Design, Development, and Testing of a Mission Control System for the MARIUS AUV *

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

Institute for Systems and Robotics
Instituto Superior Técnico
Av. Rovisco Pais, 1096 Lisboa Codex, Portugal
E-mail address: antonio@isr.ist.utl.pt

Abstract

This paper describes the design, development, and sea testing of a Mission Control System for the MARIUS Autonomous Underwater Vehicle (AUV). The design methodology builds on the key concept of *Vehicle Primitive*, which is a parameterized specification of an elementary operation performed by the vehicle. A Vehicle Primitive is obtained by coordinating the execution of a number of concurrent *System Tasks*. Vehicle Primitives are activated to form *Mission Procedures*, which are executed as determined by *Mission Programs*, and 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 its 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. Vehicle Primitives and Mission Procedures are developed and implemented in the vehicle's computer system using the specially designed 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.

The paper provides a brief summary of the theoretical issues addressed in the course of designing the Mission Control System for the MARIUS AUV, discusses relevant implementation details, and describes the results of a series of sea tests for system design validation carried out in Sines, Portugal.

*This work was supported by the Commission of the European Communities under contract MAS2-CT92-0021 of the MAST-II programme, and by the Portuguese PRAXIS programme under contract 3/3.1/TPR/23/94.

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 sizeable 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 [6], whereas the latter can be tackled using well established theoretical tools from the field of Continuous and Discrete-Time Dynamical Systems [10].

Early references in this vast area include the pioneering work of K.S. Fu [13], Saridis [23, 24] and Albus [1], which 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 [2] and [27], 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 [27] a large number of publications 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.), MT-88 and TUNNEL SEA LION (IMTP, Russia), and PTEROA (Japan).

A recent publication [7] 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 the underwater vehicle mission control, IST has designed a first version of a Mission Control System for the MARIUS AUV [3]. This paper provides a brief summary of the framework for design, analysis and implementation of the Mission Control System proposed, and reports the results of a series of sea tests for system validation conducted in Sines, Portugal. The work reported here 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 [17] and the references therein.

The methodology adopted by IST 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 the underwater vehicle. A Vehicle Primitive is obtained by coordinating the execution of a number of concurrent *(Vehicle) System Tasks*. Vehicle primitives are activated to form *Mission Procedures* as determined by *Mission Programs*, and 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 its 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 the specially developed 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.

At the core of the Mission Control System implementation is the 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. 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 missions and makes the task of loading and unloading different Vehicle Primitives trivial. Currently, the CORAL Engine runs on the GESPAC-based target computer network of MARIUS.

The methodology exposed was initially developed to implement Vehicle Primitives. However, it was soon realized that the same methodology could be extended to implement a first kernel of its Mission Control System. Following the methodology adopted, a Mission Program can be effectively embodied into a - higher level - Petri Net description that controls

the scheduling of Vehicle Primitives concurring to the execution of a particular mission. Furthermore, the Mission Program can be generated using the graphic approach described above. During real-time operation, the Mission Control System can report 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 organization of the paper is as follows: Section 2 describes the MARIUS AUV, and discusses 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. This section builds heavily on Petri net theory, which is briefly summarized for the sake of completeness. Section 4 discusses the problem of formal mission verification. Section 5 describes the Mission Control System of the MARIUS AUV, and illustrates the basic steps involved in the design of a Mission Program for a simple mission example. Finally, Section 6 focuses on practical issues. It describes the set-up for mission execution and mission follow-up from a shore station, and reports the results of running the mission described at sea.

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 Figure 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 [3, 20] 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 back thrusters for cruising, four tunnel thrusters for station keeping maneuvers, and rudders, elevator and ailerons 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 2.5m/s. At the speed of 1.26m/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 [4, 20], where the reader will find the description of two envisioned mission scenarios in Danish and Portuguese coastal waters, focusing 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 surface survey back to the starting point while performing an undulating maneuver to determine the spatial extension of the boundary layer. 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 [4]. Those include good platform stability and maneuverability, robustness against vehicle parameter variations, low sensitivity to external disturbances, error recovery capabilities, payload carrying capacity, 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 basic vehicle systems and their interconnections can be identified in Figure 2, which

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

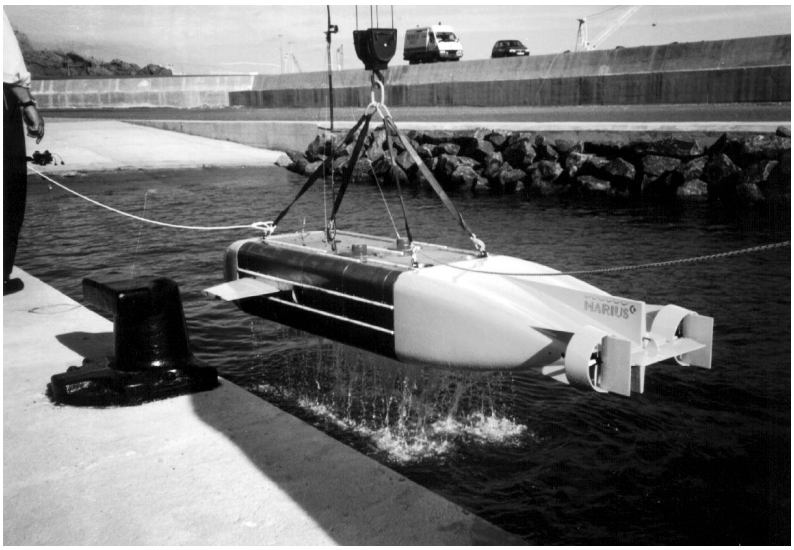

Figure 1: The MARIUS Vehicle.

(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. 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 of 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 3 km, at a frequency of 12 KHz. At the heart of the link is a digital multi-modulation 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, see [3]. 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

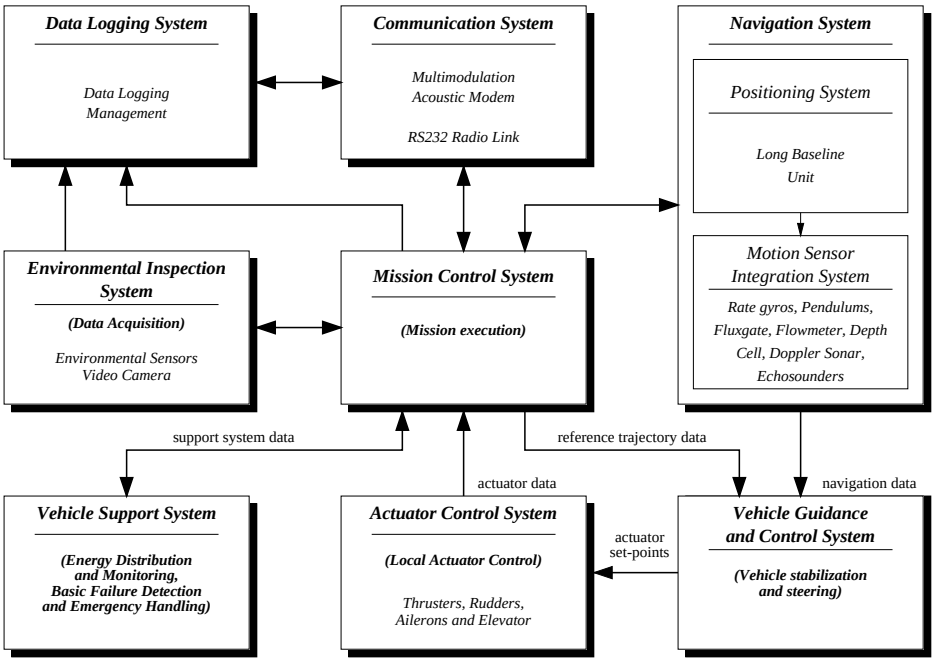

Figure 2: Vehicle System Organization.

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 stored can be used for on-line consistency analysis, and for 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.

At present, the vehicle computer network includes two MC68020+FPU (microprocessor and math coprocessor) 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 multitasking 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: System Design and Implementation

This section proposes 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. The framework proposed 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 [9] and NPS, USA [15].

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. The entities described can be designed and implemented using two specially developed programming environments named CORAL and ATOL, and provide the mechanisms for mission execution that rely on successive transitions among system states, driven by asynchronous events. For the sake of completeness, the section starts with a review of the necessary background material in the important area of Petri Net theory, as applied to the study of Discrete Event Systems.

3.1 Petri Net Theory: A Framework for the Study of Discrete-Event Systems

This section provides a brief summary of the basic concepts of Petri net theory, and how it can be applied to the study of discrete event systems (DES). The nomenclature and the style of presentation have been strongly influenced by the material in the textbooks of Cassandras [6] and Peterson [22], which contain excellent introductions to the subject.

A Petri net is a device that manipulates *events* according to well-defined rules. Since rules can be arbitrarily complex, Petri nets are naturally suited to represent a very large class of discrete event systems. In the theory of Petri nets, events are referred to as *transitions*. In order for a given transition to occur, a set of enabling conditions must be satisfied. Information related to those conditions is stored in elements called *places*. Places associated with the conditions required for a transition to occur are viewed as *inputs* to that transition. Other places, with conditions that are affected by the occurrence of a transition, are viewed as the *outputs* of that transition. In what follows, $P = \{p_1, p_2, ..., p_n\}$ denotes a finite set of places, $T = \{t_1, t_2, ..., t_m\}$ denotes a finite set of transitions, A is a set of arcs that consists of a subset of $(P \times T) \cup (T \times P)$, and $w : A \to \mathcal{Z}_+$ is a weight function that assigns positive integers (weights) to a set of arcs. A Petri net is formally defined as a four-tuple

$$(P, T, A, w).$$

The following additional notation is required: the symbol $I(t_j) = \{p_i : (p_i, t_j) \in A\}$ denotes the *set of input places* to transition t_j, while $O(t_j) = \{p_i : (t_j, p_i) \in A\}$ is the set of output places from transition t_j.

Associated with a Petri net, there is a *Petri net graph* consisting of two types of nodes: circles representing places, and bars representing transitions, see Figure 3. The arcs that connect places and transitions represent elements of the arc set A. Each arc is shown together with an integer representing its weight. The absence of an integer means that the weight is 1. Clearly, if there is an arc directed from

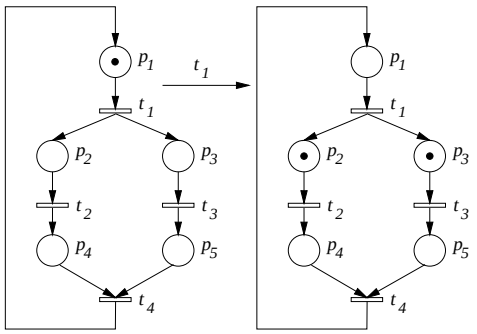

Figure 3: Example of a Petri net

p_i to t_j, then $p_i \in I(t_j)$. Similarly, an arc directed from t_j to p_i means that $p_i \in O(t_j)$.

To be of practical use, a Petri net requires a mechanism indicating whether the conditions under which events can occur are met or not. This is done by assigning *tokens* to places. If a token appears in one place, this means that the condition described by that place is satisfied. The way in which tokens are assigned to places in a Petri net is defined by a *marking function* $\mu : P \to \mathcal{N}^n$, which maps the set of places P to the n-tuple $\mathbf{x} = \mathbf{x}(P) = [x(p_1), x(p_2), ..., x(p_n)]'$ of nonnegative integers, with $x(p_i)$ denoting the number of tokens in place p_i. When using Petri net graphs, a token assigned to place p_i is indicated by a dark dot positioned in that place. The original definition of Petri net is then naturally modified to obtain a *marked Petri net*, a five-tuple $(P, T, A, w, \mathbf{x}_0)$ where (P, T, A, w) is a Petri net and $\mathbf{x}_0$ is an initial marking.

Since the execution of a Petri net is controlled by the number and distribution of tokens in the net, it is natural to define the *state* of a Petri net as its marking $\mathbf{x}$. It follows from the above definitions that the state space X of a Petri net with n places consists of all n-dimensional vectors with nonnegative integer entries. A transition $t_j \in T$ in a marked Petri net is said to be *enabled* if $x(p_i) \geq w(p_i, t_j)$ for all $p_i \in I(t_j)$, that is, if the number of tokens in each input place p_i to the transition t_j is at least as large as the weight of the arc connecting p_i to t_j. If enabled, a transition may *fire* and change the state of a Petri net by removing tokens from its input places and creating new tokens which are distributed to its output places. The motion of tokens through the net is specified by a *state transition function* $\phi : X \times T \to X$ defined, for each enabled transition t_j, by $x_{k+1}(p_i) = x_k(p_i) - w(p_i, t_j) + w(t_j, p_i), i = 1, 2, ..., n$, where $x_k(p_i)$ and $x_{k+1}(p_i)$ denote the number of tokens in place p_i before and after t_j fires, respectively. According to this definition, if p_i is an input place to the transition t_j, it loses as many tokens as the weight of the arc connecting p_i to t_j. If p_i is the output place of a transition t_j, then it gains as many tokens as the weight of the arc connecting t_j to p_i. It is important to remark that if there are several enabled transitions

in a Petri net, the firing of those transitions occurs in a non-deterministic manner.

Given a marked Petri net $(P, T, A, w, \mathbf{x}_0)$, its execution produces a *firing sequence* $\{t^1, t^2, ...\}$, where t^k denotes the k-th transition fired. The corresponding *marking sequence* is $\{\mathbf{x}_0, \mathbf{x}_1, ...\}$, where $\mathbf{x}_{k+1}$ is easily obtained from $\mathbf{x}_k, t^k$ and the transition function ϕ. For the Petri net of Figure 3, with initial marking $\mathbf{x}_0 = [1, 0, 0, 0, 0]'$, it is easily seen that the segments

$$[1,0,0,0,0]' \stackrel{t_1}{\rightarrow} [0,1,1,0,0]' \stackrel{t_2}{\rightarrow} [0,0,1,1,0]' \stackrel{t_3}{\rightarrow} [0,0,0,1,1]'...$$

and

$$[1,0,0,0,0]' \stackrel{t_1}{\rightarrow} [0,1,1,0,0]' \stackrel{t_3}{\rightarrow} [0,1,0,0,1]' \stackrel{t_2}{\rightarrow} [0,0,0,1,1]'...$$

of firing and marking sequences may occur, where the symbol above a right arrow denotes the transition that was fired. Notice that the transitions t_2 and t_3 are enabled simultaneously (parallel events). However, their firing cannot occur simultaneously. The first sequence captures the situation where transition t_1 fires earlier than t_2.

The study of the logical or qualitative behaviour of Petri nets can be carried out by resorting to rigorous analysis methods that build on the concepts introduced above, see [6, 22, 16] and the references therein. In the analysis, the definitions of liveliness and boundedness of a marked Petri net play fundamental roles. A transition t_j of a marked Petri net is said to be *live* if for every marking obtained from the initial marking $\mathbf{x}_0$, there exist subsequent firing and marking sequences in which t_j is enabled. A marked Petri net is live if every transition is live. A marked Petri net is said to exhibit a *deadlock* if there is at least one marking obtained from the initial marking $\mathbf{x}_0$ such that no transition will be enabled. Clearly, if a Petri net is live, then it does not exhibit deadlocks. The absence of deadlocks is one of the key design strategies in the design of automated manufacturing systems, as it guarantees that no process will be waiting for a particular event that will never occur [16]. A marked Petri net is said to be *bounded* if for each initial marking $\mathbf{x}_0$ there exists a positive integer $k = k(\mathbf{x}_0)$ such that $\mathbf{x}(p_i) \leq k$ for every place p_i and every possible marking obtained from $\mathbf{x}_0$. Boundedness can be viewed as a measure of stability of a discrete event system, as it will guarantee that the number of states of the system remains finite. If $k = 1$, then the Petri net is said to be *safe*. Formally, verifying that a marked Petri net is live and bounded can be done using a number of techniques that include the invariant, reachability graph and reduction methods, see for example [16] and the references therein.

At their inception, Petri nets were first used 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 [6, 22, 24, 11]. In practice, Petri nets exhibit both advantages and disadvantages over state automata for the modeling of discrete event systems. As discussed in [6], 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 [6] for an in-depth discussion):

- 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 nets are naturally oriented towards the modeling and analysis of asynchronous, discrete event systems with concurrency.

- 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.

Equipped with the basic tools for Discrete Event System modeling, it is now possible to formalize the approach to Mission Control System design proposed in the paper. This will be done by introducing successively the concepts of System Task, Vehicle Primitive, Mission Procedure and Mission Program, and explaining the software programming environments that were developed for their implementation.

3.2 System Tasks: Design and Implementation

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 to process motion sensor data into

a Navigation Task that will be responsible for determining the attitude and position of the vehicle in space. 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.*

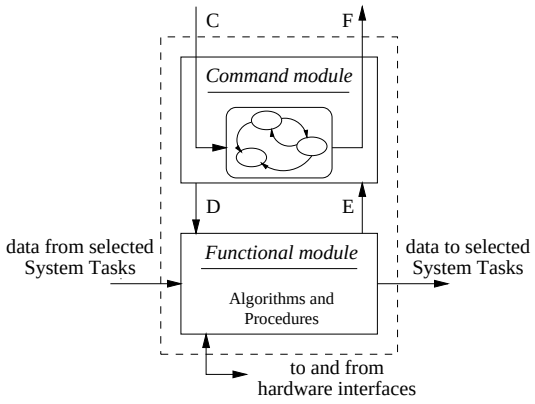

Figure 4: System Task structure.

The implementation of a System Task requires the interplay of the two modules depicted in Figure 4: 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. Formally, the state automaton that embodies the command module of System Task ST can be written as the five-tuple

$$([C, E], X, [D, F], f, g),$$

where C and E are finite alphabets of input commands and functional module messages respectively, D is a finite alphabet that configures algorithms, procedures, and data paths, F is a finite alphabet that acknowledges received commands and reports error occurrencies, X is a finite state set, $f : X \times [C, E] \rightarrow X$ is a state transition function, and $g : X \times [C, E] \rightarrow [D, F]$ is an output function.

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 sensor 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 alphabet F may include messages that report the unavailability of specific sensor data.

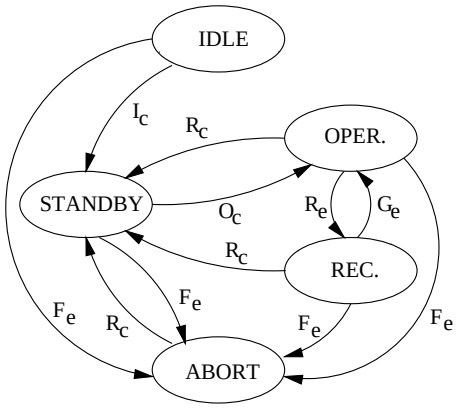

Figure 5: State diagram of a generic automaton.

The design of the functional module is carried out using well known tools from such diverse fields as navigation, guidance, and control, electrical measurements and instrumentation, communication theory, and computer science. See [12] for an example of the algorithms used for navigation of the MARIUS vehicle. The design of the command module amounts to specifying a finite state automaton that deals with the logical aspects of the System Task. Figure 5 represents the transition diagram of a possible finite state automaton, implemented in the case of the MARIUS AUV. In order to not clutter the diagram, the output messages are not included. Furthermore, only the transitions that connect two different states are represented. For the sake of clarity, the alphabet C of external commands and the alphabet E of module messages (including internal errors) have ben divided into the following classes:

$I_c \subset C$ - system initialization commands;

$O_c \subset C$ - processing algorithms or procedure selection commands;

$R_c \subset C$ - reinitialization commands;

$W_e \subset E$ - errors that do not affect the normal operation of the System Task;

$R_e \subset E$ - errors that affect the normal operation of the System Task, but from which the selected operation mode can recover internally (e.g. when no sensor data are available during a sampling interval);

$F_e \subset E$ - errors that affect the normal operation of the System Task, and from which the selected operation mode cannot recover (e.g. drastic sensor failure).

$G_e \subset E$ - messages issued at the end of internal recovery procedures.

The states of the automaton are the following:

IDLE - The System Task was started, but only the Command Module process is running. No data or algorithm initialization were performed.

STANDBY - An initialization or reset type command was executed. The System Task has all its resources available, and is ready to start.

OPERATIONAL - The System Task is running and accepts operational commands.

RECOVER - A recoverable error has occurred. The System Task is recovering locally from the error condition detected.

ABORT - A fatal error has occurred. The System Task requires a reset type command to recover from its current state.

3.3 Vehicle Primitives: Design and Implementation using CORAL

The concept of Vehicle Primitive plays a key role in the general framework for Mission Control System design adopted in this paper. A Vehicle Primitive corresponds to an atomic, clearly identifiable action performed by 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 message 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.

Using the Petri net formalism presented in Section 3.1, 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.

3.3.1 The CORAL Vehicle Primitive Editor and Generator. System Task Calls.

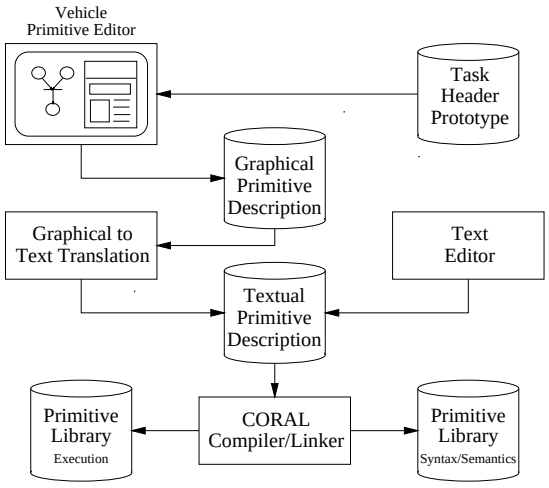

Figure 6: CORAL: A Vehicle Primitive programming environment.

Based on the framework introduced, a Vehicle Primitive programming environment named CORAL has been developed. Figure 6 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, LR1 synchronous language CORAL. A CORAL compiler/linker 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.

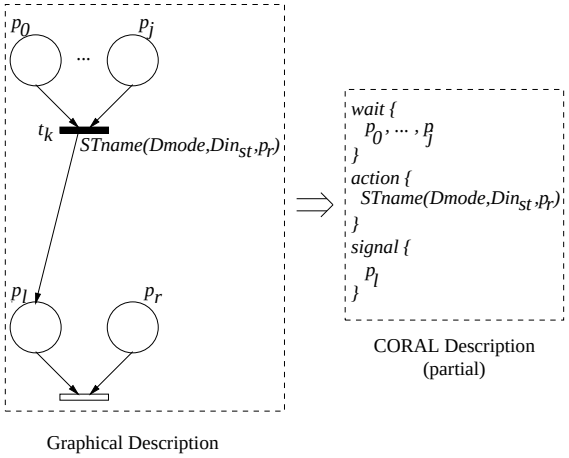

Figure 7: CORAL/ System Task interface.

The organization of CORAL can be explained in very simple terms by focusing on a example that illustrates the design of a Petri net corresponding to a given Vehicle Primitive. This is done with the help of Figure 7, which illustrates how the design of a subset of a generic Petri net is done, and how the equivalent CORAL language description is obtained. In order to understand the figure and the design methodology adopted, three basic concepts are required:

i) *System Task Calling Header. Binding of System Tasks and Vehicle Primitives.* - The firing of a generic transition 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, as part of the alphabet F. The marking of the Petri net can be formalized by introducing a binding function $\mathcal{F}_b$ as follows: given an alphabet F consisting of a finite number of messages $f_1, f_2, ..., f_n$, and given a finite set $P_m = (p_1, p_2, ..., p_n)$ of n places, $\mathcal{F}_b : F \rightarrow \mathbf{x}(P_m)$ is defined by $\mathcal{F}_b(f_i) = [0, 0, ...0, 1, 0, ...]$, where the element 1 occurs at the $i-th$ position.

ii) *Wait, Action* and *Signal* keywords - to describe a Petri net, the CORAL language uses three basic keywords: *wait*, *action*, and *signal*. The formal equivalence between the textual description of a Petri net using those keywords and its underlying Petri net graph, can be explained by referring to Figure 7, and examining the input and output sets of a particular transition t_k. The following equivalence relationships follow immediately:

$$\begin{aligned} I(t_k) &\Leftrightarrow wait\{p_0, ..., p_j\}, \\ O(t_k) &\Leftrightarrow action\{STname(Dmode, Din_{st}, p_r)\} \\ &\quad\ signal\{p_l\}. \end{aligned}$$

In this case, the System Task called has only one output message. Its occurrence will activate, through the proper binding function, the marking of place p_r. The extension to more complex Vehicle Primitives is immediate.

iii) *Global places* - The CORAL language allows to define global places that can be shared among the Vehicle Primitives.

At the present stage of development, the Vehicle Primitive Library Editor and Generator can be run on a PC/DOS or on a Unix Workstation.

3.3.2 The CORAL Engine

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. Figure 8 shows a schematic representation of the CORAL engine data structure and its communication mechanisms, as it processes a single Vehicle Primitive.

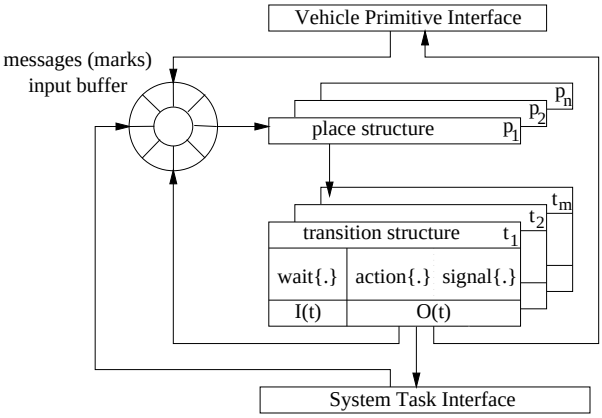

Figure 8: CORAL Engine schematic description.

The CORAL Engine accepts input messages corresponding to the markings of the Vehicle Primitive's Petri net being run, checks for the current set of enabled transitions, and issues output messages that

correspond to the new markings determined by the firing of those transitions. In practice, this is done by executing a CORAL engine synchronous loop described by the following sequence of actions: for each message in the input buffer,

(1) - update the number of marks in the corresponding place.

(2) - for the current state, check for the set of enabled transitions.

(3) - choose one transition t_k from the set of enabled transitions.

(4) - update the number of marks in the set of input places $I(t_k) \Leftrightarrow (wait\{.\})$.

(5) - issue messages in order to update the number of marks in the set of outputs places, $O(t_k) \Leftrightarrow (action\{...\}\ signal\{...\})$;

(6) - repeat (2) through (5) until the set of enabled transitions has been exhausted.

This cycle is repeated until the input buffer is empty.

It is important to remark that the CORAL Engine has a fixed structure, and the implementation of a new Vehicle Primitive simply requires that a new data set produced by the CORAL compiler be input to that Engine. This property makes the task of loading and unloading different Vehicle Primitives trivial. In particular, the loading of a Vehicle Primitive is an atomic operation performed by a CORAL module named Loader, and by the CORAL Engine itself. Each time a message requesting a certain Vehicle Primitive arrives in the Loader, the CORAL engine is stopped and the loading operation is performed as follows:

- The Loader performs a search over the *Vehicle Primitive Library* to get the desired Vehicle Primitive description;
- The CORAL engine transition set is actualized;
- The CORAL engine place set is actualized;
- The initial marking of the Vehicle Primitive is sent to the input buffer

After the loading operation has been performed, the CORAL engine may resume its normal operation mode.

3.4 Mission Programs and Mission Procedures: Design and Implementation

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 plan 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 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 embody the mechanisms for the coordinated 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.

3.5 Vehicle Primitive Calls. Binding of Mission Procedures and Vehicle Primitives.

Following the methodology adopted in the previous subsection, simple Mission Programs could in principle be embodied into - higher level - Petri nets that would implement the necessary Mission Procedure structures. This has in fact been done in the case of the MARIUS vehicle, as explained in Section 5.3. 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 5. *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 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.

At this point, it is important to understand how the successive binding between Mission Procedures, Vehicle Primitives and System Tasks is performed. As an illustrative example, suppose a Mission Procedure requires that all System Tasks be initialized at the beginning of its execution, and that the transition to an ABORT state at any time, in any System Task automaton, be known at the Mission Program level. This can be simply done by calling a specially designed Initialization Vehicle Primitive that will initialize all System Tasks and observe the occurrence of an ABORT in any System Task automaton. Assume the Primitive Petri net includes two places p_i and p_j that are marked when the Initialization phase has ended successfully, and when one of ABORT condition is detected, respectively. Then, a calling to that Primitive - at the beginning of the Mission Procedure - with an header where P_m consists of two places p_{InitOK} and p_{Abort}, will establish a binding between P_{InitOK} and p_1 and between p_{Abort} and p_2 that will be effective throughout the duration of the Mission Program. With the hierarchical binding of System Tasks to Vehicle Primitives and of Vehicle Primitives to Mission Procedures, p_{InitOK} and p_{Abort} will be marked when the Initialization phase has been performed successfully, and when at least one of the System Tasks entered an ABORT state. This information can be used for further processing at the Mission Procedure level.

3.6 The ATOL Programming Environment.

The analysis of even a simple mission plan programmed using the methodology explained above, will convince the reader that the complexity of the resulting Petri net structure can become unwieldy. See Section 5.3 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 a specific environment for Mission Program/Mission Procedure design and implementation, named ATOL, which is currently being developed and is briefly introduced in the sequel.

The ATOL software environment consists of the ATOL programming language, together with a set of development tools that allows for Mission Program *syntactic* and *semantic* analysis, and Mission Program execution. An ATOL program consists of a series of control flow and reactive statements used to schedule Actions specified by Mission Procedures and Vehicle Primitives.

The ATOL language belongs to the class of imperative, reactive synchronous programming languages [9], which are dedicated to writing programs that continuously react to events coming from the environment. ATOL is based on the assumption that the environment does not interfere with the application during the reactions. Furthermore, a reaction to an external event will start from a set of control points and finish by reaching another set of control points. Reactive statements are provided in order to control the Mission Program execution.

An important constraint was taken into consideration during the design of the ATOL Language: *an ATOL statement should always admit a logically equivalent Petri net model.*

Due to space limitations, an exhaustive definition of the ATOL language and the ATOL programming environment are not provided here. The reader is referred to the next section for a diagram illustrating the functionalities of ATOL.

3.7 Mission Control System Organization

The framework for Mission Control System design and implementation proposed in this paper leads to the general structure of Figure 9, 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, an on-line checking mechanism for the syntax and semantics of ATOL statements, and formal Mission Program verification tools.

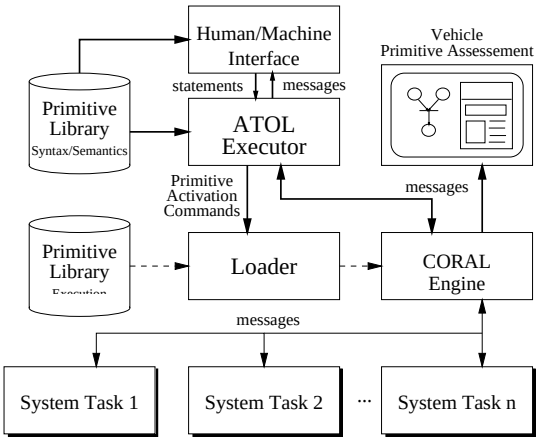

Figure 9: Mission Control System Organization.

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, the status of any Vehicle Primitive can be displayed on a *Vehicle Primitive Assessment module* that allows visualizing the flow of markings on the corresponding Petri nets. See Section 5.3 for more details.

4 Formal Mission Verification

The operation of unmanned underwater vehicles in hazardous environments poses great demands on their robustness against external disturbances and internal hardware failures, as well as on adequate, repeatable vehicle behaviour, in response to operator commands and vehicle system errors. Those requirements dictate a solid approach to system development that includes the use of rigorous analysis tools to assess the expected behaviour of the vehicle before it is launched. In practice, that poses the formidable task of formally verifying all vehicle systems, including that in charge of Mission Control. However, in spite of the plethora of theoretical methods available for the analysis of time-driven and event-driven systems, little progress has been done towards the development of powerful computer based tools to assess the performance of Mission Control systems, even at a purely logical level. Notable exceptions include the pioneering work pursued at INRIA in France, with applications to mobile robots. See [9] and the references therein. The need for formal logical verification tools can hardly be overemphasized: the design of even a simple mission using the software programming environment proposed in Section 3 may generate an hierarchical structure of Mission Procedures, Vehicle Primitives and System Tasks that is hard to analyze by direct inspection of their text/graphical descriptions. As a result, logical errors are bound to occur during the design phase that may lead to serious deadlock problems, or to the vehicle behaving in a totally unacceptable manner in response to internal or external events.

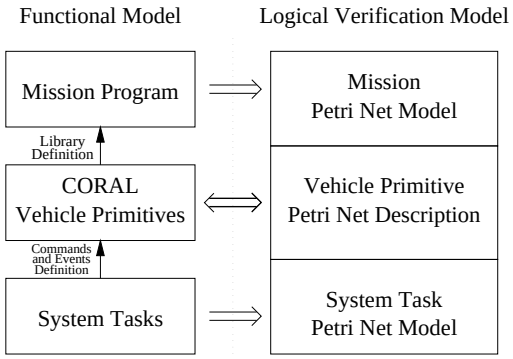

Figure 10: Logical verification model.

Motivated by the work described in [9], this section discusses very briefly how Petri net theory can be brought to bear on the formal logical verification of Mission Control Systems designed according to the formalism proposed in this paper. The key idea can be simply explained by referring to Figure 10, which shows how the System Task Automata, Vehicle Primitives, and Mission Procedures of a given mission can be converted to a set of merged Petri net models. These can be formally verified at any hierarchical level, using classical Petri net analysis tools.

System Task Verification Model In Section 3, a System Task was shown to contain a Command module embodied in a state automaton that activates and/or synchronizes the System Task's operation modes, in response to commands sent by a calling Vehicle Primitive. Using well known results from Discrete Event System theory, the verification model of a System Task's state automaton can be equivalently described in terms of a Petri net

$$(P_{ST}, T_{ST}, A_{ST}, w_{ST}, \mathbf{x}_0), \tag{1}$$

where P_{ST}, T_{ST}, A_{ST}, and w_{ST} are easily obtained from the state set, input and output alphabets, state transition function, and output function of the automaton, and $\mathbf{x}_0$ is derived from its initial state [6].

Vehicle Primitive Verification Model The formal verification model of a Vehicle Primitive is obtained by merging the Vehicle Primitive's Petri net model with the Petri nets corresponding to the System Tasks called. The merging of Petri nets can be done using the binding process described in general terms in Section 3.3.1. Furthermore, by exploit-

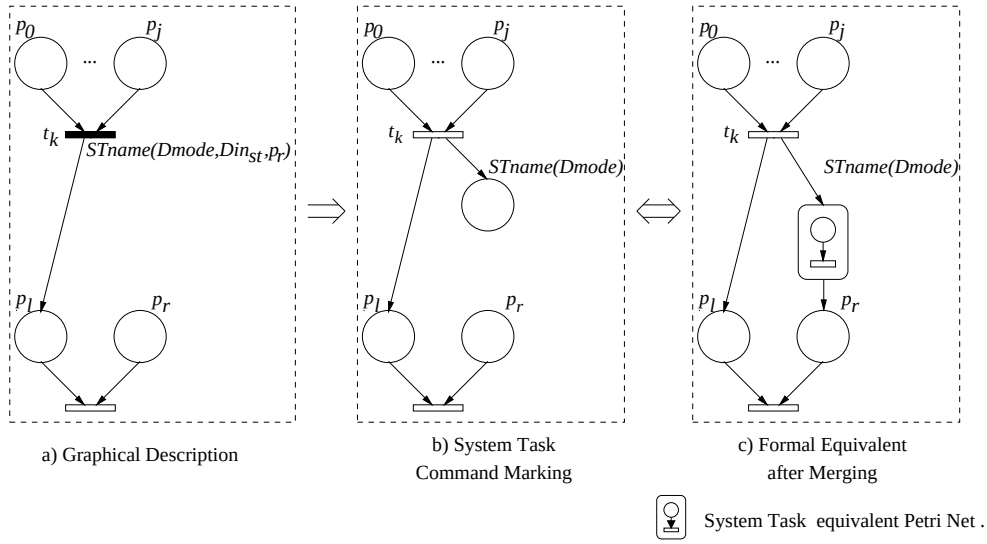

Figure 11: *Vehicle Primitive* formal verification model.

ing the Petri net representation of System Task automata, the sequence of operations depicted in Figure 11 will lead to the general verification model of Figure 11.c, which should be compared against Figure 7.

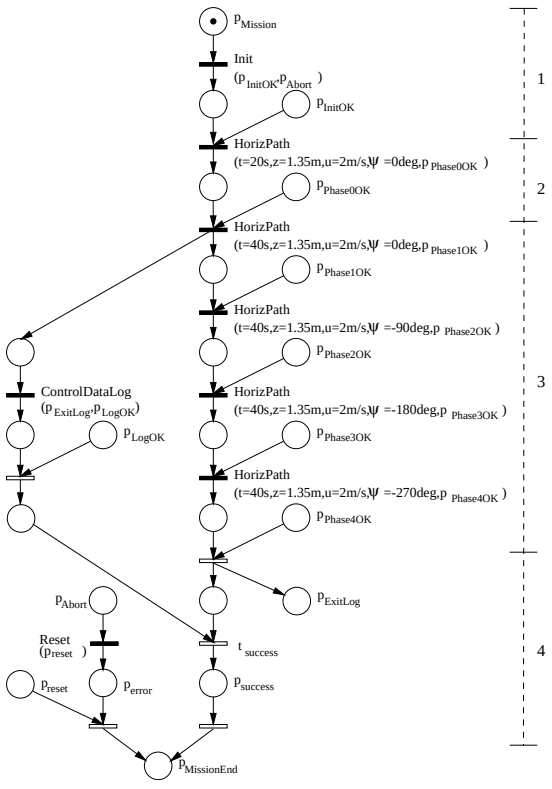

Figure 12: *Mission* structure for formal verification.

Mission Verification Model The formal verification model of a Mission is obtained by merging the mission's equivalent Petri net model with the equivalent Petri net models of the Mission Procedures and Vehicle Primitives called. This can be done by using a binding process that is similar to that described in Section 3.5, leading to the structure in Figure 12. Notice how, by proper design, the complete Petri net model was made to include the following relevant places: p_{init} - the entry point to the mission; { p_{error1}, ..., p_{errorn} } - a set of places that reflect possible mission errors; $p_{success}$ - a place that is marked in case the mission is completed successfully, and p_{end} - a place that is marked if the mission is completed. Assume that by construction there is a set of places, denoted P_E, that hold information related to System Task and Vehicle Primitive errors. Then, answers to the representative set of questions

1. Is there a set of conditions under which the mission can be completed?

2. Is there a set of conditions under which the mission can be completed successfully?

3. In the absence of errors, will the mission be completed successfully?

4. In the presence of errors, will the mission be completed?

can in principle be obtained by examining the properties of reachability and liveliness of the global Petri net [6]. In particular, the answer to the third question will be in the affirmative if it can be proved that transition $t_{success}$ is *potentially fireable* [19] with respect to a conveniently defined initial marking vector such that $x(p_{init}) = 1$ and $\mathbf{x}(P_E) = \mathbf{0}$, where $\mathbf{0}$ denotes the zero vector. The reader will find in [8, 9] an illuminating discussion of this circle of ideas, together with a description of some computer based tools for formal verification of missions that build on the analysis of state automata.

5 Mission Control of the MARIUS AUV: System Design and Implementation

This section aims at bridging the gap between the theoretical framework of Section 3 and the practice 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 6, when describing the results of the tests at sea.

5.1 System Tasks

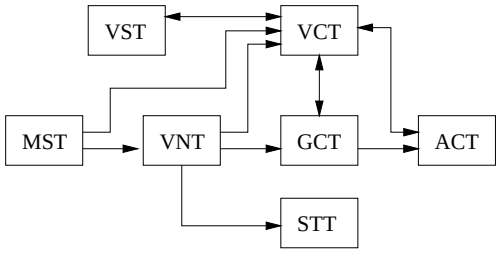

Figure 13: Vehicle System Tasks: data communication paths.

The System Tasks of MARIUS and their data interconnection structure are depicted in Figure 13, which should be compared against the diagram of Figure 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 Figure 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 it will become clear from the context. However, the following System Tasks deserve special consideration:

Motion Sensor Task (MST) - The Motion Sensors 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 3.3, 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 $p_{yawauto}$ denotes a place in the calling Petri net that will be marked when a specific message in the alphabet F of 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.

The Task structure of Figure 13 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 a shared memory communication mechanism.

5.2 Vehicle Primitives

A selected set of Vehicle Primitives included in the Mission Control System of MARIUS is described in the sequel. Due to space limitations, 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 Section 3.3 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 in any of the System Tasks, respectively (see the explanation in Section 3.5)

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 KeepHeading 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.

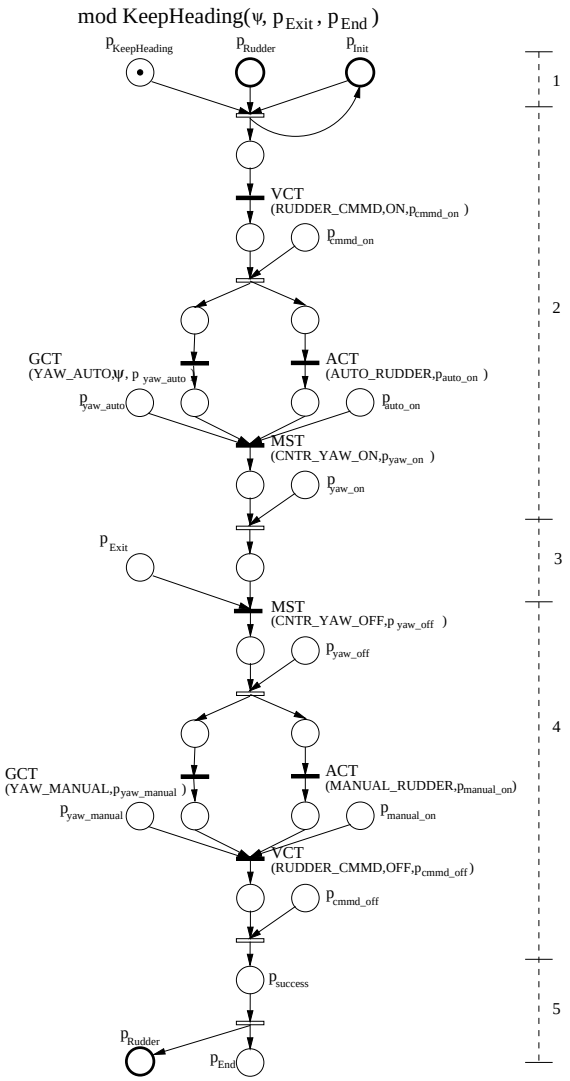

Figure 14: KeepHeading Vehicle Primitive - p_{Rudder} and p_{Init} are global places.

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 Figure 14.

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 3.3, $P_{pre} = p_{Init}$, 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.

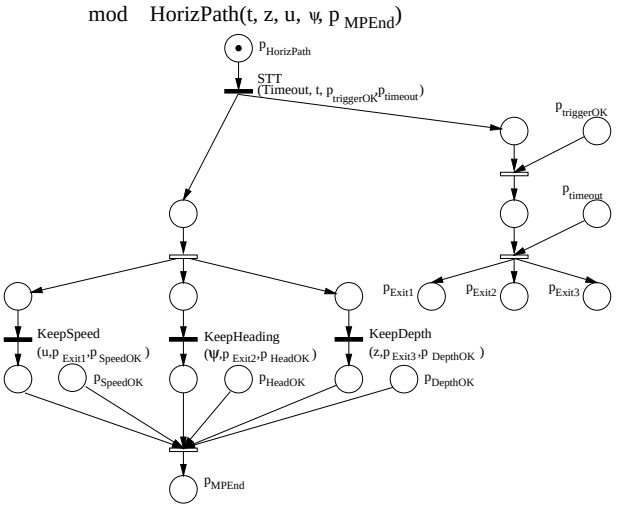

Figure 15: Mission Procedure described in CORAL.

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.

5.3 Mission Design 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 Figure 15. 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 an *STT* timeout 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 Figure 16, 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 $t = 40$ s. In parallel, the Vehicle Primitive *ControlDataLog* is called to start logging control loop data for later off-line analysis.

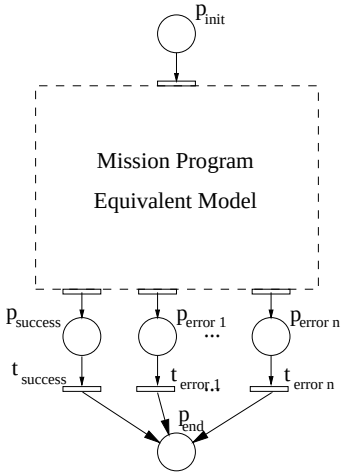

Figure 16: Mission program described in CORAL.

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.

6 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 5.3. 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 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 of 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 Figure 17. The PC dedicated to mission control follow-up enabled the display of the mission Petri net network, together with the respective marking sequences. Figures 18 through 21 display some of the data acquired in the course of the mission, which was executed to perfection. Figures 18 and 19 show the commanded and measured heading, and the rudder activity, respectively. Figures 20 and 21 show the slight variations in heading and depth caused by the wave action in shallow water.

References

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

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

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

[4] C. Bizingre, P. Oliveira, A. Pascoal, F. Pereira, J. Pignon, E. Silva, C. Silvestre, J. Sousa, "Design of a Mission Management System for the Autonomous Underwater Vehicle MARIUS," *Proceedings of the Symposium on Autonomous Underwater Vehicle Technology*, Cambridge, MA, July 1994.

[5] R. Byrnes, S. Kwak, R. McGhee, A. Healey, M. Nelson, "Rational Behaviour Model: An Implemented Tri-Level Multilingual Software Architecture for Control of Autonomous Vehicles," *Proc. 8th International Symposium on Unmanned Untethered Submersible Technology*, Durham, New Hampshire, September1992, pp. 160–179.

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

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

[8] B. Espiau, D. Simon, K. Kapellos, "Formal Verification of Missions and Tasks," *Proc. US/Portugal Workshop on Undersea Robotics and Intelligent Control*, Lisbon, Portugal, March 1995, pp. 73–77.

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

[10] G. Franklin, J. Powell, M. Workman, *Digital Control of Dynamic Systems*, Addison-Wesley, 1990.

[11] P. Freedman, "Time, Petri Nets, and Robotics,"*IEEE Transactions on Robotics and Automation*, Vol. 27, No. 4, Aug. 1991.

[12] D. Fryxell, P. Oliveira, A. Pascoal, C. Silvestre e I. Kaminer, Navigation, Guidance and Control Systems of AUVs: An Application to the MARIUS Vehicle, To appear in *IFAC Control Engineering Practice*, March, 1996,

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

[14] A. Healey,"Tactical/Execution Level Coordination for Hover Control of the NPS AUV I Using Onboard Sonar Servoing," *Proceedings of IEEE Symposium on Autonomous Underwater Vehicle Technology* , Cambridge, Massachusetts, pp. 129-138, 1994.

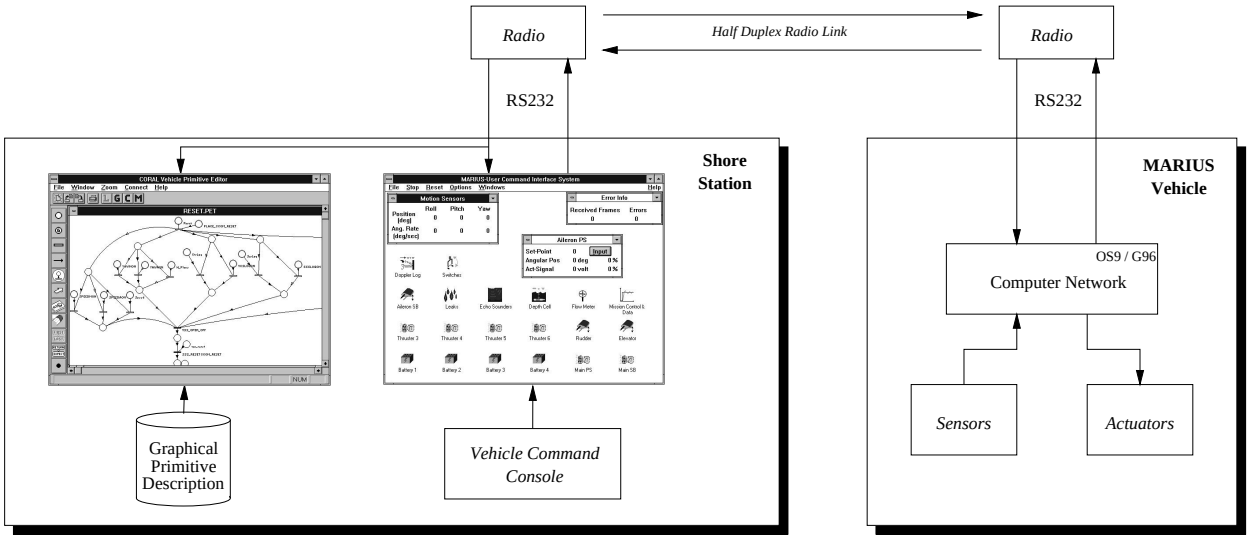

Figure 17: MUCIS - MARIUS user interface unit.

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

[16] M. Jeng, F. DiCesare, 'A Review of Synthesis Techniques for Petri Nets with Applications to Automated Manufacturing Systems," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 23, No. 1, Jan/Feb. 1993, pp.301–312.

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

[18] D. Marco, A. Healey, R. McGhee, "Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion," to appear in the *Journal of Robotics*, 1996.

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

[20] A. Pascoal, "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, May 1994.

[21] A. Pascoal, F. L-Pereira, A. Bjerrum, K. Christiansen, J. P-Pignon, G. Ayela, C. Petzelt, "Development of a Self-Organizing Underwater Vehicle," *Proc. MAST Days and EUROMAR Market*, Brussels, March 1993.

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

[23] G. Saridis, "Towards the Realization of Intelligent Controls," *IEEE Proceedings*, Vol. 67,No.8, 1979.

[24] G. Saridis, "Analytical Formulation of the Principle of Increasing Precision with Decreasing Intelligence for Intelligent Machines," *Automatica*, Vol. 25, pp. 461–467.

[25] V. Silva, P. Oliveira, C. Silvestre, A. Pascoal , " CORAL: A Software Package for the Design and Implementation of Real Time Mission Control Systems for Autonomous Underwater Vehicles, *MAST-SOUV report*, October 1994.

[26] D. Simon, B. Espiau, E. Castillo, K. Kapellos, "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, Dec. 1993, pp. 213–229.

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

[28] F. Wang, K. Kyriakopoulos, A. Tsolkas, G. Saridis, "A Petri-Net Coordination Model for an Intelligent Mobile Robot," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 4, July/August 1991,pp. 777–789.

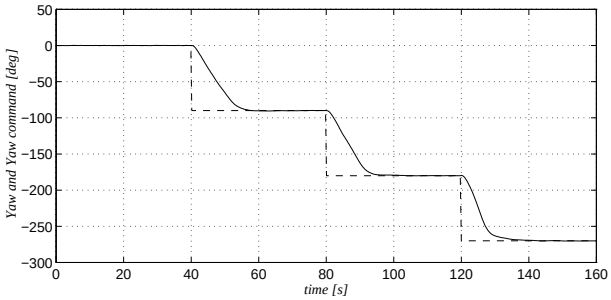

Figure 18: Commanded and measured heading.

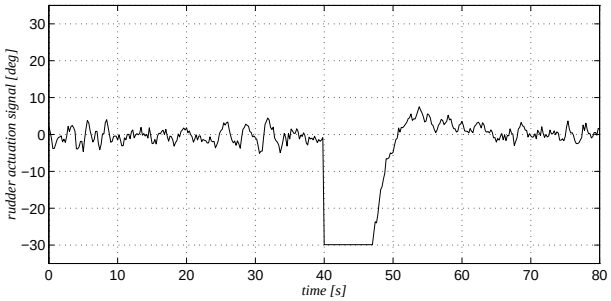

Figure 19: Rudder deflection.

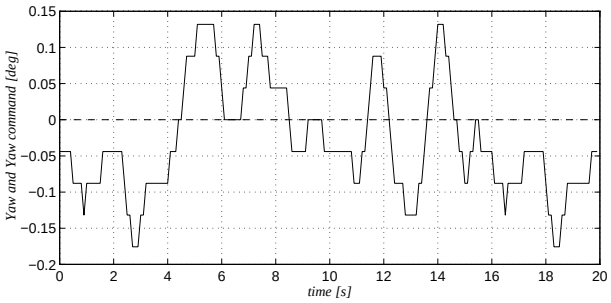

Figure 20: Measured heading (zoom in).

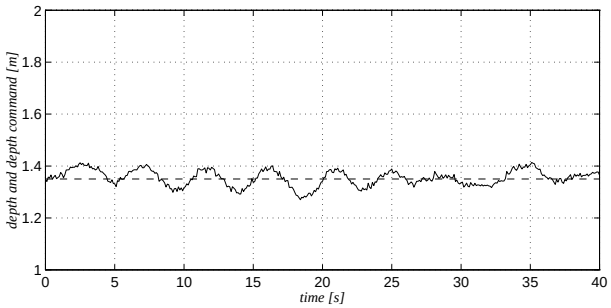

Figure 21: Commanded and measured depth.