A Mission Programming System for an Autonomous Sailboat

José C. Alves
Faculty of Electrical and Computer Engineering
University of Porto / INESC TEC
Porto, Portugal
Email: jca@fe.up.pt

Nuno A. Cruz
Faculty of Electrical and Computer Engineering
University of Porto / INESC TEC
Porto, Portugal
Email: nacruz@fe.up.pt

***Abstract*—Robotic sailing vehicles can provide the support for carrying out long ocean sampling missions, using solely renewable energy for propulsion and for powering the computing, communication and electromechanical systems. The basic automatic sailing tasks required to visit a sequence of waypoints has already been correctly addressed by various teams. However, an effective system for specifying long term autonomous missions is necessary to fill the gap between the developers of the robotic platforms and the scientific end users, mainly interested in the data they can get. This paper presents a simple, flexible and easy to use mission programming system implemented in the autonomous sailing boat FASt. A mission is programmed by defining events and assigning to them sequences of high level actions. The support of conditional statements allows the implementation of a basic control flow mechanism to make route decisions during the deployment of the mission. Examples are presented to illustrate the construction of mission programs.**

I. Introduction

Autonomous sailboats have gained particular attention in the last few years for their unique ability to maintain long term unassisted operations in the sea surface [1]. Using the wind as the only source of energy for propulsion, the low power requirements of the onboard computing systems can be guaranteed by renewable energy sources like photovoltaic panels or wind turbines. The superior navigation capability of sailboats in terms of speed and maneuverability, when compared to other long endurance surface vessels like wave gliders [2], makes them a valuable asset for ocean sensing and sampling tasks requiring a continuous presence at the sea surface. In addition, the low level of self-generated noise is a relevant feature of autonomous sailing boats for underwater acoustics applications like tracking marine mammals [3],[4].

A few recent successful projects have demonstrated the capability of robotic autonomous sailboats for performing autonomous missions in the ocean and even survive to harsh conditions. The Saildrone project is the latest known mature design, demonstrated in the end of 2013 with a 100 day autonomous mission in the Pacific ocean, and an announced plan to try a circumnavigation [5]. Other projects of small sea worth autonomous sailing robots have also been developed in Europe, as the Austrian ASV Roboat exploited in research on marine mammals [6], the BeagleB project of the University of Aberystwyth, United Kingdom [7], the Vaimos sailboat from ENSTA Bretagne, France [8] and the Portuguese FASt, addressed by this work [9].

To be effective for accomplishing fully autonomous missions, robotic sailboats must be able to complement their robustness and navigation proficiency with convenient support tools for an easy setup, programming and monitoring of missions. Closing this gap is crucial to enable their exploitation by the typical end users that usually are not aware of the supporting robotic technologies.

Defining in advance a fully autonomous mission for a robotic sailboat can easily go beyond the simple programming of a list of waypoints to visit. In addition to perform a certain pre-programmed course, a sailing robot also has to execute specific tasks at precise locations, to gather data from the onboard sensing devices or collect physical samples of the air/water environment. The implementation of complex mission tasks may also require a close interaction between the sampling processes and the sailboat's navigation system, to dynamically adapt the course plan to varying and unpredicted conditions. This is particularly important for autonomous sailboats, which have their navigation ability greatly influenced by the surrounding environmental conditions.

In this paper we present the mission programming environment developed for the FASt autonomous sailboat. This is based on a simple and intuitive textual language for specifying a mission by defining key waypoints, events and actions. The mission execution environment provides access to a set of relevant variables shared with the sailboat's control and navigation system to allow creating missions that interact with the sailboat. The support of a basic set of conditional instructions allows the implementation of a simple, albeit flexible, control flow mechanism to dynamically control the course and the sequence of mission tasks. This mission programming tool has been developed as part of the METASail command and control console, an integrated system for mission planning, supervision and analysis, described in detail in [10].

Besides this introduction, the paper is organized as follows. Section II presents related work on mission programming systems for robotic marine vehicles. In section III we introduce our robotic sailing boat FASt that is the target of the system presented in the paper. Section IV presents the mission programming concept and the related execution model. The mission programming language is detailed in section V and in section VI we present a set of selected examples that illustrate the application of the main features of the language. Finally section VII draws the final conclusions and finalizes the paper.

II. Related work

The problem of planning complex missions for unmanned marine vehicles has been addressed by other teams, with features adapted to specific requirements of their vehicles and systems. The absence of a common framework to support all the features needed by each project has motivated the development of proprietary systems, difficult to reuse in different scenarios. All the mission programming systems known were designed for motorized surface or underwater vehicles and do not provide support for navigation tasks and constraints associated to sailing.

In the case of underwater vehicles, almost all communication systems rely on acoustics, that enable long range communications but with a high latency and a low data rate. For that reason, there has been particular interest in the development of specific compact languages for mission command and control [11] to minimize the communications requirements.

Neptus [12] is a modular mission planning, simulation and analysis framework designed for managing the cooperative deployment of heterogeneous surface and underwater vehicles, autonomous or remotely operated. Missions are created with various types of pre-built basic maneuvers and conditions to trigger transitions between maneuvers.

Another system is the ROAZ mission control [13], developed for a specific electric autonomous surface vehicle. Missions are specified as XML files by defining basic maneuvers with assigned actions and test functions to evaluate conditions.

The DELFIM mission control system [14] is an interactive tool for designing and supervising missions of motorized surface vehicles using an execution model based on Petri nets. A mission is designed with a graphics interface and rely on vehicle primitives (e.g. maintain a heading) to build mission procedures (follow a path) and construct mission programs with them.

The MOOS-IvP system [15] is an open source project that provides a software infrastructure for building autonomous vehicles and program complex missions by instantiating behaviors that also include conditions. The MOOS-IvP software is designed to run in a dedicated computer for handling the autonomy decisions, receiving the vehicle position and other data from the onboard computer performing the low level control.

III. The FASt Autonomous Sailboat

FASt is a $2.5\,\mathrm{m}$ LOA, $50\,\mathrm{Kg}$, autonomous unmanned sailing boat designed and built at the University of Porto, Portugal (figure 1). The sailboat has a highly configurable low power computing system running an embedded version of the Linux operating system (uCLinux) and an onboard local area network easily permits to expand the computing power or to connect other network-enabled devices. The payload capacity, either in terms of weight and dry space inside the hull (a few kilograms and liters), allows a flexible mission-specific customization. The sailboat also has an electric winch that can be controlled to wind out a desired length of line for deploying equipment at precise depths.

Electric power is provided by a $45\,\mathrm{Wp}$ solar panel and a set of Li-Ion batteries with a total capacity of $194\,\mathrm{Wh}$. The onboard sensors provide the wind direction and speed, the heading, pitch and roll angles, the geographic position, course and speed over ground and the angle of the boom. A WiFi link is used for remote access to the onboard computer, mainly to upload mission programs and for debugging purposes.

The sailboat can be controlled manually with a conventional radio-control, that is useful for the launch and recovery operations. The autonomous mission starts as soon as the radio-control is switched off and can be interrupted to return temporarily to manual control by switching on the radio control.

Fig. 1. The FASt autonomous sailing boat (LOA $2.5\,\mathrm{m}$).

IV. The mission programming model

The mission programming model is based on executing sequences of actions triggered by events detected during the accomplishment of a mission. A basic event is the arrival to a given destination waypoint, and the most primitive action to perform is setting a new destination point. While this behavior is sufficient to program a path along a set of waypoints for a robotic marine vessel, it does not allow to define other tasks or, more importantly, to program route decisions depending on acquired data.

We also extended the mission programming model by creating a mission execution environment that allows to execute arbitrary tasks synchronized to navigation related events, timing events, or conditions associated to data acquired from sensors.

The programming environment uses a set of variables or *registers* (resembling the configuration and status registers of micro-controllers) shared with the mission execution and navigation system, to retrieve and evaluate the status of the sailboat navigation system and of the external sampling devices.

Events trigger the execution of actions. There are three types of events triggered by different conditions. The *initial event* defines the list of tasks to be accomplished when the autonomous mission starts. The *waypoint arrival* event happens when the target waypoint is reached. The *asynchronous event*

is defined as a condition set with the status registers that is continuously evaluated during the operation of the sailboat.

The mission execution process works as a stack machine, allowing to create mission programs that implement a call-return behavior. When an event is triggered (e.g. waypoint reached) the list of actions assigned to that event is pushed into the stack. Actions are popped sequentially from the stack and executed until no more actions exist and the mission program terminates. The actual behavior performed by the sailboat upon completion of a mission is determined by the last action programmed. If a new destination is not defined, the sailboat will maintain as target the last target waypoint, performing a station keeping like behavior.

V. Mission Programming Language

To ease the process of specifying missions we developed a simple programming language with an easy-to-learn and intuitive set of statements. A mission program can be created as a plain text file and can also be edited with an interactive graphics tool described in [10]. Such a program includes global definitions, the specification of events and associated tasks.

A. Global definitions

A mission program starts by defining a list of the relevant static waypoints and the specification of the geographic limits of the safe region allowed for the accomplishment of the mission. Either the target waypoints and the safe area points can be defined as lat/lon coordinates or as a displacement and bearing relative to any other waypoint. Figure 2 shows an example of waypoint and safe area definitions and the corresponding view in the graphics mission editor.

An extensive set of configuration variables used by the autonomous sailboat control and navigation system may also be globally defined to supersede default values defined in a startup configuration file. Examples are parameters that control the way some sailing maneuvers are performed, as for example the minimum beating angle, the width of a corridor to sail upwind legs or the minimum distance to reach a target waypoint.

B. Shared register file

The interaction between the mission program and the FASt's control and navigation system is done with a set of variables seen from the program point of view as an array (shared register file, figure 3). All the variables are 32-bit integers and are organized into five sections: timers, clocks, counters, data read from the sailboat's sensors and also variables gathered from other applications running in the sailboat's computer (as, for example, data acquisition programs that interface to external sampling devices).

Timers and clocks count seconds down to zero and up, respectively[1] and counters are set, incremented or decremented with specific statements included in the action lists. The status variables hold the current values of internal FASt variables, including data read from navigation sensors, the status of the actuators and also other internal variables relevant to the

[1]Counting up seconds in 32 bits only overflows after 136 years.

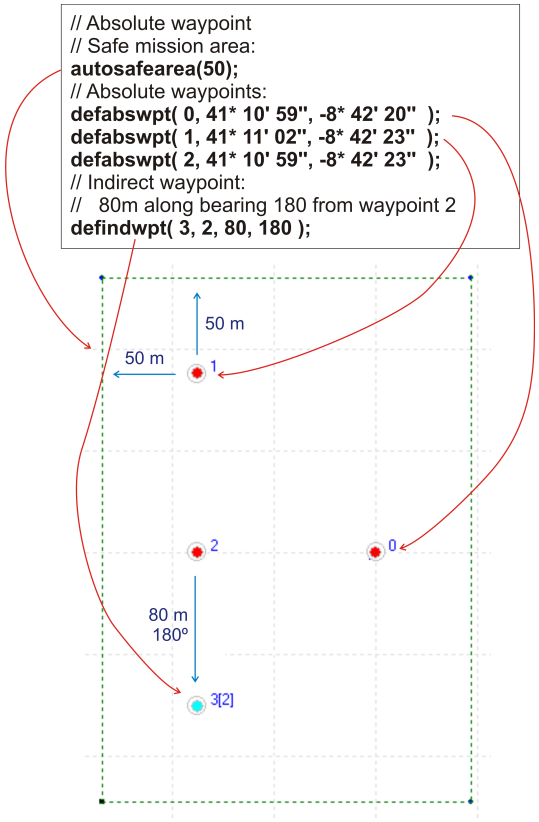


Fig. 2. Definition of waypoints and an automatic safe navigation area set by expanding the bounding box of the waypoint by 50 m.

knowledge of the state of the sailboat control and navigation system. A section of the shared register file is reserved to receive data from external programs. Any of these registers can be used to set conditional expressions that implement the control flow mechanism in the mission programming system and are referred with the identifiers shown in figure 3.

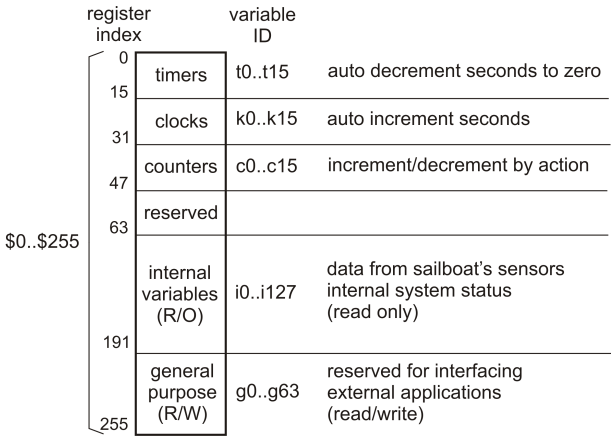


Fig. 3. The organization of the shared register file visible to the mission programming environment.

C. Conditions

One important feature of the mission programming system is the ability to specify missions with a control flow mechanism. This is done with the establishment of conditions that can be used in various contexts in a mission program, as detailed in the next section. Conditions are created with single relational operators between a shared register file variable and a constant, or between any two variables, as exemplified in figure 4.

```
// If clock 0 (register $16) is less-than 200 seconds
//   goto waypoint 1, else goto waypoint 2
... if( k0.lt.200, 1, 2 ) ...
// If internal register 13 ($77) equals the general
//    purpose register 21 ($213), goto waypoint 3
... if( i13.eq.g21, 3) ...
```

Fig. 4. Setting conditions for implementing the control flow mechanism.

D. Events and actions

As stated before, a mission program is created with lists of actions assigned to trigger events. When a trigger event occurs, the statements (or actions) defined for that event are executed sequentially. There are three types of trigger events currently supported. The `initial` event is automatically triggered when the autonomous mission starts and thus represents the mission program entry point. The waypoint event is assigned to a previously defined waypoint and triggers when that waypoint is reached. The asynchronous event allows to program a behavior similar to an interrupt handling system in microprocessors by setting a trigger condition that is continuously evaluated, and the associated list of actions to execute when the condition is evaluated to true (comparable to an *interrupt handler*). To control the way the trigger events are processed, they can be individually or globally enabled or disabled.

Figure 5 presents an example of a simple mission program consisting in starting to sail to waypoint 0 and then circling forever along the waypoints 1, 2, 0. The example in figure 6 adds a counter and a conditional statement to execute 5 laps around the waypoints 0, 1 and 2 and then exit to waypoint 3.

```
// Navigation program entry point,
// Sail to waypoint 0
@initial{ goto( 0 ); }
// When reaching waypoint 0, sail to waypoint 1
@0{ goto(1); }
@1{ goto(2); }
// Return to waypoint 0
@2{ goto(0); }
```

Fig. 5. A simple mission program. This example assumes the definitions of 3 waypoints with IDs 0, 1 and 2.

The execution of a statement (or action) can block the execution of the forthcoming actions until it concludes (*blocking statements*) or the execution can proceed immediately to the next statement (*non-blocking statements*). The blocking statements are those that specify navigation-related tasks or other actions that require a non-deterministic time to complete, as, for example, executing a sampling process.

The mission execution system implemented in the sailboat accepts a basic set of *internal* statements that is sufficient for describing complex missions in a very flexible way (table I). In addition to these, additional *external* actions can be easily executed by the mission execution system, specified as shell command-line instructions. External actions are performed by programs cross-compiled for the FASt main computer and adhering to a code template that include primitives for receiving arguments and returning results via UDP sockets. The external program requests input arguments from the status register array and upon completion provides return results (if any) that are placed in register locations enabled for writing. External actions can be invoked in blocking or non-blocking execution mode depending on the relevance of its completion for the continuation of the execution of the mission program. An example of non-blocking external action is a program that drives an external data logger to start acquiring data while the rest of the mission proceeds. On the other hand, an example of a blocking external action is the execution of a data acquisition or analysis process whose results are necessary to decide what to do next.

```
// Set counter c0 with value 0, sail to waypoint 0:
@initial{ setcounter(c0,0); goto( 0 ); }
// When reaching waypoint 0, increment counter 0
//    if counter 0 is less than 6, sail to waypoint 1
//    else exit to waypoint 3
@0{ inccounter(c0); if(c0.lt.6, 1, 3); }
@1{ goto(2); }
@2{ goto(0); }
```

Fig. 6. Adding a conditional statement to count the number of laps in the mission listed in figure 5.

E. Programming timeouts

The asynchronous event and the `abort()` statement allows to easily define global timeouts for any navigation-related statement. This only needs to set a timer at the beginning of the navigation command and program an asynchronous event to abort the current maneuver when the timer expires. Figure 7 exemplifies the utilization of this feature.

```
...
// Create asynchronous event (timeout on timer 3)
//   and assign it the ID 1 (disabled by default)
@when(1,t3.eq.0){ disablecond(1); abort( ); }
// Start the mission: sail to waypoint 1
@initial { goto(1); }
// At waypoint 1, sail to waypoint 4 within 1000 seconds
//   then go to waypoint 5
@1{ settimer(t3,1000); enablecond(1); \
      goto(4); disablecond(1); goto(5);  }
// At waypoint 4 disable the asynchronous event 1
//   and sail to waypoint 6
@4{ disablecond(1); goto(6); }
...
```

Fig. 7. Programming timeouts for navigation-related actions.

F. Complex maneuvers

A station keeping navigation is performed with the statement `stationkeep(R,cond)`. Whent this statement is executed, two waypoints are defined along a line perpendicular to the true wind direction, centered in the current position and separated by 80% of the diameter of the station keeping region

TABLE I. SUMMARY OF THE MISSION PROGRAMMING LANGUAGE.

Global definitions

`defabswpt(wpID,lat,lon[,"label"])`	Create an absolute waypoint with ID `wpID`. The optional argument `"label"` is a text string to display in the graphics console.
`defindwpt(wpID,refwpID,dist,angle[,"label"])`	Create an indirect waypoint with ID `wpID` obtained as the projection of `refwpID` along `dist` meters and `angle` degrees.
`defabssa(saID,lat,lon)`	Define an absolute point with ID `saID` to form the safe navigation area.
`defindsa(saID,refsaID,dist,angle)`	Create an indirect waypoint to define the safe area, with the ID `saID`, obtained as the projection of the safe area point `refsaID` along `dist` meters and `angle` degrees.
`defindsw(saID,refwpID,dist,angle)`	Create an indirect waypoint to define the safe area, with the ID `saID`, obtained as the projection of the waypoint `refwpID` along `dist` meters and `angle` degrees.
`autosafearea(dist)`	Define the safe area as a rectangle exploded by `dist` meters to the outside of the bounding box enclosing all the waypoints.
`setvar(varname,value)`	Set internal variable `varname` with value `value`. This statement can also be used as an action.

Trigger events

`@initial{actions...}`	Trigger the initial actions (this represents the mission program entry point).
`@wpID{actions...}`	Trigger upon arrival to the waypoint defined with the ID `wpID`.
`@when(condID, cond){actions...}`	Trigger when condition `cond` is true and assign the ID `condID` to this event.

Statements for specifying actions

`goto(wpID)`	Go to the waypoint defined by `wpID`.
`gotorel(D,brg)`	Go to a waypoint defined by the distance `D` along bearing `brg` relative to the current position.
`whilegoto(cond, wpID)`	While condition `cond` holds true, go to waypoint `wpID`.
`follow(rIDlat,rIDlon,cond)`	Go to the waypoint defined by the coordinates latitude/longitude read from registers `rIDlat` and `rIDlon`, while condition `cond` is true.
`stationkeep(R,cond)`	Perform a station keeping navigation pattern within a radius of `R` centered in the current position while condition `cond` is true.
`scanarea(D,wp1,...,wpn)`	Execute a navigation pattern to scan the area defined by the list of waypoints `wp1,...,wpn`, with parallel lines separated by `D` meters.
`hold([cond])`	Hold while `cond` is true by loosing sails and suspending the navigation control system. If the condition is not specified, the sailboat enters hold mode and the program continues to the next action. The hold mode exits with the execution of a navigation command (e.g. `goto()`).
`if(cond,true_wpID[,false_wpID])`	if condition `cond` is true, go to waypoint `true_wpID`, else (optional argument) go to waypoint `false_wpID`.
`settimer(tID,value)`	Set timer `tID` with `value`.
`setcounter(cID,value)`	Set counter `cID` with `value`.
`setclock(kID,value)`	Set clock `kID` with `value`.
`inccounter(cID)`	Increment counter `cID`.
`deccounter(cID)`	Decrement counter `cID`.
`setregister(rID,value)`	Set register `rID` with value `value`.
`enablecond(condID)`	Enable asynchronous evenb defined by `condID`.
`disablecond(condID)`	Disable asynchronous evenb defined by `condID`.
`setvar(varname,value)`	Set internal variable `varname` with value `value`.
`external("external_command", bmode)`	Start the external program `external_command` in blocking or non-blocking mode, according to the value of parameter `bmode`.
`abort()`	Abort current blocking action under execution.

(shown in figure 8 as the two red dots labeled 1 and 2 within the hatched circle representing the station keeping region). The station keeping navigation is performed by continuously sailing with side wind (reaching) and gibing at those waypoints, until the condition `cond` is evaluated to false (figure 8).

Conditions can be created with the timers or clocks to set a fixed time for the station keeping operation and the conditions can also evaluate results returned by external applications, to conclude the action under control of an external program (e.g. start a sampling process and conclude when receiving a completion acknowledge).

G. Scan area

Due to the wind constraints, a zig-zag lawn mower navigation pattern performed by a sailboat must be set according to the wind direction, to avoid doing upwind legs and guarantee the accuracy of the track. The mission programming system includes the statement `scanarea(...)` to automatically set a zig-zag navigation pattern for scanning an area, executing parallel legs separated by a given distance and perpendicular to the true wind direction. The area to scan is set as a closed polygon defined by a list of geographic marks. When the statement is executed, a temporary destination waypoint is set upwind the scan area region. Then, a list of waypoints is generated to navigate with side wind (reaching) along legs that cross the area of interest and force gibing at each end.

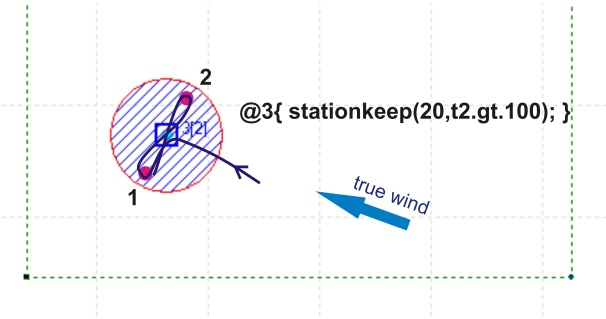


Fig. 8. Illustration of the implementation of station-keeping around waypoint 3, with a radius of 20 m, during 100 s. The waypoints 1 and 2 are generated automatically when the command is executed.

Figure 9 illustrates the implementation of the `scanarea(...)` action. The large red dots outside the scan area region (in green) indicate the approximate position of the waypoints generated automatically, according to the current true wind direction. The dark blue line shows the approximate path followed by the sailboat.

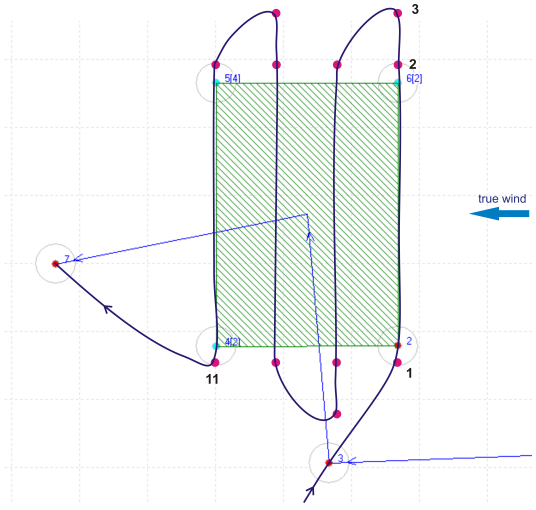


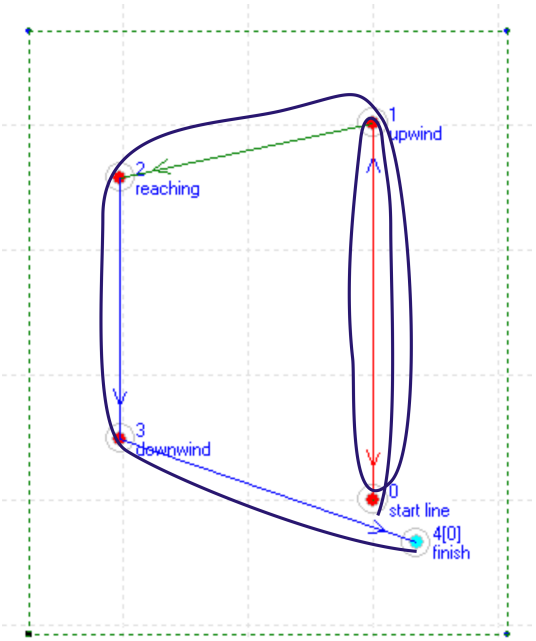
Fig. 9. Performing the scan area action in the area defined by the points 2, 6, 5 and 4, with a line separation of 10 m. The waypoints defined outside the shaded area are defined automatically to force the lawn mower pattern.

VI. Examples of Mission Programs

Besides the basic waypoint visiting mission shown in figure 5, this system allows to program complex missions that can interact with external systems or devices attached to the sailboat's computing system or local network, as illustrated in the next examples.

A. Regatta with a trapezoidal course

This scenario considers a trapezoidal course with 4 buoys (represented by the waypoints 0, 1, 2, 3 and 4), as usually set for regattas of sailing dinghies (figure 10). In this example we consider the course as visiting the waypoints in the order 0–1–0–1–2–3–4 rather than rounding them, as it is required in real sailing regattas. If the true wind speed raises above 20 knots the course is canceled and the sailing boat should return immediately to the finish point. If the course is not completed within the maximum time of 15 min, the sailboat should also go to the finish point.




```
autosafearea(30);
defabswpt(0,  41.6781425, -8.7073669, "start line");
defabswpt(1, 41.6792488, -8.7073669, "upwind");
defabswpt(2,  41.6790885, -8.7081069, "reaching");
defabswpt(3, 41.6783218, -8.7081069, "downwind");
defindwpt(4, 0, 20, 135, "finish");

@when( 1, t0.eq.0 ){  abort(); goto(4); }
// variable i7 has the averaged true wind speed
//     wind speed is in tens of knot:
@when( 2, i7.gt.200 ){ abort(); goto(4); }

// start mission, set the time limit to 15 minutes
@initial{ setcounter(c0,0); settimer(t0, 900); \
          enablecond(1); enablecond(2);       \
          goto(0); }
@0{ goto(1); }
@1{ inccounter(c0); if(c0.lt.2, 0, 2 ); }
@2{ goto(3); }
@3{ goto(4); }
// hold forever:
@4{ hold(t0.eq.t0); }
```

Fig. 10. A typical dinghy regatta course and a mission program that implements it.

B. Sampling data with external devices

We consider here that the sailboat is equipped with a sonar, a remotely controlled CTD attached to a winch and an autonomous underwater sound recorder. The CTD and the sound recorder start and stop the acquisition process with

the command-line shell scripts `ctd -start`, `ctd -stop`, `sr -start` and `sr -stop`. The winch operation is also controlled with an external software application that receives as argument the desired depth and is called from the mission program in blocking mode. The control of the sonar is integrated in the sailboat's software system and is started/stopped with the value written in the shared register variable `g9` (0:off, 1:on). Figure 11 presents a program that implements this mission.

The sailboat starts by going to point 1 to record the underwater sound for 2 minutes, while holding the navigation processes to avoid the noise generated by the electric actuators; then it performs a CTD profile in location 2: switch on the CTD, unwind the winch to the desired depth, recover the CTD and switch it off.

The sailboat continues to point 3 and starts scanning the area delimited by points 10, 11, 12, 13, 14 and 15 to perform a bathymetry log with $10\,\mathrm{m}$ of separation between the scan lines. This operation has to be completed within $2\,\mathrm{hours}$. At the end the sailboat must return to point 4 and hold forever.

```
// Set the condition to abort the scan area operation:
@when(1,t2.eq.0){ disablecond(1); abort(); }

@initial{ goto(1); }

// Switch on the sound recorder, hold for
//   two minutes, switch off the recorder
//   and continue to waypoint 2:
@1{ settimer(t1, 120); \
      external("sr -start", 0); \
      hold(t1.gt.0); \
      external("sr -stop",0); \
      goto(2); }

// Enter hold mode, start the CTD acquisition,
//   wind out the winch down to 10 m, stop
//   the CTD and proceed to waypoint 3
@2{ hold( ); \
      external("ctd -start", 0); \
      external("winch -depth 10", 1); \
      external("winch -depth 0", 1);   \
      external("ctd -stop", 0); \
      goto(3); }

// Start the bathymetry: enable the timeout timer
//   switch on the sonar, and do the scan area
@3{ settimer(t2,7200); \
      enablecond(1); \
      setregister(g9, 1); \
      scanarea(8, 10, 11, 12, 13, 14, 15 ); \
      setregister(g9, 0); \
      goto(4); }

@4{ hold(t0.eq.t0); };
```

Fig. 11. A program to implement the data acquisition mission described in the text. The definition of waypoints and safe area is not represented.

VII. Conclusions and future work

In this paper we presented the mission programming system of the autonomous sailboat FASt. A mission program is created by setting the reference waypoints, defining events and assigning actions to them, allowing to easily specify dynamic missions whose behavior depend on the state of the sailboat control and navigation system and also external systems. Mission programs are created as plain text files using a simple and intuitive language and can also be edited with an interactive graphics editor.

Future developments aim to test the programming environment in real operations and develop a framework to allow an easy integration in the command and control system of other robotic vehicles. Although this environment has been created for a specific sailing robot, the implementation of the mission compiler and mission execution components support easily the integration of new commands for adapting to other platforms.

Acknowledgement

The authors would like to acknowledge the support of the Faculty of Engineering of the University of Porto and the Department of Electrical and Computer Engineering.

This work is financed by the ERDF European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-037281

References

[1] P. F. Rynne and K. D. von Ellenrieder, "Unmanned autonomous sailing: Current status and future role in sustained ocean observations," *Marine Technology Society Journal*, vol. 43, no. 1, pp. 21–30, 2009.

[2] R. Hine, S. Willcox, G. Hine, and T. Richardson, "The wave glider: A wave-powered autonomous marine vehicle," in *OCEANS 2009, MTS/IEEE Biloxi-Marine Technology for Our Future: Global and Local Challenges*. IEEE, 2009, pp. 1–6.

[3] A. Silva, A. Matos, C. Soares *et al.*, "Measuring underwater noise with very high endurance surface and underwater autonomous vehicles," in *Proceedings of the OCEANS 2013 MTS-IEEE Conference, San Diego*. IEEE, 2013.

[4] H. Klinck, R. Stelzer, K. Jafarmadar, and D. K. Mellinger, "Aas endurance: An autonomous acoustic sailboat for marine mammal research," in *Proceedings of the International Robotic Sailing Conference (IRSC2009), Matosinhos, Portugal*, 2009, pp. 43–48.

[5] "Saildrone," http://www.saildrone.com (accessed July 2014), 2014.

[6] R. Stelzer and K. Jafarmadar, "The robotic sailing boat ASV Roboat as a maritime research platform," in *Proceedings of 22nd International HISWA Symposium*, 2012.

[7] C. Sauzé and M. Neal, "Design considerations for sailing robots performing long term autonomous oceanography," in *International Robotic Sailing Conference, Breitenbaum, Austria*, 2008, pp. 21–29.

[8] O. Ménage, A. Bethencourt, P. Rousseaux, and S. Prigent, "Vaimos: Realization of an autonomous robotic sailboat," in *Robotic Sailing 2013*. Springer, 2013, pp. 25–36.

[9] J. C. Alves and N. A. Cruz, "FAST—an autonomous sailing platform for oceanographic missions," in *Proceedings of the MTS-IEEE Conference—Oceans'2008*, 2008.

[10] ——, "METASail a tool for planning, supervision and analysis of robotic sailboat missions," in *Robotic Sailing 2014 (to appear)*.

[11] T. Schneider and H. Schmidt, "Unified command and control for heterogeneous marine sensing networks," *Journal of Field Robotics*, vol. 27, no. 6, pp. 876–889, 2010.

[12] J. Pinto, P. Calado, J. Braga, P. Dias, R. Martins, E. Marques, and J. Sousa, "Implementation of a control architecture for networked vehicle systems," in *Navigation, Guidance and Control of Underwater Vehicles*, vol. 3, 2012, pp. 100–105.

[13] N. Dias, C. Almeida, H. Ferreira, J. Almeida, A. Martins, A. Dias, and E. Silva, "Manoeuvre based mission control system for autonomous surface vehicle," in *OCEANS 2009-EUROPE*. IEEE, 2009, pp. 1–5.

[14] J. Alves, P. Oliveira, R. Oliveira, A. Pascoal, M. Rufino, L. Sebastiao, and C. Silvestre, "Vehicle and mission control of the DELFIM autonomous surface craft," in *Control and Automation, 2006. MED'06. 14th Mediterranean Conference on*. IEEE, 2006, pp. 1–6.

[15] M. R. Benjamin, P. M. Newman, H. Schmidt, and J. J. Leonard, "An overview of MOOS-IvP and a users guide to the IvP helm autonomy software," MIT Technical report, August 2010.