Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Towards a Mission Control Language for AUVs ⋆

Narcís Palomeras * **Pere Ridao** * **Marc Carreras** *
Carlos Silvestre **

* *University of Girona. Edifici Politecnica IV, Campus Montilivi Girona, Spain*
npalomer@eia.udg.es
** *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 AUV. The mission is easily described using an imperative-like pseudo-code called Mission Control Language (MCL) that allows sequential/parallel, conditional and iterative task execution. MCL can be automatically translated into a Petri net, to formally describe the mission thread of execution. Then the MCS executes the Petri net in real-time over a generic layer that communicates with a particular control architecture using predefined actions and events. Concepts are illustrated with a simple mission.

1. INTRODUCTION

A mission controller is the part of a control architecture that is in charge of defining high-level phases to be carried out in order to fulfil a predefined mission. Each high-level phase is a *task* that can execute a *vehicle primitive* (basic robot commands or behaviours). The mission controller must define how the mission is divided into a set of tasks and how primitives are *called* to fulfil each task. As described in Marco et al. [1996], the development of a Mission Control System (MCS) for an Autonomous Underwater Vehicle (AUV) lies at the intersection of a Discrete Event System (DES) in charge of enabling/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.

Several mission control systems for AUVs have been designed over the past decade. In 1994 the Institute for Systems and Robotics (ISR), from Portugal, started the development of a mission management system called Coral for the AUV MARIUS, Oliveira et al. [1998]. The system was based on Petri nets in charge of activating the vehicle primitives needed to carry out the mission. Simultaneously, the Naval Postgraduate School (NPS) from Monterey was developing a hybrid control system composed of three layers using the Prolog language as a rule-based mission specification in the higher layer, Marco et al. [1996]. Another control architecture, which has a mission control system called Helm, is the Mission Oriented Operating Suite (MOOS) designed between the Massachusetts Institute of Technology and the Oxford University by Paul Michael Newman see Newman [2005]. Helm decides the most suitable action commands from a set of prioritised mission goals. The Sauvim Task Description Language (STDL) developed by the University of Hawaii, see Kim and Yuh [2003], instead of using a graph uses a descriptive imperative language. Other MCS proposed in the literature are the AUV Scripting Language (ASL) used by the commercial AUV Gavia and the also scripting languages [1] used by the Autosub AUV, developed by the National Oceanography Center of Southampton, Perrett and Pebody [1997], and the Remus AUV, originally designed by the Woods Hole Oceanographic Institution (WHOI) and now manufactured and sold by Hydroid, Allen et al. [1997].

Each MCS approach is very particular and dependent on the particular application it was designed for. Every institution, research group or company, has developed its particular MCS based on the missions they want to achieve, the kind of AUVs they have and the control architecture used in these AUVs. However, several similarities can be found. For example, all studied AUVs have a set of basic primitives that can be called from the MCS. These primitives are often named differently in each system: commands, vehicle primitives, behaviours, etc. They are generally DCS in charge of achieving a simple goal (keep a specific orientation or depth, achieve a way-point, etc.) or used to enable/disable sensors, loggers, to take an image, etc. There are also similarities in the execution formalism in charge of describing the DES used to activate/desactivate the vehicle primitives depending on the produced events. Even though most of them use different formalisms, they can be easily related. For example, MOOS uses a state machine which is a particular class of a Petri net, the basic formalism used by Coral. Sauvim and Gavia use an imperative language (STDL and ASL respectively) and, as will be explain in Section 2, one of

⋆ This research was sponsored by the Spanish government (DPI2005-09001-C03-01), FREEsubNET (MRTN-CT-2006-036186) and Fundação para a Ciência e a Tecnologia (ISR/IST plurianual funding) through the POS_Conhecimento Program that includes FEDER funds. Thanks to J. Suy to help us to formally define the MCL.

[1] Scripting languages are often distinguished from typical programming languages because are typically interpreted.

978-1-1234-7890-2/08/$20.00 © 2008 IFAC

10.3182/20080706-5-KR-1001.2209

the goals of this paper is show how is possible translate an imperative languages into a Petri net. NPS uses prolog rules (logic programming) to describe their mission. This particular use of the language can also be translated to the Petri net formalism as described in Marco et al. [1996]. Therefore, Petri nets are presented as a versatile choice for the execution formalism.

Section 2 describes the MCS detailing how primitives are called using tasks and how these tasks are defined and joined, through control structures, to setup missions. Section 3 test this MCS using the ICTINEU^{AUV} within the context of a simple mission defined using the proposed Mission Control Language (MCL). Finally, Section 4 and Section 5 present the results obtained in the mission as well as the conclusions and future work.

2. MISSION CONTROL SYSTEM

In this paper a new proposal for a MCS based on the Petri net formalism (see Murata [1989]) is presented. Instead of using graphic tools to describe the mission Petri net, our approach is based on the definition of an MCL which automatically compiles into a Petri net. The adoption of this formalism will allow us to construct a reliable mission control net joining small nets, previously evaluated, ensuring that the whole control net accomplish a set of required properties.

2.1 Architecture Abstraction Layer

Our intention has been to design a MCS as generic as possible and to allow for an easily tailoring to different control architectures. To achieve this goal an Architecture Abstraction Layer (AAL) is used. This layer is in charge of the communications between the MCS and the vehicle architecture making it architecture-independent. The AAL offers an interface based on two types of signals: *Actions* and *Events*. *Actions* enable or disable basic primitives within the vehicle architecture. *Events* are basically used to notify changes in the state of a primitive. The communication between the MCS and the vehicle architecture is done using a message exchange system through the AAL. This layer receives actions (A) from the MCS. Actions contain a primitive identifier (a_j) and a set of parameters (π_k). At the same time, the AAL receive events from the vehicle architecture notifying state changes in the vehicle primitives. These state changes are translated by the AAL as marked or unmarked places inside the MCS. The AAL depends on the control architecture being used allowing the MCS to remain architecture-independent. With the AAL, it is possible use this MCS approach in different vehicles with different architectures, it is only necessary to define the basic actions that can be executed by the vehicle architecture and the events that it can be transmitted to the MCS and reprogram the AAL with the mapping between the MCS messages and the vehicle primitives.

2.2 Primitives

Primitives are basic robot functionalists offered by the vehicle control architecture. A discussion of the functionalists for the MARIUS vehicle can be found in Oliveira et al. [1998]. For an AUV a basic primitive can be keep a certain depth (*KeepDepth*), or a certain heading (*KeepHeading*) or navigate towards a 3D waypoint (*GoToWayPoint*) for instance. All primitives have a goal to achieve or to keep. For instance, the goal of the *KeepDepth* primitive is to keep the robot at a constant depth within an uncertainty interval. The goal of the *GoToWayPoint* primitive is to drive the robot inside a particular uncertainty sphere of acceptance centered in the desired waypoint. In general, a primitive can be enabled (ON) or disabled (OFF). While it is enabled, a primitive has three main states: 1) seeking the goal, 2) goal has been achieved, or 3) goal has been lost. First, the primitive will start seeking the goal (1). Once the goal is achieved (2), eventually, the primitive can lose it reaching state (3). Its worth noting that the primitive can switch between states (2) and (3) depending on the system disturbances. Finally, a primitive can also be disabled.

Attending to these common features, a generic primitive can be modeled using a Petri net as shown in Fig. 1. This model can be used as a guided line to generate the primitive code, that runs on the vehicle architecture, ensuring that the primitive input-output behaviour satisfies the prespecified requirements and it can be safely executed by the supervisor. This is, once a primitive is enable it must evolve free of deadlocks until it is disabled. In terms of Petri net theory this means that it is necessary ensure that all the siphons [2] in the Petri net are controlled and, if we want made the net reusable, it has to exist a unique final state which coincides with the initial marking of the net. Places E (enable) and D (disable) receive a token when the MCS sends an enable/disable action while places *S1* (state 1: goal not achieved), *S2* (state 2: goal achieved) and the *OFF* (primitive off) correspond to the internal states of the primitive. These five places are the fusion places used to merge and control the vehicle primitive from a higher level control structure called *task* (see Section 2.3). Whenever one of these states change, an event is sent from the vehicle control architecture to the MCS in order to update the tokens in the corresponding places. It is worth noting the role of the C place which is initially marked. This place was added to ensure that the places *S1* (goal achieved) and *S2* (goal lost/unachieved) can not be simultaneously marked. Hence, C marked indicates that the primitive is enabled and seeking its goal but neither *S1* nor *S2* have been yet achieved. When the primitive is disabled we must ensure that *S1* and *S2* become unmarked and C recovers its initial marking state.

The desirable properties of the net, can be checked studying its invariants [3] and siphons. Hence, analysing the primitive model three place invariants are found:

$$ON + OFF = 1 \tag{1}$$

$$C + S1 + S2 = 1 \tag{2}$$

$$-OFF - D + E = -1 \tag{3}$$

There are also two siphons (4 and 5) which are controlled by the invariants 1 and 2 respectively.

$$Siphon_1 = \{ON,\ OFF\} \tag{4}$$

[2] Siphons are defined in Iordache and Antsaklis [2006] as a non empty set of places that accomplish $\bullet S \subseteq S\bullet$ where $\bullet S$ and $S\bullet$ are the pre-set and post-set of all input/output transitions in the set of places S.

[3] Place invariants are sets of places whose weighted token count remains constant for all possible markings.

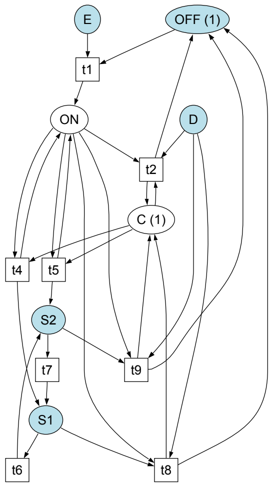

Fig. 1. Vehicle primitive Petri net model.

$$Siphon_2 = \{C,\ S1,\ S2\} \tag{5}$$

According to Iordache and Antsaklis [2006] if all the siphons in a Petri net are controlled by an invariant and they are correctly marked in M_0 (6) it is possible to ensure that they will not lose all their tokens preventing deadlocks.

$$M_0 = \{OFF,\ C\} \tag{6}$$

Considering an alternative intial state M_0=$\{OFF,\ C,\ E,\ D\}$ it is possible to check which is the cover-ability graph of this Petri net when it is enabled and disabled by a superior control structure. This study reveals a single final state Sf_1(7):

$$Sf_1 = \{OFF,\ C\} \tag{7}$$

Since this final state has the same marking than M_0 we can conclude that the primitive model is reusable, which means, that primitive implementation inside the vehicle can be reusable (e.g. the *KeepDepth* primitive can be run more than one time without re-initialisation).

2.3 Tasks

Tasks are the basic building block of the MCL. In MCL two main constructions are provided to deal with tasks: 1) tasks patterns and 2) the tasks. Task patterns are models from which particular tasks can be derived through an instantiation process. A task pattern can be defined to use a primitive to achieve a certain goal (f.i. *AchieveGoal*). Another task pattern can be defined to use a primitive to keep a certain goal (e.g. *KeepGoal*). It is also possible to use a task pattern involving the parallel execution of several primitives (*KeepTwoGoals*). Task patterns are defined in terms of generic primitives, which can then be instantiated to a particular primitive to generate a task (f.i. *AchieveGoal* $\rightarrow$ *AchieveWaypoint*, *KeepGoal* $\rightarrow$ *KeepDepth*). For the sake of simplicity let us assume the task *AchieveWayPoint* derived from the *AchieveGoal* task pattern through the instantiation of its generic primitive to the *GoToWayPoint* primitive. The task (*AchieveWayPoint*) begins enabling the primitive (*GoToWayPoint*) which will run until one of the following conditions holds: 1) the primitive achieves its goal (way-point reached), 2) the primitive realises it will not be able to achieve its goal (f.i. motion failure), 3) the task time-out expires before concluding the primitive and 4) the task is aborted. Conditions (1) and (2) are raised by the primitive, condition (3) is raised by the task itself and condition (4) is raised by a hierarchically superior control structure. The above-mentioned task can be modelled with the Petri net shown in Fig. 2. When a superior control structure marks the place *B* (begin) the vehicle primitive is enabled (place *E* marked) if it was previously disabled (place *OFF* marked). Once the primitive is enabled the task can be finalised if: 1) the time-out of the transition *t03* expires, 2) if a higher level control structure marks the place *A* aborting the task or 3) the primitive marks the place *S1* (unable to achieve goal) or *S2* (goal achieved). If the place *S1* is marked or the taks timeout expires the primitive is disabled (place *D* marked) and the place *FAIL* (task not achieved) is marked when the primitive is completely disabled (place *OFF* marked). When the place *S2* is the one marked, after disabling the primitive the marked place will be the place *OK* (task achieved). Finally, if the task is aborted (place *A* marked) it finalises with place *ABORTED* marked once the primitive is disabled. Places *E* and *D* are used to send actions from the MCS to the control architecture while places *OFF*, *S1* and *S2* are used to evaluate the state of the primitive. The tokens in these places, only change when an appropriate event is sent by the control architecture to the MCS. A mutex place called *MTX* and initially marked is also included in the net to ensure mutual execution of the task.

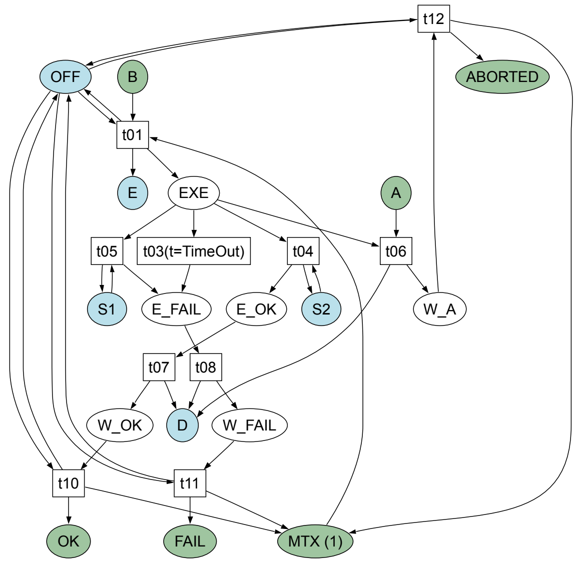

Fig. 2. Basic control task Petri net.

When the Petri nets of the task and the vehicle primitive model are composed using the fusion places $\{E,\ D,\ OFF,\ S1,\ S2\}$ (see Iordache and Antsaklis [2006] for Petri net composition operations) the resulting Petri has six place invariants:

$$ON + OFF = 1 \tag{8}$$
$$C + S1 + S2 = 1 \tag{9}$$
$$OFF + D - E + EXE + E\_OK + E\_FAIL = 1 \tag{10}$$
$$ABORTED + OK - MTX + FAIL + B = 0 \tag{11}$$
$$ABORTED + A - WA = 1 \tag{12}$$
$$-ABORTED - OFF + MTX - D + E - A + W\_OK + W\_FAIL = -1 \tag{13}$$

One more siphon is added to the ones presented in equation 4 and 5 and they still being controlled by invariants 8, 9 and 10+12+13.

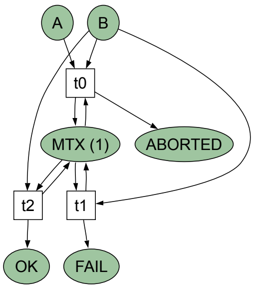

Fig. 3. Reduced Petri net model for a task.

$$Siphon_1 = \{ON, OFF\} \tag{14}$$

$$Siphon_2 = \{C, S1, S2\} \tag{15}$$

$$Siphon_3 = \{MTX, EXE, E\_OK, E\_FAIL, W\_OK, W\_FAIL, W\_A\} \tag{16}$$

Generating the coverability tree from the initial state $M_0 = \{B,\ A,\ MTX,\ OFF,\ C\}$ all places are 1-bounded and three final states, 17, 18 and 19, are obtained. All of them include the places C, OFF and MTX marked as well as one of the three final places: OK, $FAIL$ or $ABORTED$.

$$Sf_1 = \{A,\ FAIL,\ MTX,\ OFF,\ C\} \tag{17}$$

$$Sf_2 = \{A,\ OK,\ MTX,\ OFF,\ C\} \tag{18}$$

$$Sf_3 = \{ABORTED,\ MTX,\ OFF,\ C\} \tag{19}$$

This results show that it is possible for an aborted task to end in a goal achieved (OK) or a goal not achieved ($FAIL$) state instead of the intuitive $ABORTED$ state. This happens when the primitive ends before the requested abort command has been executed. If the initial state is $M_0 = \{B,\ MTX,\ OFF,\ C\}$ the final states reached are the two expected states (20, 21) indicating that the mutex is free (MTX), the primitive is disabled (OFF), C has its initial marking and $FAIL$ or OK are marked exclusively depending on the success of the primitive. Like in the previous case, it is possible to reuse the task Petri net. However, attention must be paid to the abort place A since it can remain marked after the task execution (this problem will be further referred as the abort problem).

$$Sf_1 = \{FAIL,\ MTX,\ OFF,\ C\} \tag{20}$$

$$Sf_2 = \{OK,\ MTX,\ OFF,\ C\} \tag{21}$$

Another interesting analysis involves studying what happens if two or more task structures try to enable the same primitive simultaneously. In this situation the place OFF in the primitive net acts as a *mutex* avoiding the second task to fire transition *t01* until the primitive is disabled again.

Fig. 2 has shown a Petri net able to model the internal states of a task. Nevertheless, from the external point of view, it is interesting to find a reduced model which starting in the same initial state reaches exactly the same final states (see Oliveira [2003]). Fig. 3 shows this reduced model for a control task. Reachable states from $M_0 = \{B,\ A,\ MTX\}$ are presented in equations 22, 23 and 24 and final states obtained from $M_0 = \{B,\ MTX\}$ are shown in equations 25 and 26. With this simplified Petri net model it is possible to reach the same final states from a supervisor point of view than the ones presented in 17 to 21. This reduced model will be of interest in Section 2.5.

$$Sf_1 = \{A,\ FAIL,\ MTX\} \tag{22}$$

$$Sf_2 = \{A,\ OK,\ MTX\} \tag{23}$$

$$Sf_3 = \{ABORTED,\ MTX\} \tag{24}$$

$$Sf_4 = \{FAIL,\ MTX\} \tag{25}$$

$$Sf_5 = \{OK,\ MTX\} \tag{26}$$

2.4 Task Patterns and task instantiation

In our proposal, a task is defined as $\tau = (N, R, \Pi)$, being N an ordinary Petri net that defines the internal thread of execution, $R = (A, Ev, \gamma, \alpha)$ a set of actions (A) and events (Ev) as well as the functions linking actions and events to fusion places of N (γ, α) and Π the collection of all the parameters needed by the actions A. To define a task pattern is necessary define a Petri net N that will include several fusion places ($Fp_{control}$={*B, A, MTX, OK, FAIL, ABORTED*} and $Fp_{primitive}$={*E, D, OFF, S1, S2*}). All $Fp_{primitive}$ must be related to primitive actions or events using R. Several tasks can use the same task pattern to describe its internal execution flow enabling/disabling different primitives. As exposed in Section 2.3 several task patterns can be defined, even that their internal structure can be different, all of them have to be reducible to the same structure presented in Fig. 3. To instantiate a task $\tau_i = (N, R, \Pi_i)$, a task pattern $\tau_p = (N, R_p)$ and a mapping function $M(R_p)$ are necessary to relate every *generic* action or event in R_p to a set of *particular* actions or events in R . A task can be seen as a function in an imperative language with a set of parameters Π_i that must be defined before call the task. The instantiation of these parameters Π_i will be done in the MCL main code. It is possible use the same task several times changing only this set of parameters due to that the tasks are reusables. It is worth noting that the MCS is not deciding the control actions needed to guide the robot, it only predefines the set of active primitives and their configuration (Π_i). Hence, the realtime guidance of the robot is the responsibility of the set of these enabled primitives at a certain time. Petri nets are only used as a structure able to represent the execution flow of these primitives.

2.5 Control Structures

A mission is defined as $M = (\Gamma, N)$, where Γ is a set of tasks τ_k and N is a Petri net which defines the tasks control flow. In a task, a Petri net defines the internal execution flow control that is the interplay of vehicle actions/events executed within a mission, while in control structures, the Petri net is used to model the execution flow control of the different tasks involved. A task is called like a function in an imperative language. It has two entry point places begin (B) and abort (A) and three possible returning points the places task achieved (OK), task not achieved ($FAIL$) and task aborted ($ABORTED$). Additionally, the MTX place is used as a *mutex* to avoid parallel execution of multiple instances of the same task. Different control structures can be used to join tasks. Again, these control structures must be reduced, from a supervisor point of view, to a structure like the one presented in Fig. 3. Thanks to this constraint it is possible join not only tasks but control structures too. Here, fusion places between tasks an control structures or between control structures and control structures are Fp = {*B, A, MTX, OK, FAIL, ABORTED*}. The B place is responsible for the initialisation of all the parameters needed in the primitive task. Since the tasks are treated as functions, whenever a control structure wants to execute the same task, the same Petri net is used avoiding duplicates. This allows us to keep the mission Petri net smaller. Only the parameter Π_i sent to this task changes from one instantiation to another.

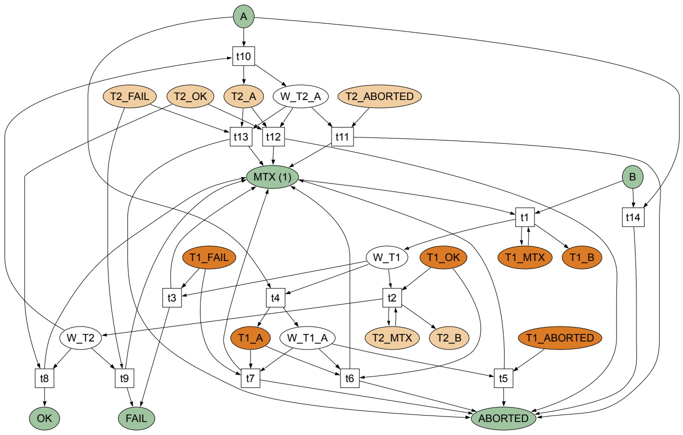

Fig. 4. Sequence control structure Petri net.

MCL provides different control structures: sequential and parallel task execution, conditional and iterative control flow and even a monitoring task. If tasks can be seen in MCL as function calls, these control structure can be seen as control flow structures in a high-level programming language.

The basic control structure in MCL is the one used to sequence two structures [4]. Fig. 4 shows the Petri net used for this purpose. In the sequence Petri net of Fig. 4 there are three groups of fusion places:

$$FP_1 = \{T1\_B, T1\_A, T1\_MTX, T1\_OK, T1\_FAIL, T1\_ABORTED\} \quad (27)$$

$$FP_2 = \{T2\_B, T2\_A, T2\_MTX, T2\_OK, T2\_FAIL, T2\_ABORTED\} \quad (28)$$

$$FP_3 = \{B, A, MTX, OK, FAIL, ABORTED\} \quad (29)$$

Fusion places 27 and 28 are used to connect the structures that have to be sequenced. Fusion place 29 is used to connect this control structure with another superior control structure to hierarchically build a mission. If fusion places FP_1 and FP_2 are connected to two Petri net models corresponding to two other structures represented by the net shown in Fig. 3, the reachable states from $M_0 = \{B, A, MTX, T1\_MTX, T2\_MTX\}$ or from $M_0 = \{B, MTX, T1\_MTX, T2\_MTX\}$ are exactly the same final states obtained in 22 to 26. It is worth noting that, as stated in the previous section, when an abort request is sent to a task (or in general to a subordinated structure), the task can finalise in three possible states: *FAIL*, *OK*, or *ABORTED*. For this reason, the Petri net of the control structure shown in Fig. 4 has been carefully designed to drain the remaining token located in the *A* place of the subordinated structure when it finalises in the *FAIL* or the *OK* state after an abort request. In this design, an abort request sent to the control structure is propagated through *t4* or *t10* to the subordinated tasks. It is guaranteed that if the abort request arrives to a subordinated structure, independently of its final state (*FAIL*, *OK* or *ABORT*) the final marking of the structure will remain the same of its initial marking, ensuring its reusability.

Other control structures are included in MCL like *parallel* that executes two structures in parallel and finalises when both have finalised, *while* that iterative executes a body structure while the result of executing a conditional structure is *OK* ending the iteration otherwise. The *if-then-else* control structure is connected with three structures. Depending on the response of the first one (*condition*) the second (*then*) or the third (*else*) structure is executed. The last proposed control structure (thought that others can be developed) is the *monitor*. This control structure executes two structures in parallel. If the first one finalises before the second the *monitor* finalises in a *FAIL* state. Otherwise, if the second structure finalise before the first one, the first is aborted and *monitor* finalises with an *OK*.

[4] From here in advance we will refer as *structures* both control structures and tasks.

3. TESTING THE MCL

To test the MCL a simple mission has been programmed and executed using the ICTINEU^{AUV}, Ribas et al. [2007]. To be able to use the MCL with the ICTINEU^{AUV} its control architecture has been tailored to the MCS through the AAL. It is worth noting that since the tasks Petri nets are reliable (they are free of deadlocks and they are reusable) and it has been shown that the control structures can be designed to obtain a reliable Petri net from the composition of several, subordinated and reliable Petri nets, the final mission is also structurally reliable. Hence, no analysis is needed at the compilation time. For this reason, the aim of this section is to show how a mission can be encoded in MCL, and how it can be easily executed by an AUV.

3.1 Mission Definition

The mission, inspired in the Student Autonomous Underwater Challenge - Europe (SAUC-E) (Ribas et al. [2007]), consists on a survey trajectory in a water tank looking for a target (a cross). Using a primitive able to detected the cross and a *monitor* structure, the survey is aborted when the cross is found and the vehicle is submerged to drop a marker over the cross before surfacing. Otherwise, if the cross is not found, the vehicle surfaces at the end of the survey trajectory. The aim of this experiment is to show how a mission can be programmed with MCL using their control structures.

3.2 Programming The Mission

Algorithm 1 presents the proposed mission. The survey function is a sequence of tasks that drives the vehicle through a set of predefined way points. The mission starts executing the *FindCross* task, which programs the camera to detect the cross, together with the survey function. If the survey finalises before the *FindCross* task is able to detect the cross the vehicle surfaces and the mission ends but, if the *FindCross* task detects the cross, the *monitor* control structure aborts the survey and returns an *OK* that allows the *if-then-else* control structure to execute the sequence of tasks *AchieveDepth* and *DropMarker* before surface.

To automatically translate the MCL code into a Petri net, an MCL compiler is under development. It uses a two steps compilation; in the first step the imperative code composed by primitive tasks an control structures is translated into functional code (Algorithm 2) in which every control structure is a function whose parameters are other functions (control structures or tasks). The second step is used to translate the functional program into the whole control Petri net joining the small Petri

Algorithm 1 Mission code

```
function Survey() {
    AchieveWayPoint(WayPoint_1);
    AchieveWayPoint(WayPoint_2);
    ...
}
mission {
    if(monitor(Survey(), FindCross())) then {
        AchieveDepth(CrossDepth);
        DropMarker();
    }
    AchieveDepth(SurfaceDepth);
}
```

nets described above using fusion places as exposed in Section 2.

Algorithm 2 Functional mission code

```
sequence(if-then-else( monitor( Survey(), FindCross() ), sequence( AchieveDepth(CrossDepth), DropMarker()), NULL), AchieveDepth(SurfaceDepth) )
```

4. RESULTS

The mission has been tested in a 16x8x5 meters water tank with the ICTINEU^{AUV} using a compass, a Doppler Velocity Logger (DVL) and a pressure sensor for the navigation and a B&W video camera to detect the cross on the floor. Due to the small available space and the perturbations on the compass when the vehicle is near the walls the survey trajectory is far from be optimal, however, our aim is not the navigation but to present a simple and powerful method to define a reactive mission for an AUV combining simple actions. Fig. 5 shows the resulting trajectory estimated using a DVL sensor compared with the desired trajectory. The vehicle starts doing a typical survey trajectory until the cross is detected. In this moment the survey is aborted to submerge the vehicle, drop a marker and finally surface.

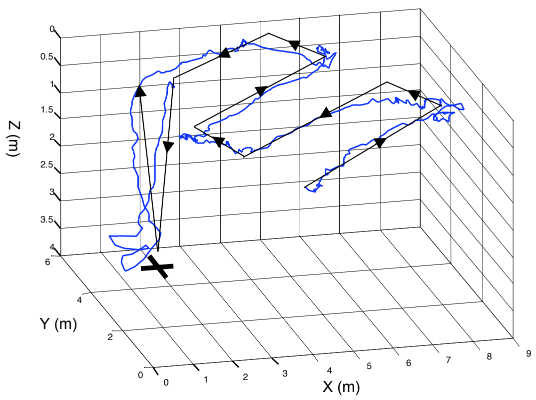

Fig. 5. Estimated 3D trajectory during the experiments using DVL data compared with the desired trajectory.

5. CONCLUSIONS & FUTURE WORKS

This is an ongoing research project to design and implement a flexible MCS easy to be tailored to different AUV control architectures. After a brief introduction about the state of the art of MCS for AUVs a MCS based on Petri nets have been presented. In this work, Petri nets are used to safely model the behavior of a vehicle primitive, a task and a control structure. All these Petri net structures have been designed free of deadlocks and reusable. It has been shown that it is possibly to compose tasks and control structures using control structures to generate the whole mission control Petri net. Instead of using graphical tools to describe the mission, our approach uses the MCL which compiles a high level mission description into a Petri net. MCL presents agreeable properties of simplicity and structure programming as well as facilities for sequential/parallel, conditional and iterative task execution. MCL can be easily tailored to different control architectures through a clear interface based on *actions* and *events*. The AAL is responsible for mapping the *actions* into executable vehicle primitives and the vehicle primitives state changes into *events*. Another interesting facility of MCL is its capability to expand the language through the definition of new task patterns or control structures. The results reported here where obtained by manually translating the mission into the Petri net which was then automatically executed in the AUV. Finalise the MCL compiler as well as define more task patterns and control structure to make the language reacher and more powerful are the next step to obtain a complete MCS.

REFERENCES

B. Allen, R. Stokey, T. Austin, N. Forrester, R. Goldsborough, M. Purcell, and C von Alt. Remus: a small, low cost auv; system description, field trials and performance results. In *OCEANS '97. MTS/IEEE Conference Proceedings*, volume 2, pages 994–1000, 1997.

M. V. Iordache and P. J. Antsaklis. *Supervisory Control of Concurrent Systems, A Petri Net Structural Approach.* Birkhäuser, 2006.

T. W. Kim and J. Yuh. Task description language for underwater robots. In *Intl. Conference on Intelligent Robots and Systems*, 2003.

D.B. Marco, A.J. Healey, and R.B. Mcghee. Autonomous underwater vehicles: Hybrid control of mission and motion. *Autonomous Robots*, 3:169–186, 1996.

T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.

Paul Michael Newman. *MOOS - Mission Orientated Operating Suite.* Department of Engineering Science Oxford University, 2005.

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, special issue on Underwater Robotics*, 29(10), 1998.

Rodolfo Oliveira. Supervision and mission control of autonomous vehicles. Master's thesis, Department of Electrical and Computer Engineering, Instituto Superior Técnico, Lisbon, Portugal, August 2003.

J.R. Perrett and M. Pebody. Autosub-1. implications of using ditributed system architectures in auv development. In *International Conference on Electronic Engineering in Oceanography*, 1997.

D. Ribas, N. Palomeras, P. Ridao, M. Carreras, and E. Hernandez. ICTINEU AUV wins the first sauc-e competition. In *IEEE International Conference on Robotics and Automation*, 2007.