Mission Control of the MARIUS AUV: System Design, Implementation, and Sea Trials

P. Oliveira, A. Pascoal, V. Silva, C. Silvestre

Department of Electrical Engineering/Institute for Systems and Robotics

Instituto Superior Técnico (IST)

Av. Rovisco Pais 1, 1096 Lisboa Codex, Portugal

E-mail address: antonio@isr.ist.utl.pt

Abstract

This paper describes the design and implementation of a Mission Control System for the MARIUS Autonomous Underwater Vehicle (AUV). The framework adopted for system design builds on the key concept of *Vehicle Primitive*, which is a parameterized specification of an elementary operation performed by the vehicle. Vehicle Primitives are obtained by coordinating the execution of a number of concurrent *System Tasks*, which are parameterized specifications of classes of algorithms or procedures that implement basic functionalities in an underwater robotic system. Vehicle Primitives are in turn logically and temporally chained to form more abstract *Mission Procedures*, which are executed as determined by *Mission Programs*, in reaction to external events.

System Task design is carried out using well established tools from continuous/discrete-time dynamic system theory, and finite state automata to describe their logical (event-based) interaction with Vehicle Primitives. The design and analysis of Vehicle Primitives and Mission Procedures build on the theory of Petri nets, which are naturally oriented towards the modeling and analysis of asynchronous, concurrent discrete event systems. Vehicle Primitives and Mission Procedures can be developed and implemented on the vehicle's distributed computer system using the specially designed software programming environments *CORAL* and *ATOL*, respectively. The first is a set of software tools that allows for graphically building a library of Vehicle Primitives embodied in Petri nets, and running them in real-time. The latter provides similar tools for Mission Procedure programming, but relies on a reactive synchronous programming language as a way to manage the potential complexity introduced by the occurrence of large Petri net structures. Whereas the first has been fully implemented, the latter has been specified but is still under development. Thus, at this stage of development, Mission Procedures and Mission Programs are effectively embodied into - higher level - Petri net structures that control the scheduling of the Vehicle Primitives that are necessary to execute a given mission.

The paper provides a summary of the general methodology adopted for the design and implementation of Mission Control Systems for underwater robots, and describes its application to the control of the MARIUS AUV. The paper introduces the experimental set-up for mission programming, mission execution, and mission follow-up from a support station, and describes the sequence of steps involved in programming and running a selected mission with the vehicle at sea.

1 Introduction

Among the challenges that face the designers of underwater vehicle systems, the following is of the utmost importance: design a computer based *Mission Control System* that will

- enable an operator to define a vehicle mission in a high level language, and translate it into a mission plan.
- provide adequate tools to convert a mission plan into a Mission Program that can be formally verified and executed in real-time.
- endow an operator with the capability to follow the state of progress of the Mission Program as it is executed, and modify it if required.

Meeting those objectives poses a formidable task to underwater system designers, who strive to develop vehicles that can be programmed and operated by end-users that are not necessarily familiarized with the most intricate details of underwater system technology. Identical problems face the designers of complex robotic systems in a number of areas that include advanced manipulators, industrial work cells, and autonomous air and land vehicles. The widespread interest of the scientific community in the design of Mission Control Systems for advanced robots is by now patent in a sizable body of literature that covers a wide spectrum of research topics focusing on the interplay between event driven and time-driven dynamical systems. The former are within the realm of Discrete Event System Theory (Cassandras, 1993), whereas the latter can be tackled using well established theoretical tools from the field of Continuous and Discrete-Time Dynamical Systems (Franklin et al., 1990).

Mission Control: previous work and new requirements. Early references in this vast area include the pioneering work of K.S. Fu (Fu, 1970), Saridis (Saridis, 1979; Saridis, 1989) and Albus (Albus, 1988), who set the ground for the study of learning control systems, intelligent machine organization, and general architectures for autonomous undersea vehicles, respectively. For an overview of recent theoretical and applied work in the field, the reader is referred to (Antsaklis and Passino, 1993) and (Valavanis et al., 1995), which contain a number of papers on the design of advanced control systems for unmanned underwater vehicles, combined underwater vehicle and manipulator systems, intervention robots, and air vehicles.

Spawned from the availability of small embedded processors and the ever increasing capabilities of underwater communications and acoustic sensors, there is now considerable interest in validating the theoretical approaches to Mission Control System design with experiments conducted with prototype vehicles. The reader will find in (Valavanis et al., 1995) a large number of papers describing vehicles operated by a number of universities and research institutes in Europe, Asia and the US, and on the state of development of their Mission Control Systems. Representative vehicles include VORTEX (IFREMER, France), ROBY (Istituto Automazione Navale, Italy), MARIUS (operated by the Instituto Superior Tecnico, Portugal on behalf of the European Commission), PHOENIX (Naval Postgraduate School, U.S.A), ODYSSEY (M.I.T. Sea Grant Programme, U.S.A.), OTTER (MBARI/Stanford, U.S.A.), OCEAN EXPLORER (Florida Atlantic University, U.S.A.), MT-88

and TUNNEL SEA LION (IMTP, Russia), and PTEROA (Japan). A recent publication (Coste-Maniere et al., 1995) provides a very lucid presentation of some of the problems encountered in establishing a common syntax and framework for cooperation among the researchers in the field, and clearly identifies different issues/paradigms that warrant further investigation in the area of software and hardware architectures for underwater robotics.

As part of the international effort to develop advanced systems for underwater vehicle mission control, IST has developed the first version of a Mission Control System for the prototype MARIUS AUV (Ayela et al., 1995). This paper provides a brief summary of the framework adopted for the design and implementation of the Mission Control System proposed, describes its implementation thus far on the computer network resident on-board the AUV, and reports the results of a series of tests conducted with the vehicle at sea in Sines, Portugal. The research and development work undertaken by IST was driven by the need to develop efficient mission programming/execution tools that would

- allow for the modeling of discrete event systems where the transitions between events are enabled according to arbitrary rules.
- simplify the task of decomposing or modularizing potentially complex systems.
- allow for the use of formally established logical verification methods.
- lend themselves to real-time implementation without resorting to software packages requiring high computational power.
- simplify the steps that are necessary to go from concept to practice, effectively providing all the software tools to automatically generated target code to be run on the vehicle computer network.
- provide simple graphic interfaces for mission programming, debugging, and mission follow-up during sea trials.

The first three issues were tackled by adopting the formalism of Petri net theory. The remaining issues were instrumental in determining the specification and development of two new software applications named ATOL and CORAL.

Due to space limitations, the presentation purposely avoids focusing on the theoretical foundations of Mission Control System design and analysis, which have been pursued in (Silva et al., 1995; Silva, 1996). The reader will find in (Oliveira et al., 1996) an abridged version of the work in (Silva et al., 1995; Silva, 1996), which has been influenced by the solid body of research carried out by INRIA/IFREMER in France, with applications to the VORTEX vehicle, and at NPS in the U.S. with applications to the PHOENIX vehicle, see (Lee and McGhee, 1994) and the references therein.

Mission Control: A Petri net based approach. The methodology adopted for the design of a Mission Control System for the MARIUS AUV builds on the key concept of *Vehicle Primitive*, which is a parameterized specification of an elementary operation performed by an underwater vehicle (e.g., keeping a constant vehicle speed, maintaining a desired heading, holding a fixed altitude over the seabed, or taking video images of the seabed at pre-assigned time intervals). Vehicle Primitives are obtained by coordinating the execution of a number of concurrent *(Vehicle) System Tasks*, which are parameterized specifications of classes of algorithms or procedures that

implement basic functionalities in an underwater robotic system (e. g., the Vehicle Primitive in charge of maintaining a desired heading will require the concerted action of System Tasks devoted to motion sensor data acquisition, navigation and vehicle control algorithm implementation, and actuator control). Vehicle Primitives can in turn be logically and temporally chained to form *Mission Procedures*, aimed at specifying parameterized robot actions at desired abstraction levels. For example, it is possible to recruit the concerted operation of a set of Vehicle Primitives to obtain a parameterized Mission Procedure that will instruct the vehicle to follow an *horizontal path* at a constant speed, depth and heading, for a requested period of time. Mission Procedures allow for modular *Mission Program* generation, and simplify the task of defining new mission plans by modifying/expanding existing ones.

Using the methodology adopted System Task design is carried out using well established tools from continuous/discrete-time dynamic system theory, and finite state automata to describe their logical interaction with Vehicle Primitives. The design and analysis of Vehicle Primitives and Mission Procedures build on the theory of Petri nets, which are naturally oriented towards the modeling and analysis of asynchronous, discrete event systems with concurrency. This approach leads naturally to a unifying framework for the analysis of the logical behaviour of the discrete event systems that occur at all levels of the Mission Control System. Vehicle Primitives and Mission Procedures can be developed and implemented using special software programming environments named *CORAL* and *ATOL*, respectively. The first is a set of software tools that allows for graphically building a library of Vehicle Primitives embodied in Petri nets, and running them in real-time. The latter provides similar tools for Mission Procedure programming, but relies on a reactive synchronous programming language as a way to manage the potential complexity introduced by the occurrence of large Petri net structures. Whereas the first has been fully implemented, the latter has been specified but is still under development.

At the core of the Mission Control System implementation is the specially developed CORAL software programming environment, which consists of two fundamental modules: i) the *Vehicle Primitives Library Editor and Generator*, and ii) the *CORAL Engine*. The main goal of the Library Editor and Generator is to embody each Vehicle Primitive into a Petri net description, and to assemble a set of translated Vehicle Primitives into a Vehicle Primitive Library. The definition of each Primitive can be input either graphically through a CORAL graphic input interface, or directly using the CORAL language. A CORAL compiler/linker is in charge of processing the Vehicle Primitives inputs and of assembling the corresponding output data in the Vehicle Primitives Library. As an intermediate step, the CORAL graphic input interface produces a CORAL Graphics Library for later use during real-time operation. Currently, the Vehicle Primitives Library Editor and Generator can be run on a PC/DOS or on a Unix Workstation. During real-time operation, each Vehicle Primitive is executed by the CORAL Engine, which sends commands to and receives responses from the Vehicle System Tasks. Both the CORAL Engine and the software that implements the System Tasks run on the GESPAC OS-9 based target computer network of MARIUS. It is important to remark that the CORAL Engine remains fixed, and that the implementation of a new Vehicle Primitive simply requires that a new data set produced by the CORAL compiler be added to the Vehicle Primitives Library. This fact is important, as it simplifies the programming of new Vehicle Primitives, and makes the task of loading and unloading existing ones trivial.

The CORAL software environment was initially developed to implement Vehicle Primitives, only. However, it was realized early on that CORAL could be used to implement a first kernel of a complete Mission Control System for the AUV, while the ATOL software programming environment was being implemented. This was in fact done in the course of the MARIUS AUV project (Ayela et al., 1995), leading to a methodology whereby Mission Procedures and Mission Programs are

effectively embodied into - higher level - Petri Net descriptions that control the scheduling of Vehicle Primitives concurring to the execution of a particular mission. Furthermore, Mission Procedures and Mission Programs are generated using the graphic approach described above. During real-time operation, the Mission Control System reports its state to a Mission Assessment System implemented on a PC/DOS machine (using an aerial link during surface testing, or the acoustic communication link while diving). This information is then displayed on a computer screen using the CORAL Graphics Library described before. The set-up developed allows for easy programming of missions, and endows the system developer with a graphic interface to evaluate the state of progress of the mission based on the evolution of tokens in a Petri Net.

The paper is organized as follows: Section 2 introduces the MARIUS AUV, and discusses very briefly its mission requirements and functional organization. Section 3 describes the basic framework adopted for Mission Control System design and implementation using the software programming environments CORAL and ATOL. Section 4 describes the Mission Control System that was actually implemented on the AUV computer network, and illustrates the basic steps involved in the design of a Mission Program for a simple mission example. Section 5 describes the set-up for mission execution and mission follow-up from a support station, and reports the results of running the mission described, at sea. Finally, the paper concludes with a discussion of the limitations of the current Mission Control System, and indicates the steps that are being taken to improve it.

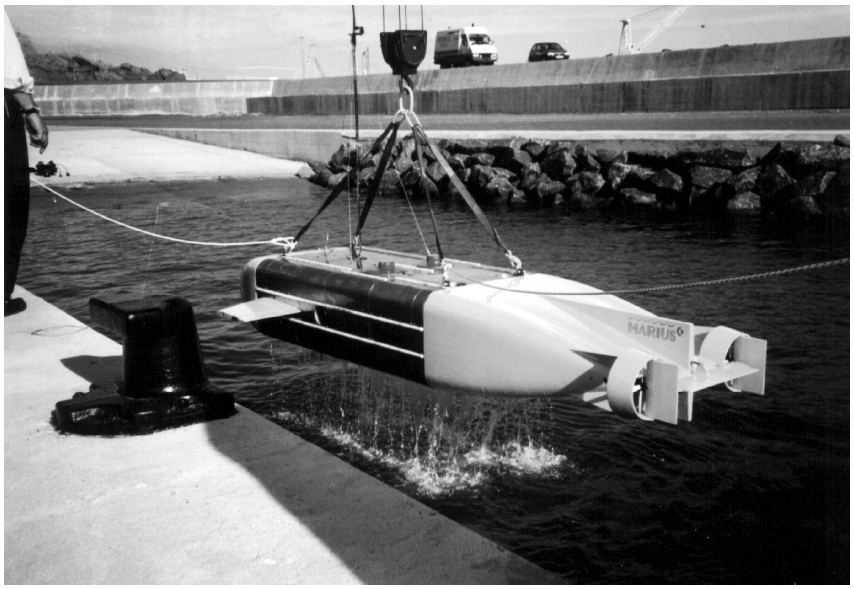

Figure 1: The MARIUS Vehicle.

2 The MARIUS AUV. Mission Control Requirements and Vehicle System Organization.

To motivate and better focus the presentation that follows, this section provides a brief description of the MARIUS AUV - depicted in Fig. 1 - and outlines some of its envisioned mission scenarios. For details on the design, development and testing of the vehicle, including its computer network, actuators and sensors, see (Ayela et al., 1995; Pascoal, 1994) and the references therein.

The MARIUS vehicle is 4.5 m long, 1.1 m wide and 0.6 m high. The vehicle is equipped with two main propellers for cruising, four tunnel thrusters for station keeping maneuvers, and rudders, elevator and ailerons (forward planes) for vehicle steering in the vertical and horizontal planes. Attached to the top part of the hull are two transducers that are part of the vehicle's acoustic communication and long baseline positioning systems. The vehicle has a dry weight of 1060 kg, a payload capacity of 50 kg, and a maximum operating depth of 600 m. Its maximum rated speed with respect to the water is 2.5 m/s. At the speed of 1.26 m/s, its expected mission duration and mission range are 18 h and 83 km, respectively.

2.1 Mission Control Requirements

The mission requirements for the MARIUS AUV have been analyzed in (Pascoal, 1994), where the reader will find the description of two envisioned mission scenarios in Danish and Portuguese coastal waters that focus on civilian applications. One of the scenarios takes place in the North Sea/Skagerrak - Kattegat area, and aims at localizing and estimating the spatial extension of areas with high sedimentation or lateral input of phytoplankton to the benthos, and establishing their correlation with an important pelagic front area. The extension of the frontal zone (50 km $\times$ 50 km), the water depths (from 30 to 150m) involved, and the need to probe for interesting features underwater in an unsupervised manner, make traditional surveying using divers or towed sensors very costly or inadequate to the task at hand. The mission envisioned consists of a series of grid surveys along linear tracks, each track being traversed twice: this leads to a bottom survey in one direction, followed by a near the surface survey back to the starting point, while performing an undulating maneuver to determine the spatial extension of a boundary layer area. During part of the survey, the vehicle is required to cruise at constant speed and height above the seabed, while acquiring data on water temperature, salinity and fluorescence. The taking of video images and photographs at sites on a pre-determined grid is also required while the vehicle is hovering. During operation close to the seabed, the vehicle should be able to detect and avoid unforeseen obstacles.

Guided by a detailed analysis of the missions envisioned, a basic set of performance requirements for the MARIUS AUV has been specified in (Pascoal, 1994). Stated qualitatively, the main specifications include a good compromise between platform stability and maneuverability, robustness against vehicle parameter variations, low sensitivity with respect to external disturbances, error recovery capabilities, and the possibility to program and follow the execution of vehicle missions using user-friendly interfaces. These considerations led to the basic vehicle system organization that is explained in the sequel.

2.2 Vehicle System Organization

The following basic MARIUS AUV systems and their interconnections can be identified in Fig. 2:

Vehicle Support System (VSS) - The Vehicle Support System controls the distribution of energy to the electrical and electromechanical hardware installed on-board the vehicle, and monitors its energy consumption. This system is also in charge of detecting basic hardware failures and triggering appropriate emergency reflexive maneuvers whenever required (e.g., upon detection of a leak in a pressure container, it forces the vehicle to surface by inflating a lift bag).

Actuator Control System (ACS) - The Actuator Control System is responsible for controlling the speed of rotation of the propellers and the deflections of the ailerons, rudders and elevator.

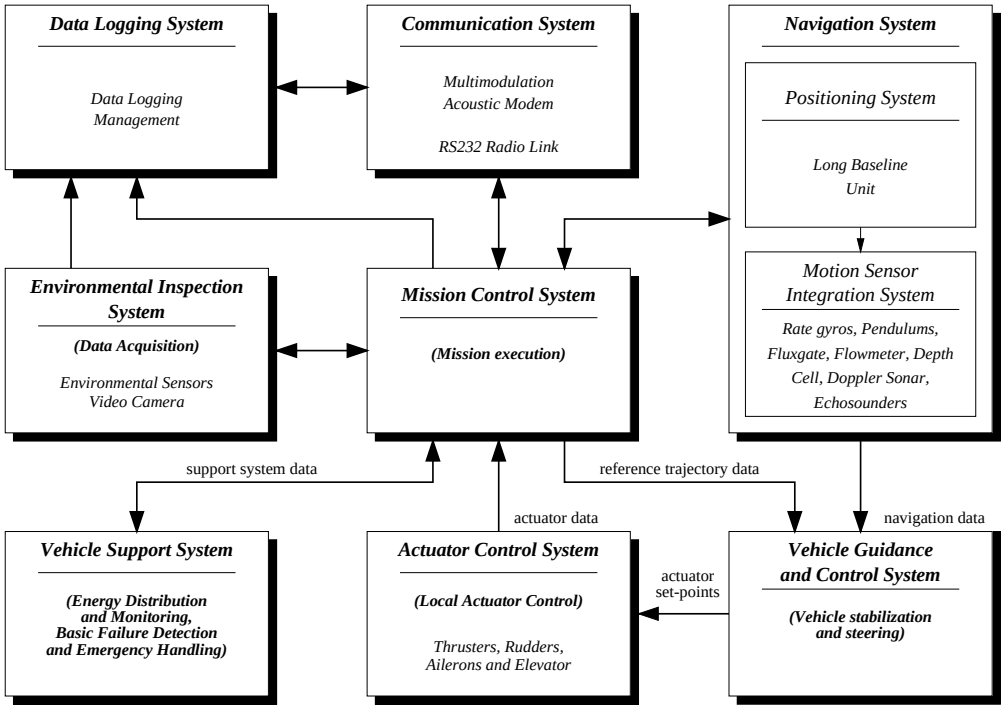

Figure 2: Vehicle System Organization.

Actuator set points are provided by the Vehicle Guidance and Control System. Actuator data are fed back to the Mission Control System for vehicle status assessment.

Navigation System (NS) - The Navigation System provides estimates of the linear position and velocity of the vehicle, as well as its orientation and angular velocity. This system merges information provided by the Positioning System (a long baseline unit with a network of transponders) and a Motion Sensor Integration System. The motion sensor package includes the following units:

- 3 rate gyros, 2 pendulums and 1 fluxgate (Watson Attitude & Heading Reference Unit AHRS-C303).
- 1 flowmeter TSA-06-C-A (EG& G Flow Tech.).
- 1 depth cell DC 10R-C (Transinstruments).
- 2 echosounders ST200 (Tritech).
- 1 Doppler Log TSM 5740 with 4 beams in a Janus configuration, operating at 300 KHz (Thomson-ASM).

The outputs of the Navigation System are input to the Vehicle Guidance and Control System, and sent to the Mission Control System for vehicle performance assessment.

Vehicle Guidance and Control System (VGCS) The Vehicle Guidance and Control System accepts as inputs reference trajectories issued by the Mission Control System, and navigational data provided by the Navigation System. It outputs commands to the Actuator Control System (set points for the speed of rotation of the propellers and deflection of the surfaces), so that the vehicle will achieve precise trajectory tracking in the presence of shifting sea currents and vehicle parameter

uncertainty. In applications where precise trajectory tracking is not required, the Guidance Module is not activated. In that case, the Vehicle Control Module is responsible for achieving accurate tracking of set-points that include the vehicle's desired speed, depth and heading.

Communication System (COMS) - The Communication System controls a bidirectional link that is used by the operator to issue mission directives to the Mission Control System, and by the vehicle to relay back information regarding its internal state and/or the state of progression of the mission. Two distinct modes of operation are possible: i) via an RS232 radio link, when the vehicle is at the surface or submerged, and pulling a small buoy with an antenna; ii) via an acoustic modem, otherwise. The bidirectional acoustic link of MARIUS enables real-time communications with a support ship in shallow water, up to a distance of 3km, at a frequency of 12KHz. At the heart of the link is a digital multimodulation acoustic transmission system whose modulation schemes and baud rates can be remotely reconfigured during operation. Achievable baud rates vary from 20 bit/s using CHIRP modulation, up to 2400 bit/s using PSK (Ayela et al., 1995). For operation in shallow waters, the main difficulty facing this system is to achieve communications at distances exceeding 3 km in the face of multipath propagation, rapidly changing channel characteristics and Doppler shift.

Environmental Inspection System (EIS) - The Environmental Inspection System collects data from a suite of environmental sensors that measure conductivity, temperature, pressure, turbidity, fluorescence, oxygen and pH. A video camera is included to provide close-up images of the seabed. Data acquisition is controlled by the Mission Control System.

Data Logging System (DLS) - The Data Logging System acquires and stores internal vehicle data. The data acquired can be used for on-line system status assessment, data consistency analysis, and post-mission processing.

Mission Control System (MCS) - Based on a Mission Program obtained from a set of mission specifications, the Mission Control System sequences and synchronizes the execution of the basic vehicle system tasks that concur to the execution of that mission, and provides inbuilt recovery to vehicle and mission level faults.

2.3 Vehicle Computers

To implement the above systems, the MARIUS vehicle is equipped with an open, distributed hardware/software architecture that simplifies the task of incrementally adding sensor and actuator interfaces, as well as processing power.

Currently, the vehicle computer network includes two MC68020+FPU (microprocessor and math co-processor) based computers, together with a more advanced MC68030+FPU computer. The first two units are dedicated to navigation, guidance, control, and vehicle support management tasks. The third unit is dedicated to implementing the Mission Control System. All the computers run the OS/9 operating system, which allows for real-time multi-tasking operation, process and memory management, and interprocess communication facilities that include shared memory and events. The computer system chosen is built around the proven Motorola MC680X0 family of general processors boards, supported on the GESPAC G96 bus. The vehicle's Local Area Network (LAN) has been designed to allow easy upgrading from the current RS232 19.2 Kbaud point to point communication network to an industrial standard Bit Bus, and to an ethernet network at a later stage.

3 Mission Control of Underwater Robotic Systems: A General Framework for System Design and Implementation

This section describes briefly a general framework for the design and implementation of Mission Control Systems for Underwater Robotic Systems that is well rooted in the area of Discrete Event System theory (Cassandras, 1993). See also (Silva et al., 1995; Silva, 1996) for complete details and (Oliveira et al., 1996) for an abridged version. The framework described arose in the course of designing a Mission Control System for the MARIUS AUV, as the need for a solid foundation to system design became a matter of great concern. The work described was strongly influenced by and builds upon the results obtained a number of researchers in the field. The reader will therefore find that the nomenclature and the structure for Mission Control proposed will at times reflect the inspiring influence of the excellent body of work conducted at INRIA, France (Espiau et al., 1995; Simon et al., 1993) and at the NPS, USA (Healey et al., 1996).

The presentation of the Mission Control System structure is motivated with simple examples and leads, in a crescendo, to an organizational diagram that captures the interaction among such entities as *System Tasks*, *Vehicle Primitives* and *Mission Procedures*, which are defined in the sequel. Vehicle Primitives and Mission Procedures can be designed and implemented using two programming environments named CORAL and ATOL, which provide the mechanisms for mission execution that rely on successive transitions among system states, driven by asynchronous events. The design and analysis of both entities build on the theory of Petri nets, which are naturally oriented towards the modeling and analysis of asynchronous, discrete event systems with concurrency (Murata, 1989). Petri nets were first conceived to formally study the mechanisms of communications between asynchronous components of a computer system. Since then, they have found widespread use in the design and analysis of real-world systems in the areas of manufacturing, networking and software engineering, as well as in robotic applications, see for example (Cassandras, 1993; Peterson, 1981; Saridis, 1989; Freedman, 1991).

In practice, Petri nets exhibit both advantages and disadvantages over state automata for the modeling of discrete event systems. As discussed in (Cassandras, 1993), the most suitable approach to system modeling will very much depend on the specific application. However, Petri nets do exhibit very interesting properties that make them specially attractive for a structured approach to system modeling. The following three properties are specially relevant (see (Cassandras, 1993) for an in-depth discussion):

- Petri nets allow for the modeling of discrete event systems where the transitions between events are enabled according to arbitrarily complex rules; furthermore, they are naturally oriented towards the analysis of asynchronous, discrete event systems with concurrency.

- Petri nets are a convenient tool to decompose or modularize potentially complex systems. In fact, combining multiple systems can be often reduced to a simple operation whereby a set of original nets is kept unaltered, and only a few places/transitions are added to represent the coupling effects among the original systems. This is in striking contrast to state automata, where the combination of multiple systems often increases the complexity of the global state model. Using Petri nets, the individual system components can be easily identified and the level of their interactions displayed clearly, thus simplifying the task of incremental modeling.

- Petri net theory provides well developed analysis methods, such as invariants and reachability trees, which can lead to useful tools for the detection of potential anomalies in the behaviour of discrete event systems.

In what follows, it is assumed that the reader is familiar with the necessary background material on Petri net theory, as applied to the study of Discrete Event Systems. See for example (Cassandras, 1993; Murata, 1989) for excellent introductions to the subject and (Oliveira et al., 1996) for a short summary of the most important concepts.

System Tasks. The concept of System Task arises naturally out of the need to organize into distinct, easily identifiable classes, the algorithms and procedures that are the fundamental building blocks of a complex Underwater Robotic System. For example, in the case of an AUV, it is convenient to group the set of all navigation algorithms that process motion sensor data into a Navigation Task that will be responsible for determining the attitude and position of the vehicle, and their respective rates. A different task will be responsible for implementing the procedures for multi-rate motion sensor data acquisition. In practice, the number and type of classes adopted is dictated by the characteristics of the Robotic System under development, and by the organization of its basic functionalities, as judged appropriate by the Robotic System designer. These considerations lead naturally to the following definition:

A Vehicle System Task (abbv. *System Task - ST*) *is a parametrized specification of a class of algorithms or procedures that implement a basic functionality in an Underwater Robotic System.*

The implementation of a System Task requires the interplay of two modules: *i)* a *Functional module* that contains selected algorithms and procedures and exchanges data with other System Taks and physical devices, and *ii)* a logical *Command module*, embodied in a finite state automaton, that receives external commands, produces output messages, and controls the selection of algorithms, procedures, and data paths to and from the Functional module.

In the case of the Navigation Task mentioned above, the Functional module may contain a set of algorithms that are selected according to the type of motion data available, and the type of precision required. Data are received from a specific task devoted to motion sensor data acquisition, and output to the task that implements the guidance and control algorithms.

The design of the Functional module is carried out using well known tools from such diverse fields as navigation, guidance and control, instrumentation and measurements, communication theory, and computer science. The design of the Command module amounts to specifying a finite state automaton (Cassandras, 1993) that deals with the logical aspects of the System Task.

Vehicle Primitives. The concept of Vehicle Primitive plays a central role in the general framework for Mission Control System design described in this paper. A Vehicle Primitive corresponds to an atomic, clearly identifiable mode of operation of an Underwater Robotic System, and constitutes the basic building block for the organization of complex robot missions. A Vehicle Primitive will require, for its implementation, the coordinated execution of a number of concurrent System Tasks.

As an illustrating example, consider the case of an AUV mission that consists of a seabed survey along a single track. The mission described can be broken down into a number of Vehicle Primitives which include, among others, those in charge of keeping a constant vehicle speed, maintaining a desired heading, holding a fixed altitude over the seabed, and taking video images of the seabed at pre-assigned time intervals. In particular, the Vehicle Primitive for speed keeping will coordinate the execution of the vehicle system tasks that are responsible for measuring the vehicle's speed, running a local speed control loop, controlling the thrust delivered by the propellers, and providing the required mechanical, hardware and software resources. The above set of considerations motivate the following definition:

A Vehicle Primitive (VP) is a parameterized specification of an elementary operation mode of an Underwater Robotic System. A Vehicle Primitive corresponds to the logical activation and synchronization of a number of System Tasks that lead to a structurally and logically invariant behavior of an underwater robot.

Associated with each Vehicle Primitive, there are sets of *pre-conditions* and *resource allocation* requirements that must be met in order for the Primitive to be activated, as well as a set of Vehicle Primitive *errors*. During operation, a Vehicle Primitive will generate messages that will trigger the execution of a number of System Tasks. The conditions that determine the occurrence of those events are dictated by the logical structure of the Vehicle Primitive itself, and by the types of messages received from the underlying Vehicle System Tasks. The normal or abnormal termination of a Vehicle Primitive will generate a well defined set of *post-conditions* that are input to other Vehicle Primitives, and will release the resources that were appropriated during its execution.

By exploring the use Petri nets for the modeling of discrete event systems (Cassandras, 1993), it is possible to show that a Vehicle Primitive can be embodied in a Petri net structure defined by the five-tuple $(P_{VP}, T_{VP}, A_{VP}, w_{VP}, \mathbf{x}_{VP_0})$, where P_{VP}, T_{VP}, and A_{VP} denote sets of places, transitions, and arcs respectively, w_{VP} is a weight function, and $\mathbf{x}_{VP_0}$ is the initial Petri net marking. The set of places P_{VP} can further be decomposed as $P_{VP} = P_{pre} \cup P_{res} \cup P_{err} \cup P_{loc} \cup P_{pos}$, where $P_{pre}, P_{res}, P_{err}, P_{pos}$, and P_{loc} denote the subset of places that hold information related to the pre-conditions, resource allocation, errors, post-conditions, and the remaining state of the Petri net, respectively.

In the structure described in (Oliveira et al., 1996), the firing of a generic transition in the Petri net will start the execution of a System Task, which is called through an header with the structure

$$STname(Dmode, Din_{st}, P_m)$$

where $STname$ is the System Task name, D_{mode} is a string of input data that specifies the particular algorithm or procedure to be executed, and Din_{st} is a set of numerical data that are input to the algorithms or procedures. In the case of the task that implements the vehicle control system, $Dmode$ may select a particular algorithm for depth control, in which case Din_{st} will contain the reference for depth in meters. The last calling parameter plays a key role in merging System Tasks with Vehicle Primitives, by indicating a finite number of places in the Petri net that must be marked according to the type of output messages issued by the System Task automaton.

Based on the framework introduced, a Vehicle Primitive programming environment named CORAL has been developed. The left side of Fig. 3 depicts the organization of the CORAL software tools that are available to *edit* and *generate* a *Library of Vehicle Primitives* which implement the complete set of atomic actions required for a specific Underwater Robotic System. Each Vehicle Primitive, embodied in its equivalent Petri net, can be input either graphically via a CORAL graphic input interface, or via a textual description using the declarative, synchronous language CORAL. A CORAL compiler/linker implemented through an LR1 parser (Aho, 1985) is in charge of accepting the Vehicle Primitive textual descriptions, and producing a Vehicle Primitive Library that is an archive containing the syntax and semantic descriptions of all Vehicle Primitives, as well as the data sets required for their execution.

In order to run the Vehicle Primitives described before, a CORAL Engine has been developed that accepts Vehicle Primitive descriptions and executes them in real-time (Silva et al., 1995).

Mission Procedures/Mission Programs. Given a mission to be performed by an Underwater Robotic System, the generation of the corresponding mission plan requires the availability of a set of entities aimed at specifying *Robot Actions* at a number of abstraction levels. Those entities -

henceforth referred to as *Mission Procedures* - allow for modular Mission Program generation, and simplify the task of defining new mission plans by modifying/expanding existing ones.

As a motivating example, consider the case of a simple mission scenario where an underwater vehicle is required to perform a survey operation at constant speed and depth, along a square shaped path with a given length. Clearly, the mission described can be performed by executing four times an Action whereby the vehicle is asked to follow an horizontal path at constant speed, depth, and heading, during a pre-specified period of time. With this mission structure, the set-point for heading changes 90 degrees each time the Action is executed. The Mission Procedure that specifies the above Robot Action should allow for the parameterization of speed, depth, heading, and Action time, and for a clear definition of the mechanisms that coordinate the Vehicle Primitives required for its execution.

The above introduction motivates the following definition:

A Mission Procedure is a parameterized specification of an Action of an Underwater Robotic System. A Mission Procedure corresponds to the logical and temporal chaining of Vehicle Primitives - and possibly other Mission Procedures - that concur to the execution of the specified Action.

According to the definition, the execution of a robot mission entails the execution of a number of well defined Actions specified by Mission Procedures, which in turn synchronize the operation of Vehicle Primitives. In practice, the activation of Mission Procedures and Vehicle Primitives will be triggered by conditions imposed by the mission plan structure, and by messages received from the underlying Vehicle Primitives during the course of the mission.

Clearly, Mission Programs can in principle be embodied into - higher level - Petri nets that implement the necessary Mission Procedure structures. This has in fact been done in the case of the MARIUS vehicle, as explained in Section 4.3. Using that approach, the CORAL software environment can be used to program missions graphically, and to execute them using the CORAL Engine. To do that, a mechanism similar to that used to link Vehicle Primitives and System Taks was developed. The firing of a transition at the Mission Procedure level will start the execution of a Vehicle Primitive, which is called through an header with the structure

$$VPname(Din_{vp}, P_m),$$

where $VPname$ is the Vehicle Primitive name, and Din_{vp} is a set of numerical data that are input to the Vehicle Primitive. The last calling parameter plays a key role in binding Mission Procedures and Vehicle Primitives, by indicating an ordered set $P_m = (p_1, p_2, ..., p_n)$ of places in the Petri net for the Mission Procedure that must be put in correspondence with an ordered set of n places in the Vehicle Primitive Petri net. The latter set is well-defined, as it is explicitly listed in the matching header that is part of the Vehicle Primitive description, see the example in Section 4. *The binding described becomes effective from the time the Vehicle Primitive is called until a new call is executed.*

An header with the same structure, and leading to the same type of binding, can be used to call Mission Procedures from a general Mission Program. The header is simply written as

$$MPname(Din_{mp}, P_m),$$

where $MPname$ is the Mission Procedure name, and Din_{mp} is a set of numerical data that are input to the Mission Procedure. The meaning of the last calling parameter is similar to that used in the calling of a Vehicle Primitive.

The analysis of even a simple mission plan that is programmed using the methodology described will convince the reader that the complexity of the resulting Petri net structure can become unwieldy. See Section 4 for a detailed example. Furthermore, the approach described does not lend

itself to capturing situations where the mission plan includes logical, as well as procedural statements (e.g., do loops for the repeated execution of Mission Procedures and Vehicle Primitives, etc.). These considerations motivated the need to define and develop a specific environment for Mission Program/Mission Procedure design and implementation, named ATOL, which relies on a reactive synchronous programming language as a way to manage the potential complexity introduced by the occurrence of large Petri net structures.

The framework for Mission Control System design and implementation proposed in this paper leads naturally to the general structure of Fig. 3 (right side), which captures the interaction among System Tasks, Vehicle Primitives, and Mission Program/Mission Procedures, at both programming and run-time. In the figure, the Human/Machine Interface provides the user with a text editor, and an on-line checking mechanism for the syntax and semantics of ATOL statements.

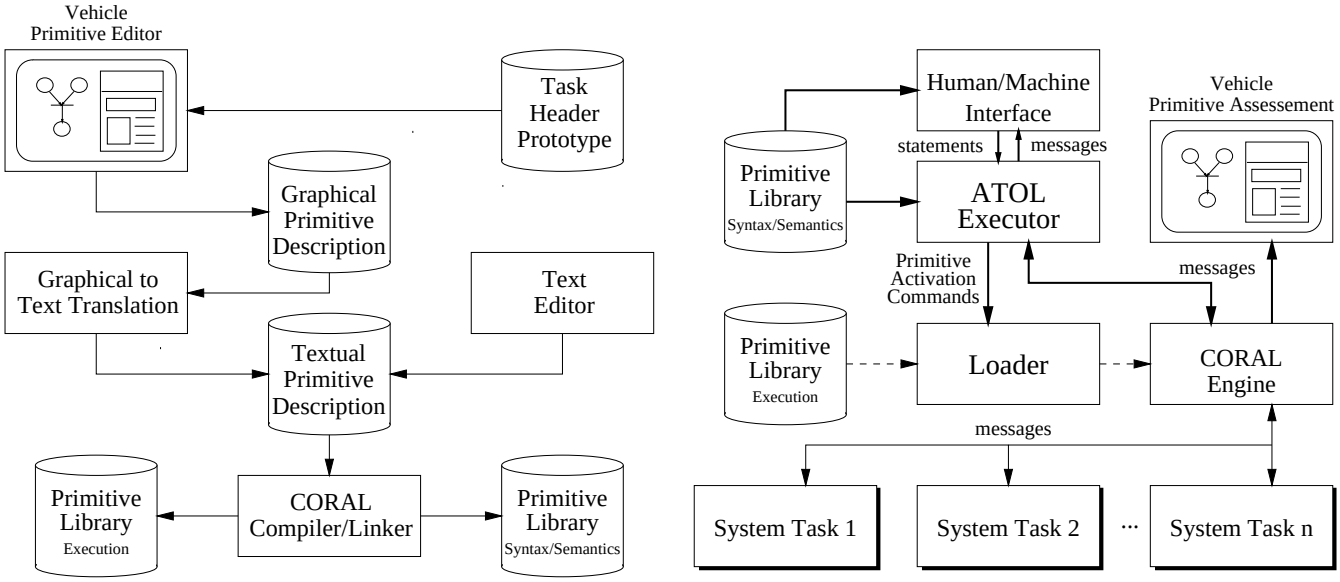

Figure 3: Mission Control System Organization.

From an execution point of view, the ATOL Executor - running an ATOL Mission Program - issues commands to the CORAL Loader, which transfers selected Vehicle Primitive descriptions from the Vehicle Primitive Library to the CORAL Engine. The Engine runs the Primitives selected by interacting with the System Tasks, and issues messages that condition the execution of the ATOL Mission Program. During mission execution, and should a proper communication channel between the vehicle and a central command station be available, the status of any Vehicle Primitive can be displayed on a *Vehicle Primitive Assessment module* that allows visualizing the flow of markings in the corresponding Petri nets.

4 Mission Control of the MARIUS AUV: Mission Programming

This section aims at bridging the gap between the theoretical framework of Section 3 and the practical aspects of Mission Control, by describing the basic System Tasks and Vehicle Primitives implemented for MARIUS, and explaining how a mission can be programmed using the CORAL

software environment. The mission described is simple, yet it captures the key steps involved in mission programming using a graphical editor. The same mission will be revisited in Section 5, when reporting the results of the tests at sea. It is important to point out that the mission described can be programmed in a much more elegant and efficient manner using the ATOL software environment (Silva et al., 1995). However, that will not be done here due to space limitations.

4.1 System Tasks

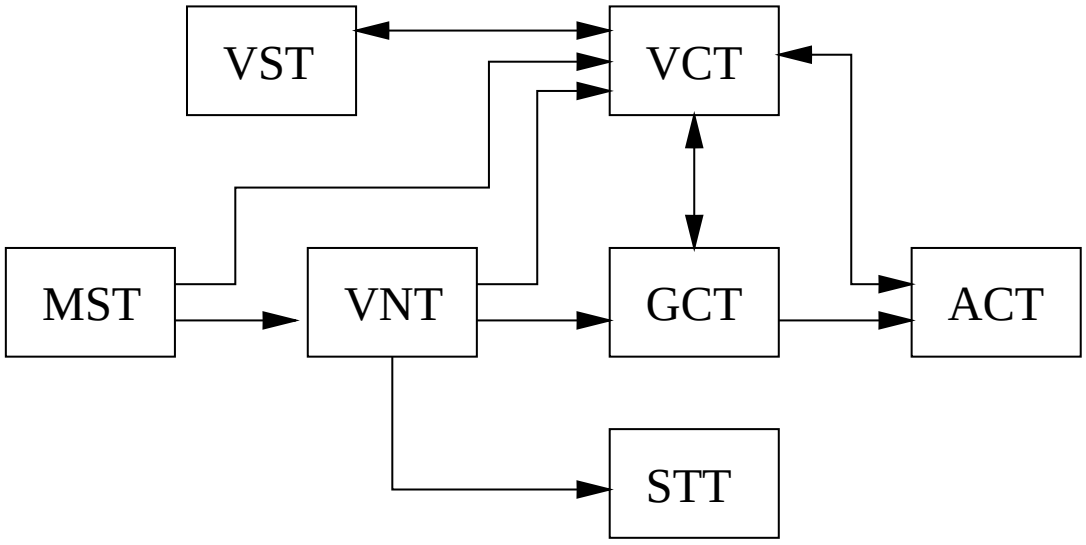

Figure 4: Vehicle System Tasks: data communication paths.

The System Tasks of MARIUS and their data interconnection structure are depicted in Fig. 4, which should be compared against the diagram of Fig. 2. The following System Tasks can be identified:

- *Vehicle Support Task (VST)*
- *Actuator Control Task (ACT)*
- *Vehicle Navigation Task (VNT)*
- *Motion Sensor Task (MST)*
- *Guidance and Control Task (GCT)*
- *Vehicle Communications Task (VCT)*
- *Space and Time Task (STT)*
- *Vehicle Log Task (VLT)*

As expected, many of the System Tasks can be put in close correspondence with the blocks that appear in the Vehicle System Organization of Fig. 2, as they specify classes of algorithms and procedures that implement basic vehicle system functionalities. For that reason, their formal

definitions will not be given here, as they will become clear from the context. However, the following System Tasks deserve special consideration:

Motion Sensor Task (MST) - The Motion Sensor Task manages the operation of all motion sensing instrumentation packages installed on-board the vehicle. This task reads data from selected sensor suites, and routes it to the Vehicle Navigation Task and to the vehicle operator through the Vehicle Communication Task.

Space and Time Task (STT) - The Space and Time Task is used to provide space and time Mission Control Program synchronization. It relies on the availability of navigational data provided by the Vehicle Navigation Task, and on the vehicle operating system real time resources.

Vehicle Log Task (VLT) The Vehicle Log Task manages the logging of relevant data inside the vehicle. This task receives data packets from the other vehicle tasks and stores them for post-mission analysis.

As explained in Section 3, each of the abovementioned System Tasks can be called using a calling header of the type $STname(Dmode, Din_{st}, P_m)$. In the case of the Guidance and Control System Task (GCT), for example, the header is of the form

$$GCT(YAW\_AUTO, \psi, p_{yaw\_auto}),$$

where YAW_AUTO selects a particular mode of operation that implements an automatic control loop for yaw, and ψ is the yaw set-point. The symbol p_{yaw_auto} denotes a place in the calling Petri net that will be marked when a specific message issued by the System Task's automaton acknowledges that the mode of operation requested has been entered. For the sake of brevity, the headers for the other System Tasks are not described here. However, their meaning will be clear in the examples that follow; see also (Oliveira et al., 1996).

The Task structure of Fig. 4 is supported by the MARIUS's distributed computer architecture described in Section 2.3. The connections shown were implemented using a message passing mechanism with asynchronous writing, synchronous reading type protocols. Each System Task is in close correspondence with two OS9 operating system processes, which implement the finite state automaton of the task Command module and the task algorithms and procedures, respectively. The two processes communicate with each other using the shared memory process communication mechanism.

4.2 Vehicle Primitives

A selected set of Vehicle Primitives included in the Mission Control System of MARIUS is described in the sequel. For the sake of brevity, only the Vehicle Primitive in charge of implementing the control loop for yaw is explained in detail, at the end of the section.

Init(p_{End}, p_{Abort}) - This primitive is in charge of initializing all vehicle System Tasks. The execution of an *Init*(.,.) command drives the state of every System Task automaton to STANDBY, and switches the operation mode of the vehicle to manual. At that point, an operator can command directly the vehicle thrusters and control surfaces, and examine sensor and actuator data transmitted via a radio link, and displayed on a command console. An *Init*(.,.) command will also make available to other Vehicle Primitives a number of vehicle resources that include the rudders, elevator, ailerons, and thrusters. This is done by *marking a set of global places that can be shared by the Petri nets embodying those Vehicle Primitives* (see (Oliveira et al., 1996) for the definition of global places in CORAL). The symbols p_{End} and p_{Abort} denote

places that hold information related to the successful conclusion of the Primitive execution, and to the occurrence of an ABORT condition in any of the System Tasks, respectively.

KeepSpeed(v,p_{Exit},p_{End}) - The *KeepSpeed* Primitive is responsible for keeping the forward speed of the vehicle at a desired set-point. To implement the required speed control loop, this Vehicle Primitive coordinates the execution of the System Tasks that are in charge of motion sensor data acquisition (*MST*), navigation (*VNT*), dynamic control (*GCT*), and actuator control (*ACT*). The Primitive requests the two back thrusters as resources required for its execution, and releases them after completion for possible use by other Vehicle Primitives. The calling parameters v, p_{Exit} and p_{End} consist of the set-point for speed, a Petri net place that must be marked to exit the Primitive execution, and a place that is marked at the end of the Primitive execution, respectively.

KeepDepth(z,p_{Exit},p_{End}) - The *KeepDepth* Primitive is responsible for driving the depth coordinate of the vehicle to a desired set-point. This Primitive calls the same System Tasks as the KeepSpeed Primitive. The resources required are the elevator and the ailerons. The calling parameters z, p_{Exit} and p_{End} consist of the set-point for depth, a place that is marked to exit the Primitive execution, and a Petri net place that shall be marked at the end of the Primitive execution, respectively.

KeepHeading(ψ,p_{Exit},p_{End}) - The *KeepHeading* Primitive is responsible for driving the heading of the vehicle to a desired set-point. Its structure, which is similar to that of the *KeepSpeed* and *KeepDepth* Primitives, will be explained in detail later in the text.

ControlDataLog(p_{Exit},p_{End}) - The *ControlDataLog* Primitive is in charge of logging the vehicle's feedback control loop data. The System Tasks involved in this Vehicle Primitive are those responsible for motion sensor data acquisition (*MST*), navigation (*VNT*), dynamic control (*GCT*), actuator control (*ACT*), and general data logging (*VLT*). The calling parameters consist of a Petri net place (p_{Exit}) that must be marked to stop the Primitive execution, and a place (p_{End}) that is marked at the end of the Primitive execution.

Reset(p_{End}) - The *Reset* primitive can be called *at any time* to reinitialize all vehicle System Tasks. Its execution drives the state of all System Task automata to STANDBY, and switches the operating mode of the vehicle to manual. Furthermore, it makes all vehicle resources available for later use. The symbol p_{End} denotes a place that is marked at the end of the Primitive execution.

The Vehicle Primitive in charge of controlling the heading of the vehicle is now described in detail, by referring to the corresponding Petri net model of Fig. 5. In the figure, transitions drawn in black are associated with commands that are sent to System Tasks.

Vehicle Primitive KeepHeading: Petri net structure. The calling header

$$VPname(Din_{vp}, P_m) := KeepHeading(\psi, p_{Exit}, p_{End})$$

clearly identifies a numerical set point for heading (ψ), a Petri net place that must be marked to stop the Primitive execution, and a place p_{End} that will hold information related to the termination of the Primitive execution. The only *pre-condition* for the execution of the Vehicle Primitive is the initialization of all System Tasks (see the *Init* command). There are no post-conditions, and the

required Primitive *resource* is the rudder. Following the notation of Section 3, $P_{pre} = \{p_{Init}\}$, and $P_{res} = \{p_{Rudder}\}$. It is important to stress that p_{Init} and p_{Rudder} are *global places*, which are initially marked by the *Init* Vehicle Primitive. Five basic phases can be identified in the *KeepDepth* Vehicle Primitive:

During phase 1, the pre-condition is checked and the required resource is requested. In fact, the first transition is enabled only if the places p_{Rudder} and p_{Init} are marked.

Phase 2 configures the vehicle System Tasks that are required to implement the heading controller. The *VCT* is asked to disable direct command of the rudder from the console, and the *ACT* is enabled to receive the rudder commands directly from the yaw controller (*AUTO_RUDDER*). In parallel, the *GCT* is requested to switch on the heading controller (command *YAW_AUTO*), and to accept the set-point for ψ given in the System Primitive call. Finally, the *MST* is called to output the yaw measurements periodically to the *GCT*. Notice that in this case there is no need to activate the Navigation Task, as the yaw measurement is directly available from the motion sensor unit installed on-board the vehicle.

In phase 3, the vehicle runs the closed loop control system that is in charge of keeping the heading of the vehicle at the desired set point, until the place p_{Exit} is marked externally.

Phase 4 sets the vehicle heading operation mode back to manual. This is done by issuing a set of commands to the System Tasks activated during phase 2, but in reverse order.

In the last phase of execution of the Primitive, the resource requested at the beginning is released. This is done by marking the place p_{Rudder} (notice that two places with the same label were used, to simplify the drawing). The place p_{End} is marked to signal the end of the Vehicle Primitive.

4.3 Mission Programming using CORAL

The mission example described here consists of tracing a square shaped trajectory, at constant depth and speed of 1.35 m and 2.0 m/s, respectively. The square maneuver is obtained by requesting the vehicle to change its heading by -90 deg every 40 seconds. The initial heading is 0 deg.

The design of the Mission Program involves a Mission Procedure named *HorizPath*, whose implementation using the CORAL programming environment is shown in Fig. 6. This Mission Procedure parametrizes the action of keeping constant heading ψ, depth z, and speed u of the vehicle, for a period of time t. The corresponding calling header is $MPname(Din_{vp}, P_m) := HorizPath(t, z, u, \psi, p_{MPEnd})$.

The *HorizPath* Mission Procedure starts by setting a timer to generate a timeout after the required execution time has elapsed. This is done by issuing directly to the Space and Time Task an STT command with the required Mission Procedure duration time t. To perform the maneuver, three Vehicle Primitives are called in parallel: *KeepSpeed* with a velocity set-point u, *KeepDepth* with a depth set-point of z, and *KeepHeading* with a heading set-point of ψ. The generation of a timeout terminates the execution of *HorizPath* by exiting the three Vehicle Primitives.

The Mission Program can be explained with the help of Fig. 7, which shows four distinct phases:

In phase 1, all vehicle System Tasks are initialized by calling the *Init* Vehicle Primitive.

In phase 2, the *HorizPath* Mission Procedure is called for a period $t = 20$ s, with a velocity set-point of $u = 2$ m/s, a depth set-point of $z = 1.35$ m, and an heading set-point of $\psi = 0$ deg. At the end of this phase, the vehicle is headed north, and ready to start the required square maneuver.

Phase 3 calls the *HorizPath* Mission Procedure repeatedly, with heading set-points of 0 deg, -90 deg, -180 deg, and -270 deg, while maintaining the remaining input set-points equal to those

in phase 2. The required duration of each Mission Procedure call is $t = 40\,\mathrm{s}$. In parallel, the Vehicle Primitive *ControlDataLog* is called to start logging control loop data for later off-line analysis.

Finally, in phase 4 the vehicle is placed in manual mode. The logging of control loop data is stopped, and the mission ends normally if no errors are reported. Should an error occur during the mission, the place p_{Abort} will be marked, and a *Reset* command will be issued. This will bring the vehicle to the default manual mode, and the mission is aborted.

5 Mission Control of the MARIUS AUV: Tests at Sea

In order to assess the performance of the Mission Control System of MARIUS, a series of tests were conducted at sea in Sines, Portugal, in January 1996. The tests included programming and running the mission described in Section 4.3. Throughout the mission, the vehicle pulled a buoy with an antenna, thus enabling radio communications between the vehicle and a shore station. The software for Vehicle and Mission Control was run on the computer network installed on-board the AUV. The shore station consisted of two IBM PCs running the MS Windows multi-task operating system, and a Vehicle Command Console. A man-machine interface named MUCIS (MARIUS-User Command Interface System) was developed for the tests, to enable manual remote control when required, and to assess the internal state of the vehicle and the state of progression of the mission during mission execution, see Fig. 8. The PC dedicated to mission control follow-up enabled the display of the mission Petri net network, together with the respective marking sequences. Figures 9 through 12 display some of the data acquired in the course of the mission, which was executed to perfection. Figures 9 and 10 show the commanded and measured heading, and the rudder activity, respectively. Figures 11 and 12 show the slight variations in heading and depth caused by the wave action in shallow water.

6 Conclusions

The paper introduces a general framework for the design and implementation for Mission Control Systems for underwater robotic systems, and describes its application to the development of a Mission Control System for the MARIUS AUV. The main thrust of the research and development reported is twofold: i) introduction of a new framework for Mission Control System design and implementation that builds on the theory of Petri nets, and ii) development of special software environments that provide the tools that are necessary to go through all the stages of mission programming, followed by automatic generation of target code that runs on the vehicle computer network. The general methodology adopted for Mission Control System design and implementation was evaluated during sea trials, in the course of a mission that was programmed and run using the CORAL software programming environment. The results obtained show the usefulness and reliability of the system developed. However, they also show that further work is necessary to overcome the major limitations of the current Mission Control System, namely, the difficulty in managing the potential complexity introduced by the occurrence of large Petri nets, and the impossibility of programming missions using logical, as well as procedural statements. These issues are currently being studied, and have so far led to the complete specification of a new software programming environment named ATOL, which relies on a reactive synchronous programming language as a way to overcome the limitations described. Work is now under way to complete the development of ATOL. Future work will address the very important problems of mission state assessment, and on-

line mission modification based on the incorporation of failure detection mechanisms and feedback loops at all levels of Mission Control.

7 Acknowledgements

This work was supported by the Commission of the European Communities under the contract MAS2-CT92-0021 of the Marine Science and Technology (MAST) - II Programme, and by the Portuguese PRAXIS Programme under contract 3/3.1/TPR/23/94.

We would like to thank António Aguiar, Pedro Encarnação, Francisco Freire, Daniel Fryxell, Luis Sebastião, and Manuel Rufino from IST for their valuable contribution to the development and testing of the MARIUS vehicle, and Prof. Anthony Healey from the NPS, Monterey, California, for his generous advice during some of the crucial test phases. We gratefully acknowledge the support of the Administração do Porto de Sines (APS), who spared no efforts to provide the facilities for vehicle transportation, launching and testing. Our warmest thanks go to Eng. Edgar Pacheco, Cmt. Aureliano, Cmt. Fontes, and Cmt. Taveira, from the APS, who participated actively in the preparation of the tests at sea. Finally, we express our deep appreciation for the work carried out by the APS crane operators, Mr. Mário and Mr. Delgado.

References

Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D., 1985, *Compilers Principles, Techniques and Tools*, Addison-Wesley Publishing Company.

Albus, J., 1988, System Description and Design Architecture for Multiple Autonomous Undersea Vehicles, National Institute of Standards and Technology, Technical Note 1251, September 1988.

Antsaklis, P., Passino, K., 1993, *An Introduction to Intelligent and Autonomous Control*, Kluwer Academic Publishers.

Ayela, G., Bjerrum, A., Bruun, S., Pascoal, A., Pereira, F-L., Petzelt, C., Pignon, J-P., 1995, Development of a Self-Organizing Underwater Vehicle - SOUV, *Proceedings of the MAST-Days and Euromar Conference*, Sorrento, Italy, pp. 1253–1269.

Cassandras, C., 1993, *Discrete Event Systems. Modeling and Performance Analysis*, Aksen Associates Incorporated Publishers.

Coste-Maniere, E., Wang, H., Peuch, A., 1995, Control Architectures: What's Going On?, *Proc. US/Portugal Workshop on Undersea Robotics and Intelligent Control*, Lisbon, Portugal,pp. 54–60.

Espiau, B., Kapellos, K., Jourdan, M., Simon, D., 1995, On the Validation of Robotic Control Systems, Part I: High Level Specification and Formal Verification, Internal Report N0. 2719, INRIA, November 1995.

Franklin, G., Powell, J., Workman, M., 1990, *Digital Control of Dynamic Systems*, Addison-Wesley Publishing Company.

P. Freedman, 1989, Time, Petri Nets, and Robotics, *IEEE Transactions on Robotics and Automation*, Vol. 27, No. 4, pp. 417–433.

Fu, K. S., 1970, Learning Control Systems-Review and Outlook, *IEEE Transactions on Automatic Control*, Vol.AC-15, No.2.

Healey, A., Marco, D., McGhee, R., 1996, Autonomous Underwater Vehicle Control Coordination using a Tri-Level Hybrid Software Architecture, *Proceedings of the IEEE Robotics and Automation Conference*, Minneapolis.

Lee, M., McGhee, R., Editors, 1994, *Proceedings of the IARP 2nd Workshop on Mobile Robots for Subsea Environments*, Monterey, California, May 1994.

Murata, T., 1989, Petri Nets: Properties, Analysis, and Applications, *Proceedings of the IEEE*, Vol. 77, No. 4, pp.541–580.

Oliveira, P., Pascoal, A., Silva, V., Silvestre, C., 1996, Design, Development and Testing of a Mission Control System for the MARIUS AUV, *Proceedings of the 6th IARP (International Advanced Robotics Program) Workshop on Underwater Robotics*, Toulon, France.

Pascoal, A., 1994, The AUV MARIUS: Mission Scenarios, Vehicle Design, Construction and Testing, *Proceedings of the 2nd Workshop on Mobile Robots for Subsea Environments*, Monterey Bay Aquarium, Monterey, California USA, pp. 127–140.

J. Peterson, 1981, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Inc..

Saridis, G., 1979, Towards the Realization of Intelligent Controls, *Proceedings of the IEEE*, Vol. 67, No.8, pp. 1115-1133.

Saridis, G., 1989, Analytical Formulation of the Principle of Increasing Precision with Decreasing Intelligence for Intelligent Machines, *Automatica*, Vol. 25, No.3, pp. 461–467.

Silva, V., 1996, A Real Time Mission Control System for Autonomous Vehicles, MSc. Thesis, Instituto Superior Técnico, November 1996.

Silva, V., Oliveira, P., Silvestre, C., Pascoal, A., 1995, Design and Implementation of Real Time Mission Control Systems for Autonomous Underwater Vehicles: The CORAL and ATOL software programming environments, Internal report - ISR, December 1995.

Simon, D., Espiau, B., Castillo, E., Kapellos, K., 1993, Computer Aided Design of a Generic Robot Controller Handling Reactivity and Real Time Control Issues, *IEEE Transactions on Control Systems Technology*, Vol. 1, No. 4, pp. 213–229.

Valavanis, K., Saridis, G., Pascoal, A., Lima, P., Pereira, F-L., editors, 1995, *Proc. of the Joint U.S./Portugal Workshop on Undersea Robotics and Intelligent Control*, Lisbon, Portugal, March 1995.

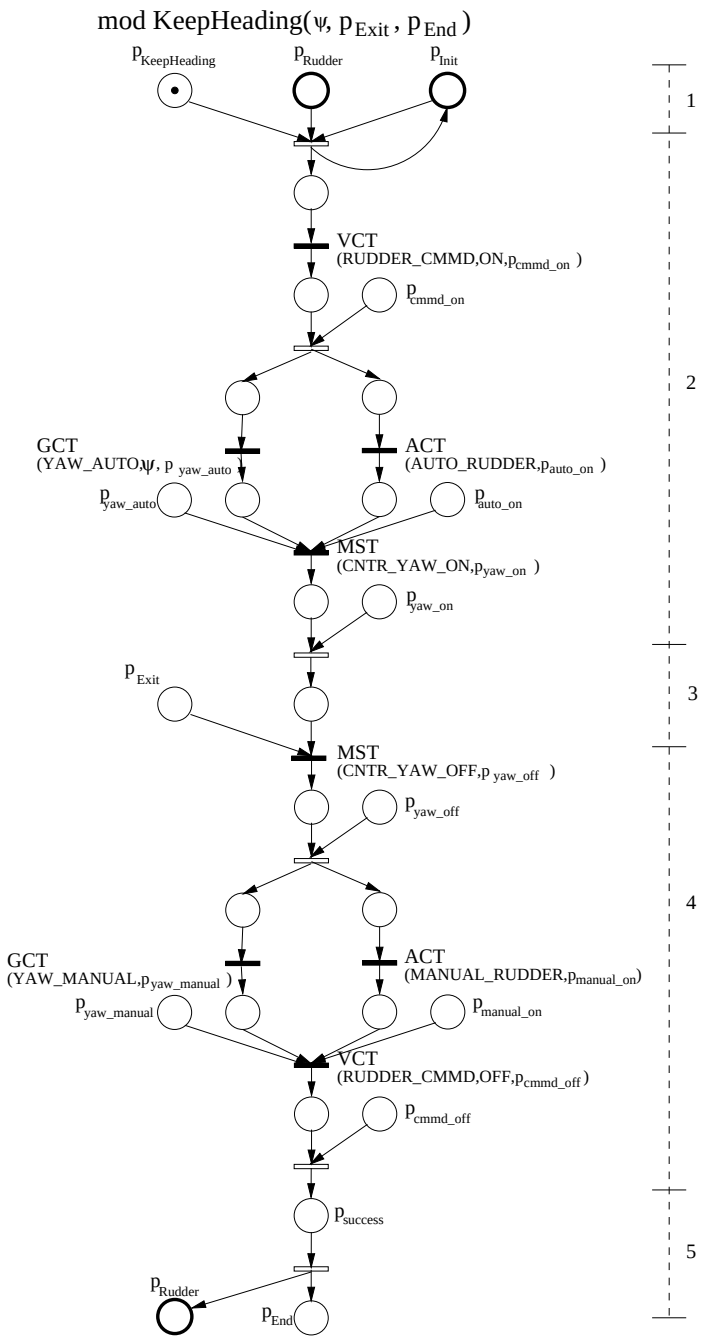

Figure 5: KeepHeading Vehicle Primitive - p_{Rudder} and p_{Init} are global places.

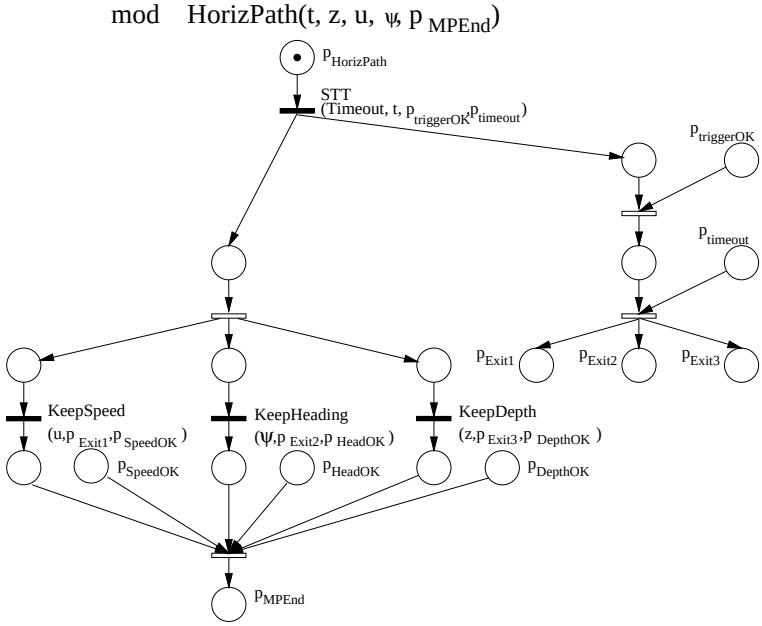

Figure 6: Mission Procedure described in CORAL.

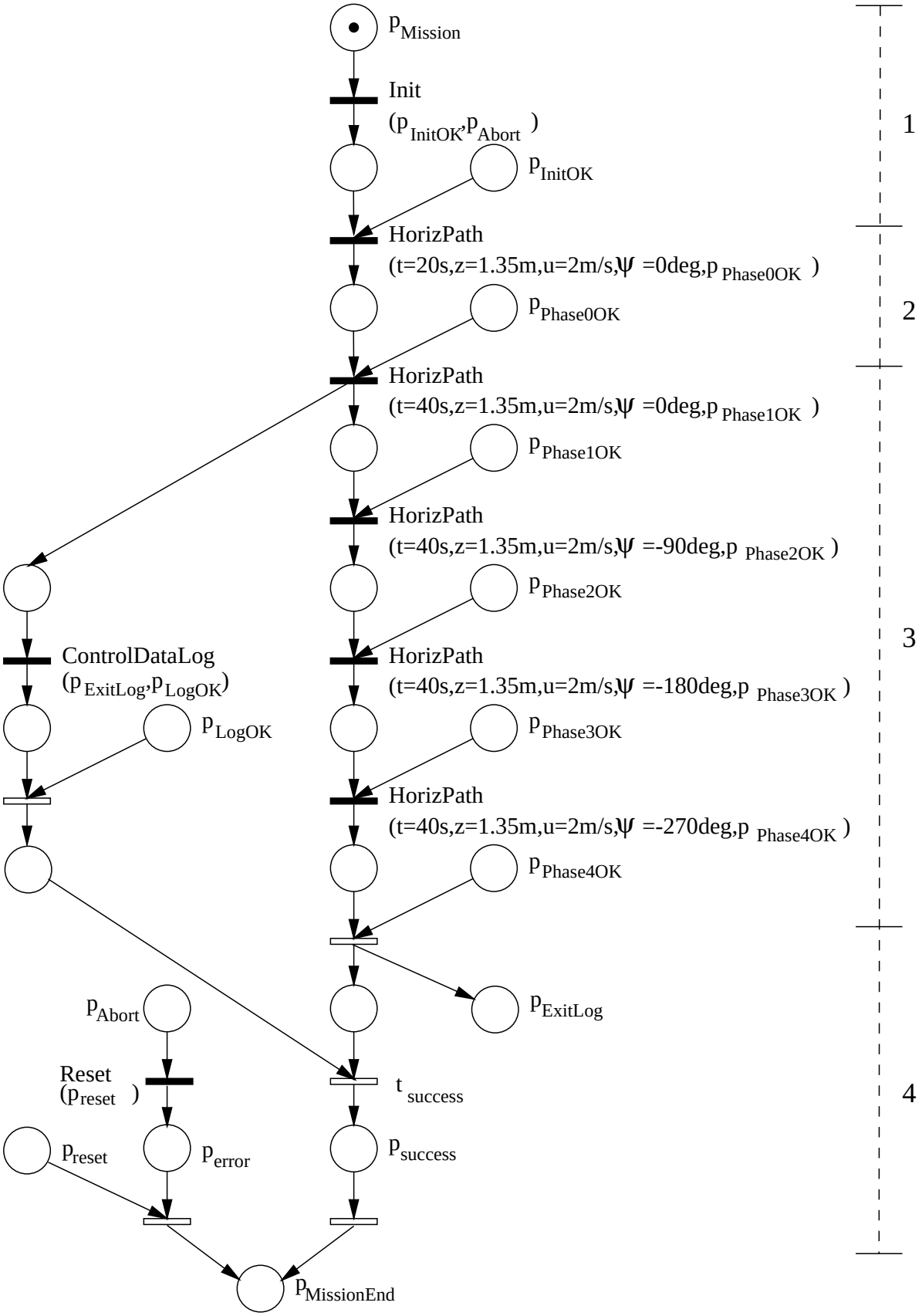

Figure 7: Mission program described in CORAL.

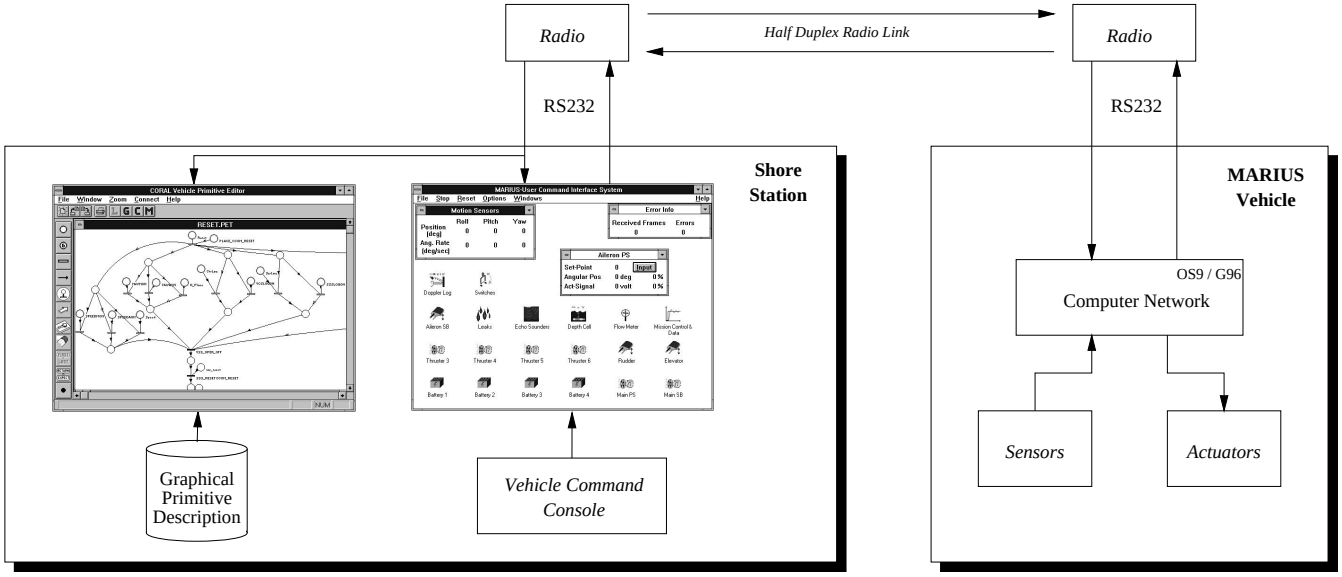

Figure 8: MUCIS - MARIUS user interface unit.

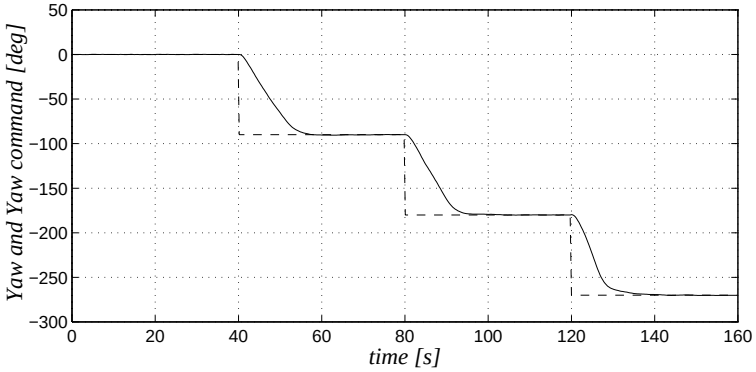

Figure 9: Commanded and measured heading.

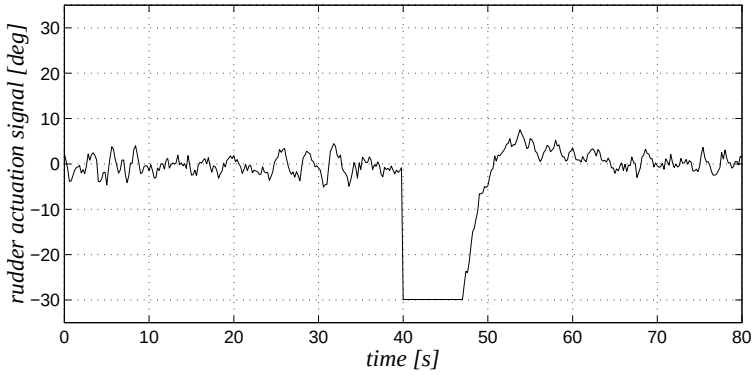

Figure 10: Rudder deflection.

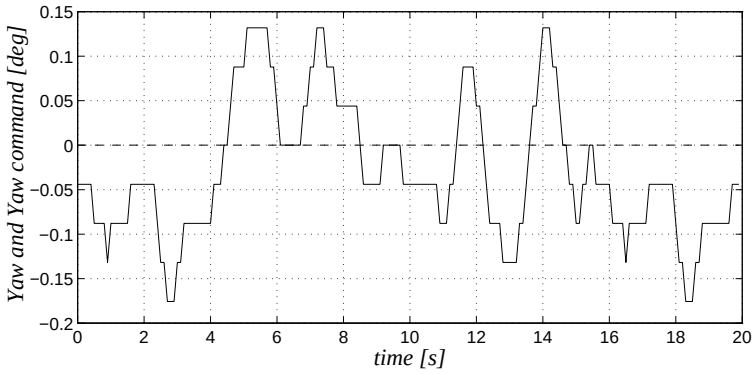

Figure 11: Measured heading (zoom in).

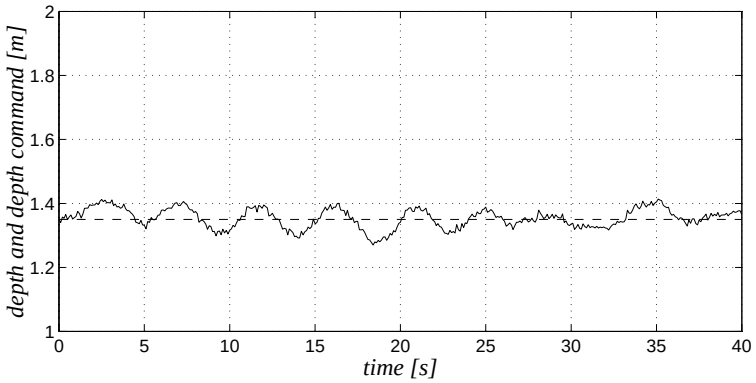

Figure 12: Commanded and measured depth.