MCL: A MISSION CONTROL LANGUAGE FOR AUVS

N. Palomeras * **P. Ridao** * **M. Carreras** * **J. Batlle** *

* *University of Girona*
PIV Building, Campus Montilivi 17071 Girona, Spain
npalomer@eia.udg.es

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 Mission Control Language named MCL, that allows for sequential/parallel, conditional/unconditional and iterative task execution. MCL can be automatically compiled into a Petri net, to formally describe the mission thread of execution. Then the MCS executes the Petri net in real-time over an architecture abstraction layer that communicates with a particular control architecture using a set of custom defined actions and events.

Keywords: Autonomous vehicles, Architectures, Petri-net.

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 be executed by means of some *vehicle primitives* (basic robot commands or behaviours). The mission controller must define how the mission is *divided* into a set of tasks and how primitives are *combined* 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 actions when some events are produced and the Dynamic Control Continuous System (DCS) used for every action 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 for the AUV MARIUS (Oliveira *et al.* 1998). The system was based on Petri nets and was 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 (Newman 2005). Helm decides the most suitable action commands from a set of prioritized mission goals. The Sauvim Task Description Language (STDL) developed by the University of Hawaii (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

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

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. 6-9 Oct. 1997).

Section 2 describes the Mission Control Language (MCL) detailing how 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. Finally, sections 4 and 5 present the results obtained in the mission as well as the conclusions and future work.

2. MISSION CONTROL LANGUAGE

In this section a new proposal for a MCS based on the Petri net formalism (Murata 1989) and described using a MCL is presented. An abstraction layer used to isolate the MCS from the vehicle control architecture is also introduced.

2.1 Architecture Abstraction Layer

Our intention has been to design the MCL 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 communicating the MCS with the vehicle architecture keeping it architecture-independent. The AAL offers an interface based on two types of signals: Actions and Events. Actions provoke the execution of basic primitives within the vehicle architecture or the activation of a timer. An Action can also provoke the evaluation of an expression. In the case of a boolean expression it produces a true or false event. Events are also used to notify goal achievements, timeout expirations and system exceptions. The AAL is also in charge of communicating the vehicle system variables (position, velocity, acceleration, altitude, ...) to the MCS (figure 1). 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 containing an identifier (a_j) and a set of parameters (π_k) and the AAL has to associate every identifier with a vehicle primitive, a timer or an expression evaluation. At the same time, the AAL has to notify to the MCS the values of all the system variables and the events produced by the achievement of a goal (way-point achieved, timeout expired, ...) or the firing of any possible exception (water inside, high temperature, ...).

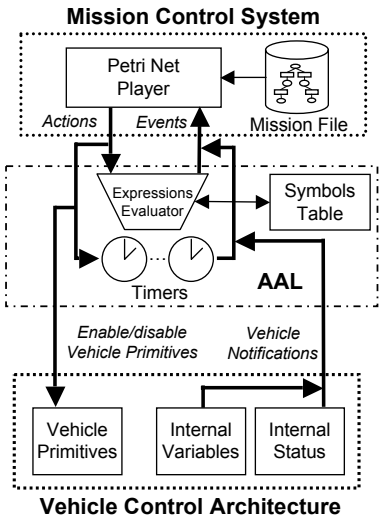

Fig. 1. Architecture Abstraction Layer between the MCS and a particular architecture.

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 a similar control architectures. It is only necessary to define the basic actions that can be executed in the vehicle architecture, the variables and events that it can transmit to the MCS and reprogram the AAL with the mapping between the MCS messages and the vehicle primitives (algorithm 1).

Algorithm 1 Tailoring MCL with O^{2}CA^2

```
Actions {
    KeepDepth(enable, depth, priority);
    Timer(enable, time);
    ...
}
Events {
    KeepDepth::KDepth_ev(Achieved);
    Timer::Timer_ev(Achieved);
    CrossDetector::CDetected_ev(Achieved);
    ...
}
System_variables {
    float depth;
}
Resources {
    Depth_DOF;
    ...
}
```

2.2 Tasks

A task can be defined as a flow of states, where each state is characterised by a set of enabled actions pursuing a set of related goals. When these goals are fulfilled, the state changes, disabling certain actions and enabling others. In our approach, this control execution is modelled with an ordinary Petri net extended as follows: (1) Every transition has a set of events and a set of actions associated; (2) a transition t is enabled if each

input place p of t is marked with 1 token (we use ordinary Petri nets) but can fire only if its associated events are received. Finally, (3) when a transition t is fired, in addition to removing 1 token from each input place p of t, and adding 1 token to each output place p of t, the set of associated actions is executed. With these extensions it is possible to disambiguate the conflicts in the Petri net by firing only the transitions whose events have been received.

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 transitions (γ, α) and Π the collection of all the parameters needed by the actions. The events are produced by the control architecture and transmitted through the AAL. There is a library of basic actions (A) that the vehicle can execute. Therefore, the MCS has the work of executing the current task by executing the set of related actions with their corresponding parameters $a_j(\pi_k)$. It is worth noting that the MCS is not deciding the control actions needed to guide the robot, it only fixes the set of active primitives (behaviours, controllers, ...) and their configuration. Hence, the realtime guidance of the robot is the responsibility of the set of these enabled primitives at a certain time. Petri nets are used as a structure able to represent the parallel/sequential execution thread of the actions involved in a task. All the tasks start with a place called **Begin** and end with two places called **Task achieved** and **Task not achieved**. In order to illustrate how a task is implemented, let us consider the $AutoDepth(depth, timeout, priority)$ task shown in figure 2. This is a very simple task involving only one behaviour and a timeout. The corresponding Petri net has a transition to enable the *timer* and the $KeepDepth$ behaviour (t_1) and two transitions more (t_2 and t_3) waiting for the events timeout expiration or depth achievement. Places p_3 and p_4 are pure indicatives of the state in which the Petri net is. Transitions t_4 and t_5 are used to deactivate the *timer* and the $KeepDepth$ behaviour. Using the Petri net transition rules described above, the task presented in figure 2 does not present any conflict; if the desired depth is achieved by the vehicle before the timeout expires, the **task achieved** place will be marked with a token, or else **task not achieved** will be the marked.

Several tasks can use the same Petri net to describe its internal execution flow. For this reason we differentiate between tasks and task patterns. A task pattern is defined as $\tau_p = (N, R_p)$, being N an ordinary Petri net, defining the internal thread of execution, and $R_p = (A, Ev, \gamma, \alpha)$ a set of generic actions and events together with the

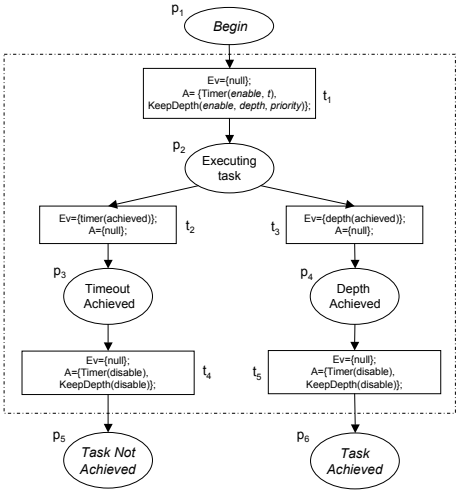

Fig. 2. AutoDepth(depth, t, priority) Task Petri net.

mapping functions from transitions to actions and events. Typical patterns include **Achieve One Goal** (figure 2) or **Keep One Goal** among others. While the first tries to achieve a certain goal through the execution of certain behaviour and ends when the goal is achieved or when the timeout expires, the second tries to achieve and keep a certain goal during a period of time. Of course, other task patterns can be defined. Primitive tasks are obtained through the instantiation of task patterns. 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 to a set of particular actions or events. A vector Π_i with all the expressions needed to define the parameters used in the particular actions is also required. The expressions in Π_i vector will be evaluated every time a task is called. It is possible to instantiate several primitive tasks using the same task pattern changing only the $M(R_p)$ function. Algorithm 2 shows how task patterns are defined and instantiated using MCL.

2.3 Tasks Joining

A mission is defined as $M = (\Gamma, PN)$, where Γ is a set of tasks τ_k and PN is a Petri net which defines the tasks control flow. While within a task, a Petri net defines the internal thread of execution of vehicle actions, within a mission, the (PN) Petri net defines the thread of execution of the tasks. A task is called like a function in an imperative language. It has an entry point (the **Begin** place) and two possible returning points (the places **Task achieved** and **Task Not Achieved**). Different control structures can be used to join tasks. Again, these control structures are encapsulated between these three places as in any primitive task and hence, every control structure can be used as a primitive task as well (figure 3). The union between these structures and the primitive tasks is

Algorithm 2 Task Pattern Definition and Instantiation

AchieveOneGoal (**a**: bh, **a**: timer, **a**: disable_bh, **a**: disable_timer, **e**: timer_ev, **e**: bh_ev) {
  places {
    p1: **Abort**; p2: **Begin**; p3 ; p4 ; p5 ,
    p6 : **Achieved**; p7 : **Not_Achieved**;
  }
  transitions {
    t1 : **Ev**={null}; **A**={null};
    t2 : **Ev**={null}; **A**={bh, timer};
    t3 : **Ev**={timer_ev}; **A**={null};
    t4 : **Ev**={bh_ev}; **A**={null};
    t5 : **Ev**={null}; **A**={disable_bh, disable_timer};
    t6 : **Ev**={null}; **A**={disable_bh, disable_timer};
  }
  arcs {
    $p1 \rightarrow t1; p2 \rightarrow t2; p3 \rightarrow t1; p3 \rightarrow t3; p3 \rightarrow t4;$
    $p4 \rightarrow t5; p5 \rightarrow t6; t1 \rightarrow p4; t2 \rightarrow p3;$
    $t3 \rightarrow p4; t4 \rightarrow p5; t5 \rightarrow p6; t6 \rightarrow p7;$
  }
}
AutoDepth(**float** depth, **int** timeOut, **int** p = 4): **AchieveOneGoal** {
  a: bh: KeepDepth(enable, depth, p): KDepth_ev;
  a: timer: Timer(enable, timeOut): Timer_ev;
  a: disable_bh: KeepDepth(disable): NULL;
  a: disable_timer: Timer(disable): NULL;
  e: timer_ev: timer::Timer_ev($true$);
  e: bh_ev: KeepDepth::KDepth_ev($true$);
  Resources: Depth_DOF;
}

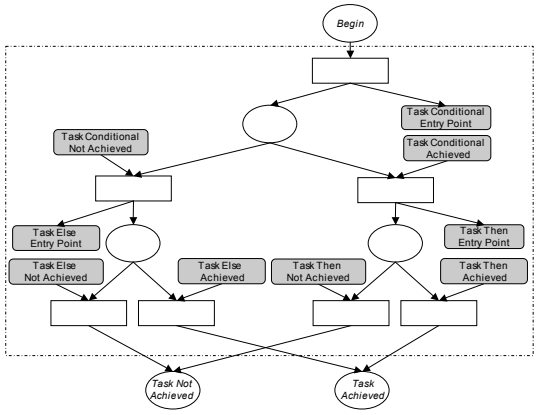

Fig. 3. IF-THEN-ELSE control structure.

marked as the **Entry Point**, the **Task achieved** and the **Task Not Achieved** junctions. These junctions have to be joined with the **Begin**, **Task achieved** and **Task Not Achieved** places of the primitive task respectively and, moreover, the transition that enables the **Begin** place of every primitive task is responsible for the initiation of all the parameters needed by this task. Since the tasks are treated as functions, whenever a control structure wants to execute the same task, the same Petri net is used instead of using a duplicate. This allows us to keep the Petri net as small as possible. Only the parameter value Π_i defined by the preceding transition changes from one instance to another.

Using this simple implementation when the same task is called several times sequentially the approach works fine, however, if a primitive task is called by two parallel threads, an error due to condition races could appear. For instance, imagine that while a task is being executed, a new token arrives at its **Begin** place. This means that its parameters have been changed and the task has re-entered before its finalisation. Since recursion is not allowed within our approach, to prevent for erroneous task re-entries a mutual exclusion (mutex) will be automatically added by the compiler to every primitive task to ensure the mutual exclusion during its execution. This mutex is just a place initialised with one token inside. When the task is enabled the token is lost avoiding it to be re-entered. After the execution of the task the token in the mutex place is restored. This same mechanism is used to control the access of other resources. For instance, if two tasks want to control the vehicle in the same Degree Of Freedom (DOF), a place is added and linked with the two tasks to symbolise the shared resource and avoid their simultaneous execution.

2.4 Control Structures

MCL provides different control structures: **Conditional** and **Unconditional** sequencing, **If-Then-Else**, **While**, **For** and **Parallel** structures. In the **Conditional sequencing**, the execution of the second task is conditioned by the achievement of the previous one. Tasks can also be **Unconditionally sequenced**. Then, the last task is executed independently of the result of the preceding one. The **Parallel AND/OR** structure allows for the parallel execution of tasks. By using the two output places of each task (**Task Achieved** and **Task Not Achieved**) it is possible to join tasks using **If-Then-Else** or **While** constructions. If a mission programmer wants to use a boolean expression in a **If-Then-Else** or **While** structure instead of the result of a primitive task, it is also possible use a special structure to evaluate boolean expressions. It is actually a kind of primitive task. Another control structured provided by the MCL is an exception handling mechanism. It uses a **Try-Catch** construction present in most popular high level programming languages (figure 4). To support exception handling, the tasks have to be abort-able. If several tasks are under execution and the system provokes an exception, the mission control system has to abort the tasks and execute another set. In parallel with the execution of the **Try** block, a transition is waiting for the exception event defined in the **catch** block. If the exception event is received, the **Try** block is aborted and the code inside the **catch** block is executed. When either

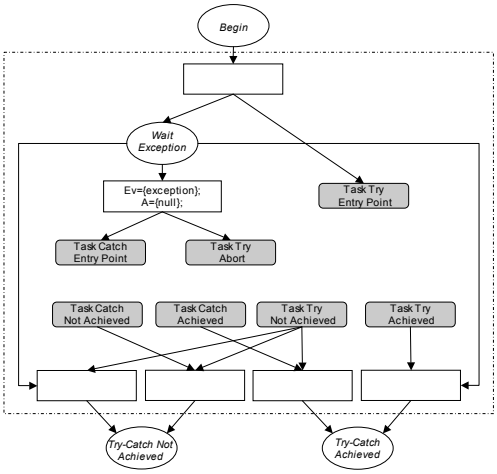

Fig. 4. TRY-CATCH exception handling structure.

the **Try** or the **Catch** block end, the execution flow continues after the **catch** block.

3. TESTING THE MCL

To test the MCS a simple mission is programmed with MCL and executed using the ICTINEU^{AUV}. The mission consist on locating a submerged cross at the bottom of a water tank and mark it once it has been detected.

3.1 Tailoring MCL to the O^{2}CA^2 Architecture

O^{2}CA^2 is a behaviour-based control architecture (Palomeras et al. 2006) that uses the proposed MCS as the upper deliberative layer. Figure **??** shows the schema of the two behaviour-based levels (vehicle-level and task-level) and the relation of these with the MCS through the AAL. All the executable actions within the task-level and the events generated by the O^{2}CA^2 Control Architecture have to be defined using the MCL and programmed in the AAL. Most of the actions on O^{2}CA^2 consist on enabling/disabling the robot behaviours in charge of controlling the vehicle movement and the events are related with the achievement or not of the behaviour goals. The system variables are also defined using MCL (algorithm 1) and their access programmed through the AAL. Once compiled, MCL will automatically generate a code skeleton for the AAL. This skeleton has to be manually completed by the programmer. It is worth noting that the tailoring of MCL to the control architecture has to be programmed once, being shared by all the missions.

3.2 Programming The Mission

The definition of task patterns and the set of primitive tasks used to enable/disable these behaviours as well as the resources to avoid the mutual execution of them is also defined in this tailoring section (algorithm 1). Once the MCS is tailored to the vehicle's control architecture, programming the whole mission is simple. An example of a preliminary mission inspired with the Student Autonomous Underwater Challenge - Europe (SAUC-E) 2006 (Ribas et al. 2007) is presented in algorithm 3. Given a particular depth the surveyed trajectory is carried out while looking for the cross. When the cross is detected, the survey is aborted to submerge the vehicle, drop a marker over the cross and surface again. Otherwise, if the cross is not found, the vehicle surfaces at the end of the trajectory.

Algorithm 3 describe the proposed mission using MCL. The **FindCross** task which programs the camera to produce a **CDetected_ev** event when the vehicle passes over the cross, is started when the mission begins. Next, a **try-cath** structure is enabled. The **try** block is composed by a **FollowWaypoints** behaviour in charge of performing the survey movement in **parallel** with an **AutoDepth** controller to control the vehicle depth. If the **CDetected_ev** event is raised the **try** block is aborted and the **catch** block is executed **submerging** the vehicle, making it to drop a marker near the cross and bringing it back to the surface to end the mission.

Algorithm 3 Mission code

```
mission {
    FindCross();
    try {
        parallel{
            FollowWaypoints(WPList, velocity);
        } or{
            AutoDepth(0.2, t);
        }
    } catch(CrossDetector::CDetected_ev(true)) {
        AutoDepth(3.5, t);
        DroopMarker();
        AutoDepth(0, t);
    }
     AutoDepth(0, t);
}
```

To automatically translate the MCL code into a Petri net, an MCL compiler is under development. It uses a three steps compilation; in the first step the imperative code composed by primitive tasks an control structures is translated into a functional code (algorithm 4) in which every primitive task and every control structure are functions whose parameters are instance of functions. The second step is used to translate the functional program into the corresponding Petri nets and joining them as exposed in section 2.3. Last step is used to add the resources joined with the corresponding task primitives.

Algorithm 4 Functional mission code

seq(FindCross(), **try_catch**(CDetected_ev, **parallel_or**(FollowWaypoints(...) , AutoDepth(...)), **seq**(AutoDepth(...), **seq**(DroopMarker(), AutoDepth(...)))))

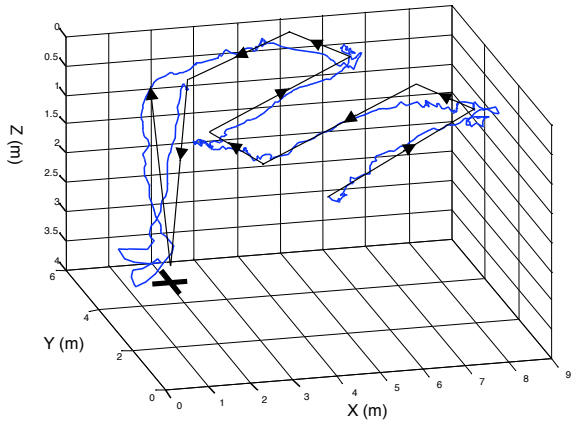

Fig. 5. Estimated 3D trajectory during the experiments using DVL data compared with the desired trajectory.

4. RESULTS

The mission has been tested in a 16x8x5 meters water tank with the ICTINEU^{AUV} using a compass 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 of the compass when the vehicle is near to the walls the trajectory is far from be the 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. Figure 5 shows the resulting trajectory estimated using a Doppler Velocity Logger (DVL) sensor compared with the desired trajectory. The vehicle starts doing a typical survey trajectory at a predefined depth until the **CDetected_ev** event is raised. In this moment the survey is aborted to submerge the vehicle, drop a marker and finally surface.

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. Instead of using graphical tools for programming the Petri nets, our approach defines the MCL which compiles into the Petri net description of the mission. MCL presents agreeable properties of simplicity and structure programming as well as facilities for sequential/parallel, conditional/unconditional and iterative task execution. It also provides capabilities for event/exception handling like the try-catch mechanism. To ensure mutual exclusion over critic resources, a mutual exclusion mechanism is also provided. 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 state changes into events. Another interesting facility of MCL is its capability to grow the language through the definition of new task patterns and task primitives. The results reported here where obtained by manually translating the mission into the Petri net which was then automatically executed with the AUV. Immediately future work will consist on finalise the MCL compiler.

REFERENCES

Allen, B., R. Stokey, T. Austin, N. Forrester, R. Goldsborough, M. Purcell and C von Alt (6-9 Oct. 1997). Remus: a small, low cost auv; system description, field trials and performance results. In: OCEANS '97. MTS/IEEE Conference Proceedings. Vol. 2. pp. 994–1000.

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

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

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

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

Oliveira, P., A. Pascoal, V. Silva and C. Silvestre (1998). Mission Control of the MARIUS AUV: System Design, Implementation, and Sea Trials.

Palomeras, N., M. Carreras, P. Ridao and E. Hernandez (2006). Mission control system for dam inspection with an auv. IROS.

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

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