TRANSNAV
http://www.transnav.eu
the International Journal
on Marine Navigation
and Safety of Sea Transportation
Volume 18
Number 2
June 2024
DOI: 10.12716/1001.18.02.01

Design of Software for Underwater Vehicles Operating in a Swarm

T. Praczyk & K. Naus
Polish Naval Academy, Gdynia, Poland

ABSTRACT: The paper presents a design of software for underwater vehicles constructed as part of the European Defence Agency project entitled SABUVIS II. Since the aim of the project is to develop a technology for a swarm of underwater vehicles, the software of these vehicles, in addition to the functionalities typical for each autonomous underwater vehicle, must have functionalities dedicated to the swarm of vehicles. The paper presents the design of both the vehicle-side and the shoreside part of the software.

1 INTRODUCTION

Autonomous Underwater Vehicles (AUV) are underwater robots that operate on their own without human support. Thanks to the ability to operate independently, they can replace people in the implementation of dangerous underwater missions, for example, during work related to the maintenance of oil rigs or underwater pipelines.

These vehicles can also operate as part of teams or swarms. In the first case, the vehicles usually operate at a distance from each other and the common task of the team is divided into separate subtasks that can be performed independently by each team member. Vehicles operating in teams must also usually be fully equipped with all navigation and sensory devices, which results from the fact that they operate at a certain distance from other vehicles and have to manage on their own.

Another option is to operate in a swarm. In this case, the vehicles form a closely cooperating group, moving in close proximity to each other. This approach means that a swarm of vehicles can be considered as a dispersed single underwater vehicle, or a vehicle-swarm, consisting of component vehicles, each of which may be responsible for a different part of the task of the vehicle-swarm.

Distribution of competences over many different vehicles makes individual vehicles in a swarm cheaper, smaller, lighter and easier to operate than super-vehicles operating on their own. The problem, however, in this case is the organization of cooperation between the vehicles so that they can really be considered as one compact organism.

The main objective of European Defense Agency project category B entitled "Swarm of Autonomous Biomimetic Underwater Vehicles" (SABUVIS II) to which the current paper is devoted is to design and implement a swarm of closely cooperating AUVs, including the leaders that are responsible for global swarm navigation and the followers responsible for a specific swarm task.

The cooperation of vehicles within a swarm, as mentioned before, is a serious challenge. In order for the vehicles to cooperate with each other, operate close to each other in a specific formation, certain

conditions must be met. These conditions apply to both the hardware and software.

When it comes to the hardware, it is important to equip the vehicles with sensors that enable omnidirectional observation of the vehicle surroundings so that the vehicles are able to track other vehicles in the swarm and detect obstacles. The necessary equipment of vehicles is also appropriate navigation. In the case of the leader, these must be systems that allow navigation over long distances, while in the case of followers, simpler systems operating over short distances are sufficient.

In the case of the software part, it is important to handle all hardware devices and on-board hardware systems, provide the vehicle with information about the external environment, control vehicles at high and low-level, control vehicle in an autonomous as well as remote control mode, and handle emergency situations.

The paper presents a software architecture design for vehicles built in the SABUVIS II project. The software consists of a vehicle-side part and a shoreside part. The presented architecture assumes the use of MOOS-IvP [1] which is a set of open source C++ modules for providing autonomy on robotic platforms, in particular autonomous marine vehicles.

2 DESIGN OF HIGH-LEVEL CONTROL SYSTEM INCLUDING VEHICLE SOFTWARE COMPONENTS

The high-level control system (HLCS) is understood in the paper as a software system which makes high-level decisions regarding movement of the vehicle, e.g. turn right, increase speed, go down. The high-level decisions are then transformed into low-level decisions for vehicle drive, rudder and other executive elements. The task of low-level decisions is to reach a desirable state of the vehicle in the form of desirable values of vehicle parameters, i.e. heading, depth, and speed.

In addition to the HLCS working on the vehicle, there is also another system, called Deck Unit (DU), which closely cooperates with the HLCS and which is situated on the shoreside. Both HLCS and DU are design in the same technology, namely, based on MOOS IvP framework [1].

MOOS IvP is a set of open-source C++ modules for providing autonomy on robotic platforms, in particular autonomous marine vehicles. The abbreviation MOOS comes from “Mission Oriented Operating Suite" whereas the abbreviation IvP stands for "Interval Programming”.

In the paper, the high-level architecture of MOOS IvP software for shoreside and vehicle-side is specified. The architecture is presented in the form of list of MOOS IvP applications which constitute the basic component of each MOOS IvP architecture and the network of communication links between the components. Application MOOSDB plays a central role in both architectures being a communication medium between most applications belonging to the same MOOS community

The vehicle-side architecture is depicted in Figure 1. It consists of eighteen MOOS IvP applications, i.e.: MOOSDB, pJANUS-MOOSBridge, pShare, pLow-LevelSoftBridge, pMission, pRemote, pHelmIvP, pNodeReporter, pNavigation, pNavDevicesHandlers, pCamera, pEchosounder, pSonar, pBMS, pLogger, pGenVehicleState, pContactMgrV20, and pObstacleMgr. All the applications are outlined below.

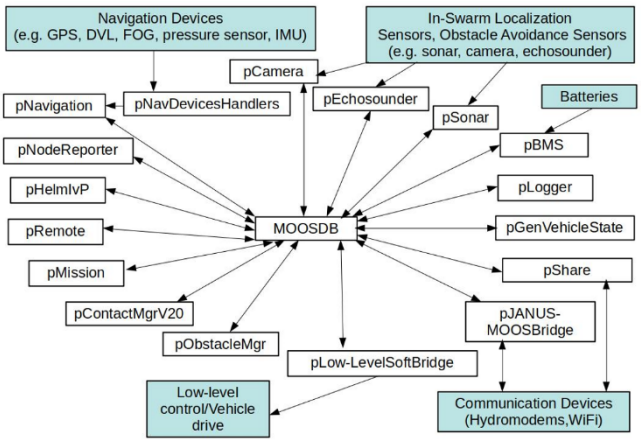

Figure 1. Vehicle-side software architecture

2.1 *MOOSDB*

As already mentioned, MOOSDB is a communication medium between other MOOS IvP applications. It stores information for other applications in the form of double or string variables. Each variable has a unique name for identification and the content. Other applications can be both producers and consumers of the information – they can produce some information by directly writing/updating/sending to MOOSDB a value of named variable (operation Notify in C++ code of an application) or consume information by subscribing for a selected variables (operation Register in C++ code of an application). Subscription for a variable means that a subscribing application is periodically fed with, say, mails including a list of variable-name, variable-value pairs.

2.2 *pJANUS-MOOSBridge*

This application handles acoustic communication link from/to the vehicle. It reads/writes messages from/to hydro-modems via JANUS intermediate layer of the acoustic communication system. The messages which are to be sent (selected MOOSDB variables) are appropriately encoded in the form previously defined data structures and then passed on to the JANUS layer. In turn, messages received from JANUS are decoded from the structures and then send to MOOSDB as variables.

There are at least two variables sent from the vehicle to the shoreside or the leader vehicle , i.e. NODE_REPORT_LOCAL / NODE_REPORT and REPORT_APP_LOCAL / REPORT_APP. The first variable stores complete information about vehicle including such parameters as: (x,y), (latitude,longitude), depth, speed and other available parameters. All the parameters are encoded as

parameter-name/parameter-value pairs, converted into strings and combined. In turn, REPORT_APP_LOCAL / REPORT_APP store information about state of all key applications in MOOS community which is run on main vehicle computer. The state of each app is expressed in the form of a single integer which can be then converted into binary value.

NODE_REPORT_LOCAL / REPORT_APP_LOCAL are variables relating to the own vehicle whereas NODE_REPORT / REPORT_APP are in fact NODE_REPORT_LOCAL / REPORT_APP_LOCAL variables read by the shoreside and renamed into NODE_REPORT / REPORT_APP.

2.3 *pShare*

The difference between pJANUS-MOOSBridge and pShare is a communication channel - pJANUS-MOOSBridge uses acoustic underwater channel whereas pShare works on the surface and it uses WiFi channel. In effect, pShare can send more information in shorter time than pJANUS-MOOSBridge. For that reason, pShare, in addition to NODE_REPORT and REPORT_APP sends also APPCAST variable which stores detailed state of a selected MOOS IvP app. This app has to be indicated by APPCAST_REQ variable which is sent from the shoreside by an appcast viewer, e.g. pMarineViewer.

2.4 *pLow-LevelSoftBridge*

This app is meant for communication with low-level control software which according to assumptions made in the project works on other computer than high-level control system or even on micro-controller. The task of pLow-LevelSoftBridge is to send desired motion parameters of the vehicle stored in variables DESIRED_HEADING, DESIRED_SPEED and DESIRED_DEPTH to low-level software whose task is to convert the parameters into commands for a vehicle drive.

2.5 *pMission*

This app manages missions of the vehicles, that is, starts and stops mission, and sets parameters of the mission such as: sequence of waypoints, march speed, type of formation, distances between vehicles, depth, security region, max depth, max distance. Moreover, it also sends to MOOSDB a report with mission status, that is, whether the mission is ready, run, or stopped. All the information about the mission which is processed (parsed) by pMission is sent in MISSION variable which is a string with parameter-name/parameter-value pairs separated with commas.

Each time before the mission is started pMission examines the state of the vehicle. If the state makes it possible to start the mission, it is started and state of the vehicle is appropriately updated through changing MOOS variables which collectively determine the state. Otherwise, a message is sent to the shoreside with the information about reasons of start mission fail.

2.6 *pRemote*

This app is a counterpart of pMission with respect to variable REMOTE-, and remote-control operation mode. pRemote reads REMOTE variable from MOOSDB, parses it, and sends appropriate commands (MOOS variables) to low-level control or activates vehicle behavior responsible for moving the vehicle to a point indicated in REMOTE variable.

Like pMission, also pRemote examines the state of the vehicle before activating a remote-control action. Activation or refusal of the action is associated with changing the state of the vehicle or a message to the shoreside with the information about reasons of remote-control action fail.

2.7 *pHelmIvP*

This is a standard MOOS IvP app which is meant for making decisions in autonomous mode. The key element of pHelmIvP is a set of behaviors which are run if the vehicle is in an operational mode assigned to each behavior. Behaviors activated in the same mode generate vehicle actions which are reduced by pHelmIvP (the so-called IvP Solver which is a component of pHelmIvP) to a single action through a process of behavior reconciliation. The same behaviors can have different instances activated in different vehicle modes.

The list of behaviors used by pHelmIvP includes both behaviors available in MOOS IvP and behaviors specific for SABUVIS II project:

1. OpRegion behavior – this is a standard MOOS IvP behavior which provides different safety functionalities, that is: (i) “do not go beyond a region” specified by a convex polygon being the parameter of the behavior, (ii) “do not operate longer than” a time limit specified in a behavior parameter, (iii) “do not go down deeper than” a depth limit specified in a behavior parameter. If some safety parameters are exceeded by the vehicle, pHelmIvp enters the mode which stops the vehicle.
   OpRegion behavior will be active during the whole operation of each vehicle. The parameters of the behavior will be determined at the very beginning of the mission by the operator and they will remain constant.
2. ConstantDepth behavior – this is also a standard MOOS IvP behavior which as its name implies cares about keeping the vehicle on a constant depth which is a parameter of the behavior. Like OpRegion behavior also ConstantDepth one will be active during the whole vehicle mission. This time, however, parameters of the behavior can change, depending on the swarm trajectory defined by the operator.
3. MaxDepth behavior – this standard MOOS IvP behavior which does not “deactivate” pHelmIvP like OpRegion. It keeps the vehicle above a threshold which is a parameter of the behavior. Like the above behaviors, also MaxDepth will be

active during the whole vehicle mission for one invariable depth threshold.

4. AvoidCollision behavior – this is a standard MOOS IvP behavior which controls vehicle heading in case of an obstacle on the vehicle way. Maneuvers of the vehicle are made only on the horizontal plane. The obstacles are in the form of convex polygons which group obstacle point detected by vehicle sensors like sonar. AvoidCollision behavior will have to be used very carefully because in the swarm there will be a risk to mistake other swarm members for obstacles.
   This behavior will be used on the vehicles if sensors are able to differentiate other vehicles from true obstacles.
5. AvoidCollisionUpDown behavior – this is also collision avoidance behavior which however controls depth of the vehicle. It is assumed the collision avoidance behavior will be made by three different behaviors, i.e. ConstantDepth, AvoidCollision, and AvoidCollisionUpDown. The first behavior will keep the vehicle close to one predefined depth. If an obstacle is detected on the way, at the safe distance from the vehicle, AvoidCollisionUpDown is activated which forces the vehicle to decrease depth (go up). If the vehicle passes over the obstacle, ConstantDepth will "pull" the vehicle back to the desired depth. However, if the obstacle is close to the vehicle, for example, it reaches the surface and, in consequence, it cannot be avoided by "jumping" over it, then AvoidCollision is activated.
   Like AvoidCollision also this behavior will be used on the vehicles if sensors are able to differentiate other vehicles from true obstacles. AvoidCollisionUpDown behavior is a new behavior which will be implemented within the project.
6. KeepFormation behavior – this behavior is the main objective of the project. The task of this behavior is to keep a follower vehicle in the swarm. Its implementation will depend on the specific swarm concept. For example, keeping the formation by following the leader in the so-called fixed formation will be require a different algorithm than long cloud formation. The task of this behavior will be to control heading and speed of the vehicle based on the information derived from vehicle sensors and the distance to the leader acquired via communication system.
   As already mentioned, this behavior will be only applied on follower vehicles.
7. NewWayPoint behavior – this behavior will be a modification of a standard MOOS IvP behavior meant for leading the vehicle along a path defined by a sequence of (x,y) way-points. In contrast to the standard behavior, NewWayPoint will take into account the sea current which can push the vehicle away from the predefined path. In order to avoid such situation, NewWayPoint will appropriately adjust vehicle heading and speed to the distance to the path. This behavior will be applied mainly on the leader vehicles, although, follower vehicles will be also equipped with this behavior.

2.8 *pNodeReporter*

pNodeReporter is a core MOOS IvP app which is meant for generating NODE_REPORT_LOCAL variable storing all the key information about the state of the vehicle. Each piece of the information is in the form of parameter-name/parameter-value pair encoded as a string. All the pieces are combined into a single variable by separating them with commas. NODE_REPORT_LOCAL should be read by all communication apps, i.e. pShare and pJANUS-MOOSBridge, and sent as NODE_REPORT variables to external nodes. pShare reports are sent directly to the shoreside (when on the surface), whereas, pJANUS-MOOSBridge reports are sent either to the shoreside (leader) or to the leader (followers).

Example NODE_REPORT_LOCAL is given in Figure 2.

```
NODE_REPORT_LOCAL = "NAME=alpha,TYPE=UUV,TIME=1252348077.59,X=51.71,Y=-35.50,
                    LAT=43.824981,LON=-
70.329755,SPD=2.00,HDG=118.85,YAW=118.84754,
                    DEP=4.63,LENGTH=3.8,MODE=MODE@ACTIVE:LOITERING"
```

Figure 2. Example NODE_REPORT

2.9 *pNavigation*

This app will be responsible for fixing navigation parameters of the vehicle including (x,y) position, latitude, longitude, depth, heading, pitch, roll, and speed. The parameters should be published in MOOSDB as appropriate variables, e.g. NAV_X, NAV_Y, NAV_DEPTH, NAV_HEADING, NAV_SPEED, NAV_LAT, NAV_LON, NAV_ROLL, NAV_PITCH. To fix the parameters, pNavigation will be fed with different navigation devices and sensors like IMU, DVL, FOG, pressure sensor, GNSS, satellite compass.

pNavigation will work according to different algorithms. The simplest of them will be Extended Kalman Filter. It will be also possible to equip follower vehicles with deep learning navigation system in which the position and spatial orientation of the vehicle will be produced by deep learning neural network supplied with the set of IMUs.

2.10 *pNavDevicesHandlers*

This is a single app or a set of apps. Their task is to provide information generated by different navigation devices/sensors to pNavigation. For the purpose of high speed of processing in pNavigation, the communication between pNavDevicesHandlers will be performed without participation of MOOSDB as a communication medium. To this end, communication mechanisms provided by operating system, for example, sockets will be applied.

2.11 *pCamera*

The task of this app is to handle a camera and to provide the whole system the information about noticed objects such as other vehicles and obstacles. This app is crucial for collision avoidance and keeping formation behaviors. In order to fulfill the task,

pCamera has to implement video processing algorithms which should be able to detect and classify visible objects. Since the algorithms require time-consuming calculations, the implementation of pCamera should consider processing on GPU.

pCamera should publish two variables, i.e. TRACKED_FEATURE and NODE_REPORT. The first variable corresponds to noticed obstacles (position of obstacle in local coordinate system) whereas the second variable corresponds to other vehicles detected.

2.12 *pEchosounder*

This is an app for handling echosounder(s) and for providing the system the information about distance to the bottom and possible objects above the vehicle.

2.13 *pSonar*

This app is a counterpart of pCamera, however, with respect to sonar which is the sensor of longer range than camera which makes it the primary sensor when navigating in the swarm. Like pCamera, pSonar has to be able to detect and classify objects visible in sonar images. What is more, it has to able to differentiate echoes coming from the sea bottom and sea surface. In order for pSonar to be able to fulfill its tasks, it has to implement advanced video stream processing algorithms on GPU.

Like pCamera, also pSonar should publish two variables, i.e. TRACKED_FEATURE and NODE_REPORT. The first variable corresponds to noticed obstacles (position of obstacle in local coordinate system) whereas the second variable corresponds to other vehicles detected.

2.14 *pBMS*

This is an app for handling batteries. It has to provide information about all faults, analyze the situation regarding energy consumption, and notify the system if there is not enough energy to continue the mission. The mentioned information should be published to MOOSDB in the variable REPORT_BMS.

2.15 *pLogger*

This standard MOOS IvP app is meant for recording selected publications to MOOSDB during the whole MOOS session. It is an app essential from the point of view of post-session analysis. The app is configurable, that is, it allows for synchronous and asynchronous logging and selection of MOOS variables which are to be logged.

2.16 *pGenVehicleState*

It is crucial app for safety of the vehicle. It is assumed that selected MOOS apps (the ones which are necessary for the vehicle to operate) periodically publish their state to MOOSDB in the form of integer variable REPORT_(APP) where APP is a label of the app. pGenVehicleState monitors the state of all apps, generates one cumulative report (REPORT_APP_LOCAL) for the shoreside (state of all apps plus current action performed by the vehicle, e.g. remote control: turn right, mission, returning to a specified point, waiting for orders, breakdown) and appropriately reacts if the state of the vehicle is insufficient with respect to performed mission/action.

2.17 *pContactMgrV20*

This is a standard MOOS IvP app whose task is to generate alerts regarding other vehicles visible around. The information about these vehicles is given in NODE_REPORT variable which can be published by vehicle sensors (pCamera, pSonar) and communication apps (pShare, pJANUS-MOOSBridge). The alerts are then read by appropriate behaviors (AvoidCollision, AvoidCollisionUpDown, and KeepFormation) which adjust operation of the vehicle to the situation around.

2.18 *pObstacleMgr*

This is a counterpart of pContactMgrV20 with regard to motionless obstacles reported in variable TRACKED_FEATURE. In addition to alerts which indicate parameters of observed obstacles, and which are then used by appropriate vehicle behaviors, the task of pObstacleMgr is also to compress information provided by a sequence of TRACKED_FEATURE publications. pObstacleMgr reads positions of the obstacles and builds convex polygons for each obstacle. This way, the behaviors do not deal with sequences of points but with compressed objects.

3 DESIGN OF HIGH-LEVEL CONTROL SYSTEM INCLUDING DECK UNIT SOFTWARE COMPONENTS

The shoreside architecture is depicted in Figure \ref{fig:moosBrzeg}. It consists of nine MOOS IvP applications, i.e.: MOOSDB, pJANUS-MOOSBridge, pJoystick, pMVEventHandler, pVehicleStateViewer, pMarineViewer, uXMS, uPokeDB, pShare. All the applications are outlined below.

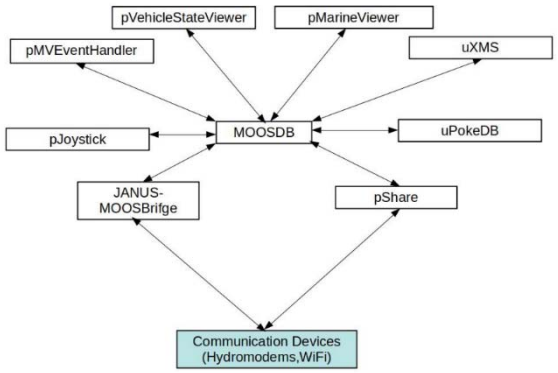

Figure 3. Shoreside software architecture

3.1 *MOOSDB*

MOOSDB application is already described in section 2.1 MOOSDB

3.2 *pJANUS-MOOSBridge*

In order to handle communication with a vehicle pJANUS-MOOSBridge application has to know which MOOS variables are to be sent and received. Outgoing messages relate to MOOS variables subscribed by pJANUS-MOOSBridge application. There are three such variables, i.e. REMOTE, MISSION, and NAVIGATION. All of them are string variables including a sequence of parameters encoded as strings and separated by commas.

REMOTE variable includes parameters for remote-control, e.g. turn right, speed up, go to the point (10,10), heading 90 deg., speed 1 m/s. MISSION variable includes actions to perform regarding vehicle mission, i.e. start or stop mission, or it defines parameters of mission, e.g. safety parameters (max depth, max distance to cover, accepted operational area), type of swarm, distances between individuals, path (for swarm leader), communication intervals (for swarm leader). And, NAVIGATION variable includes parameters for navigation system of each vehicle. The key parameter is the origin of the coordinate system, the same for all vehicles – geographical coordinates in the form of latitude and longitude.

In addition to outgoing MOOS variables, pJANUS-MOOSBridge has to know incoming variables. They will be encoded in appropriate data structures and sent in a compressed form. The name of the variable will not be encoded as a string but as an integer. Meanwhile, MOOSDB needs the name of the variable not an integer. In consequence, pJANUS-MOOSBridge needs mapping between integers and variable names.

3.3 *pShare*

This application is a counterpart of pJANUS-MOOSBridge, however, with respect to WiFi communication link. It is a standard MOOS IvP app which needs to be configured in a ".moos" configuration file which defines the whole MOOS community – each app belonging to the community is defined in the file by a list of configurable app parameters with their default values. pShare has two parameters, i.e. configuration of input channel (number of port used by pShare for input and multicast number if used) and a list of output MOOS variables with assigned parameters of output channel to each of them (IP and the number of port of pShare working on a vehicle). Detailed specification of pShare can be found in [2].

In addition to REMOTE, MISSION, and NAVIGATION variables, pShare will also send APPCAST_REQ variable. This variable is generated by each appcast viewer application like pMarineViewer and it is directed to selected appcast apps with request to send the so-called appcast reports which include the state of the app in the form of appropriately formed string.

3.4 *pJoystick*

This application is meant to handle Joystick. In order for pJoystick to start working, it has to be intentionally activated by sending REMOTE_CONF message with activation string as a variable value, e.g. REMOTE_CONF="activate". The same applies to stopping the app. In this case, REMOTE_CONF should take deactivating value, e.g. REMOTE_CONF="deactivate". Once activated, pJoystick reads messages sent by Joystick, converts them into REMOTE messages, and finally sends them to MOOSDB using Notify operation. After receiving new value of REMOTE, MOOSDB sends it to pShare and pJANUS-MOOSBridge which send it further to the vehicle.

3.5 *pMVEventHandler*

This app is intended to handle events generated by Deck Unit app called pMarineViewer [3] (see further). pMarineViever is a standard MOOS IvP app which generates three types of events, i.e.: (i) pressing one of four buttons – each button has variable-name/variable-value pair assigned in ".moos" configuration file, (ii) changing app context by clocking an option in the menu, (iii) pressing a mouse button. pMVEventHandler will cooperate with pMarineViewer by intercepting events generated by the latter app and by sending appropriate orders to pMarineViewer to display objects on the map which relate to the events. For example, if pMarineViewer is in generating path for the swarm mode (an appropriate menu option) then each mouse click on the map will be received by pMVEventHandler which will instruct pMarineViewer to draw a way-point in the clicked position and to link it with neighboring way-points, through drawing a line. After completing path definition, a button on pMarineViewer can be pressed which should run pMVEventHandler window aimed at specifying other mission parameters, e.g. depth, distance for each vehicle, safety region.

The whole functionality of pMVEventHandler will be strongly linked to pMarineViewer functionality which in turn will depend on tasks which the entire Deck Unit will have to perform. At the point of writing this paper, three main functionalities have been identified, i.e. mission definition (path consisting of a sequence of way-points), monitoring of swarm (individual vehicles) behavior on the map, monitoring of MOOS IvP appcast applications. Only the first functionality will require pMVEventHandler reaction.

3.6 *pVehicleStateViewer*

The task of this app is to view the state of each vehicle. This state will be defined in a number of MOOS variables. In fact, pVehicleStateViewer will do the same what pMarineViewer can do, without viewing geoinformation. The idea is however, to organize the view in a more convenient way than it is done by pMarineViewer. That is, a user will be able to choose vehicle and MOOS variable describing its state. Then, all the parameters stored in the variable will be displayed in a compact form. What is more,

the task of pVehicleStateViewer will be also to generate log files for each vehicle. Each log will include selected parameters from different variables. pVehicleStateViewer will be configured either in a ".moos" or a separate XML file.

3.7 *pMarineViewer*

This app has already been mentioned above. It is a standard MOOS app which has two main tasks. First, it is the only app which is able to process geoinformation, that is, to define way-points determining the desired path of the swarm, and to monitor position of each vehicle. Second, it is also meant for monitoring different MOOS variables and state of MOOS apps. Example widow of pMarineViewer is depicted in Figure 4.

An additional functionality of pMarineViewer is to notify MOOSDB about new values of variables defined in ".moos" configuration file. It means that if we know expected values of the variables in advance, we can assign variable-name/variable-value pairs to selected action components (different buttons, menu) of pMarineViewer and activate them at any moment.

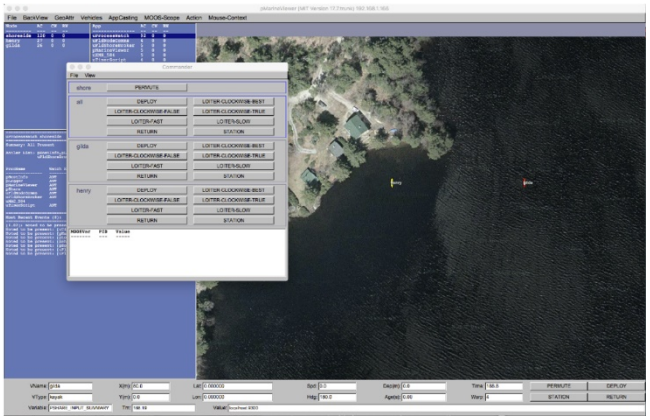

Figure 4. Example widow of pMarineViewer app [3]

3.8 *uXMS*

This is a standard terminal-based MOOS app for viewing the content of MOOSDB. The content can be displayed in different manner. The user can choose variables to show, can show history of variable updates or can show variables published by a selected app. Example window of uXMS is depicted in Figure 5.

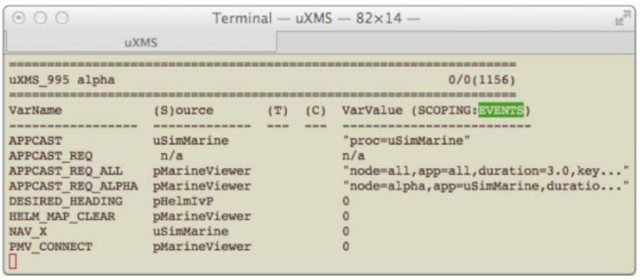

Figure 5. Example window of uXMS app [4]

3.9 *uPokeDB*

This is a terminal-based app for poking MOOSDB, that is, for updating the content of MOOSDB. To this end, one indicates MOOS community (".moos" file), and a list of variable-name/variable-value pairs.

uPokeDB is a supporting app for pMVEventHandler and pMarineViewer which is able to set values of variables which are not handled by the two latter apps.

4 SUMMARY

The paper presents the software architecture for underwater vehicles operating in a swarm. The proposed architecture is consistent with the MOOS-IvP approach in which we are dealing with many applications cooperating with each other using one medium, which is the MOOSDB application.

On the vehicle side, the architecture includes applications responsible for the following functionalities: handling navigation and observation devices and sensors, handling the battery system, generating logs, communication, high and low-level control including swarm control, remote control, handling emergency situations, generation of messages about the status of the vehicle. In turn, on the shoreside, the architecture includes applications responsible for the following functionalities: communication, monitoring the status of vehicles, generating output messages and commands.

REFERENCES

[1] Link: https://oceanai.mit.edu/moos-ivp/pmwiki/pmwiki.php?n=Main.HomePage
[2] Link: www.moos-ivp.org : Manifest - P Share browse (mit.edu)
[3] Link: https://oceanai.mit.edu/ivpman/pmwiki/pmwiki.php?n=IvPTools.PMViewer
[4] Link: https://oceanai.mit.edu/ivpman/pmwiki/pmwiki.php?n=IvPTools.UXMS