Execution Control of the NGC tasks for ROVs
P. Coletta, R. Bono, G.Bruzzone, M. Caccia, G. Veruggio
Consiglio Nazionale delle Ricerche
Istituto Automazione Navale
Via De Marini, 6
16149 Genova
Italy

Abstract
The problem of designing an interface between the continuous-state and the discrete-state domains of intelligent control architectures is addressed in this paper, focusing on the case of a hierarchical navigation, guidance and control architecture for unmanned underwater vehicles. The proposed interface represents the underlying continuous-state execution level as a discrete event system using a Petri net formalism. The correct behavior of the execution level is ensured, checking that no forbidden state is reached and that the proper task activation and deactivation order is respected. Methodologies for the off-line generation of the Petri net representation of the Execution level and real-time automatic system reconfiguration are presented.

Keywords: underwater vehicles, Petri nets, hybrid systems.

1. Introduction
To date, a relatively large set of navigation, guidance and control (NGC) algorithms have been developed for maneuvering unmanned underwater vehicles (UUVs) with various degrees of precision in different environmental conditions and with different sensor configurations and mission goals [1][2]. The NGC tasks are usually embedded in the execution level of intelligent control architectures for UUVs [3][4][5], while asynchronous decisions about the proper tasks required to accomplish the current mission are made by the upper organization and coordination levels, in the case of (semi-)autonomous systems, or by the human operator, in the case of tele-operated systems. In any case, an interface is required, which represents the underlying asynchronous/continuous-state execution level as a discrete event system (DES). The Interface generates events from the continuous state domain to the discrete-state domain and maps symbols in the opposite versus, and ensures the correct behavior of the execution level, checking that no forbidden state is reached and that the proper task activation and deactivation order is respected. See, for instance, [6] and [7] for a thorough discussion on architectures for hybrid control and autonomous systems.

In order to allow the (semi-)autonomous execution of complex missions, research in the field of underwater robotics has focused on the design and implementation of mission control systems (MCS) capable of providing conditional execution of basic tasks for mission management, failure detection and resource allocation [8]. As clearly discussed in [9], basic MCS requirements represent the capabilities of enabling the human operator to define the vehicle’s mission and to interact with the execution of the mission program in a high level language [10], and of providing suitable tools for automatic conversion and formal verification of a mission plan into a mission program [11]. MCSs rely on the description of the execution level as a set of concurrent state machines, represented with hierarchical concurrent state machine [8] or Petri net formalism [9]. Because of their capabilities in modeling the connection of several state machines and providing analysis methods, e.g. invariants and reachability trees, enabling the detection of potential anomalies in the behavior of the DES, Petri nets have been usually adopted for MCS modeling. A detailed example of Petri net-based Mission Control System is presented in [9], while an interesting analysis of the relations between Prolog and Petri net representations is reported in [1]. The definition of generic vehicle primitives, representing the elementary operations performed by an UUV, and the adoption of hierarchical intelligent control architectures allowed the integration of mission control systems and execution levels developed by different research groups, as in the case of the mission control of the NPS Phoenix AUV [12] and CNR-IAN Romeo ROV [13] performed by the IST-ISR CORAL/Petri net mission controller.

In this framework, this paper focuses on the design of an execution control module for UUVs that ensures the correct behavior of the execution level and simplifies the coordination activities by embedding the automatic reconfiguration capabilities which can be derived by the analysis of the hierarchical I/O relationships between the NGC tasks. The resulting execution control module provides a suitable interface for both an automatic mission control task coordination modules and a ROV pilot for tele-operation. To guarantee that the execution level, modeled as a Petri net, respects some correctness rules, a feedback is added to the Petri net to ensure that forbidden states are avoided. This is done using well known results from the literature on feedback control of Petri Nets based on place invariants [14]. The result is an execution control module which filters higher level commands, neglecting those that could lead to undesired states and generates the proper sequence of commands for the execution level when the requested operation cannot be performed in a single step.

The paper is organized as it follows. Section 2 describes the system architecture, discussing the role played by the Execution Control module the paper deals with. A
suitable representation of the Execution level in terms of Petri nets, embedding a set of basic rules linking the task I/O relationships to the structure of the system architecture, is discussed in section 3. A basic module able to execute the desired task activation/deactivation commands, maintaining the system in a sequence of admissible states, is presented. This module can automatically reconfigure the system in real-time if needed. Section 4 and 5 report some short notes on the state of development and testing of the system and current research respectively.

2. System architecture

According to the conventional paradigm of intelligent hierarchical control architectures [15], Romeo’s Control System is constituted by a synchronous Execution Level performing navigation, guidance and control tasks and an asynchronous Coordination Level that schedules the suitable NGC tasks on the basis of the events generated by NGC variables monitoring. This situation can be accommodated in the hybrid system framework presented in [6], which considers simultaneously the control and decision-making issues. Adopting such hybrid systems point of view, an Interface is needed to convert the continuous-time (synchronous) signals of the Execution level to discrete-event (asynchronous) symbols of the Coordination Level and vice versa.

As shown in Figure 1, where the basic system architecture is sketched, the Actuation and Sensor Systems provide the Execution level with a set of logical propulsion and sensing modules embedding the interfaces with physical devices. The Execution Level (EL) relies on a dual-loop hierarchical guidance and control system uncoupling the management of the vehicle’s kinematics (guidance), both absolute and environment-based, and dynamics (control) [2]. The operational variables $\chi$, that describe the guidance kinematics task functions, and robot’s velocities $\xi$ are estimated by a Motion and Environment Estimation module which processes basic internal (compass, gyro, inclinometers, depthmeter, Doppler velocimeter, etc.) and external (echo-sounders, etc.) sensor measurements.

Events $E$, signaling particular interactions of the robot with the operating environment and the status of advancement of NGC tasks, are generated by the Execution Event Generator which monitors the tasks’ state and performances $\Theta$. On this basis, the Coordination Level dynamically schedules the guidance, control and motion estimation tasks in order to reach the desired goal. The Execution Control module checks if the commanded activation/deactivation $f$ of the EL tasks can be executed. In the case this leads to a forbidden state, the Execution Control determines a suitable sequence of commands $\{f\}$ which enables the execution of the desired one. It is worth noting that the EC module only executes commands received by the coordination level: dynamic re-planning to handle unexpected events is performed by the upper level.

Thus, the Interface between the Coordination and Execution levels relies on two basic modules generating basic NGC events from the continuous state domain to the discrete-state domain and guaranteeing the correct behavior of the execution level checking that the proper activation and deactivation order is respected. A possible representation, implementation and application of the latter module, e.g. the Execution Control, is discussed in the following.

3. Execution control

3.1 Execution level: tasks, variables and connection rules

The Execution level embeds a set of elementary tasks, i.e. software components capable of performing specific NGC operations, which communicate through variables, i.e. shared memory used for task I/O. An example is given in Figure 2, where tasks (circles) and variables (squares) related to the control of the ROV motion in the vertical plane are represented.

3.1.1 Tasks and variables nomenclature

According to the their semantic, variables can be classified in estimation or reference variables. The Estimation Variables $EV$ contain the values measured by sensors (e.g. depth $z$, altitude $h$ and vertical force $f_w$ in the example of Fig. 2), as well as the outputs of the filtering algorithms (e.g. estimated depth $\hat{z}$, heave $\hat{w}$ and altitude $\hat{h}$), while the Reference Variables $RV$ represent the references to be tracked by the control tasks.
The set of all variable of the Execution level is denoted with symbol $V$, while the two subsets of $V$, $EV$ and $RV$, are such that $V = EV \cup RV$ and $\emptyset = EV \cap RV$.

Denoting with $T$ the set of all the tasks, the sets of input and output variables of each task are defined as it follows:

- $i_t \subseteq V$ and $i_V \subseteq V$ denote respectively the set of input and output variables of task $t \in T$;
- $RV = i_t \cap RV \subseteq RV$ and $r_V = i_V \cap RV \subseteq RV$ denote respectively the set of input and output reference variables of task $t \in T$;
- $EV = i_t \cap EV \subseteq EV$ and $e_V = i_V \cap EV \subseteq EV$ denote respectively the set of input and output estimation variables of task $t \in T$.

Assuming that all the output variables of a task belong to the same subset ($EV$ or $RV$), i.e. $RV \cap e_V = \emptyset$, $T$ is partitioned into two subsets: the set of the Estimation Tasks, $ET = \{ t \in T : i_t \cap EV = \emptyset \}$, constituted by all the tasks with estimation variables as output, and the set of the Reference Tasks (referred also as Control Tasks), $RT = \{ t \in T : t \in ET \}$, such that $ET \cap RT = \emptyset$ and $ET \cup RT = T$.

Estimation tasks cannot have reference variables as input, i.e. $i_t \in ET \Rightarrow r_V i_t \in \emptyset$.

It is worth noting that, in addition to the guidance and control tasks tracking position (e.g. autoDepthPI, autoDepthP, autoAltitudePI) and velocity (e.g. autoAltitudePI) references, the set of the reference tasks also includes the tasks with no output variables (e.g. verticalActuators). In this case, a logical actuator task, when activated, switches on the corresponding logical propulsion system, which maps the required vertical force onto the vehicle’s thruster thrusts.

Actuator tasks are an example of border tasks, that interface the execution level with logical modules such as the propulsion and sensor systems and the operator. Logical sensor tasks are estimation tasks that turn on/off the corresponding devices (e.g. depthmeter, forcemeter, altimeter). When active, they write the measurements received from the sensors to the proper estimation variables of the execution level.

In the same way, the interface with the upper level is managed defining a set of border tasks (e.g. zRefOp, hRefOp, wRefOp, fwRefOp) that receive the reference values from the pilot, and write them to reference variables of the execution level. The Logical Pilot tasks enable the human operator to interact with each level of the control architecture according to the tele-operation paradigm.

### 3.1.2 Task connection rules

The execution of complex missions requires that the connections between tasks be dynamically established according to mission events and requirements. Each task does not a priori know which tasks produce/consume its input/output variables, and suitable task activation/deactivation operations determine these connections at any time. Thus, the set $T$ of all tasks can be partitioned into the two subsets $T_R$ (the set of all running/active tasks) and $T_I$ (the set of all idle/inactive tasks).

Since the number of possible configuration becomes intractable for a human operator as soon as the number of task increases, an automatic supervision is needed to ensure that only correct configurations are reached.

Four basic rules, linking the task I/O relationships to the structure of the control and motion estimation architecture, enable the verification of the correctness of any task configuration. The rules describe the conflicts and the dependencies of the tasks, and must be respected at any time instant.

**Rule 1.** "no concurrent writing": each variable can be written by at most one running task:

$$\forall v \in V, \text{ if } \exists t_a, t_b \in T_R : v \in t_a^V \wedge v \in t_b^V \Rightarrow t_a \equiv t_b$$

For instance, in the example of Figure 2, only one task among the depth and altitude controllers and the external heave operator can be running at any time and write the reference heave $w^*$. 

**Rule 2.** "no concurrent tracking": each reference variable can be read by at most one running task:

$$\forall v \in RV, \text{ if } \exists t_a, t_b \in T_R : v \in t_a^V \wedge v \in t_b^V \Rightarrow t_a \equiv t_b$$

This constraints, together with the next, embodies the hierarchical structure of the control leg of the NGC architecture. In the example, only one depth controller can be running at any time and track the reference depth $z^*$. It is worth noting that the conflict would also exist with a depth controller generating the reference pitch rate.
Rule 3, “complete tracking of generated references”: if a reference task is running all its output reference variables must be read by a running task:
\[
\forall v \in RV, \text{if } \exists t_a \in T_R: v \in T_{RV}^a \Rightarrow \exists t_b \in T_R: v \in T_{RV}^b
\]
This rule ensures that each control chain is activated from the bottom as usually happens during human teleoperation. This prevents instability problems which could take place in case the reference values generated by an integral controller are not immediately applied to the system. Thus, in the example, in order to enable the operator to command the vehicle heave, the “verticalActuators” logical task has to be activated firstly. This originates the situation of a reference task tracking a reference variable which is not generated by any active task. To guarantee system stability in this typical operating conditions, the reference tasks are provided of suitable activation functions setting the reference variables in such a way of zeroing the required control action.

Rule 4, “complete writing of consumed estimation variables”: if a task is running (both a reference or estimation task), all its input estimation variables must be written by a running task.
\[
\forall v \in EV, \text{if } \exists t_a \in T_R: v \in T_{EV}^a \Rightarrow \exists t_b \in T_E: v \in T_{EV}^b
\]
This constraint embodies the hierarchical structure of the sensing leg of the NGC architecture. In the example, the model-based estimator of the vehicle vertical motion cannot run if no depth measurements are available.

It is worth noting that the above discussed rules enable the execution of banks of filters, i.e. Estimation Tasks, generating Estimation Variables both for control and monitoring purposes.

In order to guarantee system stability in typical operating conditions, the reference tasks are provided of deactivation functions, generating suitable commands when the corresponding task is switched off. In particular, tasks generating position commands, when deactivated, set their reference outputs equal to their current estimates, while controllers generating velocities and force and/or torque set their output to zero. This behavior, for instance, enables the vehicle to keep the current depth when commands from the external operator are disabled, and to remain almost still in the vertical plane during a transition between depth and altitude control.

3.2 Petri net representation

A method to design the Execution Control Level (ECL) will be introduced in the following. The proposed procedure guarantees that the rules on the behavior of the EL, stated in the previous section, are respected. This is done by representing the EL as a Discrete Event System, and applying well known results about the control of DES. A discussion about this topic is beyond the scope of this paper, and the reader is referred to [16] and the references therein for a survey on the subject. Moreover the specialization of the general theory developed in [16] to the Petri nets framework [17] is fully described in [14]. In the following it will be showed how to build a Petri net describing the EL behavior, and how to transform the previously stated rules to place invariants. Each NGC task can be modeled as a simple finite state machine (FSM), having only the two states idle and running. Because the connection of several states machines is no longer a FSM the whole ECL can be better modeled by a Petri net. So each task \( t \) of the ECL can be modeled by the simple net depicted in Figure 3: the initial marking is such that the sum of tokens in the net is 1, i.e. the initial state of the task is idle or running. Of course the net is such that a token is always present in one of the two places \( x_l \) or \( x_R \).

![Figure 3. Petri Net representation of a task](image)

In order to embed in the net the constraints given by the above discussed rules, results presented in [14] can be applied. Describing the net using the matrix notation, the following state equation is obtained:
\[
\begin{bmatrix}
  x_l^{t+1}  \\
  x_R^{t+1}
\end{bmatrix} =
\begin{bmatrix}
  x_l^{t}  \\
  x_R^{t}
\end{bmatrix} +
\begin{bmatrix}
  1 & 1 & f_A \\
  1 & 1 & f_D
\end{bmatrix}
\]
(5)

The description of the whole ECL can be simply obtained by multiplying the dimension of the state for the number of tasks in the execution level, and by describing each task with a Petri net. So the following net results:
\[
\begin{bmatrix}
  x_l^{t+1}  \\
  x_R^{t+1}
\end{bmatrix} = x_l^t + D f_k
\]
(6)
with:
\[
x = \begin{bmatrix} x_l^{t}, x_R^{t}, x_l^{t-1}, x_R^{t-1}, \ldots, x_l^{0}, x_R^{0} \end{bmatrix}
\]
(7)
\[
f = \begin{bmatrix} f_A^t, f_D^t, f_A^{t-1}, f_D^{t-1}, \ldots, f_A^{0}, f_D^{0} \end{bmatrix}
\]
(8)
and
\[
D = \begin{bmatrix}
  -1 & 1 & 0 & 0 & \ldots & 0 & 0  \\
  1 & -1 & 0 & 0 & \ldots & 0 & 0  \\
  0 & 0 & -1 & 1 & \ldots & 0 & 0  \\
  0 & 0 & 1 & -1 & \ldots & 0 & 0  \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots  \\
  0 & 0 & 0 & 0 & \ldots & 0 & -1  \\
  0 & 0 & 0 & 0 & \ldots & 0 & 1
\end{bmatrix}
\]
(9)
where \( N \) is the cardinality of set \( T \). Of course, at this stage, the net describes a set of unconnected tasks, i.e. a model where the activation/deactivation of all tasks are independent. To ensure that the rules stated in the previous section are respected the net must be modified. This could be operated manually, by adding an
appropriate number of places and arcs to the net. Of course the problem becomes intractable for a human as long as the number of tasks grows up. The approach proposed here is to transform the rules (1), (2), (3), and (4) in such a way that they can be expressed as place invariants on the marking of the net. Simple matrix operations allows to modify the net in such a way that it possesses the requested invariants. Thus, the net describes a model that behaves according to the prescribed rules. 

Consider, for instance, the Rule 2, which forbids concurrent tracking of the same reference variable:

\[ \forall v \in RV, \exists t_a, t_b \in T_k : v \in RV, t_a \wedge v \in RV, t_b \Rightarrow t_a \equiv t_b. \]

The fact that if two or more reference tasks have a common input reference variable then they cannot be in running state at the same time can be expressed as a constraint on the marking of the net:

**Rule 2:** \[ \sum_{i,v \in RV} x_i^v \leq 1, \quad \forall v \in RV \]  
(10)

In fact, if there exist two tasks \( t_a \) and \( t_b \) with a common input variable, then the sum of tokens in the places \( x_i^v \) and \( x_i^v \) can be at most one, so only one of the two tasks is allowed to be in state running (or, equivalently, if they are both in state running then they must be the same task).

In a similar way Rule 1 can be expressed by the constraint:

**Rule 1:** \[ \sum_{i,v \in V} x_i^v \leq 1, \quad \forall v \in V \]  
(11)

Denoting with \( RT_{\sim v} \) the set of reference tasks having \( v \) as input reference variable, and \( ET_{v} \) the set of estimation tasks having \( v \) as output estimation variable, Rules 3 and 4 can be expressed by the following constraints on the marking of the net:

**Rule 3:** \[ \forall v \in CV, \forall t \in CT : v \in t^{CV}_i, x_i^v + \sum_{t \in CT, t \neq v} x_i^v \geq 1 \]  
(12)

**Rule 4:** \[ \forall v \in EV, \forall t \in T : v \in t^{EV}_i, x_i^v + \sum_{t \in CT, t \neq v} x_i^v \geq 1 \]  
(13)

The above defined constraints can be transformed into equality constraints according to the method well described in [14] (the reader is referred to that paper for a deeper insight of the procedure). Putting all things together a new controlled net having the following form is obtained:

\[ X_{k+1} = \left[ \begin{array}{c} x \\ \tilde{x}_C \end{array} \right] = \left[ \begin{array}{c} x \\ \tilde{x}_C \end{array} \right] + \left[ \begin{array}{c} D \\ D_c \end{array} \right] f^*_k \]  
(14)

where \( x \) denotes the state of the original net, \( \tilde{x}_C \) denotes the state of the controlling net, \( D \) is the original transition matrix and \( D_c \) is the transition matrix of the controlling net. The new net (14) describes a model that behaves according to the desired rules. Defined

\[ X_k = [x_1 \ldots x_M]^T \geq 0 \Leftrightarrow (X_i)_k \geq 0, \forall i \in [1,M] \]  
(15)

\[ X_k < 0, \text{otherwise} \]

a set of transitions \( f^*_k \) can be fired at time \( k \) if and only if all the places in the net at time \( (k+1) \) have a non negative marking, i.e.

\[ X_{k+1} = X_k + \tilde{D} f^*_k \geq 0 \]  
(16)

It is worth noting that the Petri net (14) can be built off-line on the basis of rules (1)...(4) and the static task input/output mapping defined by the user.

### 3.3 Execution Control Engine

The Execution Control Engine embodies the Petri net representation of the Execution level presented in the previous section and acts as a run-time filter on the commands from the Coordination level and/or the human operator. Before allowing a task activation/deactivation to be executed the Execution Control Engine checks if the corresponding transition \( f^*_k \) in the modified net is enabled. In this case the command is sent to the Execution level and the net marking is updated accordingly, otherwise a suitable sequence of commands such that the request can be satisfied has to be determined and executed.

In the ideal case of transitions firing in zero time, given a requested transition and a net state \( \{f^*_k, X_k\} : X_{k+1} < 0 \), the Execution Control Engine should find a transition vector \( \tilde{f}^*_k : X_{k+1} = X_k + \tilde{D} f^*_k \geq 0 \) AND \( \tilde{f}^*_k \wedge f^*_k = f^*_k \)  
(17)

where \( \wedge \) is the array AND operator, and \( \tilde{f}^*_k \) is chosen such that it minimizes the number of fired transitions. It is worth noting that the latter condition guarantees that the final state includes the desired one.

Typical ROV operating conditions do not guarantee that transitions fire in zero time. This is, for instance the case of the estimators, which require an initialization time and can fail because of bad sensor measurements.

Thus, the Execution Control Engine is requested to determine a sequence of transition vectors, activating/deactivating only one task,

\[ \{\tilde{f}^*_k \ldots \tilde{f}^*_k\} : X_{k+1} = X_k + \sum_{i=0}^{m} \tilde{f}^*_k \geq 0, \forall j \in [0,m] \]  
(18)

AND \( \sum_{i=0}^{m} \tilde{f}^*_k \wedge f^*_k = f^*_k \)

i.e. each task activation/deactivation maintains the system in an admissible state (see the former condition). In this case, a solution minimizing the number of transitions is selected. This problem can be solved in real-time by adopting conventional graph search algorithms.
4. Execution Control implementation
The Execution Control module discussed in the previous section has been implemented and integrated in the Romeo’s ROV control architecture [18]. In particular, an off-line Petri Net Generator (PNG), running on a PC Windows, and an on-line Execution Control Engine (ECE), running on a MVME162 board supporting VxWorks and located on-board the vehicle, have been developed in C++ language. In particular, the PNG generates a file describing the controlled Petri net (14) on the basis of the reference and estimation I/O variables specified by the user for each task. The generated file is processed by the real-time ECE to build a suitable data structure, and process commands as seen in section 3.3. Tests carried out during Romeo’s operations at sea demonstrated the execution control module is able to automatically reconfigure the NGC system according to pilot requests. results are reported in [19].

5. Conclusions
The problem of designing an execution control module for UUVs that ensures the correct behavior of the execution level and embeds basic automatic reconfiguration capabilities has been addressed in this paper. First application results during Romeo operations are quite promising. Current research is focusing on the introduction of task priorities, and the improvement of the performances of the graph search module.

References