTIL_CPA_DIST` are given the highest utility. These two parameters, and the raw CPA calculations, determine the form of the function. The magnitude, or weight, of the function is determined by the range between the two vehicles and two further configuration parameters. At ranges greater than `PWT_OUTER_DIST`, the weight is set to zero. At ranges less `PWT_INNER_DIST`, the weight is set to 100 % of the user-configured priority weight. The weight varies linearly when the range between vehicles falls somewhere between.

Closest Point of Approach Calculations and Caching

The production of IvP functions for this behavior is potentially CPU intensive compared to other behaviors, primarily due to the repeated calculations of CPA values. Recall from Sect. 2.3.6.2 that IvP functions built with the reflector tool are built by repeated sampling of the underlying function. This repeated sampling, and the existence of common partial calculations between samples, allows for caching of intermediate results to greatly speed up sampling for this behavior. This is implemented in a C++ class called a CPAEngine in separate utility library for use in other behaviors that reason about CPA, such as the CutRange and Trail behaviors.

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

$$\text{dist}^2(\theta, v, t) = k_2 t^2 + k_1 t + k_0, \tag{2.2}$$

where

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

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

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

Since there is no "maximum" distance, this stationary point always represents the time of the CPA, and therefore $t_{\min} = -k_1/2k_2$. The value of $t_{\min}$ may be in the past, i.e., less than zero, if the two vehicles are currently opening range. On the other hand, $t_{\min}$ may occur after t, the time length of the candidate maneuver $\langle \theta, v, t \rangle$. Therefore the value of $t_{\min}$ is clipped by $[0, t]$. Furthermore, $t_{\min} = 0$ in the special case when the two vehicles have the same heading and speed (the only case where k_2 is zero). The actual CPA value is obtained by substituting $t_{\min}$ back into (2.2):

$$\text{cpa}(\theta, v) = \sqrt{k_2 {t_{\min}}^2 + k_1 t_{\min} + k_0}. \tag{2.3}$$

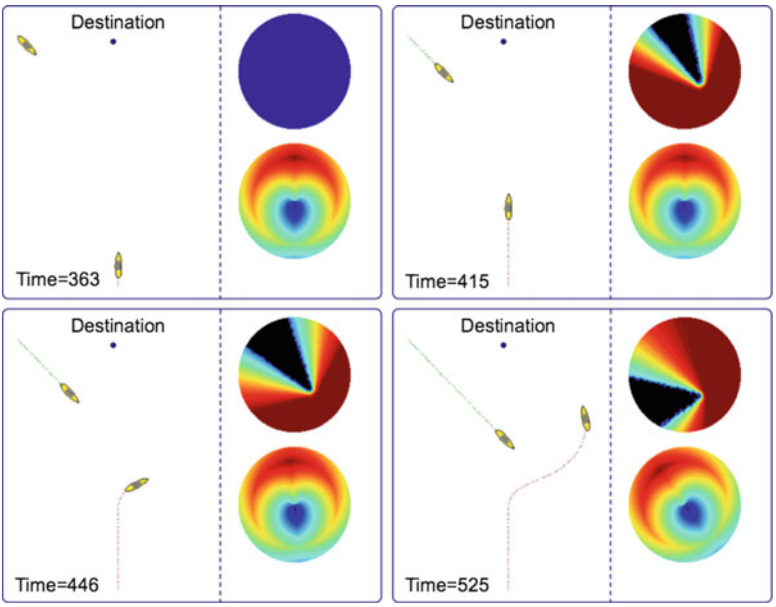

Fig. 2.20 The AvoidCollision and waypoint behaviors: The bravo vehicle maneuvers to a destination point with the waypoint behavior. The IvP objective function produced by the waypoint behavior is the lower function shown on the *right* in each frame. Beginning in the second frame, time=415, the AvoidCollision behavior becomes active and begins to produce IvP objective functions shown in the *upper right* of each frame. At each point in time, the helm chooses the heading and speed that represent the optimal decision given the pair of IvP functions

In the generation of a single IvP function with $\text{cpa}(\theta, v)$ as a component of each sample of the decision space, intermediate values (2.2) may be cached that have the same values of current vehicle position (x, y) and current position and trajectory of the other vehicle (x_b, y_b, θ_b, v_b). A further cache, normally of size 360, is typically used for terms involving θ, ownship heading.

The AvoidCollision behavior is shown in Fig. 2.20 in a simple scenario working with the waypoint behavior in simulation. In Fig. 2.20, the effect of the AvoidCollision behavior in two fielded UUVs is shown.

2.5 Conclusions

In this chapter two architectures were described in detail. The MOOS publish-subscribe middleware is both an architecture and mechanism for inter-process communication and process scheduling but also as an open-source software project, a collection of substantial applications for sensing, communications, autonomy, debugging, and post-mission analysis. The IvP Helm is a behavior-based architecture, unique in its use of the IvP model for multi-objective optimization for resolving

competing autonomy behaviors. It is also an open-source project that includes many well-tested vehicle behaviors and autonomy tools for creating, debugging, and analyzing autonomy capabilities.

Acknowledgements The prototype of MOOS was developed by Paul Newman at MIT under the GOATS'2000 NURC Joint Research Program, with ONR support from Grant N-00014-97-1-0202 (Program Managers Tom Curtin, Code 322OM, Jeff Simmen, Code 321OA, Tom Swean, Code 321OE, and Randy Jacobson, Code 321TS). The development of the Nested Autonomy concept for environmental acoustic sensing and the MIT component of the GLINT'08 experiment was funded by the Office of Naval Research under the GOATS program, Grant N-00014-08-1-0013 (Program Manager Ellen Livingston, ONR Code 321OA). The development of the unified communication, command, and control infrastructure and the execution of the SWAMSI09 experiment was supported by ONR, Grant N-00014-08-1-0011 (Program Manager Bob Headrick, Code 321OA).

The IvP Helm autonomy software and the basic research involved in the interval programming model for multi-objective optimization has been developed under support from ONR Code 311 (Program Managers Don Wagner and Behzad Kamgar-Parsi). Prior prototype development of IvP concepts benefited from the support of the In-house Laboratory Independent Research (ILIR) program at the Naval Undersea Warfare Center in Newport, RI.

The NATO Undersea Research Centre (NURC) has supported the development of the MOOS-IvP Nested Autonomy concept by conducting 7 major field experiments, in which MIT LAMSS has been a partner, including GOATS'2000 and GOATS'2002, FAF'2003, FAF'2005, CCLNet'08, GLINT'08, and GLINT'09. Without the world-class seagoing experiment capabilities of NURC, with its state-of-the-art RVs, NRV Alliance and CRV Leonardo and their outstanding crew, and NURC's excellent engineering and logistics support, the Nested Autonomy concept and the underlying MOOS-IvP software base would not have reached the level of sophistication and robustness that it has achieved.

References

1. Arkin RC (1987) Motor schema based navigation for a mobile robot: an approach to programming by behavior. In: Proceedings of the IEEE conference on robotics and automation, Raleigh, NC, pp 264–271
2. Arkin RC, Carter WM, Mackenzie DC (1993) Active avoidance: escape and dodging behaviors for reactive control. Int J Pattern Recognit Artif Intell 5(1):175–192
3. Benjamin MR (2004) The interval programming model for multi-objective decision making. In: Technical report AIM-2004-021, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA
4. Benjamin M, Battle D, Eickstedt D, Schmidt H, Balasuriya A (2007) Autonomous control of an unmanned underwater vehicle towing a vector sensor array. In: International conference on robotics and automation (ICRA), Rome, Italy
5. Benjamin M, Schmidt H, Leonard JJ. http://www.moos-ivp.org
6. Bennet AA, Leonard JJ (2000) A behavior-based approach to adaptive feature detection and following with autonomous underwater vehicles. IEEE J Oceanic Eng 25(2):213–226
7. Brooks RA (1986) A robust layered control system for a mobile robot. IEEE J Robotics Automation RA-2(1):14–23
8. Carreras M, Batlle J, Ridao P (2000) Reactive control of an AUV using motor schemas. In: International conference on quality control, automation and robotics, Cluj Napoca, Rumania
9. Dantzig GB (1948) Programming in a linear structure. Comptroller, US Air Force, Washington, DC

10. Khatib O (1985) Real-time obstacle avoidance for manipulators and mobile robots. In: Proceedings of the IEEE international conference on robotics and automation, St. Louis, MO, pp 500–505
11. Kumar R, Stover JA (2001) A behavior-based intelligent control architecture with application to coordination of multiple underwater vehicles. IEEE Trans Syst, Man, and Cybernetics - Part A: Cybernetics 30(6):767–784
12. Newman PM (2003) MOOS - a mission oriented operating suite. In: Technical report OE2003-07, MIT Department of Ocean Engineering
13. Pirjanian P (1998) Multiple objective action selection and behavior fusion. Ph.D. thesis, Aalborg University
14. Riekki J (1999) Reactive task execution of a mobile robot. Ph.D. thesis, Oulu University
15. Rosenblatt JK (1997) DAMN: a distributed architecture for mobile navigation. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA
16. Rosenblatt JK, Williams SB, Durrant-Whyte H (2002) Behavior-based control for autonomous underwater exploration. Int J Inform Sci 145(1–2):69–87
17. Williams SB, Newman P, Dissanayake G, Rosenblatt JK, Durrant-Whyte H (2000) A decoupled, distributed AUV control architecture. In: Proceedings of 31st international symposium on robotics, Montreal, Canada, pp 246–251

http://www.springer.com/978-1-4614-5658-2

Marine Robot Autonomy
Seto, M.L. (Ed.)
2013, IX, 382 p. 147 illus., 102 illus. in color., Hardcover
ISBN: 978-1-4614-5658-2