A PETRI NET APPROACH TO THE DESIGN OF ROV CONTROL SYSTEMS

Paolo Coletta[*,**]

[*]**Institute for Naval Automation (CNR–IAN),**
National Research Council of Italy,
Via De Marini 6, 16149 Genova, Italy.
[paolo@ian.ge.cnr.it]

[**] **Department of Communication, Computer and System Sciences,**
University of Genoa,
Via Opera Pia 13, 16145 Genova, Italy.

Abstract

A method to design the control architecture for ROVs is presented, and is based on the idea of using the Petri net formalism. The amount of system specification required from the designer is reduced to the declaration of the input and output variables of the tasks. Using this information solely, a characterization of the "safe" behaviour is derived. Then this behaviour is described as a set of invariants on the marking of a Petri net. A supervisor that enforces these invariants is derived, and it allows the on-line verification of the safety of the control architecture, and the derivation of proper activation strategies. The application of the methodology to the design of the control architecture of Romeo, the CNR–IAN's ROV, is presented.

1 Introduction

The focus of papers concerned with mobile robots navigation is mostly directed toward the investigation of control and filtering techniques. The architectural aspects and the research of methodologies for the control system design have not gained the same attention. In [1] an information theoretic approach has been adopted, and hierachical control architectures are proposed. Three layers are usually used: in [2] the three levels are denoted, from bottom up, *Functional Level*, *Execution Control Level* or *Executive*, and *Decision Level*. The functional level embeds the elementary control tasks, that are executed in real time for guaranteeing the effective control of the vehicle. The execution control level provides the interface between the slow symbolic processing of the decision level and the fast, continuous state processing of the functional level. The higher decision level is responsible of the planning and decision making issues. Although the notation is not homogeneous among the various authors, the necessity of the interface between the continuous state and the discrete state world has been stressed elsewhere (see, e.g., [2], [3]). In [2] the interface is seen as a means to manage the logical constraints and dependencies among the tasks of the functional level, whereas in [3] these topics are not faced, and the focus is more on the stability issues of the resulting hybrid architecture. In the framework of underwater robotics, the functional level of intelligent control architectures ([4, 5, 6]) usually embeds the basic navigation, guidance and control algorithms, while asynchronous decisions about the configuration of the tasks required to accomplish the mission goals are made by the upper levels, in the case of (semi-)autonomous systems, or by the human operator, in the case of tele-operated systems.

The focus of this paper is on the design of the execution control level of the control architecture for ROVs, and on the usage of the Petri net formalism for the synthesis of this component. Petri nets are a well-know mathematical tool for the description of systems where concurrency, parallelism, and resource contention are involved (see [7] for an introduction on this topic). They have been used in various fields related to robotics ([8, 9, 10, 11, 12]), and in earlier versions of this work ([13, 14]). In [8] high-level timed Petri nets are extended with the concept of modules in order to simplify the specifi-

cations of complex control systems. In [9] Petri nets are used for solving the assembly sequence planning problem. Thus, the scope of the use of the Petri net is to provide a clear description of the conflicts and dependencies of the single operations, in order to obtain, from the Petri net model, only valid sequences of operations. Moreover, Petri nets are used in [11] to provide formal specifications of the entire architecture of the mobile robot. In that work, the formalism allows a complete representation of the interactions among the levels (execution, coordination, and organization, in their notation) of the control system. Finally, in [12] Petri nets are used along the whole development cycle of the control architecture, that is, in the specification, validation and code generation phases.

In the underwater robotics field, Petri nets have even been used in the development of mission control system (MCS) capable of providing resource, mission, and failure management. Basic MCS requirement include the capability of enabling the human operator to design in a high-level language the vehicle mission, and to interact with the execution, in case of semi-autonomous vehicles. In [15] hierarchical concurrent state machines are used to describe the functional level, while in [16] a Petri net formalism is preferred.

This paper is organized as follows. Section 2 gives a brief description of the Petri nets formalism. Section 3 provides a general overview of the proposed control architecture, and a more detailed description of the execution level is drawn in Section 4. The proposed design technique is introduced in Section 5, while Section 6 gives a description of the application of the technique to the CNR–IAN's ROV Romeo, and provides a view of the exploitation of the system in at-sea operations. Finally, Section 7 is devoted to the conclusions and the perspectives for future works.

2 Petri nets

Petri nets ([7]) are a well-know graphical and mathematical tool for the representation of discrete-event systems; the brief description presented here serves only to establish the notation used throughout the paper. Formally, a Petri net is a quadruple $\mathcal{N} = (P, T, I, O)$, where P and T are the set of places and transitions respectively, $I \subseteq P \times T$ is the set of arcs from places to transitions, $O \subseteq T \times P$ is the set of arcs from transitions to places. A marking is a function that assigns to each place $p \in P$ a non-negative integer number, representing the number of tokens in that place; that is, $x : K \times P \rightarrow \mathbb{Z}$, $x(\cdot, p) \geq 0 \quad \forall p \in P$, and $K = \{0, 1, 2, \ldots\}$ is a time-index set. Thus, $\mathcal{MN} = (P, T, I, O, x, \underline{x}_0)$ is a marked Petri net. The symbol $\underline{x}$ denotes the marking vector, that is, the ensemble of the markings of each place: $\underline{x}_k = [x(k, p_1), x(k, p_2), \ldots, x(k, p_N)]'$, where $\{p_1, p_2, \ldots, p_N\}$ is the set P. Hence, $\underline{x}_0$ is the initial marking of the net.

A transition $t \in T$ is said to be enabled at time k if there are tokens in its input places, i.e., if $x(k, p) > 0 \; \forall p : (p, t) \in I$. When a transition is enabled, it may eventually fire, and when this happens the marking of the Petri net is update by removing one token from each input place of the transition, and adding one token to each output place. More precisely, each entry of the marking vector $\underline{x}_k$ is updated as follows:

$$x(k+1, p) = \begin{cases} x(k, p) + 1 & if\,(p, t) \in I \\ x(k, p) - 1 & if\,(t, p) \in 0 \\ x(k, p) & otherwise\,, \end{cases}$$

where $p = p_1, p_2, \ldots, p_N$, and k is the firing index ($k = 0, 1, 2, \ldots$). This behaviour can be described by the following matrix equation:

$$\underline{x}_{k+1} = \underline{x}_k + D\underline{f}_k$$

where $\underline{f}_k$ is the k_{th} firing vector. The M entries of the firing vector (where M is the number of transitions) are all $0s$, except one 1 in the i_{th} position, indicating that transition i fires at the k_{th} firing. The matrix D is an $N \times M$ matrix, where each entry d_{ij} is:

$$d_{ij} = \begin{cases} -1 & if\,(i, j) \in I \\ +1 & if\,(j, i) \in O \\ 0 & otherwise\,. \end{cases}$$

Given the initial marking $\underline{x}_0$, we may define a place invariant as the integer vector $\underline{\mu}$ that satisfies $\underline{\mu}'\,\underline{x}_k = \underline{\mu}'\,\underline{x}_0$, for $k = 1, 2, \ldots$. Place invariants will be used in Section 5 to ensure the correct behaviour of the execution control level.

3 System architecture

The execution level is made of a collection of tasks, whose activation/deactivation is established by the coordination level, and verified by the execution control module. The coordination level handles the management of complex mission plans, and determines the required configuration of the execution level necessary to reach the current mission goals. A possible implementation of the coordination level

relies on reactive planner-based solutions. Recent results on this topic [17], where the usage of spreading activation networks is proposed, are being investigated. The system architecture is depicted in Figure 1.

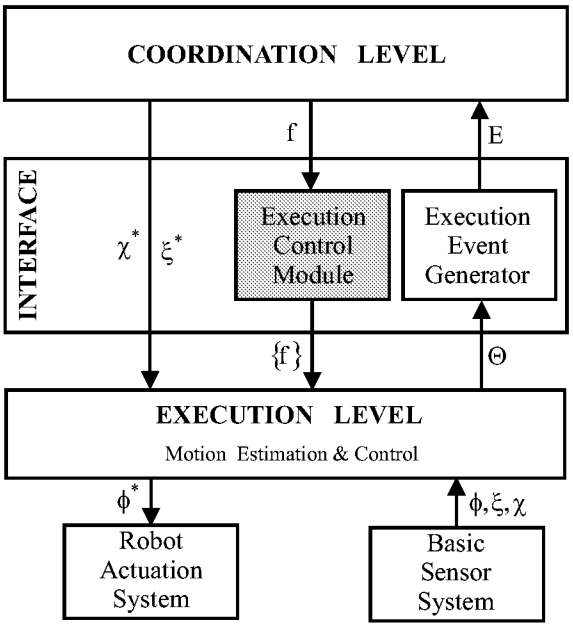

Figure 1: Control system architecture.

The interface is made up of two main modules, one receiving requests from the coordination level and generating command sequences for the execution level (the execution control module, in Figure 1); the other one generating events on the basis of the state of tasks at the execution level (execution event generator in Figure 1).

4 Execution level

The execution level embeds a set of elementary tasks, i.e., software components with the ability to perform specific motion estimation or control functions. Tasks communicate among them by means of variables, that is, shared memory used to store the inputs and outputs of tasks. In the case of underwater vehicles, consider a control task computing the horizontal force that must be applied by the thrusters in order to keep the desired horizontal speed. This tasks needs, as inputs, the current estimate of the horizontal speed and the desired reference speed. It provides as output the horizontal force that has to be applied. In this example we have one task (the horizontal speed control task), and three variables (the horizontal estimated speed, the horizontal reference speed, the computed horizontal force). Each task can be seen as an indipendent entity, only needing the input and output variables for its proper working. This greatly simplifies their actual implementation, since the development of the various tasks can be done independently. Not all tasks are used at the same time: according to the current mission goals, it may occur that one is running, while others are not active. We'll use the term *running* to refer to tasks in use, whereas the unused tasks will be referred as *idle*. Of course, this is not a static property, in that it may change during time. The purpose of the execution control module is to guarantee that, at any time instant, only proper combinations of idle and running tasks are reached.

4.1 Control architecture example

In complex control architecture a high number of tasks may be involved; in the following we'll refer to the simple example depicted in Figure 2 to provide a meaningful description of the proposed methodology.

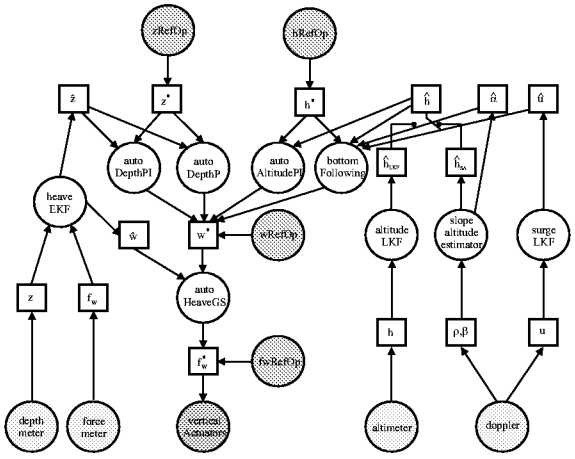

Figure 2: Example of a control architecture: only a part of the vertical motion is considered here.

The depicted system architecture represents a small subset of the control system of Romeo, and is mainly focused on the control of the vertical motion of the vehicle. The tasks and variables considered in Figure 2 are the followings: *vertical actuators* is the control task that provides the lower level interface with the vehicle thrusters, applying the desired vertical force f_w^*. *zRefOp*, *hRefOp*, *wRefOp*, and *fwRefOp* are the control tasks interfacing with the upper levels of the control architecture (or with the human operator), that is, their output is

the requested reference value for variables z^*, h^*, w^*, and fw^* respectively (depth, altitude, vertical speed, and vertical force). *auto depth PI* and *auto depth P* are "equivalent" control tasks that compute the vertical speed w^* necessary to keep the prescribed depth z^*. *Auto altitude* and *bottom following*, although computing the same output reference value w^* are not equivalent tasks, in that they have different requirements (the bottom following algorithm requires the knowledge of the slope of the seabed). *Auto heave GS* is the task computing the required vertical force. In the estimation leg *depth meter*, *force meter*, *altimeter*, and *doppler* are the estimation tasks interfacing with the logical sensors; *heave EKF*, *altitude LKF*, *slope altitude estimator*, *surge LKF* are estimation tasks, computing the estimated values of the heave, altitude seabed slope, and surge variables. See, for instance, [18] for a more detailed description of the tasks composing the ROMEO control system. Finally, there is a switch involving variables $\hat{h}, \hat{h}_{LKF}$, and $\hat{h}_{SA}$. The possibility of switching among estimation variables has been introduced for the purpose of providing the greater flexibility. The switch means that variable $\hat{h}$ can be given the same value of one of the two sources ($\hat{h}_{LKF}$ and $\hat{h}_{SA}$ in this case), or none of them. In facts, it is possible for variable $\hat{h}$ not to be given any value: in this case its value is undefined and tasks having $\hat{h}$ as input cannot be activated.

Note that the approach is somewhat different for the estimation or control cases. In the case where estimation variables are involved, it is possible to have different instances of variables with the same meaning, i.e., referring to the previous example, the variables $\hat{h}, \hat{h}_{LKF}$, and $\hat{h}_{SA}$ all refer to the same physical quantity (the distance from the seabed, in the underwater vehicles field). Anyway, $\hat{h}_{LKF}$ and $\hat{h}_{SA}$ are estimated with different algorithms, and $\hat{h}$ can be one of the two according to some criteria. Thus, even when the position of the switch is such that $\hat{h}$ and $\hat{h}_{SA}$ have the same value, the task *slope altitude estimator* can be kept active and continue providing its estimate of the altitude; this can be useful for monitoring or fault detection purposes. Hence, even bank of filters are easily accomodated in the proposed framework. On the contrary, as far as control variables are concerned, when different control algorithms compute the same physical quantity (e.g., the forward speed), they use the same variable as output. This, as will become clearer in the following sections, creates a conflict between the control tasks. In facts, having a reference value computed by means of different methods is not of great utility and, in most practical cases, it is impossible since a control algorithm cannot be kept running without applying its output. Consider two control algorithms, one with proportional/integral action, and one with the proportional part only. The integral one would soon "saturate" if kept running while the output of the only-proportional one is used.

Tasks *auto depth P* and *auto depth PI* have the same set input and output variable, that is they are topologically equivalent. Control tasks with this characteristic are called *equivalent*, and a special operation to switch among them will be introduced. This allows to change the low-level control algorithms without affecting the upper levels configuration.

4.2 Nomenclature

In order to describe the execution level by means of the Petri net formalism, a more precise definition of its various components is needed. This is the scope of the following definitions. Let us denote with symbols $\mathcal{T}$ and $\mathcal{V}$ the sets of all tasks and variables, respectively. The items of the two sets are furtherly distinguished according to their relation with the control or navigation leg of the architecture. Hence, the set $\mathcal{V}$ is made up of the two disjoint subsets $\mathcal{EV}$, the set of estimation variables, and $\mathcal{CV}$ the set of control variables. Variables must belong to one and only one of the two categories, that is, $\mathcal{V} = \mathcal{EV} \cup \mathcal{CV}$ and $\emptyset = \mathcal{EV} \cap \mathcal{CV}$. For each task $t \in \mathcal{T}$, it is possible to define the sets of input and output variables:

- ${}^{\mathcal{V}}t \subseteq \mathcal{V}$ and $t^{\mathcal{V}} \subseteq \mathcal{V}$ denote the set of input and output variables of task t, respectively.
- ${}^{\mathcal{CV}}t \triangleq {}^{\mathcal{V}}t \cap \mathcal{CV} \subseteq \mathcal{CV}$ and $t^{\mathcal{CV}} \triangleq t^{\mathcal{V}} \cap \mathcal{CV} \subseteq \mathcal{CV}$ denote the set of input and output control variables of task t, respectively.
- ${}^{\mathcal{EV}}t \triangleq {}^{\mathcal{V}}t \cap \mathcal{EV} \subseteq \mathcal{EV}$ and $t^{\mathcal{EV}} \triangleq t^{\mathcal{V}} \cap \mathcal{EV} \subseteq \mathcal{EV}$ denote the set of input and output estimation variables of task t, respectively.

Therefore, the partitioning of the set $\mathcal{T}$ into the two disjoint subsets $\mathcal{ET}$ and $\mathcal{CT}$ of the estimation and control tasks respectively can be easily derived as $\mathcal{ET} \triangleq \{t \in \mathcal{T} : t^{\mathcal{V}} \cap \mathcal{EV} \neq \emptyset\}$ and $\mathcal{CT} \triangleq \mathcal{T} \setminus \mathcal{ET}$. Besides this static partitioning of the tasks, a time-varying partitioning can be established between the running and the idle tasks. That is, there exist two subsets of $\mathcal{T}$, $\mathcal{T}_R$ and $\mathcal{T}_I$, such that $\mathcal{T}_R \cup \mathcal{T}_I = \mathcal{T}$ and $\mathcal{T}_R \cap \mathcal{T}_I = \emptyset$. $\mathcal{T}_R$ is the set of running tasks, whereas $\mathcal{T}_I$ is the set of the idle ones. Since the belonging of one task to one of the two subset is a

time-varying property, it would be more precise to represent these sets as functions of time: however, in the following the time dependence will be dropped for notational simplicity.

Referring to the example in Figure 2, the sets $\mathcal{ET}$, $\mathcal{CT}$, $\mathcal{EV}$, $\mathcal{CV}$, are, respectively, { *force meter*, *altimeter*, *doppler*, *altitude LKF*, *slope altitude estimator*, *surge LKF*, *heave EKF* }, { *zRefOp*, *hRefOp*, *wRefOp*, *fwRefOp*, *auto depth P*, *auto depth PI*, *auto altitude PI*, *bottom following*, *auto heave GS* }, { h, ρ, β, u, $\hat{h}$, $\hat{\alpha}$, $\hat{u}$, z, f_w, $\hat{z}$, $\hat{w}$ }, and { z^*, h^*, w^*, f_w^* }.

4.3 Rules

The definitions in the previous section let us state the rules that determine the correctness of the execution level. These rules regard the logical behaviour only, that is, stability issues are not considered here. Thus, the following rules are derived basing only on the topology of the execution level, i.e., on the interconnections among the tasks:

(R1) $\forall v \in \mathcal{V}, \exists t_a, t_b \in \mathcal{T}_R : v \in t_a^{\mathcal{V}} \wedge v \in t_b^{\mathcal{V}} \Rightarrow t_a \equiv t_b$

(R2) $\forall v \in \mathcal{CV}, if \exists t_a, t_b \in \mathcal{T}_R : v \in {}^{\mathcal{CV}}t_a \wedge v \in {}^{\mathcal{CV}}t_b \Rightarrow t_a \equiv t_b$

(R3) $\forall v \in \mathcal{CV}, if \exists t_a \in \mathcal{T}_R : v \in t_a^{\mathcal{CV}} \Rightarrow \exists t_b \in \mathcal{T}_R : v \in {}^{\mathcal{CV}}t_b$

(R4) $\forall v \in \mathcal{ET}, if \exists t_a \in \mathcal{T}_R : v \in {}^{\mathcal{EV}}t_a \Rightarrow \exists t_b \in \mathcal{ET} \cap \mathcal{T}_R : v \in t_b^{\mathcal{EV}}$

Rule (R1) states that, at any time instant, there can not be two or more running tasks having the same output variable. Rule (R2) ensures that there are no control tasks tracking the same reference value at the same time. Roughly speaking, this rule establishes the uniqueness of the control strategy. In (R3) it is affirmed that generated reference values must be tracked. Finally, rule (R4) states that each input estimation variable of a running task must be the output of another (necessarily, estimation) running task. That is, estimation tasks must be activated from bottom up, and before the control tasks using their outputs.

5 Petri net-based design of the execution control module

Tasks at the execution level are represented, in the execution control module, by means of Petri nets.

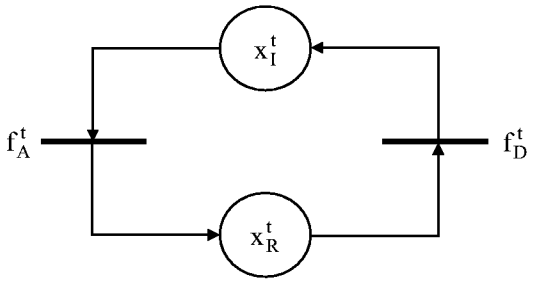

Figure 3: Sketch of the Petri net representing a simple task.

Each task can be represented by means of the simple Petri net depicted in Figure 3.

The places x_I^t and x_R^t refer to task t being in status *Idle* or *Running* respectively. The initial marking of the net ensures that only one of the two places is marked with one token. Moreover, the place invariant $x_I^t + x_R^t = 1$ is respected for $t = 1, 2, \ldots, Q$, where Q is the number of tasks in the control system. This means that the obvious condition of a task not being idle and running at the same time is always respected. The operation to take a task from state idle to state running, i.e., firing transition f_A^t in Figure 3, is called the activation of a task, whereas the opposite, i.e., the firing of transition f_D^t, is called deactivation of a task. The execution level does not activate or deactivate tasks on its own, but it always performs such operations in response to commands received from the upper levels. Thus, all transitions in the Petri net can be assumed controllable. Note that, in view of the recently appeared results on the supervision of Petri nets having both uncontrollabe and unosservable transition [19], this assumpion is not limiting the applicability of the proposed methodology to this simple type of representation.

Once described the single tasks, the representation of the whole architecture has to be found. At the first glance, the overall net is simply composed of the several simple independent nets, one for each task. However, such a description implies that the activation/deactivation of the tasks has no constraint, and thus it violates the rules stated in the previous section. Anyway, the description obtained so far is the base for applying the method proposed in [20] for the supervision of the control architecture. The applicability of that procedure relies on the representation of the net with the matrix notation. Hence, let first see how the simple net and the overall net are represented with this formalism.

Each task is simply described by:

$$\begin{bmatrix} x_I^t \\ x_R^t \end{bmatrix}_{k+1} = \begin{bmatrix} x_I^t \\ x_R^t \end{bmatrix}_k + \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} f_A^t \\ f_D^t \end{bmatrix}_k .$$

The whole net is obtained by juxtaposing the states of the single taks, i.e.:

$$\begin{bmatrix} x_I^1 \\ x_R^1 \\ x_I^2 \\ x_R^2 \\ \vdots \\ x_I^Q \\ x_R^Q \end{bmatrix}_{k+1} = \begin{bmatrix} x_I^1 \\ x_R^1 \\ x_I^2 \\ x_R^2 \\ \vdots \\ x_I^Q \\ x_R^Q \end{bmatrix}_k + [T] \begin{bmatrix} f_A^1 \\ f_D^1 \\ f_A^2 \\ f_D^2 \\ \vdots \\ f_A^Q \\ f_D^Q \end{bmatrix}_k$$

where

$$[T] = \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & -1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & -1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 1 & -1 \end{bmatrix} .$$

The switch among the estimation variables, that has been described in section 4, is represented as a set of parallel estimation tasks, as can be seen in Figure 4. The case where the switch is opened (i.e., the destination variable is not written) corresponds to having all tasks in the idle state. Moreover, the tasks are subject to rule (R1), and hence the uniqueness of the switch position is guaranteed, since all tasks have a common output variable. On the contrary, the case where equivalent control tasks are present needs to be described in a somewhat different way. In facts, the rules previously stated require the activation of control tasks to start from the lower levels toward the highers. Thus, in the case that one wants to substitute a control task that is required by another running control task at an upper level, the respect of the rules would imply the deactivation of the control tasks down to the lower level, and the activation with the new configuration up to the desired level. This solution is clearly undesiderable, and it is preferable to have an atomic operation allowing the switch between the equivalent tasks, leading the system from a valid configuration to another valid one, neglecting the intermediate unsafe situation. To this aim, equivalent control tasks are represented as in Figure 5, that is, transitions to switch from one configuration directly to the other one are added, in order to avoid to go through the idle state.

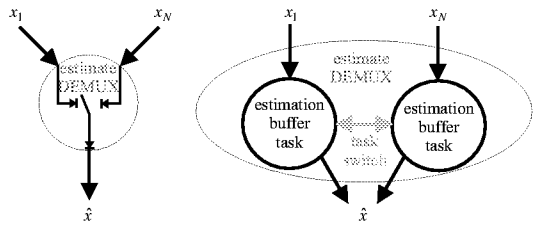

Figure 4: Representation of the switch among estimation variables.

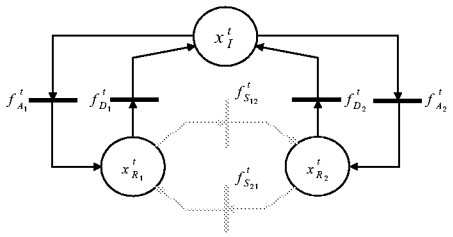

Figure 5: Sketch of the Petri net representing the switch between two equivalent control tasks.

The corresponding net dynamics, using the matrix notation, is given by the following:

$$\begin{bmatrix} x_I^t \\ x_{R_1}^t \\ x_{R_2}^t \end{bmatrix}_{k+1} = \begin{bmatrix} x_I^t \\ x_{R_1}^t \\ x_{R_2}^t \end{bmatrix}_k + [S_t] \begin{bmatrix} f_{A_1}^t \\ f_{D_1}^t \\ f_{A_2}^t \\ f_{D_2}^t \\ f_{S_{12}}^t \\ f_{S_{21}}^t \end{bmatrix}_k$$

where

$$[S_t] = \begin{bmatrix} -1 & 1 & -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 & 1 & -1 \end{bmatrix} . \quad (1)$$

Of course, the representation can be easily extended to cases involving more than two tasks. As in the case of the other tasks, the whole net is obtained by the juxtaposition of the nets representing the single tasks switches. That is, the transition matrix of the net including all the control tasks involved in equivalence relations has the following block structure:

$$S \triangleq \begin{bmatrix} S_1 & 0 & 0 & \cdots & 0 \\ 0 & S_2 & 0 & \cdots & 0 \\ 0 & 0 & S_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & S_w \end{bmatrix}$$

where w is the total number of equivalence relations, and each $S_i, i = 1, 2, \ldots, w$ has the same

structure as (1), but with an eventually different number of rows, according to the number of possible alternatives for the given control strategy.

Let us denote with symbol D the matrix of the net dynamics containing both the nets related to normal tasks and control tasks involved in equivalence relations. That is, $D \triangleq \begin{bmatrix} T & 0 \\ 0 & S \end{bmatrix}$, and the net dynamics is given by:

$$\underline{x}_{k+1} = \underline{x}_k + D\underline{f}_k \tag{2}$$

Now, the rules of the previous section must be converted into invariants on the marking of the net. Consider rule (R1), it can be expressed by means of the following set of inequalities:

$$\forall v \in \mathcal{V} \sum_{t_i : v \in t_i^{\mathcal{V}}} x_R^{t_i} \leq 1 \tag{3}$$

In facts, suppose that there exits two distinct tasks violating rule (R1), i.e., t_a and t_b such that $v \in t_a^{\mathcal{V}} \wedge v \in t_b^{\mathcal{V}}$, and such that they are both in state running at the same time (t_a and $t_b \in \mathcal{T}_R$); for these tasks we would have $x_R^{t_a} + x_R^{t_b} = 2$. That is, violation of rule (R1) leads to a violation of inequality (3). Hence, the enforcement of the inequality implies the respect of the rule.

In an analogous manner, the inequalities related to rules (R2), (R3), and (R4) can be obtained as:

$$\forall v \in \mathcal{CV} \sum_{t_i : v \in {}^{\mathcal{CV}}t_i} x_R^{t_i} \leq 1, \tag{4}$$

$$\forall t \in \mathcal{CT}\ x_R^t = 1 \Rightarrow \forall v \in {}^{\mathcal{CV}}t \sum_{t_i \in {}^{\mathcal{CT}}t_v} x_R^{t_i} \geq 1, \tag{5}$$

and

$$\forall t \in \mathcal{T}\ x_R^t = 1 \Rightarrow \forall v \in {}^{\mathcal{EV}}t \sum_{t_i \in t_v^{\mathcal{ET}}} x_R^{t_i} \geq 1, \tag{6}$$

where ${}^{\mathcal{CT}}t_v \triangleq \{x \in \mathcal{CT} : v \in {}^{\mathcal{CV}}t\}$ and $t_v^{\mathcal{ET}} \triangleq \{x \in \mathcal{ET} : v \in t^{\mathcal{ET}}\}$.

Inequalities (3), (4), (5), and (6) can be transformed into equalities, as it will be showed in the following. The method used is the one proposed in [20], and is reported here for ease of reference. The inequalities are grouped and written in matrix form as

$$L\underline{x} \leq \underline{b},$$

where L is a $N_c \times N$ matrix, $\underline{b}$ is a $N_c \times 1$ vector, and N_c is the number of constraints. Inequalities are rewritten as equalities by adding slack variables:

$$L\underline{x} + \underline{x}_c = \underline{b},$$

where $\underline{x}_c$ is a $N_c \times 1$ vector. Practically, N_c places are added to the original net. It has been shown ([20]) that the incidence matrix relative to these new places can be obtained as:

$$D_c = -L\,D$$

and the initial marking of these new places is given by:

$$\underline{x}_{c_0} = \underline{b} - L\underline{x}_0 .$$

Summing up, the following net is obtained:

$$\begin{bmatrix} \underline{x} \\ \underline{x}_c \end{bmatrix}_{k+1} = \begin{bmatrix} \underline{x} \\ \underline{x}_c \end{bmatrix}_k + \begin{bmatrix} D \\ D_c \end{bmatrix} \underline{f}_k , \tag{7}$$

where the vectors $\underline{x}$ and D denote the state and the transition matrix of the original net, respectively (see Equation (2)), while $\underline{x}_c$ and D_c denote the state and the transition matrix of the controlling net, respectively.

6 Application example

The design procedure introduced in Section 5 has been sactisfactorly used for the design of the logic part of the control architecture of Romeo, an underwater ROV developed by CNR–IAN. The vehicle ([21]) is used both for research purposes, that is, as a test-bed to experiment new control and navigation algorithms, and for scientific purposes, that is, as a carrier of scientific instruments in underwater environments. Thus, it is requested that the control architecture is flexible enough, in order to easily add new tasks to the system and integrate them into the existing architecture. The resulting software environment is made of two main on-line components: the *execution engine* and the *execution control engine*, implementing, respectively, the execution level and the execution control module. Moreover, an off-line visual editor, named *execution level designer*, has been developed; it is a Windows program that provides the environment for the configuration of the control system. That is, it provides a way to specify the tasks input and output variables, and the switches among the estimation variables.

A sketch of the relations among the various software tools is depicted in Figure 6.

6.1 Execution level designer

This module, running under Windows, provides the environment to design and configure the above-described modules. By means of a Graphical User Interface it is possible to declare new types of tasks,

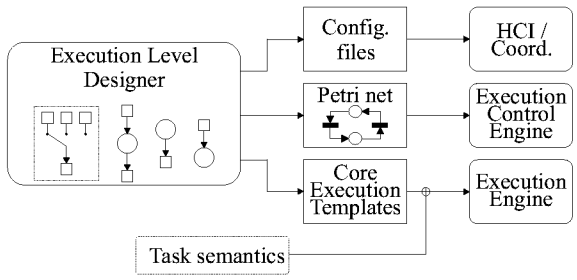

Figure 6: Software tools for the design of the control architecture of Romeo.

in this case the *C++* skeleton code for these tasks will be generated by this module. For each type of task, it is possible to specify, beside the number of input and output variables, a set of parameters (e.g., a proportional and integral constant in a PI regulator); the code to manage these parameters will be automatically generated. Moreover, in the same environment, it is possible to specify the number and names of the variables used in the execution level, and, for each task, to establish the links with the input and output variables. Basing only on this topological information, the execution level designer will compute the structure of the Petri net used by the execution control engine, i.e., the matrix $\begin{bmatrix} D \\ D_c \end{bmatrix}$ in equation (7). Summing up, this module generates the *C++* code to implement the control architecture. More precisely it generates:

- the execution level runtime engine and EL scheduling order. The scheduling order is computed by this module basing on the input/output relationships among the tasks. The human operator can choose the ordering where multiple alternatives are present.
- *C++* skeleton code of the execution level tasks. That is, this module generates the skeleton of the classes used to implement the tasks of the execution level.
- Code to instantiate tasks and variables of the execution engine.
- Full implementation of the execution control engine, including the initialization of the Petri net with the off line-computed structure.
- Configuration files for the Human Computer Interface customization. These files include the friendly names of the tasks of the execution level, the mapping of these names to the identifiers used in the execution control engine and in the execution engine, and the pre-computed topologically equivalent groups of control tasks.

6.2 Execution control engine

The execution control engine embeds the Petri net representation generated by the execution level designer. It receives commands (task activations or deactivations) from the upper levels of the architecture, or from the human operator, and translates the commands into firing vectors. The feasibility of the obtained firing vectors is verified, and, if the resulting marking is a valid one, the commands are sent to the execution engine. The marking of the net is updated upon receiving the confirmation that the command has been correctly executed. In fact, the execution engine could also reply in a negative way; this may happen, for example, when the activation of a task is not possible due to some physical constraint, as could be the temporary unavailability of a physical device. Another job performed by the execution control engine is the transformation of an unfeasible command into a proper sequence of commands such that, after having correctly executed these new ones, the initial command turns out to be feasible. This is realized by computing a firing sequence that leads the Petri net to a state where the original firing vector is enabled. Thus, the commands corresponding to the computed firing sequence and to the original firing vector are sent to the execution engine. The conflict resolution is based on a preemptive policy, that is, if the last request leads to a configuration where some tasks are in conflict with others that were already running, the running tasks are deactivated and the new ones are activated.

6.3 Execution engine

Tasks at the execution level are implemented by means of *C++* classes. A virtual base class is provided, that embeds the structure necessary to implement a state machine. To create new tasks, a *C++* class must be derived from the base class, implementing the methods that are called at each state. For most purposes, it is sufficient to implement the virtual method *run*, that is the one called when the task is in the running state. Moreover, it is also possible to implement the others methods, that are called when the task is in the idle state, during the initialization and deinitialization phases, and at every change of the state. The run-time engine is a simple scheduler that is activated periodically, and that, on its own, activates all the tasks. More pre-

cisely, for each task and in an order that is provided a priori (computed by the off-line module), it checks the state of the task and calls the corresponding method (e.g., if a task is in the running state, it calls the *run* method for that task). Furthermore, it also checks if the task has changed state from the previous turn and, if this has happened, calls the appropriate handler (another virtual method of the task, that can be redefined in derived classes). Another job performed by the run-time engine is to check if a task is in the initialization phase from a too long time (this timeout can be specified during the design phase by means of the off-line module). If that is the case, the task is reset to the idle state.

6.4 Exploitation

After intensive lab tests performed executing a large number of missions in the CNR–IAN Underwater Virtual World, successful trials have been performed in the Aegean Sea in the framework of the ARAMIS project. Moreover, many at-sea operations in the Tirrenian Sea nearby Genoa have been done. During all these missions, the control system has behaved satisfactorly, and the work performed by the human operator has proven to be simplified.

7 Conclusions

A method to design the control architecture for mobile robots has been proposed in this paper. The method relies on the Petri net formalism, and on the feedback control of Petri nets basing on place invariants. The proposed methodology allows the on-line supervision of the behaviour of the tasks at the execution level, and the automatic reconfiguration according to the current mission goals. The application to a real-world example, the Romeo ROV, has been described, including the software tools developed for the design and implementation of the proposed architecture. Future works will concern the extension of the methodology to a more general class of tasks, hence including tasks having uncontrollable behaviours. On the application side, the event generator module and the upper level of the architectures are the current research topics.

Acknowledgments

The author wishes to thank all the staff of the Robotics Department of the Naval Automation Institute for the support in the use of the ROMEO system, where the proposed methodology has been tested.

The ARAMIS project (Advanced Rov Package for Automated Mobile Investigation of Sediments) was supported by the EC in the framework of the MAST III programmee (DGXII MAS3 CT970083).

References

[1] K. Valavanis and G. N. Saridis, *Intelligent Robotic Systems: Theory, Design and Applications.* Kluwer Academic Press, 1992.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *International Journal of Robotic research*, vol. 17, pp. 315–337, 1998.

[3] R. Fierro and L. Lewis, "A framework for hybrid control design," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 27, pp. 765–773, 1997.

[4] Byrnes, M. Nelson, S. Kwak, R. McGhee, and A. Healey, "Rational behaviour model: an implemented tri-level multilingual software architecture for control of autonomous underwater vehicles," in *8th International Symposium on Unmanned Untethered Submersible Technology*, pp. 160–178, 1993.

[5] E. Le Rest, L. Marcé, and V. Rigaud, "VORTEX-PILOT: a top-down approach for AUVs mission telerobotics language," in *OCEANS'94*, (Brest, France), pp. 102–107, September 1994.

[6] H. H. Wang, R. L. Marks, S. M. Rock, and M. J. Lee, "Task-based control architecture for an untethered unmanned submersible," in *8th International Symposium on Unmanned Untethered Submersible Technology*, pp. 137–148, September 1993.

[7] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–579, 1989.

[8] A. Caloini, G. Magnani, and M. Pezzè, "A technique for designing robotic control systems based on Petri nets," *IEEE Transaction on Control Systems Technology*, vol. 6, pp. 72–87, January 1998.

[9] T. Cao and A. C. Sanderson, "Task decomposition and analysis of robotic task plans using Petri nets," *IEEE Transaction on Industrial Electronics*, vol. 41, pp. 620–630, December 1994.

[10] T. Cao and A. C. Sanderson, "AND/OR net representation for robotic task sequence planning," *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, vol. 28, pp. 204–218, May 1998.

[11] F. Wang, K. Kyriakopoulos, A. Tsolkas, and G. Saridis, "A Petri-net coordination model for an intelligent mobile robot," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, pp. 777–789, July/August 1991.

[12] L. Montano, F. García, and J. Villarroel, "Using the Petri net formalism for specification, validation, and code generation in robot-control applications," *The International Journal of Robotic research*, vol. 19, pp. 59–76, January 2000.

[13] R. Bono, G. Bruzzone, M. Caccia, P. Coletta, and G. Veruggio, "Execution control of the NGC tasks for ROVs," in *ICRA 2001 - International Conference on Robotics and Automation*, May 2001.

[14] G. Bruzzone, M. Caccia, P. Coletta, and G. Veruggio, "Petri net-based execution control of robotic tasks," in *MCCA 2001 - Mediterranean Conference on Control and Automation*, June 2001.

[15] A. Girard, S. Smith, and K. Ganesan, "A convenient format for discrete event control of autonomous underwater vehicles," in *IARP-1st Internaltional Workshop on Autonomous Underwater Vehicles for Shallow Water and Coastal Environment*, pp. 125–130, 1998.

[16] P. Oliveira, A. Pascoal, V. Silva, and C. Silvestre, "The mission control system of MARIUS AUV: System design, implementation, and tests at sea," *Int. J. on Sys. Science - Special Issue on Underwater Robotics*, vol. 29, pp. 1065–1080, 1998.

[17] S. Bagchi, G. Biswas, and K. Kawamura, "Task planning under uncertainty using a spreading activation network," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 30, pp. 639–650, November 2000.

[18] M. Caccia and G. Veruggio, "Guidance and control of a reconfigurable unmanned underwater vehicle," *Control Engineering Practice*, vol. 8, pp. 21–27, January 2000.

[19] J. Moody and P. Antsaklis, "Petri net supervisors for DES with uncontrollable and unobservable transitions," *IEEE Transactions on Automatic Control*, vol. 45, pp. 462–476, March 2000.

[20] K. Yamalidou, J. Moody, J. Lemmon, and P. Antsaklis, "Feedback control of Petri nets based on place invariants," *Automatica*, vol. 32, pp. 15–28, January 1996.

[21] M. Caccia, R. Bono, G. Bruzzone, and G. Veruggio, "Unmanned underwater vehicles for scientific applications and robotics research: the Romeo project," *Marine Technology Society Journal*, vol. 34, pp. 3–17, 2000.