The 2009 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 11-15, 2009 St. Louis, USA

Using Petri nets to specify and execute missions for Autonomous Underwater Vehicles

Narcís Palomeras, Pere Ridao, Marc Carreras
University of Girona, Edifici Politecnica IV,
Campus Montilivi 17071 Girona, Spain
Carlos Silvestre
Institute for Systems and Robotics.
Instituto Superior Tecnico Lisbon, Portugal

***Abstract*— This paper presents the design and implementation of a Mission Control System (MCS) for an Autonomous Underwater Vehicle (AUV) based on Petri nets. In the proposed approach the Petri nets are used to specify as well as to execute the desired autonomous vehicle mission. The mission is easily described using an imperative programming language called Mission Control Language (MCL) that formally describes the mission execution thread. A Mission Control Language Compiler (MCL-C) able to automatically translate the MCL into a Petri net is described and a real-time Petri net player that allows to execute the resulting Petri net onboard an AUV are also presented.**

I. INTRODUCTION

There are several potential AUV applications being exploited by various organisations around the world: environmental monitoring, oceanographic research and maintenance/monitoring of underwater structures are just a few examples. AUVs are attractive for the use in these areas because of their size and their non-reliance on human operators. However, with the utilisation of AUVs and Intervention-Autonomous Underwater Vehicles (I-AUV) appears a new necessity: How to define a mission for these autonomous vehicles? A new set of tools is required to allow scientific and industrial operators to describe a mission, upload it into the vehicle and execute it in real-time. This set of tools is called Mission Control System (MCS). A MCS is the part of a control architecture that is in charge of coordinating the high-level phases to be carried out by the vehicle in order to fulfil a predefined mission. Each high-level phase is denominated as a *task* which can be executed by means of activating some vehicle *primitives* (basic robot commands or behaviours). The MCS must define how the mission should be divided into a set of tasks and how primitives are combined to fulfil each task.

The development of a MCS for an AUV lies at the intersection between a Discrete Event System (DES) responsible for enabling and disabling basic primitives when some events are produced and the Continuous State Dynamic Control System (DCS) used for every primitive to achieve a specific goal. The DES must ensure the consistency in the resulting controller avoiding to drive the vehicle into a dead-lock situation and simultaneously ensuring the reachability of one of the final states described in the mission. This is the main reason for choosing the Petri net formalism as the DES representation to model, program and execute AUV Missions.

One of the first works in this area using Petri nets was the coordination model for mobile robots proposed in [1] where some Petri Net Transducers (PNT) were used to translate the commands generated for the organisation level into something understandable for the execution level. In marine robotics, the researchers of the IST in Lisbon developed a MCS called Coral to be used in the MARIUS AUV [2]. The system was based on Petri nets in charge of activating the vehicle primitives needed to carry out the mission. The French AUV Redermor [3], designed for military applications of inspection and mine recovery, also uses Petri nets for modelling the Behaviours and a Lisp interpreter is employed to execute them in real time. In the CNR-ISSIA, Italy, a system based on Petri nets was designed to control an underwater robot [4]. In other fields like the RoboCup, Petri nets have been used by several teams [5] [6] for coordination and control purposes.

Following on the previously related works, a generic MCS for AUVs based on Petri nets was proposed in [7]. The generalities of this MCS are reviewed in Section II. Because the direct manipulation and construction of the Petri nets used in the MCS rapidly becomes cumbersome for complicated missions, a high level language called Mission Control Language (MCL) able to automatically compile into a Petri net is presented in Section III. The compiler developed to translate a MCL Program into a Petri net is described in Section IV and the software in charge of executing the resulting Petri net in real-time is introduced in Section V before the conclusions.

II. THE MISSION CONTROL SYSTEM

As mentioned above, the MCS of a vehicle deals with the mission definition as well as with its execution. In this paper, a new proposal for a MCS based on the Petri net formalism is presented. Instead of using graphic tools to describe the mission Petri net, our approach uses a Mission Control Language (MCL) which automatically compiles into a Petri net. The adoption of this formalism will allow us to

This research was sponsored by FREEsubNET (MRTN-CT-2006-036186) and the Spanish government (DPI2008-06548-C03-03).

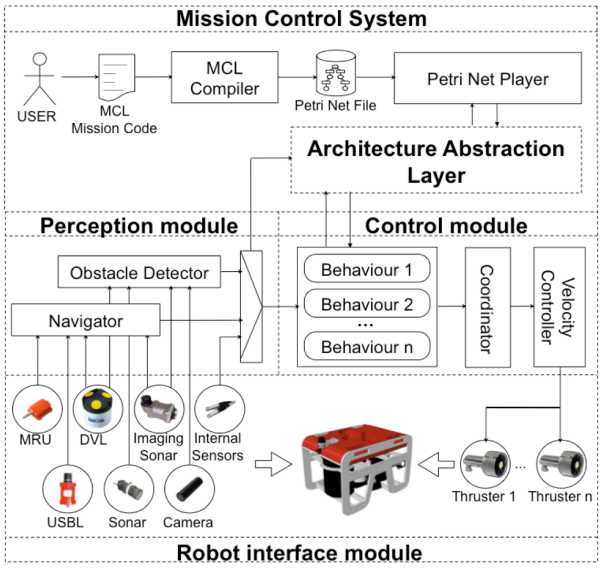

Fig. 1. Complete architecture for the Ictineu AUV.

construct a reliable mission control Petri net by joining small Petri nets, used as Petri net Building Blocks, equipped with the required properties to ensure that the whole mission Petri net accomplishes the set of desired properties.

A. Architecture Abstraction Layer

Our intention has been to design a MCS as generic as possible and to allow for an easy adaptation to different control architectures. To achieve this goal an Architecture Abstraction Layer (AAL) is used (see Fig. 1). The AAL is in charge of the communications between the MCS and the vehicle architecture making it architecture-independent. It offers an interface based on two types of signals: *Actions* and *Events*. *Actions* enable or disable basic primitives within the vehicle architecture. *Events* are used for the vehicle architecture to notify changes in the state of its primitives. The AAL depends on the control architecture being used allowing the MCS to remain architecture-independent. With the AAL, it is possible to use this MCS approach in different vehicles with different architectures. This is achieved by defining the basic actions that can be executed by the vehicle and the events that can be transmitted back to the MCS. Then the AAL must be reprogrammed with the mapping between the MCS messages and the vehicle primitives.

B. Primitives

Primitives are basic robot functionalities offered by the vehicle control architecture. A discussion of the basic functionalities for an AUV can be found in [2]. For an AUV a primitive can range from a basic sensor enabling (*enableCompass*) to a complex behaviour activation (e.g. navigate towards a 3D way-point: *goToWayPoint*).

Primitives have a goal to be achieved. For instance, the goal of the *KeepDepth* primitive is to drive the robot at a constant depth within an uncertainty interval. In general, a primitive can be enabled or disabled sending an action from the MCS through the AAL. Depending if the primitive is able to achieve its goal or not, if a failure is detected, etc. different events can be generated by the primitive being submitted through the AAL to the MCS.

C. Petri net Building Blocks

Petri net Building Blocks are the basic building structures of our MCS. These Petri net Building Blocks are used to program a mission as well as to execute it. They are defined using the Petri net formalism and must ensure some properties:

1) Petri net Building Blocks must be free of deadlocks.
2) The reachability graph of every Petri net Building Block must show that the set of places marked when all the possible transitions have already been fired after starting in the rest-state with a valid input marking are the places marked in the rest-state plus a valid combination of output places.
3) All the structures used in a mission must share the same interface, that is, they must have the same input and output places.

To achieve the first property, several techniques for deadlock avoidance using Petri nets can be found in the literature (see [8]). The second property ensures the reachability of the desired states when the Petri net Building Blocks are executed and progress from the begin state towards the end states. This property can be tested through a reachability analysis. It is well known the state explosion problem when performing a reachability analysis, however, at this level, Petri net Building Blocks are kept small enough and hence the reachability analysis is not a problem.

The third property introduces the concept of *interface*. A Petri net interface is composed by a set of input places P_i where $\forall_{p_k \in P_i} \bullet p_k = \emptyset$ and a set of output places P_o where $\forall_{p_k \in P_o}\ p_k\bullet = \emptyset$ in which $P_i \cup P_o$ are used as a *fusion places* to build more complex structures using different Petri net Building Blocks. It is possible to design any kind of interface but all the structures used to define a mission must share the same interface. Fig. 2(a) shows an example of a task with an interface composed by $P_i = \{begin,\ abort\}$ and $P_o = \{ok,\ fail\}$.

There are two different types of Petri net Building Blocks: *Tasks* and *Control Structures*. Both of them accomplish the above presented properties, however, their characteristics and usefulness differs as detailed hereafter.

D. Tasks

Tasks are Petri net Building Blocks that communicate with the robot architecture by means of *Actions* and *Events*. Actions are associated to transitions in the Petri net being executed whenever the related transition is fired. Events communicate changes detected by the vehicle architecture to the mission Petri net. Every event is associated to a particular place. If an specific event is triggered by a primitive in the control architecture, its related place receives a token.

Missions are programmed commanding the control flow of tasks. Task execution triggers the execution of the primitives

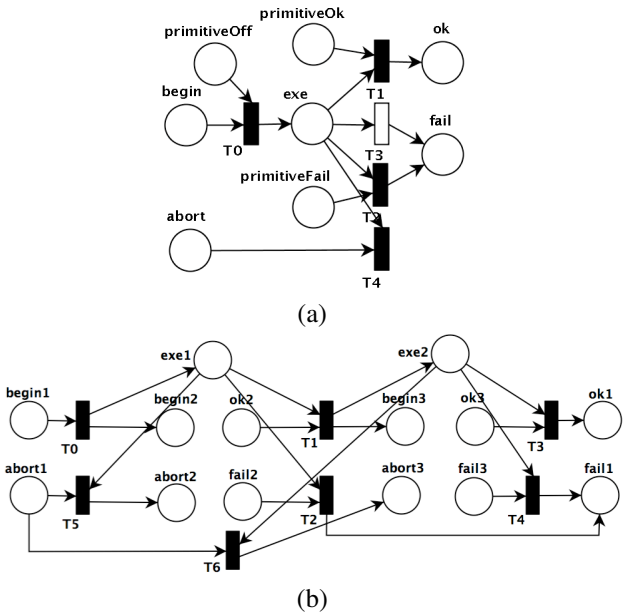

Fig. 2. (a) Simple *Task* with time-out. (b) Sequence control-structure.

TABLE I

DESCRIPTION OF ELEMENT IN FIG. 2

Task	
T_0	Send action *Enable primitive*.
$T_1 ... T_4$	Send action *Disable primitive*.
T_3	Timed transition. Fired after a time-out.
P_{Ok}, P_{Fail}	Event place. Receives a token when the associated primitive changes its state to Ok/$Fail$.
P_{Off}	Event place. Receives a token when the associated primitive changes its state to off.
Control Structure	
$begin_2$, $abort_2$, ok_2, $fail_2$	Fusion places (*interface*) to the first structure in the sequence.
$begin_3$, $abort_3$, ok_3, $fail_3$	Fusion places (*interface*) to the second structure in the sequence.

as well as the detection of the corresponding events. Fig. 2(a) shows a task able to launch a primitive and to stop it when a *primitiveOk* or a *primitiveFail* event is raised or the T_3 time-out expires (see Table. I for a description about some places and transitions in Fig. 2).

E. Control Structures

The Petri net Building Blocks used to aggregate other Petri net Building Blocks with the objective of modeling more complex actions by controlling the flow among different execution paths are called *control structures*. The resulting net is a new Petri net Building Block that satisfies the desired Petri net properties, as inherited from the original Petri net Building Blocks, this operation will be denominated by *composition*. It is worth noting that it is not necessary to span the whole reachability tree of the resulting Petri net to ensure the deadlock free as well as the state reachability properties. Spanning it, would have a very high computational cost as the complexity of the Petri Net that results from the composition operation can be very large. These properties

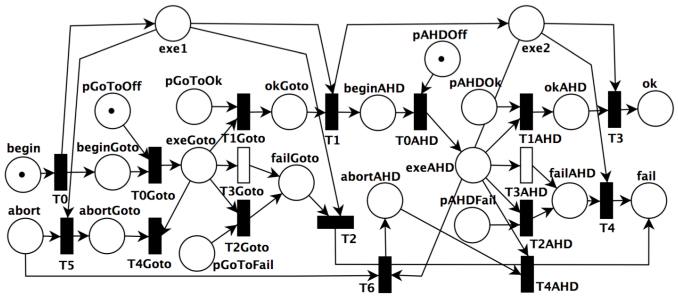

Fig. 3. Composition of the tasks *Goto()* and *AchieveHeading()* with the control structure SEQ.

are guaranteed by construction and hence a mission program implemented according to these rules progresses from its starting state to an exit state without sticking into a deadlock. It can be proved that this set of Petri net Building Blocks is closed with respect to the composition operation. Hence, the result of a composition of some Petri net Building Blocks with the above presented properties is a new Petri net Building Block, for which the same properties hold by construction without the need of further verification.

If two tasks called *GoTo()* and *AchieveHeading()*, each one described by the Petri net reported in Fig. 2(a), are composed by the *sequence* control structure shown in Fig. 2(b), the resulting Petri net will be as the one presented in Fig. 3.

Depending on the selected interfaces, different control structures can be defined. Based on the above presented interface $P_i = \{begin,\ abort\}$ and $P_o = \{ok,\ fail\}$, MCL provides some popular control structures while others may be defined by the programmer, if needed.

- **Sequence:** It is used to execute one Building Block after another (see Fig. 2(b)).
- **Try-Catch-Do:** Executes the *Try* block in parallel with the *Catch* block. If the former finishes before the later, the *Catch* block is cancelled and the execution continues after the *Try-Catch-Do* structure. However, if the *Catch* block finishes first, the *Try* block is aborted and the *Do* block is executed.
- **Parallel-And:** Executes two Building Blocks in parallel. If both Building Blocks finish in an *ok* place the whole control structure finishes with an *ok*. Otherwise, the *Parallel-And* finishes with a *fail*.
- **Parallel-Or:** Executes two Building Blocks in parallel. The first structure to finish aborts the other. The *Parallel-Or* finishes with the final state of the first Building Block to end.
- **If-Then-Else:** Executes the Building Block inside the *If* statement and depending if the Building Block ends with an *ok* or a *fail* the Building Block inside the *Then* statement or the *Else* statement is executed respectively.
- **While-do:** Executes the Building Block inside the *While* statement. If this Building Block finishes with an *ok* executes the *do* statement, otherwise ends with an *ok*. If the *do* statement finishes with an *ok* executes again the *while* statement otherwise ends the whole structure

with a *fail*.

III. The Mission Control Language

In this section, the structure of a MCL program is presented.

A. Actions and Events

The primitive behaviours as well as the events must be specified in the MCL program. Actions are defined using the $action_{id} = Primitive(list\_of\_parameters)$ command where the parameters can be literals ('*l*') or variables ('*st*') that must be instantiated during the mission definition. Events are defined by the command $place\_id(initial\_number\_of\_tokens) = event\_id$. When the event $event\_id$ is generated in the robot control architecture, the place $place\_id$ of the Petri net mission receives a token. Example 1 defines the necessary actions and events used in the task *Goto()*.

It is worth noting that the way in which actions and events maps are implemented depend on the control architecture of the robot. Hence, the definition of this map is used to tailor MCL to a particular control architecture. Therefore different MCL mission programs for the same vehicle will share the same version of these maps.

Example 1 Actions & Events Definition

```
actions {
    enableGoto = GotoPrimitive(l: enable, st: waypoint);
    disableGoto = GotoPrimitive(l: disable);
    ...
}
events {
    pGotoOff (1) = eventGotoDisabled;
    pGotoOk(0) = eventGotoOk;
    pGotoFail(0) = eventGotoFail;
    ...
}
```

B. Building Block Patterns

To define a new Petri net Building Block, it is necessary to specify its internal structure called Building Block Pattern. This is done by defining a set of places (P) with the $place_{id}(number\_of\_tokens).connection_{id}$ command, a set of transitions (T) using the $transition_{id}$ command and a set of arcs (A) defined using the $source_{id} \rightarrow destination_{id}$ command. It is necessary to indicate the *connection identifier* for every place. This connection identifier is used to distinguish which places belong to the interface ($connection\_id = 1$) and which of them do not ($connection\_id = 0$). Moreover, the connection identifier is used in the control structures to specify how Petri net Building Blocks are connected among them. For instance, the *sequence* control structure shown in Fig. 2(b) which sequences two Building Blocks must have internal fusion places (with $connection\_id = 2$ and 3) to connect with the interface of the two structures to be sequenced. Both the initial number of tokens of a place and the connection identifier are set to 0 by default.

It is also possible to use any available Petri net editor[1] with Petri Net Mark-up Language (PNML)[2] support to define a Building Block Pattern and include the file in the MCL program using the command $pattern_{id} = pnml\_file$;.

The Building Block Pattern for the task presented in Fig. 2(a) is described using MCL notation in Example 2.

Example 2 Building Block Pattern Definition

```
AchieveOneGoal {
    places {
        begin.1; abort.1; ok.1; fail.1; exe;
        primitiveOff; primitiveOk; primitiveFail;
    }
    transitions {
        T0; T1; T2; T3; T4;
    }
    arcs {
        begin.1 → T0; primitiveOffT0; T0 → exe;
        primitiveOk → T1; exe → T1; exe → T2; exe → T3;
        exe → T4; primitiveFail → T3; abort.1 → T4;
        T1 → ok.1; T2 → fail.1; T3 → fail.1;
    }
}
```

C. Tasks

To define a task a Building Block Pattern must be instantiated. Once the Building Block Pattern is chosen the related actions, events and time-outs must be associated to the corresponding places and transitions. The task header is composed by a list of parameters corresponding to those used in the actions linked to the transitions. For instance, if a task has to execute the primitive *enableGoto* it must include in its header the parameter *waypoint* used in this action.

Example 3 shows a task instantiated from the task pattern presented in Example 2. The task is used to guide a robot towards a way-point having an associated time-out.

Example 3 Task Definition

```
AchieveGoto(waypoint): AchieveOneGoal {
    a: enableGoto → T0;
    a: disableGoto → T1, T2, T3, T4;
    t: 120 → T3;
    e: pGotoOff → primitiveOff;
    e: pGotoOk → primitiveOk;
    e: pGotoFail → primitiveFail;
}
```

D. Control Structures

The following control structures are provided in the MCL: **sequence (;)**, **parallel-and**, **parallel-or**, **if-then-else**, **try-catch-do** and **while-do**. Each one of these control structures can be overloaded defining a new pattern with the same name. When a control structure is used to aggregate two structures (**sequence**, **parallel** and **while**) two sets of interfaces with connection identifiers 2 and 3 must be provided. If a control structure aggregates three control structures (**if-then-else** or **try-cath-do**) three sets of interfaces with connection identifiers 2, 3 and 4 must be provided instead.

E. Mission Program

Once the tasks and the control structures have been defined a mission can be coded in MCL. This is the only section that must be rewritten for every new mission if the same vehicle control architecture is used. MCL control structures are very similar to those provided by other popular languages and tasks can be seen as function calls. Hence, programming a new mission using MCL becomes very simple.

[1]For instance the *pipe2* editor available on: http://pipe2.sourceforge.net
[2]http://www.pnml.org

Example 4 describes a test mission to perform a survey path while gathering images. The robot uses an acoustic modem to check the availability of USBL position fixes through the modem. The program switches between AUTONOMOUS_USBL or AUTONOMOUS mode depending on their availability. When not available, navigation is done by FOG-DVL dead reckoning. Water leakage as well as the battery level are being checked during all the survey mission.

Example 4 MCL code to define a survey Mission

```
mission {
  try {
    Devices(ON);
    Mode(AUTONOMOUS_USBL);
    Logs(ON);
    parallel {
      while (True()) {
        if(HeartBeat(TIMEOUT)) {
          Mode(AUTONOMOUS_USBL)
        } else {
          Mode(AUTONOMOUS)
        }
      }
    } or {
      AchieveAltitude(SafeAltitude);
      parallel {
        KeepAltitude(SafeAltitude)
      } or {
        ObstacleAvoidance(true)
      } or {
        Goto(START_WAYPOINT);
        Ligths(ON);
        Camera(DOWNLOOKING,ON,FREQUENCY);
        Holonomic2DPath(CtHeading, GridPath, TIMEOUT);
        Ligths(OFF);
        Camera(DOWNLOOKING, OFF, FREQUENCY);
        Goto(END_WAYPOINT)
      };
      Surface();
      Logs(OFF);
      Devices(OFF)
    }
  } catch {
    parallel {
      Timeout(MISSION_TIMEOUT)
    } or {
      LowBattery(MinBatteryLevel)
    } or {
      WaterLeakage()
    }
  } do {
    AbortMission()
  }
}
```

IV. The Mission Control Language - Compiler

The process to generate a Petri net from a MCL program is divided in four steps: (1) Generate a generic Petri net for every pattern; (2) Instantiate the Petri nets of the task patterns adding the actions, the events and the time-outs to build all the Petri net tasks; (3) Use the mission program to generate a tree in which the nodes are the control structures and the leaves are the tasks; (4) Traverse the tree aggregating first the control structures among them and then the tasks to build the whole Petri net.

The procedures involved in the implementation of steps (1) and (2) are trivial. While the code is being parsed by the compiler a Petri net is generated for every task. The compiler only checks if the patterns are correctly connected with valid places and transitions or vice-versa and that all the '*st*' variables used by the actions in a task appear also in the task header.

Id	Task
0	True
1	HeartBeat
2	Mode

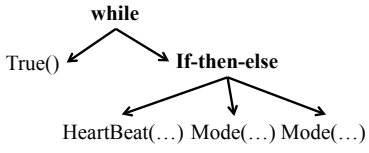

Fig. 4. Fragment of Example 4 processed as an AST.

Step (3) can be seen as the translation of the imperative code written by the user into a functional form. This process consists of building an Abstract Syntax Tree (AST). Fig. 4 shows how lines 7 to 13 from Example 4 are translated to its corresponding AST. Finally, in step (4) the recursive Algorithm 1 is applied. The AST generated in step (3) and a vector relating the referenced tasks with their $connection_{id}$ are used as input parameters (see Fig. 4). The algorithm explores the tree and composes all the control structures recursively. If a task is found during this process it is renamed with a valid connection identifier and their parameters are added to its enabling control structure. When the whole tree is explored, each Petri net task is included in the mission. Note that Petri net tasks are not replicated, instead, only one Petri net task structure is included in the whole Petri net mission even though it can be called several times from different control structures. Moreover, all the places that reference the same event are joined to keep only one place for every primitive event.

Algorithm 1 GeneratePN(AST t, vector<task> vt) **ret** PN

```
if t.root = "mission" then
  //If the tree root tag is 'mission' call GeneratePN(..) for its first child
  pn = GeneratePN(t.child[0], vt)
  //Compose every task with the rest of the control structures
  for i = 0 to size(vt) do
    pn = Compose(vt[i], pn)
  end for
  for i = 0 to size(events) do
    pn = Compose(event[i], pn)
  end for
else
  //If the root is not 'mission', load the control structure indicated in it
  pn = Load(ControlStructure[t.root])
  //For every child, do:
  for i = 0 to size(t.child) do
    if t.child[i].type = "task" then
      //If it is a task change the connection id. for the id. of this task
      ChangeConnectionsId(i, vt, pn)
      //Put the parameters of this task in the pn transition that enables it
      TakeParameters(i, vt, pn)
    else
      //If the child is another control structure call GenaratePN(..) for this child
      pnTmp = GeneratePN(t.child[i], vt)
      //Compose both PN
      pn = Compose(pnTmp, pn)
    end if
  end for
end if
//All places with the same event reference are groupped
GroupEvents(pn)
return pn
```

The $Compose()$ function is used to aggregate several Petri nets using their fusion places. It joins the places, transitions and arcs from the two nets and, while fusion places are shared combining the arcs and adding the tokens. Fig. 3 shows the aggregation of two tasks within a *sequence* control

structure.

V. The Real-Time Petri Net Player

The code generated by the MCL-C represents a single Petri net nammed Mission Program, described using an XML-based interchange format for Petri nets called Petri Net Markup Language (PNML). A particular extension had to be introduced in the PNML in order to properly implement the communication facilities offered by the AAL in the language, namely to define the *actions* that are sent from the Petri net player to the vehicle control architecture and the *events* generated there to be fed back. The Petri net player implements in real-time the discrete event system described by the PNML file applying the basic Petri net transition rule to execute the whole Mission Program Petri net. To do this, it has to control all the timers associated to timed transitions, fire enabled transitions non-deterministically, send the vehicle primitive *actions* to the AAL and transform the *events* received from the AAL to marked places within the Mission Program. This object implemented in C++ uses the Transmission Control Protocol (TCP/IP) to communicate with the AAL and is executed in real-time inside the AUV control architecture.

As shown in Figure 1, the program describing the mission is written by the user using the MCL. The MCL-C transforms this code into a Petri net described using PNML and the real-time Petri net player executes this Mission Program communicating with the vehicle control architecture through the AAL. The rest of the picture shows the Ictineu AUV architecture [9] with a set of behaviours that act as vehicle primitives and some perception modules which turn the raw data gathered from the sensors into estimated variables to be used by the robot behaviours.

As an example of how the Petri net player executes a mission let us consider the steps performed to execute the Petri net in Fig. 3. First, the $begin^{seq}$ place (the superscript refers to the Petri net that the corresponding place or transition belongs to) is marked with a token. Then transition T_0^{seq} fires marking the places $begin^{goto}$ and exe_1^{seq}. Because the **GoTo** primitive was formerly deactivated (the $primitive_{off}^{GoTo}$ place was initially marked) the transition T_0^{goto} fires, launching the *enablePrimitiveGoTo* action through the AAL. This action starts the **GoTo** control behaviour within the control architecture which is responsible for the robot guidance. Moreover, the exe^{goto} place becomes marked reflecting the fact that the **GoTo** control behaviour is currently running. When the vehicle arrives to its destination, the place $primitive_{ok}^{goto}$ receives a token and the *GoTo()* task finalises firing the transition T_1^{goto}. This firing launches the *disablePrimitiveGoTo* action that will disable the **GoTo** primitive restoring the marking of the $primitive_{off}^{Goto}$ place. Simultaneously, the place ok^{goto} is marked. If the vehicle is unable to reach the desired waypoint or the time-out controlled by the timed transition T_3^{goto} fires, the same *disablePrimitiveGoTo* action will be sent but the $fail^{goto}$ place, will get marked instead. If the *Goto() task* finalises marking ok^{goto}, *task AchieveHeading()* is executed in the same way as *Goto()*. However, if the *Goto()* task finalises marking $fail^{goto}$, then the place $fail^{seq}$ is marked finalising the execution of the whole control structure.

VI. Conclusions and future work

This paper presented the results of an ongoing research project which aims at designing and implementing a flexible MCS easy to be tailored for different control architectures. Following a brief introduction on the need of using AUVs and the typical functions required to a MCS some relevant MCSs based on Petri nets are commented. After this introduction, the proposed MCS and the MCL are presented. In the approach presented in the paper, Petri nets are used to safely model the tasks and the control structures. All these Petri net Building Blocks have been designed free of deadlocks and reusable. It has been shown that it is possible to compose Petri net Building Blocks to generate the whole mission. To describe the mission, instead of directly manipulate the Petri nets using graphical tools, a high level imperative language, called MCL, which compiles into a Petri net is used. MCL presents agreeable properties of simplicity and structured programming and offers the means for sequential/parallel, conditional and iterative task execution. The MCL-C is the tool in charge of translating the MCL language into a Petri net that can be automatically executed by the vehicle using a real-time Petri net player.

The MCL-C has been completely programmed resorting to ANTLR[3] and a preliminary version can be download from http://eia.udg.edu/∼npalomer. All tests have been done using the Ictineu AUV [9] and the e-puck robot using the webots simulator[4]. Future work will involve adding multiple vehicle coordination capabilities in the MCS/MCL allowing the management of missions involving several robots.

References

[1] F. Wang, K. Kyriakopoulos, A. Tsolkas, and G. Saridis, *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 21, no. 4, pp. 777 – 789, Jul 1991.

[2] P. Oliveira, A. Pascoal, V. Silva, and C. Silvestre, "Mission control of the marius auv: System design, implementation, and sea trials," *International Journal of Systems Science*, Jan 1998.

[3] M. Barbier and J. Lemaire, "Procedures planner for an a.u.v." *12th international symposium on Unnmanned Untethered Submersible Technology*, p. 8, 2001.

[4] M. Caccia, P. Coletta, G. Bruzzone, and G. Veruggio, "Execution control of robotic tasks: a petri net-based approach," *Control Engineering Practice*, vol. 13, no. 8, pp. 959–971, 2005.

[5] V. Ziparo and L. Iocchi, "Petri net plans," *Proc. of ATPN/ACSD Fourth International Workshop on Modelling of Objects, Components, and Agents*, 2006.

[6] H. Costelha and P. Lima, "Modelling, analysis and execution of multi-robot tasks using petri nets," *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, Jan 2008.

[7] N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre, "Towards a mission control language for auvs," *17th IFAC World Congress*, 2008.

[8] M. V. Iordache, J. O. Moody, and P. J. Antsaklis, "Automated synthesis of deadlock prevention supervisors using petri nets," *Technical Report of the ISIS Group at the University of Notre Dame*, p. 56, Jul 2002.

[9] D. Ribas, N. Palomeras, P. Ridao, M. Carreras, and E. Hernandez, "Ictineuauv wins the first sauc-e competition," *Robotics and Automation, 2007 IEEE International Conference on*, pp. 151 – 156, Mar 2007.

[3] http://www.antlr.org

[4] http://www.cyberbotics.com