# MPL—A Mission Planning Language for Autonomous Surface Vehicles

Henrique M.P. Cabral, José C. Alves, Nuno A. Cruz, José F. Valente, and Diogo M. Lopes

**Abstract.** In this paper we present the specification of a scripting language for mission planning in Autonomous Surface Vehicles. Besides the specification of common missions based on a sequence of waypoints, the main goal was to extend the scope of application to specific need of autonomous sailing boats. These include, for example, the ability to react to environmental conditions in real time and reprogram accordingly. The proposed Mission Planning Language (MPL) hides the details of underlying control and navigation components, and focuses on the high-level procedures of a mission. We discuss the requirements and principles behind the language and show how oceanographic data collection missions can be significantly improved by such a facility. Finally, we illustrate an application example inspired on a real scenario of the FEUP Autonomous Sailboat (FASt).

## 1 Introduction

Autonomous surface vehicles (ASVs), in particular autonomous sailboats, provide a particularly interesting platform for oceanographic experiments and data collection. Their potential for large autonomy, together with the ability to fit many different types of sensors and establish constant communication with a remote base station, allow to carry missions of several weeks or months without expensive and time-consuming local monitoring [3, 4].

However, and despite the advances in control algorithms for these vessels in recent years, the specification of a mission remains, from the point of

Henrique M. P. Cabral · José F. Valente · Diogo M. Lopes
Master in Electrical and Computer Engineering, University of Porto,
Faculty of Engineering
e-mail: henrique.cabral@fe.up.pt

José C. Alves · Nuno A. Cruz
Department of Electrical and Computer Engineering, University of Porto,
Faculty of Engineering

DOI: 10.1007/978-3-319-02276-5_11, © Springer International Publishing Switzerland 2014

view of the end users, difficult and error-prone, sometimes requiring case-by-case adjustments to the control software. Presently several applications use friendly graphical user interfaces for specifying, simulating, monitoring and analysing the behaviour of autonomous vehicles [5, 8]. Although such environments hide the complexity of the low level systems and increase the tolerance to faults due to lack of proficiency of operators, they also constrain the variety of possible tasks that the systems are able to accomplish. For example, if a given mission planning system for an autonomous vessel only allows to define a sequence of waypoints, it may be impossible to specify reactions to unanticipated events such as dynamic obstacles or changes in environmental conditions.

Although there has been some effort to provide tools for high level mission planning specification, based on flexible and highly parameterizable commands, these have been mainly focused on motorized autonomous surface vehicles [6]. In the particular case of autonomous sailing boats, a typical mission needs to take into account the environmental conditions, in special the wind and sea state. Furthermore, as sailing boats have the potential for long term deployments, they may end up facing weather conditions that are impossible to predict accurately when the mission is planned. The ability to react to real time data and reprogram the remaining tasks according to these dynamic conditions is essential for successfully completing a mission and it is specially critical when safety is an issue.

In this paper we describe a mission planning language (MPL) capable of both high-level specifications and also low level control over the system operation. In this way, all control details and navigation restrictions (such as wind conditions in sailboats) are hidden from the user, and the same mission script could be executed transparently by both a sailboat and a powerboat. We present the language requirements and specification, as well as a discussion of the design principles behind MPL and some real use cases based on previous missions with the FEUP Autonomous Sailboat (FASt) [1].

## 2 Requirements

As stated in the introduction (Sect. 1), the main requirements for MPL are ease of use and, simultaneously, providing experienced users with the ability to specify lower-level control:

**Control-oriented.** Unlike the detailed algorithms underlying the operation of the vehicle, a mission plan typically involves very few calculations or complex data structures. Instead, the user should be able to specify simple[1]

[1] From their point of view, not the system's. Often, what seems simple to a user ("follow a circle") may require more or less complex maneuvers by the vehicle. These should be entirely transparent.

actions directly, as well as higher control structures (loops, conditionals) over these actions.

**Modularity.** Often during a mission—in particular during data collection—the same composite action must be repeated at different points in time and space. The language should provide mechanisms for the definition and reuse of such actions, such as functions or procedures.

**Abstraction.** Most missions should not have any need to access lower-level features of the system, such as direct actuator control. Instead, common primitives should conceal the user from such details, guaranteeing stability on the face of hardware and control system alterations.

**Transparency.** At the same time, more demanding applications should still be able to tweak the core functionality to satisfy their own requirements. This includes direct hardware access, as well as system state variables and control procedures.

Taking into account the usage context, both from the user's and the implementer's perspective, other requirements appear:

**Instant feedback.** Since operator failure or error in specifying the mission may easily lead to loss of data, time, or even the vehicle, it is vital that they receive immediate and accurate feedback in order to evaluate their actions.

**Extensibility.** If the language is to be used in a wide variety of ASVs, the designer cannot presume to foresee every possible action the user may want to perform. MPL should, therefore, provide mechanisms to define new commands and structures, particularly for hardware control.

**Resource efficiency.** Given the reduced resources under which many ASV systems operate, the language should be easy to parse and have a compact representation for internal use.

**Easy integration.** The language must be easy to couple with existing control systems, as well as accurate simulation environments for fast evaluation of the navigation pattern determined by a navigation script. This is particularly important for sailboats, where the true navigation path cannot be entirely planned due to the mostly unforeseeable wind and sea conditions at the time of the mission.

All considered, the requirements exposed above point to a command-based, interpreted language, with uniform parsing and a simple interface with the control system, as well as higher-level features such as loops and conditionals. As such, we have decided to model MPL on Tcl [7], which provides a well-tested boilerplate obeying several of the requirements.

## 3 The Language

MPL is, by design, a domain-specific language (DSL), in that it provides facilities especially designed for the control of ASVs. On the other hand, taking into account the variety of tasks one may want to perform in this category, it must also allow the creation of even more specialised languages. This thought informs not only the syntax but also the primitives supplied by the language. These will be presented and discussed in this section.

### *3.1 Execution Context*

Within MPL, the *execution context* is defined as the set of all procedures (Sect. 3.3) and variables (Sect. 3.2) accessible to the user script at the current point in time. Separate user contexts are used between global execution and procedures, allowing the user to define local variables and nested procedures.

### *3.2 Variables*

MPL variables can be either user variables or system variables. The distinction between string, numerical, or boolean types exists only for the values a user variable may take[2]. Strings may be of arbitrary length and are encoded internally as UTF-8; numerical values can be either real (stored as double precision IEEE 754 floating-point values) or integer (stored as 64-bit signed integers); and boolean values are stored as unsigned 8-bit values, with 0 interpreted as "false" and anything else as "true".

**User variables.** These correspond to regular, user-defined variables. They may take up any value of the types described below, and the type of value stored may change over the course of execution. User variables exist solely within the context of the script.

**System variables.** To access or modify any external system data (such as configurations, sensor values, or actuations), system variables must be used. These have direct correspondence with memory locations in the system and will always reflect the current value at those locations. Together with system-specific primitives, they make up the interface between user script and system code that is specific to MPL.

[2] The lack of an array or list value type is due only to the simplified syntax used (see also Sect. 3.4).

## *3.3 Procedures*

MPL supports user-defined procedures through the `proc` primitive (Sect. 3.5.1). The initial execution context for a procedure contains, besides the procedures defined in the calling context, only system variables and the variables defined at the beginning of the procedure (i.e. those corresponding to its arguments). This means that no outside user variables are accessible by default. To bring external variables into the current execution context, the procedure may invoke the `extern` primitive. Additionally, procedure definitions may nest, since each is defined relative to the current execution context (see also Sect. 3.1).

Procedures may end and return a value through the `return` primitive. If no `return` statement is given, the value of the last executed statement is returned.

## *3.4 Syntax*

A MPL script is a string, consisting of a series of commands separated by newlines. Each command, in its turn, is composed of a series of words, separated by whitespace (except newlines). The first word of a command is used to determine the procedure to be executed, which is passed the remaining words as arguments. In the particular case that the first letter of the first word of the command is a hash ("#"), the remainder of the line is ignored.

To enable the construction of complex commands, MPL provides two features. Both work by literal substitution into the initial command, forming a single word.

**Command nesting.** Anything between brackets ("`[]`") is considered to be a script in itself and is executed within the current context. Once everything up to the closing bracket is executed, the result of the last command is substituted for the original text.

**Variable substitution.** MPL variables can be referred using the dollar sign ("`$`"). Whenever the symbol occurs and is followed by a valid variable name (composed by alphanumeric characters and/or underscores), the value of that variable is substituted into the word.[3]

The third type of substitution is used to include special characters without attaching to them their usual meaning:

**Backslash substitution.** Whenever a character is preceded by a backslash ("\"), it is inserted in the word and the backslash is eliminated. This includes

[3] As mentioned before, there is no particular syntax for variables containing array values. A possibility would be to use, as in Tcl, `$var(<index>)`, but this has not yet been implemented.

characters that would otherwise hold special meaning, such as dollar signs or brackets.

All three substitutions are performed left to right, and no character is parsed more than once. This way, once a substitution is made, it will not trigger other substitutions. Furthermore, the result of a substitution is always included in the current word, never separated into two or more words.

The two remaining constructions, while somewhat redundant, provide a more convenient way of writing complex words without resorting to escaping (using backslashes) all whitespace or special characters:

**Double quotes.** Mainly intended for character strings. If a word starts with a double quotation mark (“`"`”), it is only terminated by the next double quote. Everything between the two marks is included in the word, but not the marks themselves.

**Curly braces.** If a word starts with an opening brace (“`{`”), everything until the next closing brace (“`}`”) is considered to be part of the word (except for the braces themselves). Unlike in quote-enclosed strings, no substitutions are performed.

This concludes the exposition of the language syntax. All of these constructions aim, in one way or another, to improve the flexibility of the basic command structure while maintaining syntactic “sugar” low and parsing simple. Additional syntax and meaning can always be added by specific commands, either native or user-defined.

## *3.5 Primitives*

MPL primitives are designed to achieve two main goals: first, to enable the use of basic language features (namely, variables, control structures, and procedures); and second, to provide elementary navigation and control for the ASV. Accordingly, they will be presented in this section according to their purpose and functionality.

*Note:* in the following, `[<param>]` denotes an optional parameter, while `<par1|par2>` denotes a choice between two parameters.

### 3.5.1 Basic Language Features

**`set`** `<name> [<value>]`

Return the value of the variable `<name>`. If `<value>` is given, set it as the value of the variable. The variable is created if it doesn’t exist.

**`if`** `<cond> <body1> [<body2>]`
The condition `<cond>` is evaluated (in the same way as an `expr` argument—see Sect. 3.5.4), and its result is interpreted as a boolean value (true or false). The statements in `<body1>` are executed if the result is true. If `<body2>` is given, it is executed if the result is false.

**`while`** `<cond> <body>`
The condition `<cond>` is evaluated, and its result is interpreted as a boolean value. If true, the statements in `<body>` are executed. The condition is reevaluated at the end of each execution, stopping only when it becomes false.

**`proc`** `<name> <args> <body>`
Create a new procedure `<name>`. Whenever the procedure is invoked as part of a command, the contents of `<body>` will be executed, with the supplied arguments being assigned to the local variables with names listed in `<args>`.

**`extern`** `<var>`
Look for the variable `<name>` in the global execution context, and bring it into the current context.

**`return`** `[<val>]`
Return from the current procedure. If `<val>` is specified, it's used as the return value for the procedure. Otherwise, the return value is empty.

**`halt`**
Stop execution of script.

### 3.5.2 Navigation

**`coordinate`** `<lat> <lon>`
Returns a coordinate value for the specified latitude/longitude pair (in decimal degrees), to be assigned to a variable such as a waypoint, or used in other navigation commands.

**`add`** `<coord> <dist> <direction>`
Produces a new coordinate from `<coord>` by adding the indicated distance, `<dist>` (in metres), in the specified direction (angle in degrees, clockwise from North).

*Note:* other convenience primitives could be included here, such as straightforward unit conversions (nautical miles to kilometres, decimal degrees to DMS, etc.).

### 3.5.3 Control

**go** `<coord> [<body>] [-at]`

Direct the vehicle to the indicated point `<coord>`. If `<coord>` contains more than one point, follow the points in sequence. Optionally, if a `<body>` is supplied, its commands are executed in parallel while the final point hasn't been reached. The `-at` option can be used with a `<body>` to execute it at every point.

**follow** `<var> [<body>]`

Instruct the vehicle to follow a moving point. The variable `<var>` is read once per control loop to find the point coordinates. If the optional `<body>` is specified, it is executed in parallel with the command.

**station** `<coord> <dist> [-square]`

Keep the vehicle within `<dist>` of the point `<coord>`. The strategy to keep the vehicle in place is implementation- and vehicle-dependent (such as following "eights" for a sailboat or motor control for powerboats). The metric can be changed to keep the vehicle within a square of side `<dist>` using `-square`.

**stop**

Make the vehicle come to a stop, possibly by maneuvering to a favourable position first. Note that currents and wind may still affect the position.

### 3.5.4 Miscellaneous

**expr** `<str>`

Evaluate the result of the arithmetic expression contained in the string `<str>`. Supports basic arithmetic operators (`+`, `-`, `*`, `/`) and relational operators (`==`, `!=`, `<`, `<=`, `>`, `>=`), as well as parenthesis grouping. Arithmetic operators take precedence over relational operators, and precedence among them is as usual. Associativity is left-to-right. Whitespace does not affect evaluation.

**log** `<str>`

Write text string `<str>` to log, together with timestamp. Depending on the implementation, this may be a local log or transmitted to a base station.

**time** `[-iso|-asc]`

Return the current system time as a 64-bit Unix time value. If the `-iso` option is used, the time is returned instead as an ISO 8601 string[4]. If the

[4] `<YYYY>-<MM>-<DD>T<HH>:<MM>`

`-asc` option is used, the time is returned as a string in the format of the standard C library function `asctime`[5].

**`timer`** `<interval> [<body>]`
Start a timer with a duration of `<interval>` milliseconds. The procedure returns only when the time is reached. If `<body>` is specified, it is executed when this happens.

**`run`** `<file>`
Load and execute the MPL script contained in `<file>`. The script is executed within the current context.

## 4 Use Case Analysis

In order to showcase the flexibility and applicability of MPL, we will now present a real situation based on a mission with FASt. In this mission, the boat was fitted with an electrical winch at the stern and a hydrophone was attached to the end of the spool line. The objective was to sail to a set of predetermined points, stopping at each point to lower the device to three different depths for a certain length of time.

Typically, the command interface for FASt would not allow such detailed procedures to be programmed, forcing us to adjust the existing communication protocols with mission-specific changes. This was neither desirable nor elegant. However, using MPL to script the mission, the task becomes straightforward. The used code may be found in Appendix. Some things worth remarking:

- The convention used for FASt MPL is that all system variables have names beginning with an underscore (“_”). In particular, `_position` contains the current GPS position of the boat, `_depth` contains the current hydrophone depth, and `_battery` contains the battery charge in percentage (0-100).
- Since actuator commands work through memory-mapped registers, some wait cycles are required to allow the system to reach the desired state (line 22).
- Nesting the procedure `lower` within `measure` makes sense for this application, since it isn't required anywhere else. If finer-grained control was necessary, it could be brought outside without changing any other part of the script.
- The usage of variables named `depth_1`, `depth_2`, and `depth_3` strongly points to the usefulness of a list or array construction. This was recognised

[5] `<Www> <Mmm> <dd> <hh>:<mm>:<ss> <yyyy>`

as an important feature early in development, but hasn't yet been included to keep the syntax and parsing as simple as possible (see footnotes 2 and 3). As mission scripts grow larger, it will quickly become necessary to include it.

## 5 Conclusions

In this paper we presented a mission planning language (MPL) that provides a simple to use and flexible framework for planning high-level missions for ASVs. Although this is in an early stage of development, a significant set of features have already been seamlessly integrated into the FASt autonomous sailing boat. This proved to be an easy way to program complex missions that react to dynamic conditions perceived from the on-board sensors in real time. MPL is also well-suited to integration with high-level mission simulators, so that users can plan a complete mission offline with a high degree of confidence in the correctness of the real behaviour of the vehicle.

We believe MPL has the potential to be used in all ASV command systems, but also to be expanded to include other autonomous vehicles, such as AUVs and UAVs. For this, the existence of a standard implementation of the basic language is paramount, and that will be the objective of future developments.

## Appendix

```
# useful constants
set winch_perim [expr "2*3.14159*0.2"]

# define a square to take measurements at the vertices
set sw_corner [coordinate 38.408137 -9.134102]
set se_corner [add $sw_corner 1000 90]
set ne_corner [add $se_corner 1000 0]
set nw_corner [add $sw_corner 1000 0]

# define measurement depths (metres)
set depth_1 10
set depth_2 20
set depth_3 30

# get initial position to return to
set home $_position

# initial battery check
if {$_battery < 30} {
    log "Low battery, refusing to perform mission"
    stop
    halt
}

# go to a point and perform measurements while stopped
proc measure {point} {
   # lower hydrophone to given depth (in metres) and wait some time (minutes)
   proc lower {d t} {
        extern winch_perim
        set _winch [expr "($d-$_depth) / $winch_perim"]
        while {$_depth < $d} { }
        timer [expr "$t*60000"]
    }

    # go to given point and lower hydrophone to specified depths
    log "Going to point ($point)"
    go $point {
        log "Point reached, lowering hydrophone to $depth_1"
        lower $depth_1  1
        log "Lowering to $depth_2"
        lower $depth_2  1.5
        log "Lowering to $depth_3"
        lower $depth_3  2
        log "Raising hydrophone"
        lower 0         0
    } -at
}

# main mission
measure $sw_corner
measure $se_corner
measure $ne_corner
measure $nw_corner

# return home
go $home {stop} -at
```

## References

1. Alves, J.C., Cruz, N.A.: FAST—an autonomous sailing platform for oceanographic missions. In: Proceedings of the MTS-IEEE Conference—Oceans'(2008)
2. Benjamin, M.R., Schmidt, H., Newman, P.M., Leonard, J.J.: Nested autonomy for unmanned marine vehicles with moos-ivp. Journal of Field Robotics 27(6), 834–875 (2010)
3. Caccia, M.: Autonomous surface craft: prototypes and basic research issues. In: 14th Mediterranean Conference on Control and Automation, MED 2006, pp. 1–6. IEEE (2006)
4. Cruz, N.A., Alves, J.C.: Autonomous sailboats: an emerging technology for ocean sampling and surveillance. In: OCEANS 2008, pp. 1–6. IEEE (2008)
5. Meier, L., Tanskanen, P., Heng, L., Lee, G.H., Fraundorfer, F., Pollefeys, M.: Pixhawk: A micro aerial vehicle design for autonomous flight using onboard computer vision. Autonomous Robots 33(1-2), 21–39 (2012)
6. Newman, P.M.: Moos-mission orientated operating suite. Massachusetts Institute of Technology. Tech. Rep 2299(08) (2008)
7. Ousterhout, J.: Tcl-a universal scripting language. lecture at MIT (1995)
8. Zurich, E.: Qgroundcontrol: Ground control station for small air land water autonomous unmanned systems (2013), `http://qgroundcontrol.org/` (accessed July 2013)