Strategic Level Mission Control - An Evaluation of CORAL and PROLOG Implementations for Mission Control Specifications *

A. J. Healey † , D. B. Marco † , P. Oliveira ‡ , A. Pascoal ‡ , V. Silva ‡ , C. Silvéstre ‡
† Autonomous Underwater Vehicles Laboratory
Department of Mechanical Engineering
Naval Postgraduate School, Monterey, CA. 93943

‡ Institute for Systems and Robotics
Instituto Superior Técnico
Av. Rovisco Pais, 1096 Lisboa Codex, Portugal

Abstract

This paper presents the use of the software programming environments PROLOG and CORAL for the implementation of the Strategic Level of the NPS Phoenix vehicle. Whereas PROLOG provides a rule based mission control specification language, CORAL builds on a graphical interface to describe mission programs using Petri nets. The paper describes the interfacing of CORAL with the Tactical level of the vehicle, and details the programming and execution of a vehicle mission that was run in the NPS test tank.

1 Introduction

This paper describes ongoing work between the Naval Postgradute School (NPS) and the Instituto Superior Técnico (IST) of Lisbon, as part of the joint US/Portuguese activity on the subject of autonomous/semiautonomous underwater vehicle mission control. The NPS has designed and operates the research testbed Phoenix vehicle. The IST has led the team that designed and built the MARIUS vehicle, which has undergone extensive tests at sea. Both vehicles have similar shapes and Execution level controllers implemented in a GESPAC M68030 based architecture running the OS-9 real time operating system. Different approaches, however, have been taken in the design of the higher level mission controllers for the vehicles.

A Phoenix mission program is specified by a set of PROLOG rules that reflect Mission and Vehicle doctrines, and are executed as specified. The PROLOG inference engine cycles through the predicate rules to manage the discrete event logical aspects of mission related decisions. It transitions states, and generates commands to the Tactical level of the vehicle, which is in charge of implementing basic 'vehicle primitives' [8].

Mission control of MARIUS can be achieved by specifying a mission program that is embodied in a Petri net structure. A software programming environment named CORAL allows for graphically constructing the required Petri nets, and executing them on a CORAL software Engine that is implemented on the vehicle's computer network [11].

In the summer of 1995, an experiment was conducted to evaluate the difficulties and fundamental differences in performing mission control using the two different approaches. A common mission was agreed on, and the NPS Phoenix vehicle was used as an experimental platform for evaluation of both mission control concepts. In the experiment, the CORAL programming environment was used to implement the Strategic level of the vehicle, thus effectively replacing the existing PROLOG implementation. No Execution or Tactical level software needed to be changed. Only the interfacing functions between the Phoenix Tactical level and the CORAL calls to 'vehicle primitives' had to be brought into line.

The paper describes the CORAL software environment for mission programming, and the interfacing that was necessary to link it with the Phoenix Tactical level software. Since the Phoenix control system runs with two processors (SUN and GESPAC) and two different operating systems (Unix and OS-9), the first issue was to ensure that the CORAL code - generated through the graphical development of a Petri net specification - would correctly open socket communications between the two processors and initialize the Phoenix vehicle correctly. Later issues

*The work of IST was supported by the Commission of the European Communities under contract MAS2-CT92-0021 of the MAST-II programme, by the Portuguese PRAXIS programme under contract 3/3.1/TPR/23/94, and by two NATO Scholarships during the summer of 1995.

described in the paper dealt with the correct sequencing of the 'vehicle primitive' control functions: the starting and stopping of filters, the issuing of control function set points and, finally, the orderly shutdown of the control networks. Mission specification Petri nets for a mission example will be described and compared against the equivalent PROLOG specification.

The organization of the paper is as follows. Section 2 describes the NPS Phoenix vehicle and its Mission Control System, and reviews the use of PROLOG as a tool for mission control specification. Section 3 introduces CORAL as an alternative framework for Mission Control specification using Petri net theory, and describes the interfacing of CORAL with the Tactical level of the vehicle. Finally, Section 4 details the programming and execution of a simple tank mission using CORAL and PROLOG. The paper ends with the conclusions in Section 5.

2 The NPS Phoenix Vehicle. Mission Control Specification using PROLOG

For several years, the Naval Postgraduate School (NPS) has been engaged in research and development of advanced control technology for unmanned underwater vehicles. As part of that development effort, the NPS has built the research testbed vehicle named NPS Phoenix. The vehicle is equipped with eight plane surfaces and two propulsion motors for flight control. Two vertical thrusters provide for heave and pitch control, and two transverse thrusters for heading and lateral movement control. A free flooded fiber glass dome supports two forward-looking sonar transducers, a downward-looking sonar altimeter, a water speed flow meter, and a depth pressure cell. Motion sensors mounted internally are used to measure angles and rates for roll, pitch and yaw, respectively. The vehicle has a length of 2.13 meters and a dry weight of 175 kg. Sufficient energy storage (1100 Wh) is provided by 4 lead acid gel batteries for approximately 3 hours of operational testing.

2.1 Mission Control System Organization

With the objective of building an ever increasing level of automatic capability into the vehicle, a tri-level software control architecture comprising Strategic, Tactical, and Execution levels has been developed. The architecture provides for mission control capabilities, and eases the reconfiguration of control software code as missions become more complex or vehicle capabilities change. The three levels separate the software into easily modularized units encompassing a wide range of functions from intense discrete state transitions to the interfacing of *asynchronous* data updates with the real time *synchronized* controllers that stabilize the vehicle motion in response to commands.

The Strategic level uses PROLOG as a rule based mission control specification language. Its inference engine cycles through the predicate rules to manage the discrete event logical aspects of mission related decisions. It transitions states, and generates the commands that drive the vehicle through its mission. Error recovery procedures from failures in the mission tasks or the vehicle subsystems are included as transitions to 'error' states that ultimately provide commands to the servo level control for appropriate recovery action.

The Tactical level, currently written in C, is a set of functions that interface with the PROLOG predicates, returning TRUE/FALSE in response to commands and queries. The Tactical level is also interfaced to the real time Execution level controller through asynchronous communications using script type message passing through a non-blocking socket.

The Execution level commands the vehicle subsystems to activate 'behaviours' that correspond to those commanded. Communication from the Tactical level to the Execution level takes place through a single socket. By the design of this hierarchical control system, the Tactical level runs asynchronously and retains the mission data file and the mission log file in global memory. It sends the command scripts to the Execution level, and requests data for the evaluation of state transitions.

2.2 Mission Control Implementation

The Mission Control System of the Phoenix vehicle, illustrated in Figure 1, is currently implemented in hardware using three networked processors. All Execution level software is written in C and runs on a GESPAC MC68030 processor in a separate card cage inside the boat. Connected in the same card cage is an ethernet card and an array of real time interfacing devices for communications to sensors and actuators. The execution level code containing a set of functions in a compiled module is downloaded first and run to activate any mission. It starts the communications socket on the GESPAC side, and waits for the higher level controller to start.

The Strategic level PROLOG rules are compiled and linked together with the supporting Tactical level C language functions into a single executable process called 'mission control', that is run in a SUN Sparc 4 laptop computer and linked through ethernet and a non-blocking socket to the GESPAC processor. Upon starting, it first opens the SUN side of the communications socket, initiating the ethernet link between both SUN and GESPAC processors, then sending sequenced control commands to the vehicle. All vehicle control functions, with the excep-

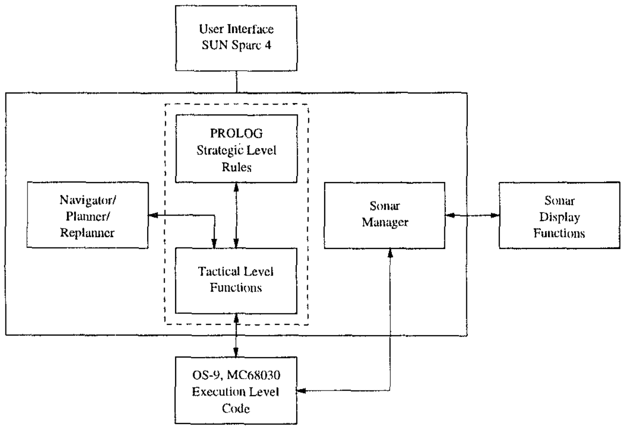

User Interface SUN Sparc 4
PROLOG Strategic Level Rules
Navigator/ Planner/ Replanner
Sonar Manager
Sonar Display Functions
Tactical Level Functions
OS-9, MC68030 Execution Level Code

Figure 1: Outline of the Phoenix Networked Controller.

tion of the transmission of sonar imaging data, communicate using a message passing mechanism through that socket.

A second SUN process called the 'Sonar Manager' is opened which runs asynchronously in the SUN and with equal priority to the 'Mission Control'. This process is linked through a separate socket to the GESPAC for the purpose of the reception and handling of sonar imaging data. This process is activated if and when sonar is activated by the the Strategic level rules. The 'Sonar Manager' captures data that is sent out from the Execution level as soon as it has been acquired, and then processes and passes the data to be displayed on an IRIS Graphics workstation for visualization purposes.

3 Mission Control using CORAL

The Strategic Level of the Phoenix vehicle generates messages that trigger the execution of a number of Tactical level functions. The conditions that determine the occurrence of those events are dictated by the logical structure of the mission being performed - as embodied in a set of PROLOG rules - and by the types of messages received from the Tactical Level. Clearly, this motivates an alternative approach to the implementation of the Strategic Level using Petri nets, which are naturally oriented towards the modeling of asynchronous, discrete event systems with concurrency. This approach has been pursued at IST in the course of developing a Mission Control System for the MARIUS AUV [1], leading to a software environment named CORAL for the design and implementation of Petri net structures [11].

This section describes the use of CORAL as an alternative software environment for the programming of underwater vehicle missions. For the sake of completeness, the section starts with a review of the necessary background material on Petri Net theory. The nomenclature and the style of the presentation have been strongly influenced by the material in the textbooks of Cassandras [3] and Peterson [12], which contain excellent introductions to the subject.

3.1 Petri Net Theory

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 $w : A \rightarrow \mathcal{Z}_+$ is a weight function that assigns a positive integer (weight) to an arc.

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* $\mu : P \rightarrow \mathcal{N}^n$, which maps the set of places P to the n-tuple $\mathbf{x} = [x(p_1), x(p_2), ..., x(p_n)]'$ of nonnegative integers, where $x(p_i)$ denotes the number of tokens in place p_i. A Petri net is formally defined as a five-tuple

$$(P, T, A, w, \mathbf{x}_0),$$

where $\mathbf{x}_0$) is the initial Petri net marking. 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 1. 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 p_i to t_j, then $p_i \in I(t_j)$. Similarly, an arc directed from t_j to p_i means that $p_i \in O(t_j)$. When using Petri net graphs, a token assigned to place p_i is indicated by a dark dot positioned in that place.

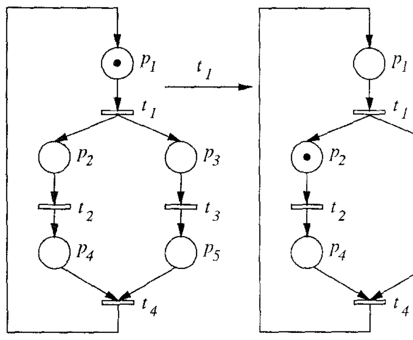


Figure 2: Example of a Petri net

Since the execution of a Petri net is controlled by the number and distribution of tokens in the net, it is natural to identify the *state* X of a Petri net with its marking $\mathbf{x}$. It follows from the above definitions that the state space $\mathbf{x}$ 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* $\phi : X \times T \rightarrow X$ 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.

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 [3, 12, 9] and the references therein. This issue has been addressed in [11], in the context of formal mission verification [5, 6].

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 [3, 12, 14, 9].

3.2 The CORAL Development Environment. Implementation issues.

This section introduces CORAL as a software environment for the design and implementation of Petri net structures, and explains its interfacing to the Tactical level of the NPS Phoenix vehicle. The development of CORAL for the MARIUS AUV is documented in [11]. The reader will easily recognize the modifications that were needed to interface it to the Tactical level software.

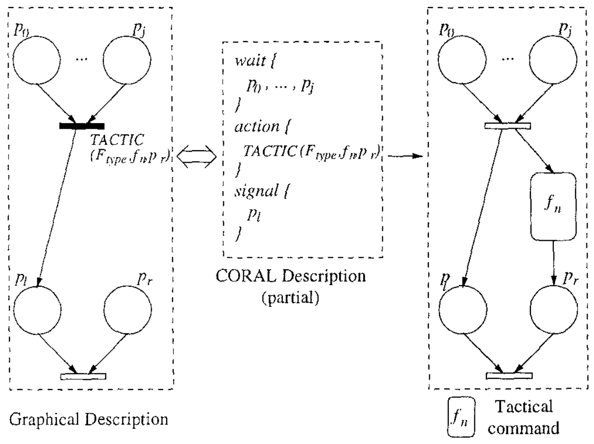


Figure 3: CORAL/TACTIC level Interface.

The organization of CORAL can be explained in very simple terms with the help of Figure 3, 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, two basic concepts are required:

i) *Tactical Level Calling Header.* - The firing of a generic transition will start the execution of a Tactical Level command, which is evoked through an header with the structure

$$TACTIC(F_{type}, f_n, P_m)$$

where $TACTIC$ specifies a Tactical Level function interface that will be detailed later, F_{type} identifies the type of function or particular algorithm to be executed, and f_n is the specific name of the function to be called. The last calling parameter set P_m indicates a finite set of places in the Petri Net that will be marked depending on the type of message received from the tactical function.

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 3, and examining the input and output sets of a particular transition t_k. The following equivalence relationships follow immediately:

$$\begin{aligned} I(t_k) &\Leftrightarrow wait\{p_0, ..., p_j\}, \\ O(t_k) &\Leftrightarrow action\{TACTIC(F_{type}, f_n, p_r)\} \\ & \quad\; signal\{p_l\}. \end{aligned}$$

In this case, the tactical level function called has only one output message, and its occurrence will activate the marking of place p_r. The extension to more complex Petri Net structures is obvious.

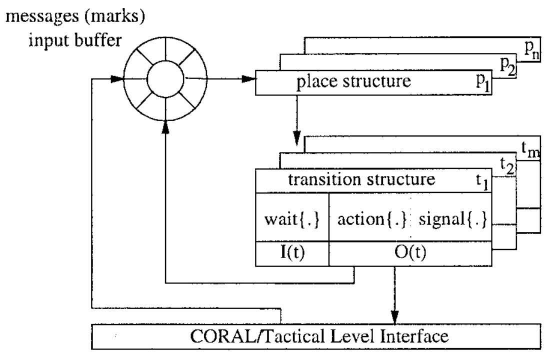


Figure 4: CORAL Implementation Structure.

A CORAL Engine has been developed that accepts Petri net descriptions and executes them in real-time. Figure 4 shows a schematic representation of the CORAL Engine data structure and the communication mechanisms that implement a Petri Net. The CORAL Engine accepts input messages corresponding to the markings of the 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 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.

The CORAL Engine is complemented with a set of software design tools that allow editing and generating a Mission Library containing the description of the Petri nets for each phase of a given mission, see Figure 5. The Mission

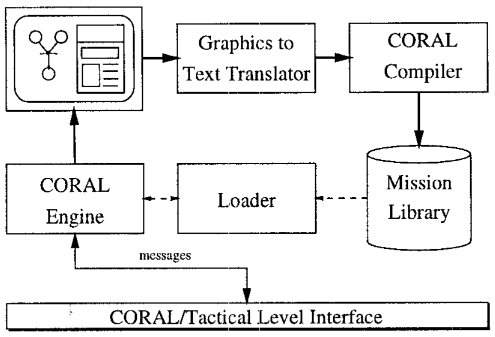
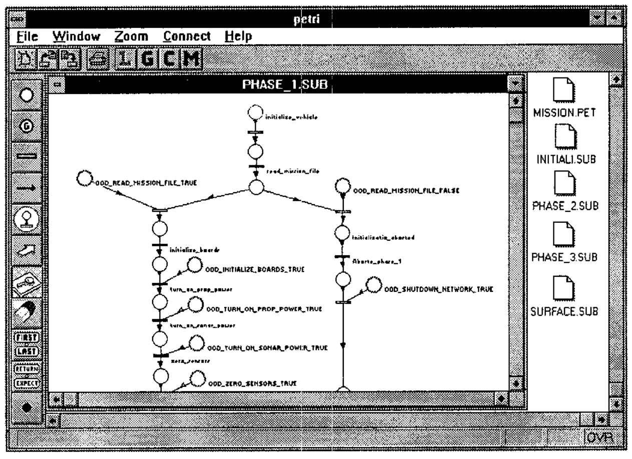


Figure 5: CORAL Development Environment

Library can be constructed by using a Graphical Editor to describe the structure of each mission phase, and generating the corresponding textual CORAL language descriptions. Mission execution is achieved by loading successively each mission phase from the *Mission Library* into the CORAL Engine, and running it [11]. Figure 6 shows the graphical interface for Mission Library editing.



petri
File Window Zoom Connect Help
PHASE_1.SUB
MISSION.PET
INITIALI.SUB
PHASE_2.SUB
PHASE_3.SUB
SURFACE.SUB
OOD_READ_MISSION_FILE_TRUE
OOD_READ_MISSION_FILE_FALSE
OOD_INITIALIZE_BOARDS_TRUE
OOD_SHUTDOWN_NETWORK_TRUE
OOD_TURN_ON_PROP_POWER_TRUE
OOD_TURN_ON_SONAR_POWER_TRUE
OOD_ZERO_SENSORS_TRUE
OVR

Figure 6: CORAL Graphical Editor

To complete the software tools that are required to implement the Strategic level, an interface between the CORAL Engine and the Tactical level was specified that reflects the constraints imposed by the existing software architecture. The following classes of functions were identified:

TACTIC(EXEC, function, place) - starts an Execution level function. A place will be marked after the function has been executed.

TACTIC(ASK, predicate, place_1, place_2) - asks for the logical value of a predicate. Place_1 (resp. place_2) will be marked if the predicate is true (resp. false).

TACTIC(REPORT, string) - the Strategic level reports a string (message) to the operator console.

TACTIC(OOD, function, place) - The OOD is commanded to execute a function. A place will be marked after the function is executed.

4 A Mission Example

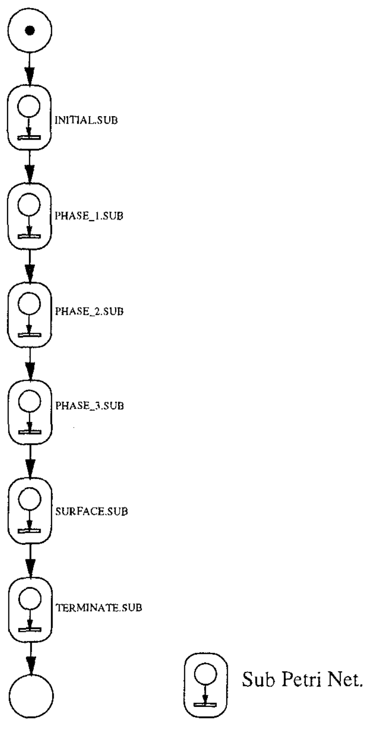


Figure 7: Mission Program using CORAL

This section illustrates the use of PROLOG and CORAL to run a simple mission with the NPS Phoenix vehicle. As shown in Figure 8, the difference between the two approaches lies on the implementation of the Strategic level of the vehicle.

The mission selected has an initialization phase, a phase where the vehicle submerges below the water surface to get away from surface suction effects, a submerge to depth phase, and a return to the surface. The mission is simple, yet it serves the purpose of illustrating that the software environments for Strategic level implementation may be changed as long as the 'vehicle primitive' calls are common.

Figure 7 depicts the global Petri net for the mission, with each mission phase being implemented as a sub Petri net. Due to lack of space, only the sub Petri net of phase

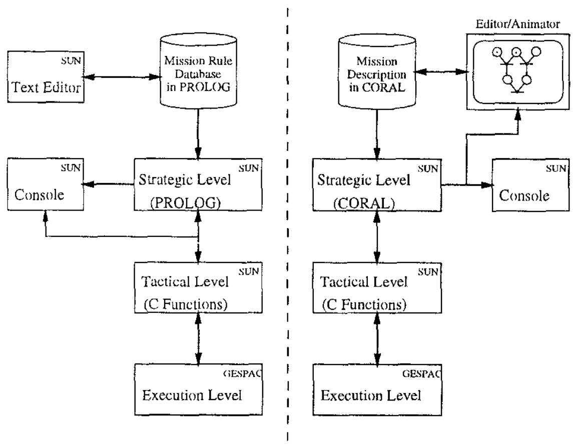


Figure 8: Strategic Level Implementation using PROLOG and CORAL.

two will be described here, which should be compared against the original PROLOG rule set of Figure 10. The initial stage of this phase consists of reading the next set-point for depth from the mission data file, starting the depth filter, and sending the vehicle command to use vertical thrusters to submerge to the depth set point beginning the submerge maneuver . While submerging, a timer is started with a pre-defined timeout in order to limit the duration of the maneuver. During the maneuver, the Tactical level is continuously asked to check if the commanded depth has been reached, the pre-defined timeout has expired, and if a system problem has occurred. In case the answer to the first question is affirmative, this phase of the mission is successfully completed and the mission can proceed. Otherwise, the emergency surface maneuver is activated and the mission is aborted.

The mission described was programmed using the CORAL Graphical interface described in section, and ran in the SUN Workstation devoted to Strategic level implementation. During mission execution, the Graphical interface allowed for the display of the state of the Petri nets being run, by showing the evolution of their marking sequences on the screen.

5 Conclusions

This paper described an experiment whereby the CORAL programming environment developed by IST was interfaced to the Tactical level of the NPS Phoenix vehicle. No Execution or Tactical level software needed to be changed. Only the interfacing functions between the Phoenix Tactical level and the CORAL calls to 'vehicle primitives' had to be designed. The relative simplicity with which an example mission was jointly programmed and run paves the way for future joint activities, and demonstrates that true inter-group cooperation on the subject of underwater vehicle mission control is within reach.

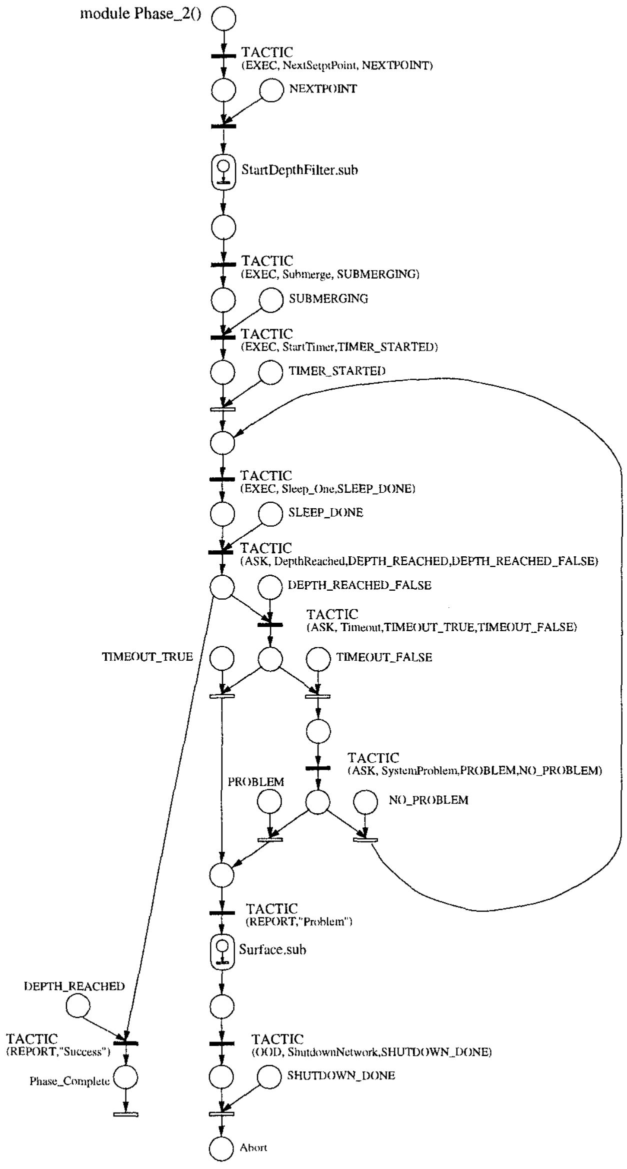


Figure 9: Mission phase 2 in CORAL

```
done:- current_phase(mission_abort),
 ood('shutdown_network',X).

% Dive to starting depth using thrusters
execute_phase(2):- exec_next_setpt_data(X),
 start_depth_filter, exec_submerge(X),X==1,
 exec_start_timer(X), repeat, phase_completed(2).

start_depth_filter:- ask_depth_filter_off(X),
 X==1, exec_start_depth_filter(X),
 exec_sleep(1,X).

start_depth_filter.

phase_completed(2):- exec_sleep(1,X),
 ask_depth_reached(X), X==1, asserta(complete(2)).

phase_completed(2):- ask_time_out(X),X==1,
 exec_surface(X), repeat, ask_surface_reached(X),
 X==1, asserta(abort(2)).

phase_completed(2):- ask_system_problem(X), X==1,
 exec_surface(X), repeat, ask_surface_reached(X),
 X==1, asserta(abort(2)).

next_phase(2):- complete(2),
 retract(current_phase(2)),
 asserta(current_phase(3)).

next_phase(2):- abort(2),
 retract(current_phase(2)),
 asserta(current_phase(mission_abort)).
```

Figure 10: Mission phase 2 in PROLOG

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] A. Healey, D. Marco, R. McGhee, D. Brutzman, R. Cristi,"Evaluation of a Tri-Level Hibrid Control System for the NPS PHOENIX Autonomous Underwater Vehicle," *Proc. US/Portugal Workshop on Undersea Robotics and Intelligent Control*, Lisbon, Portugal, March 1995, pp. 78-86.

[9] M. Der 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.

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

[11] P. Oliveira, A. Pascoal, V. Silva, C. Silvestre, "Design, Development, and Testing of a Mission Control System for the MARIUS AUV", *Proceedings of the 3rd Workshop on Mobile Robots for Subsea Environments*, Toulon, France, March 1996.

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

[13] 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.

[14] 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.