A MISSION CONTROL SYSTEM FOR AN AUTONOMOUS UNDERWATER VEHICLE

Narcís PALOMERAS ROVIRA

Dipòsit legal: GI-253-2012
http://hdl.handle.net/10803/69957

A Mission Control System for an Autonomous Underwater Vehicle

Narcís Palomeras Rovira

ViCoRob

University of Girona

A thesis submitted for the degree of

PhD in Computer Engineering

Supervisors:

Pere Ridao Rodriguez and Carlos Jorge Ferreira Silvestre

July 2011

Contents

List of Figures

LIST OF FIGURES

Acronyms

AAC Architecture Abstraction Component.

AC Asymmetric Choice.

AHRS Attitude and Heading Reference System.

AHRS-FOG Attitude Heading Reference System - Fiber Optic Gyroscope.

AI Artificial Inteligence.

ANTLR Another Tool for Language Recognition.

API Application programming interface.

ASC Autonomous Surface Craft.

ASL Autonomous System Lab.

AST Abstract Syntax Tree.

AUV Autonomous Underwater Vehicle.

BBP Behavior-Based Planning.

BFS Breadth-First Search.

BNF Backus Normal Form.

CCD Charge Coupled Device.

CCR Concurrency and Coordination Runtime.

CIRS Centre d'Investigacio en Robotica Submarina.

COLA2 Component Oriented Layer-based Architecture for Autonomy.

CORBA Common Object Request Broker Architecture.

CURV Cable-Controlled Underwater Recovery Vehicle.

DES Discrete Event System.

DFS Depth-First Search.

DGPS Differential Global Positioning System.

DOF Degree of Freedom.

DSL Domain Specific Language.

DSS Decentralized Software Services.

DSTL Defence Science and Technology Lab.

DVL Doppler Velocity Log.

EFC Extended Free-Choice.

EKF Extended Kalman Filter.

FC Free-Choice.

FOG Fiber Optic Gyroscope.

FSM Finite State Machine.

GPS Global Positioning System.

GUI Graphic User Interface.

HIL Hardware In the Loop.

IAUV Intervention Autonomous Underwater Vehicle.

ICE Internet Communications Engine.

IDL interface Definition Language.

IIS Institute of Industrial Science.

ILP Integer Linear Program.

INRIA Institut National de Recherche en Informatique et en Automatique.

IP Internet Protocol.

IPC Inter Process Communication.

ISE International Submarine Engineering.

ISR Institute for Systems and Robotics.

JAUS Joint Architecture for Unmanned Systems.

LBL Low BaseLine.

LED Light-Emitting Diode.

LOS Line Of Sight.

MAS Multi Agent System.

MCL Mission Control Language.

MCL-C Mission Control Language - Compiler.

MCS Mission Control System.

MES Mitsubi Engineering Shipbuilding.

MG Marked Graph.

MOOS Mission Oriented Operating Suite.

MRDS Microsoft Robotics Developer Studio.

MRG Mobile Robotics Group.

MRU Motion Reference Unit.

MRU-FOG Motion Reference Unit-Fiber Optic Gyroscope.

MSIS Miniking Mechanically Scanned Imaging Sonar.

NAC Natural Actor Critic.

NASA National Aeronautics and Space Administration.

NOCS National Oceanography Center of Southampton.

NPS Naval Postgraduate School.

O2CA2 Object Oriented Control Architecture for Autonomy.

OOI Object Of Interest.

PCL Point Cloud Library.

PID Proportional Integral Derivative.

PNBB Petri Net Building Block.

PNML Petri Net Markup Language.

PNP Petri Net Player.

POP Partial Order Planning.

POSIX Portable Operating System Interface for Unix.

RAP Reactive Action Packages.

REST Representational State Transfer.

RL Reinforcement Learning.

ROS Robot Operating System.

ROV Remotely Operated Vehicle.

RPC Remote Procedure Call.

RPL Reactive Plan Language.

SAUC-E Student Autonomous Underwater Challenge-Europe.

SBPI Supervision Based on Place Invariant.

SCADE Software Critical Application Development Environment.

SDL Specification and Description Language.

SM State Machine.

SOAP Simple Object Access Protocol.

SPA Sense Plan and Act.

STL Standard Template Library.

STRIPS Stanford Research Institute Problem Solver.

T-REX Teleo-Reactive EXecutive.

TAO The Ace Orb.

TCP Transmission Control Protocol.

TDL Task Description Language.

TecGraf Computer Graphics Technology Group.

UdG University of Girona.

UDP User Datagram Protocol.

UJI Universidad Jaume I.

UL-S Unconstrained Least Squares.

ULV Ultra Low Voltage.

UML Unified Modeling Language.

UPC Universitat Politècnica de Catalunya.

URIS Underwater Robotic Intelligent System.

USB Universal Serial Bus.

USBL Ultra Short BaseLine.

UUV Unmanned Underwater Vehicle.

VICOROB Computer Vision and Robotics.

VPL Visual Programming Language.

WHOI Woods Hole Oceanographic Institution.

XML Extensible Markup Language.

YARP Yet Another Robot Platform.

Chapter 1

Introduction

There are plenty of places in our world in which a man has never been. Most of these places are hidden in the depths of our seas and oceans. Maybe for this reason, the underwater exploration began many years ago. It started investigating the deepness of the seas, trying to understand its physical and chemical characteristics and learning about the life forms that inhabit this realm [ScienceEncyclopedia, 2011]. Despite underwater research started with manned vehicles, like the bathyspheres build by Charles William Beebe (1877-1952) and Auguste Piccard (1884-1962) or the famous *Alvin* deep-sea submersible build in 1964 by Allyn Vine (1941-1994), first unmanned underwater vehicles appeared soon.

In the 1950s, the Royal Navy builts a pioneer Remotely Operated Vehicle (ROV) named *Cutlet* to recover practice torpedos. One decade later, the US Navy funded most of early ROV technology with what they called the Cable-Controlled Underwater Recovery Vehicle (CURV). This ROV was used to perform deep-sea rescue operations as well as to recover objects from the ocean floor like the ill-fated USS Thresher and the hydrogen bomb lost by the US Navy in the coast of Palomares in Spain. Building on this technology base, the offshore oil and gas industry created the work class ROVs to assist in the development of offshore oil fields. More than a decade later, ROVs became essential during the 1980s when much of the new offshore development exceeded the reach of human divers. Technological development in the ROV industry has accelerated and today's ROVs perform numerous tasks in many fields. These tasks range from simple inspection of sub-sea structures, cable and platforms to connecting pipelines and placing underwater manifolds. They are used extensively both in

the initial construction of a sub-sea development and the subsequent repair and maintenance.

One might consider that ROVs constitute the state of the art for most of the technological applications nowadays. However, the tether cable imposes some constraints such as limiting the working area and making the operation more difficult. These limitations triggered research into a new generation of underwater robots, this time autonomous, mainly developed for inspection purposes and to collect data and samples. Most of these vehicles are prototypes developed by research centers, although, some of them are commercially available. There are several potential Autonomous Underwater Vehicle (AUV) applications being explored by various organizations around the world [Davis, 1996]: environmental monitoring, oceanographic research, underwater archeology [Conte et al., 2010] and maintenance/monitoring of underwater structures are just a few examples. AUVs are attractive for use in these areas because of their size and their non-reliance on human operators. Also, AUVs can be deployed in larger numbers and for longer periods. Moreover, in restricted environments such as kelp forests or under ice, AUVs perform better than typical ROVs. Given the potential applications and advantages of AUVs, it is no wonder that academic and commercial organizations around the world are conducting research using these vehicles.

Although technological advances in marine vehicles offer high reliable hardware, the software that controls the AUVs has the complex, and not yet solved, task to adapt the AUV's actions to the unpredictable changes produced in the environment to fulfill a mission plan. This thesis focuses on the software which takes all these decisions: the vehicle's control architecture.

A robot architecture contains elements that range from device drivers to components in charge to perform high level tasks like tracking a cable or planning a mission. As exposed in Kortenkamp and Simmons [2008], several questions should be answered before implementing a robotic architecture: For which kind of missions the robot is designed for? Which set of actions are needed to perform this mission? Which data is necessary to do it? Who are the robot users? How the robot will be evaluated?. These are some questions to be answered before facing the development of a robot architecture. An organized analysis of each one allows a successful implementation of a particular architecture.

Starting from the kind of mission to be performed by current AUVs, they range from surveys, inspections, intervention operations or target location missions. In

order to achieve them, AUVs generally have a set of *primitives* which describe simple tasks that the vehicle is able to perform autonomously like *"go to a point"*, *"track a cable"* or *"reach a desired depth"*. The data needed by these primitives is acquired by a complete sensor suite. However, to improve the robustness of the gathered data, auxiliary components are designed to fuse, filter and refine those data. AUV users are mainly scientists or engineers who want a simple way to accurately control all the steps taken during the development of a mission. This is one of the reasons that predefined plans are more popular in underwater robotics than in other fields like surveillance [Saptharishi et al., 2002], mail delivering [Beetz et al., 2001] or tour guide robots [Burgard et al., 1999] where some goals are given and is the robot who actually plans the mission.

This thesis describes the development of a control architecture for an AUV, with the main focus on the design and implementation of the components used to define and execute autonomous missions for a single or multiple AUVs. We refer to the set of components used to describe and execute a mission by an autonomous vehicle as a Mission Control System (MCS).

Following sections describe the motivations, objectives and the organization of the thesis.

1.1 Motivations

During the last decade, the Centre d'Investigacio en Robotica Submarina (CIRS) has developed several ROV and AUV prototypes. In parallel with the construction of these vehicles the necessary software to control them has been developed. What initially was a set of drivers to access the thrusters and the few sensors of these early ROVs and AUVs, has become a complex control architecture composed of drivers, controllers, processing units and autonomous behaviors. The reactive layer of a new behavior-based architecture named Object Oriented Control Architecture for Autonomy (O2CA2) was developed some years before in the CIRS [Ridao et al., 2002]. Although O2CA2 allowed the execution of autonomous tasks, a system to define and execute a mission, in which the set of behaviors under execution may be modified at anytime, was not included. Therefore, the main motivation of this dissertation is to enhance the features of this previous architecture developing a system adequate to describe and execute a mission plan for an AUV. The system has to be easy to use by a standard AUV user, a scientific

or an engineer, and has to provide mechanisms to ensure the safety of the defined mission before and during its execution. Moreover, the MCS has to be as generic as possible, being able to be adapted to various vehicles. The possibility to define missions in which multiple vehicles are coordinated as well as to increase the deliberation capabilities of the autonomous vehicles will be also considered.

1.2 Goal of the thesis

The way in which a mission is described and executed by an autonomous vehicle is closely related to several factors: the used control architecture, the mission to execute and the available hardware are among the principal ones. However, many similarities can be found between different AUVs and missions. Therefore, the main goal of this thesis is described as follows:

Develop a system to define and execute missions for autonomous underwater vehicles, simple to use from the users point of view, and easily adaptable to different control architectures.

This work has been carried out at CIRS, part of the Computer Vision and Robotics (VICOROB) group at the University of Girona (UdG). The developed MCS has been integrated in the available underwater vehicles: Ictineu AUV [Ribas et al., 2007] and Sparus AUV [Hurtos et al., 2010].

1.2.1 Objectives

The goal of this thesis can be divided into the following more specific objectives:

State of the art: Study the principal control architectures for autonomous vehicles as well as the main MCSs presented in the literature, with a special attention to generic systems as well as to systems tailored to underwater vehicles.

Generic: Implement a MCS as generic as possible simplifying its adaptation to different control architectures.

Mission control system: Propose a simple and formal method to control the execution flow of robot actions. The system has to respond to robot actions

and environmental events which drive the vehicle from a given initial state to a desired final state. Special attention will be given to the inherent parallel structure of autonomous missions assessing the correct ending of the execution flow avoiding getting stuck in undesired deadlocks.

Multiple-vehicle coordination: Study the definition of multiple-vehicle missions by means of coordination constraints. Given the strict communication restrictions found in the underwater environment in terms of band width, particular attention will be paid to bound the amount of information to be interchanged to ensure the coordination.

Enhance deliberation: Although the study of the best deliberation mechanism for an AUV control architecture is not an objective of this dissertation, we are interested on studying the interface between the proposed MCS and a deliberative upper layer based on planning.

Testing: Test experimentally, using different marine robots, the functionalities and the simplicity of use of the proposed MCS.

1.3 Outline of the thesis

After this introductory chapter, a state of the art about MCSs, mainly focusing on AUVs, is presented. Chapter 3 introduces the experimental platforms in which the MCS has been tested as well as the control architecture used by them. How missions are defined, verified and executed using the proposed system is described in Chapter 4 and Chapter 5. An insight of multiple-vehicle mission coordination is given in Chapter 6. Finally, how to interface a deliberative layer with the proposed MCS is shown in Chapter 7. The thesis finalizes with some results in Chapter 8 before the conclusions. A brief description of each chapter is presented next.

Chapter 2: The aim of this chapter is to overview different approaches presented in the literature to deal with the problem of defining and executing a mission. The elements that conform these solutions are grouped in what we call MCS. As MCSs are closely related with the vehicle's control architecture being used, an overview about control architectures is also presented.

Chapter 3: This chapter introduces the experimental platform in which the proposed MCS has been developed and tested. It includes the control architecture where the MCS is integrated as well as the vehicles used in the experimental phase.

Chapter 4: This chapter develops how autonomous vehicle missions are coded using Petri nets. It begins comparing a mission with a formal Discrete Event System (DES). Then, the chapter describes how the vehicle primitives are modeled using Petri nets and are supervised by means of Petri net structures named tasks. How the task execution flow is controlled by means of Petri net control structures is presented next. The properties that have to be verified for each one of these Petri net models/structures as well as how they are composed to generate a full mission is also introduced.

Chapter 5: Connecting different Petri net structures to build a complex mission can be a cumbersome and error-prone work if the user has to deal directly with the Petri nets using graphical tools. To overcome this difficulty, a Domain Specific Language (DSL) named Mission Control Language (MCL) is proposed in this chapter, which allows to define and compose Petri net structures in a simple way.

Chapter 6: The purpose of this chapter is to further develop the presented methodology to deal with the coordination of multiple vehicles. Three coordination constraints have been studied: mutual exclusions, tasks ordering and tasks synchronization.

Chapter 7: Although, automated planning techniques are out of the scope of this thesis, the purpose of this chapter is to study the interface between well-known automated planning algorithms and the proposed MCS.

Chapter 8: This chapter presents four representative experiments: a dam inspection mission in the context of an industrial application, a visual survey in a zone of scientific interest, a multiple-vehicle mission to gather georeferenced data about several Object Of Interest (OOI) and a cable tracking mission in which planning techniques are employed.

Chapter 9: This chapter concludes the thesis by summarizing the work and pointing out contributions and future work. It also comments on the re-

search evolution and the publications accomplished during this research project.

Appendices: These chapters incorporates additional information. An introduction about Petri nets is presented in Appendix A. Appendix B details the proposed control structures introduced in Chapter 4. Finally, the complete Backus Normal Form (BNF) grammar for the MCL is included in Appendix C.

Chapter 2

State of the art

The aim of this chapter is to review various approaches presented in the literature to deal with the problem of defining and executing a mission. The elements that conform these solutions are grouped in what we call Mission Control System (MCS). Since MCSs are closely related with the vehicle's control architecture that is being used, an overview about control architectures is first presented to see how the modules in charge of programming and executing a mission have evolved during the last decades. Next, several well studied MCS alternatives presenting different features are reviewed. The chapter finalizes with a summary and some conclusions.

2.1 Overview of control architectures

The first attempt at building autonomous robots began around the mid-twentieth century with the emergence of Artificial Inteligence (AI). The approach begun at that time was known as *Traditional AI*, *Classical AI* or *Deliberative approach*. Traditional AI relied on a centralized world model for integrating sensory information for reasoning in order to generate actions in the world, following the Sense Plan and Act (SPA) pattern [Nilsson, 1980]. Its main architectural feature was that sensing flowed into a world model, used afterwards by the planner to build a plan which was then executed without continuously relying on the sensors that created the model. The design of the classical control architecture was based on a top-down philosophy. The sequence of phases usually found in a traditional deliberative control architecture can be seen in Figure 2.1. Firsts robots using a SPA

Figure 2.1: Phases of a classical deliberative control architecture.

deliberative control architecture began in the late 1960s with the Shakey robot at Stanford University [Fikes and Nilsson, 1971; Nilsson, 1984]. Mission plans were described by the Shake robot using Stanford Research Institute Problem Solver (STRIPS) [Fikes and Nilsson, 1971] and executed by the PLANEX system. Programing different missions within the same application domain, like for instance package delivering, was quite easy using STRIPS. However, changing the mission domain implied changing the world model, the planning operators, the associated low level actions, and so on. Many other robotic systems were built with the traditional AI approach [Albus; Chatila and Laumond, 1985; Huang, 1996; Laird and Rosenbloom, 1990; Lefebvre and Saridis, 1992], all of them sharing the same kind of problems. Planning algorithms failed with non-trivial solutions and the integration of the world representations was extremely difficult and, as a result, planning in a real world domain took a long time. Also, the execution of a plan without involving sensing was dangerous in a dynamic world. Only structured and highly predictable environments were proved to be suitable for classical approaches.

In the middle of the 1980s, motivated by the poor performance of robots in real environments, a number of scientists began rethinking the general problem of organizing intelligence. Among the most important opponents to the AI approach were Rodney Brooks [Brooks, 1986], Rosenschein and Kaelbling [Rosenschein and Kaelbling, 1986] and Agre and Chapman [Agre and Chapman, 1987]. They criticized the symbolic world used by traditional AI and promoted a more reactive approach with a strong relation between the perceived world and the actions. They implemented these ideas using a network of simple computational elements, connecting sensors to actuators in a distributed manner. There were no central models of the world explicitly represented. The model of the world was the

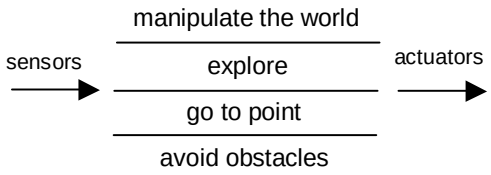


Figure 2.2: Structure of a behavior-based control architecture.

real one as perceived by the sensors at each moment. Leading the new paradigm, Brooks proposed the *Subsumption Architecture*, built as an stack of layers of interacting Finite State Machines (FSMs). This FSMs were called *behaviors*, representing the first approach to a new field called *Behavior-based Robotics*. The behavior-based approach used a set of simple parallel behaviors which reacted to the perceived environment proposing the response that the robot should take in order to accomplish the behavior's goal, see Figure 2.2. Whereas SPA robots were slow and tedious, behavior-based systems were fast and reactive. There were no problems with world modeling or real-time processing because they constantly sensed the world and reacted to it. Several successful robot applications were built using the behavior-based approach [Connell, 1992; Horswill, 1993]. A well known example of behavior-based robotics is Arkin's motor-control schemas [Arkin, 1989], where motor and perceptual schemas are dynamically connected to each other.

Despite the success of the behavior-based models, they soon reached the limits in their capabilities. Limitations when trying to undertake long-range missions and the difficulty to optimize the emerging robot behavior were the most significant ones. Also, since multiple behaviors can be active at any time, behavior-based architectures need an arbitration and coordination mechanism that allows higher-level behaviors to override signals from lower-level behaviors. Therefore, selecting the proper behaviors to achieve robustness and efficiency in accomplishing goals was a key point that has to be addressed. Generally speaking, in a behavior-based architecture there was no system to define a mission plan combining a set of behaviors or schemas. In essence, robots needed to combine the planning capabilities of the classical architectures with the reactivity of the behavior-based architectures, attempting a compromise between bottom-up and top-down methodologies. This evolution was named *Hybrid Architectures* , being this the most common paradigm nowadays. Usually, an hybrid control architec-

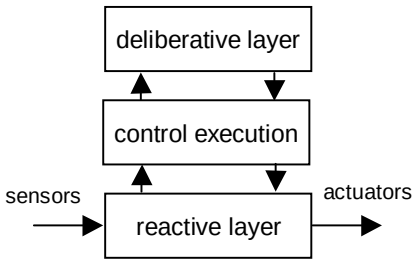


Figure 2.3: The hybrid control architecture structure.

ture is structured in three layers: the reactive layer, the control execution layer and the deliberative layer, see Figure 2.3. The reactive layer takes care of the real time issues related to the interactions with the environment. It is composed by basic robot behaviors relying on some sort of feedback control connecting sensors to actuators. Each behavior may be designed using different techniques, ranging, but not constrained to, from nonlinear control [Barrett et al., 1996] to reinforcement learning [Carreras et al., 2003; El-Fakdi et al., 2010; Peters, 2007]. The control execution layer interacts between the upper and the lower layers, supervising the accomplishment of the tasks assigned to the vehicle. This layer acts as an interface between the numerical reactive and the symbolic planning layers. It is responsible of translating high-level plans into low-level actions by means of enabling/disabling the behaviors in the reactive layer, at the appropriate moment and using the correct parameters. Also, the control execution layer monitors the behaviors being executed and handles the exceptions that may occur. The deliberative layer transforms the mission description into a sequence of actions, a plan. It determines the long-range tasks of the robot based on high-level goals.

One of the first architectures which combined reactivity and deliberation was proposed by James Firby. In his thesis [Firby, 1989], the first integrated three-layer architecture was presented. From there, hybrid architectures have been widely used. One of the best known is Arkin's AURA [Arkin and Balch, 1997], where a navigation planner and a plan sequencer was added to its initial behavior-based motor-control schemas architecture. The Planner-Reactor architecture [Lyons, 1992] and the Atlantis [Gat, 1991] used in the Sojourner Mars explorer are well known examples of hybrid architectures too.

Although most of nowadays architectures use the three-layer model not all the proposed systems have deliberative capabilities. Most of them are limited

to interpret a mission predefined by a human operator, instead of automatically generating a mission plan using an on-board planner or reasoner. This is the case for most Autonomous Underwater Vehicle (AUV) architectures that despite following a hybrid model, only few of them include on-board deliberation capabilities. A summary of 25 existing AUVs as well as a review of 11 AUV control architectures was presented by Valvanis in 1997 [Valavanis et al., 1997]. All of the studied architectures are based on hierarchical, subsumption or hybrid models. Ridao [Ridao et al., 2000], surveyed also several control architectures for AUVs organizing them among deliberative, reactive and hybrid architectures too.

Nowadays, control architectures for autonomous vehicles are composed by a set of components distributed among layers according to their functionalities. These components are independent entities that can communicate with any other component in the architecture. The essence of the hybrid model in which some components react fast to input stimulus while others process the data to take long term decisions is still valid, however, the hierarchy among the components is becoming more vague.

2.2 Mission Control Systems review

All autonomous vehicles that are intended to execute missions autonomously need a MCS. Several common points can be found among the systems presented in the literature. Focusing on underwater vehicles, first point consists in the kind of missions that AUVs have to deal with. Most underwater autonomous missions can be classified into three main categories: scientific, industrial and military missions. In scientific missions, AUVs are used to perform surveys in challenging environments such as deep waters [Whitcomb, 2000], under the ice [Kunz et al., 2008] or in hydrothermal zones [Yoerger et al., 2007]. Surveys are useful to evaluate sand and gravel deposits, examining hydrothermal vents or to obtain bathymetric maps. Industrial applications in which AUVs are employed range from cable deployment [Kao et al., 1992] or dam inspection [Palomeras et al., 2009a] to environmental monitoring [Griffiths et al., 1998]. Finally, military applications which include the use of AUVs are focused on localization and recovery of underwater targets [Anderson and Crowell, 2005], mine counter measure missions, port security or inspection of hostile environments.

More similarities are found with the control architecture model implemented

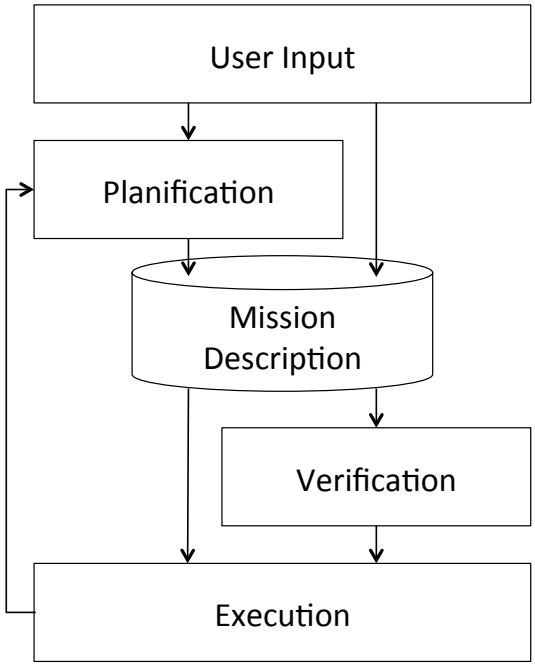


Figure 2.4: MCS classification.

by the AUVs. Although old architectures follow the subsumption model, current control architectures follow a layer-based approach with or without on-board deliberative capabilities. However, both control architectures models share a common feature: all of them have a set of basic actions that are executed to achieve a simple goal like reach a way-point, keep an altitude or enable a sensor. These basic actions are named commands, actions, primitives, tasks, behaviors or schemas, but from now on, all of them will be referenced as *vehicle primitives* or simply *primitives*. A mission description has to contain the vehicle primitives to be executed according to the events raised in the vehicle's control architecture and the mission goals. Then, it seems appropriate to represent this mission plan as a Discrete Event System (DES). How the user specifies this DES changes widely from one system to another, but a first division can be done depending on how the mission plan is built: either automatically using AI techniques or manually predefined by a user. In the first group procedural reasoning [Marco et al., 1996], schema-based reasoning [Turner, 1995], plan repairing [Patrón et al., 2007], Multi Agent System (MAS) [gav, 2011], Reactive Action Packages (RAP) [Firby, 1987] or general planning and scheduling techniques [Rajan et al., 2007] can be used in

order to obtain a mission plan from a given list of goals and constraints. MCSs that use AI techniques provide several advantages: the possibility to re-plan a new mission while it is under execution, the ability to undertake long missions that can hardly be predefined, a better response when dealing with unknown or changing environments or the fact of avoiding human errors when defining the mission plan are just few of them. The second main group is composed by MCSs based on predefined plans. Despite the advantages shown by deliberative MCSs, predefined plans are currently the state of the art in the underwater robotics domain. When using predefined mission plans the ultimate responsibility is given to the mission programmer who must analyze the mission requirements and write a suitable code for each possible alternative. This feature can be seen as a disadvantage because gives to the human operator the responsibility to anticipate all possible states in the mission but also as an advantage because it ensures a predictable robot behavior. In the literature, popular ways to predefine a mission plan are Domain Specific Language (DSL) that range from basic scripts [Kao et al., 1992; Nagahashi et al., 2005; Perrett and Pebody, 1997] to high-level [urb, 2011; Kim and Yuh, 2003] or visual languages [Johns and Taylor, 2008]. DSLs combine a description of primitives to execute to fulfill a mission together with the inclusion of error handling mechanisms like a maximum depth or minimum altitude values, safety or forbidden zones or alternative missions to be executed when a specific event is raised. A different approach employed to encode predefined plans is the utilization of well known formalisms to model not only the mission plan but also the framework in which the plan is going to be executed. Petri nets [Barbier et al., 2001; Barrouil and Lemaire, 1999; Caccia et al., 2005; Costelha and and, 2007; Oliveira et al., 1998] and state machines [Bohren, 2011; Boussinot and de Simone, 1991; Silva et al., 1999] are the most popular formalisms used by this purpose. The use of well known formalisms increases the complexity when encoding a mission as well as limits their expressiveness, however, simplifies its analysis and execution.

Figure 2.4 shows the main blocks that a MCS may contain. First block represents the user interaction with the system. A mission definition given by a user may include the detailed list of primitives to execute or simply a set of goals to achieve. If the latter is used, a component to plan the mission according to the user input is needed. However, many systems omit this step and is the user itself who completely defines the mission to execute. Regardless of how the mission

has been defined (using planning techniques or a predefined plan) a mission plan can be seen as a DES that sequences the primitives to execute according to some events. Once a mission is defined, there are systems which verify it before its execution. A mission verification consists, basically, with the systematically study of all the alternatives that may occur during the mission execution. Finally, the executive component has to interpret the mission plan and transform it into a sequence of vehicle primitives triggered by some kind of events raised by the vehicle's primitives. If the mission is defined by a formalism, this process is as simple as applying a set of well defined rules. If an error is detected while a mission is under execution, the executive component may ask the planning component to re-plan the mission. If the system does not provide any on-board deliberation system, then the mission plan has to include the actions to perform for all possible events.

A review of some popular MCSs, mainly in the underwater robotics domain, is presented in next section. Systems are classified based on whether they include artificial intelligence techniques to automatically plan a mission or not. Focusing on the latter, we differentiate among those that use a well known formalism to describe a mission and the ones that use DSLs or scripts.

2.3 Mission planning systems

Not many successful approaches use deliberative modules on-board an AUV to plan a mission. The ORCA architecture [Turner, 1995] is one of them. It has been designed to fulfill long duration missions where preprogrammed plans are not possible because they become obsolete and fail. Then, an on-board intelligent controller able to create and update the mission plan is needed. To implement this controller, a schema-based reasoning system is used. Given a set of goals with constraints, the environment description and few partial plans, the reasoning system tries to accomplish each goal. To this end, all the goals as well as the dependencies between them, are added in a task list. The reasoning system chooses the highest priority task and applies a procedural-schema to complete it, see Figure 2.5. Every procedural-schema consists of a set of primitives that try to complete a simple task. When the task is completed, the whole process is repeated until all goals have been successfully fulfilled. The reasoning system has a library of primitives and a library of procedural-schemes.

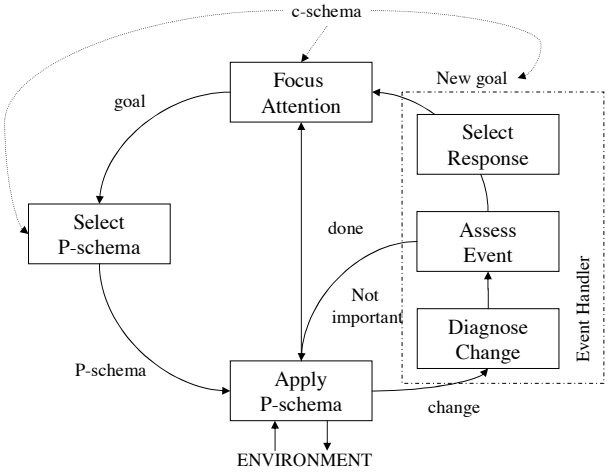


Figure 2.5: Orca: Intelligent adaptive reasoning system. Extracted from Turner [1995].

A different a hi-level software architecture with deliberative capabilities is presented by Chang [Chang et al., 2004]. It is composed by a mission level, a task level and a behavior level. Using this software architecture the AUV receives the mission file from the surface workstation. The mission management computer plans the global path according to the electronic sea chart and the restrictions of the mission file. Then, the task level plans the task sequences based on the global path and global rule databases. The task coordination module coordinates the task sequences according to clock events, hardware failure events and obstacle avoidance events. The task coordination may trigger the real-time mission re-planning when it fails to coordinate tasks. In the worse condition, the AUV might give up the former mission and re-plan the task sequences to return to the callback point. The task coordination module is also responsible for translating the task into vehicle primitives. The behavior level arbitrates conflicts between vehicle primitives or conflicts between vehicle primitives and the sea environment, and sends then the vehicle primitives parameters to the motion control computer. The mission planning procedure as well as the task coordination are modeled using Petri nets.

The Ocean Systems Laboratory in the Heriot-Watt University has also developed an architecture with planning capabilities to carry out multiple AUV missions [Evans et al., 2006]. They are using on-line plan repairing algorithms [Patrón et al., 2007] in order to obtain automated plans similar to the ones defined

by the user.

The Teleo-Reactive EXecutive (T-REX) [Rajan et al., 2007] developed by the Monterey Bay Aquarium Research Institute combines planning and scheduling techniques in a complex MCS. It is a port of the Remote Agent architecture developed by the National Aeronautics and Space Administration (NASA)'s Deep Space 1 spacecraft. T-REX is a hybrid executive for integrating primitive robot actions into higher level tasks. It aims to simplify this integration by using a model and a planner to generate necessary actions automatically where appropriate, to enable temporal constraints to be specified and enforced, and to handle recovery from failures through re-planning. T-REX uses the constraint-based temporal planning paradigm for representing and reasoning about plans. In T-REX, a time-line is a core primitive for representing all the states in the past, present and future. T-REX applies the SPA paradigm placing planning at the core of a control loop in a systematic way and embracing the divide and conquer idea to enable a planning centric system to scale well, enabling a spectrum of variously reactive and deliberative behaviors. With T-REX, not only robust, safe and compliant executions are provided but also high-level programming capabilities and goal-directed commanding through its on-board planner.

RAP [Firby, 1987], have been defined in the literature to deal with the functions done by a MCS. In RAP, each package contains a goal and a set of task nets to achieve this goal. Each task net contains basic actions or other goals that must be achieved using other packages. Planning techniques are needed to know which task net is more suitable to achieve a particular package goal. Despite RAP has not been widely used in the underwater domain, several autonomous vehicle architectures use it to specify its missions [Firby et al., 1995]. Moreover, other popular systems like the Reactive Plan Language (RPL) [McDermott, 1994], have been inspired on RAP.

2.4 Predefined mission systems

Most of the MCS available in AUVs opt for predefined missions in which the user describes all the steps to be performed instead of making use of deliberative techniques. The problem with this approach is that the user has to take into account all the alternative states that can arise during the mission execution and define a suitable response for each of them. To simplify this process, some sys-

tems provide error handling mechanisms to simplify the definition of emergency routines when a failure is detected.

In the following sections, some solutions that use predefined mission plans in their MCS are reviewed. Two categories are identified: first, systems based on scripts or DSLs and then systems based on well known formalisms. In general, systems from the first group are easier to use from an operator point of view, and allow for a higher degree of expressiveness. On the other hand, systems in the second group have to be adapted to the mission definition and execution domain and its expressiveness may be limited by the imposed formalism but provide tools to simplify its verification and execution.

2.4.1 Script and language based MCS

Numerous systems use a DSL or a script to describe the steps to carry out a mission. In order to review some of the systems presented in the literature three sub-groups have been established: those systems developed by research centers to be used by AUVs, solutions employed in commercial AUVs and, finally, generic solutions that can be used for any autonomous vehicle.

2.4.1.1 Research systems

One of the firsts AUV control architecture including a MCS was the Proteus mission executor developed in 1992 by the International Submarine Engineering (ISE). The adopted solution [Kao et al., 1992] uses an architecture based on Brooks' concept of subsumption [Brooks, 1989] where several behaviors are combined in different levels. Outputs from different independent behaviors can either be combined when they are from the same competent level (cooperation) or subsumed by other outputs from a higher competent level (subsumption). Based on this architecture they propose a MCS that runs into a level without worrying about collisions or other emergencies, since other levels are in charge of controlling these situations. To achieve task sequencing, a script is used. The script consists in an ordered sequence of operations to be performed within a particular context. Each script contains a collection of tasks organized into steps, where each step may contain any number of task threads. All the tasks in the same step start simultaneously and the step finalizes when all tasks terminate. All these steps are executed sequentially, see Figure 2.6.

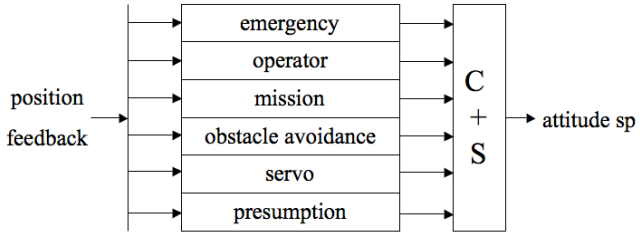


Figure 2.6: Navigation Layered Block Diagram where mission layer outputs are subsumed by the outputs from the emergency, operator or obstacle avoidance layers which are running at higher competent levels. Extracted from Kao et al. [1992].

The Naval Postgraduate School (NPS) from Monterey developed a hybrid mission control system [Marco et al., 1996] composed by three layers or levels: the strategic level, the execution level and the tactical level. The execution level is in charge of controlling the motion and stabilization of the vehicle acting in real time over the robot's actuators. The strategic level is based on a logic and procedural reasoning system that uses Prolog as an inference mechanism to execute a rule-based language in which the primitives are controlled and linked. The Prolog language describes a DES that sequences the primitives to execute in the mission plan. Finally, the tactical level joins the strategic level with the execution level. This level recollects data from the execution level to supervise the primitives execution.

The Autosub [McPhail and Pebody, 1997; Perrett and Pebody, 1997] is an AUV developed by the National Oceanography Center of Southampton (NOCS). Its MCS, named Mission Control Node, enables the vehicle to follow a 3D course through the sea. The Autosub has three main sub-control systems: propulsion and speed control, position control and finally depth and altitude control. The mission control node executes a list of commands specified in a mission script. Every command includes a mission line number, a stop flag, a start event and the time-out for the next event. When a command or a sensor rises an event, the command with this start event begins, see Figure 2.7. To activate the sub-control systems, a specific depth, orientation or velocity parameters have to be sent to one of these three modules. The execution of all commands within a script is sequential except when a *stop*, *surface* or *abort* event occurs. In this case, the current mission script is ignored and a built-in emergency control script

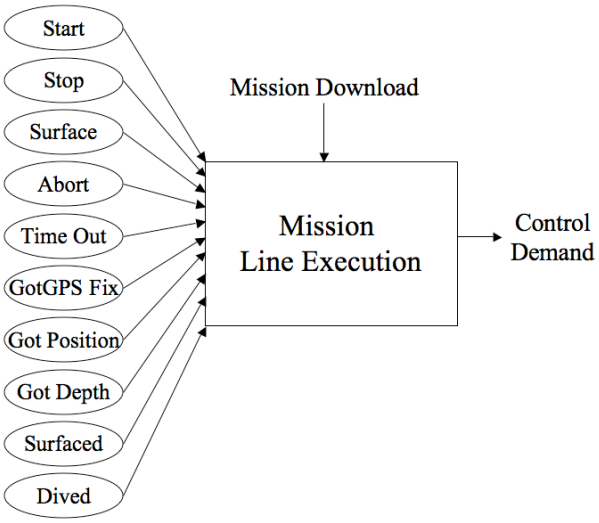


Figure 2.7: Autosub mission control node showing event inputs, mission script processor and command output. Extracted from McPhail and Pebody [1997].

is executed to bring the required response.

The Autonomous System Lab (ASL) in the University of Hawaii has designed a DSL to define autonomous missions for the underwater robot SAUVIM [Kim and Yuh, 2003]. The SAUVIM task description language is a high level language that makes use of a library of primitive actions. The library contains primitives such as motion commands, input/output commands to control specific hardware and application-specific commands such as depth or speed control. The task description language used to execute these primitives has basic utilities such as numeric operations, conditional, loop and manipulation commands, array of variables or mathematical functions. Hence, by using this language, a mission plan has the same aspect as a C program. All the situations in the mission have to be predicted, including failure situations.

In 2003, the Mitsubi Engineering Shipbuilding (MES) constructed the r2D4. The vehicle was initially developed by the Institute of Industrial Science (IIS) [Nagahashi et al., 2005] to investigate hydrothermal vent systems. r2D4 may operate in two modes: Ordinary and Relative Course Mode. In ordinary course mode the following elements must be specified by the user: the list of way-points defined by their longitude, latitude and depth/altitude and the actions used to achieve these way-points; the route to go back to the recovery point if the mission

is terminated during diving; and a roughly drawn map of depth up to which r2D4 can reach without encountering obstacles. In response to each expected trouble during the mission, several procedures are beforehand prepared taking into account the emergency return course and the safety map. In the relative course mode the r2D4 makes a thorough investigation around the environmental anomaly point detected by the payload sensors.

2.4.1.2 Commercial systems

In addition of research centers, there are companies that sell their own AUVs. These vehicles use to include a MCS to allow their operators to define and execute autonomous missions. Remus [Allen et al., 1997] is probably one of the most popular commercial AUVs. It was originally designed by the Woods Hole Oceanographic Institution (WHOI) and nowadays is manufactured and sold by Hydroid. Remus uses a program based on an ASCII mission file that contains a sequence of objectives. An objective is a task that the vehicle must complete before attempting the next objective. An entire mission is completed when the entire list of objectives is completed following its sequence. Any quantifiable goal can be an objective: reaching a particular geographic location or pressing a button on the host computer. Due to the simplicity of this script-based system, the Remus MCS can be modified. For instance in [Pang et al., 2003], a Behavior-Based Planning (BBP) used to describe missions such as chemical plume tracing or deepest point search is used instead. In this example, a set of behaviors whose priorities change depending on some factors are used to control the vehicle.

Another commercial AUV that has become quite popular is the GAVIA AUV System, developed by Hafmynd LTD [gav, 2011]. Its distributed software architecture is composed by a MAS called *intelligent artificial crew*, that comprises a full crew responsible for the safe navigation of the vessel, together with scientific personnel responsible for meeting the goals of the mission. Mission execution follows a mission plan expressed in Extensible Markup Language (XML) scripting language. The mission plan can contain fixed paths consisting of 3D way-points and 3D lines or patterns, and dynamic paths determined in real-time by an onboard scientist analyzing sensor data. Reaction to exceptional or critical operational conditions is handled by the captain of the intelligent artificial crew who, with the aid of his crew, oversees the proper running of the vehicle. This relieves the planner of foreseeing every possible situation and leaves it free to concentrate

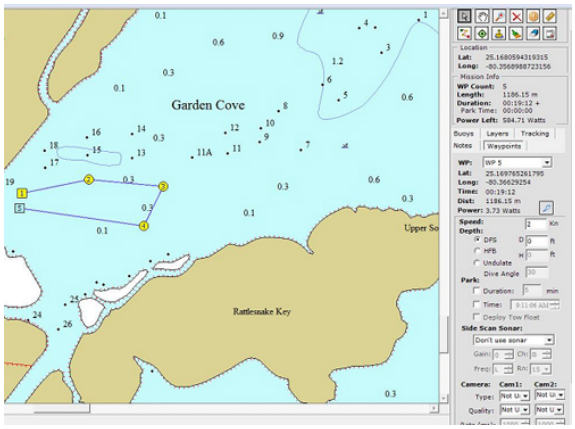


Figure 2.8: VectorMAP screen shoot. Extracted from *www.iver-auv.com*.

on the goals of the mission. Although the *intelligent artificial crew* introduces some deliberative techniques to plan trajectories or react to some errors, missions are mostly predefined using an XML scripting language.

A windows software application named *VectorMAP* is used to plan missions for the whole family of Iver [Anderson and Crowell, 2005] AUVs. Missions are created using georeferenced charts or imagery in a simple drag and drop methodology as shown in Figure 2.8. Way-points are added by a simple point and click method and the vehicle's behavior or sensor logging can be modified in every way-point. A *lawnmower* sweep can be added effortlessly by selecting the area the Iver will scan and *VectorMAP* will automatically synthesize the mission way-points. Once the mission is programed it can be easily imported in the vehicle in order to execute it.

2.4.1.3 Generic systems

Despite MCSs proposed by research or commercial institutions for an specific AUV, exists several solutions to define a mission that can be applied to any sort of autonomous vehicles. Some of these solutions are provided by standard frameworks like the *urbiscript* language that has been developed since 2003 by Jean-Christophe Baillie in the Cognitive Robotics Lab of ENSTA Paris, and now is further developed by the Gostai company together with the Urbi framework [urb, 2011]. The urbiscript language can be described as an orchestration script language. It can be used to glue together C++ components into a functional

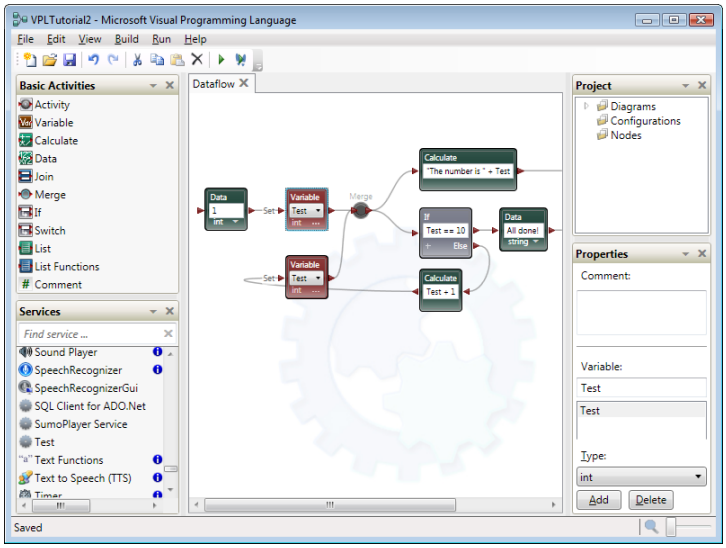


Figure 2.9: VPL screen shoot. Extracted from *msdn.microsoft.com*.

behavior, the CPU-intensive algorithmic part being left to C++ and the behavior scripting part being left to the script language which is more flexible, easy to maintain and allows dynamic interaction during program execution. As an orchestration language, urbiscript also brings some useful abstractions to the programmer by having parallelism and event-based programming as part of the language semantics. The scripting of parallel primitives and reactions to events are core requirements of most robotic and complex AI applications.

Another DSL integrated into a specific framework is the Microsoft Visual Programming Language (VPL) (see Figure 2.9). It has been developed by Microsoft for the Microsoft Robotics Developer Studio (MRDS) [Johns and Taylor, 2008]. VPL is designed on a graphical dataflow-based programming model. Rather than series of imperative commands sequentially executed, a dataflow program is more like a series of workers on an assembly line, who do their assigned task as the materials arrive. As a result VPL is well suited to programming a variety of concurrent or distributed processing scenarios. Despite it is targeted for beginner programmers with a basic understanding of concepts like variables and logic, VPL is not limited to novices. It may appeal to more advanced programmers for rapid prototyping or code development.

Lua is a lightweight multi-paradigm programming language designed as a scripting language with extensible semantics as a primary goal. Lua [Ierusalim-

schy et al., 2006] has a relatively simple C Application programming interface (API) compared to other scripting languages. It was created in 1993 by members of the Computer Graphics Technology Group (TecGraf) at the Pontifical Catholic University of Rio de Janeiro, in Brazil. It is commonly described as a multi-paradigm language, providing a small set of general features that can be extended to fit different problem types, rather than providing a more complex and rigid specification to match a single paradigm. Lua is widely used as a scripting language by game programmers. However, several attempt to use it into the robotics field has been done. For instance, a high-level decision making process for the autonomous humanoid robot Nao has been implemented using it [Niemueller et al., 2009].

Task Description Language (TDL) [Simmons and Apfelbaum, 1998] is another programming language that extends the C++ programming language to include asynchronous constrained procedures, called Tasks. It allows defining a mission and compiling it into a pure C++ file executed on the vehicle by means of a platform-independent task management library. TDL has been designed to simplify the function control at the task-level. For this purpose, TDL applies highly non-linear code that otherwise would be difficult to understand, debug and maintain using pure C++. The TDL is based on task trees that can be dynamically modified. Each node corresponds to a basic action or a goal that is expanded using other task trees.

2.4.2 Formalism based

MCS based on formalisms can be divided in two major groups. The first one, uses a formalism to define a set of states, in which actions are executed, and the flow between these states, when specific events are raised. These systems are similar to DSL systems. The main difference is that the imposed formalism reduces the expressiveness of the language, for instance, DSLs allow to define emergency missions, safety depth and altitude values or forbidden zones, while on the other hand, formal methods allow a simpler execution and reachability analysis. The second group of formal MCSs use the chosen formalism, not only to define the mission to execute but to model the vehicle architecture and the mission as a whole. Then, the vehicle components, the data flow or even the sequence of primitives to execute are modeled in a single DES. This model can be analyzed

to prove state reachability, resources availability or deadlock avoidance among others.

2.4.2.1 Formal mission description

The solution developed in the ISE [Kao et al., 1992] has been presented as a scripting MCS. However, it could also be considered as a formal mission description system because it is possible to relate the scripts used to define a mission with a formal FSM.

SMACH [Bohren, 2011], which stands for *State MACHine*, is a Python library to build hierarchical state machines. SMACH is useful to execute complex plans in which all possible states and state transitions can be explicitly described. It provides fast prototyping of complex state machines, however, it is not recommended for unstructured tasks or low-level systems that require high efficiency. SMACH is integrated in the Robot Operating System (ROS), and hence, ROS components can be executed from SMACH states by means of a standard interface for preemptive tasks named *actionlibs*. User data can also be passed between different SMACH states. SMACH does not provide any tool for verification or analysis purposes. However, generated hierarchical state machines can be evaluated from any third part FSMs verification software. Figure 2.10 shows and example of a hierarchical state machine encoding the steps to perform in order to autonomously recharge a vehicle.

The cross platform software for robotics Mission Oriented Operating Suite (MOOS) [Newman, 2005] includes also its own module to define and execute predefined missions: the *Helm*. MOOS has been used to develop several architectures for AUVs [Eickstedt and Sideleau, 2009; Schneider and Schmidt, 2010]. The Helm job is to take navigation data and, given a set of mission goals, decide the most suitable actuation commands. The multiple mission goals take the form of prioritized tasks within the Helm. Helm is designed to allow at sea reloading of missions. This makes for very rapid turnaround of missions in a research-oriented field trip. The Helm has two modes: on-line and off-line. When off-line no tasks are run and the Helm makes no attempt to control actuators. In this mode the vehicle can be teleoperated. When on-line, the vehicle motors are under the control of the Helm. Mission goals are accomplished by deciding which vehicle actions should be undertaken. To accomplish these goals, the Helm has a mission plan described by a set of prioritized primitive tasks. The sequential

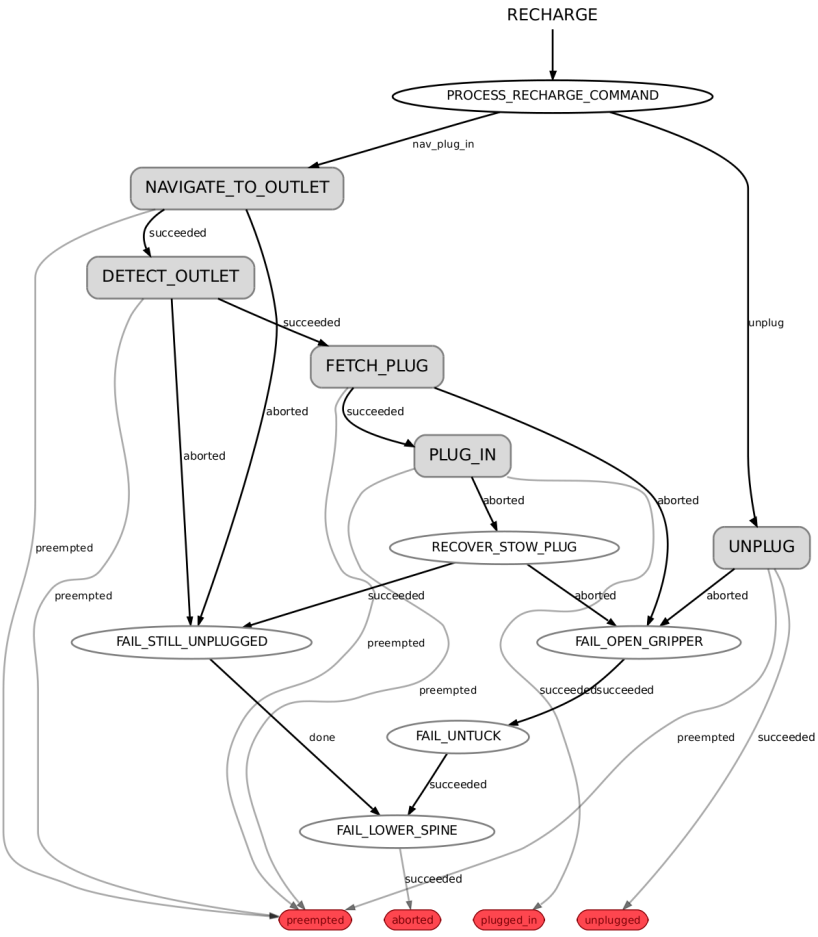


Figure 2.10: SMACH mission example. Extracted from *www.ros.org*.

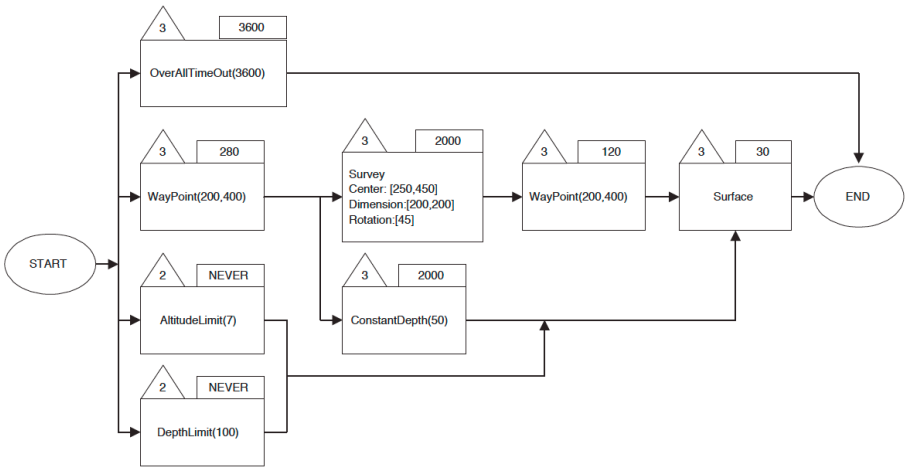


Figure 2.11: Example of a typical mission plan in Helm over MOOS. Extracted from Newman [2005].

execution of these primitives is achieved using messages. Several primitives can be executed simultaneously. Every primitive needs an activation message to start its execution and generates a message when it finishes. A primitive can finalize when its corresponding goal has been fulfilled, when a time-out has expired, or when it does not receive incoming data from its related objects. The finalization message can activate zero, one, or more primitives. Also, other object messages can activate a primitive. When two or more primitives try to use the same resource, the primitive with the highest priority wins. All these rules, primitives and interfacing methods allow the definition of a mission by means of a network of tasks that can be approximated to a marked graph as shown in Figure 2.11.

A well known system used to describe a DESs is Esterel [Boussinot and de Simone, 1991]. Its development started in the early 1980s, and was mainly carried out by a team of Ecole des Mines de Paris and Institut National de Recherche en Informatique et en Automatique (INRIA). Esterel is a synchronous programming language for the development of complex reactive systems. The imperative programming style of Esterel allows the simple expression of parallelism and preemption. As a consequence, it is very well suited for control-dominated model designs. Esterel could be used to model the mission as well as the vehicle control architecture. However, despite it has been reported in several papers [Anisimov et al., 1997; Nana et al., 2005] as a synchronous missions programming system suitable for AUVs that follow the model checking approach, authors are not aware

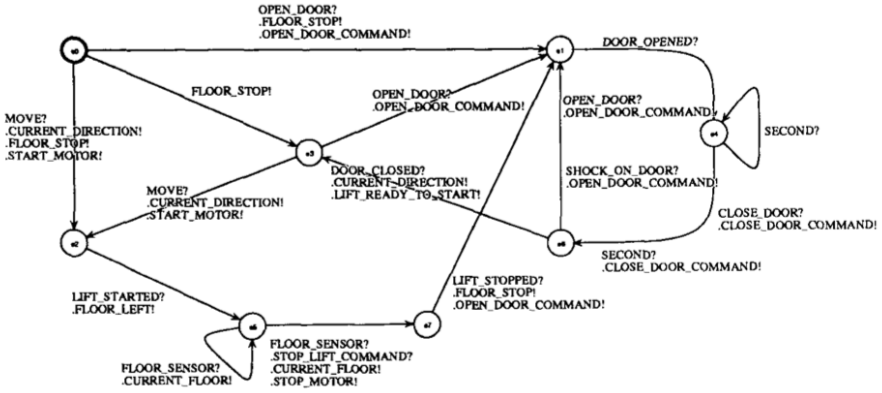


Figure 2.12: Lift automaton produced by Esterel compilation. Extracted from Boussinot and de Simone [1991].

of any AUV using it to define their missions. Esterel formal approach allows to compile programs into efficient FSMs, see Figure 2.12, and easily verify and execute the automatically obtained FSMs. Inspired with Esterel there are other formally defined synchronous programming languages like Lustre [Halbwachs et al., 1991]. Lustre is a data-flow language for programming reactive systems like the industrial Software Critical Application Development Environment (SCADE).

2.4.2.2 Formal mission and framework description

Some MCSs not only describe the mission to execute as a DES but also models the primitives to be executed, the control flow among them and the resources that primitives may need. In a vehicle's control architecture, typical resources to model are the access to sensors or actuators as well as the availability of some variables that are provided by other components.

A popular system implementing this approach is the Marius' control architecture, named Coral [Oliveira et al., 1998] developed in 1994 by the Institute for Systems and Robotics (ISR) from Lisbon, Portugal. It has several systems in charge of controlling some vehicle capabilities like the acoustic communications, an obstacle detector and a navigator. Moreover, a mechanism based on message exchanging is used for communication purposes. To define a mission, vehicle primitives are used to activate and synchronize one or more systems. However, these vehicle primitives can not be executed arbitrarily: there are some preconditions that have to be accomplished in order to be enabled and, once executed,

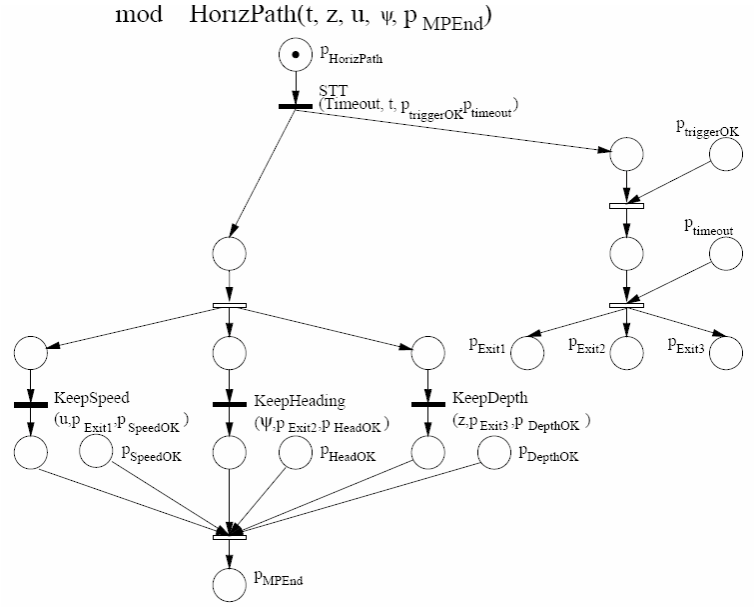


Figure 2.13: Mission Procedure described in Coral. Extracted from Oliveira et al. [1998].

they activate some post-conditions. Therefore, to define a mission, a valid sequence of vehicle primitives have to be enchained. Petri nets are the formalism selected to describe these systems, the primitives and the flow among primitives as shown in Figure 2.13. Some time later [Oliveira and Silvestre, 2003], the way in which missions were defined using Coral was enhanced. The proposed solution mediates the interaction with real-time systems on board a vehicle and/or between participating vehicles in the same mission scenario. A temporal module is used to monitories a model for each vehicle primitive and if the primitive model becomes unresponsive, the temporal module aborts it and an emergency mission is executed instead.

In the CNR-ISSIA, Italy, there is another similar approach, also based on Petri nets, designed to control an underwater robot [Caccia et al., 2005]. The presented architecture is a transfer in to the marine robotics field of the concept of execution control levels originally introduced by R. Alami in 1998 [Alami et al., 1998]. This architecture uses Petri nets to describe the execution flow as well as to model sensors and controllers, see Figure 2.14. When a task requires some sensors or actuators, the Petri net that describes its behavior tries to enable first the desired sensors/actuators and then runs the task. Similarly, if a value can be

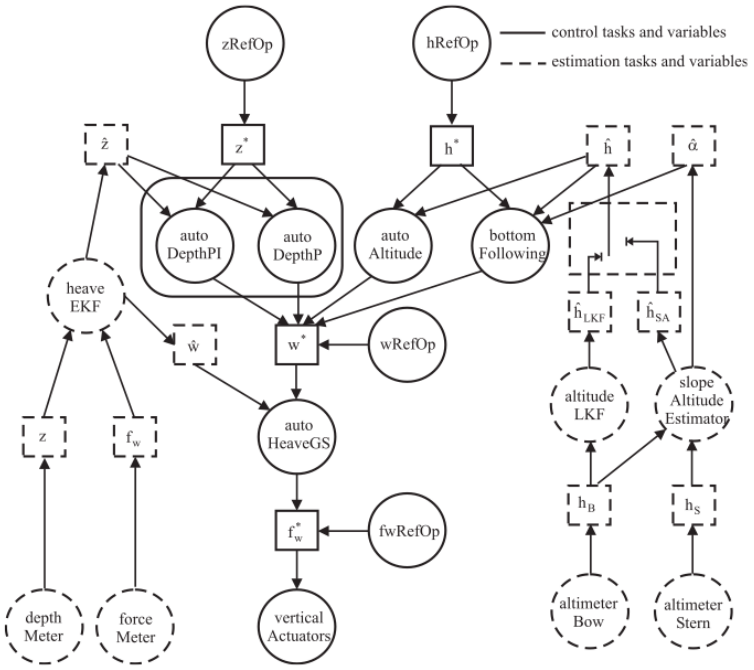


Figure 2.14: Romeo vertical motion control system. Extracted from Caccia et al. [2005].

estimated by different systems, a demultiplexer built using a Petri net structure, allows to connect all the possible versions of this value with the input of the task that requires it. A Supervision Based on Place Invariant (SBPI) [Moody and Antsaklis, 1998] is used to check that a task can be only in three states: initialized, running or idle. Moreover, SBPIs are used to avoid or to allow the simultaneous write/read of variable values. Once the whole system is defined as a Petri net, an unfolding algorithm [McMillian, 1995] plans the actions to perform in order to transform the current Petri net state into the goal state. The architecture was later extended with an event handling mechanism to allow an operator to interact with the mission evolution [Bibuli et al., 2007].

Not all the systems that model the mission to execute as well as its execution framework are based on Petri nets. In LSTS/FEUP [Silva et al., 1999] the TEJA CASE [Deshpande and de Sousa, 1997] tool is used to graphically model the dynamic behavior of the software architecture using an hybrid state machine paradigm. When the whole application is modeled, C++ code is automatically generated from this model. The architecture is divided in three layers:

abstraction layer, functional layer and coordination layer. The abstraction layer provides low level access to the devices (sensors and actuators). Functional layer includes primitives and navigation algorithms that do no directly depend on the available hardware. Finally, the coordination layer is used to accomplish correct mission execution and tolerance to unexpected events. A mission file describes the response of the coordination layer to each triggered event. Those responses are implemented using actions available in the functional layer. There are also emergency primitives to guarantee the ultimate robustness.

The approach for the mission control sub-system proposed in [Champeau et al., 2000] is to formalize an object framework based on the use of formal languages for the primitive design. The Unified Modeling Language (UML) formalism, is used to define the sequence diagram for the mission expression as well as the class diagram for the description of the objects integrated in the mission and the objects used on the embedded platform. Then, the Specification and Description Language (SDL) formalism is used to formalize the behavior of the object and support formal verification to increase the trust in the embedded code. From this point of view, a mission is a set of objects which exchange messages and each message reception triggers an action defined by a method in the object class. The static part of this real time object model is described using the UML formalism while the SDL formalism is used for the behavior of the object. Once a mission is defined, it is possible to automatically build the state space from the SDL formalism in order to check deadlocks, infinite loops or if any queue length is exceeded.

Finally, there are solutions that combine the use of a formalism like the Petri nets with a script-alike approach with handling error mechanisms. For instance, this is the case of the Redermor [Barbier et al., 2001; Barrouil and Lemaire, 1999], a French AUV used in military applications mainly for inspection and mine recovery missions. Redermor uses a programming and a monitoring execution system called ProCoSA. The system is composed by a Graphic User Interface (GUI) to build the Petri nets in which places represent behavioral states and transitions represent actions, events and messages. A Petri net player automata, developed in Lisp, is also included. The actions to be performed by the autonomous vehicle are implemented then by independent sub-systems. The communication between the Petri net player and these actions is socket based. To specify a mission, the user has to provide a set of way-points in 3D, its priority, a time-out and

the process to be executed at each point. Some constraints like vehicle autonomy limitations, forbidden zones, maximum and minimum depth limits and fixed obstacles have to be provided too. Using all this data, a planner selects an optimized itinerary and a guidance navigator generates high level orders to follow this optimal path while avoiding obstacles. The State estimator computes the vehicle state using the information provided by the map processor and generator which is in charge of analyzing the sonar images. Finally, the mission controller supervises the phases of the mission and the correct execution of all procedures.

2.5 Summary

In order to summarize all reviewed systems Figure 8.7 presents a table including the most relevant information from each one. Outlined elements are:

- The architecture model in which the MCS is included.
- The primary missions to execute.
- How autonomous missions are described.
- The availability of on-board deliberation systems.
- The presence/absence of error handling mechanisms to simplify the mission description.
- The language/formalism used to describe a mission plan.
- The inclusion of verification techniques.

2.6 Survey conclusions

Once finished the review of some control architectures and their MCSs found in the literature, several conclusions can be drawn in order to establish the guidelines for developing a new MCS. First of all, lets summarize the main alternatives and highlight their strengths and weaknesses.

Vehicle/Architecture	Institution	Year	Type of architecture	Primary missions to execute	How the mission is described	On-board deliberation	Error Handling Mechanisms	Mission codification	Verification	Comments	Cite
Proteus Mission Executor	ISE	1992	Subsumption	Exploration	Script	No	No	Marked graph	No	Use a subsumption architecture	*Kao et al. 1992*
MARIUS/CORAL	IST/ISR	1994	-	Scientific	Petri net GUI	No	Error Missions	Petri net	Yes	Used in scientific missions. Includes mission verification	*Oliveira el at. 1998b*
Phoenix	NPS	1996	3-layers	Scientific/Military	Prolog	Procedural Reasoning	No	Prolog/Petri nets	-	3-layers architecture used in Scientific and military missions	*Marco et al. 1996*
Autosub	NOCS	1997	-	Scientific	Script	No	Built-in emergency control scripts	FSM	-	Used in scientific missions	*McPhail and Pebody 1997*
-	LSTS/FEUP	1999	3-layers	-	Hybrid State Machines GUI	No	Emergency Behaviors	C++ code	Yes	3-layers architecture. Includes mission verification.	*Silva and Martins 1999*
Redermor/ProCoSA	-	1999	-	Military	Petri net GUI/Lisp	Path Planning	Constraints	Petri nets (high level orders)	Supervision while executing	Used in military missions	*Barbier et al. 2001a*
Simulated Vehicle	ENSIETA	2000	Object-based	-	UML and SDL models	No	-	-	Checks the state space from the SDL models	Verifies the state space from the SDL models	*Champeau et al. 2000*
SAUVIM	ASL	2003	3-layers	Scientific/Military	DSL (Task description language)	No	No	C++ code	-	3-layers architecture	*Kim and Yuh 2003*
r2D4	IIS	2003	-	Scientific	Script	Path planning	Emergency Error System + Constraints definition	-	-	Used in scientific missions	*Nahohashi et al. 2005*
Simulated Vehicle	Harbing enginering university	2004	3-layers	Scientific	Global path + restrictions	On-board planner	Replanning	Petri nets	-	3-layers architecture used in scientific	*Chang et al. 2004*
Generic Vehicle/MOOS	Oxford Mobile Robotics Group	2005	Component-based	Scientific/Military	Graph of priorized primitives	No	No	State Machine	No	Used in scientific and military missions	*Newman 2005*
Romeo	CNR-ISSIA	2005	-	Scientific	Petri nets	Petri net unfolding planner	-	Petri nets	Yes (SBPI)	Used in scientific missions. Verification through SBPIs	*Caccia et al. 2005*
EAVE III/ORCA	University of Maine	2005	Hybrid	Scientific	Plan/Goals	Context-Sensitive adaptive reasoning	Plan adaptation	-	-	Deliberative architecture used in scientific missions	*Turner 1995*
Nessie/Ocean Shell	OSL HWU	2006	Hybrid	Scientific/Industrial	Plan/Goals	On-board Planning	Plan repairing	-	-	Deliberative architecture used in scientific missions and industrial missions	*Evans et al. 2006*
Generic Vehicle/T-REX	MBARI	2007	Hybrid	Scientific	Plan/Goals	On-board Planning & Scheduling	Replanning	-	-	Deliberative architecture used in scientific missions	*Rajan et al. 2007*
Remus	WHOI	2001	-	Scientific/Industrial /Military	Script	No	No	State Machine	No	Can be ussed for multipurposse missions	*Allen et al. 1997*
Gavia	Hafmynd LTD	1997	-	Scientific/Industrial /Military	XML Scripting Language	On-board MAS	MAS	-	-	Can be ussed for multipurposse missions	*gav 2011*
Iver	Ocean Server	2003	-	Scientific/Industrial /Military	GUI	No	Constraints	State Machine	No	Can be ussed for multipurposse missions	*Anderson and Crowell 2005*
Generic Vehicle/SMACH	WillowGarage/ROS	2010	ROS based architecture	-	Script	No	No	Finite State Machine	No	ROS based architecture	*Bohren 2011*
Lua	TecGraf	1993	Architecture Independent	-	Script	No	No	Finite State Machine	No	Architecture Independent	*Ierusalimschy et al. 2006*
Urbiscript	Gostai	2003	Urbi based Architecture	-	DSL	No	No	-	No	Urbi based Architecture	*urb 2011*
VPL	MRDS	2008	MRDS based architecture	-	GUI	No	No	-	No	MRDS based architecture	*Johns and Tylor 2008*
Esterel	Ecole des Mines de Paris and INRIA	1980	Architecture Independent	-	Script	No	No	Finite State machine	Yes	Architecture Independent, automatically generated automatas can be verified	*Boussinot and de Simone 1991*
Task Description Language	Carnegie Mellon University	1998	3-layers Task Control Architecture	-	DSL (C++ extension)	Ttask tree expanding	Exception handling	Task Trees coded in C++		3-layers Task Control Architecture	*Simmond and Apfelbaum 1998*
RAP	Yale	1987		-	Plan/Goals	Reactive planning and execution	Exception handling	Task nets inter-preted trhough Lisp		-	*Firby 1987*

Figure 2.15: Summary table.

- MCS based on AI techniques are very versatile. They allow to define a mission with just few lines (specifying only the goals) and to take care of errors or unpredictable events automatically by means of on-board deliberative techniques. On the other hand, they are more complex and make difficult to control the vehicle's final behavior as well as to verify generated plans before its execution.

- MCS based on predefined plans described by DSLs or scripts present a good trade off between simplicity of use and functionalities. The inclusion of error handling mechanisms increases its expressibility at expenses of difficulting its interpretation (execution) and verification.

- Formalism based MCS are probably the hardest to use. Describing a mission using a formalism can be a hard work and error-prone. Despite GUIs may simplify this process, the absence of error handling mechanisms or automatic deliberative techniques implies that all states that can occur during a mission have to be considered in advance. On the other hand, the capability of not only modeling the mission but also parts of the control architecture together with the use of well known formalisms allows for a more complete and systematic verification as well as simplifies the mission execution.

Therefore, a MCS that takes advantage of the strengths identified in the above summary should include:

- An unified model in which not only the mission plan is modeled but also the framework in which the mission will be executed. This model allows to verify the mission reachability, the availability of resources and the presence of deadlocks. In the literature, formalisms like FSMs or Petri nets are presented as appropriate ways to model and analyze a mission plan and its execution framework.

- In order to deal with unlikely situations it is worth to simplify how responses for unexpected events are provided. Having a systematic way to include the possible alternatives that can be produced during a mission plan or the inclusion of error handling mechanisms that simplifies the definition of these alternatives may help. Systems based on planning algorithms are the ones

who best deal with this problem. However, their non predictable behavior, the amount of time needed to generate each new plan and their complexity makes them not always the best option. Predefined plans based on DSL have shown to be a trade off between formalism based missions and complex deliberative systems.

- DSLs must provide not only sequential, iterative or conditional execution of primitives but also structures to capture errors, simplify parallelism and allow the preemptive execution of primitives. It is worth noting the inherent parallel structure of autonomous missions.

Then, our proposal is to develop a MCS based on a DSL providing sequential, iterative, conditional, parallel and preemptive execution of primitives and, furthermore, that could be automatically translated into a well known formalism. Thus, even though the mission plan will be described by means of a formalism that can be systematically analyzed and executed, the human operator will deal with a more friendly DSL that simplifies parallelization as well as error handling mechanisms. The use of on-board deliberative systems have been proved effective in several situations, however, their study is out of the scope of this dissertation. Then, only the interface with a deliberative system will be studied.

2.6.1 The Petri net formalism

The approach presented in this thesis, models the components involved in the execution of a mission as well as the mission itself using a single formalism, thus, being easy to analyze the resulting system and to check several properties. Elements typically checked in a mission are: the reachability of final states, the absence of deadlocks in the mission or the availability of resources. However, as can be seen in Chapter 4, our approximation to the verifying problem is not the usual one: instead of defining a mission and then check if several properties hold, these properties are first checked for small blocks that compose the mission so the resulting mission of connecting these blocks inherently holds the same properties. This approach allows to perform a reachability analysis to each small DES, representing a primitive or a control structure, instead of the final DES encoding the whole mission which is usually large and requires much time to be analyzed.

Therefore, according to our propose a formalism has to be first choosen and then a DSL has to be defined in accordance with the constraints imposed by this formalism. FSMs are the most popular and widely known way to formally encode a DES. However, Petri nets present some advantages compared with them: Petri nets are more compact, are better suited to express parallelism and also can represent a larger class of languages than standard FSMs [Murata, 1989]. Moreover, several precedents in the underwater robotics literature [Barbier et al., 2001; Barrouil and Lemaire, 1999; Caccia et al., 2005; Oliveira et al., 1998] as well as in other domains [Costelha and and, 2007; Ziparo and Iocchi, 2006] point us to use Petri nets to define a formal MCS.

Then, Petri nets seems to be the more adequate formalism to encode and execute a mission for an autonomous vehicle. However, there are several types of Petri nets that can be used for this task. We started testing marked graphs, a subgroup of Petri nets in which each place p have exactly one input transition and one output transition [Palomeras et al., 2006a]. Marked graphs allow concurrency but not conflict, and for this reason, are easy to use but it is difficult to model concepts as preemptive execution or some complex control structures. Therefore, we changed to ordinary Petri nets that allow concurrency and conflict symmetrically and asymmetrically. We have also take advantage of timed transitions to model time-outs as well as to delay the firing of some transition. In the proposed framework, timed transitions may represent 2 different things: a transition that once has been enabled its firing is delayed by a timer or a transition that once has been enabled their firing depends on the completion of another process, generally continuous. The latter, should be modeled by an hybrid Petri net, however, to simplify the whole framework, they are modeled as timed transitions in which the time that passes from its enabling until its firing is unknown. Appendix A presents an introduction to Petri nets as well as the main algorithms applied to verify, supervise and execute them.

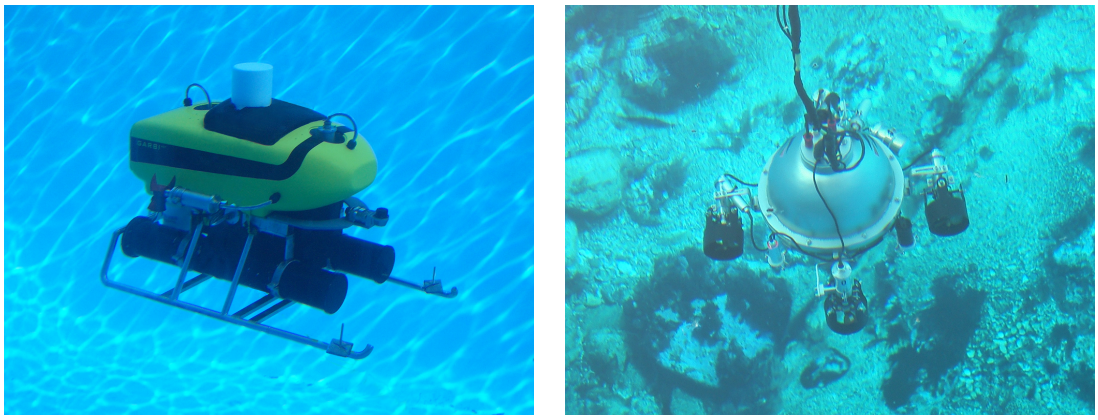
Chapter 3

Experimental platform

The experimental platform introduced in this chapter encompasses all aspects in which the proposed Mission Control System (MCS) has been developed and tested. The MCS is the set of tools within the vehicle's control architecture that are implemented to define, verify and execute a mission. Although one of our objectives is to define a MCS as generic as possible, to obtain experimental results the proposed MCS has been implemented in the context of a tangible control architecture. Therefore, this chapter will discuss about the control architecture where the proposed system is integrated as well as the vehicles that operate with this architecture.

3.1 Vehicle experimental platforms

In 1995 the University of Girona (UdG) built its first Unmanned Underwater Vehicle (UUV) in collaboration with the Universitat Politècnica de Catalunya (UPC). The vehicle was called Garbi [Amat et al., 1996] and was conceived as a Remotely Operated Vehicle (ROV). In 2005 the vehicle was rebuild and converted into Garbi AUV, as shown in Figure 3.1(a). This vehicle uses four thrusters and includes a simple sensor suite: 2 compasses, 2 pressure sensors a water speed sensor and water leakage sensors. Garbi dimensions are: 1.3m long, 0.9m high and 0.7m wide with a maximum speed of 1 knot and a weight of approximately 150Kg. In 2001 a smaller UUV was also designed with the aim of developing a light weight, low cost Autonomous Underwater Vehicle (AUV) to be used as a research platform in a water tank testing facility. This spherical shaped vehicle called



(a) (b)

Figure 3.1: (a) Garbi AUV and (b) Uris AUV.

Underwater Robotic Intelligent System (URIS) [Batlle et al., 2004] incorporates 4 thrusters, a magnetic compass, a pressure sensor, water speed sensors, Differential Global Positioning System (DGPS), water leakage sensors and a computer vision system. Its radius is about 35cm and the weight is approximately 35kg. URIS is shown in Figure 3.1(b).

The experience gained in the group with the development of these previous vehicles made it possible to build two new AUVs of reduced weight and dimensions with remarkable sensorial capabilities and easy maintenance. These two vehicles, *Ictineu* AUV built in 2006 and *Sparus* AUV built in 2010, are the main test platforms of this thesis.

3.1.1 Ictineu

Ictineu AUV, shown in Figure 3.2, is the result of a project started in 2006. During the summer of that year, the Defence Science and Technology Lab (DSTL), the Heriot-Watt University and the National Oceanographic Center of Southampton organized the first Student Autonomous Underwater Challenge-Europe (SAUC-E), an European wide competition for students to foster research and development in underwater technology. Ictineu AUV was originally conceived as an entry for the SAUC-E competition by a team of students collaborating with the Computer Vision and Robotics (VICOROB) group in the UdG [Ribas et al., 2007]. Although the competition determined many of the vehicle's specifications, Ictineu was also

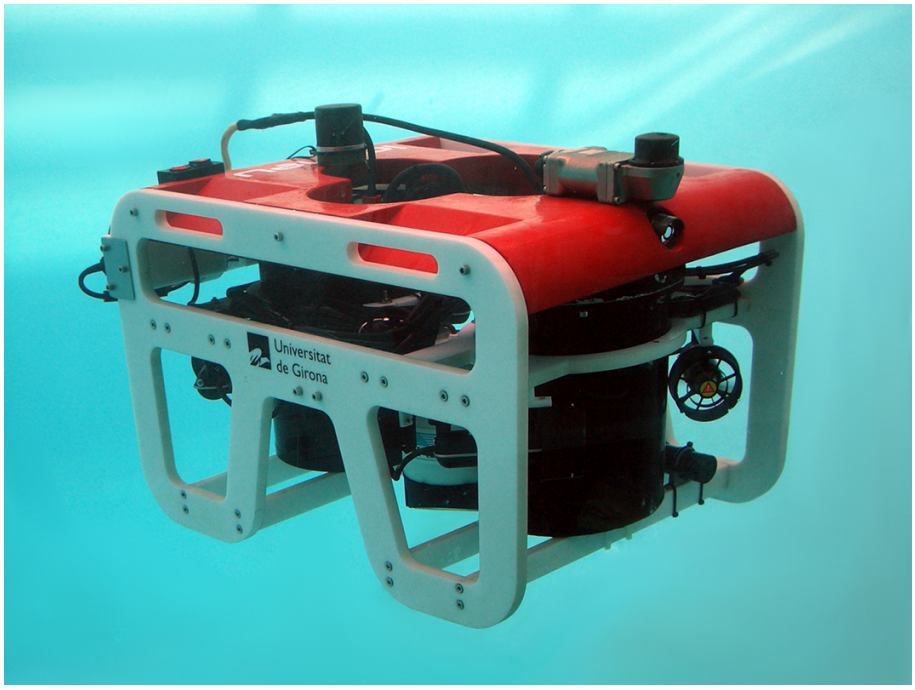


Figure 3.2: Ictineu AUV.

designed keeping in mind its posterior use as experimental platform for various research projects in our laboratory.

Ictineu AUV was tailored around a typical open frame design. This configuration has been widely adopted by commercial ROVs because of its simplicity, toughness and reduced cost. Although the hydrodynamics of open frame vehicles is known to be less efficient than a closed hull type vehicles, they are suitable for applications not requiring movements at high velocities or traveling long distances. The robot chassis is made of Delrin, an engineering plastic material which is lightweight, durable and resistant to liquids. Another aspect of the design is the modular conception of its components which simplifies upgrading the vehicle and makes it easier to carry out maintenance tasks. Some of the modules (the thrusters and most of the sensors) are watertight and therefore, are mounted directly onto the vehicle chassis. On the other hand, two cylindric pressure vessels made of aluminum house the power and computer modules while a smaller one made of Delrin contains a Motion Reference Unit (MRU). Their end-caps are sealed with conventional O-ring closures while the electrical connections with

other hulls or external sensors are made with plastic cable glands sealed with epoxy resin. Ictineu is propelled by six thrusters that allow it be fully actuated in Surge (movement along X axis), Sway (movement along Y axis), Heave (movement along Z axis) and Yaw (rotation around Z axis) achieving a maximum speeds of 3 knots. It is passively stable in both Pitch and Roll Degree of Freedom (DOF) as its meta-center is above the center of gravity. This stability is the result of an accurate distribution of the heavier elements at the lower part of the frame combined with the effect of technical foam placed in the top, which provides a slightly positive buoyancy to the vehicle.

One of the main objectives of the laboratory was to provide the underwater robot with a complete sensor suite. The robot includes a Tritech Miniking Mechanically Scanned Imaging Sonar (MSIS) designed for use in underwater applications such as obstacle avoidance and target recognition. Also, the robot is equipped with a SonTek Argonaut Doppler Velocity Log (DVL) specially designed for applications which measure ocean currents, vehicle speed over ground and as an altimeter using its 3 acoustic beams. The particular spatial distribution chosen to place the acoustic sensors within the vehicle frame avoid dead zones, improving their overall performance. Moreover, Ictineu AUV has a compass which outputs the sensor heading (angle with respect to the magnetic North), a pressure sensor for water column pressure measurements and a Xsens MTi low cost miniature Attitude and Heading Reference System (AHRS) which provides a 3D orientation (attitude and heading), 3D rate of turn as well as 3D acceleration measurements. Finally, the robot is also equipped with two cameras. On one hand, a forward-looking color camera, mounted on the front of the vehicle and, on the other hand, a down looking Tritech Super SeaSpy color Charge Coupled Device (CCD) Underwater Camera, located in the lower part of the vehicle. The latter is mainly used to capture images of the seabed for research on image mosaicking while the former is intended for target detection and tracking, inspection of underwater structures and to provide visual feedback when operating the vehicle in ROV mode. Nowadays, Ictineu AUV is used as a research platform for different underwater inspection projects which include dams [Ridao et al., 2010], harbors, shallow waters and cable/pipeline inspections [El-Fakdi et al., 2010].

3.1.2 Sparus

Sparus AUV, shown in Figure 3.3, is the second experimental platform used in this thesis. It was built by a group of students in the UdG to face the 2010 edition of SAUC-E competition and was devised with the main goal of having a small and simple torpedo-shaped vehicle with hovering capabilities.

Sparus is designed with one vertical thruster for the Heave DOF and 2 horizontal thrusters for the Surge and Yaw DOFs. The 3 thrusters, supplied by Seabotix, are integrated in a classical torpedo shape AUV. The vertical thruster is placed in the center of the vehicle, where the center of gravity and buoyancy are located. The horizontal thrusters are placed in the back, separated from the longitudinal axis of the vehicle to generate a torque for the Yaw DOF. The mechanical structure and components are therefore organized around this configuration. The front of the vehicle contains all the sensors and the battery housing. The back of the robot contains a second housing for the electronics, computer and inertial navigation system. Having the batteries in a separate housing increases the weight, length and expense of the vehicle but on the other hand it minimizes the downtime between missions by allowing battery packs to be quickly interchanged. Also, potentially explosive gases that can build up from the batteries do not interfere with sparking and high temperature electronics.

The main structure is made of aluminum profiles and stainless steel clamps that hold the two pressure housings. The battery housing is hold with only one clamp in the top, to allow its easy and fast replacement for a second battery housing. The housings were made of aluminum which easily transmits the internal heat to the environment, it is easy to machine and it is strong enough to withstand the sea pressure (they were designed for 100m depth). In order to eliminate screws, a 1.8mm nylon thread before the O-ring secures the end-caps. The electronics housing can be easily opened from the back of the vehicle, allowing the access to the electronics and computer in few minutes.

To give the vehicle the required buoyancy, technical foam is distributed all over the top part of the robot. It is strategically located to place the buoyancy center at the same longitudinal position as the gravity center but above it, ensuring pitch and roll stability. The vehicle is trimmed with lead weights, in order to bring the gravity center where the vertical thruster is placed. Small zinc anodes were added to eliminate the corrosion due to sea water and different metals on the vehicle. Finally, to reduce the water drag and to protect the components, a

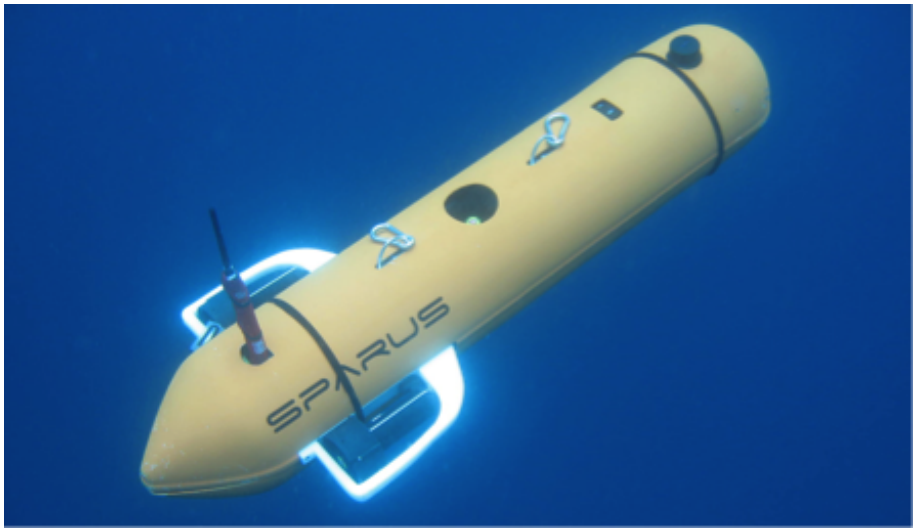


Figure 3.3: Sparus AUV.

two-part ABS skin covers the AUV.

For connecting all the external components, two types of underwater connectors were used: *Subconn*® for the high current connections and *Lumberg*® for the low current parts. Also, the umbilical cable, which is often connected and disconnected, uses a *Subconn*® connector. The main battery switch is IP68 rated and covered with resin. The WiFi adapter, also covered with resin, is placed on the top of the vehicle. It also has the option to be detached to float on the surface while keeping the connection with the AUV through a 5 meter Universal Serial Bus (USB) cable. The final dimensions of the vehicle are 1.22m length by 0.23m diameter, and the weight is around 30kg.

The on-board embedded computer has been chosen as a trade-off between processing power, size and power consumption. An Ultra Low Voltage (ULV) Core Duo processor with the 3.5" small form factor was selected. The vehicle is also equipped with a complete sensor suite composed by two color video cameras (forward-looking and down-looking), a AHRS MTi from XSens Technologies, a Micron imaging sonar from Tritech, an echo-sounder, a pressure sensor and a DVL from LinkQuest which also includes a compass/tilt sensor. Temperature, voltage and pressure sensors as well as water leakage detectors are installed into the pressure vessels for safety purposes. The on-board computer, the sensors and the three thrusters are powered by two battery packs. The first one, at

12V, powers the computer, the electronics and the sensors while the second one, at 24V, provides the power to the thrusters. Each battery pack has 10Ah of capacity, which allows for an autonomy of 2.5 hours.

3.2 COLA2 architecture

The architecture used in all the underwater vehicles available at UdG was the Object Oriented Control Architecture for Autonomy (O2CA2) [Ridao et al., 2002]. The O2CA2 was a behavior-based control architecture [Brooks, 1986] that implemented a reactive layer in which a set of behaviors were able to perform some specific tasks. With the inclusion of a coordinator [Carreras et al., 2001], it was possible to enable several of these behaviors simultaneously in order to perform a more complex tasks. However, traditional behavior-based architecture limitations appeared when trying to undertake long-range missions. To transform this reactive architecture into a layer-based architecture, a MCS implementing the functionalities of a deliberative and an execution layer has been developed. The MCS is an independent set of components that can be connected with the O2CA2 reactive architecture but also with any other reactive architecture/layer implemented by an autonomous vehicle or manipulator [Palomeras et al., 2010a]. The new layer-based control architecture, named Component Oriented Layer-based Architecture for Autonomy (COLA2), has been implemented in Ictineu AUV and Sparus AUV. It is based on software modules that encapsulate a set of related functions or data named components. Components may exist autonomously from other components in a network node having the ability to communicate with each other. The architecture is organized in three layers following the hybrid model [Arkin and Balch, 1997; Firby, 1989]: the *mission* layer, the *execution* layer and the *reactive* layer. The mission layer obtains a mission plan by means of an on-board automatic planning algorithm or compiling a high-level mission description given by a human operator. The mission plan, described using the Petri net formalism, is then interpreted by the execution layer. It is executed by means of enabling/disabling the vehicle primitives contained in the reactive layer, see Figure 3.4. The proposed MCS implements the first two layers, mission and execution, while the reactive layer is implemented in our vehicles as a new version of the O2CA2.

To simplify the development of each component in the COLA2, an unified

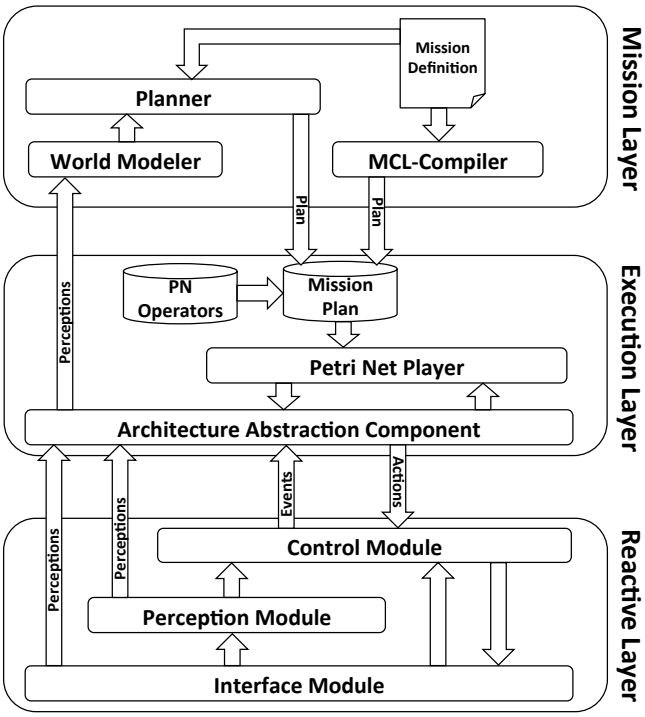


Figure 3.4: Three-layer organized control architecture.

framework has been created to simplify its programming as well as the communication among all the components. Therefore, COLA2 has been developed using the custom framework that is detailed next.

3.2.1 Generic and custom frameworks for developing control architectures for autonomous vehicles

A software framework is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality [Riehle, 2000]. Frameworks have distinguishing features that separate them from libraries or normal user applications. Some of these features are:

- The overall program's flow control is not dictated by the application implemented using the framework but by the framework itself.
- Frameworks have a default behavior that must actually be a useful behavior.

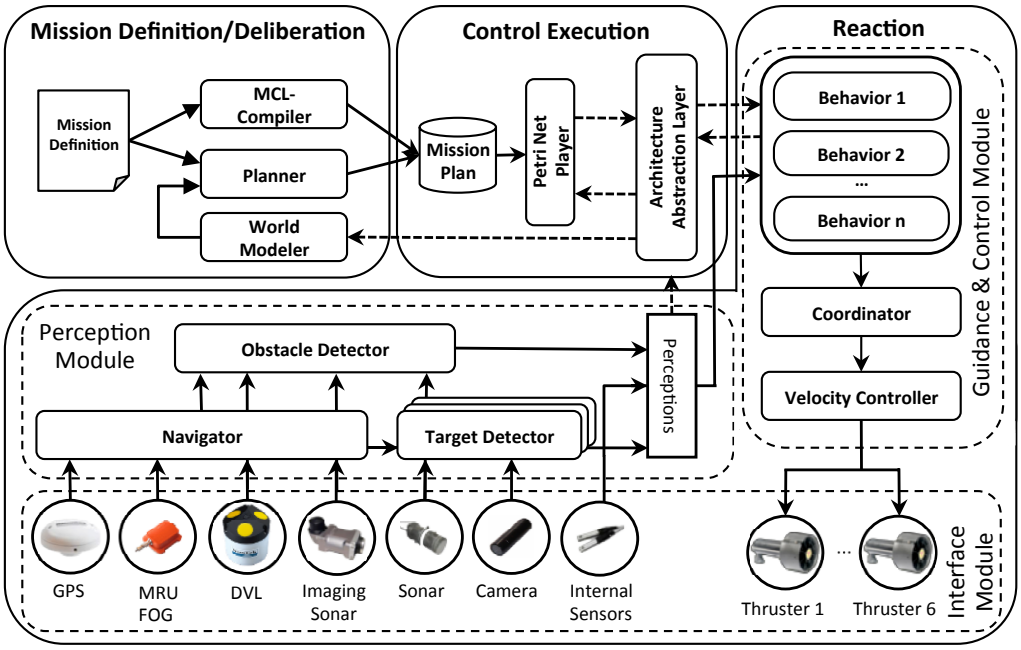


Figure 3.5: Example of a three-layered component based control architecture.

- A framework can be extended by a user but, in general, is not allowed to be modified.

There are several generic frameworks that claim to simplify the process to create a control architecture for a robot. They provide tools to build the components that compose the control architecture, simplify the communications among them and offer simulation tools among others.

Some of these frameworks are more suitable for a kind of robots than others. Orocos [oro, 2011], for instance, is mainly used to control industrial manipulators but it has also been successfully applied to autonomous vehicles like the Berlin Racing Team that took part of the Urban Grand Challenge Competition in 2007 [Kolagheichi-Ganjineh, 2008]. It is a toolkit composed of a set of C++ libraries for advanced machine and robot control that provides kinematics and dynamics functions as well as Bayesian filters. On the other hand, the Yet Another Robot Platform (YARP) is a middle-ware mainly designed for humanoid robots. It supports building a robot control system as a collection of programs. These programs are communicated in a peer-to-peer way, with a family of connection types that can be swapped in and out to match the desired requirements. YARP contains

a set of libraries, protocols, and tools to keep modules and devices cleanly decoupled. An example of an humanoid using YARP is the iCub [Sandini et al., 2004]. Focused in mobile robots there are several solutions and, probably, the most widely used is the Player Project [Biggs et al., 2010]. The Player robot server is a robot control interface that supports a wide variety of robots. Most of them are small mobile robots like e-puck [e-p, 2011], iRobot create [iRo, 2011] or LEGO Mindstorms [LEG, 2011]. Player's client/server model allows robot control programs to be written in several programming languages like C++, Tcl, Java, or Python and to run on any computer with a network connection to the robot. It supports multiple concurrent client connections to devices. In fact, Player makes no assumptions about how to structure the robot control programs, thus being more *minimal* than other robot interfaces. Player also includes two simulation back-ends named Stage and Gazebo. They are a 2D and 3D multi-robot simulators, respectively, including realistic sensor feedback and physically plausible interactions between objects. Another toolset originally initiated by the United States Department of Defense to develop an open architecture for the domain of unmanned systems is the Joint Architecture for Unmanned Systems (JAUS). JAUS is built on the five principles of: vehicle platform independence, mission isolation, computer hardware independence, technology independence and operator use independence. An open source implementation of the JAUS project, named OpenJAUS, is currently supported by academic and industrial people. OpenJAUS includes all of the software and sample code necessary to standardize an unmanned system. It is based on components, and provide tools for message exchange, service creation, node management and simulation tools among others. Orca is another open-source framework for developing component-based robotic systems. It provides the means for defining and developing the building-blocks which can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks. It uses a commercial open-source library for communication and interface definition and tools to simplify the components development. The cross platform software for robotics research named Mission Oriented Operating Suite (MOOS) developed by Paul Newman and the Oxford Mobile Robotics Group (MRG) [Newman, 2005] has received also a good reception. It has not been used only in mobile robots but in underwater vehicles too. It contains a set of libraries to build independent components that are communicated among each other by means of a central blackboard. Several com-

ponents are included in MOOS to simplify common operations like data logging, component managing, vehicle teleoperation, navigation or mission definition and execution.

Although all these solutions come from institutions or foundations, the lack of an standard solution has caught the attention of some companies that have decided to offer their own environments for robot control and simulation. The first one to get some popularity among the robotics community was the Webots [Web, 2011], developed since 1996 by Cyber Robotics. Webots is a mobile robot prototyping and simulation software that handles a wide variety of standard simple mobile robots. Unlike the rest of the solutions, the main use of Webots is to simulate mobile robots or teleoperate them from a base computer instead of build a control architecture to be used within them. Microsoft, one of the biggest software companies in the world, has also developed his own platform named Microsoft Robotics Developer Studio (MRDS) [Johns and Taylor, 2008]. It offers the Concurrency and Coordination Runtime (CCR) that makes it easier to handle asynchronous input and output data, eliminating the conventional complexities of manual threading, locks, and semaphores. Lightweight state-oriented Decentralized Software Services (DSS) framework enables also to create program modules that can inter-operate on a robot as well as on connected PCs using a simple, open protocol. Moreover, the MRDS includes also simulation tools as well as a simple drag-and-drop Visual Programming Language (VPL) that makes it easy to create robotics applications. The VPL provides the ability to take a collection of connected blocks and reuse them as a single block elsewhere in the program. MRDS also support a number of languages including C# and Visual Basic .NET, JScript, and IronPython. The main criticism received by MRDS is its totally dependence of Microsoft tools and operative system. Gostai, the company that builds the Jazz robot, has also its own software platform to control robots and complex systems in general named Urbi [urb, 2011]. Urbi includes a C++ component library called UObject that comes with a robot standard Application programming interface (API) to describe motors, sensors and algorithms. Once all the sensors and actuators in the robot are defined using UObject, the urbiscript orchestration script language can be used to glue the components together and describe high level behaviors but with embedded parallel and event-driven semantics to make the job easier. Finally, there is the solution proposed by Willow Garage named Robot Operating System (ROS) [WillowGarage, 2010]. Willow Garage

is the manufacturer of PR2 and Texai robots but also leads the development of the open source ROS as well as supports the development of other popular projects like the OpenCV [GaryBradski, 2010], the Point Cloud Library (PCL) and the Player project [Biggs et al., 2010]. ROS is a set of libraries and tools for building robotics applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, etc. In only three years, over than 50 robots are using ROS and more than 1600 public packages are available. Moreover, Player, Stage and Gazebo have been adapted for being used within the ROS platform, Urbi integrates ROS support, Orocos tool-chains allows to use their libraries into a ROS stack and YARP supports the *TCPROS*, which is the transport layer for ROS messages and services to communicate their components. Unlike other systems, ROS is a meta-operating system for a robot providing the services expected from an operating system.

Despite the facilities provided by these or other frameworks many developers decide to roll their own solutions. The overhead produced by these packages, the necessity to adapt the own code to them or the desire to have a fully customized and controlled solution are some of the reasons for doing it. However, as the features offered by these solutions increase, the arguments against their use decrease. And intermediate solution has been proposed to develop a custom framework that can be easily connected to several *generic* frameworks. Thus, it is possible to keep a small control architecture easy to maintain and very adapted to our necessities but also, communicate our custom components with components developed using popular generic frameworks.

The functional requirements offered by the proposed framework are:

- Simplify the creation and the addition of new components.
- Be as less invasive as possible when building the components, minimizing the coupling with the system and the rest of components.
- Allow to enable and disable components during the initialization and run-time.
- Allow to simultaneously execute tasks in each component.
- Provide simple mechanisms to communicate the components among them.

- Allow components to rise discrete events that can be used by a hierarchically superior component.
- Allow components to receive action commands that can be sent by a hierarchically superior component.
- Allow components to save information related with its configuration in a persistent way.
- Enable component logging features.
- Allow to distribute components among the nodes of a network.
- Provide an homogeneous interface to all the components to access input/output devices as serial ports, Ethernets, frame-grabbers, etc.
- Allow the communication with generic frameworks like, ROS, MOOS, the Player project, etc.

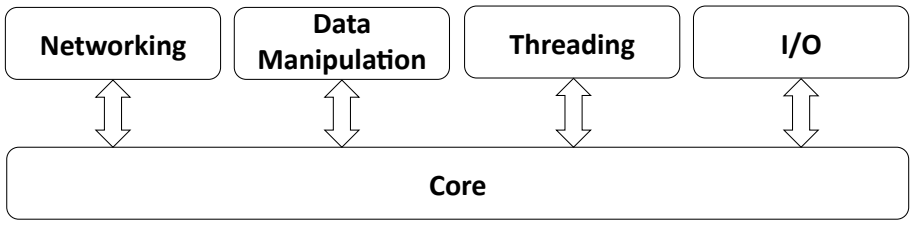


Figure 3.6: Framework modular design.

The framework has been divided in five modules as shown in Figure 3.6: the main module, named core, and four additional modules that provide the rest of the functionalities.

- The *Core Module* manages the communication among components together with the networking module. It is also responsible to transmit and receive events and actions, to control the components configuration, record the logs and manage the component's distribution.
- The *Networking Module* allows different network nodes to communicate with each other, and the whole system to communicate with external systems.

- The *Data Manipulator Module* provides tools to serialize and deserialize data structures into an Extensible Markup Language (XML) string or an XML string plus a byte array. Once all the structures are serialized and therefore share the same format, they can be easily transmitted between components.
- The *Threading Module* provides classes to create new execution threads allowing to execute parallel threads inside each component. Periodic, non-periodic and real-time threads are supplied.
- The *Input/Output Module* gives an uniform and easy access to most of the input/output devices available in an autonomous vehicle.

A key point when developing a control architecture is to decide how their components communicate between them. If multiple components are running into a multi-thread process, and hence sharing the same memory space, communication can be done by means of shared variables. However, when each component is an independent process that may run in an independent network node a communication protocol must be established. In order to communicate components among them it is necessary to define the messages to transmit and the mechanism to transport these messages. Some well-known communication packages are Common Object Request Broker Architecture (CORBA), Simple Object Access Protocol (SOAP) or, in the robotics field, Inter Process Communication (IPC). CORBA [Henning and Vinoski, 1999] allows a component to call a remote method implemented in another object. Like in many other communication packages, an interface Definition Language (IDL) is used to describe each message and once compiled, the necessary code to handle the communication is automatically generated. Several implementations of CORBA are available. The Ace Orb (TAO) is a popular choice in robotics [TAO, 2011] because of its real-time capabilities. Similar to CORBA there is the Internet Communications Engine (ICE) [ICE, 2011]. ICE is an object-oriented middleware that provides object-oriented Remote Procedure Call (RPC) and publish/subscribe functionalities. SOAP is a RPC protocol for exchanging structured information relying on the XML as its message format. The IPC [James, 2011] developed by the Carnegie Mellon University is another popular choice in the robotics community. It uses the publish/subscribe messaging pattern instead of the less flexible request/reply protocol.

Standard frameworks provide also their own solutions. Some of them are based on popular middle-wares like Orca that uses ICE or Orocos that uses CORBA. Orocos, however, includes two additional *toolchains* to enable communication among components: the Portable Operating System Interface for Unix (POSIX) MQueues asynchronous communications protocol and the communication system offered by ROS named TCPROC. YARP may use also the TCPROS mechanism to communicate its components, or programs, in a peer-to-peer way. However, many other transport protocols to carry data may be used like Transmission Control Protocol (TCP), User Datagram Protocol (UDP), multi-cast, local, XML/RPC, etc. In ROS, the components, named nodes, exchange information by means of publishing ROS messages to topics. A message is a simply data structure, comprising typed fields defined by a message description language. Also, request/reply communications among nodes can be done via *services* which are defined by a pair of messages: a request and a reply. All the messages are transported over the TCPROS transport layer that uses standard TCP/Internet Protocol (IP) sockets. Other frameworks use simple TCP sockets, like the Player project, or develop its custom communication protocol, like OpenJAUS, based on JAUS messages that are communicated using the functions provided by the OpenJAUS library to send and receive them. MOOS presents a different approach. Instead of providing a component to component communication, all the components have a connection to a central data base. Then, each component can publish data to this data base, register for notifications on named data and collect notifications on named data. This configuration is easier to setup but more inefficient and even more because messages are sent using the string format. Another different approach is the implemented by the MRDS. It uses the DSS as a state-oriented service model that combines the notion of Representational State Transfer (REST) with a system-level approach for building high-performance, scalable applications. In DSS services are exposed as resources which are accessible programmatically. By integrating service isolation, structured state manipulation, event notification, and formal service composition, DSS addresses the need for writing high-performance, observable, loosely coupled applications running on a single node or across the network. This solution is similar to the HTTP-REST software architecture style for distributed hypermedia systems but specially designed to be used in the context of a robotic application.

The communication strategy proposed in our custom framework to communicate components between them as well as with external systems involves the cooperation of several modules. The data manipulator module serializes and deserializes the data to be communicated while the core module and the networking module are used to transmit it. A well known software design pattern named *proxy* has been used to separate the message transmission mechanism from the rest of the code. The proxy pattern provides a surrogate for another component to control the access to it. The proxy acts as an intermediary between the client and the target component implementing the same interface than the latter. Then, each time that a message is received by a proxy, through a direct reference, it is transmitted to a proxy server. The proxy server is who really delivers the message to the target component through a local reference. Communications between the proxy and the proxy server are done through the network by means of TCP or UDP sockets, ROSTCP, CORBA, IPC, or any other available communication package. However, from the point of view of a component, all the communications are done through a local reference. In the current implementation, a custom network protocol over a persistent TCP connection has been chosen to communicate proxies and proxy servers. If this network protocol has to be changed for any other communication package it implies only to re-implement the proxy and the proxy server. Moreover, as all the components share the same interface, only one proxy and one proxy server must be implemented for each communication package being used.

Figure 3.7 shows how components A and B or C and D are communicated between them through a local reference because they are running in the same network node, and hence, sharing the same address space. However, to communicate component A with component C, a message has to be sent from component A to *Proxy* C, through a local reference. Then, the *Proxy* C component, in *Node I*, sends a message to the *Proxy Server* in *Node II* using a custom protocol over a TCP socket. Finally, the *Proxy Server* sends the received message to the component C, again, through a local reference. To enable this communication system it is necessary to have a proxy instance for each distributed component and a proxy server in each network node with references to all the distributed components in this node.

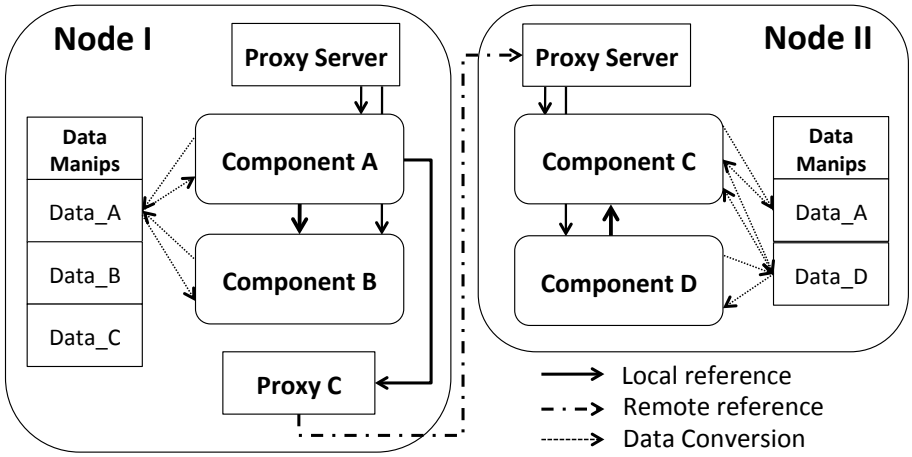


Figure 3.7: Example of communication between components.

3.2.2 Reactive layer

The reactive layer implemented in our vehicles is based on the O2CA2 reactive architecture presented in [Ridao et al., 2002], however, it has been updated and re-implemented using the previously presented framework. Its goal is to execute basic primitives in order to fulfill the missions defined at the mission layer. Primitives are basic robot functionalities offered by the robot control architecture. For an AUV, a primitive can range from a basic component that checks the battery level (e.g. *batteryMonitor()*) to a complex component that navigates towards a 3D way point (e.g. *goto(x, y, z)*). Primitives have a goal to be achieved. For instance, the goal of the *achieveAltitude* primitive would be to drive the robot at a constant altitude.

The reactive layer is very dependent on the sensors and actuators being used. It is divided in three modules: the *vehicle interface module*, the *perception module* and the *guidance and control module* (see Figure 3.5).

- The vehicle interface module contains components, named *drivers*, which interact with the hardware. It includes sensor drivers, used to read data from sensors and actuator drivers, used to send commands to the actuators. An additional function provided by the drivers is to convert all the data to the same units as well as to reference all the gathered data to the vehicle's fixed body frame.

- The perception module receives the data gathered by the vehicle interface module. Perception module components are called *processing units* and the main ones are: the navigator, the obstacle detector and the target detectors. The navigator processing unit estimates the vehicle position and velocity merging the data obtained from the navigation sensors by means of a Kalman filter [Ribas et al., 2010]. The obstacle detector measures the distance from the robot to the obstacles, mainly detected using acoustic sensors. Target detectors process acoustic or visual images to extract the most relevant features from them. Multiple target detectors have been programmed in order to detect different objects [El-Fakdi et al., 2010; Ribas et al., 2007].

- The guidance and control module includes a set of behaviors, a coordinator and a velocity controller. Behaviors receive data from the vehicle interface or perception modules, remaining independent of physical sensors and actuators used. Simple behaviors can be programmed as simple controllers, however, when the number of parameters to tune increases, it may be difficult to adjust them. Then, Reinforcement Learning (RL) techniques can be used to improve the adaptability of vehicle behaviors to the environment [Carreras et al., 2003; El-Fakdi et al., 2010]. The second component, the coordinator, combines all the responses generated by the behaviors in a single one [Carreras et al., 2001]. Extract 3.1 shows the coordination rule. Basically, each behavior generates a response including a velocity set-point for each DOF, an activation level for each DOF and a priority. The activation level is used to indicate over which DOFs the primitive acts. Responses are sorted by their priority and those with higher priority dominate over the others. The last component, the velocity controller, takes the velocity set-point provided by the coordinator, turns it into a velocity and computes a response for each thruster to achieve the desired velocity. A simple Proportional Integral Derivative (PID) for each DOF is used for this task.

The COLA2 architecture can be run in simulation mode. Then, instead of using the driver components available in the interface module, a module that simulates the vehicle dynamics as well as its environment is used. This module, named *Neptune* simulator [Ridao et al., 2004a], sends navigation data to the navigator processing unit while receives thruster set-points from the vehicle velocity

controller. It is also able to artificially generate camera images and acoustic sensor data for the obstacle detector and the target detectors processing units. Therefore, for the perception module and the guidance and control module there is no difference in interfacing the Neptune simulator or the real sensor/actuator drivers.

Algorithm 3.1: Behavior coordination responses algorithm.

```
struct bhResponse
    float response[NDOF] ;
    float actLevel[NDOF] ;
    int priority ;
coordinator( vector<bhResponse> responses )
    float k = 1 ; //Coordination tuning parameter
    vector<bhResponse> sortedResp = sortByPriority(responses) ;
    for int j = 1 ; j < sortedResp.size() ; j++ do
        for int i = 0 ; i < NDOF ; j++ do
            sortedResp[j].reponse[i] = sortedResp[j-1].response( i ) *
            sortedResp[j-1].actLevel( i ) + ( k - responses[j-1].actLevel( i ) )
            * sortedResp[j].actLevel( i ) * sortedResp[j].response( i ) ) ;
            sortedResp[j].actLevel[i] = sortedResp[j-1].actLevel( i ) + ( k -
            sortedResp[j-1].actLevel( i ) ) * sortedResp[j].actLevel( i ) ) ;
    sortedResp[sortedResp.size() - 1].priority = 1 ;
    return sortedResp[sortedResp.size() - 1] ;
```

3.2.3 Execution layer

The execution layer acts as the interface between the reactive layer and the mission layer, translating high-level plans into low-level commands. Additionally, the execution layer monitors the primitives being executed in the reactive layer. The functionalities of this layer have been included in the proposed MCS. Different approaches can be found in the literature to implement this layer: from *lisp* [Barbier et al., 2001] to *prolog* [Healey et al., 1996] interpreters that translate high-level plans into basic commands, to more conventional alternatives that use state machines [Newman, 2005] or Petri nets [Caccia et al., 2005; Oliveira et al., 1998] to relate the Discrete Event System (DES) that describes an autonomous mission with the primitives under execution.

In COLA2, the execution layer is composed by two main components: the Architecture Abstraction Component (AAC) and the Petri Net Player (PNP), see Figure 3.4. The AAC is located at the bottom of the execution layer and keeps the mission and execution layers, vehicle-independent. Therefore, the reactive layer is the only one tied with the vehicle's hardware. The AAC provides an interface to the reactive layer based on three types of signals: *actions*, *events* and *perceptions*.

- Actions enable or disable basic primitives within the vehicle's reactive layer. For instance, an action can enable a primitive which controls the vehicle's depth pointing to a desired set-point and the maximum time to reach it.
- Events are triggered in the reactive layer to notify changes in the state of its primitives. Following the last example, an event can announce that the desired depth has been reached within the required time or that the time has run out.
- Perceptions, meaning specific sensor or processing unit values, are transmitted from the reactive layer to the mission layer in order to be used to extract relevant information about the current world state when an on-board planner is used, as it will be shown in Chapter 7. Therefore, the execution layer is not using the perceptions, just transfering them from the reactive to the mission layer.

Although the three layers that compose the COLA2 have been entirely implemented using the proposed framework, the AAC allows to connect the execution layer and the mission layer, which compose the MCS, to other vehicles that already implement its own reactive layer. To this end, the actions that must be carried out are:

- Map the primitives that the particular system is able to perform into actions that the MCS can execute.
- Map the notifications produced by these primitives into interpretable events for the MCS.
- Specify which perceptions have to be transmitted from the reactive to the mission layer.

Particularly, in COLA2, these are the points to take into account:

- Since the primitives that the proposed reactive layer can execute are provided by the behaviors within the control and guidance module but also by the processing units in the perception module, actions to enable, disable and reconfigure all these components must be defined.
- Each enabled primitive is able to raise notifications indicating its state. Basically, two notifications may be produced: *the primitive has achieved its goal* or *the primitive is not able to achieve its goal*. The AAC has to receive these notifications and map them into an event.
- Some values, produced by sensor components, processing units, behaviors or even actuators, are useful to work out the current world state. These data have to be transmitted from their source components to the component that uses then to extract the knowledge about the current world state: the world modeler. In COLA2, navigation data from the navigator component, safety information collected by water sensors or the battery monitor and boolean information produced by object detector processing units is transmitted to build the current world state.

The second module included in the execution layer is the PNP. The PNP executes mission plans using the Petri net formalism by sending actions and receiving events through the AAC. The execution layer behaves as a DES which connects high-level discrete plans, given by the mission layer, with low-level continuous primitives, in the reactive layer. The PNP controls all the timers associated to timed transitions, fires enabled transitions following the Petri net transition rule, sends actions from the execution layer to the reactive layer and, if necessary, fires enabled transitions in the mission layer when events are received. The PNP component uses TCP sockets to communicate with the AAC. Hence, the AAC can be implemented together with the reactive layer, using the same framework, in order to simplify the communication with all its components. For instance, if a vehicle implementing a reactive layer using ROS wants to be controlled by the proposed MCS, an AAC can be implemented in ROS to simplify the access to all the nodes in the reactive layer. The AAC is then communicated with the PNP through a TCP socket and no more changes have to be performed in the execution or mission layers neither into the native ROS reactive layer.

How Petri nets are used to define a mission and how the PNP interprets mission plans is presented in chapters 4 and 5.

3.2.4 Mission layer

Predefined plans are the current state of the art for AUV missions. However, when dealing with unknown changing environments with imprecise and noisy sensors, off-line plans can fail. The difficulty of controlling the time in which events happen, energy management, sensor malfunctions or the lack of on-board situational awareness may cause predefined plans to fail during execution as assumptions upon which they were based are violated [Turner, 2005]. Therefore, it is worth to study the inclusion of an on-board planner with the ability to modify or re-plan the original plan when dealing with missions in which an off-line plan is susceptible to fail. Thus, a compromise between predefined off-line plans and automatically generated on-line plans is desirable. Our solution starts with the introduction of a high-level language, named Mission Control Language (MCL) to describe off-line plans that can be automatically compiled into a Petri net following a set of desired properties, as introduced in Chapter 5. Next, the interface with an on-board planner able to automatically sequence *planning operators* to fulfill a set of given goals is studied in Chapter 7.

3.2.5 Implementation

The C++ language and a set of well known libraries have been chosen to develop the framework and therefore the components that compose the COLA2 control architecture. The main libraries used are Standard Template Library (STL) [Internationa, 2010] and Boost [Dawes et al., 2010] for basic structures, algorithms, threading and device access, Poco [Engineering, 2010] that provides logging facilities and an XML parser, OpenCV [GaryBradski, 2010] for image processing tasks and Another Tool for Language Recognition (ANTLR) [Parr, 2010] to develop the high-level mission definition language compiler presented in Chapter 5.

To solve the repetitive functions that arise when developing a control architecture, a set of custom libraries have been implemented too:

- A *Numeric* library that provides the definition of constants, units conversion, 2D and 3D algebraic transformations, etc.

- A library to simplify the development of an Extended Kalman Filter (EKF) for navigation and mapping purposes.
- A library to simplify the development of PID controllers mainly used by behaviors in the guidance and control module.

The three layers composing the COLA2 have been implemented using these libraries/framework. However, the components of the reactive layer and the AAC can be implemented using a different framework and connected to the proposed MCS through a standard TCP socket in order to keep the proposed MCS vehicle independent.

Chapter 4

Defining a mission using Petri nets

Chapter 2 introduces several Mission Control Systems (MCSs) used by underwater vehicles to define and execute autonomous missions. Each mission plan is basically a Discrete Event System (DES) which determines the vehicle primitives to be executed, in accordance with the events raised in the vehicle control architecture, to fulfill a mission. Despite the multiple alternatives proposed in the literature, well known formalisms like Finite State Machines (FSMs) or Petri nets have been proved to be the most suitable way to describe a mission formally. Petri nets and FSM are a good choice because they are well studied and naturally oriented towards the modeling and analyzing of asynchronous and concurrent DESs. Moreover, an appropriated use of these formalisms leads naturally to a unifying formal framework for the analysis of the logical behavior of the DES that occurs at all levels of the MCS as well as simplifies its posterior execution. Finally, using well known formalism it is possible to guarantee basic properties verifying the resulting DES before its execution. Compared with FSMs, Petri nets are more compact and better suited to express parallelism and can also represent a larger class of languages than standard FSMs. Both Petri net and FSM formalisms provide reachability analysis techniques to check basic properties as deadlock avoidance or the reachability of final states from an initial state. However, Petri nets provide also linear-algebraic algorithms that can be used to obtain more information about its behavior. Therefore, Petri nets have been choose as the formalism to describe the mission plans for Autonomous Underwater Vehicles

(AUVs) following the methodology proposed in this dissertation. Section 2.6.1 presents a an extended discussion about why Petri nets have been used as the base formalism in the proposed MCS.

Several elements have to be defined to program a mission plan. We not only pretend to model the flow between vehicle primitives but the framework in which these primitives are executed too. Therefore, these primitives are modeled first, by means of Petri nets, to describe their behavior. Primitives are part of the control architecture and therefore, some resources can be modeled together with them. Next, to control the execution flow of primitive models as well as to supervise its own execution a set of Petri net structures named Petri Net Building Blocks (PNBBs) has been proposed. PNBBs are the basic elements used to define a mission plan and are divided in two main classes: *tasks* and *control structures*. *Tasks* are used to supervise the execution of primitive models, while *control structures* are used to control the execution flow between PNBBs (tasks as well as control structures) sequentially, in parallel, iteratively or conditionally.

Once primitive models and PNBBs are defined, a mission can be defined just composing them according to some basic rules. If a set of desired properties is hold for all these basic control structures (primitive models, tasks and control structures), the mission plan obtained through the composition of these structures will accomplish these same properties without need of further verification.

After the formal definition of a DES, the chapter describes how the vehicle primitives are modeled and supervised using tasks as well as how their execution flow is controlled by means of control structures. The properties that have to be verified for each PNBB as well as how these PNBBs are composed among them are also introduced.

4.1 Discrete Event System

To execute a mission, an AUV has to enable/disable a number of vehicle primitives in a specific order. This order will be determined by two factors: the mission to perform and the changes produced in the environment or in the vehicle itself while executing these vehicle primitives. As seen in previous chapters, *actions* are issued to the execution layer to enable/disable vehicle primitives and *events* are received from these primitives informing about changes in the environment or in the vehicle itself. Then, a mission plan can be seen as a DES responsible of

defining which actions must be executed in each state according to the received events.

Definition 4.1.1. *A Discrete Event System is a discrete-state, event-driven system, that is, its state evolution depends entirely on the occurrence of asynchronous discrete events over time [Cassandras and Lafortune, 2007].*

Then, a formal definition for a DES is

$$\Sigma = \{S, A, E, \gamma\}, \tag{4.1}$$

where

- $S = \{s_1, s_2, \cdots\}$ is a finite or recursively enumerable set of states;
- $A = \{\lambda, a_1, a_2, \cdots\}$ is a finite or recursively enumerable set of actions plus the null action λ. A is also known as the set of output events;
- $E = \{\lambda, e_1, e_2, \cdots\}$ is a finite or recursively enumerable set of events plus the null event λ. E is also known as the set of input events; and
- $\gamma : S \times E \to 2^S \times A$ is a state-transition function.

The state-transition function $\gamma(s, e) = (s', a)$, where $s, s' \in S$, $e \in E$ and $a \in A$, represents the system's behavior in response to the detected events. This is when the system is in state s and the event e is received, the system should change its state to s' and execute the action a. Since null events and actions (λ) are included in E and A respectively, the system can change its state even if no events are received ($\gamma(s, \lambda) = (s', a)$). Similarly, a state-transition may be produce without launching any action ($\gamma(s, e) = (s', \lambda)$).

In this dissertation, Petri nets conform the formalism chosen to describe the DES to be used for describing autonomous missions. Appendix A presents some general concepts about Petri nets.

4.2 Primitives

As introduced in previous chapters, primitives are basic robot functionalities offered by the vehicle control architecture. For instance, typical primitives for an

AUV are: reach a certain depth (*AchieveDepth*), navigate towards a way-point (*Goto*) or detect that an alarm has raised (*Alarm*). In general, primitives have a goal to achieve, for instance, the goal for the *Goto* primitive is to drive the robot inside a particular sphere of acceptance centered in a target way-point.

Chapter 2 shows that one of the benefits of using a formalism appears when not only the mission plan is defined using the formalism but also the framework in which the mission is executed. In control architectures like Coral [Oliveira et al., 1998], each system in charge of controlling some vehicle capabilities is modeled using Petri nets. Similar happens in the architecture proposed by the he CNR-ISSIA [Caccia et al., 2005] where Petri nets are used to describe the execution flow as well as to model sensors and controllers. As the proposed approach wants to be vehicle independent, it is not possible to model the elements that compound the vehicle (or the reactive layer) and then use these elements to define a mission. Instead, we model the primitives that the vehicle is able to execute regardless how they are implemented or the dependencies that they have. Then, some properties have to hold:

- Each primitive can be enabled regardless of which other primitives are already enabled.
- If two or more primitives can not be enabled simultaneously, because they exclusively share some resources, they have to be implemented as a single primitive.
- Primitives have to take care to behave as expected in the primitive model. Therefore, although a software or hardware failure is produced the primitive have to finalize with a valid final state.

From the point of view of a DES, what we care about primitives are the *actions* that they can receive and the *events* that the primitives may raise. A generic primitive model have been defined in which only two actions can be sent to a primitive: one for enabling it and another for disabling it. However, this assumption can be easily enriched adding some parameters to these actions. For instance, the action to enable the primitive that drives a robot to a specific way-point can be parametrized with the location (x, y and z) of this way-point or if a primitive implements several behaviors that share a resource, it can be parametrized to select which one to execute. On the other hand, each primitive

may raises several events, but to keep things simple, only two of them will be considered in this generic primitive model: an *ok* event informing that an already enabled primitive has achieved its goal and a *fail* event informing that a failure or time-out happened while the primitive was enable.

The behavior of a primitive able to receive two actions and to raise two events can be modeled using a Petri net. If only the relationship between actions and events is expressed, a general Petri net model can be build. Figure 4.1 presents three models for primitives increasing in complexity. Next, these three models are discussed.

- The enable/disable primitive model shown in Figure 4.1(a), models the actions *enable* and *disable* but any event. To enable the primitive, place *enable* has to be marked. If it was previously disabled (place *off* marked) the transition $T0$ fires sending an enabling action to the vehicle primitive and marking the place *exe* that indicates that it is under execution. When the primitive is enabled, it can be disabled marking the *disable* place, that produces the firing of $T1$ sending a disable action to the vehicle primitive and marking again the *off* place.

- The enable/disable, ok model, see Figure 4.1(b), presents a primitive which is pursuing a certain a goal. When the primitive is enabled (place *exe* marked), if the goal that the primitive is seeking is reached the *ok* event is raised in the vehicle primitive and it is received in the primitive model firing $T2$ and marking the place *ok*. Transition $T2$ is a non-immediate transition that depends on the reception of an external event to fire. Here, transition $T2$ is related with event *ok*, then, to fire, transition $T2$ have to be enabled and event *ok* received. The primitive may be disabled from the *ok* state (when place *ok* is marked) or from the *exe* state (when place *exe* is marked). All the transitions that restore the token to place *off* (here $T1$ and $T3$) sent also a disabling action to the vehicle primitive.

- The enable/disable, ok/fail model shown in Figure 4.1(c) introduces a primitive with four main states: (i) disabled, (ii) seeking a goal, (iii) the goal has been achieved and (iv) an error has occurred. In the first and second states, place *off* or *exe* are marked respectively. In the third state, while place *exe* is marked, the vehicle primitive running in the robot control architecture

is responsible for checking the *achievement conditions*. When these conditions are achieved, an *ok* event raises firing transition $T2$ and marking the *ok* place. On the other hand, if an error is detected by the vehicle primitive while it is under execution, the *fail* event raises firing $T3$ and marking the *fail* place. In this model, the primitive can be disabled from states *exe*, *ok* or *fail* by firing transitions $T1$, $T6$ or $T7$ respectively.

These primitive models must be used as a guide line to generate the vehicle primitives that runs on the vehicle architecture and those primitives must ensure that they behave like the model that represents them. So, it is possible to ensure that the input-output behavior of the primitive satisfies the pre-specified requirements and then, it can be safely executed by a supervisory Petri net. The three presented models contain non-immediate transitions (named *enabling* and *disabling*) that are fired when a hierarchically superior Petri net decides it. When transition $T0$ in Figure 4.1(a), (b) or (c) fires, an enabling action is sent from the primitive model to the vehicle primitive through the Architecture Abstraction Component (AAC). Disabling actions are sent from all the transitions that restore the token to the *off* place. The events raised by the vehicle primitive are also related to non-immediate transitions. These transitions ($T2$ in Figure 4.1(b) and $T2$ and $T3$ in Figure 4.1(c)) models the continuous process executed in the vehicles primitive that detects if the primitive goal is achieved, $T2$ fires, or if the primitive is unable to achieve its goal and therefore fails, $T3$ fires. In a particular primitive, for instance the *AchieveDepth* primitive, the firing of $T2$ can be subjected to the condition:

"If the vehicle has kept its depth during n seconds within an error margin of plus/minus m meters respect to a desired depth, then the ok event that produces the firing of T2 is raised."

Thus, when a condition is achieved by the vehicle primitive, an event is sent from the reactive layer to its corresponding Petri net model in the MCS through the AAC to fire the transition that is related with this event. As only one primitive model is associated to each vehicle primitive and only one transition is associated to each event belonging to a primitive, when an event is received in the MCS the transition associated to this event is always enabled and ready to fire if the vehicle primitive has been designed according to the primitive model.

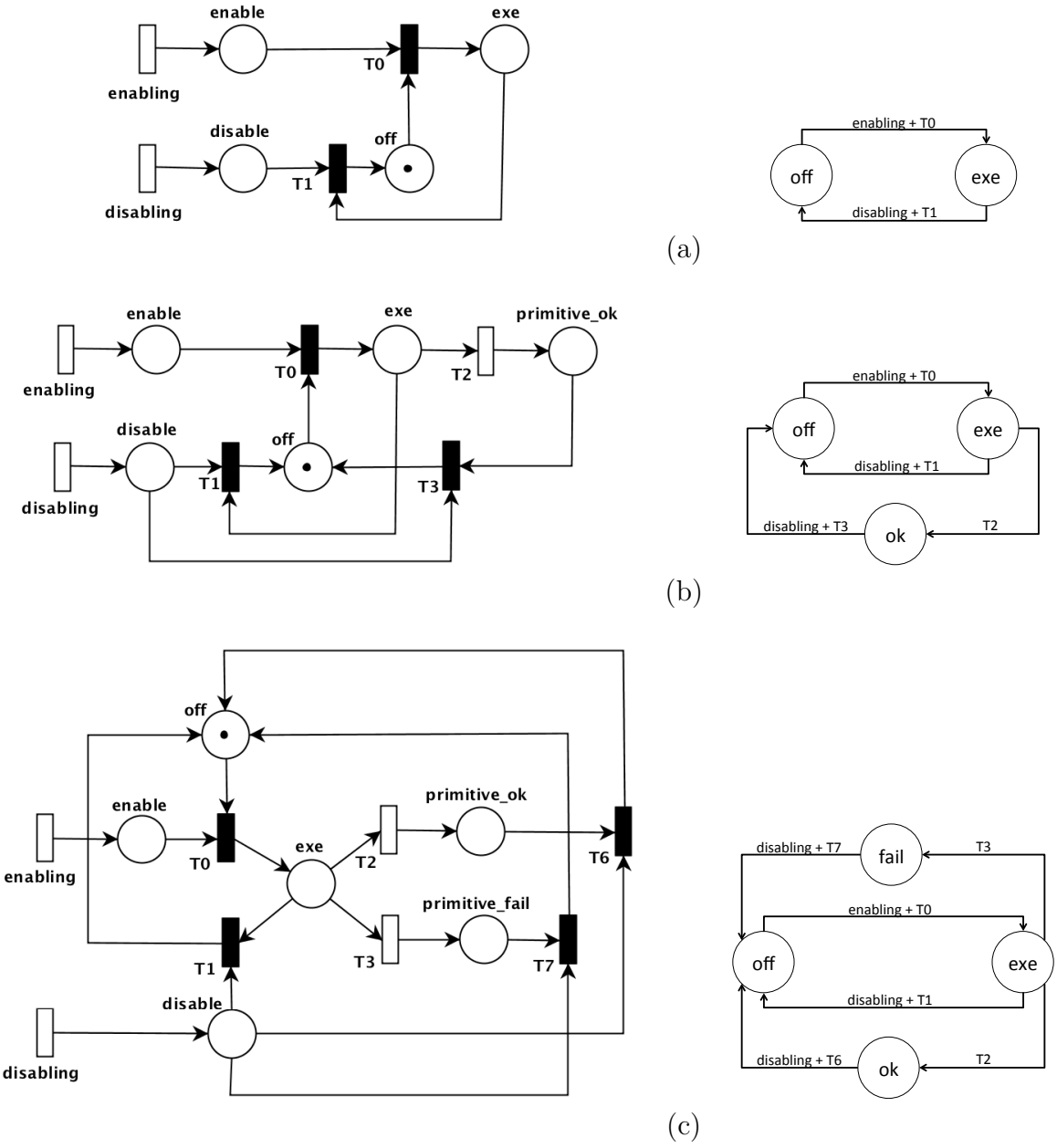


Figure 4.1: Petri net model of three different robot primitives and its corresponding state machine.

4.2.1 Primitive verification

Once a Petri net model for a vehicle primitive has been formally defined it can be verified. Basically, we want to check that the Petri net model evolves free of deadlocks from an initial state, where the Petri net is enabled, to a final state and, moreover, that once disabled it can be reused. The desirable properties of the net can be checked performing a reachability analysis or, alternatively, studying its invariants, siphons and traps, see Appendix A. When performing a reachability analysis, it is well known the high computational burden due to the state explosion problem. However, at this level, Petri nets modeling vehicle primitives are kept small so this analysis can be easily done. The reachability graph presented next, is computed considering that:

Each vehicle primitive is "connected" (receives actions and sent events) with only one primitive model and can be only disabled from a hierarchic supervisor if it was previously enabled for this same supervisor.

The reachability graph for the primitive model presented in Figure 4.1(c) is shown in Figure 4.2. It contains 8 states. Each one of these states corresponds to a marking vector in the Petri net in which $S_i = \{\mu(enable)\ \mu(disable)\ \mu(exe)\ \mu(off)\ \mu(ok)\ \mu(fail)\}$. Figure 4.2 shows how the Petri net can evolve free of deadlocks from the initial state S_{Off} towards any other state and come back later to S_{Off}. Because the primitive initial and final states are the same, the primitive model is reusable (e.g. the *AchieveDepth* primitive can be run more than one time without reinitialization). It is worth noting that 4 *tangible* states, see Appendix A, appear in the reachability tree: $S_{Off} = \{0\ 0\ 0\ 1\ 0\ 0\}$, $S_{Exe} = \{0\ 0\ 1\ 0\ 0\ 0\}$, $S_{Ok} = \{0\ 0\ 0\ 0\ 1\ 0\}$ and $S_{Fail} = \{0\ 0\ 0\ 0\ 0\ 1\}$. S_{Off} is the rest state, here the primitive is disabled, place *off* is marked, and waiting to be enabled. In S_{Exe}, the primitive is seeking for a goal, place *exe* is marked. S_{Ok} appears when the goal has been achieved, place *ok* is marked, while S_{Fail} state indicates that a failure has happened, place *fail* is marked. The other four states $S_{Enabled} = \{1\ 0\ 0\ 1\ 0\ 0\}$, $S_{disabled\_1} = \{0\ 1\ 1\ 0\ 0\ 0\}$, $S_{disabled\_2} = \{0\ 1\ 0\ 0\ 1\ 0\}$, and $S_{disabled\_3} = \{0\ 1\ 0\ 0\ 0\ 1\}$ are vanishing states. Figure 4.1(c) shows a simplified reachability graph in which only tangible states are present.

Although a reachability analysis is enough to check all the desired properties it is possible to analyze siphons, traps and invariants to prove the deadlock free

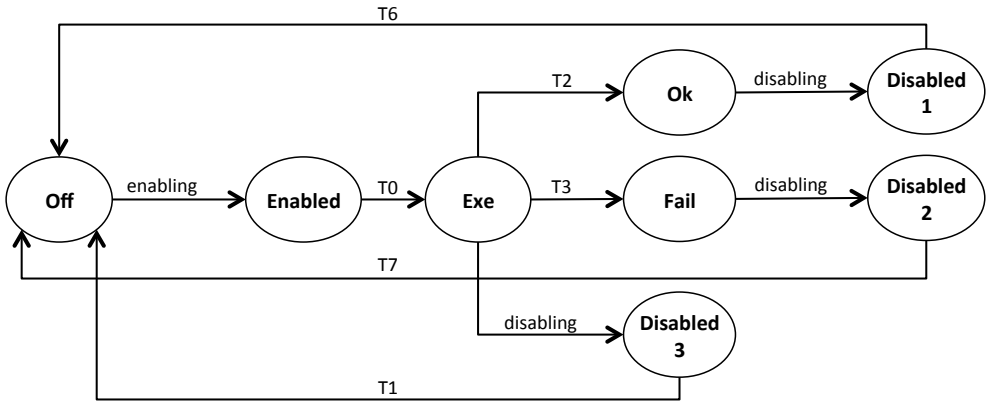


Figure 4.2: Primitive model reachability graph with vanishing states.

condition. When analyzing them, only the trap

$$trap \ = \ \{exe, \ off, \ ok, \ fail\} \tag{4.2}$$

is found. One place invariant is also found involving the same set of states:

$$exe + off + ok + fail \ = \ 1 \tag{4.3}$$

According to Iordache and Antsaklis [2006a], if all the siphons in a Petri net are controlled by an invariant or a trap and they are marked in their initial state (in $S_{Disabled}$ the $\mu(off) = 1$) it is possible to ensure that they will not lose all their tokens and hence, to ensure that a deadlock will never happen.

4.3 Petri Net Building Blocks

PNBBs are the basic building structures used to encode a mission. They are Petri nets with a definite functionality that implement a specific *interface*.

We understand as an interface the set of places belonging to a Petri net that are used to connect this Petri net structure with the others. All the PNBBs have at least one interface I_1 (or only I), named external interface, that is composed

by a set of places where

$$\begin{aligned} I &= I_{input} \cup I_{output} \\ I_{input} \cap I_{output} &= \emptyset \\ \forall p \in I_{input},\ \bullet p &= \emptyset \\ \forall p \in I_{output},\ p\bullet &= \emptyset. \end{aligned} \tag{4.4}$$

The places that compose the interface are called fusion places. A state that contains a place $p \in I_{input}$ marked, is named an *input state* while a state that contains a place $p \in I_{output}$ marked, is named an *output state*. According to the number of places in the interface, different interfaces may be defined. Figure 4.3 presents four PNBBs that implement a different interface each. Moreover, a block diagram that exemplifies the behavior of each interface is also shown.

- Figure 4.3(a) shows a PNBB with $I_{input} = \{begin\}$, $I_{output} = \{ok\}$ that can be only enabled and when a condition is achieved an *ok* state is marked. Its interface includes one input place and one output place. When the place *begin* is marked, the non-immediate transition $T0$ is enabled. When $T0$ fires the output place *ok* is marked.

- Figure 4.3(b) shows a PNBB with $I_{input} = \{begin\}$, $I_{output} = \{ok, fail\}$ that once it is enabled it can achieve a successful final state, place *ok* marked, or an unsuccessful final state, place *fail* marked. Its interface includes one input place and two output places. When the place *begin* is marked, the non-immediate transitions $T0$ and $T1$ are enabled and in conflict. The first one to fire will mark the output place *ok* or *fail*.

- Figure 4.3(c) shows a PNBB with $I_{input} = \{begin, abort\}$, $I_{output} = \{ok\}$ that includes two input places and one output place in its interface. When the structure is enabled, place *begin* marked, if place *abort* becomes marked $T1$ fires immediately removing the token in the *begin* place. Otherwise, if $T0$ fires the output place *ok* is marked.

- Figure 4.3(d) corresponds to a PNBB with $I_{input} = \{begin, abort\}$, $I_{output} = \{ok, fail\}$ that can be enabled and aborted. While the structure is enabled it can reach a desired goal state, marking place *ok*, or it can fail trying to achieve this state, marking place *fail*. Non-immediate transitions $T0$ and

$T1$ are enabled when the PNBB is started and may fire marking the output place *ok* or *fail*. However, if place *abort* becomes marked once the PNBB has started, $T2$ fires immediately removing the token in the *begin* place disabling transitions $T0$ and $T1$.

Two different types of PNBBs have been designed: *tasks* and *control structures*. Tasks are used to supervise primitive models. They implements only one interface, the external interface I. However, control structures, that are used to compose tasks as well as other control structures among them, implement multiple interfaces ($I_1 ... I_n$). Control structures implement an external interface as all the PNBBs (named I_1) but also implement several internal interfaces ($I_2 ... I_n$) in order to compose other PNBBs among them as explained later.

4.3.1 PNBBs verification

One of the major concerns when programming a mission for an AUV is to be able to verify some properties of the defined mission before its execution. Our approach to perform this verification consists on checking some properties to all the PNBBs used to compose the whole mission. If these properties holds for each PNBB they will hold also for the whole mission without need of further verifications. Next, the conditions to be checked for each PNBB are enumerated:

1. **Common Interface**: all the PNBBs used in a mission implements the same interfaces (external if tasks or external and internal if control structures).

2. **Reachability Condition**: from all possible initial states, the PNBB should evolve free of deadlocks until reaching a valid final state.

3. **Reusability**: task PNBBs have to be reused during the execution of a mission avoiding to be duplicated. Then the marking of all its places must be the same in the initial and final states except for the places belonging to the external interface.

All the PNBBs used to define a mission must share the same interface. Then, as the interface presented in Figure 4.3(d) is the more general one, it is the one that have been used in this dissertation. However, any other interface with a well

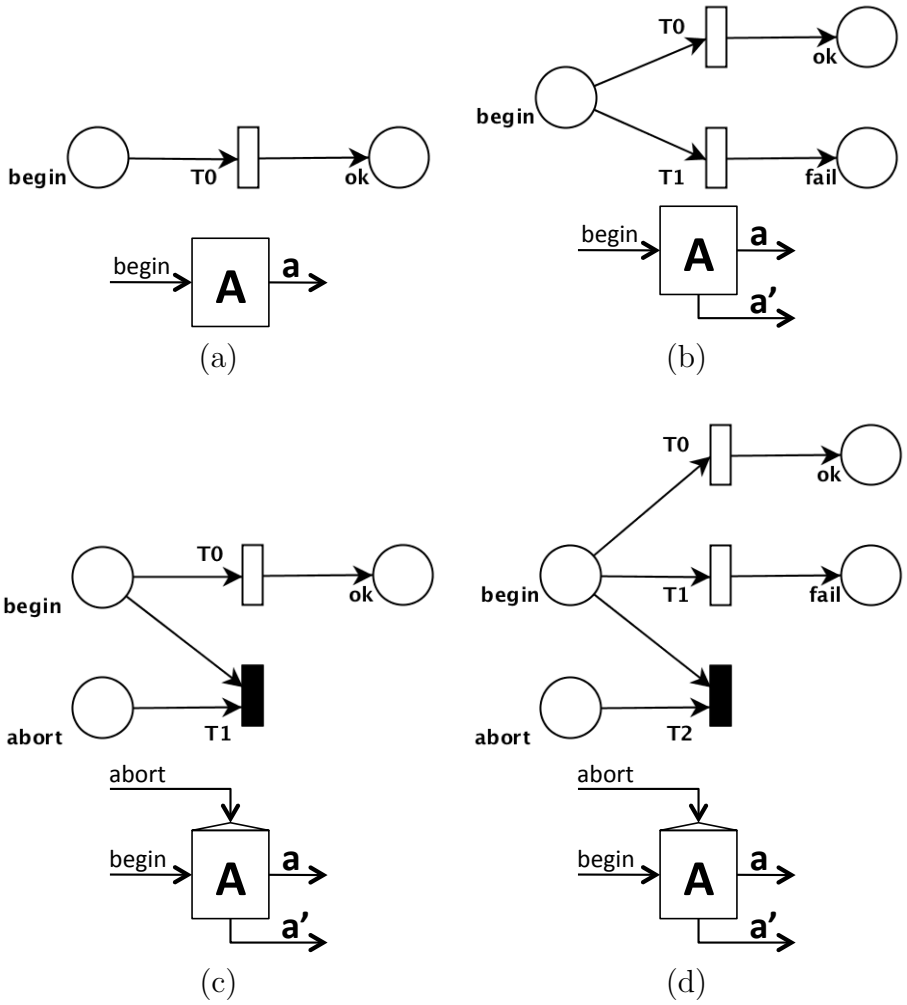


Figure 4.3: (a) One input one output interface, (b) one input two outputs interface, (c) two inputs one output interface, (d) two inputs two outputs interface.

defined set of input and output places could be used. It is worth noting that standardizing and limiting the number of actions and events for the tasks it is easy to systematize the definition of a mission.

To evaluate the last two properties, a reachability graph or an analysis based on siphons, traps and invariants can be performed. Since the PNBBs have been designed small enough, a small computational burden is expected if the reachability graph is build.

4.4 Tasks

Tasks are PNBBs that supervise the execution of a primitive model. Tasks have an external interface in order to be connected with other PNBBs but also are able to communicate with vehicle primitives by means of *actions* and *events*. The execution of an action involves sending a message from the MCS to the vehicles control architecture. This message will enable or disable one of the vehicle primitives. A set of parameters can be associated with these actions. Events communicate changes detected by the vehicle primitives to the MCS. Every event is associated to a particular non-immediate transition that will fire once enabled if its related event is received.

Figure 4.4 shows an example of a task, with the interface presented in Figure 4.3(d), and the relationship with the primitive model presented in Figure 4.1(c). The task may receive two events, *ok* and *fail*, and send two actions, *enable* and *disable*. Hence, its interface is defined by:

$$\begin{aligned} I_{input} &= \{begin,\ abort\} \\ I_{output} &= \{ok,\ fail\} \\ I = I_{input} \cup I_{output} &= \{begin,\ abort,\ ok,\ fail\} \end{aligned} \tag{4.5}$$

When the transition $TT0$ in Figure 4.4 fires, the primitive model is enabled. $TT0$ replaces here the *enabling* transition shown in Figure 4.1(c). Transitions $TT1$, $TT2$ and $TT3$ disable the primitive model. These transitions replace then the transition *disabling* in Figure 4.1(c). The events sent from the vehicle primitive to the supervisor task work as follows: when the *ok* event is received, transition $T2$ fires marking the *primitive_ok* place in the primitive model but also the *is_ok* place. Similarly, if the *fail* event raises, places *primitive_fail* and *is_fail* are

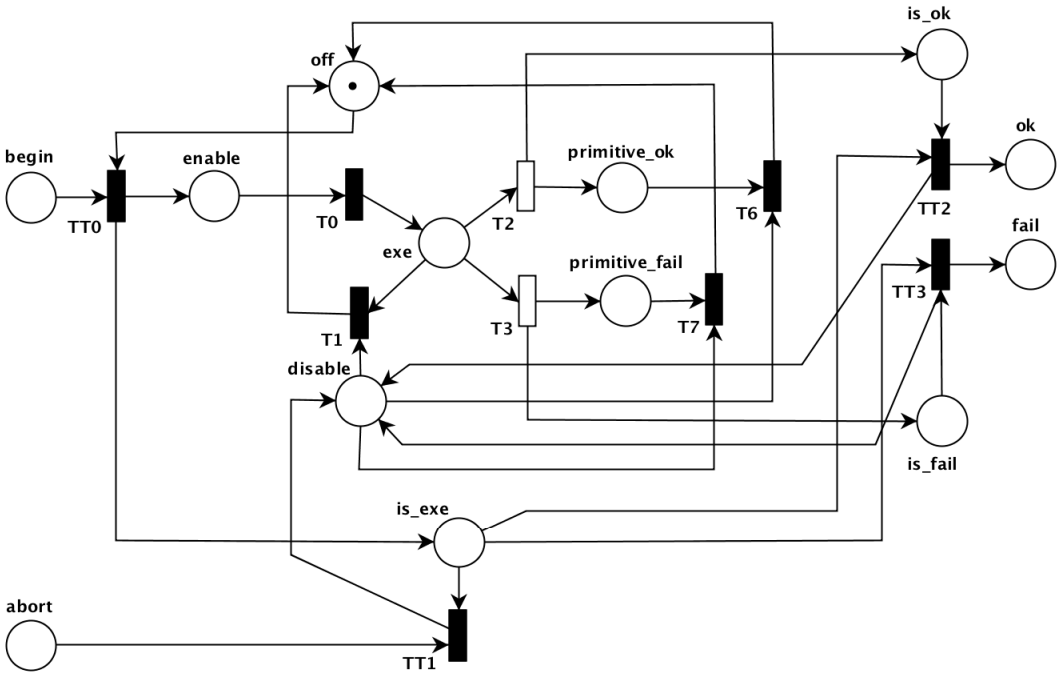


Figure 4.4: Example of a task PNBB and its relationship with the primitive model presented in Figure 4.1.

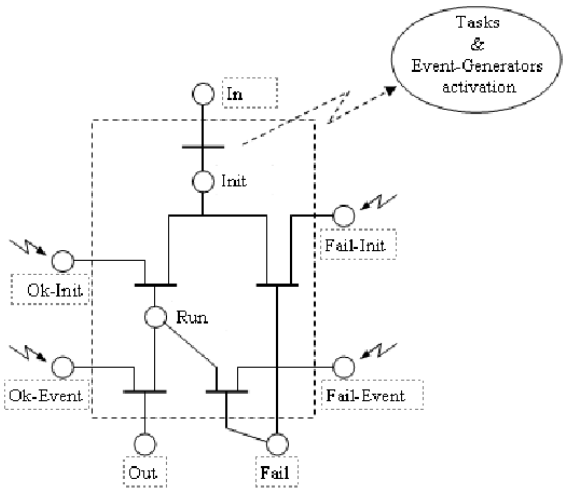


Figure 4.5: Execution action structure. Extracted from Bibuli et al. [2007].

marked. An importance difference with respect to the primitive model shown in Figure 4.1(c) is the *off* place. This place acts as a mutual exclusion avoiding to execute the primitive if it is already under execution. In the primitive model the *off* place was connected with the transition $T0$ while here it is connected with $TT0$. The mutual exclusion has been moved one step backward avoiding the multiple enabling of a primitive, however, the behavior in the primitive model has not changed.

The task presented in Figure 4.4 is very similar, in essence, to the *execution action structure* introduced in Bibuli et al. [2007] where each execution action structure, that is equivalent to the proposed PNBB tasks, is internally composed by a simple Petri net whose marking defines the action state as shown in Figure 4.5.

4.4.1 Task verification

When a superior control structure marks the input place *begin*, transition $TT0$ fires enabling the vehicle primitive, place *enable* marked. Once the primitive is enabled it is disabled if and only if:

1. The primitive achieves its goal raising an *ok* event which fires $T2$. The firing of $T2$ marks the place *is_ok* that produces the firing of $TT2$ disabling

the primitive.

2. A failure is detected within the primitive and the *fail* event, that fires $T3$, is raised. The firing of $T3$ marks the place *is_fail* that produces the firing of $TT3$ disabling the primitive.

3. The task is aborted by a hierarchic PNBB marking the input place *abort* in the interface. If the task is under execution when the place *abort* is marked, transition $TT1$ fires disabling the primitive.

If the first or the second conditions happens, the task disables the primitive marking again the *off* place and marking the *ok* or *fail* output place in the interface. However, if the latter happens, *abort* marked, the primitive is disabled, place *off* becomes marked, and the rest of tokens inside the task are removed.

The reachability graph for the Petri net in Figure 4.4 is shown in Figure 4.6. It contains only four states where S_i=$\{\mu(begin)$, $\mu(abort)$, $\mu(ok)$, $\mu(fail)$, $\mu(is\_exe)$, $\mu(is\_ok)$, $\mu(is\_fail)$, $\mu(enable)$, $\mu(disable)$, $\mu(off)$, $\mu(exe)$, $\mu(primitive\_ok)$, $\mu(primitive\_fail)$ $\}$. $\{begin$, *abort*, *ok*, *fail*$\}$ are the set of places that compose the interface, $\{enable$, *disable*, *off*, *exe*, *primitive_ok*, *primitive_fail*$\}$ are the set of places in the primitive model and $\{is\_exe$, *is_ok*, *is_fail*$\}$ are the rest of places that compose the PNBB task. The four tangible states are $S_{Task\_Disabled} = \{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\}$, $S_{Task\_Exe} = \{0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\}$, $S_{Task\_Ok} = \{0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\}$ and $S_{Task\_Fail} = \{0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\}$. From the initial state $S_{Task\_disabled}$ in which only the place *off* is marked two final states can be reached. Both of them have the place *off* marked and the only difference is because or place *ok*, in S_{Task_ok}, or place *fail*, in S_{Task_fail}, have been marked depending on how the primitive has finalized. If the task has been hierarchical aborted, place *abort* has received a token, while it was under execution, S_{Task_exe}, state $S_{Task\_disabled}$ is reached again. Then, the task evolves from the initial state to a final state within deadlocks except if it is aborted. In any case, from any valid marking in the input interface,*begin* or *begin* and later *abort*, only one place in the output interface, *ok* or *fail*, is marked or any if the task is aborted. The rest of places preserve its original marking making the structure reusable.

The reachability graph presented in Figure 4.6 has been build according to one assumption:

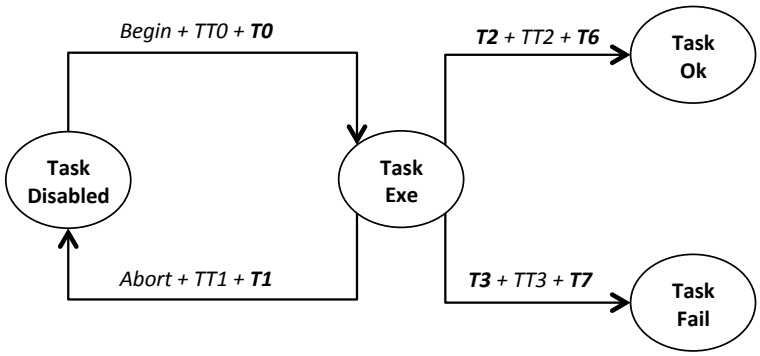


Figure 4.6: Task PNBB reachability graph.

A task PNBB can only be aborted by the hierarchical superior PNBB that has started it and only after being started.

Even though primitives are well supervised by tasks, if the vehicle primitive fails and it is not capable of raising an *ok* or a *fail* event the primitive model, and therefore the task that supervise it, may never end. To solve this problem two alternatives can be employed:

1. Model a time-out in the supervisor task using a timed transition as shown in Figure 4.7. When the timed transition ($TT4$) fires, the primitive is disabled as if a *fail* event had occurred.

2. Trust in a hierarchical superior structure to abort the task. However, we discourage the use of this alternative because of the potential risks involved.

Figure 4.8 shows the reachability graph for a timed task. The only difference with respect to the reachability graph shown in Figure 4.6 is that the state S_{Task_fail} can be reached if a *fail* event is received firing $T3$ and then $TT3$ and $T7$ but also if a time-out happens firing $TT4$ and then $TT5$ and $T1$.

4.5 Control structures

In order to execute the PNBB tasks sequentially, in parallel, or using iterative or conditional structures another kind of PNBBs are used. These PNBBs called

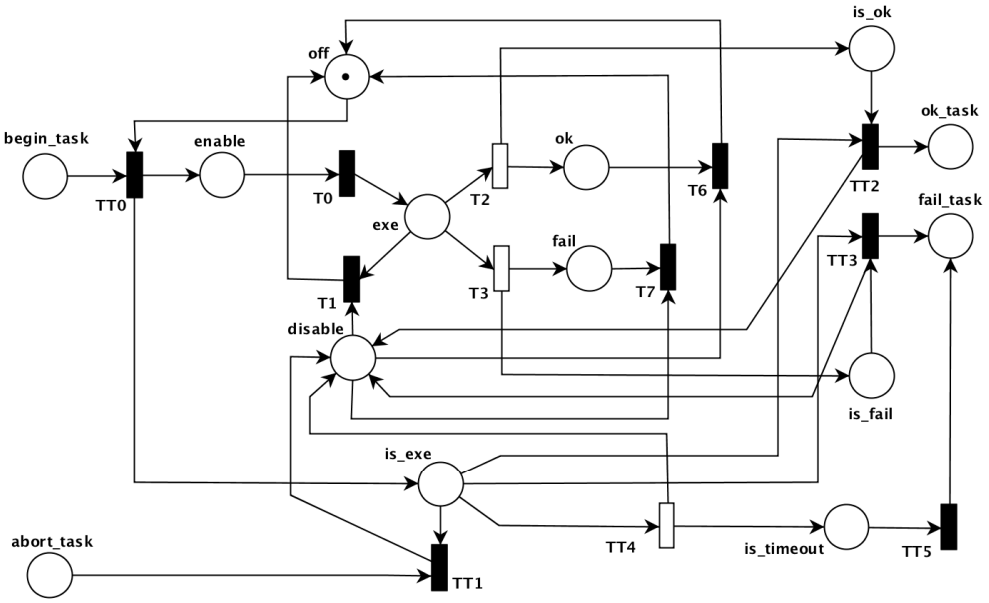


Figure 4.7: Example of a task PNBB with a timed transition ($TT4$) able to disable the execution of the supervised primitive if a time-out happens.

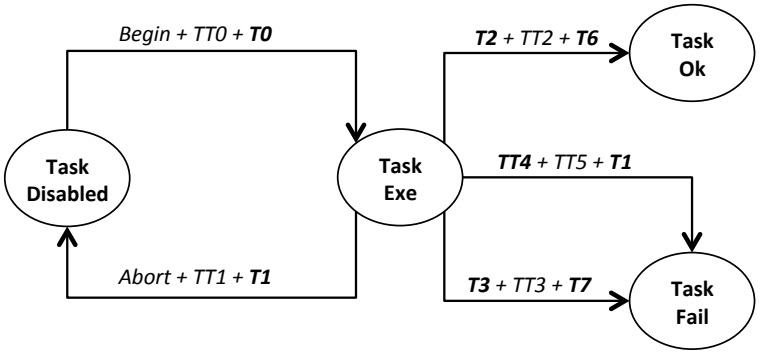


Figure 4.8: Reachability graph for the timed task PNBB shown in Figure 4.7.

control structures are used to aggregate tasks as well as other control structures with the objective of modeling more complex actions.

PNBBs are highly dependent on their interface. On one hand, the interface of a task depends on the primitive models being used. For instance, if a primitive that only generates an output event is used, a single input single output interface, i.e. Figure 4.3(a), could be used to supervise it. On the other hand, depending on the desired control flow between PNBBs, it is possible that a more complex interface could be needed. If a primitive that can generate two output events has to be controlled, from the point of view of a task, the interface presented in Figure 4.3(b) should be enough to supervise it, however, if we want to execute the primitive preemptively, the interface in Figure 4.3(d) must be used instead to be able to abort the task during its execution. As one of the requirements of our system is that all the PNBBs must share the same interface, the most general will be always used. In general, if the interface shown in Figure 4.3(a) is used, it is only possible to execute PNBBs in sequence or in parallel. Using an interface with two different outputs (*ok* and *fail*), as the one shown in Figure 4.3(b), conditional as well as iterative constructions are also possible. With the addition of an *abort* input place, see Figure 4.3(d), a rich set of preemptive control structures can be designed.

PNBBs are composed among themselves by means of their interface places. It is very interesting, for analysis purposes, to find a generic reduced model which, starting in the same initial state, reaches exactly the same final states [Oliveira and Silvestre, 2003]. Then, these generic reduced model may be used to set-up complex control structures. Careful designing those aggregated structures, we may enforce them to accomplish the same safety properties exhibited by primitive models and tasks, achieving a simple method for building safe mission plans through the composition of safe PNBBs, while avoiding tedious computational expensive verification methods.

Figure 4.9 presents a PNBB implementing the interface $I = \{begin, abort, ok, fail\}$ introduced in (4.5). From the interface point of view, this Petri net behaves in the same way that the tasks presented in Figure 4.4 and Figure 4.7. This means that the same reachability graph is produced. Then, it may be used as a reduced Petri net model to represent a generic PNBB.

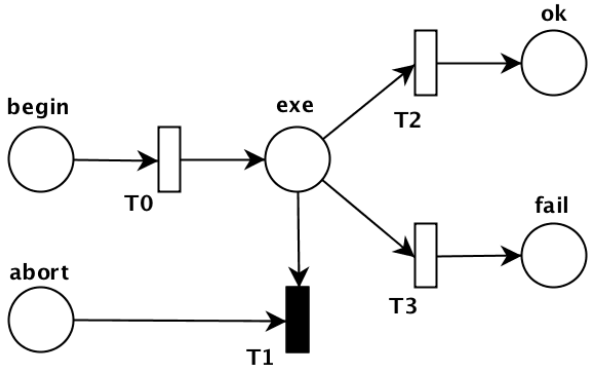
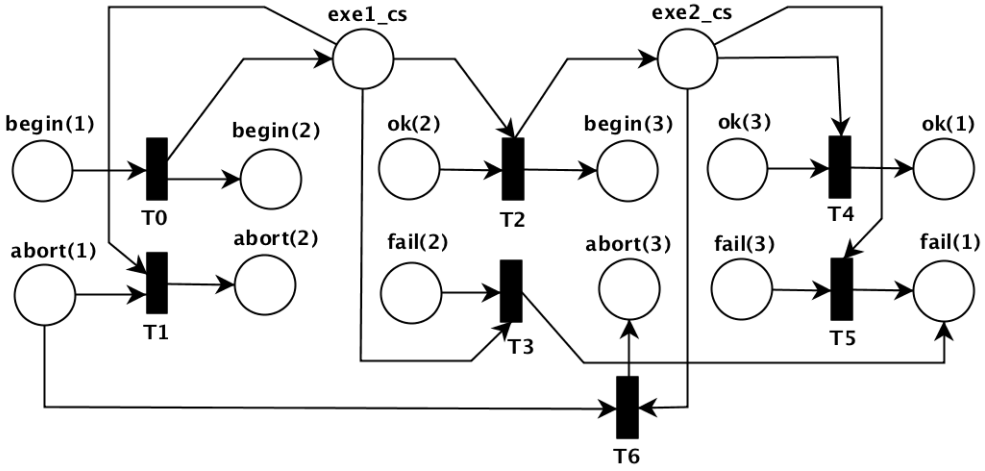
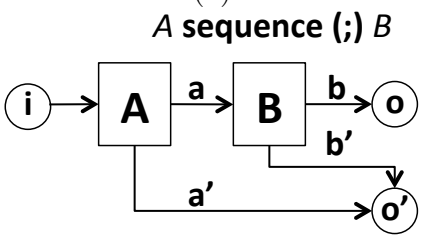


Figure 4.9: Example of a simplified PNBB.



(a)



(b)

Figure 4.10: (a) Control structure used to sequence two PNBBs and (b) its schematic model.

4.5.1 Sequence control structure

The control structure shown in Figure 4.10 is used to compose two PNBBs to be executed sequentially. While Figure 4.10(a) is the real Petri net control structure that carry out this functionality, Figure 4.10(b) is a schematic model of this sequence control structure. Blocks A and B, in the schematic model, represent the two PNBBs in the sequence. Arrows a and b are the respective *ok* outputs for each block while a' and b' correspond respectively to the *fail* outputs for the PNBBs A and B. The control structure model begins with the input i that goes towards the A block and ends with the output state o if *ok* or o' if *fail*.

The sequence control structure in Figure 4.10(a) contains the same interface that the PNBB in Figure 4.9 but triplicated: $I_1 = \{begin_1, abort_1, ok_1, fail_1\}$, $I_2 = \{begin_2, abort_2, ok_2, fail_2\}$ and $I_3 = \{begin_3, abort_3, ok_3, fail_3\}$.

Control structures not only have to implement an external interface, I_1, in order to be connected with other PNBBs but also one or more internal interfaces, here I_2 and I_3, composed by the same input/output places to connect PNBBs between them. For instance, if two PNBBs have to be sequenced, the external interface I_1, belonging to the first PNBB, must be composed with the internal interface I_2 in the control structure and the external interface I_1, belonging to the second PNBB, must be composed with the internal interface I_3 in the control structure as shown in Figure 4.11. If there is only one PNBB that have to be executed two times in sequence, it is also possible to compose the external interface I_1 of this PNBB task with both I_2 and I_3 internal interfaces in the sequence control structure. Then, places $begin_1$ and $abort_1$, in the task interface, will receive two input arcs while places ok_1 and $fail_1$ will have two output arcs each. Thus, a control structure contains one ore more internal interfaces in which other PNBBs can be connected through composition, and an external interface that can be used to connect this PNBB with other control structures hierarchically.

To compose two Petri nets using its interface the following operand is defined:

Definition 4.5.1. *Given a Petri net $N_A = \{P_A, T_A, A_A\}$ where: P_A is the set of all places in N_A, T_A is the set of all transitions in N_A and A_A is the set of all arcs in N_A and at least one interface I_j where $j \in \mathbb{N}/\{0\}$ and I_j is the set of all places in P_A that belongs to the interface j. Given a second Petri net with an interface I_k defined as $(N_B = \{P_B, T_B, A_B\},\ I_k)$, the composition operand can be applied to $(N_A, I_j) \oplus (N_B, I_k)$ if and only if the following conditions hold:*

$I_j \equiv I_k \rightarrow$ The input places and the output places in interfaces I_j and I_k must be the same (property of common interface).

$(P_A/I_j) \cap (P_B/I_k) = \emptyset \rightarrow$ The rest of the places in both Petri nets have to be different. This can be easily done by temporally adding a prefix to each place identifier before perform the composition.

$T_A \cap T_B = \emptyset \rightarrow$ The transitions of both Petri nets have to be different. This can be easily done by temporally adding a prefix to each transition identifier before perform the composition.

$A_A \cap A_B = \emptyset \rightarrow$ The arcs of both Petri nets have to be different.

The resulting Petri net after applying the composition operand $(N_A, I_j) \oplus (N_B, I_k)$ is $N_{AB} = \{P_{AB}, T_{AB}, A_{AB}\}$ where

P_{AB} contain $(P_A/I_j) \cup (P_B/I_k) \cup P'$ where P' is a set of places resulting from the fusion of both interfaces I_j and I_k. For each place p_j belonging to I_j and for each place p_k belonging to I_k where $p_j \equiv p_k$ a simple place p_{jk} is added to P_{AB}. Moreover, $\mu(p_{jk}) = \mu(p_j) + \mu(p_k)$ and $\bullet p_{jk} = \bullet p_j \cup \bullet p_k$ and $p_{jk}\bullet = p_j \bullet \cup p_k \bullet$.

T_{AB} contains $T_A \cup T_B$.

A_{AB} contains $A_A \cup A_B$.

Figure 4.11 shows the composition of the sequence control structure in Figure 4.10(a) with two PNBBs as the one presented in Figure 4.9. The sequence control structure is defined by $N_{CS} = \{P_{CS}, T_{CS}, A_{CS}\}$ where:

$P_{CS} = \{begin_1, abort_1, ok_1, fail_1, begin_2, abort_2, ok_2, fail_2, begin_3, abort_3, ok_3, fail_3, exe2_0, exe3_0\}$;

$T_{CS} = \{T_0, T_1, T_2, T_3, T_4, T_5, T_6\}$;

$A_{CS} = \{begin_1 \rightarrow T_0, abort_1 \rightarrow T_5, abort_1 \rightarrow T_6, T_0 \rightarrow begin_2, T_0 \rightarrow exe2_0, T_5 \rightarrow abort_2, exe2_0 \rightarrow T_1, exe2_0 \rightarrow T_2, T_6 \rightarrow abort_3, ok_2 \rightarrow T_1, fail_2 \rightarrow T_2, T_1 \rightarrow begin_3, T_2 \rightarrow fail_1, exe3_0 \rightarrow T_3, exe3_0 \rightarrow T_4, ok_3 \rightarrow T_3, fail_3 \rightarrow T_4, T_3 \rightarrow ok_1, T_4 \rightarrow fail_1\}$;

The two PNBBs are defined by $N_{PNBB1} = \{P_{PNBB1}, T_{PNBB1}, A_{PNBB1}\}$ and $N_{PNBB2} = \{P_{PNBB2}, T_{PNBB2}, A_{PNBB2}\}$ where

$P_{PNBB1} = \{begin_1, abort_1, ok_1, fail_1, exe_0\}$;

$T_{PNBB1} = \{T_0, T_1, T_2, T_3\}$;

$A_{PNBB1} = \{begin_1 \rightarrow T_0, abort_1 \rightarrow T_1, T_0 \rightarrow exe_0, exe_0 \rightarrow T_1, exe_0 \rightarrow T_2, exe_0 \rightarrow T_3, T_2 \rightarrow ok_1, T_3 \rightarrow fail_1\}$;

$P_{PNBB2} = \{begin_1, abort_1, ok_1, fail_1, exe_0\}$;

$T_{PNBB2} = \{T_0, T_1, T_2, T_3\}$;

$A_{PNBB2} = \{begin_1 \rightarrow T_0, abort_1 \rightarrow T_1, T_0 \rightarrow exe_0, exe_0 \rightarrow T_1, exe_0 \rightarrow T_2, exe_0 \rightarrow T_3, T_2 \rightarrow ok_1, T_3 \rightarrow fail_1\}$;

The composition shown in Figure 4.11 appears after compound the interface $I_{CS2} = \{ begin_2, abort_2, ok_2, fail_2\}$ with the N_{PNBB1} using the interface $I_{PNBB1} = \{ begin_1, abort_1, ok_1, fail_1\}$ and the interface $I_{CS3} = \{ begin_3, abort_3, ok_3, fail_3\}$ with the N_{PNBB2} using the interface $I_{PNBB2} = \{ begin_1, abort_1, ok_1, fail_1\}$.

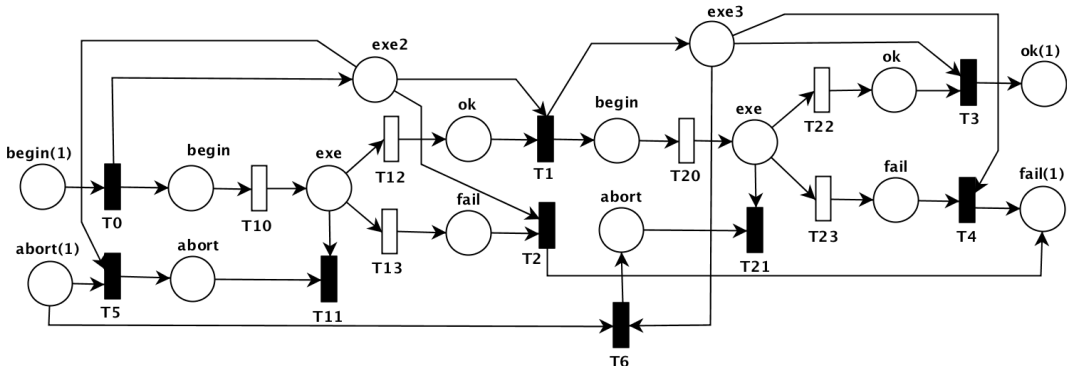


Figure 4.11: Example of a two simplified PNBBs composed with the sequence control structure.

4.5.1.1 Sequence control structure verification

Analyzing the sequence control structure once connected with two task PNBBs shown in Figure 4.11, no invariants, siphons or traps are found. The reachability graph is deadlock free and three different final states can be reached after firing all the enabled transitions like in Figure 4.6 or Figure 4.9:

1. When the control structure is aborted during its execution, the internal PNBBs are aborted in cascade and all the tokens are removed. The resulting state coincides with the initial state.

2. If both internal PNBBs finalize with an *ok* the sequence control structure finalizes with its place ok_1 marked.

3. If one of the internal PNBBs finalizes with a *fail*, the whole control structure finalizes with the $fail_1$ place marked.

The assumption in which the reachability analysis for the task PNBBs has been done says that: *"A task PNBB can only be aborted by the hierarchical superior PNBB that has started it and after it has been started."*. Looking at Figure 4.11 the only transition how aborts the first PNBB is $T5$ and $T5$ can not be fired until place *exe2* has a token. As *exe2* only receives a token when $T0$, that is the transition who starts the first PNBB, fires. Then, it is proved by construction that the PNBB will be aborted only after it has been started and by the same PNBB who has started it.

This deadlock and reachability analysis must be done in all the control structures used for defining a mission. Then, the resulting net after aggregating some PNBBs will be a new PNBB that satisfies the desired Petri net properties, as inherited from the original PNBB [Palomeras et al., 2008]. It is worth noting that it is not necessary to span the whole reachability graph 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 are guaranteed by construction and hence a mission plan, implemented according to these rules, progresses from its starting state to an exit state without sticking into a deadlock. It is also proved that this set of PNBBs is closed with respect to the composition operation. No need for post-verification is one of the main

differences between the proposed system and others who provide also verification capabilities.

4.5.2 Parallel control structure

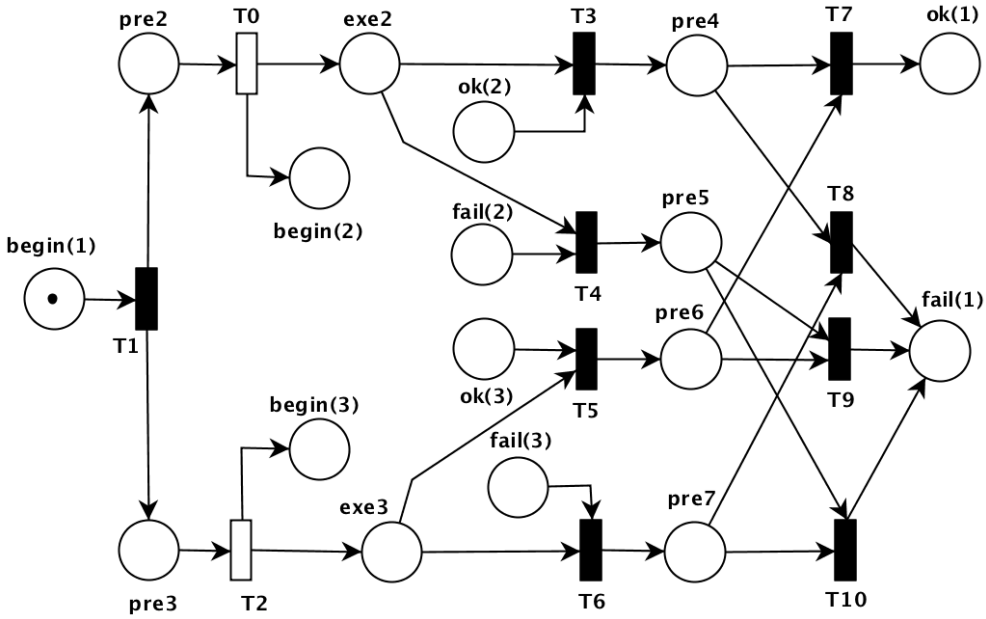


Figure 4.12: Example of a parallel-and control structure without an abort mechanism.

Another control structure is shown in Figure 4.12. Here, a *paralell-and* control structure able to execute two PNBBs in parallel is presented. It is important to note that the control structure in Figure 4.12 can not be aborted because it implements the interface shown in Figure 4.3(b). The same control structure but with all the mechanisms that allow to abort it are shown in the Appendix B.

When dealing with parallelism a question arises:

What happens if a user tries to execute the same task in two different execution threads that coincide in the time?

Figure 4.13 presents this problem. Note that Figure 4.13 contains the same *parallel-and* control structure shown in Figure 4.12 trying to initialize the same task in two parallel threads. To improve the readability of Figure 4.13, all the

places and transition belonging to the control structure, the task or the primitive model have the prefix *cs_*, *t_* or *pr_* respectively before its identifier except for the control structure interface (here *begin*, *ok* and *fail*). When the parallel-and control structure is started, the place *pr_off* in the primitive act as a mutual exclusion allowing only to fire *pr_T0* only once until the task has been disabled again. Thus, it is not possible to ensure which instantiation will be executed first, by the natural undeterministic behavior of Petri nets, but it is possible to ensure that if a task is already under execution, when a new instantiation is produced, this new instantiation will wait until the task finalizes to start it again with the new parameters provided by this second instantiation. Thus, what is really happening is that the task is automatically sequenced avoiding major problems instead of being executed in parallel.

4.5.3 Additional control structures

Based on the interface presented in Figure 4.3(d), some popular control structures are defined in our MCS while others may be defined by the mission programmer if necessary. Schemes of all these control structures are shown in Figures 4.14 and 4.15. Next, they are detailed.

- **Not**: It is used to negate the output of a PNBB, see Figure 4.14(a). Then, if the PNBB *A* finalizes with the place *fail* marked (*a'*), place *ok* in the Not structure will become marked (*o*) and vice-versa.
- **Sequence**: It is used to execute one PNBB after another, see Figure 4.14(b). If any PNBB finishes with a *fail* the whole structure finalizes with a *fail* (*o'*) otherwise it finalize in the *ok* state (o).
- **If-Then-(Else)**: Executes the PNBB inside the *If* statement, PNBB *A* in Figure 4.14(c) and (d), and depending if the PNBB ends with an *ok* or a *fail* the PNBB inside the *Then* statement, PNBB *B*, or the *Else* statement, PNBB *C*, if available is executed respectively.
- **While-Do**: Executes the PNBB inside the *While* statement, PNBB *A* in Figure 4.14(e). If this PNBB finishes with an *ok* executes the *do* statement, PNBB *B*, otherwise ends with an *ok* (o). If the *do* statement finishes with an *ok* executes again the *A* block. Otherwise ends the whole structure with a *fail* (o').

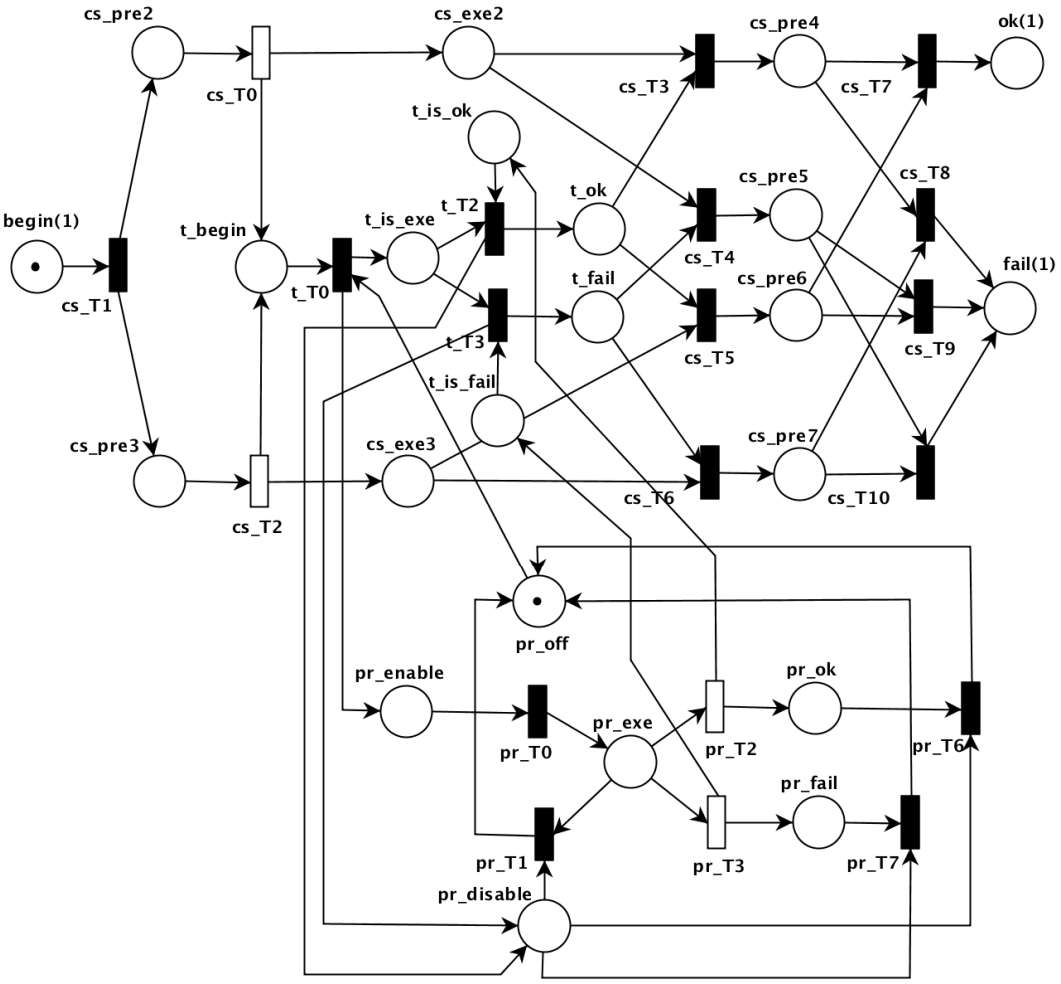


Figure 4.13: Example of a parallel control structure trying to execute the same task in parallel.

- **Try-Catch**: Executes the *Try* PNBB. If it finishes with a *fail* the *Catch* PNBB is executed as shown in Figure 4.14(f). The structure **Try** *A* **Catch** *B* is equivalent to the combination of the structures **If** (**Not**(*A*)) **Then** *B*, however, the necessary code when using this combination of control structures, see Chapter 5, becomes harder to follow than the use of the well-known structure *try-catch*.

- **Parallel-And**: Executes two PNBBs in parallel, see Figure 4.15(a). If both PNBBs finish in an *ok* place the whole control structure finishes with an *ok* (*o*). Otherwise, the *Parallel-And* finishes with a *fail* (*o'*). See that the structure finalizes only when both PNBB *A* and *B* have finished.

- **Parallel-Or**: Executes two PNBB in parallel. The first structure to finish aborts the other, see Figure 4.15(b). The *Parallel-Or* finishes with the final state of the first PNBB to end.

- **Monitor-Condition-Do**: Executes de PNBB *Monitor* and *Condition*, PNBBs *A* and *B* in Figure 4.15(c) and (d) in parallel. If the former finalizes the latter is aborted and the output is the formers output. Otherwise, if the later finalizes first, the *Monitor* PNBB is aborted and the *Do* statement, PNBB *C*, is executed.

- **Monitor-WhileCondition-Do**: The same structure than the **Monitor-Condition-Do** but when the *Do* PNBB finalizes with an *ok* instead of finalize the whole structure, the *Monitor* and *Condition* blocks are executed again, see Figure 4.15(d).

See that control structures *not*, *sequence*, *if-then-else*, *while-do* and *try-catch* don't allow parallelism. They only allow sequential, conditional and iterative control of tasks in a similar way than other popular languages. However, with the introduction of the *parallel* and *monitor* structures a high degree of parallelism can be achieved. This is possible because the inclusion of the *abort* input place in the interface that allows to cancel an execution thread when another one finalizes. Then, when building all these control structures special care has to be taken in order to be able to abort all the primitives that are running in a PNBB simultaneously as well as to remove all the tokens in the PNBB when the *abort* place is marked. The Petri net for each one of these control structures as well as a simplified schema are found in Appendix B.

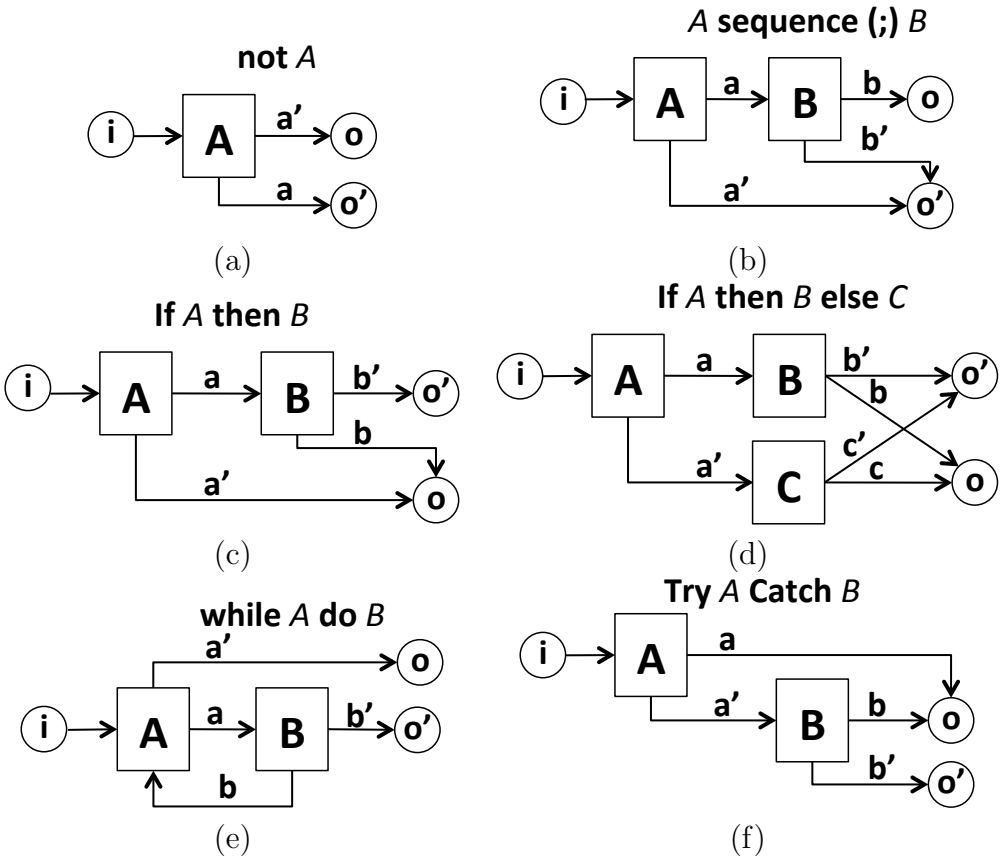


Figure 4.14: Non parallel control structures simplified models.

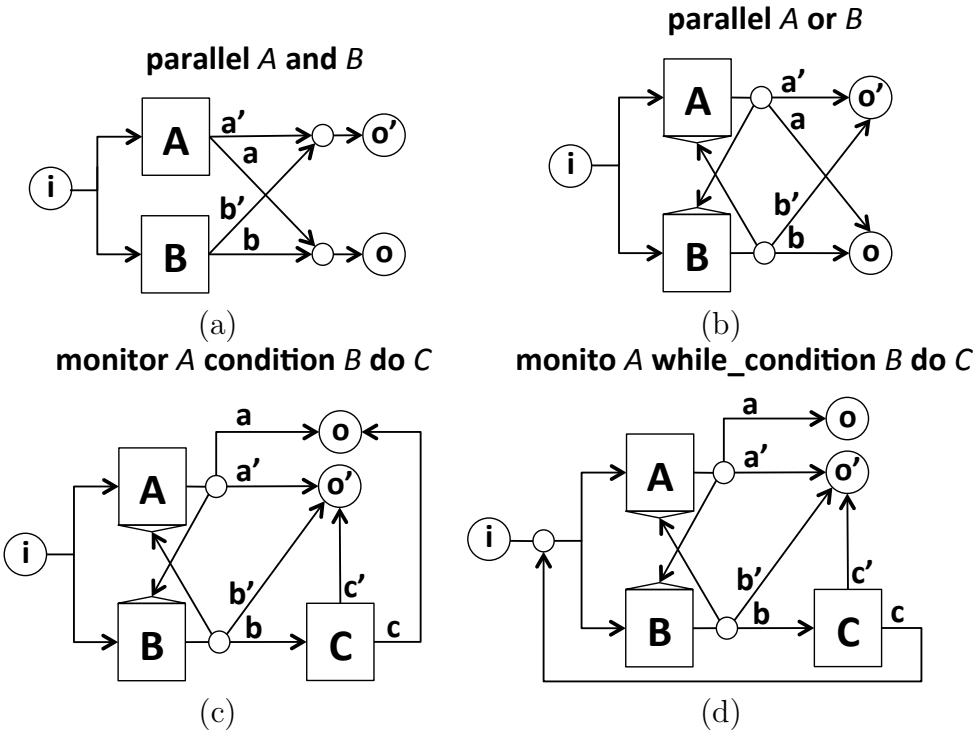


Figure 4.15: Parallel control structures simplified models.

Chapter 5

Mission Control Language

Previous chapter has presented a methodology to define a mission plan by connecting small Petri nets named Petri Net Building Blocks (PNBBs). The use of a formalism such as Petri nets provides several advantages. However, connecting PNBBs among them to build a complex mission can be a cumbersome and error prone work if the user has to deal directly with the Petri nets using graphical tools or even more if Petri nets are textually encoded. To avoid these problems, a Domain Specific Language (DSL) called Mission Control Language (MCL) is proposed. MCL allows to define the PNBBs and compose them in a simple way. Therefore, instead of using graphical tools to describe a mission or dealing with Petri nets textually encoded, our approach uses a DSL which automatically compiles into a Petri net. MCL is used to describe the *actions* and the *events* needed to communicate the Petri net tasks with the robot primitives. It also may specify the Petri net structures for the PNBBs, tasks and control structures, used in the mission, relating their transitions with the previously defined actions and events, in case of tasks, and defining which places belong to each interface. Once all these elements have been described a mission can be coded. Then Mission Control Language - Compiler (MCL-C), composes all the PNBBs used to define a mission among them to obtain a final Petri net mission plan that is described using a standard Extensible Markup Language (XML) format called Petri Net Markup Language (PNML) [web, 2010].

5.1 The MCL programming paradigm

The programming paradigm with more similarities to how MCL defines a mission by composing PNBBs is the *functional* where each PNBB can be seen as a function. For example, if a mission to submerge a robot 3 meters, navigate to the way-point (4, 6) and then surface have to be coded using the tasks *keepDepth* and *goto* and the *sequence* control structure, the following functional code will be obtained:

$$sequence(\ sequence(\ KeepDepth(\ 3\ ),\ Goto(\ 4,\ 6\ )\ ),\ KeepDetph(\ 0\ )\ ).$$

Using the functional programming paradigm, tasks are defined as functions whose parameters are the values needed by the primitives to operate. These values are essentially the parameters for the actions that have to be transmitted from the Mission Control System (MCS) to the vehicle primitives. Control structures are defined as functions in which their parameters are either tasks or other control structures. The output of each function, or PNBB, depends on the number of output places in its external interface. For instance, if the places belonging to the external output interface are $I_{output} = \{ok, fail\}$, then the output of each function is a boolean value (*ok* or *fail*).

Although the functional programming paradigm is the closer way to define a mission following the presented methodology, to simplify the operators work, a mission can be encoded in MCL using an *imperative* programming style. The imperative programming paradigm is more familiar and easier to understand for an operator than a functional one. Thus, the MCL has been developed to make use of it. Moreover, the additional constraints have been assumed to simplify even more the mission encoding process.

- The interface presented in Figure 4.3(d) has been selected to be used by all the PNBBs involved in the mission. This interface is composed by the places $I = \{begin, abort, ok, fail\}$
- It is possible to create new tasks, however, they have to implement the selected interface.
- Ten control structures, the ones presented in Section 4.5.3, have been predefined to be used within MCL. The user may modify the behavior of these control structures but no their interfaces. Additional control structures

may be added. However, they could not be used following the imperative paradigm, only the functional one.

Applying the presented constraints, the final user only has to deal with the definition of the following elements:

- Define the actions that each vehicle primitive may receive (enable and disable).
- Define the events that each vehicle primitive may raise (ok and fail).
- From the predefined task patterns or from a new one, instantiate the actions and events of each primitive in order to supervise it.
- Describe the mission plan using the previously instantiated tasks and the predefined control structures.

Once the vehicle primitives are defined by means of their actions and events and supervised by means of PNBB tasks, the only element that must be rewritten for each new mission is the mission plan.

Following sections describe the necessary elements to define a mission using MCL, the main algorithms used to translate a mission coded in MCL into a Petri net and how to execute the resulting Petri net mission plan.

5.2 Actions and events

The actions used to enable/disable the primitives as well as the events that can be generated by them must be specified in the MCL program. Actions are defined using the command

$$actions\{\ (action_{id} = primitive(\ command,\ list\_of\_parameters\ ))^*\ \}$$

where the *command* is the activity to be executed by the primitive (normally enable or disable it) and the *list of parameters* are the variables that must be specified by the user in the mission plan, see Section 5.6, when the task is called. Events are defined by the command

$$events\{\ (event\_id)*\ \}$$

Extract 5.1 shows an example in which the actions and events used by the *Goto()* task are defined. The mapping of actions and events is implemented in the Architecture Abstraction Component (AAC). Each time that an action is sent from the MCS through the AAC, the latter has to ask to the related vehicle primitive to execute the command with the parameters instantiated in the mission plan. Similarly, each time that a primitive raises an event the AAC has to map this event with an *event_id* and send it to the MCS. Hence, this mapping is used to tailor the MCS to a particular set of primitives defined in the AAC.

Extract 5.1: Actions & events definition.

```
actions {
    enableGoto = GotoPrimitive( c: enable, v: way-point ) ;
    disableGoto = GotoPrimitive( c: disable ) ;
    ...
}
events {
    eventGotoOk ;
    eventGotoFail ;
    ...
}
```

5.3 PNBB patterns

To define a new PNBB, it is necessary to specify its structure composed by a set of places (P), a set of transitions (T) and a set of arcs (A). To do it, the following commands are used:

$$places\{\ (place\_id\ (number\_of\_tokens).interface\_id)*\ \}$$
$$transitions\{\ (transition\_id)*\ \}$$
$$arcs\{\ (source\_id \rightarrow destination\_id)*\ \}$$

The interface that each place belongs has to be indicated by ($interface\_\ id$). If the *interface_id* is 0 means that the place does not belong to any interface, however, places belonging to the external interface have $interface\_id = 1$ and places belonging to an internal interfaces, used in control structures to specify how PNBBs are composed, have values from $interface\_id = 2...n$. Both *number_of_tokens* and $interface\_id$ are set to 0 by default. A graphic Petri net editor

with PNML support, for instance the *pipe2* [Bonet et al., 2010], may be used to define a PNBB pattern instead of defining it textually.

The PNBB pattern for the task shown in Figure 4.4 is described in Extract 5.2 using the MCL notation.

Extract 5.2: Building Block Pattern definition.

```
AchieveOneGoal {
    places {
        begin.1; abort.1; ok.1; fail.1; is_exe; is_ok; is_fail; enable; disable;
        off(1); exe; primitive_ok; primitive_fail;
    }
    transitions {
        TT0; TT1; TT2; TT3; T0; T1; T2; T3; T6; T7;
    }
    arcs {
        begin.1 → TT0; TT0 → is_exe; TT0 → enable; enable → T0; T0 →
        exe; exe → T1; exe → T2; exe → T3; T1 → off; off → TT0; T2 →
        primitive_ok; T3 → primitive_fail; primitive_ok → T6; primitive_fail
        → T7; T6 → off; T7 → off; T2 → is_ok; T3 → is_fail; TT1 →
        disable; disable → T1; disable → T6; disable → T7; TT2 → disable;
        TT3 → disable; is_ok → TT2; is_exe → TT2; is_exe → TT1; is_exe
        → TT3; is_fail → TT3; abort.1 → TT1; TT2 → ok.1; TT3 → fail.1;
    }
}
```

5.4 Tasks

To define a task, a PNBB pattern must be connected to the set of actions and events belonging to a primitive. Once the PNBB pattern used to supervise a primitive is chosen, the actions and events related to this primitive must be associated to the corresponding transitions in the pattern. Tasks have a header composed by the list of all the parameters used in their actions. For instance, if a task has to execute the action *enableGoto* it must include in its header the parameter *way-point* used by this action. Extract 5.3 shows an instance of the task pattern presented in Extract 5.2 that is used to supervise the primitive *goto* that guides a robot towards a way-point.

Extract 5.3: Task definition.

Goto(*way-point*): **AchieveOneGoal {**
a: *enableGoto* → T0;
a: *disableGoto* → T1, T6, T7;
e: *eventGotoOk* → T2;
e: *eventGotoFail* → T3;
}

5.5 Control structures

The ten control structures presented in Section 4.5 are implemented in the MCL: not, sequence, if-then, if-then-else, while-do, try-catch, parallel-and, parallel-or, monitor-condition-do and monitor-while_condition-do. When a control structure is used to aggregate two structures, i.e. sequence or parallel, two sets of internal interfaces must be provided. If a control structure is used to aggregates three control structures, i.e. if-then-else or monitor-condition-do, three sets of internal interfaces must be provided instead. It is possible to extend the MCL to include new foreseen control structures. Nevertheless, if new structures are defined, the MCL have to be used as a functional programming language rather than an imperative one.

5.6 Mission plan

Once the tasks and the control structures have been defined, a mission plan can be coded using MCL. This is the only section that must be rewritten for each new mission if the same set of primitives is used. Although MCL is in essence a functional language, the developed compiler is able to understand predefined control structures in an imperative form. As predefined control structures are very similar to those provided by other popular languages and tasks can be seen as function calls, programming a new mission using MCL becomes very simple.

Extract 5.4 shows how the MCL is used to program a very simple mission. In this example the vehicle has to achieve a desired depth and then go to a way-point while keeping this depth. Once the way-point is achieved the vehicle surfaces. If any error is produced, the vehicle enables a recovery beacon. Note that the

semicolon (*;*) symbol is used to sequence two PNBBs instead of delimiting the end of a sentence as usual in popular imperative languages like *C* or *Java*. Then, the semicolon (*;*) is used to sequence the task *Depth(15, "achieve")* with the *parallel-or* control structure and this one with the *Depth(0, "achieve")* task. However, as the two blocks inside the *parallel-or* control structure contain only one element, a task, it is not necessary to add a semicolon at the end of each one, because no sequencing is required. The same happens by the last PNBB of each block in the *try-catch* control structure.

Extract 5.4: MCL mission definition.

```
mission
    try
        Depth( 15, "achieve" ) ;
        parallel
            Depth( 15, "keep" )
        or
            Goto( 24, 12 )
        ; Depth( 0, "achieve" )
    catch
        StartRecoveryBeacon()
```

The same mission could be coded in a functional way as shown in Extract 5.5. However, its is more difficult to read.

Extract 5.5: Funcional mission definition.

```
mission(
  try-catch(
    seq(
      seq(
        Depth( 15, "achieve" ),
        parallel-or( Depth( 15, "keep" ), Goto( 24, 12 ) )
      ),
      Depth( 0, "achieve" )
    ),
    StartRecoveryBeacon() )
)
```

For a formal description about the MCL grammar using the Backus Normal Form (BNF), the reader is pointed to the Appendix C which includes a description of all the predefined control structures as well as all the necessary sections to program a mission using MCL.

5.7 The Mission Control Language - Compiler

The process to generate a Petri net from a MCL program is performed by the MCL-C. This process is divided in four main steps:

1. Generate a Petri net for each PNBB pattern.
2. Add the actions and the events to the Petri net patterns in order to build the PNBB tasks.
3. Use the mission plan to generate an Abstract Syntax Tree (AST) in which the nodes are the control structures and the leaves are the tasks.
4. Traverse the AST composing first the control structures among them and then the tasks to build the whole Petri net mission plan.

The procedures involved to implement steps (1) and (2) are trivial. While the code is being parsed by the compiler, a Petri net for each pattern is generated and then actions and events are connected to a copy of these patterns creating the tasks. The compiler only checks if the arcs in the patterns are correctly connected with valid places and transitions or vice-versa. The MCL-C also checks that all the parameters used by the task actions appear also in the task header.

Step (3) can be seen as the translation of the imperative code written by the user into a functional code. This process consists of building an AST, Figure 5.1(a), for instance, shows how the code in Extract 5.4 is transformed into an AST. Finally, in step (4), a recursive algorithm named *composeMission*, shown in Extract 5.6, is applied. The previously generated AST, named *t*, and an initially empty vector of tasks, named *vt*, relating how each task external-interface has to be composed with the control-structure internal-interfaces, are used as input parameters. The algorithm explores the AST and composes all the control structures recursively following Definition 4.5.1. When a task is found during this

process, if it is not included in vector *vt*, the *interface_id* of the places belonging to its external interface are renumbered and the modified task is included in vector *vt*. When the whole AST has been explored, every task in the vector *vt* is composed with as many internal interfaces as necessary. Note that tasks are not replicated, instead, only one task PNBB is included in the final Petri net mission plan, even though it can be called several times from different control structures, see Figure 5.1(b) and Figure 4.13 in previous chapter. The no replication of tasks is due to each task is connected to a single vehicle primitive that may depend on vehicle resources that don't have to be replicated.

The resulting Petri net after compiling the mission shown in Extract 5.4 includes three tasks (*Depth*, *Goto* and *StartRecoveryBeacon*) and four control structures (two *sequences*, one *parallel-or* and a *try-catch* that wraps the whole mission).

5.8 The real-time Petri net player

Once the MCL mission has been compiled, a single Petri net called mission plan and coded using PNML is obtained. A particular extension had to be introduced in the PNML in order to properly implement the communication facilities offered by the AAC in the language, namely to define the actions that are sent from the Petri Net Player (PNP) to the vehicle control architecture and the events generated there to the PNP. The PNP executes in real-time the Discrete Event System (DES) described by the Petri net mission plan applying the basic Petri net transition rule. To do it, the PNP has to fire enabled transitions, send the vehicle primitive actions through the AAC and fire the event-related enabled transition when the corresponding event is received from a vehicle primitive. This component is implemented in C++ and uses the Transmission Control Protocol (TCP) to communicate with the AAC. Then, to execute an MCL mission for a particular Autonomous Underwater Vehicle (AUV) it is only necessary to build and AAC, within the AUV control architecture, and communicate this component with the PNP using standard TCP sockets. Section 3.2.3 details more information about this topic. Next, the three main algorithms implemented in the PNP are shown.

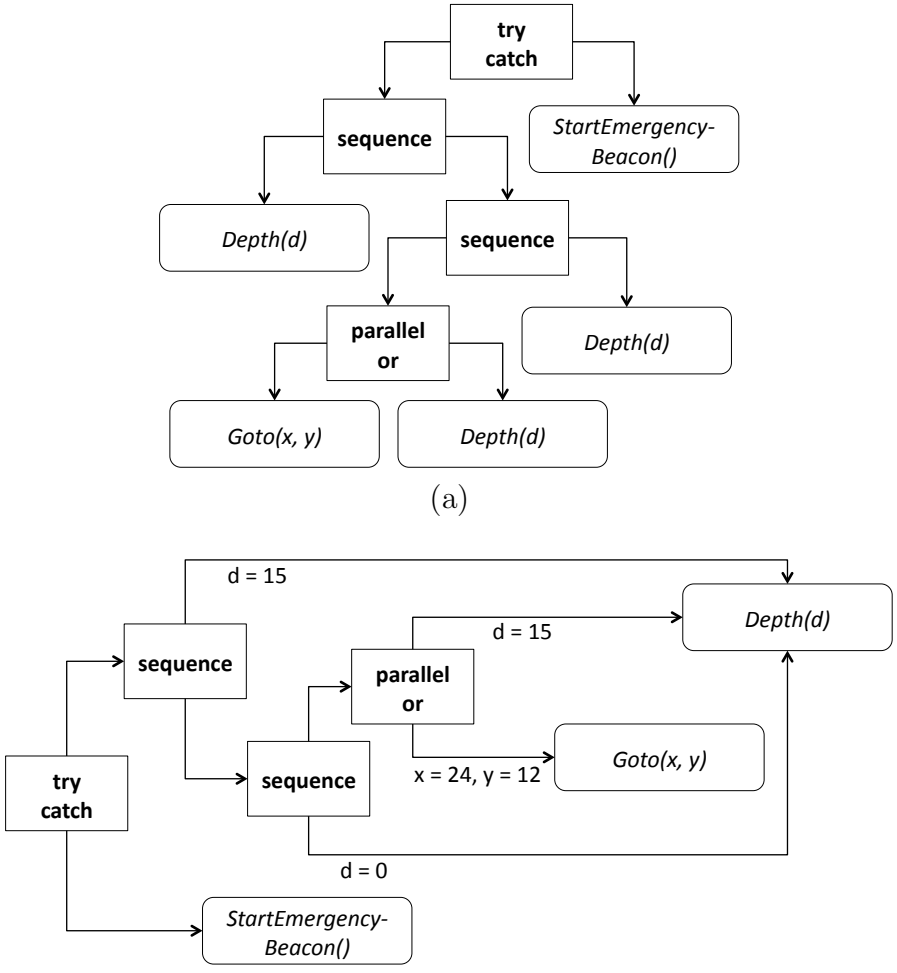


Figure 5.1: (a) AST from Example 4. (b) AST from Example 4 once separated tasks from control structures.

Extract 5.6: function *composeMission.*

```
Input: AST t, vector<task> vt
Output: PetriNet pn
if t.root = "mission" then
    /* If the tree root tag is 'mission' call
    composeMission(..)  for its first child                    */
    pn = composeMission( 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 ) ;
else
    /* else, load the indicated control structures            */
    pn = load( ControlStructure[t.root] ) ;
    /* For every child, do                                    */
    for i = 0 to size( t.child ) do
        if t.child[i].type = "task" then
            /* When a task is found change the interface id.  */
            changeInterfaceId( 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
            composeMission(..)  for this child                 */
            pnTmp = composeMission( t.child[i], vt ) ;
            /* compose both Petri nets                        */
            pn = compose( pnTmp, pn ) ;
```

Extract 5.7: function *receiveEvent.*

```
Input: PetriNet pn, Event ev
boolean found = false;
iterator<Transition> t = pn.transition.begin() ;
while not found and t ≠ pn.transition.end() do
    if t → ev = ev and isEnabled( pn, t ) then
        found = true ;
        fire( pn, t ) ;
    else
        t++ ;
```

The code in Extract 5.7 is executed each time that an event is received through the AAC. This function checks if any transition in the Petri net mission plan is related with the received event. If there is a related transition, and it is enabled, the usual case, the transition is fired. The code shown in Extract 5.8 is executed when the mission starts, or each time that a transitions fires because and event has been received. Function *enabledTransitions()* returns all enabled transitions except the ones that are related with an event. Next, function *ready2fireTransitions()* takes the subset of transitions that are ready to fire, immediate transitions as well as those timed transitions whose timers have expired, and also, increments the timer for all enabled timed transitions. From the set of ready to fire transitions, function *selectRandomly()* choses a random transition. Once the transition to fire is selected it is fired using the code presented in Extract 5.9.

Extract 5.8: function *playPetriNet.*

```
Input: PetriNet pn
/* Take all the enabled transitions in the Petri net mission
plan                                                        */
q = enabledTransitions( pn ) ;
while size( q ) > 0 do
    /* Take all the transitions ready to fire in q          */
    qf = ready2fireTransitions( q ) ;
    if size( qf ) > 0 then
        /* Select, randomly, a transition t ∈ qf            */
        t = selectRandomly( qf ) ;
        fire( pn, t ) ;
        q = enabledTransitions( pn ) ;
    else
        /* Wait to see if any timed transition is ready to fire
        in a while                                          */
        wait() ;
```

Firing a transition t, involves removing a token from all the places in the pre-set of t, $\bullet t$, adding a token to all the transition in the post set of t, $t\bullet$, and sending all the actions associated to the transition t, through the AAC, to the corresponding primitives.

Extract 5.9: function *fire*.

Input: PetriNet pn, Transition t

for ***action*** $a \in t$ **do**

```
    /* Send all the available actions of the ready-to-fire
    transition t                                              */
    sendAction( a ) ;
```

for ***place*** $p \in \bullet t$ **do**

```
    /* Remove 1 token from all the places in the pre set of t
    */
    μ(p) = μ(p) − 1 ;
```

for ***place*** $p \in t\bullet$ **do**

```
    /* Add 1 token to all the places in the post set of t     */
    μ(p) = μ(p) + 1 ;
```

Figure 5.2 shows the whole picture about how a mission is defined an executed using the proposed MCS. The mission program is written by the user in MCL, then the MCL-C transforms this code into a Petri net coded in PNML. This PNML can be seen as the byte-code that is finally executed by the PNP in real time on the vehicle control architecture.

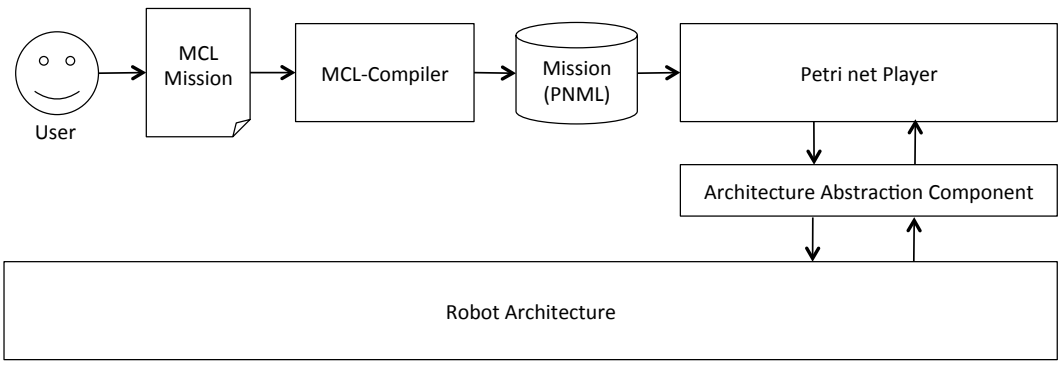


Figure 5.2: Mission definition and execution schema.

Chapter 6

Coordination of multiple vehicles

The purpose of this chapter is to further develop the presented methodology to deal with the coordination of multiple vehicles. Lets imagine a mission in which a couple of AUVs, each one equipped with a robotic arm, has to collaborate to build an underwater construction. If each AUV and manipulator is considered as an independent robot, a set of constraints must be defined between them to coordinate their work.

Coordination of multiple vehicles can be studied from different points of view. From Artificial Inteligence (AI) techniques based on auction mechanisms [Murillo et al., 2007] to searching algorithms for coordination and cooperation [Caiti et al., 2008]. In the underwater domain, several control algorithms to perform multiple vehicle formations [Edwards et al., 2004] or cooperative path-following [Vanni et al., 2008] have been developed recently. However, our idea presents a completely different approach. It consists of describing an independent missions for each entity and then, coordinate the execution of these missions introducing some constraints between them. Therefore, the proposed method follows these steps:

1. Program a mission for each robot obtaining an independent Petri net for each one as presented in previous chapters.

2. Define a set of constrains to enforce a coordinated behavior among all the robots involved in the mission.

3. Automatically synthesize the necessary Supervision Based on Place Invariants (SBPIs) to carry out each constraint.

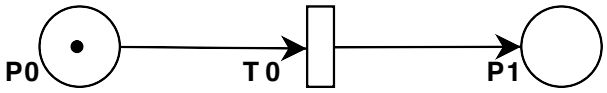


Figure 6.1: Simplified sequence of two tasks.

4. Connect the independent robot mission between them by means of the SBPIs previously synthesized generating a centralized Petri net mission in which all the constraints are satisfied.

5. Check that the centralized mission is deadlock free.

6. Partitioning the centralized Petri net mission into as many decentralized Petri nets as entities involved in the mission, keeping the same behavior than in the centralized net while minimizing the communication between the decentralized nets.

Three coordination constraints have been studied: mutual exclusions, tasks ordering and tasks synchronization. To simplify the Petri net mission plan, obtained when compiling a mission coded in MCL, a more compact representation, than the one presented in previous chapters, will be used to explain how the constraints work. In this representation each place acts either a task instantiation or a slack place. When a place representing a task is marked, has a token, it means that the task is under execution. All places representing a task are 1-bounded. Control structures are replaced by slack places that guide the control flow among task-places. These simplifications are done to improve the comprehension of the presented multiple-vehicle coordination mechanism. To illustrate this compact representation Figure 6.1 shows the sequence of two tasks, represented by places $P0$ and $P1$, in which $P0$ is under execution. This figure can be compared with Figure 4.11 in which two tasks are also sequenced. However, while in Figure 4.11 all the mechanisms to send and receive actions and events as well as to abort the tasks are shown, the simplified representation presented in Figure 6.1 hides all this stuff.

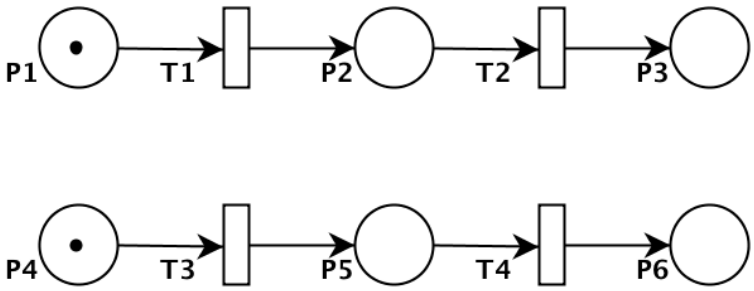


Figure 6.2: Example of two simple Petri net missions.

6.1 Coordination constraints

In following sections, mutual exclusion, ordering and synchronization constraints used for achieve multiple-vehicle coordinated missions are detailed. To illustrate the coordination constraints, Figure 6.2 shows a compact representation of two missions. Both missions describe the sequence of three tasks represented by $P1$, $P2$ and $P3$ for the first entity and $P4$, $P5$ and $P6$ for the second.

The incidence matrix, see Appendix A, for the first vehicle mission (D^{v1}) and for the second vehicle mission (D^{v2}) as well as their initial markings (μ_0^{v1} and μ_0^{v2}) are defined as

$$D^{v1} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \tag{6.1}$$

$$\mu_0^{v1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{6.2}$$

$$D^{v2} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \tag{6.3}$$

$$\mu_0^{v2} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{6.4}$$

6.1.1 Mutual exclusion

The first constraint to study is the mutual exclusion, popularly called *mutex*. A typical example of a mutex appears when several vehicles can not be in the same location at the same moment. For instance, imagine that two AUVs, used to build an underwater infrastructure, share a single docking station to recharge their batteries. The access to this resource should be protected by a mutex.

Formally, a mutex is defined as:

Definition 6.1.1. *A mutual exclusion is a pair (M, β), where M is the set of tasks, represented as places, involved in the mutual exclusion and $\beta \in \mathbb{N} \setminus \{0\}$ is the maximum number of tasks in M which can be simultaneously under execution (e.g. if $\beta = 1$, then the maximum number of marked places in M must be 1).*

For instance, if the tasks represented by places $P2$ and $P5$ in Figure 6.2 are in mutual exclusion ($M = \{P2, P5\}$) and only one of the tasks can be running simultaneously ($\beta = 1$), then the SBPI described by Proposition 6.1.1 can be used to connect both Petri net missions in order to build a centralized Petri net in which the constraint is enforced.

Proposition 6.1.1. *Let W be a set of independent Petri nets and (M, β) a mutual exclusion defined over W. If constraint (6.5) holds, the mutual exclusion property defined above is satisfied.*

$$l = [l_1 \cdots l_i \cdots l_n], \text{ with } l_i = \begin{cases} 1 & \text{if } p_i \in M \\ 0 & \text{otherwise} \end{cases} \qquad l\mu \leq \beta \text{ where} \tag{6.5}$$

Where l is a vector of integers with as many elements as places in the centralized Petri net (n) and μ is the marking vector of the centralized Petri net. See Appendix A for an introduction about linear state constraints and SBPIs.

Proof. Because only the tasks $l_i \in l$ related to places $p_i \in M$ are set to 1 and the rest of them are set to 0 and the task places are 1-bounded, $l\mu$ gives the number of places in M simultaneously marked. Hence, if a marked place represents a

running task and the constraint $l\mu \leq \beta$ holds, it follows that no more than β tasks in M can be simultaneously under execution. □

Applying the Proposition 6.1.1 to the Petri nets in Figure 6.2 to impose the mutual exclusion ($M = \{P2, P5\}, \beta = 1$), the following constraint is obtained.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mu(p_1) \\ \mu(p_2) \\ \mu(p_3) \\ \mu(p_4) \\ \mu(p_5) \\ \mu(p_6) \end{bmatrix} \leq 1 \;=\; \mu(p_2) + \mu(p_5) \leq 1 \tag{6.6}$$

If both Petri nets D^{v1} and D^{v2} presented in equations (6.1) and (6.3) are combined in a centralized net, the corresponding incidence matrix will be given by

$$D^{v1\&v2} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.7}$$

with the following initial marking

$$\mu_0^{v1\&v2} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{6.8}$$

To apply the SBPI computed in (6.6) to the centralized matrix in (6.7), (6.8)

equations (A.10) and (A.11) are used.

$$D_c = -\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 1 & -1 & 1 \end{bmatrix} \tag{6.9}$$

$$\mu_{c0} = 1 - \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 1 \tag{6.10}$$

Once (6.9) and (6.10) are computed the centralized Petri net that enforces the constraint ($M = \{P2, P5\}, \beta = 1$) is obtained applying (A.12) and (A.13).

$$D = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \tag{6.11}$$

$$\mu_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{6.12}$$

The Petri net obtained in (6.22), (6.23) is shown in Figure 6.3. It is worth

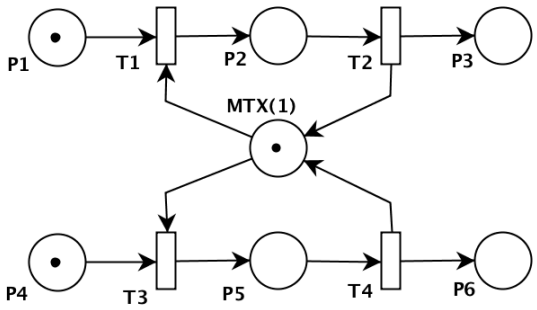


Figure 6.3: Example of the mutual exclusion ($M = \{P2, P5\}$, $\beta = 1$).

noting that if transition $T1$, in the first vehicle, fires taking the mutual exclusion, task $P4$, in the second vehicle, can not finalize until $P2$ has been disabled, $T2$ has fired. To avoid this, a slack place has been added before every $P_i \in M$. A simple transformation called *PW-Transformation* has been applied for this purpose. The goal of this transformation is to add a slack place before the places involved in a constraint to untie the finalization of these places with the execution of the successive places.

Definition 6.1.2. *The PW-Transformation over a place P_i in the Petri net $N = \{P, T, A\}$ is defined as $P' = P \cup \{P_i^{wait}\}$, $T' = T \cup \{T_i^{wait}\}$ where $\bullet P_i^{wait} = \bullet P_i$, $P_i^{wait}\bullet = \{T_i^{wait}\}$, $T_i^{wait}\bullet = \{P_i\}$ and $\bullet P_i = \{T_i^{wait}\}$.*

The PW-Transformation does not modify the Petri net behavior, however, the additional waiting place allows a previous task to finalize even if the next task in the sequence can not be executed due to a mutual exclusion or any other constraint. Figure 6.4 shows the final centralized Petri net once the constraint ($M = \{P2, P5\}, \beta = 1$) and the PW-Transformation have been applied.

6.1.2 Ordering

The notion of order appears quite naturally when describing distributed systems. Ordering between two tasks appears when one task can be only executed after the termination of another one. Recalling the AUV builders scenario, if the robotic arm that each AUV includes is considered an independent entity, two ordering constraints must be established between the vehicle and the manipulator to move a block. First, the vehicle must be moved to the position in which the building

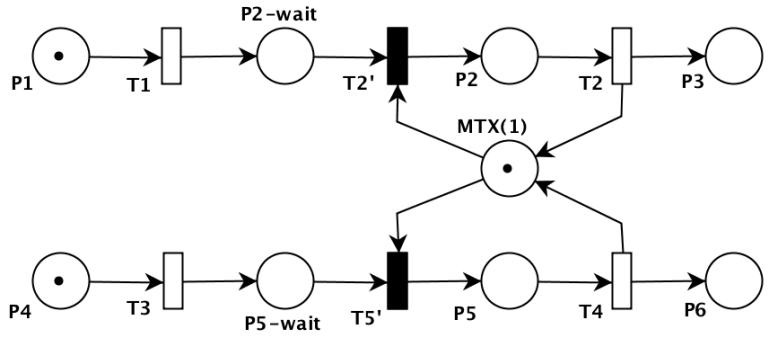


Figure 6.4: Example of the mutual exclusion ($M = \{P2, P5\}$, $\beta = 1$) after apply the PW-Transformation.

block is and then, the manipulator can grasp it. Next, the AUV have to navigate to the position in which the building block has to be released, only once the manipulator has grasped and lifted the block.

Using these simplified Petri net missions, launching, enabling, a task represented by the 1-bounded place p with $\mu(p) = 0$ consists in marking p with one token, $\mu(p) = 1$. On the other hand, the termination of a task p currently under execution, $\mu(p) = 1$, consist in removing the token from p, $\mu(p) = 0$. Then, a formal definition of an ordering between two tasks is:

Definition 6.1.3. *An ordering pair is defined as a set of two places* $O = \{p_s, p_w\}$ *where* p_w*, that stands for the* waiting *place, can not be marked before* p_s*, that stands for the* signaling *place, has been unmarked.*

Again, a PW-Transformation must be applied before adding the ordering constraint. This transformation must be applied to the place $p_w \in O$.

To ensure the ordering constraint, an extended form of $l\mu \leq b$ is used. This form

$$cv \leq b \tag{6.13}$$

is described in Iordache et al. [2002b] and uses the Parikh vector (v) to control the Petri net behavior. The Parikh vector contains a counter for every transition in the system. Every counter is initialized to 0 and when a transition t_j fires, the corresponding element v_j in the Parikh vector is incremented. For ordering the two places contained in an ordering pair, transitions $p_s\bullet$ and $\bullet p_w$ are used.

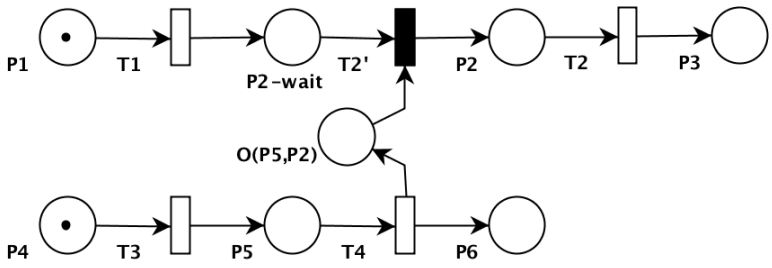


Figure 6.5: Example of the ordering $O = \{P5, P2\}$ after applying the PW-Transformation.

Proposition 6.1.2. *Let W be a set of mission Petri nets and $O = \{p_s, p_w\}$ an ordering pair defined over W, then it can be shown that if the constraint (6.14) holds, the properties of the ordering pair are satisfied.*

$$\mathbf{c} \cdot \upsilon \leq 0 \text{ where}$$
$$\mathbf{c} = [c_1 \cdots c_j \cdots c_m], \text{ with } c_j = \begin{cases} -1 & \text{if } t_j \in p_s\bullet \\ 1 & \text{if } t_j \in \bullet p_w \\ 0 & \text{otherwise} \end{cases} \tag{6.14}$$

Where $\mathbf{c}$ is a vector of integers with as many elements as transitions in the centralized Petri net (m) and υ is the Parikh vector of the centralized Petri net. For an introduction about about general linear vector constraints in Petri nets see Iordache and Antsaklis [2002].

Proof. With the Parikh vector it is possible to register the number of firings of each transition. To ensure the order between both tasks, the number of firings of the transition immediately after the place p_s, $p_s\bullet$, must be always equal or greater than the number of firings of the transition before the place p_w. If the transition before place p_w, $\bullet p_w$, fires before the transition after the place p_s, the constrain synthesized in *Proposition 6.1.2* will be bigger than 0 making itself false. □

Applying the Proposition 6.1.2 to the Petri nets in Figure 6.2 to impose the

ordering $O = \{P5, P2\}$, the following constraint is obtained:

$$\begin{bmatrix} 1 & 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} v(t_1) \\ v(t_2) \\ v(t_3) \\ v(t_4) \end{bmatrix} \leq 0 \;=\; v(t_1) - v(t_4) \leq 0 \tag{6.15}$$

To apply a supervisor using the Parikh vector, a general linear constraint has to be used. An extract of the equations presented in Iordache and Antsaklis [2002] is presented here for the readers convenience. The SBPI shown in (A.9) is enhanced with the Parikh vector term, being

$$L\mu + Cv \leq b. \tag{6.16}$$

Then, to compute the general linear constraint the following equations are used:

$$D_c^+ = max(0, -LD - C) \tag{6.17}$$

$$D_c^- = max(0, LD + C) \tag{6.18}$$

Where the *max* operator is defined as follows.

Definition 6.1.4. *If A is a matrix, $B = max(0, A)$ is the matrix of elements $B_{ij} = 0$ for $A_{ij} < 0$ and $B_{ij} = A_{ij}$ for $A_{ij} \geq 0$.*

Then, following the ordering example and applying (6.17), (6.18) and (A.11)

$$D_c^+ = max(0, \begin{bmatrix} -1 & 0 & 0 & 1 \end{bmatrix}) = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.19}$$

$$D_c^- = max(0, \begin{bmatrix} -1 & 0 & 0 & 1 \end{bmatrix}) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \tag{6.20}$$

$$\mu_{c0} = 0 - \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 0 \tag{6.21}$$

the controlled Petri net D and its marking vector μ_0 are obtained.

$$D = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \tag{6.22}$$

$$\mu_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{6.23}$$

Figure 6.5 shows an example of the ordering constraint $O = \{P5, P2\}$ applied over the nets presented in Figure 6.2 where, additionally, the PW-Transformation has been applied to $P2$.

6.1.3 Synchronization

Synchronization constraints are used to fix rendezvous among tasks allowing them to be launched simultaneously. Following the autonomous underwater builders scenario, if some building blocks are too heavy to be lifted by a single robotic arm, a synchronization constraint can be forced between two autonomous builders to do the task together.

Definition 6.1.5. *A set of places S is said to be synchronized if and only if $\forall p_i, p_j \in S \;/\; p_i \neq p_j$ and $\sharp \bullet p_i = \sharp \bullet p_j = 1$[1], $\bullet p_i$ is enabled if and only if $\bullet p_j$ is also enabled.*

To synthesize a set of SBPIs to enforce this constraint, a new set of places $S' = \{p_i^{wait}\}$ has to be defined where all the p_i^{wait} come from the PW-Transformation of p_i $\forall_{p_i \in S}$.

[1]Where $\sharp \bullet p_i$ means: the number of input transition of place p_i.

Proposition 6.1.3. *Let W be a set of independent Petri nets and S a synchronization set of task-places defined over W, then it can be shown that if the constraint (6.24) holds, the synchronization property is satisfied.*

$$\forall p_i^{wait}, p_j^{wait} \in S' \text{ where } p_i^{wait} \neq p_j^{wait}$$
$$cv \leq 0 \text{ where}$$
$$c = [c_1 \cdots c_j \cdots c_m], \text{ with } c_j = \begin{cases} -1 & \text{if } t_j \in {}^\bullet p_i^{wait} \\ 1 & \text{if } t_j \in p_j^{wait\bullet} \\ 0 & \text{otherwise} \end{cases} \tag{6.24}$$

Proof. A synchronization between places $S = \{p_i, p_j\}$ is equivalent to the orderings $O_1 = \{{}^\bullet{}^\bullet p_j, p_i\}$ and $O_2 = \{{}^\bullet{}^\bullet p_i, p_j\}$. Therefore the proof follows the lines of the proof presented in *Proposition 6.1.2*. □

If the tasks represented by the places $P2$ and $P5$ in Figure 6.2 have to be executed synchronously, the synchronization set $S = \{P2, P5\}$ is defined. After apply the PW-Transformation to $P2$ and $P5$ and building the set $S' = \{P2^{wait}, P5^{wait}\}$ Proposition 6.1.3 may be applied. If $\sharp S$ stands for the number of elements in the synchronization, then

$$\sharp S \cdot (\sharp S - 1) \tag{6.25}$$

is the number of orderings constraints needed to enforce the synchronization. In the proposed example $\sharp S = 2$ then, only two ordering constraints are needed. The constraints are

$$-v(T1) + v(T4') \leq 0 \text{ and}$$
$$v(T2') - v(T3) \leq 0$$

Applying equations (6.17), (6.18) and (A.11) two supervisors are synthesized as shown in Figure 6.6.

Figure 6.7 shows an extract of the missions programmed for the underwater construction scenario. Each AUV has to move at one of the extremes of a building

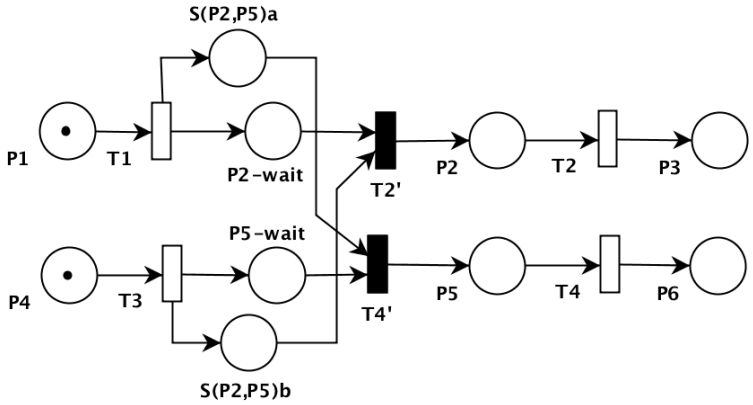


Figure 6.6: Synchronization of tasks $P2$ and $P5$.

block and once there, wait for the robotic arms to take the block. Next, AUVs move to the way-point in which the building block has to be released. Meanwhile, the two manipulators should wait until being near the block to take away. Once there, they have to grab it, and then, lift it in synchrony. When both vehicles are in the final way-point, the manipulators have to release the building block at the same time. A mission like the one shown in Figure 6.7(a) is programmed for each AUV and a mission like the one shown in Figure 6.7(b) is programmed for each robotic arm. Then, six constraints are defined:

O(AUV1_navigate_wp1,Arm1_graps),

O(AUV2_navigate_wp1,Arm2_graps),

S(Arm1_lift,Arm2_lift),

O(Arm1_lift,AUV1_navigate_wp2),

O(Arm2_lift,AUV2_navigate_wp2),

S(AUV1_keep_pose,AUV2_keep_pose,Arm1_release,Arm2_release).

Figure 6.8 shows the resulting centralized Petri net after applying the first five constraints. The last synchronization is not included because it generates 12 supervisors and, if included, the figure becomes hard to follow. Chapter 8 shows a complete example in which two AUVs and two Autonomous Surface Crafts (ASCs) are coordinated.

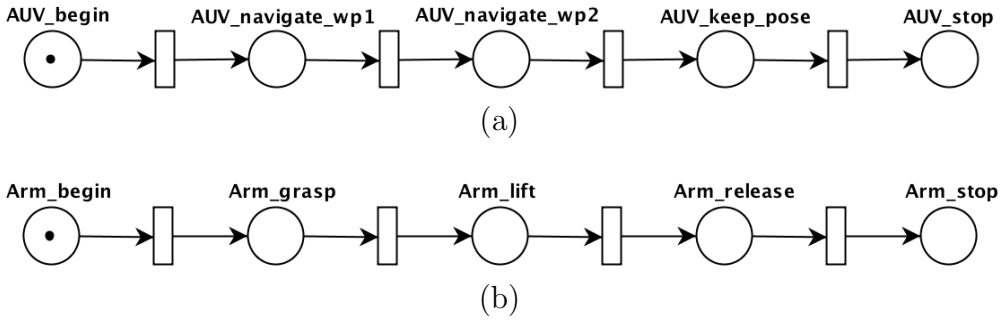


Figure 6.7: (a) AUV mission example and (b) robotic arm mission example.

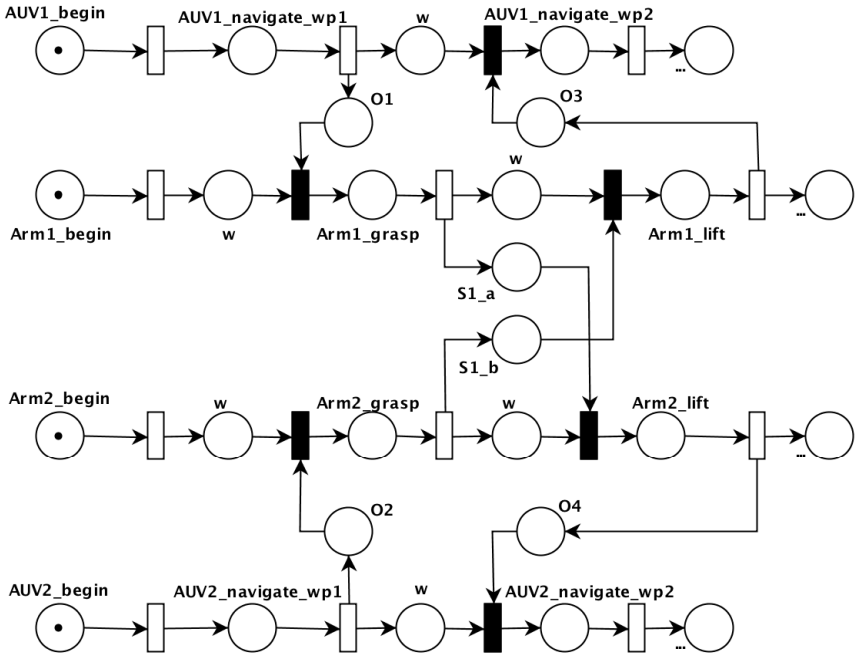


Figure 6.8: Resulting centralized Petri net after combining four independent missions with five coordination constraints.

6.2 Deadlock avoidance

After applying a set of coordination constraints among several vehicle missions it is necessary to check that the resulting centralized mission is deadlock free.

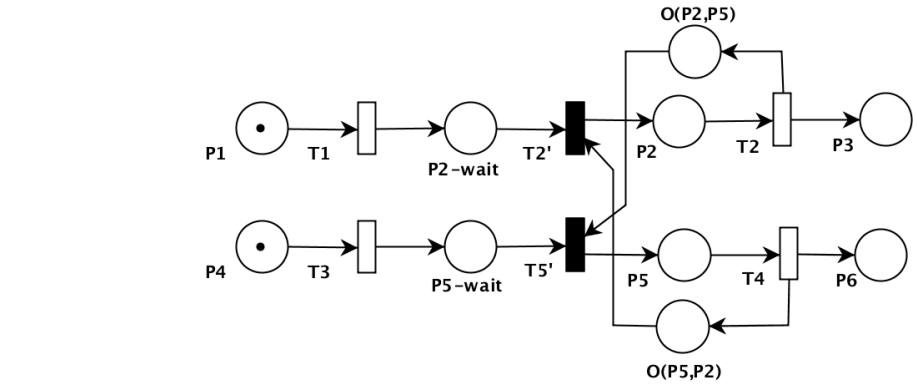
Several techniques for deadlock avoidance using Petri nets can be found in the literature [Iordache et al., 2002a; Lautenbach and Ridder, 1996]. Most of them are based on the well known necessary condition for deadlock, namely that a deadlocked ordinary Petri net contains at least one empty siphon. From this definition, to avoid a deadlock, it is necessary to ensure that none of the siphons in the net becomes empty. To achieve this, two conditions have to be accomplished:

1. All the siphons must be initially marked.

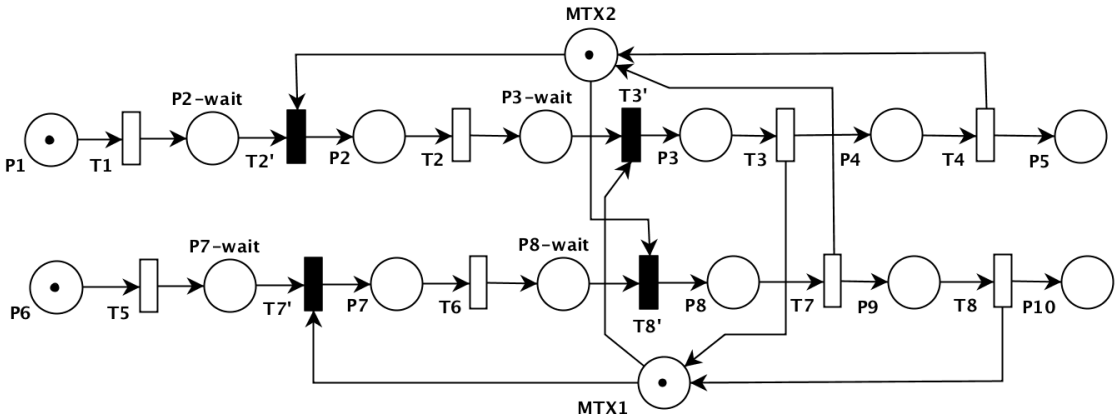
2. All the siphons must be controlled.

A siphon can be controlled by a trap or by a place invariant. If a siphon is not controlled by its own Petri net it is possible to add a constraint like $l\mu \leq b$ to control it.

Figure 6.9 shows two basic cases in which deadlocks can appear because of combined constraints. The deadlock in Figure 6.9(a) arise because place $O(P2, P5)$ has to be marked before firing $T5'$ to mark $P5$, then fire $T4$ and finally mark $O(P5, P2)$. Hence, place $O(P2, P5)$ is a predecessor of place $O(P5, P2)$ itself. But at the same time $O(P2, P5)$ can only be marked if $T2'$ fires marking $P2$ and also firing $T2$. Since place $O(P5, P2)$ must be marked in order to enable $T2'$, place $O(P5, P2)$ must also be a predecessor of place $O(P2, P5)$ itself and hence the deadlock appears. This deadlock can be detected but not avoided. Other deadlocks can be not only detected but avoided too. The deadlock shown in Figure 6.9(b) appears if $T7'$ fires later than $T2'$ but before $T3$ or if $T2'$ fires after $T7'$ but before $T7$. This deadlock can be avoided applying an extra supervisor. A procedure to check if a Petri net is deadlock free and able to add an extra supervisor if it is not, is described in the algorithm presented in Extract 6.1.



(a)



(b)

Figure 6.9: (a) A deadlock appears when two ordering constraint $O(P2, P5)$ and $O(P5, P2)$ are combined. (b) A deadlock appears when the two mutual exclusions $M([P2 \rightarrow T2 \rightarrow P3 \rightarrow T3 \rightarrow P4], P8)$ and $M(P3, [P7 \rightarrow T6 \rightarrow P8 \rightarrow T7 \rightarrow P9])$ are combined.

Extract 6.1: Deadlock avoidance procedure.

(1) Compute the invariants of the centralized net.
(2) For every vehicle mission, join the final places with the initial place through a single transition.
(3) Compute the minimal siphons and traps in the transformed Petri net resulting from step (2).
(4) For every siphon Sp that is not controlled by a place invariant or a trap, or that is not initially marked, generate a constraint $l\mu \leq 1$ using equation (6.26). If there are no uncontrolled, or initially unmarked siphons, the Petri net is deadlock free and the algorithm finalizes.
(5) If the constraint $l\mu \leq 1$ generated in step (4) produces a $D_c = [0, 0, ...0]$ applying equation (A.10), it means that it is impossible to add a supervisor to avoid the deadlock. The algorithm finalizes with a deadlocked Petri net conveniently signaled.

$$\mathbf{l} = [l_1 \cdots l_i \cdots l_{nc}], \ where \ l_i = \begin{cases} 1 & \text{if } p_i \in Sp \\ 0 & \text{otherwise} \end{cases} \tag{6.26}$$

(6) If the SBPI synthesized in step (4) is valid ($D_c \neq [0, 0, ...0]$), add the supervisor to the original Petri net, add the place invariant to the list created in step (1) and repeat from step (2).

Applying the algorithm in Extract 6.1 to the Petri net in Figure 6.9(a) the following invariants are obtained:

$$P1 + P2^{wait} + P2 + P3 = 1 \tag{6.27}$$
$$P4 + P5^{wait} + P5 + P6 = 1 \tag{6.28}$$
$$P2 + P5 + O(P2, P5) + O(P5, P2) = 0 \tag{6.29}$$

To apply the Petri net modification described in step (2), a transition (T') must be added between the end place $P3$ and the begin place $P1$ as well as between $P6$ and $P4$ as shown in Figure 6.10. Four minimal siphons are obtained in the transformed Petri net. All of them are trap or invariant controlled but

$$Sp = \{P2, P5, O(P2, P5), O(P5, P2)\} \tag{6.30}$$

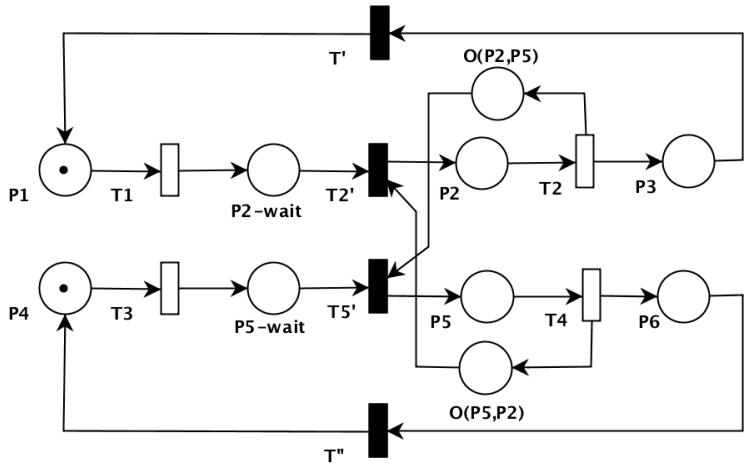


Figure 6.10: Applying the deadlock avoidance procedure to Figure 6.9(a).

is not initially marked. If a supervisor is synthesized to initially mark this siphon, the resulting supervisor happens to be not valid ($D_c = [0, 0, ..., 0]$) and the algorithm terminates at step (v).

When applying the same algorithm to the Petri net in Figure 6.9(b) four invariants and five siphons are obtained. The only uncontrolled siphon is

$$Sp = \{P4, P3, P8, P9, M([P2 \rightarrow T2 \rightarrow P3 \rightarrow T3 \rightarrow P4], P8), \\ M(P3, [P7 \rightarrow T6 \rightarrow P8 \rightarrow T7 \rightarrow P9])\} \quad (6.31)$$

A supervisor is generated in the step (4) of Extract 6.1 and applied to the Petri net as shows in Figure 6.11. Since no more siphons appear, the algorithm terminates with a deadlock free Petri net.

6.3 Decentralized supervision

A set of algorithms to check if a centralized Petri net system can be distributed among several subsystems is presented in Iordache and Antsaklis [2006b]. This centralized systems can be related to the Petri net obtained in previous sections after applying a set of coordination constraints to the vehicle missions. Moreover, if the obtained system is not directly distributable, an algorithm to add

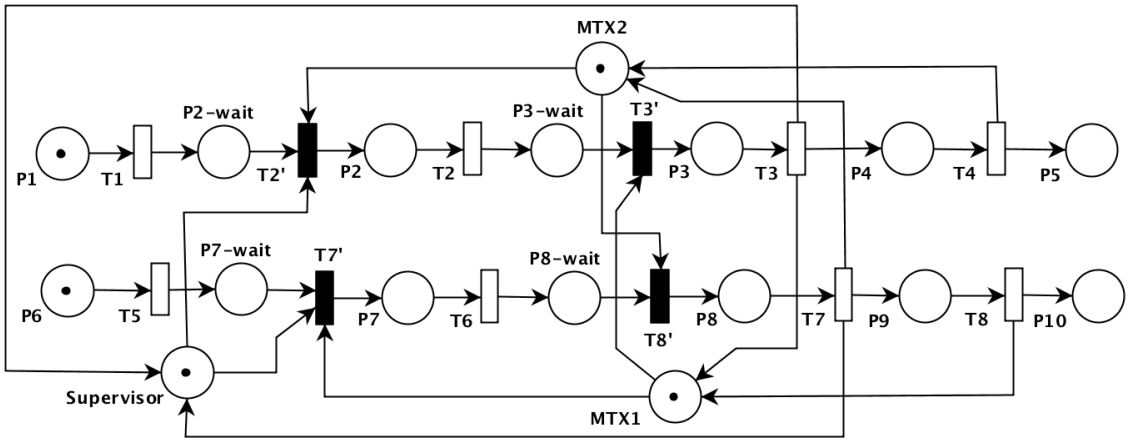


Figure 6.11: Figure 6.9(b) after applying the deadlock avoidance procedure.

minimal communication to make it distributable is introduced too. This section reproduces some of the algorithms introduced in Iordache and Antsaklis [2006b] showing how they can be used to decentralize the multiple-vehicles centralized missions previously synthesized.

6.3.1 Checking the d-admissibility of a constraint

A system is admissible if all its supervisors, the places that has been added in order to ensure the coordination constraint, control only controllable transitions and detect only observable transitions. t is a controllable transition if it is possible to add a supervisor place c so that t can be in $c\bullet$. On the other hand, t is observable if it is possible to add a supervisor place c so that t can be in $c\bullet$ or in $\bullet c$.

To check if a system that is admissible in a centralized way, *c-admissible*, is also admissible once distributed, *d-admissible*, all its constraints have to accomplish the algorithm shown in Extract 6.2. Let T_o^M and T_c^M be the observed and controlled transitions by the constraint $l\mu + c\upsilon \leq b$ being checked. ζ is the set of subsystems in which the centralized system will be split and $T_{o,i}$ and $T_{c,i}$ are the observable and controllable transitions for each subsystem. In our framework, the subsystems ζ as well as the $T_{o,i}$ and $T_{c,i}$ sets are defined for the initial uncoordinated vehicle missions. There are as many subsystems as independent vehicle missions and it is assumed that each vehicle is only capable of observing

and controlling their own transitions.

Extract 6.2: Check the d-admissibility of a constraint.

(1) Find T_o^M and T_c^M.
(2) Let ζ be the set of indices i satisfying $T_{o,i} \supseteq T_o^M$.
(3) If $\zeta = \emptyset$, declare the constraint not d-admissible and exit.
(4) Define $T_c = \cup_{i \in \zeta}\ T_{c,i}$.
(5) If T_c satisfy $T_c \supseteq T_c^M$ then constraint d-admissible else constraint not d-admissible.

In general, it can be difficult to compute T_o^M and T_c^M. Alternatively, estimates of $T_c^e \supseteq T_c^M$ and $T_o^e \supseteq T_o^M$ can be used. However, in this case the algorithm only checks a sufficient condition for d-admissibility. T_c^e and T_o^e can be calculated using the control place C generated for the SBPI synthesized by the constraint to enforce the desired behavior in the centralized system as $T_c^e = C\bullet$ and $T_o^e = \bullet C \cup C\bullet$

6.3.2 Design minimizing communication

If the algorithm in Extract 6.2 returns that a constraint is not d-admissible it is possible to introduce communication in order to make the constraint d-admissible. To characterize communication three binary variables are introduced: $\alpha_{i,j} = 1$ iff the transition t_j is communicated to subsystem s_i, $\epsilon_{i,j} = 1$ iff the transition t_j is remotely controlled by subsystem s_i and $\delta_{i,k} = 1$ iff the subsystem s_i is involved with the constraint k.

Extract 6.3 applies an Integer Linear Program (ILP) to solve (6.35) in order to minimize the communication cost of the distribution.

Extract 6.3: Design minimizing communication.

(1) Find the value of α, ϵ and β that minimize (6.35) subjected to the constraints defined by (6.32), (6.33) and (6.34).

$$\alpha_{i,j} \geq \delta_{i,j}\ \forall j \in \{f : t_j \in T_o^k \setminus T_{o,i}\} \tag{6.32}$$

$$\forall k = 1 \cdots n_c : \sum_{i=1}^{n} \delta_{i,k} \geq 1 \tag{6.33}$$

$$\forall k = 1 \cdots n_c,\ \forall x = 1 \cdots n,\ \forall j \in \{y : t_y \in T_c^k\} : \\ \delta_{x,k} \leq \epsilon_{x,j} + \sum_{i \in I_j} \delta_{i,k} \tag{6.34}$$

$$\sum_{i,j} \alpha_{i,j} c_{i,j} + \sum_{i,j} \epsilon_{i,j} f_{i,j} + \sum_{i,k} \delta_{i,k} h_{i,k} \tag{6.35}$$

Where n_c is the number of constraints, n the number of subsystems, $c_{i,j}$ is the cost of communicate the t_j firing to subsystem s_i, $f_{i,j}$ is the cost of control the t_j firing from subsystem s_i and $h_{i,k}$ is the cost that subsystem s_i intervenes in constraint k.

When talking about underwater vehicles, minimize the communication between them is critical. Underwater vehicles use to communicate through acoustic modems with very low bandwidth. Then, it is important to find techniques to minimize the amount of information to be transmitted.

6.3.3 Supervisor design for a d-admissible constraint

If the algorithm presented in Extract 6.2 shows that all the constraints are d-admissible or using the algorithm shown in Extract 6.3 a communication policy is found which makes all the constraints d-admissible, it is possible to design a decentralized supervisor applying Extract 6.4 to each constraint.

Extract 6.4: Supervisor design for a d-admissible constraint.

(1) Let μ_0 be the initial marking of N, C the control place of the centralized SBPI enforcing $l\mu + hq + cv \leq b$ and ζ the set of subsystems.
(2) $\forall_{i \in \zeta}$, let $x_i \in \mathbb{N}$ be a state variable of s_i.
(3) Define S_i, for $i \in \zeta$, by the following rules:
  3.1 - Initialise $x_i = \mu_{s0}$
  3.2 - If $t \in T_{c,i}$, $t \in C\bullet$ and $x_i < W_s(C,t)$, then S_i disables t.
  3.3 - If t fires, $t \in T_{o,i}$ and $t \in \bullet C$, then $x_i = x_i + W_s(t,C)$.
  3.4 - If t fires, $t \in T_{o,i}$ and $t \in C\bullet$, then $x_i = x_i - W_s(C,t)$.

In our framework, Extract 6.2 will be always false if the constraint to check involves two or more vehicles. However, if only the presented constraints are used, it is always possible to find a communication policy with Extract 6.3 to

decentralize them.

6.4 Multiple vehicle coordination implementation

To integrate all these algorithms into our current framework, the MCL has been extended. First, instead of being able to define only one mission per file, several missions can be defined. Second, each one of these missions have an identifier named *mission_id*. Third, each task can have a label, *label_id*, associated to it. Finally, a new section has been added in the MCL to describe the constraints involved in the mission using the syntax described in Extract 6.5.

Extract 6.5: MCL constraints definition.

```
constraints {
    mutex {
        mission_id:label_id (, mission_id:label_id)+
    }=β
    order {
        mission_id:label_id, mission_id:label_id
    }
    sync {
        mission_id:label_id (, mission_id:label_id)+
    }
}
```

Algorithms presented in Extract 6.2, Extract 6.3 and Extract 6.4 have been implemented within the MCL-C in order to obtain one PNML file for each vehicle once compiled the MCL code. The *lp_solve* library [Berkelaar et al., 2010] is used to solve, in compilation time, the ILP problem that appears in Extract 6.3.

Chapter 7

Planning

Predefined plans, as the ones presented in previous chapters, are the state of the art for today Autonomous Underwater Vehicle (AUV) missions. However, when dealing with the uncertain and unknown underwater environment with processes and other agents changing the world in unpredictable ways, and with notoriously imprecise and noisy sensors, plans can fail by several reasons [Turner, 2005]. Quoting Dwight D. Eisenhower: *Plans are worthless, but planning is everything. There is a very great distinction because when you are planning for an emergency you must start with this one thing: the very definition of "emergency" is that it is unexpected, therefore it is not going to happen the way you are planning.* The difficulty to control the time in which events happens, how to deal with resources as energy, how to respond to sensor malfunctions or the lack of on-board situational awareness may cause off-line plans to fail at execution time as assumptions upon which they were based are violated. Moreover, the difficulty to add new sub-goals during a mission diminishes the possibilities of these off-line planned systems [Rajan et al., 2007]. Consequently, an on-board planner with the ability to modify or re-plan the original plan should be included in an AUV. However, on the other hand, scientists want to ensure that the data that they need is collected where and when they specify and not when a planner decides to do it. Similarly, militaries have often been opposed to on-board planners, since it is critical for most of their missions to ensure a predictable robot behavior. Then, a compromise between off-line plans and automated on-board planning is desirable.

As presented in the survey, see Chapter 2, not many successful approaches

using deliberative modules on-board an AUV are found in the literature [Evans et al., 2006; Patrón et al., 2008; Rajan et al., 2007; Turner, 2005].

In general, to develop a system able to react to any unforeseen event is still an open question. Nowadays, algorithms are only able to react to those situations for which they have been programmed to deal with. However, the way in which the actions to take are specified can drastically change based on how the problem is defined. For instance, if when defining a mission a conditional structure is used each time that the vehicle can choose several alternatives, the size and complexity of this mission plan will grow exponentially to the number of alternatives that the vehicle can face during the mission execution. However, if it is possible to describe each vehicle primitive in such a way that an on-board planner can understand when and why this primitive should be executed, the planner itself can be able to choose the most suitable action each time simplifying, a priory, the mission description. Therefore, we propose to add some deliberative capabilities to the previously presented framework in a hierarchical way. Although, automated planning techniques are out of the scope of this thesis, the purpose of this chapter is to describe the interface of some well-known automatic planning techniques with the existing Mission Control System (MCS) from a practical point of view. Thus, simple planning techniques are used to illustrate how to interface a deliberative layer over the presented MCS.

7.1 Automated planning

Automated planning concerns the realization of strategies or action sequences. It can be defined as the reasoning side of acting that tries to choose and organize actions, to fulfill a goal, by anticipating their expected outcomes [Ghallab et al., 2004]. A typical planner takes three inputs: a description of the initial state of the world, a description of the desired goal, and a set of possible actions, all encoded in a formal language. The planner produces a sequence of actions that leads from the initial state to a state meeting the goal.

Planning can be used in diverse domain-specific areas like path-planning [Ferguson et al., 2005] or manipulation-planning [Amato and Wu, 1996]. In some of these areas the use of domain-specific approaches may be beneficial. Generally speaking, domain-specific planners use specific representations and techniques adapted to each problem while domain-independent planning use generic rep-

resentations and techniques. However, when dealing with autonomous vehicles, domain-specific approaches may not be sufficiently satisfactory because its deliberative capabilities will be limited to areas for which domain-specific planners are available. Then, domain-independent approaches, that claim to not use domain knowledge, should be used.

Planning is concerned with choosing and organizing planning actions to change the state of a system. Then, a conceptual model for this system have to be defined. Formally, this state-transition dynamic system is defined by the triple

$$\Sigma = (S, A, \gamma), \tag{7.1}$$

where:

- $S = \{s_1, s_2, \cdots\}$ is a finite set of states;
- $A = \{a_1, a_2, \cdots\}$ is a finite set of planning actions; and
- $\gamma : S \times A \to 2^S$ is a state transition function.

To simplify the planning problem, it is useful to make restrictive assumptions in order to work out a well defined model. The following assumptions define the widely used *restricted model* [Ghallab et al., 2004].

Definition 7.1.1. *The restricted model is a state-transition system like the one presented in (7.1) in which the following assumption hold:*

- *The system has a finite and fully observable set of states with complete knowledge about it.*
- *The system is deterministic and static. This means that the states only change when an operation is applied to a state and its application brings the system to a single other state.*
- *The planner handles only restricted goals. Extended goals such as states to be avoided and constraints on state trajectories are not handled.*
- *The solution plan is a linearly ordered finite sequence of operations with no duration.*

7.2 Classical planning

Despite planning-graph [Blum and Furst, 1997] and propositional satisfability [Davis and Putnam, 1960] techniques have been proved very effective in several domains, the classical planning approach has been chosen in this dissertation for its simplicity.

A classical planning problem for a state-transition system (Σ) following the restricted model introduced in Definition 7.1.1 is a triple $P = (\Sigma, s_0, g)$, where s_0 is an initial state and g corresponds to a set of goal states. Then, a solution to P is a sequence of planning actions $(a_1, a_2, \cdots, a_k)$ corresponding to a sequence of state-transitions $(s_0, s_1, \cdots, s_k)$ such that $s_1 = \gamma(s_0, a_1), \cdots, s_k = \gamma(s_{k-1}, a_k)$ and s_k is a goal state ($s_k \in g$).

In a classical planing approach, states are represented by sets of logical atoms that can be true or false and planning actions are represented by the instance of *planning operators* that change the truth value of these atoms. A state is a set of *ground atoms* of L where L is a first-order language in which every term is either a *variable symbol* or a *constant symbol*. If L has no function symbols, the set S of all possible states is guaranteed to be finite.

Definition 7.2.1. *Let L be a first-order language that has finitely many predicate symbols and constant symbols [Ghallab et al., 2004]. A* classical planning domain *in L is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:*

- $S \subseteq 2^{\textit{all ground atoms of } L}$
- $A =$ all ground instances of operators in O, where O is a set of planning operators. Each planning action $a \in A$ is a triple of subsets of L, which can be written as $a = (precond(a), effects^-(a), effects^+(a))$. The set $precond(a)$ is called the preconditions of a. The set of $effects^+(a)$ and $effects^-(a)$ are called the effects of a. These two sets of effects must be disjoint ($effects^+(a) \cap effects^-(a) = \emptyset$)
- $\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$, $a \in A$ is applicable to $s \in S$ iff $precond(a) \subseteq s$, otherwise $\gamma(s, a)$ is undefined.
- S has the property that if $s \in S$, then for every planning action a that is applicable to s, the set $(s - effects^-(a)) \cup effects^+(a) \in S$. In other words, S is closed under γ.

Definition 7.2.2. *A classical planning problem is a triple* $P = (\Sigma, s_0, g)$*, where:*

- s_0, the initial state, is any state in set S;
- g, the goal, is any set of ground literals; and
- a goal g correspond to a set of goal sates S_g where $S_g = \{s \in S | s\ satisfies\ g\}$

Thus, the *statement* of a planning problem $P = (\Sigma, s_0, g)$ is $P = (O, s_0, g)$.

Once defined the classical planning principles, how states, goals and operators are presented in our framework is introduced.

7.2.1 States

States have been introduced in Definition 7.2.1 as a set of ground atoms of L, where these ground atoms are the combination of variables and constant symbols. In our implementation the ground atoms of L are named *facts*. Thus, each state contain a set of facts representing the information that is true in the world model for this specific state. As the closed-world assumption [Reiter, 1982] is used, the facts that does not appear in the state are false. The variable symbols that compose these facts are called *entities* and each *entity* has a *type* assigned. Then, to describe a fact, one or two entities with a constant symbol representing a quality or a relation between the entities is used. For instance, if we want to express that in the initial state our vehicle is in a specific location we can use the fact

vehicle::ictineu ***In*** *location::dock*

where *ictineu* is an entity of type *vehicle*, *dock* is an entity of type *location* and ***In*** is a constant symbol that relates both entities. Entities can contain internal attributes. For instance, a *location* type entity includes three attributes: x, y and z. This extra information is used when an operator is instantiated. A file with all the entities in the knowledge database as well as their attributes must be provided by the user.

7.2.2 Initial state s_0 and goal g

Both the initial state s_0 and the goal g are two sets of facts. The former can be given by the user or computed by a component able to extract information from the world. It have to include all the facts, related to all the entities in the domain, that are true when the planner starts to compute a new plan. The latter, contains the set of facts that have to be true to consider that the mission has been fulfilled. This means that if $g \subseteq s_k$, s_k is a final state and no more operator instances have to be computed.

7.2.3 Planning operators

The transition function γ is specified generically through a set of planning operators that are instantiated into planning actions. Do not confuse planning actions with the actions used in previous chapters to enable/disable vehicle primitives.

Definition 7.2.3. *In classical planning, a planning operator is a triple*

$$o = (name(o), precond(o), effects(o)), \tag{7.2}$$

where

- $name(o)$, the name of the operator, is an expression of the form $n(x_1, \cdots, x_k)$, where n is a symbol called *operator name* and $x_1, \cdots, x_k$ are the variable symbols that appear anywhere in o, the entities;
- $precond(o)$ and $effects(o)$, the preconditions and effects of o, respectively, are generalizations of the preconditions and effects in Definition 7.2.1.

It is worth noting that n is a unique operator identifier, the variable symbols that appear anywhere in o are entities and each one of these entities has a particular type. For instance, the signature of an operator that moves a vehicle from one location to another is an expression like

move(*vehicle*, *location*, *location*),

where **move** is the operator name n and the variable symbols $x_1, \cdots, x_k$ are defined by the entity types *vehicle*, *location* and *location*. This general operator can be instanced into a particular planning action like

move(*ictineu*, *deploy*, *dock*),

where *ictineu* is a *vehicle* type entity and *deploy* and *dock* are *location* type entities. $precond(o)$ indicates the facts that must be present in the world model in order to apply the operator. Thus, if a is an instance of an operator $o \in O$, $\gamma(s, a)$ exists iff $precond(a) \subseteq s$ where $s \in S$. Finally, $effects(a)$ are the changes produced in the world if the operator is applied. $effects(a)$ are divided in $effects^{+}(a)$ and $effects^{-}(a)$, where

- $effects^{+}(a)$ are the facts that are supposed to be produced into the current state after applying the operator instance a; and
- $effects^{-}(a)$ are the facts that are supposed to be removed from the current state after applying the operator instance a.

Despite the facts that have to be true in the world in order to apply the operator, the preconditions, a set of boolean *expressions* relating the operator entities among them or with literal values, can be added to the operator o. Only when all the *expressions* are true the operator may be applied.

Extract 7.1 presents an example of a planning operator, named **move**, that moves a vehicle (v) from an initial location ($l1$) to a final location ($l2$) only if the vehicle v is in location $l1$, as stated in the precondition, and $l1 \neq l2$, as stated in the expressions.

Extract 7.1: Planning operator **move**.

move(vehicle *v*, location *l1*, location *l2*)

precond
- v in $l1$;

add
- v in $l2$;

del
- v in $l1$;

expression
- $l1 \neq l2$;

If no cost function is specified in the operator, the planner looks for the plan with the shortest sequence of planning actions, otherwise, the planner minimize the cost function.

To execute a planning action on a real robot it is necessary to associate it with a vehicle action. As exposed in previous chapters, the basic robot actions are the vehicle primitives. However, instead of relating vehicle primitives with planning operators, off-line missions programmed using the Mission Control Language (MCL) has been associated to planning operators. This pre-programmed missions, named MCL *mission operators*, are in charge to solve a particular phase of a more complex mission. Then, the planner will not sequence vehicle primitives but small pieces of MCL code containing calls to vehicle primitives. Several advantages arise from this solution:

- First of all, the number of planning operators and the length of the sequence of planning actions to fulfil a mission will be reduced. As the complexity of a classical planning problem is related to the number of states in S and also to the number of operator instances (A) applicable to each state, then the planning problem is reduced and, therefore, plans can be obtained more quickly.

- The vehicle's behavior is more predictable than executing an arbitrary combination of primitives selected by the on-board planner. The user is who chooses how to combine a set of vehicle primitives to perform a specific task nor the on-board planner. The planner only decides in which mission phase the vehicle is and which MCL mission operator is more suitable to be executed in this particular phase.

- Finally, although basic planners produce only a sequence of actions to be executed, parallelization, conditional and iterative execution of vehicle primitives can be done within the MCL mission operators by means of Petri Net Building Blocks (PNBBs) control structure.

To link the planning operators with the MCL mission operators, the extra keyword ***mcl*** has been added in the planning operator definition as shown in

Extract 7.2.

Extract 7.2: Complete planning operator **move** definition.

move(vehicle *v*, location *l1*, location *l2*)
- **precond**
  - *v* in *l1* ;
- **add**
  - *v* in *l2* ;
- **del**
  - *v* in *l1* ;
- **expression**
  - *l1* $\neq$ *l2* ;
- **mcl**
  - MclMove(*l2.x*, *l2.y*, *l2.z*) ;

Extract 7.3 shows the MCL mission operator linked by the **move** planning operator in Extract 7.2. This MCL operator looks very simple because if some error occurs when executing it, the error will be cached by the planner and a new plan will be computed.

Extract 7.3: *MclMove* MCL mission operator.

mission MclMove(*x*, *y*, *z*)
- **monitor**
  - **parallel**
    - AvoidObstacles( )
  - **or**
    - Goto(*x*, *y*, *z*)
- **condition** Alarm() **do**
  - Stop()

To initialize the parameters of the *MissionGoto* MCL mission operator, the predefined attributes of *l2* entity, ($l2.x$, $l2.y$ and $l2.z$) are used.

7.2.4 Plans

The planner generates a plan to be executed in real-time on-board an autonomous vehicle. This plan must contain an ordered list of planning actions. Furthermore, the plan contains the facts that should be true in the world each time that a

planning action is under execution. This information will be useful when running the plan on-line to know whether it continues being valid or not.

Definition 7.2.4. *A plan is defined as an ordered list of executable planning actions each one described by the triple*

$$plan = \{mcl_{op}, F_i, F_e\}, \tag{7.3}$$

where

- mcl_{op} is an instance of an MCL mission operator;
- F_i is the set of facts that should be true when the mcl_{op} is initially executed; and
- F_e is the set of facts that should be true after execute the mcl_{op}.

7.3 State-Space planner

The simplest algorithms used in classical planning are the state-space search algorithms. The search space of this algorithms is a subset of the state space: each node in the search space corresponds to a state of the world, each arc corresponds to a state transition, and a plan corresponds to a path in the search space from the initial node (initial state) to a final node (a state in which the goals have been achieved) [Ghallab et al., 2004]. Opposed to state-space planners there are plan-space planners in which nodes are partially specified plans and arcs are plan refinement operations intended to further complete a partial plan. Intuitively, a refinement operation avoids adding to the partial plan any constraint that is not strictly needed for addressing the refinement purpose. This is called the *least commitment principle* [Weld, 1994]. Planning starts from an initial node corresponding to an empty plan and the search aims at a final node containing a solution plan that correctly achieves the required goals. Planners planning in the plan-space are commonly known as Partial Order Planning (POP) algorithms. Despite several successful approaches have demonstrated the validity of POP when dealing with autonomous vehicles [Chien et al., 1998; Patrón et al., 2008], for the sake of simplicity, state-space planners will be discussed in this dissertation. The aim of this chapter is just to illustrate how a domain-independent

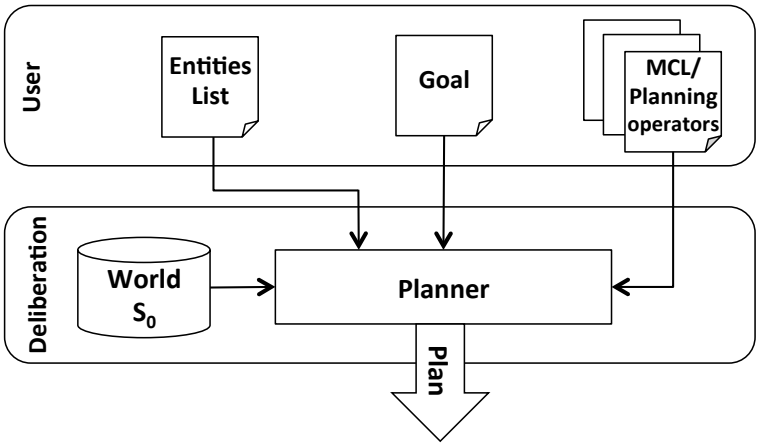


Figure 7.1: Planner components and its relations.

planner algorithm can be connected with the previously presented methodology to improve the performance of the whole MCS adding some deliberative capabilities. Then, as shown in Figure 7.1, from the MCS point of view, the planner algorithm will be a black-box that given some inputs is able to generate a plan as an output. Thus, although a simple planner algorithm will be presented in this chapter, it can be replaced by any other planner, like a POP or a graph-plan to improve its performance.

7.3.1 Search algorithms

Search techniques are general problem-solving methods. To formulate a search problem, a set of states, a set of operators, an initial state and a goal criterion have to be specified. Then, it is possible to use search techniques to solve the problem [Pearl and Korf, 1987]. If search algorithms go from the initial state s_0 to a final state s_k in which $g \subseteq s_k$ then, the algorithm performs a *forward-search*. Otherwise, if the algorithm begins at the goal g and inverses of the planning operator instances are applied until satisfying the initial state a *backward search* is used. In small scale search problems, simple search techniques are sufficient to do a systematic search. However, due to its complexity, heuristics can be introduced to guide the search process. Heuristic search makes use of the fact that most problem spaces provide some information that distinguishes among states in terms of their likelihood of leading to a goal. This information is called a heuristic evaluation function [Pearl and Korf, 1987]. In other words, the goal

of an heuristic search is to reduce the number of nodes searched in seeking a goal [Kopec et al., 2004]. Then, it is important to distinguish between the search algorithms that use heuristics and those who do not use it.

7.3.1.1 Non heuristics search algorithms

The state-space for a planning problem is defined as a tree in which the root is s_0. For each operator instance that is applied to s_0 a new state s_i appears. The whole process is then repeated for each s_i until an state that satisfies g is achieved. However, if it is possible to achieve the same state from different previous states, the tree structure can be converted into a more efficient graph structure as shown in Figure 7.2. There are two main algorithms to visit all the nodes in a graph:

- A Depth-First Search (DFS) is a technique for traversing a tree, tree structure, or graph. DFS visits the child nodes before visiting the sibling nodes, see Figure 7.3(a), that is, it traverses the depth of the tree before the breadth.

- A Breadth-First Search (BFS) is another technique for traversinga tree, tree structure, or graph. BFS visits the sibling nodes before visiting the child nodes, see Figure 7.3(b). The primary advantage of BFS is that if a nondeterministic procedure p is complete, then the breadth-first version of p will also be complete[1]. However, in most cases the BFS procedure will have a huge space requirement. For example, suppose that every node of the search space has b children, the terminal nodes are at depth d, the time needed to visit each node is $\Theta(1)$, and the space needed to store each node is also $\Theta(1)$. Then the running time for a breadth-first search will be $\Theta(b^d)$, and because a BFS must keep a record of each node visited, then the memory requirement will also be $\Theta(b^d)$.

Despite the huge space requirement of BFS, it will return the shortest plan that bring us to the solution. A standard implementation of the BFS algorithm

[1] A deterministic procedure is complete if, whenever it is invoked on a solvable problem P, it is guaranteed to return a value v not equal to failure. A nondeterministic procedure is complete if, whenever it is invoked on a solvable problem P, at least one of its execution traces will return a value v not equal to failure whenever P is solvable.

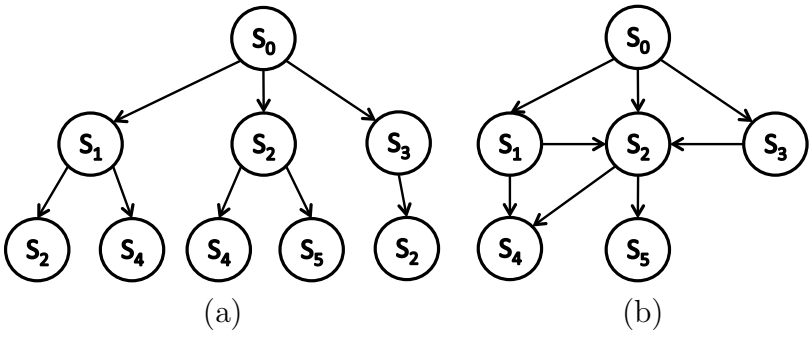


Figure 7.2: (a) Tree structure and (b) graph structure.

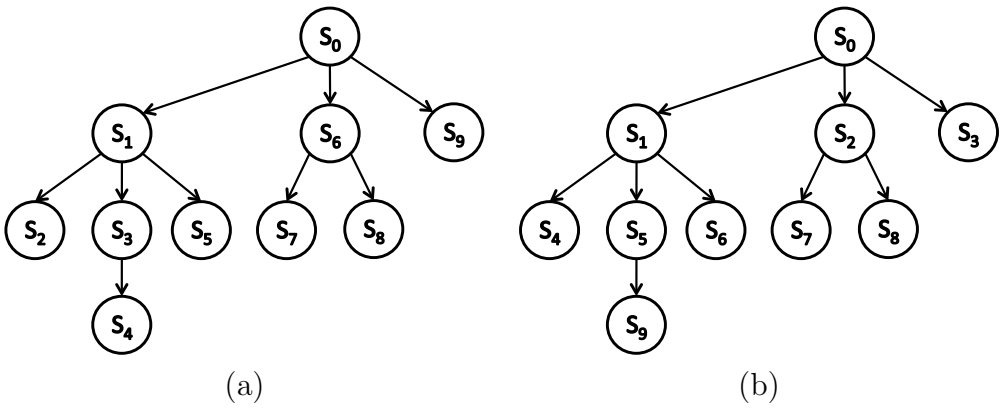


Figure 7.3: (a) Order in which the nodes are expanded using a DFS algorithm or (b) a BFS algorithm.

has been implemented following the pseudo-code presented in Extract 7.4.

Extract 7.4: BFS pseudo-code.

```
breadthFirstSearch( O, Entities, s_0, g )
    vector<action> actions = buildActions( O, Entities ) ;
    vector<state> visitedStates ;
    visitedStates.add( s_0 ) ;
    struct node
        node* n ;
        state s ;
        action a ;
    vector<node> plan ;
    plan.add( node( null, s_0, null ) ) ;
    for node p ∈ plan do
        for action a ∈ actions do
            if precond(a) ⊆ p.s then
                state newState = p.s - effects⁻(a) + effects⁺(a) ;
                if newState ∉ visitedStates then
                    visitedStates.add( newState ) ;
                    plan.add( node( *p, newState, a ) ) ;
                    if g ⊆ newState then
                        return plan ;
```

The planner algorithm based on the BFS starts building the actions by combining the planning operators, O, with all the available entities, *Entities*. To avoid visiting several times the same states, a pool of visited states, *visitedStates*, is saved. Each node in the plan is composed by the state itself, a pointer to the previous node and the action that transforms the state in the previous node to the current one. Starting from the node that includes the initial state, the algorithm checks which actions can be applied, $precond(a) \subseteq s_i$. If an action can be applied and the resulting state is not in the pool of visited states, then a new node is generated. This node contains the new state, a pointer to the current node and the action that transforms the current node with the new one. This process is repeated for all new nodes until one of them satisfies g. Then, a plan is found. To obtain the sequence of nodes to be executed the plan must be interpreted. The

pseudo-code in Extract 7.5 performs this function.

Extract 7.5: Extract list of actions from a plan graph.

```
extractRealPlan( plan )
    node n_0 = plan.first() ;
    node planNode = plan.last() ;
    vector< node > realPlan ;
    while planNode ≠ n_0 do
        realPlan.add( planNode ) ;
        planNode = planNode.n ;
    return invertVector( realPlan ) ;
```

The state in the last node of the plan satisfy g. Then, from this node, it is possible to follow the pointer to the precedent nodes until reach the initial state. This list of nodes is the real plan but in inverse order. Therefore, it must be inverted before return it.

The planning algorithm presented in Extract 7.4 do not use any cost function. However, it is simple to assign a positive weight value to each operator to be used as a cost function. Thus, the planner's goal is to obtain the sequence of planning actions leading the world model from its initial state s_0 to a final state s_k in which $g \subseteq s_k$, minimizing the total cost of all the actions being used. If all the planning operators have the same cost associated, then the algorithm in Extract 7.4 is enough to find the best solution. However, if different positive cost values are associated to each operator, some modifications have to be done. First of all, each node have to save a value indicating the cost to reach this state. Then, each time that a previously visited state is reached from a different combination of action/source node, the cost value and the preceding node/action have to be updated if the new cost is lower. Moreover, once a state that successfully accomplish g is reached, if there are nodes waiting to be evaluated with a lower cost than the current node, these nodes have to be evaluated. When there are no remaining nodes to be evaluated or these nodes have a cost bigger or equal than the solution node, the algorithm finalizes.

7.3.1.2 Heuristics search algorithms

If heuristics are used to guide the search, several well known algorithms can be used: best-first search [Pearl, 1984], A* [Hart et al., 1968] or B* [Berliner, 1979]

are just some of them. However, to use heuristics, it is necessary to assess how close to the goal, g, may bring each action. A very intuitive idea to obtain a general heuristics for a domain-independent classical planning algorithm is the *relaxation* principle [Ghallab et al., 2004]. Given an action a and a state s, the transfers function γ is defined as $\gamma(s,a) = (s - effects^-(a)) \cup effects^+(a)$. However, if the relaxation principle is applied, the transfer function γ is defined as $\gamma(s,a) = s \cup effects^+(a)$.

As the facts produced by $effects^-(a)$ are neglected in the resulting state, the simplified $\gamma(s,a)$ involves only a monotonic increase in the number of facts from s to $\gamma(s,a)$. Hence, compute the cost in which g is achieved will be easier applying this relaxation principle. The following heuristic functions extracted from Ghallab et al. [2004] are based on this relaxation idea.

Definition 7.3.1. *Let $s \in S$ be a state, f a fact, and g a set of facts. The minimum cost from s to f, denoted by $\triangle^*(s,f)$, is the sum of the cost associated to the minimum number of planning actions required to reach from s a state containing f. The minimum cost from s to g, $\triangle^*(s,g)$, is the sum of the cost of the minimum number of planning actions required to reach from s a state containing all facts in g.*

Let $\triangle(s,f)$ be an estimation of $\triangle^*(s,f)$ and $\triangle(s,g)$ be an estimation of $\triangle^*(s,g)$. $\triangle$ is given by the following equations.

$$\begin{aligned} &\triangle(s,f) = 0 && if\ f \in s \\ &\triangle(s,g) = 0 && if\ g \subseteq s \\ &\triangle(s,f) = \infty && if\ \forall a \in A, f \notin effects^+(a) \end{aligned} \tag{7.4}$$

otherwise:

$$\begin{aligned} &\triangle(s,f) = min\{cost(a) + \triangle(s, precond(a)) | f \in effects^+(a)\} \\ &\triangle(s,g) = \Sigma_{f \in g} \triangle(s,f) \end{aligned} \tag{7.5}$$

From (7.4) and (7.5) an heuristic function $h(s)$ may be defined to give an estimation of the cost from a node s to a node that satisfies the goal g of a

planning problem as

$$h(s) = \triangle(s, g) \tag{7.6}$$

Once the heuristic function $h(s)$ is defined, a standard A* algorithm can be used to perform a guided search among all possible states to find a solution for a planning problem.

7.4 Knowledge database

The restricted model introduced in Definition 7.1.1 says that the system used to plan must be static and deterministic. In general, classic planning techniques have been applied in such systems. However, when the system on which we pretend to plan is the real world, this assumption is commonly wrong. In the real world, the facts that describe a state do not only change when an action is applied, but may also change dynamically. Moreover, the application of an action does not always brings the system to a single other state, real world is nondeterministic. Therefore, it is necessary to constantly check the state of the world to detect changes on it and see whether these changes conform to the ones expected in the plan or not. If changes are different from those expected, it will be necessary to generate a new plan. To keep a simplified representation of the real world, a component named *world modeler* has been developed.

Traditionally, several control architectures for AUVs include a knowledge database to keep track of certain variables in the world [Herman et al., 1988]. In the AUV domain, these knowledge database use to collected data to build terrain elevation maps including data about the soil, vegetation, ravines, landmarks, obstacles or transponders. These data sets, use to be used by path planners to work out a safety trajectory or by on-board expert systems to decide the next region of interest to be surveyed.

In the proposed architecture, a component named world modeler is in charge to keep this knowledge database updated. To not tie the world modeler to a particular application domain, this component uses a *context provider* to be as general as possible. The context provider uses a set of scripts, defined by the user, to build the facts that compose the current world state: the knowledge database. These scripts take the perceptions received from the reactive layer to

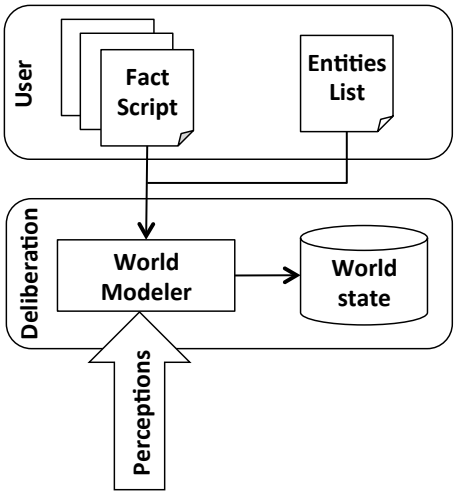


Figure 7.4: World modeler schema.

add or delete facts in the current world state as shown in Figure 7.4.

7.4.1 World modeling scripts

To describe the current world state, only the facts used by the planning operators in their preconditions and effects have to be modeled. Thus, the user have to provide scripts capable to transform the perceptions received into a set of facts. These scripts are named *fact provider* scripts.

Each script is defined by the triple

$$fp = \{id, conditions, effects\}, \tag{7.7}$$

where:

- *id* is an identifier;
- *conditions* are a set of boolean expressions involving perceptions that have to be true in order to apply the script; and
- $effects$ are the changes to be produced in the knowledge database if the script is applied. Like for the operators, *effects* are divided between $effects^+$ (add) and $effects^-$ (del).

Each script may contain one or more *condition* statements relating perceptions among them or among literals. The syntax of a *condition* is described as

<condition value1=(**perception_id**|**num**) type1=("literal"|"perception")
operator=("="|"!="|"<"|">"|"<="|">=") value2=(**perception_id**|**num**)
type2=("literal"|"perception")>

Simple boolean expressions can be defined using this syntax. When all the *conditions* inside one fact provider script are true, the $effects$ are applied. Several $effects^+$ and $effects^-$ can be included in each script following this syntax:

<del value1=("forall"|**entity_id**) type1=**entityType_id** prop=**id**
(value2=("forall"|**entity_id**) type2=**entityType_id**)?>
<add value1=("forall"|**entity_id**) type1=**entityType_id** prop=**id**
(value2=("forall"|**entity_id**) type2=**entityType_id**)?>

If the term $forall$ is used instead an **entity_id**, then a fact is generated for each entity whose type coincides with the **entityType_id**. Extract 7.6 and Extract 7.7 show two of these fact provider scripts.

Extract 7.6: low_battery fact provider script.

```
<script id="low_battery">
    <condition value1="auv.battery" type1="perception"
        operator="<" value2="30" type2="literal"/>
    <del value1="ictineu" type1="vehicle" prop="batteryOk"/>
    <add value1="ictineu" type1="vehicle" prop="batteryLow"/>
</script>
```

In Extract 7.6, when the *auv.battery* perception is sent through the Architecture Abstraction Component (AAC) to the world modeler, the fact provider script *low_battery* is executed. If the value of the perception *auv.battery* is smaller than 30%, the fact *ictineu BatteryOk* is removed from the world knowledge database while the fact *ictineu BatteryLow* is added to it.

Extract 7.7: recovery_location fact provider script.

```
<script id="recovery\_location">
    <condition value1="auv.x" type1="perception"
        operator=">" value2="-2" type2="literal"/>
    <condition value1="auv.x" type1="perception"
        operator="<" value2="2" type2="literal"/>
    <condition value1="auv.y" type1="perception"
        operator=">" value2="-2" type2="literal"/>
    <condition value1="auv.y" type1="perception"
        operator="<" value2="2" type2="literal"/>
    <condition value1="auv.z" type1="perception"
        operator="<" value2="0.2" type2="literal"/>
    <condition value1="auv.z" type1="perception"
        operator=">" value2="0.0" type2="literal"/>
    <del value1="ictineu" type1="vehicle" prop="in"
        value2="forall" type2="location"/>
    <add value1="ictineu" type1="vehicle" prop="in"
        value2="recovery" type2="location"/>
</script>
```

The script shown in Extract 7.7 modifies the fact *vehicle* ***in*** *location*. If the *ictineu* entity position (x, y, z) is near to (0, 0, 0) (plus, minus an error), all the facts *ictineu* ***in*** *whatever* are deleted from the world knowledge database and the fact *ictineu* ***in*** *recovery* is added to it.

7.5 Adding planning abilities to the proposed Mission Control System

General concepts about automatic planning as well as several search algorithms based on classical planning techniques have been reviewed. World modeling techniques to keep an updated knowledge database have been also introduced. Now, how all these algorithms are implemented as components in the previously introduced control architecture and how they are related among them and the other components in the architecture is detailed.

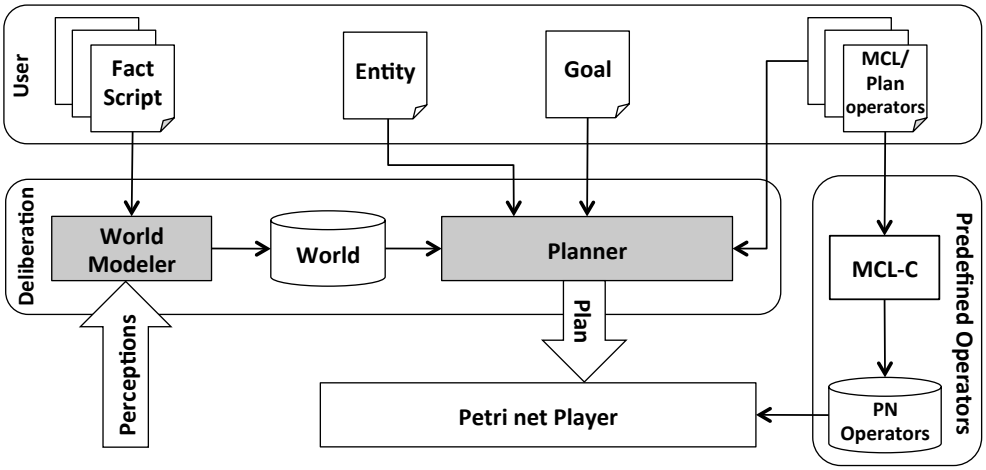


Figure 7.5: Deliberation components used to build on-line plans.

The two components added into the architecture to provide on-board planning capabilities are: a world modeler and a planner, see Figure 7.5. The former, receives perceptions through the AAC and applying a set of scripts defined by the user add and remove facts into the current world state, the kno0wledge database. The latter, is a classical domain-independent planner algorithm, that, given a set of facts provided by the world modeler and a list of available operators and entities, generates a plan to achieve the goals described by the user. The current implementation is a simple BFS algorithm working in the state-space according the restricted model presented in Definition 7.1.1.

Instead of using the vehicle primitives as basic planning operators, off-line missions programmed in MCL has been used as planning operators to solve a particular phases of a more general mission. The use of MCL mission operators allow us to control the vehicle behavior keeping the planner as simple as possible. Moreover, time and resources are not taken into account when planning, avoiding to anticipate the state of the resources as well as to take decisions based on the worst case possible that is the usual approach when planning with resources. A positive cost value associated to each planning operator is used by the planner to find the less costly combination of operators that brings to the solution. Finally, to describe a mission, the user has to provide a set of scripts used to model the world, the list of entities available in the world with all their attributes and the set of goals to achieve in addition to the planning operators.

Both components, the world modeler and the planner, work together to pro-

vide deliberative capabilities to the system. On one hand, the world modeler provides the initial state and tracks the changes produced in the world state. On the other hand, the planner generates a plan containing the a sequence of MCL mission operator instances to execute as well as the set of predictable facts in the world model before and during the execution of each operator as described in Definition 7.2.4. Planning operators contain which are the more likely changes to be produced in the world if an operator is applied but when a planning operator is executed these changes are not introduced in the world model. Only when the vehicle sensors detect that some values have changed, transmit these perceptions to the world modeler and, according to the available fact provider scripts, introduces or deletes facts in the world model. If the facts generated in the world modeler component do not correspond to the facts estimated in the plan, then it is necessary to re-plan. This is the main drawback of the proposed solution. However, our main intention is to generate new plans as quick as possible instead of having plans more accurate but slower to generate because even that ones may fail when dealing with the real environment. Thus, it is preferable to be able to rebuild a new plan faster than generate a priori more reliable plans but slower to obtain. Additional advantages of this system comes from the possibility to use any planning algorithm, even if it has not been designed to be used in real time on-board an autonomous vehicle. The utilization of MCL mission operators presents also the advantages described in Section 7.2.3.

The relation of the planner and the world modeler components with the rest of the components in the control architecture is shown in Figure 7.6. This figure presents the four control loops that appears within the proposed architecture. The lower-level control loop contains the velocity controller that operates at a frequency between 10 - 100Hz. It is in the reactive layer and it is responsible for sending set-points to the actuators. The second control loop is also in the reactive layer and runs at a frequency of 5Hz. It coordinates the primitive responses and send the resulting set-points to the velocity controller. The third control loop appears in the control execution layer. It reacts to any new event with less than a second. Using the events received from the reactive layer it controls which primitives must be in operation and which not, following a given MCL planning operator. Finally, the higher-level control loop is able to modify the plan executed in the control execution layer depending on the received perceptions. It can react to new facts in the world model within few seconds. Hence, primitives must be

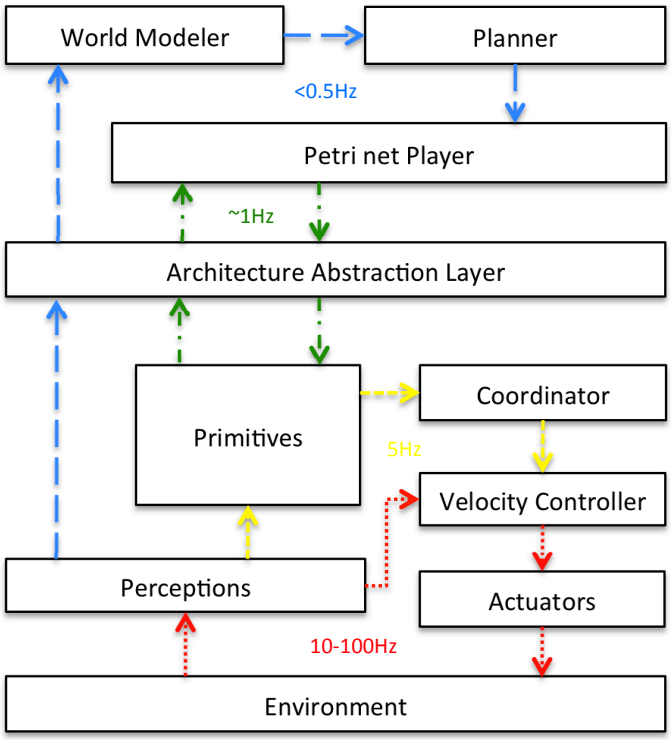


Figure 7.6: The four control loops within the proposed hybrid architecture.

able to react to fast changes in the real world while the planner has to look only for major changes in order to build the new plan to execute.

The inclusion of a this deliberative system within the previous architecture does not provide a greater degree of intelligence than the achieved using an off-line plan. However, it dramatically simplifies the way in which the mission is described avoiding, moreover, possible errors or oversights done by the user when describing the mission plan.

Chapter 8

Experimental results

Several missions have been programmed and executed using the proposed Mission Control System (MCS). The main platforms in which these experiments have been performed are the two Autonomous Underwater Vehicles (AUVs) Ictineu AUV [Ribas et al., 2007] and Sparus AUV [Hurtos et al., 2010]. Experiments have been carried on different locations including water tanks, the shoreline, rivers and dams. Moreover, several experiments have been executed in a Hardware In the Loop (HIL) simulator named Neptune [Ridao et al., 2004a]. In these missions, the identified dynamic models of an AUV [Ridao et al., 2004b] as well as an Autonomous Surface Craft (ASC) [Goden and Pascoal, 2001] have been used to obtain a more realistic results. All the vehicles, both real or simulated, implements the Component Oriented Layer-based Architecture for Autonomy (COLA2) presented in Chapter 3.

First experiments applying an earlier version of the proposed MCS were reported in several works. These experiments present a simulation of the tasks carried out in the Student Autonomous Underwater Challenge-Europe (SAUC-E) 2006 [Palomeras et al., 2006b] and the real execution performed by the Ictineu AUV in the competition's final run [Ribas et al., 2007]. This first approach was also reported in a journal article [Carreras et al., 2007] were a simplified scientific mission was performed in the shoreline. In parallel with these experiments, an industrial application to inspect a dam was studied. Again, first results were obtained using the Neptune HIL simulator [Palomeras et al., 2006a] while real divings were reported three years later with the current MCS [Palomeras et al., 2009c; Ridao et al., 2010]. Some scientific and industrial interest applications

have been performed. One of them is the acquisition of several photo-mosaics in the *Ebre* river with Ictineu AUV. This data was obtained to estimate the extension of the invasive specie of *zebra mussels*. Photo-mosaics were also gathered in the natural reserve of *Monte da Guia* in the Azores islands. The environmental information recollected was used to study and model the animals habitat [Schmiing et al., 2009]. Industrial applications of cable tracking have been studied too. While specialized cable tracking primitive algorithms have been tested in a water tank with the Ictineu AUV [El-Fakdi et al., 2010], a whole cable tracking mission involving several phases have been simulated. Finally, some missions to study the coordination among several vehicles using the Petri net formalism [Palomeras et al., 2010c] as well as a preliminary work to interface an on-board planning system within the presented MCS [Palomeras et al., 2010b] have been tested in simulation.

This chapter begins introducing the main primitives implemented to perform the proposed missions. In general, the success of an autonomous mission is highly related with the success of its primitives [Kortenkamp and Simmons, 2008]. Then, four representative missions are introduced. The first one is a dam inspection mission in the context of an industrial application and the second a visual survey in a zone of scientific interest. Both missions are defined off-line using the Mission Control Language (MCL) and executed using a single vehicle. In the first mission the vehicle is connected through an umbilical link to allow the operators to monitor and take part in the mission if necessary. Also, the umbilical is used to power-up the vehicle. In the second mission, the robot is powered by batteries and runs completely autonomous. The third mission, executed in the HIL simulator Neptune, prove how coordination constraints can be used to control a mission involving several vehicles. Here, an ASC and an AUV have to collaborate to georeference several Object Of Interests (OOIs). The last mission is also inspired by an industrial application. It presents a standard cable tracking mission scenario. This mission shows the pros and cons of using planning techniques on-board an autonomous vehicle versus a more traditional off-line mission description. While the whole mission has been simulated, the main primitives involved in it have been individually tested in a water tank.

8.1 Primitives

Despite the purpose of this chapter is to verify the MCS embodied in the control architecture, the main primitives available in the vehicles are first introduced. As the HIL simulator reproduces the interface module, implemented primitives can be executed either by a real vehicle or by a simulated one. Several primitives can be executed simultaneously controlling several Degree of Freedoms (DOFs). Then, the coordinator component have to merge all the responses following the algorithm presented in Extract 3.1. The following primitives are implemented as behavior components in the guidance and control module, see Chapter 3 for more information about the COLA2 organization.

- **heading**: Given a desired angle, the robot rotates to reach it taking the shortest direction and then keeps this orientation. A simple Proportional Integral Derivative (PID) is employed in this primitive. It uses the localization data provided by the navigator processing unit and controls only the vehicle's Yaw DOF.

- **altitude**: Moves the vehicle to a specific altitude with respect to the seabed and keeps it. A simple PID is employed in this primitive. It uses the data provided by the obstacle detector and controls only the vehicle's Heave DOF.

- **depth**: Moves the vehicle to a specific depth with respect to the surface and keeps it. A simple PID is employed in this primitive. It uses the localization data provided by the navigator and controls only the vehicle's Heave DOF.

- **surface**: Rises the vehicle to the surface. It uses the localization data provided by the navigator and controls only the vehicle's Heave DOF.

- **emergencySurface**: Performs an emergency surface dropping a safety weight. Moreover, enables a beacon to facilitate the vehicles recovery.

- **goto**: It implements a simple 2D Line Of Sight (LOS) algorithm with cross tracking error [Healey, 2006]. It is used to guide the robot towards the desired way-point. The localization data provided by the navigator is used to control the path in both the Surge and Yaw DOFs.

- **trajectory**: Performs a survey following a set of 2D way-points. Uses the same LOS guidance algorithm than the *goto* primitive. The localization data provided by the navigator is used to control the path in both Surge and Yaw DOFs. The trajectories obtained with this primitive are not very smooth because when the AUV reaches one of the way-points in the trajectory path it stops and turns over itself until heading the next way-point.

- **searchPattern**: Performs a sinusoidal trajectory following a specified direction and increasing the amplitude as it moves away from the initial point. The localization data is provided by the navigator and the behavior controls the path in both Surge and Yaw DOFs.

- **stationKeeping**: It uses the images gathered by the down-looking color camera and the localization data provided by the navigator to visually keep the vehicle's position [Cufí et al., 2002]. It controls the vehicle in all possible DOFs (Surge, Sway if available, Yaw and Heave).

There are several primitives that do not affect the vehicles movement. They are used to send an event if an alarm raises, to enable or disable a processing unit detector, to check the value of a specific variable or similar. These primitives, implemented as processing units, are listed next.

- **initializeVehicle**: Checks all the sensors and actuators available in the vehicle. If everything is ok it starts the logs of all the components and finishes successfully. Otherwise, it finalizes in a fail state.

- **stopVehicle**: finalize the logs and stops all the components.

- **alarm**: Checks several sensors and generates an event when a failure is detected or a value is out of the scope. Checked variables include pressure, temperature and battery levels, as well as water sensors and sensors/actuators status.

- **invalidPositioning**: Checks the quality of the localization data. When this quality is below a given threshold, an event is raised.

- **getPositionFix**: Waits until the Global Positioning System (GPS) component receives some valid data to correct the vehicle's position.

- **takeImages**: Enables/disables the processing unit that gathers images from the camera sensor.
- **objectDetector**: Implements a visual object detector algorithm. Once an OOI is defined, shape and color, a down-looking color camera is used to check if the OOI appears in the images gathered by the camera sensor. If the OOI is detected an event is triggered, otherwise, the primitive keeps processing the new incoming images. Several object detection strategies were developed in the context of the SAUC-E [Hurtos et al., 2010; Ribas et al., 2007].

A Petri Net Building Block (PNBB) task has been defined for each primitive in order to supervise it. These tasks will have the same name than the primitive being supervised but starting with a capital letter. From the mission point of view, instantiate a task is similar than instantiate a primitive because tasks enable a primitive when they are called and disable it when they finalize. Therefore, the *Heading*, *Altitude*, *Depth*, *Surface*, *Goto*, *Trajectory*, *SearchPattern*, *StationKeeping*, *Alarm*, *InvalidPositioning*, *GetPositionFix*, *TakeImages* and *ObjectDetector* tasks have been defined in order to be used in an MCL mission. In addition to these presented primitives/tasks, specific ones will be developed for each particular mission.

8.2 Example 1: Dam inspection

Although there are several companies claiming to provide underwater robots for dam inspection like Seabotix, VideoRay, FrugoSurvey or InuktunServices, often, none of them is providing an integral solution to the dam inspection problem. Normally, they propose the use of small class Remotely Operated Vehicles (ROVs), working as teleoperated cameras for video recording, to replace the professional divers who traditionally occupied this place. There exist very few research precedents providing an added value solution. One of the most relevant works is the ROV3 system developed by the researchers of the Institut de recherche HydroQuebec, Canada [Cote and Lavallee, 1995]. It is a small ROV, localized through an Low BaseLine (LBL) system, which makes use of a multi-beam sonar for collision avoidance. The system is able to control the distance to the wall and includes several video cameras as well as a laser system for 2D and 3D

measurements. The COMEX and the Electricité De France companies, France, developed a similar project [Poupart et al., 2000]. In this case, a ROV manufactured by COMEX was localized using 5 LBL transponders. Again, several video cameras together with 2D, double spot, laser system were used to take measurements. The Soniworks Company is selling a very accurate wired LBL navigation system to localize a ROV with centimetric accuracy. The system is combined with a GPS to georeference the imagery gathered with the ROV. Nevertheless, the system is not able to register the images to provide a big image mosaic of the surveyed area. Moreover, in all the previous systems the use of LBL makes the operation tedious due to calibration. In order to focus on real problems, our team contacted with FECSA-ENDESA Spanish hydroelectric company to identify the tasks of interest. This meeting, allowed us to identify different mission scenarios. On December 2007 first experiments in the dams of Pasteral I and Pasteral II in Girona, Spain, were carried out using Ictineu AUV. In February 2009, see Figure 8.1, new experiments were carried in the same place in order to improve the data quality obtained one year before. These new data sets allowed us to obtain high quality mosaics from the dam's wall.

Civil engineers of the hydroelectric companies, carry out periodic visual inspections of the state of the concrete. Commonly this is achieved through a careful visualization of a video recorded by a professional diver or a ROV. Our approach to this problem consist on the use of an AUV which follows a pre-programmed path facing the wall while snapping images. A localization system based on a moored buoy equipped with a Differential Global Positioning System (DGPS) receiver, an Ultra Short BaseLine (USBL) transceiver and a Motion Reference Unit (MRU) is used to georeference the AUV position. On-board, a navigation system based on a Doppler Velocity Log (DVL) and a Attitude Heading Reference System - Fiber Optic Gyroscope (AHRS-FOG) is used for the AUV navigation. During the experiments both navigation systems were not interconnected in real time but, since both equipments were time synchronized, the georeferenced trajectory of the AUV could be extracted through post processing, and hence also georeference the imagery. After the mission, the set of gathered images together with the localization data obtained in the buoy were combined and used to setup an image mosaic of the wall of the dam using an image mosaicking system which has been also developed in our lab [Garcia et al., 2001].

During the experiments carried out in December 2007, two main problems

Figure 8.1: Dam inspection setup during the experiments carried out in Girona (Spain).

were detected:

- How to detect the robot orientation with respect to the concrete of the dam due to the magnetic perturbations provoked by the iron within the concrete to the AUV compass.
- Defects in the illumination of the images taken by the AUV camera.

To deal with the first problem two solutions were proposed. First an Extended Kalman Filter (EKF) method was proposed to detect and track the wall of the dam using an imaging sonar [Kazmi et al., 2009]. The wall, a line, was represented in polar coordinates from which the distance between the robot and the wall, and their relative orientation can be easily extracted. The second alternative was to install a Fiber Optic Gyroscope (FOG) in the vehicle to avoid the magnetic disturbances. Latter solution was the implemented during the experiments. The second issue was solved improving the lighting system as well as using a more sensitive underwater camera.

8.2.1 Mission description

The dam inspection mission is described as follows. First, the vehicle is initialized checking all the subsystems and enabling the sensor logs. Then, in parallel with the rest of the mission, an alarm monitor is used to raise an event in case the pressure/temperature exceeds a threshold or if a water leakage is detected inside a pressure vessel. Using the *goto* behavior the vehicle goes to the initial way-point of the survey and, after achieving the desired orientation and distance respect to the dam wall, the survey starts. Human operators may assist the vehicle in this crucial step. Then. imagery of the dam's wall is recorded during the survey using the forward locking camera. When the survey finalizes, the camera is disabled and the vehicle goes to the recovery position where it stops. If during the mission the alarm monitor raises an event, the mission has to be aborted and the vehicle surfaces using an emergency system. As the whole mission is inside a try-catch structure, if a task is unable to accomplish its goal the try block is cancelled and the vehicle surfaces aborting the mission in a controlled manner.

A couple of extra primitives have been specially designed to perform this mission:

- **wallInspection**: Follows a sequence of 2D way-points in front of a wall. It uses the localization data provided by the on-board navigator and controls the vehicle in Surge and Heave DOFs.
- **distance**: Keeps a specific distance with respect to a vertical wall. A simple PID is employed in this primitive. It uses the data provided by the obstacle detector processing unit and controls only the vehicle's Surge DOF. This primitive has several operating modes configurable through parameters. First, it can only achieve the correct distance between the vehicle and the wall and then finalizes or achieve this distance and then keeps it until the primitive is aborted. Second, the primitive can be supervised by a human operator that tele-operates the vehicle until achieve the desired distance. These two working modes have also been added to the **heading** primitive.

In industrial applications, sometimes the user is more interested in a semi-autonomous operation approach than in a completely autonomous one. For safety reasons, a human operator can be supervising the autonomous mission

being able to abort it if something unexpected happens. Also, the combination of tele-operated tasks with autonomous ones can be useful in some situations. In this particular case, the approximation to the wall to setup the initial heading and distance could be done by a human operator while the most tedious part of keeping the desired heading and distance while the vehicle performs a full survey in front of the wall is better to be performed autonomously by the vehicle itself. Moreover, the human operator can monitor the images taken by the robot camera and also the trajectory recorded by the navigation system and abort the mission if the data is not satisfactory. The code to carry out the proposed mission is presented in Extract 8.1.

Extract 8.1: Dam inspection mission.

```
mission
  monitor
    try
      InitializeVehicle() ;
      Goto(initial_pos) ;
      parallel
        Heading( angle, timeout, "achieve", "manual" )
      and
        Distance( distance, timeout, "achieve", "manual" )
      ;
      parallel
        WallInspect( path[...] )
      or
        TakeImages()
      or
        parallel
          Heading( angle, timeout, "keep", "auto" )
        and
          Distance( distance, timeout, "keep", "auto" )
      ;
      Goto( recovery_pos );
      Surface() ;
      StopVehicle()
    catch
      Surface() ;
      StopVehicle()
  condition
    Alarm()
  do
    EmergencySurface()
```

The tree shown in Figure 8.2 shows a simplification of the automatically generated Petri net.

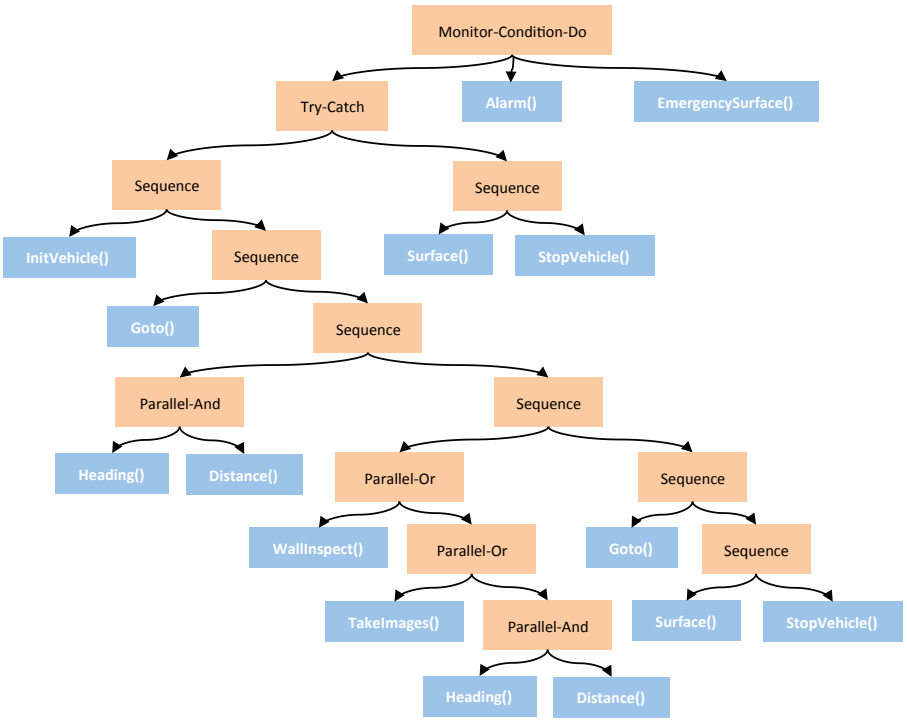


Figure 8.2: Dam inspection mission tree.

8.2.2 Results

In order to obtain the best results, several experiments were carried out keeping the robot perpendicular to the dam's wall but changing the distance to it from 1 to 4 meters. Because the hydroelectric central was generating power, the experiments had to be constrained to a part of the wall sufficiently far from the water inlet, in a very shallow area. Several type of trajectories were tested and the best results were obtained moving the robot vertically and performing the horizontal movements during the upper part. This can be explained because the on-board navigation system used for the horizontal displacement relies on a DVL that works better when the distance with the floor is above 2.5 meters while vertical navigation system relies mainly in a depth sensor that is not affected by the altitude. Figure 8.3 shows the real trajectory performed by the vehicle in one of the experiments. This trajectory has been computed off-line after the experiments integrating the sensors in the moored buoy, USBL, DGPS and MRU, and the sensors in the vehicle, DVL and Motion Reference Unit-Fiber Optic Gyroscope (MRU-FOG). The covered area was approximately of 4×9 meters with

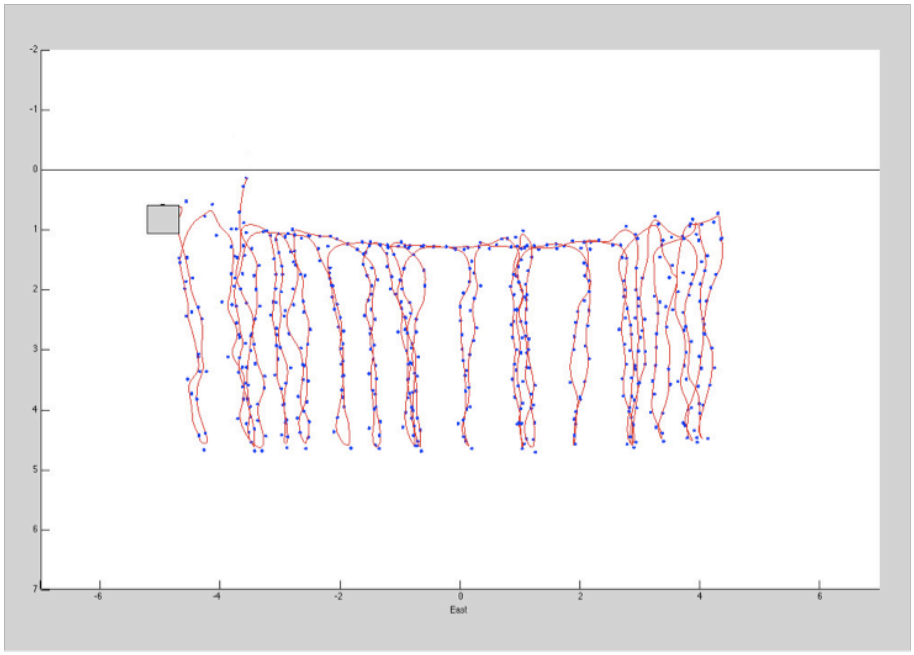


Figure 8.3: Trajectory realized by the AUV in front of the wall.

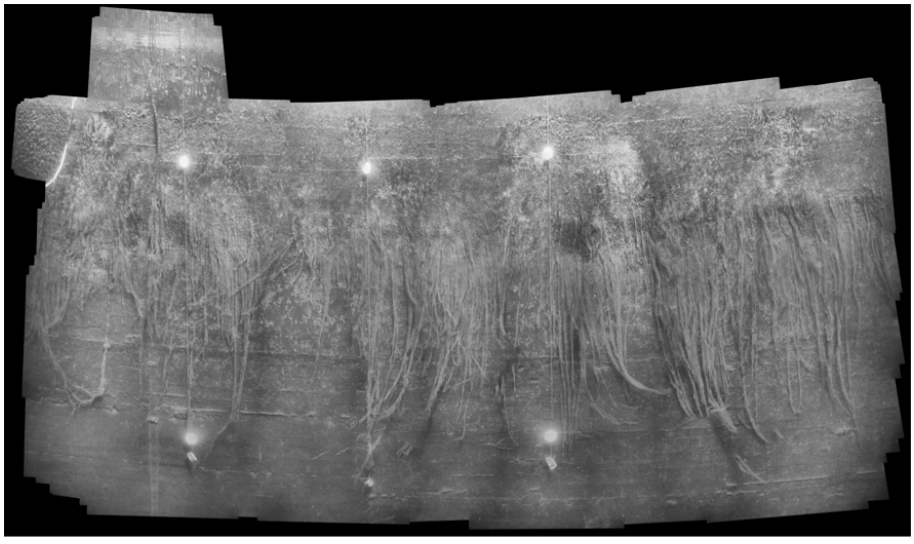


Figure 8.4: Mosaic build after the inspection.

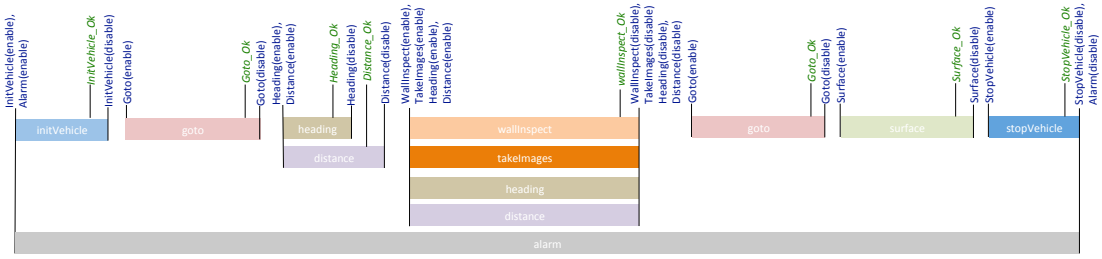


Figure 8.5: Chronogram for a successful execution.

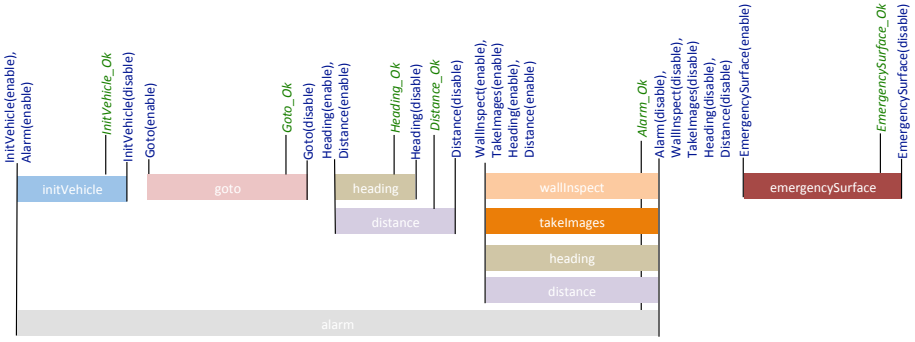


Figure 8.6: Chronogram for a mission in which an alarm is raised.

a wall distance of 1.5 meters and from 1 to 5 meters from the surface. The data position shown in Figure 8.3 was used together with all the captured images to build the mosaic presented in Figure 8.4. The mosaic is a high resolution image with more than 67Mpx, approx. 1 pixel per millimeter, in which the wall can be easily inspected. In Figure 8.4 plenty of algae can be seen on the wall as well as circular marks that were added to it to verify the result with previously known measures [Palomeras et al., 2009a; Ridao et al., 2010].

To see more accurately how actions and events are executed, two chronograms are included. Figure 8.5 shows the sequence of actions and events as well as the primitives under execution in each moment for a mission successfully executed. The length of the bars representing the primitives is purely indicative. It does not correspond with the duration that the primitives were actually running. Figure 8.6 shows a mission in which an *Alarm* event is raised while executing the mission. The chronogram shows how the *monitor-condition-do* structure disables all the primitives under execution and then, the emergency surface primitive starts.

8.3 Example 2: Visual survey

An autonomous survey to map a region of interest is presented in this example to validate a completely autonomous mission carried out using the proposed MCS. Sparus AUV, equipped with a down-looking color camera and several navigation sensors like a DVL and an Attitude and Heading Reference System (AHRS), is required to build a sea-floor photo-mosaic from a zone of scientific interest. The simplicity of use of the MCL allowed the vehicle operators to program the mission, on-board a boat with only a few minutes, once in the region of interest.

8.3.1 Mission description

The propound survey mission is composed by the following phases:

1. The AUV is deployed from the base boat being initialized through a WiFi connection.

2. The vehicle is submerged towards a predefined altitude with respect to the sea bottom.

3. While keeping the altitude and grabbing images, the AUV is guided across a set of way-points following a prefixed trajectory.

4. When the last way-point is reached, the vehicle surfaces and signals its position to be easily recovered by the base boat.

In parallel to these four main steps, several internal alarms are checked. These alarms may raise an event because a water leakage is detected, the vehicle is running out of batteries or due to a pressure or temperature alarm is catched inside one of the pressure vessels. If any of these alarms is raised an emergency stop is required. Moreover if any task is unable to complete its execution within a specific time-out or because some sensor/actuator malfunction, the mission is aborted surfacing the vehicle for recovery. Extract 8.2, presents the survey mission coded in MCL.

Extract 8.2: Survey mission.

```
mission
    monitor
        try
            InitializeVehicle() ;
            Altitude( altitude, timeout, "achieve" ) ;
            parallel
             | Trajectory( velocity, path[...] )
            or
             | TakeImages()
            or
             | Altitude( altitude, timeout, "keep" )
            ;
            Surface() ;
            StopVehicle()
        catch
            Surface() ;
            StopVehicle()
    condition
     | Alarm()
    do
     | EmergencySurface()
```

The mission is basically the sequence of the tasks *InitializeVehicle*, *Altitude*, *Trajectory*, *Surface* and *StopVehicle*. To keep the tasks as simple as possible, the trajectory-following, while keeping the altitude and gathering images from the seabed, is performed composing three tasks in parallel. The whole mission is inside a *try-catch* control structure, then, if any of these tasks fails or a time-out arrives, the catch block is executed while the rest of the tasks are aborted. Moreover, in parallel, an *Alarm* task is being executed. If any of the events checked by this task raises, the whole mission is aborted and the vehicle performs an emergency surface dropping a safety weight.

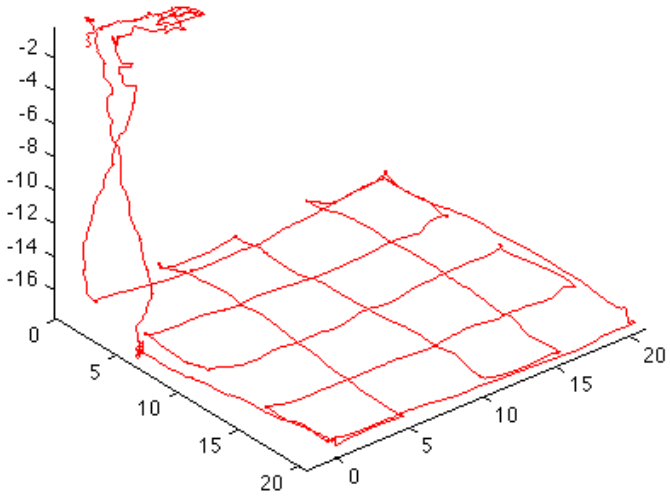


Figure 8.7: Sparus trajectory obtained by dead reckoning when performing a visual survey at the Azores.

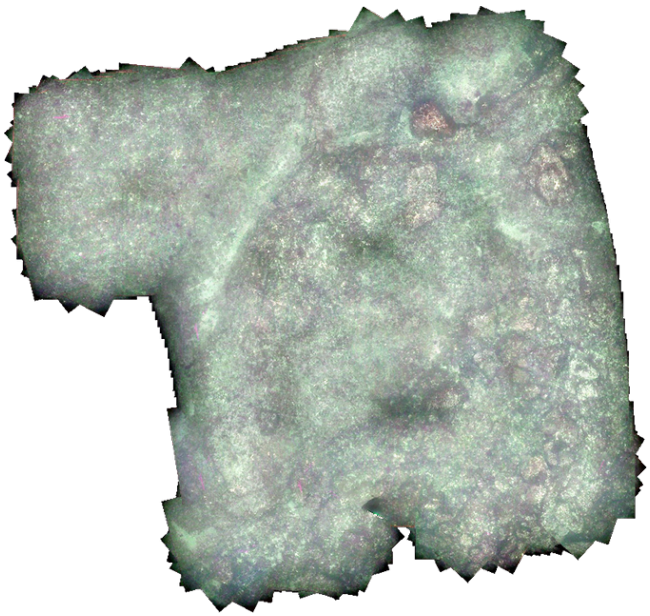


Figure 8.8: Underwater photo-mosaic from the area of interest obtained off-line.

8.3.2 Results

In summer 2010 several autonomous surveys were performed in the Azores islands to obtain seabed photo mosaics covering areas of biological interest [Schmiing et al., 2009]. Figure 8.7 shows the trajectory described by the vehicle Sparus AUV during the survey mission depicted in Extract 8.2. The trajectory was obtained from the navigator component on-board the vehicle. The sea-floor photo-mosaic shown in Figure 8.8 was built off-line using the gathered images [Garcia et al., 2001].

8.4 Example 3: Localization of OOIs

The next mission consists in georeferencing and providing on-line access to a set of images gathered from OOIs lying on the seabed. The images are grabbed by an AUV and transmitted to an ASC acting as a gateway to a base boat. Both vehicles, the AUV and the ASC, have to cooperate to perform this mission. The mission consists of the following steps:

1. The ASC carries the AUV towards the deployment area.
2. When the ASC has deployed the AUV, the underwater vehicle is submerged and the survey begins.
3. Whenever the AUV detects an OOI, it keeps its position over the object and sends a localization request to the ASC.
4. The ASC navigates towards the AUV using a simple Unconstrained Least Squares (UL-S) algorithm.
5. When both vehicles are aligned, the AUV transmits images of the detected OOI to the ASC using the vertical channel. The ASC georeferences these images and re-transmits them to a base station. Then, the survey continues.
6. When the survey is completed, the AUV sends a signal to the ASC. The surface vehicle is guided to the recovery area while the AUV surfaces and waits to be recovered.

If an error is produced or an alarm is detected on the AUV, the ASC is notified and the mission is canceled. On the other hand, if the ASC is unable to navigate

towards the AUV and vertically align with it, the AUV continues the survey without transmitting the gathered data.

In order to perform this cooperative mission additional primitives have been added to the ones previously presented:

- **gotoTarget**: Guides an ASC to a position almost vertically aligned with respect to an AUV. First, the ASC is guided through a circular trajectory to ensure non-collinearity between the AUV and the ASC poses from which GPS fixes and ranges to the AUV, provided by an acoustic modem, were gathered. The AUV is assumed to send its depth to the ASC embodied in the message packet. Then, using the 3D range and the robot depth, the 2D range between both vehicles is computed before localizing the target by means of a simple UL-S algorithm [Cheung et al., 2004]. Once the AUV has been localized, the ASC is guided towards its position.

- **receive/send data**: Receives/sends large packets of data through an acoustic modem. This primitive can be only used when the two vehicles in communication are vertically aligned.

Additionally to the tasks build to supervise the introduced primitives, two tasks that do not supervise any primitive have also been used. These task are named *Void* and *True*. Both are instantaneous and always finalize in the *ok* state. The *Void* task is used for coordination purposes while the *True* task is used in a never-ending loop. The use of never-ending constructions may be dangerous avoiding to reach a final state. These constructions have to be always supervised by hierarchic control structures able to abort them when necessary.

8.4.1 Mission description

The mission code for the AUV and the ASC are programmed below in Extract 8.3 and Extract 8.4. To coordinate the vehicles, a set of constraints between them is coded in MCL as shows Extract 8.5.

Extract 8.3: AUV cooperative mission.

```
mission
  monitor
    try
      InitializeVehicle() ;
      Altitude( altitude, timeout, "achieve" ) : deploy_auv ;
      monitor
        parallel
          Trajectory( velocity, path[...] )
        or
          Altitude( altitude, timeout, "keep" )
      while_condition
        Search( object_description )
      do
        parallel
          Void() : req_georef ;
          StationKeeping( )
        or
          SendData() : send_data
        or
          Void() : georeference_fail
      ;
      Void() : survey_done ;
      Surface() ;
      StopVehicle()
    catch
      Void() : survey_aborted ;
      Surface() ;
      StopVehicle()
  condition
    Alarm() : survey_alarm
  do
    EmergencySurface()
```

Extract 8.3 resembles the mission presented in Extract 8.2. The main difference is that the parallel structure in which the vehicle performs a trajectory while

keeping its altitude has been inserted inside a *monitor-while_condition-do* control structure. Thus, a third task named *Search* is simultaneously executed. If the *Search* task founds an OOI, then, the *do-block* is executed and the *Trajectory* and *Altitude* tasks are aborted. As the *monitor-while_condition-do* control structure is iterative, when the *do-block* finalizes the whole control structure starts again. This *do-block* contains the code to ask for being georeferenced as well as to wait, keeping the current position, until a *send_data* or a *georeference_fail* signal is received from the ASC. If the *Trajectory* task finalizes the pre-programed path, the whole monitor structure finalizes and the AUV sends a *survey_done* signal, surfaces and stops.

Extract 8.4: ASC cooperative mission.

```
mission
    InitializeVehicle() ;
    Goto( velocity, deploy_area ) ;
    Deploy() : deploy_auv ;
    parallel
        while True() do
            Void() : req_georef ;
            if GotoTarget() then
                ReceiveData() : send_data
            else
                Void() : georeference_fail
    or
        Void() : survey_done
    or
        Void() : survey_aborted
    or
        Void() : survey_alarm
    ;
    Goto( velocity, recovery_area ) ;
    StopVehicle()
```

The mission to be executed by the ASC is shown in Extract 8.4. It begins deploying the AUV before waiting for a *georeference_request*. When a request is received, the *GotoTarget* task begins and depending if it finalizes successfully or not the data between both vehicles is transmitted, *ReceiveData()* or a *geo-*

reference_fail is send to the AUV. The mission finalizes when a *survey_done*, *survey_aborted* or *survey_alarm* signal is received from the AUV. When one of these signals arrives, the *parallel-or* control structure finalizes aborting the *while true* iterative structure.

Extract 8.5: Cooperative mission constraints.

```
constraints {
    order { asc:deploy_auv, auv:deploy_auv }
    order { auv:req_georef, asc:req_georef }
    sync { auv:send_data, asc:send_data }
    order { asc:georeference_fail, auv:georeference_fail }
    order { auv:survey_done, asc:survey_done }
    order { auv:survey_aborted, asc:survey_aborted }
    order { auv:survey_alarm, asc:survey_alarm }
```

Six orderings and one synchronization constraint have been defined in Extract 8.5 to coordinate both vehicles. The first ordering, avoids the beginning of the AUV's *Altitude* task before the end of the *Deploy* task in the ASC. Similarly, the second ordering, keeps the ASC stacked in the *Void(): req_georef* task until the AUV has found an OOI. The only way for the AUV to finalize the execution of the *StationKeeping* task is to receive a synchronization signal *send_data* or an ordering signal *georeference_fail* from the ASC. On the other hand, the only way for the ASC to finalize the *while true* iterative structure is to receive a signal from the AUV indicating that the survey has finalized, it has been aborted or an alarm has been raised.

8.4.2 Results

The cooperative mission has been executed in the HIL simulator Neptune using an hydrodynamic model for the AUV [Ridao et al., 2004a] and the ASC [Goden and Pascoal, 2001]. The use of a simulated environment allows us to control all the events produced during the execution as well as to force errors, time-outs or alarms. Figure 8.9 shows the trajectory performed by the AUV, continuous blue line, and the ASC, dashed red line, as well as the position of the four OOIs to be photographed and georeferenced, black circles. Additionally, the position in which the AUV has detected the four OOIs (*) and the ASC position after

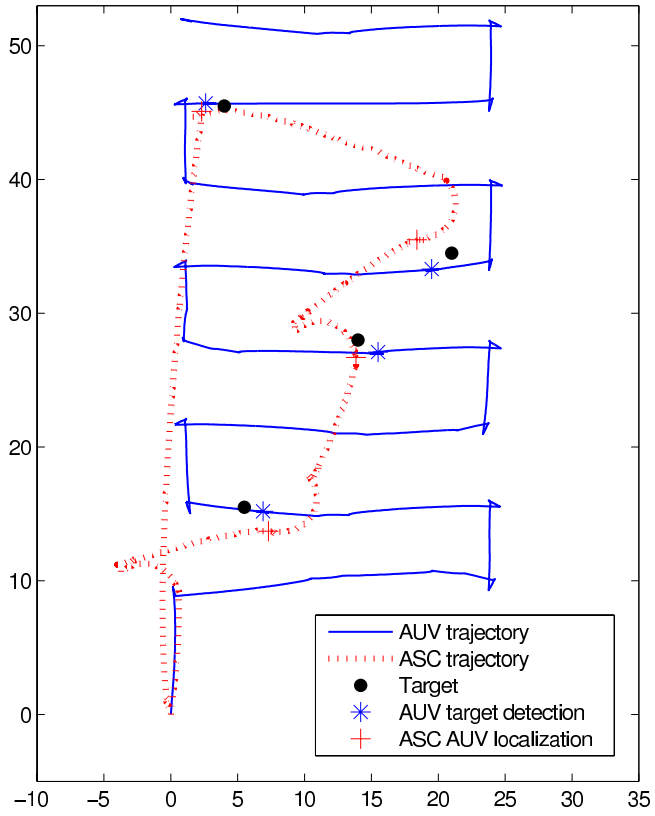


Figure 8.9: Obtained trajectories after simulating the coordinated mission.

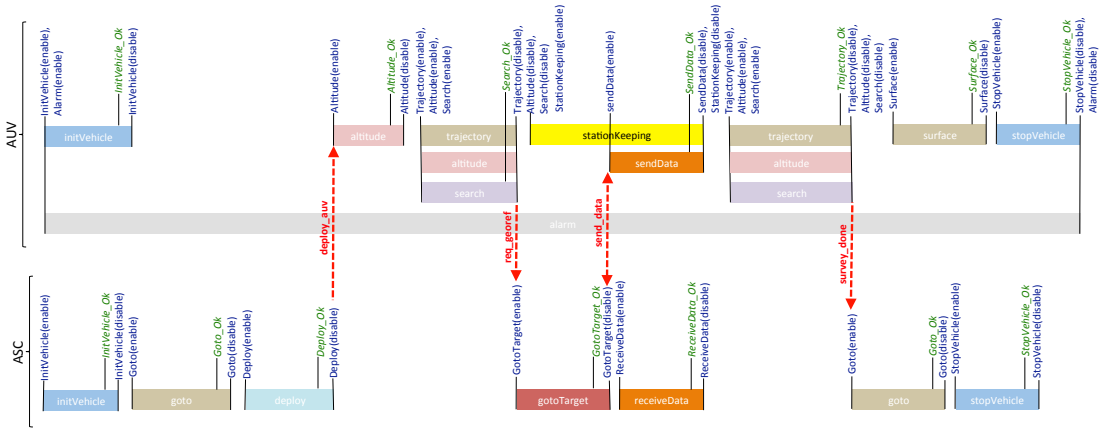


Figure 8.10: Chronogram for a coordinated mission.

localize and navigate towards the AUV (+) are also marked in Figure 8.9.

The amount of data to be communicated between both vehicles when they are not using the vertical channel has to be minimal because of the utilization of low band width acoustic modems. Then, not only the algorithm used by the ASC to discover the AUV position has to use few packages of data but also all the coordination mechanism has to coordinate the vehicles using the minimum number of data packages.

A chronogram has been included in this mission to see the sequence of actions and events as well as the coordination of tasks between both vehicles arranged in time. Figure 8.10 this chronogram in which only one OOI has been found before completing the survey.

8.5 Example 4: Cable tracking

The use of professional divers for the inspection and maintenance of underwater cables/pipelines is limited by depth and time. ROVs represent an alternative to human divers. The main drawback of using ROVs for surveillance missions resides in its cost, since it increases rapidly with depth, because of the requirements for bigger umbilicals and support ship. All those reasons point towards AUVs as an alternative solution for such missions. An AUV can be deployed from the coast without help of any ship, perform all the tracking mission by itself gathering all useful data from sensors and surface at the desired location for recovery.

Several systems have been developed for underwater cable/pipeline inspection purposes. Basically, the technology applied classifies the methodologies in three big groups depending on the sensing device used for tracking the cable/pipeline: magnetometers [Asakawa et al., 2002; Ito et al., 1994], sonar [Evans et al., 2003; Iwanowski, 1994] and vision based methods [Balasuriya and Ura, 2002; Ortiz et al., 2009]. Compared to magnetometer or sonar technology, vision cameras, apart from being a passive sensor, provide far more information with a larger frequency update, are inexpensive, much less voluminous and can be powered with a few watts. Light-Emitting Diode (LED) technology is also contributing to reducing the size of the lighting infrastructure and the related power needs, what also matters in this case. The mission proposed here consists on building a georeferenced photo-mosaic of an underwater cable. The vehicle must be able to search the underwater cable, follow it while gathering images and keep and accurate positioning during all the mission. Then, combining the vehicle's navigation data with the acquired images, the mosaic is composed off-line. For this experiments, a vision-based system developed at the University of the Balearic islands [Ortiz et al., 2009] has been chosen to track a submerged cable in a controlled environment. The vision algorithm computes the polar coordinates (ρ, Θ) of the straight line corresponding to the detected cable in the image plane. Being (ρ, Θ) the parameters of the cable line, the Cartesian coordinates $(x,\ y)$ of any point along the line must satisfy

$$\rho = x \times cos(\Theta) + y \times sin(\Theta). \tag{8.1}$$

As shown in Figure 8.11, equation (8.1) allows us to obtain the coordinates of the cable intersections with the image boundaries (X_u,Y_u) and (X_L,Y_L), thus the mid point of the straight line (x_g,y_g) can be easily computed by

$$(x_g, y_g = \frac{X_L + X_u}{2}, \frac{Y_u + Y_L}{2}). \tag{8.2}$$

The computed parameters (ρ, Θ, x_g, y_g) together with its derivatives are sent to the guidance and control module in order to be used by a primitive that tracks the cable. Figure 8.12 shows a real image of Ictineu AUV while detecting a cable.

A primitive implemented by Andres El-fakdi [El-Fakdi et al., 2010] to track an underwater cable using the Ictineu AUV has been used in this mission to have a really adaptable method to track a cable in a changing environment. The goal

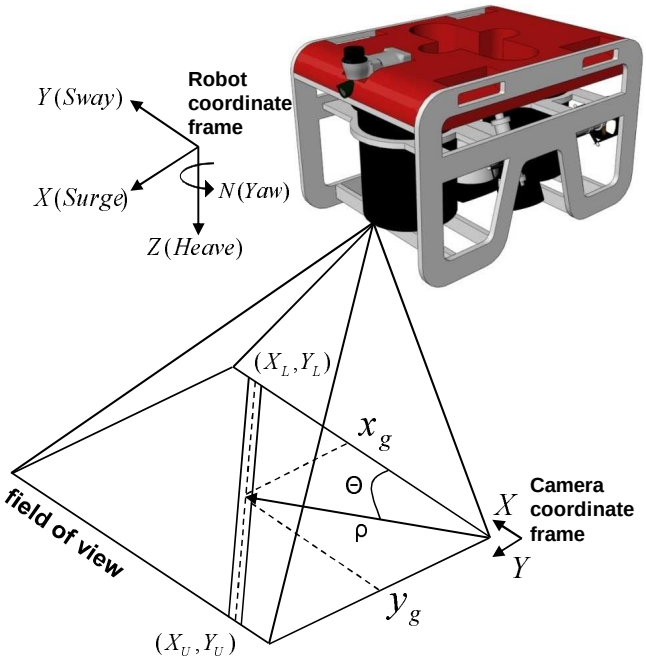


Figure 8.11: Coordinates of the target cable with respect to the Ictineu AUV.

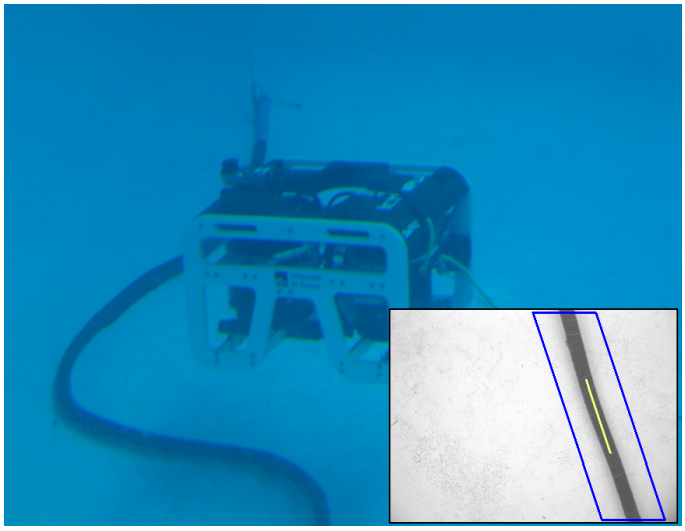


Figure 8.12: Ictineu AUV in the test pool. Small bottom-right image: Detected cable.

of this primitive is:

- **cableTracking**: Whenever an underwater cable is within the field of view of the vehicle's downward-looking camera, this primitive controls the vehicle in Surge, Sway and Yaw DOFs to guide the robot in order to follow the cable. It uses an stimulus-to-action mapping formerly learn in simulation that is in continuous adaptation by means of a Natural Actor Critic (NAC) algorithm [El-Fakdi et al., 2010].

8.5.1 Mission description

The mission to execute has been programmed using two paradigms. First, an off-line predefined mission plan coded in MCL and, second, using an on-board planner able to automatically combine a set of planning operators.

8.5.1.1 Off-line mission

Extract 8.6 shows the off-line solution. The mission starts after initializing the vehicle, taking a GPS position fix and driving the vehicle to the initial position. Then, the vehicle is submerged until an altitude of one meter with respect to the sea-floor is achieved. Thereafter, two tasks are executed in parallel, the *SearchPattern* and the *Search* task that is configured to recognize the underwater cable. If the *Search* task successfully finds the cable, both tasks are aborted and the *CableTracking* task is enabled. When the *CableTracking* task misses the cable, the *SearchPattern* and the *Search* tasks are enabled again. On the other hand, if the *SearchPattern* finishes because it is unable to find the cable before a time-out, the whole structure ends and the vehicle surfaces and goes to the recovery position. In order to keep the vehicle always localized, when the *InvalidPositioning* tasks raises an *ok* event, the *monitor* block is aborted and the *Surface* and *GetPositionFix* tasks are executed. Because when the vehicle is submerged the localization is based on dead reackoning techniques, the position uncertainty grows continuously. When this uncertainty reaches a threshold, it is necessary to take a position fix via GPS to reduces the drift. When the vehicle is correctly positioned, the execution continues. The whole mission is inside a *try-catch* control structure. If any of the tasks within this block finalizes unexpectedly, then the catch block is executed aborting the mission and driving the vehicle to the

recovery zone. Moreover, parallel to all this code, an *Alarm* task is under execution. If any event checked by this task raises, the monitor block is aborted and the vehicle comes to the surface by means of a drop-weight emergency system.

Extract 8.6: Cable tracking off-line mission.

```
mission
  monitor
    try
      InitializeVehicle() ;
      GetPositionFix( ) ;
      Goto( initial_pos ) ;
      monitor
        Altitude( altitude, timeout, "achieve" ) ;
        parallel
          monitor
            SearchPattern( search_timeout )
          while_condition
            Search( "cable_description" )
          do
            CableTracking( params_NAC )
        or
          TakeImages()
        or
          Altitude( altitude, timeout, "keep" )
      while_condition
        InvalidPositioning()
      do
        Surface() ;
        GetPositionFix()
      ;
      Surface() ;
      Goto( recovery_pos ) ;
      StopVehicle()
    catch
      Surface() ;
      Goto( recovery_pos ) ;
      StopVehicle()
  condition
    Alarm()
  do
    EmergencySurface()
```

8.5.1.2 On-board planning

If the mission is programmed as a planning problem, a set of planning operators must be first defined specifying its preconditions and effects. For this particular mission, seven planning operators have been used: *GotoOp*, *SurfaceOp*, *AchieveAltitudeOp*, *TakeFixOp*, *CheckAlarmOp*, *SearchCableOp* and *CableTrackingOp*. These planning operators are described in Table 8.1[1]. They include a piece of MCL code together with the description of their preconditions and effects.

The MCL code for the operator *SearchCableOp* is presented in Extract 8.7. This MCL operator shows three tasks executed in parallel. If the *Search* task founds the cable, then this fact will be added to the knowledge database, as expected in the plan, and the *SearchCableOp* operator will finalize in the *ok* state continuing the execution of the mission plan previously computed. However, if the planning operator finalizes because of a time-out produced by the *SearchPattern* task, then the *SearchCableOp* is aborted, the world modeler adds an unexpected fact into the knowledge database and a new plan has to be computed.

Extract 8.7: MCL code for the *SearchCableOp* planning operator.

```
mcl_plan_op SearchCableOp( timeout, cable_description, altitude )
    parallel
      | SearchPattern( timeout )
    or
      | Search( cable_description )
    or
      | Altitude( altitude, "keep" )
```

The world modeler is responsible for keeping the facts that describe the world up-to-date. Table 8.2 shows a list with some of the possible facts in the knowledge database, the scripts to generate each one of these facts, the primitives involved and if these facts can be in the initial state (W_0) or they are part of the goal state.

Once a mission plan is generated, it is composed by a sequence of planning operators with the facts that should be present in the world model before and after the operator's execution. If any operator finishes with a *fail* or the facts

[1]Do not confuse the robot's primitives, i.e. *goto*, with the PNBB tasks which supervise these primitives, i.e. *Goto*, or the planning operators, i.e. *GotoOp*.

Table 8.1: Planning operators for the cable tracking mission.

op:	TakeFixOp *Robot* **r**
pre:	**r** surface, **r** position_bad, **r** no_alarms
add:	**r** position_ok
del:	**r** position_bad
mcl:	*TakeFixMCLOp()*
op:	SearchCableOp *Robot* **r** *Object* **o**
pre:	**r** seafloor, **r** position_ok, **r** no_alarms, **r** in inspection, **o** not_found
add:	**o** found
del:	**o** not_found
mcl:	*SearchCableMCLOp(timeout, o.description, altitude)*
op:	CableTrackingOp *Robot* **r** *Object* **o**
pre:	**r** position_ok, **r** no_alarms, **o** found, **o** not_mapped
add:	**o** mapped
del:	**o** not_mapped
mcl:	*CalbeTrackingMCLOp(altitude, params_NAC)*
op:	GotoOp *Robot* **r** *Zone* **a** *Zone* **b**
pre:	**r** in **a**, **r** position_ok, **r** no_alarms
add:	**r** in **b**
del:	**r** in **a**
expr:	**a** != **b**
mcl:	*GotoMCLOp(b.x, b.y)*
op:	SurfaceOp *Robot* **r**
pre:	**r** seafloor
add:	**r** surface
del:	**r** seafloor
mcl:	*SurfaceMCLOp()*
op:	CheckAlarmOp *Robot* **r**
pre:	**r** alarm
add:	**r** no_alarm
del:	**r** alarm
mcl:	*CheckAlarmMCLOp()*

Table 8.2: Facts that can be generated during the mission execution by the world modeler scripts.

Fact	WM Script	Primitive	W_0/Goal
robot position_bad	navigation_status	InvalidPositioning	W_0
robot position_ok	navigation_status	InvalidPositioning	–
robot surface	robot_altitude	Navigator	W_0/Goal
robot seabottom	robot_altitude	Navigator	–
robot no_alarms	alarms_status	Alarm	W_0
robot alarms	alarms_status	Alarm	–
robot in deploy	robot_position	Navigator	W_0
robot in inspection	robot_position	Navigator	–
robot in recovery	robot_position	Navigator	Goal
cable not_found	cable_status	SearchCable	W_0
cable found	cable_status	SearchCable	–
cable not_mapped	mission_status	SearchCable	W_0
cable mapped	mission_status	SearchCable	Goal

that the world modeler adds in the knowledge database do not match with the ones expected by the planner, a new plan is build from the current situation. The initial plan obtained in the proposed cable tracking mission is composed by a sequence of seven planning operators: *TakeFixOp()*, *GotoOp(inspection)*, *AltitudeOp(altitude, "achieve")*, *SearchOp("cable", altitude, timeout)*, *CableTrackingOp(altitude, param_NAC)*, *SurfaceOp()* and *GotoOp(recovery)*. However, if the vehicle loses the cable, the navigation quality becomes poor or an alarm raises, these states will be mapped by one of the world modeler scripts as a new fact in the knowledge database. As this new fact will not coincide with the ones expected in the mission plan, a new plan will be generated by the on-board planner aborting the previous one. Programming a mission using planning operators and world modeling scripts can be less intuitive than coding an off-line imperative mission. However, for complex missions where a large number of events may take place, the use of an on-board planner simplifies the mission description and avoids possible errors introduced by the user.

8.5.2 Results

While the main primitives has been tested with Ictineu AUV in a water pool, the whole proposed mission has been simulated using the HIL simulator Neptune [Ridao et al., 2004a]. The simulated environment allows us controlling all the events produced during the mission execution and also gives us the opportunity to induce some errors, task failures and alarms forcing the re-planing process. Thus, it is easier to check if the vehicle reacts as specified in the mission plan. The simulation environment includes a 140 meters long cable placed on a flat bottom. The cable contains five sections that are buried in the sand in which the cable can not be detected by the computer vision system. The on-board navigation system is designed so that the vehicle can not travel more than 50 meters without getting a GPS fix.

Similar executions have been obtained by the off-line user predefined plan and the on-board automatically generated plan. The main difference is that each time that the automatically generated plan fails because the assumptions done about the world are different than the facts provided by the world modeler, a new plan has to be build. While the on-board planning is generating this new plan the vehicle keeps its position until the new plan is ready. Twelve plans has been generated by the on-board planner to complete the mission shown in Figure 8.13(a). However, due to the simplicity of the mission, less than a second have been required to compute each one of these plans. An example of some of these plans and the world state that has motivated them is shown in Figure 8.14.

Figure 8.13(a) shows the trajectory performed by the vehicle during the execution of the proposed mission. The AUV is submerged and localizes the cable, see Figure 8.13(b). Then, it tracks the underwater cable for more than 130 meters using the *cableTracking* primitive specially developed for this purpose. The Reinforcement Learning (RL) algorithm that implements this primitive is constantly updating the stimulus-to-action mapping to improve its performance. Each time that the cable was not visible by the vision system, i.e. because the cable was buried by sand, a search procedure based on the last known bearing is enabled to find it, see Figure 8.13(c). Moreover, if the vehicle navigation accuracy is below a threshold, the vehicle surfaces to take a position fix, see Figure 8.13(d). The mission ends if after several time searching for the cable it is impossible to find it, see Figure 8.13(e), or if an alarm raises, i.e. the vehicle runs out of batteries.

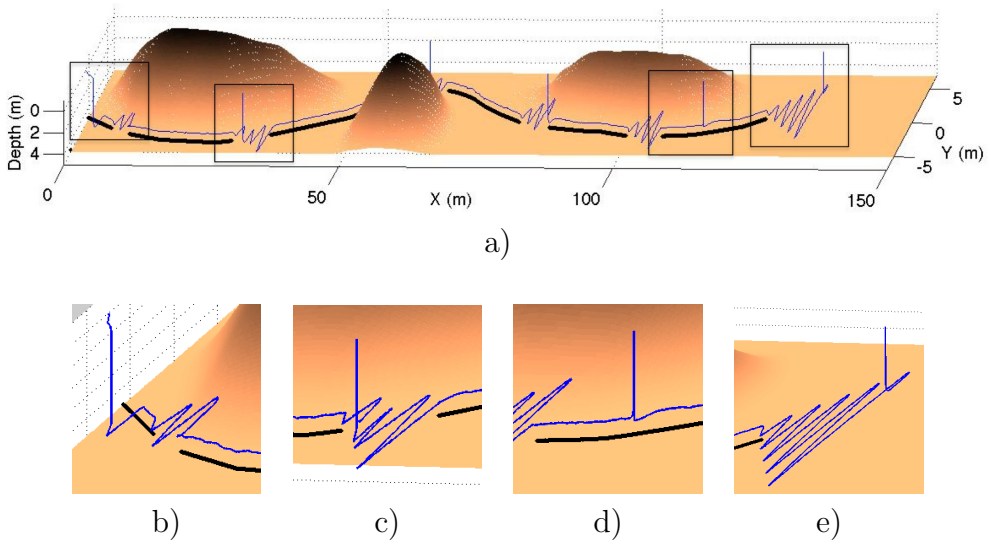


Figure 8.13: Cable tracking mission simulation.

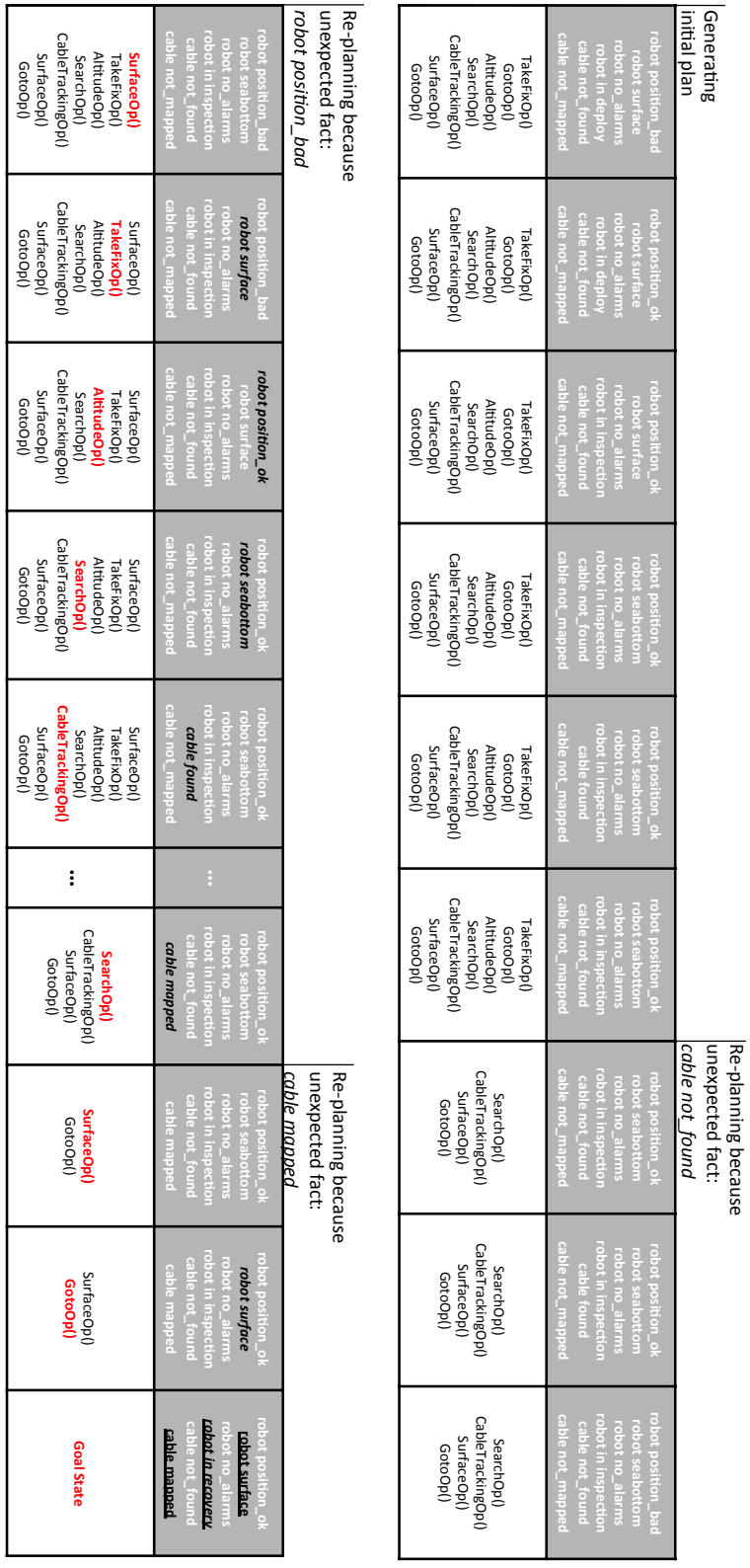


Figure 8.14: Example of on-line mission plans during the execution of the cable tracking mission.

Chapter 9

Conclusion

This chapter concludes the work presented throughout this document. It first summarizes the thesis by reviewing the contents described in each chapter. Then, the research contributions extracted from the proposals and the experiments are pointed out. In addition, some interesting future research issues are commented in the future work section. Finally, the research framework in which this thesis was developped is described before the list of related publications.

9.1 Summary

This thesis addresses the development of a Mission Control System (MCS) to define and execute missions in the context of a control architecture for an Autonomous Underwater Vehicle (AUV). Chapter 2 presents an overview of control architectures and popular MCS implemented for representative underwater vehicles. Studied systems are divided between planning-based systems, in which deliberative modules plan the mission on-line, and systems based in predefined missions, where the mission plan is described off-line by a user. Predefined plans, that are the state of the art for nowadays AUVs, are then categorized in those which use a Domain Specific Language (DSL) to describe a mission plan and those using a formalism. After comparing both, it is shown that, formalism based systems present formal verification and execution supervision while DSLs tend to be more expressive and easy to use. Therefore, a compromise solution is chosen using a DSL that can be automatically translated into a formal Petri net. Chapter 3 introduces the vehicle experimental platforms and the control architecture, named

Component Oriented Layer-based Architecture for Autonomy (COLA2), used by these vehicles. The COLA2 has been developed along the realization of this thesis and although the proposed MCS has been developed as an independent module, it has adapted to it to carry out the experimental work. The procedure to define a mission using Petri nets is described in Chapter 4. First of all, a mission is compared with a Discrete Event System (DES) in which actions enable and disable vehicle primitives while these primitives generate, as a response, events that change the state of the DES. Then, the way in which vehicle primitives are modeled using Petri nets and are supervised by means of Petri net structures named *tasks* is described. To control the execution flow between tasks, another kind of Petri Net Building Block (PNBB) is defined: the *control structures*. Using the control structures, tasks can be executed sequentially, conditionally, iteratively, in parallel and always with the possibility to abort them. It is easy to combine tasks and control structures with each other to describe complex missions from simple structures. The main advantage of building a mission from small blocks is the fact of avoiding the verification of the final mission since it is already verified by construction. Instead of dealing directly with Petri nets, a DSL that automatically translates a high level imperative language into a Petri net has been designed and developed to simplify the user's work. This language named Mission Control Language (MCL) is presented in Chapter 5. Chapter 6 presents an insight of how constraints can be added to coordinate multiple vehicles. Three constraints are presented: mutual exclusions, orderings and synchronizations. The proposed method consists in the following steps: Program an individual mission for each entity; Define a set of constrains to enforce a coordinated behavior among the entities; Automatically synthesize the necessary Supervision Based on Place Invariants (SBPIs) to deal with the constraints; Connect each independent robot mission among them by means of the previously synthesized SBPIs generating a centralized Petri net ; Check that the centralized mission is deadlock free; And finally, partition the centralized Petri net mission into as many decentralized Petri nets as entities involved in the mission, keeping the same behavior than in the centralized net while minimizing the communication between the decentralized nets. Despite planning is out of the scope of this thesis, Chapter 7 introduces how to interface the proposed MCS with an automated planning system to enhance the deliberative capabilities. Two components are added for this purpose: a state-space planner and a world modeler. The world modeler keeps a knowl-

edge database up to date according with the perceptions received by the COLA2 reactive layer and a set of scripts, provided by the user, that transform these perceptions in facts. According to the facts in the knowledge database and the goals given by the user, an on-board planner automatically selects the sequence of planning operators to execute. These planning operators are predefined missions encoded in MCL that solve a particular phase of a more general mission. Then, the planner is not sequencing directly vehicle primitives but small predefined missions. Experimental results are reported in Chapter 8. Four experiments have been carried out, two of them with the experimental platforms Ictineu AUV and Sparus AUV while the other two have been executed in simulation. The first experiment shows an industrial application, in the context of a dam inspection, in which human intervention in required to adjust some of the mission parameters. The second experiment presents a completely autonomous survey in an area of scientific interest. The third experiment is intended to validate the coordination capabilities introduced in Chapter 6. An AUV and an Autonomous Surface Craft (ASC) have to collaborate to gather georeferenced images about some Object Of Interest (OOI). Finally, a pipe tracking inspection application is used to compare two different paradigms to program a mission: an off-line user predefined mission and an on-line automatically generated mission.

9.2 Contributions

This thesis work has accomplished the proposed goal of developing a system to define and execute missions for autonomous underwater vehicles, simple to use from the user point of view, and easily adaptable to different control architectures. In the development of this goal, various research contributions were achieved. These contributions are listed below:

- **Development of a control architecture for an AUV**: Although the control architecture COLA2 is organized using the typical three-layers model (reactive, execution and mission), the combination of elements that compound each layer is new. The reactive layer is subdivided in three modules, separating the vehicle interface drivers from the perception units and these from the components that control and guide the vehicle. Moreover, a coordinator to merge multiple behavior responses is also included. The exe-

cution layer implements a common DES but also includes an Architecture Abstraction Component (AAC) to simplify the interface of the execution and mission layers with the reactive one that is particular for each vehicle. Finally, the mission layer allows to define off-line missions using a high level language but also admits the use of a planner together with a world modeler.

- **Mission definition using Petri nets:** Our proposal depicts a method to define a set of building blocks using Petri nets and shows how these blocks have to be composed among them to create a larger structure. These PNBBs are conditioned to several constraints. On one hand, they have to share a common interface that determines the number of actions/events to be received/sent for each task. On the other hand, a reachability analysis has to be performed to ensure that each block evolve free of deadlocks from a valid initial state to a valid final state. Then, it is possible to describe a large structure by means of composing PNBBs in which the checked properties will hold.

- **The Mission Control Language:** A completely new language has been designed and implemented to allow the AUV users to easily define a mission. The MCL is a high level imperative language that automatically compiles the mission program into a formal Petri net, avoiding the tedious part of programming a mission using the graphical manipulation of a Petri net.

- **Coordination of multiple vehicles:** The decentralized supervision of Petri nets theory developed by Marian V. Iordache and Panos J. Antsaklis [Iordache and Antsaklis, 2006b] has been adapted to the underwater domain allowing the coordination of multiple vehicle missions while minimizing the communication between them. This is extremely important in underwater robotics since acoustic communications are known to have a low band width.

- **Deliberation and execution interface:** A new interface to connect an automated planning algorithm with an execution layer based on Petri nets has been proposed. A world modeler allows the use of planners that have not been designed to operate on-board an autonomous vehicle. Moreover, MCL missions have been used as planning operators to ensure a predictable behavior while simplifying the whole system.

9.3 Future Work

During the development of this research work, new problems and topics of interest for future investigation have arisen. The following points are the ones which have been found as the most logical lines to continue this research.

- **Visual programming interface:** The creation of a high-level language has simplified remarkably the mission programming for autonomous vehicles. However, many AUVs always perform similar missions in which the sequence of tasks vary relatively little. One example are the vehicles used to perform surveys to collect data. For such missions would be interesting to design a visual programming interface in which the user may define the path to carry out and the actions to be performed in each way-point graphically. In addition, the output of this interface could be a program coded in MCL that the user could modify to add elements that are hard to describe from a visual environment.

- **Enhance multiple vehicles coordination:** Multiple vehicle coordination has become a popular topic in the underwater domain. Thus, new constraints can be added to the ones presented in Chapter 6 to simplify the coordination of multiple vehicles. Moreover, the algorithm to check if the combination of coordination constraints produces a deadlocked mission, see Extract 6.1, analyzes the whole centralized mission. To check if a deadlock exists, since constraints have always a similar form, a more efficient algorithm could be found. Finally, new mechanisms to allow communication between vehicle primitives should be studied. Currently, there is only communication before or after the execution of a primitive but no during its execution. However, to perform multiple vehicle formations [Edwards et al., 2004] or cooperative path-following [Vanni et al., 2008] this feature will be necessary.

- **Extend deliberation capabilities:** The world modeler as well as the planner components presented in Chapter 7 are very simple. In fact, the aim of this chapter is only to show the interface between these components and the proposed MCS. The development of a more adequate on-board planner together with a proper world modeler, exploiting the benefits of a robust execution layer, would represent a new and interesting research line.

9.4 Research framework

The results and conclusions presented in this disertation have been possible after the realization of countless simulations and experiments. All the work done during the evolution of this thesis is summarized here with references to the most relevant research publications done by the author. The complete list of publications can be consulted in the next section.

At the beginning of this thesis in 2005, there was one research platform in the Centre d'Investigacio en Robotica Submarina (CIRS), the Garbi AUV. However, in January 2006 started the construction of a new vehicle, Ictineu AUV, to face the first edition of the Student Autonomous Underwater Challenge-Europe (SAUC-E). To confront this competition, not only a vehicle had to be built but also the Object Oriented Control Architecture for Autonomy (O2CA2) had to be remodeled and a MCS developed. In parallel with this competition, the CIRS was starting a series of experiments to automatically survey the wall of a dam. For these experiments it was also necessary to have a functional MCS. Two works presenting simulated results were published reporting a preliminary version of the proposed MCS [Palomeras et al., 2006a,b]. The first one faced the SAUC-E mission while the second one dealt with the dam inspection scenario. Once the Ictineu AUV was operative, this preliminary MCS was tested on it. The first expimental test was during the SAUC-E competition [Hernandez et al., 2006; Ribas et al., 2007] in which Ictineu AUV obtained the first position. To summarize the preliminary design of the MCS an article was published in the International Journal of Control [Carreras et al., 2007] where a scientific interest mission was programmed by means of Petri nets.

To simplify the way in which missions were programmed using Petri nets, a high level language was proposed [Palomeras et al., 2007a,b]. However, before implementing the proposed language some improvements were performed in the way in which Petri nets were used to encode the missions [Palomeras et al., 2008] in order to obtain more reliable missions. After applying these modifications, the MCL was redefined [Palomeras et al., 2009b], implemented and tested [Palomeras et al., 2009a] repeating the dam inspection experiments previously presented but in a real scenario instead of a simulated one. Once the MCS was completely implemented and working not only in Ictineu AUV but also in the new vehicle Sparus AUV, a joint work was done in collaboration with the Universidad Jaume

I (UJI). The work consisted of controlling an underwater vehicle with a robotic manipulator attached to it in the context of an intervention mission [Palomeras et al., 2010a] taking advantage of the AAC to control two different reactive layers from the proposed MCS.

Finally, to improve the MCS capabilities two additional works were carried out. The first one applies the decentralized supervision of Petri nets introduced by M. V. Iordache and P. J. Antsaklis [Iordache and Antsaklis, 2006b] to the multiple underwater vehicles coordination [Palomeras et al., 2010c]. The second one defines the interface of the proposed system with an on-board planner in order to enhance the deliberative system capabilities [Palomeras et al., 2010b].

In parallel to the thesis main line, additional publications have been done in the field of remote experimentation and telerobotics [Ridao et al., 2005, 2006, 2007] as well as collaborating with the laboratory projects for the development of an Intervention Autonomous Underwater Vehicle (IAUV) [Prats et al., Submitted; Ribas et al., 2011, Submitted]. Moreover, an article summarizing the COLA2 as well as the proposed MCS has been also submitted [Palomeras et al., submitted].

9.5 Related publications

P. Ridao, E. Batlle and N. Palomeras
First steps in Remote Experimentation with UUVs.
Workshop International en Telerobtica y Realidad Aumentada para Teleoperacin, 2005.

P. Ridao, E. Hernandez, N. Palomeras and M. Carreras
Remote Training in AUV Control Using HIL Simulators.
Manoeuvring and Control of Marine Craft, 2006.

N. Palomeras, M. Carreras, P. Ridao and E Hernandez
Mission control system for dam inspection with an AUV.
IEEE/RSJ International Conference on Intelligent Robots and Systems, 2006, pp. 2551-2556.

N. Palomeras, P. Ridao, M. Carreras and E. Hernandez
Design of a Mission Controller for an Autonomous Underwater Robot.
Workshop de Agentes Físicos, 2006, pp. 167-174.

E. Hernandez, P. Ridao, M. Carreras, D. Ribas, N. Palomeras, A. El-Fakdi and F. Chung
Ictineu AUV, un Robot per a Competir.
Congrs Catal d'Intelligncia Artificial, 2006.

D. Ribas, N. Palomeras, P. Ridao, M. Carreras and E. Hernandez
ICTINEU AUV Wins the First SAUC-E Competition.
IEEE International Conference on Robotics and Automation, 2007, pp. 151-156.

M. Carreras, N. Palomeras, P. Ridao and D. Ribas
Design of a mission control system for an AUV.
International Journal of Control, 2007, v. 80(7), pp. 993-1007.

N. Palomeras, P. Ridao and M. Carreras
Defining a Mission Control Language.
Congrs Internacional sobre Tecnologia Marina, 2007.

N. Palomeras, P. Ridao, M. Carreras and J. Batlle
MCL: A Mission Control Language for AUVs.
Control Applications in Marine Systems, 2007.

P. Ridao, M. Carreras, E. Hernandez and N. Palomeras
Underwater Telerobotics for Collaborative Research.
Advances in Telerobotics, 2007. Ed. M. Ferre, M. Buss, R. Aracil, C. Melchiorri and C. Balaguer.

N. Palomeras, P. Ridao, M. Carreras and C. Silvestre
Towards a Mission Control Language for AUVs.
17th IFAC World Congress, 2008, pp. 15028-15033.

N. Palomeras, J. C. Garcia, M. Prats, J. J. Fernandez, P. J. Sanz and P. Ridao
A Distributed Architecture for Enabling Autonomous Underwater Intervention Missions.
IEEE Systems Conference, 2010, pp. 159-164.

N. Palomeras, P. Ridao, M. Carreras and C. Silvestre
Mission Control System for an Autonomous Vehicle: Application Study of

a Dam inspection using an AUV.
8th IFAC International Conference on Manoeuvring and Control of Marine Craft, 2009.

N. Palomeras, P. Ridao, M. Carreras and C. Silvestre
Using Petri nets to specify and execute missions for Autonomous Underwater Vehicles.
IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009.

N. Palomeras, P. Ridao, C. Silvestre and A. El-fakdi
Multiple vehicles mission coordination using Petri nets.
IEEE International Conference on Robotics and Automation, 2010, pp. 3531-3536.

N. Palomeras, P. Ridao, M. Carreras and C. Silvestre
Towards a Deliberative Mission Control System for an AUV.
7th IFAC Symposium on Intelligent Autonomous Vehicles, 2010.

N. Palomeras, A. El-Fakdi, M. Carreras and P. Ridao
COLA2: A control architecture for AUVs.
Journal of Oceanic Enginering, submitted.

D. Ribas, P. Ridao, LL. Magí, N. Palomeras and M. Carreras
The Girona 500, a multipurpose autonomous underwater vehicle.
Proceedings of the Oceans IEEE, 2011.

M. Prats, D. Ribas, N. Palomeras, J. C. García, V. Nannen, J. J. Fernández, J. P. Beltrán, R. Campos, P. Ridao, P. J. Sanz, G. Oliver, M. Carreras, N. Gracias, R. Marín and A. Ortiz
Reconfigurable AUV for Intervention Missions: A Case Study on Underwater Object Recovery.
Journal of Intelligent Service Robotics, submitted.

D. Ribas, N. Palomeras, P. Ridao, M. Carreras and A. Mallios
Girona 500 AUV, from survey to intervention.
Transactions on Mechatronics, submitted.

Appendix A

An Introduction to Petri Nets

Petri nets were invented in 1962 by Carl Adam Petri in his PhD thesis [Petri, 1962]. They are one of several mathematical representations of discrete distributed systems. While finite automates can only represent regular languages, Petri nets are able to describe regular but also non regular languages. Its algebraical representation is simple and thus makes its specification and analysis simpler. There are several analysis techniques based on their structural and behavioral properties to detect and prevent anomalies and errors.

Table A.1 shows a formal definition of a Petri net [Murata, 1989]. It consists of a particular class of directed graph composed of a finite set of places P, transitions T and directed arcs A, a weight function W and an initial state called *initial marking* μ_0. Arcs link places to transitions and transitions to places but never places nor transitions between themselves. Every place can accommodate zero, one or more tokens. If it is assumed that each place can accommodate an unlimited number of tokens, such Petri nets are referred to as infinite capacity nets. On the other hand, for a finite capacity net (N, μ_0), each place p has an associated capacity $K(p)$ that is the maximum number of tokens that p can hold at any time. The operator $\mu(p)$ returns the number of tokens in place p. The dynamic behavior of a system represented by a Petri net is changed according to the following transition rule:

1. A transition t is said to be *enabled* if each input place p of t ($\bullet t$, see Table A.2) is marked with at least $w(p,t)$ tokens, where $w(p,t)$ is the weight

A petri net is a 5-tuple, $PN = (P, T, F, W, \mu_0)$ where:

$P = \{p_1, p_2, ..., p_n\}$ is a finite set of places,
$T = \{t_1, t_2, ..., t_m\}$ is a finite set of transitions,
$A \subseteq (PxT) \cup (TxP)$ is a set of arcs (flow relation),
$W : A \to \{1, 2, 3, ...\}$ is a weight function,
$\mu_0 : P \to \{0, 1, 2, 3, ...\}$ is the initial marking,
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

A Petri net structure $N = (P, T, A, W)$ without any specific initial marking is denoted by N.

A Petri net with the given initial marking is denoted by (N, μ_0).

Table A.1: Formal definition of a Petri net.

$\bullet t = \{p|(p, t) \in F\}$ = the set of input places of t
$t\bullet = \{p|(t, p) \in F\}$ = the set of output places of t
$\bullet p = \{t|(t, p) \in F\}$ = the set of input transitions of p
$p\bullet = \{t|(p, t) \in F\}$ = the set of output transitions of p
where F is the set of all arcs.

Table A.2: Petri net pre-set and post-set.

of the arc from p to t.

2. An enabled transition may or may not fire. It is possible to differentiate between immediate and non-immediate transitions. Immediate transitions fire as soon as they are enabled while non-immediate transitions fire only if they are enabled and an extra process, generally represented by another Petri, net reaches an specific state. Then, it is assumed that: if an immediate and a non-immediate transition are enabled, the immediate transition will be always the first to fire.

3. A firing of an enabled transition t removes $w(p,t)$ tokens from each input place p of t, and adds $w(t,p)$ tokens to each output place p of t ($t\bullet$, see Table A.2), where $w(t,p)$ is the weight of the arc from t to p.

A Petri net is said to contain a self-loop if exists a place p which is both an input and output place of a certain transition t. A Petri net without self-loops is called a *pure* Petri net. A Petri net is said to be *ordinary* if all of its arc weights are 1's.

Figure A.1(a) shows a Petri net with four places ($P0$, $P1$, $P2$ and $P3$) and four transition ($T0$, $T1$, $T2$ and $T3$). $T0$ and $T2$ are immediate transitions while $T1$ and $T3$ are non-immediate transitions. Places and transitions are connected by arcs as shown in the figure. If the initial marking (μ_0) is $\{1, 0\ 0\ 0\}$ then $T0$ is the only enabled transition. When $T0$ fires, according to the transition rule, $P0$ will lose one token while one token will be added to $P1$ and $P2$ as shown in Figure A.1(b). In Figure A.1(b) transitions $T1$ and $T2$ are enabled, however, as $T1$ is a non-immediate transition, $T2$ is the transition that will fire next and Figure A.1(c) will be obtained. Here, both enabled transitions ($T1$ and $T3$) are non-immediate. Thus, either the first or the latter may fire.

A.1 Properties

One of the main reasons for modeling systems with Petri nets is their support for the analysis of many properties associated with problems in concurrent systems. Two types of properties can be studied with a Petri net model: (1)*Behavioral Properties*, also called *marking-dependent*, which depend on the initial marking, and (2)*Structural Properties* which are independent of the initial marking. Three

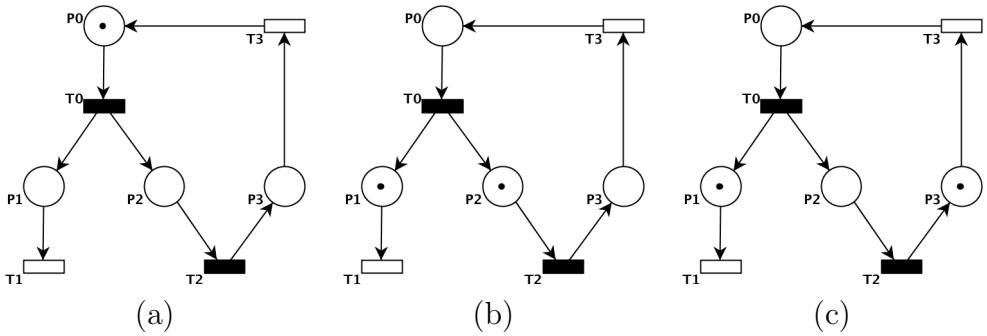


Figure A.1: (a) Petri net example in its initial state (μ_0). (b) Petri net example after firing T_0. (c) Petri net example after firing T_0 and then T_1 and T_2.

of the most important behavioral properties and one structural property are detailed next:

- **Reachability**: The firing of an enabled transition will change the token distribution, the marking, in a net according to the transition rule. A sequence of firings will result in a sequence of markings. A marking μ_n, is said to be reachable from a marking μ_0 if there is a sequence of firings that transforms μ_0 to μ_n. The set of all possible markings reachable from μ_0 in a net (N, μ_0) is denoted by $R(\mu_0)$. The set of all possible firing sequences from μ_0 in a net (N, μ_0) is denoted by $L(\mu_0)$. For example, the Petri net shown in Figure A.1 can reach an infinite number of different states, however, they can be reduced to only seven: {1, 0, 0, 0} (Figure A.1(a)), {0, 1, 1, 0}(Figure A.1(b)), {0, 1, 0, 1} (Figure A.1(c)), {0, 0, 0, 1}, {1, ω, 0, 0}, {0, ω, 1, 0} and {0, ω, 0, 1}, where ω represents any value bigger than 1.

- **Boundedness**: A Petri net (N, μ_0) is said to be k-bounded or simply bounded if the number of tokens in each place p does not exceed a finite number k for any marking reachable from μ_0, i.e. $\mu(p) \leq k$ for every place p and every marking $\mu \in R(\mu_0)$. Figure A.1 shows a Petri net with places $P0$, $P2$ and $P3$ 1-bounded but with place $P1$ unbounded. A Petri net (N, μ_0) is said to be safe if it is 1-bounded.

- **Liveness**: A Petri net (N, μ_0) is said to be live if, no matter what marking has been reached from μ_0, it is possible to ultimately fire any transition of the net by progressing through some further firing sequence. This means

that a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen. Thus, the Petri net in Figure A.1 is live.

- **Invariants**: Net invariants are one of the structural properties of Petri nets. Place invariants are sets of places whose weighted token count remains constant for all possible markings. They are represented by a *n-dimensional* integer vector x, where n is the number of places of the Petri net; non-zero entries correspond to the places that belong to the particular invariant. Then, $x\mu = x\mu_0$ for μ representing any subsequent marking of μ_0. The place invariants of a net can be computed by finding integer solutions to $xD = 0$ where x is the place invariant vector and D the incidence matrix, see Section A.2. One place invariant can be found in Figure A.1, it is represented by $\mu(P0) + \mu(P2) + \mu(P3) = 1$.

A.2 Analysis

Methods of analysis for Petri nets may be classified into three groups: i) the coverability (reachability) tree method, ii) the matrix-equation approach, and iii) reduction or decomposition techniques [Murata, 1989]. The first method involves essentially the enumeration of all the reachable markings of a Petri net ($R(\mu_0)$). Although it is applicable to all sorts of nets, in practice it may only be applied to small nets due to the complexity of the state space explosion. On the other hand, matrix equations and reduction techniques are powerful but in most cases they are only applicable to special subclasses of Petri nets or particular situations.

A.2.1 The coverability tree

The coverability tree is an analysis technique based on the construction of a tree where nodes are Petri net states and arcs represent transitions firings. Considering any possible Petri net, a tree whose root node is the initial state of this Petri net (μ_0) is builded. Then, it is possible to examine all transitions that can fire from this state, defining new nodes in the tree, and repeat this process until all possible reachable states ($R(\mu_0)$) are identified. The reachability tree is conceptually easy to construct, however, it may be infinite. Therefore, it is necessary to seek a

finite representation of this tree. This is possible, but at the expense of losing some information. The finite version of an infinite reachability tree will be called a reachability graph and can be computed using the algorithm shown in Extract A.1 [Cassandras and Lafortune, 2007] where:

- x_0 is the *root node* and corresponds to the initial state of the given Petri net (μ_0).
- A *terminal node* is a node from which no transitions can fire.
- A *duplicated node* is a node that is identical to a node already in the tree.
- $>_d$ denotes *node dominance*. If $x = [x(p_1), ..., x(p_n)]$ and $y = [y(p_1), ..., y(p_n)]$ are two states, x dominates y ($x >_d y$) if the following two conditions hold: $x(p_i) \geq y(p_i)$, for all $i = 1, ..., n$ and $x(p_i) > y(p_i)$, for at least some $i = 1, ..., n$.

Extract A.1: Compute reachability graph.

initialize $x = x_0$;
for *each new node* x **do**
  evaluate the transition function $f(x, t_j)$ for all $t_j \in T$;
  if $f(x, t_j)$ *is undefined for all* $t_j \in T$ **then**
    mark x as a terminal node ;
  else
    create a new node x' $= f(x, t_j)$;
    if $x(p_i) = \omega$ *for some* p_i **then**
      set x'$(p_i) = \omega$;
    else if *exists a node* y *in the path from the root node* x_0 *to* x *such that* $x' >_d y$ **then**
      set x'$(p_i) = \omega$ for all p_i such that x'$(p_i) > y(p_i)$;
    else
      x'(p_i) is as obtained in $f(x, t_j)$;
    end
  end
end
If all new nodes have been marked as either terminal or duplicate nodes, then stop ;

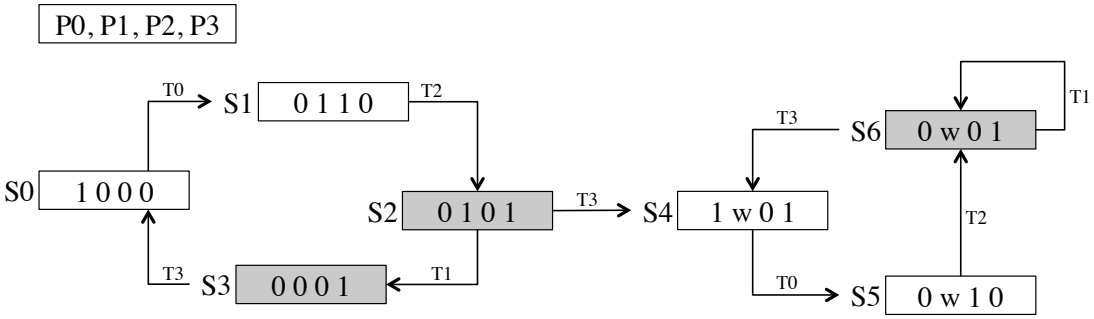


Figure A.2: Reachability analysis for the Petri net in Figure A.1 where S0, S1, S4 and S5 are vanishing states while S2, S3 and S6 are tangible states.

If Algorithm A.1 is applied to the Petri net in Figure A.1 the reachability graph shown in Figure A.2 is obtained. When the reachability graph of a Petri net with immediate and non-immediate transitions is computed, the resulting states can be *vanishing* or *tangible*. A state is called tangible if it does not possesses any outgoing immediate transition. Otherwise, the state is called vanishing.

A.2.2 The matrix equation approach

To introduce the matrix-equation approach some terms must be defined. Let $\mathbb{Z}$ be the set of integers, n be the number of places in a Petri net and m be the number of transitions. The arcs connecting transitions to places are described by the matrix $D^+ \in \mathbb{Z}^{n \times m}$ and the arcs connecting places to transitions are described by $D^- \in \mathbb{Z}^{n \times m}$, where the $i_{n \times m}$ element of matrix D^+ has a zero if there are no arcs connecting transition m with place n or the weight of the arc $(w(m, n))$ otherwise and the $i_{n,m}$ element of matrix D^- is zero if there are no arcs connecting place n with transition m or the weight of the arc $(w(n, m))$ otherwise. Note that the elements of D^+ and D^- are greater than or equal to zero. The marking of a Petri net is represented by a n dimensional integer vector μ whose k^{th} element is the number of tokens in the place p_n $(\mu(p_n))$. The transitions of a Petri net fire in discrete steps. The transitions that have to fire at the current step are represented by the m dimensional integer vector q. The j^{th} element of q is 0 if the j^{th} transition will not be fired and 1 if it will fire. A given firing vector q represents a valid possible firing if all of the transitions for which it

contains nonzero entries are enabled[1]. The validity of a given firing vector q can be determined by checking the *enabling condition*:

$$\mu \geq D^{-}q \tag{A.1}$$

Where all vector and matrix inequalities are read element-by-element with respect to the two sides of the inequality. The Petri net incidence matrix is defined as

$$D = D^{+} - D^{-} \tag{A.2}$$

Then, it is possible to work out the matrices D^{+}, D^{-} and D for the Petri net shown in Figure A.1:

$$D^{+} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$D^{-} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

When a Petri net contains no self loops, the D matrix is uniquely defined by D^{+} and D^{-} and then, we can use the following inequality as an equivalent to the enabling condition to (A.1).

[1]It is worth noting that for a transition to be fired, first it has to be enabled.

$$\begin{aligned} \mu + D^+ q &\geq D^- q \\ \mu + (D^+ - D^-) q &\geq 0 \\ \mu + Dq &\geq 0 \end{aligned} \tag{A.3}$$

When using (A.1) or (A.3) care must be taken when q indicates the concurrent firing of multiple transitions. Concurrency may not be allowed at all, in which case q would be a zero vector with a single element equal to one. Concurrency may be allowed only when each of the indicated transition firings could occur one after the other in any order.

When the transitions described by the q vector fires, the state of the Petri net changes. The state change is described by

$$\mu \leftarrow \mu + Dq \tag{A.4}$$

Thus, the Petri net transition rule previously described can be exposed by the following system:

$$\mu_{k+1} = \mu_k + Dq_k \tag{A.5}$$

$$D \in \mathbb{Z}^{n \times m}, \mu \in \mathbb{Z}^n, q \in \mathbb{Z}^m, (\mu, q \geq 0) \tag{A.6}$$

Without (A.6), system (A.5) is a rather uninteresting subclass of linear difference equations, but with (A.6), it becomes something entirely different, requiring different tools and concepts for analysis.

A.3 Siphons and Traps

A trap or a siphons is a set of Petri net places. In a trap, when one of the places becomes marked, the trap will always be marked for all future reachable markings. Similarly, once the marking of a siphon becomes empty, the siphon will remain empty. If $\bullet p$ is the set of input transitions into the place p and $p\bullet$ is the set of output transition from the place p as shown in Table A.2, then, this same notation can be used with sets of places. Let S be a set of places, then $\bullet S$ and $S\bullet$ refer to the set of input and output transitions for the entire set S. Thus, a set of places S is a siphon if and only if

$$\bullet S \subseteq S\bullet \tag{A.7}$$

In the same way, a non empty set of places S is a trap if it accomplishes

$$S\bullet \subseteq \bullet S \tag{A.8}$$

The set $S = \{P0,\ P2,\ P3\ \}$ in the Petri net shown in Figure A.1 is both a siphon and a trap because $\bullet S = S\bullet$.

A.4 Invariants Based Control

Structural invariants are important means not only for analyzing Petri nets, since they allow for the net's structure to be investigated independently of any dynamic process, but also to control the behavior of a Petri net. It is possible to supervise a Petri net in order to enforce an invariant with some of their places by means of the Supervision Based on Place Invariant (SBPI). Invariant based control relies with linear state constraints. A linear state constraint is denoted by

$$l\mu \leq b \tag{A.9}$$

where l is an integer weight vector and b is an scalar. These constraints can be interpreted as: the sum of the tokens obtained by multiplying the l vector by the current number of tokens in each place (μ) has to be always less or equal than b. In order to add one of this linear state constraint to a Petri net represented by

the incidence matrix D_p, an extra place must be added. Following equations are used for this endeavor [Moody and Antsaklis, 1998]:

$$D_c = -lD_p \tag{A.10}$$

$$\mu_{c0} = b - l\mu_0 \tag{A.11}$$

where D_c describes the Petri net controller and μ_{c0} is its initial marking. Then, the incidence matrix of the resulting Petri net (D) and its initial state (μ_0) can be obtained as:

$$D = \begin{bmatrix} D_p \\ D_c \end{bmatrix} \tag{A.12}$$

$$\mu_0 = \begin{bmatrix} \mu_{p0} \\ \mu_{c0} \end{bmatrix} \tag{A.13}$$

Not only linear marking constraints can be used to control a Petri net. It is possible to restrict the Petri net behavior enforcing constraints related with its firing vector or the Parikh vector. For a further dissertation about this topic, the reader is referred to Iordache and Antsaklis [2002].

A.5 Subclasses of Petri nets

In order to describe the Petri net subclasses three concepts have to be introduced: conflict, concurrency and confusion. Figure A.3(a) shows a Petri net where two transitions are marked but only one of them can be fired. This structure is called a conflict, a choice or a decision. Figure A.3(b) presents an example of deterministic parallel activities where two places are simultaneously marked. Finally, Figure A.3(c) and (d) present a confusion situation where conflict and concurrency are mixed. As can be seen in Figure A.3(c) a symmetric confusion appears when $T0$ and $T2$ are concurrent but in conflict with $T1$. This means that a choice between fire $T0$ and $T2$ or fire $T1$ have to be done. In Figure A.3(d) an asymmetric confusion case is shown, where $T0$ is concurrent with $T1$ but will be in conflict with $T2$ if $T1$ fires before $T2$.

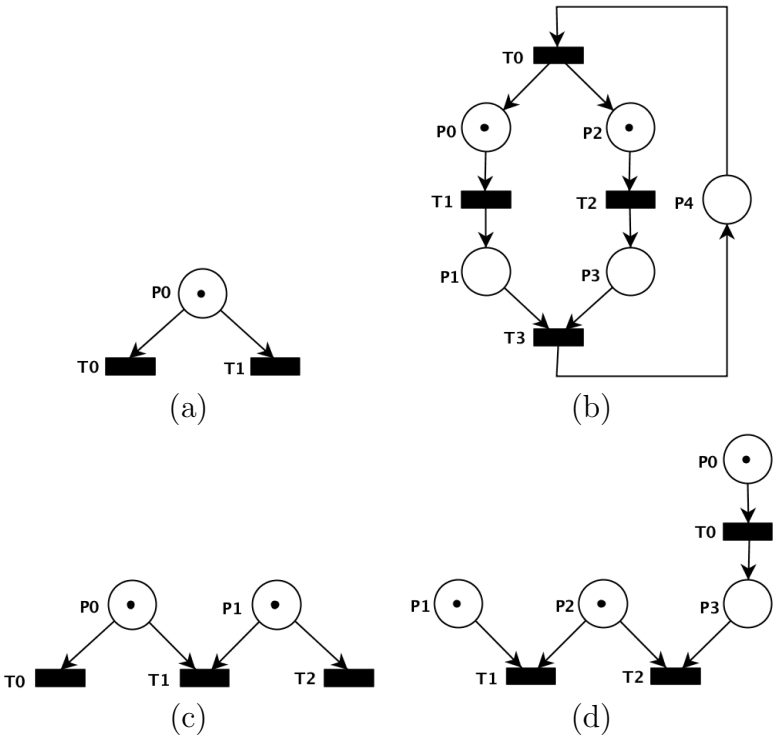


Figure A.3: (a) Conflicting Petri net structure. It exhibits non-determinism. (b) A Petri net representing deterministic parallel activities. (c) Symmetric confusion: $T0$ and $T2$ are concurrent but in conflict with $T1$. (d) Asymmetric confusion: $T0$ is concurrent with $T1$ but in conflict with $T2$ if $T1$ fires before $T2$.

Petri nets are classified based on its capabilities to support concurrency, conflict and confusion as shown in Figure A.4 and Figure A.5). Six main types of Petri nets are explained below.

A.5.1 State Machine

A State Machine (SM) is an ordinary Petri net with each transition t having exactly one input place and exactly one output place. This means that there can not be concurrency but there can be conflict, several transitions can be fired at the same time.

$$|\bullet t| = |t \bullet| = 1, \ \forall t \in T$$

A.5.2 Marked Graph

A Marked Graph (MG) is an ordinary Petri net with each place p having exactly one input transition and one output transition. This means that there can not be conflict but allows concurrency.

$$|\bullet p| = |p \bullet| = 1, \ \forall p \in P$$

A.5.3 Free-Choice

A Free-Choice (FC) net is an ordinary Petri net in which every outgoing arc from a place is either the unique outgoing arc or if there exist several outgoing arcs from this place, then every one of them must be the unique incoming arc to its corresponding transition. There can be both concurrency and conflict, but never at the same time.

$$\forall p \in P, \ |\bullet p| \leq 1 \text{ or } \bullet(p\bullet) = \{p\}; \text{ equivalently,}$$
$$\forall p_1, p_2 \in P, p_1 \bullet \cap p_2 \bullet \neq \emptyset \Rightarrow |p_1 \bullet| = |p_2 \bullet| = 1.$$

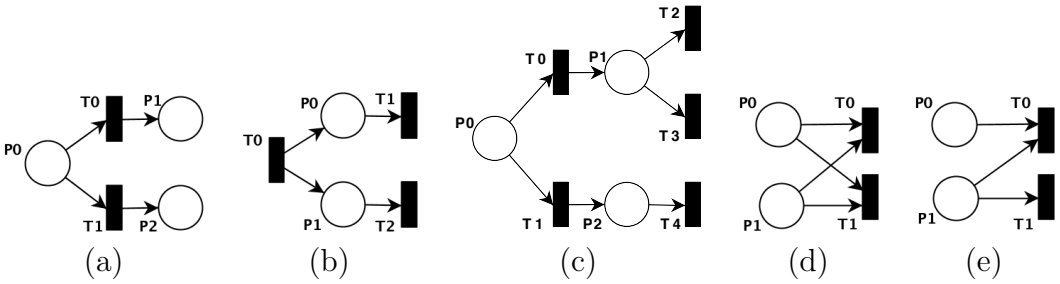


Figure A.4: Petri net subclasses: (a) State Machine (b) Marked Graph (c) Free-Choice (d) Extended Free-Choice (e) Asymmetric Choice.

A.5.4 Extended Free-Choice

An extended Extended Free-Choice (EFC) net is a FC Petri net where for every transition t with more than one incoming arc from places $p_1, ..., p_n$ the set of outgoing arcs of these places reach the same set of transitions.

$$p_1 \bullet \cap p_2 \bullet \neq \emptyset \Rightarrow p_1 \bullet = p_2 \bullet \text{ for all } p_1, p_2 \in P.$$

A.5.5 Asymmetric Choice

An Asymmetric Choice (AC) net, also known as a simple net, is an ordinary Petri net in that, concurrency and conflict are allowed, but not asymmetrically.

$$p_1 \bullet \cap p_2 \bullet \neq \emptyset \Rightarrow p_1 \bullet \subseteq p_2 \bullet \text{ or } p_1 \bullet \supseteq p_2 \bullet \text{ for all } p_1, p_2 \in P.$$

A.5.6 Ordinary Petri nets

In ordinary Petri nets confusion is allowed symmetrically and asymmetrically.

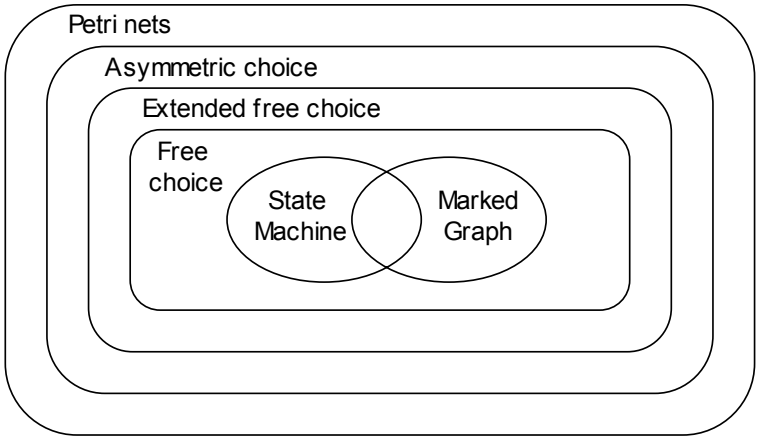


Figure A.5: Hierarchy between Petri net subclasses.

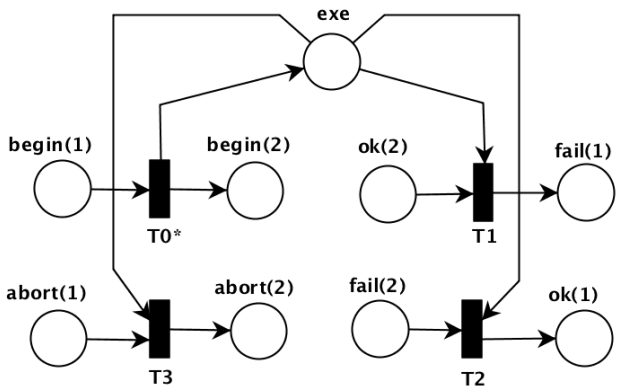
Appendix B

Control structures

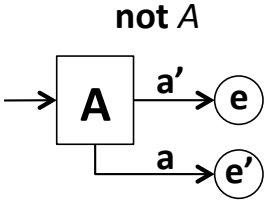
This appendix develops the control structures introduced in Chapter 4. Ten Petri Net Building Block (PNBB) control structures are shown: not, sequence, if-then, if-then-else, while-do, try-catch, parallel-and, parallel-or, monitor-condition-do and monitor-while_condition-do. For each control structure, its complete Petri net, a schematic representation and its instantiation when programing a mission using Mission Control Language (MCL) are shown. See that all PNBBs use the same interface: $I_i = \{ begin, abort, ok, fail \}$.

Table B.1: Not.

id:	Not
internal interfaces:	I_2
MCL:	not(A)



(a)

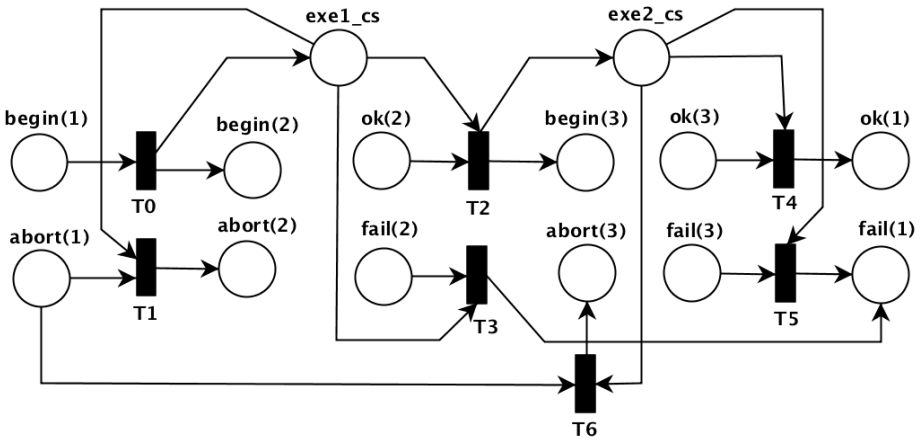


(b)

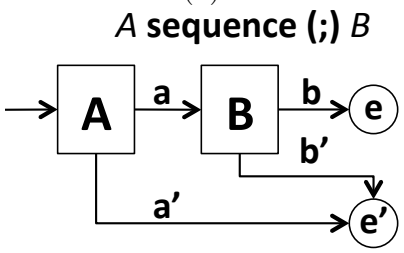
Figure B.1: (a) Petri net and (b) schematic for the control structure not.

Table B.2: Sequence.

id:	Sequence
internal interfaces:	I_2, I_3
MCL:	A ; B



(a)

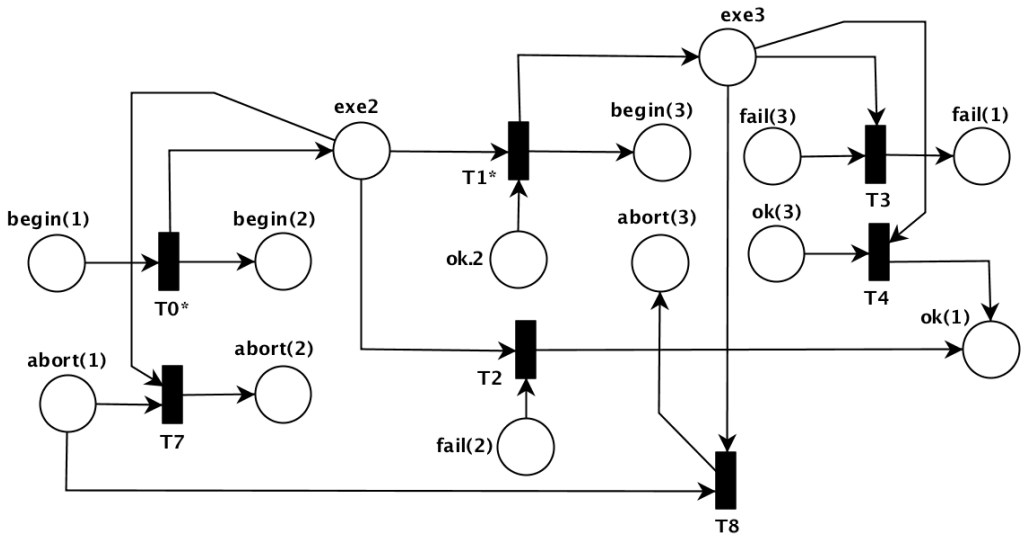


(b)

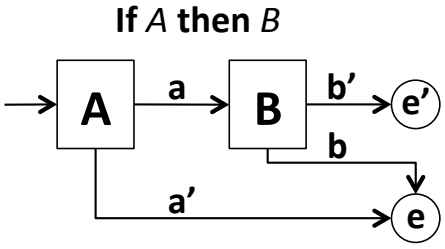
Figure B.2: (a) Petri net and (b) schematic for the control structure sequence.

Table B.3: If-Then.

id:	If-Then
internal interfaces:	I_2, I_3
MCL:	if(A) { B }



(a)

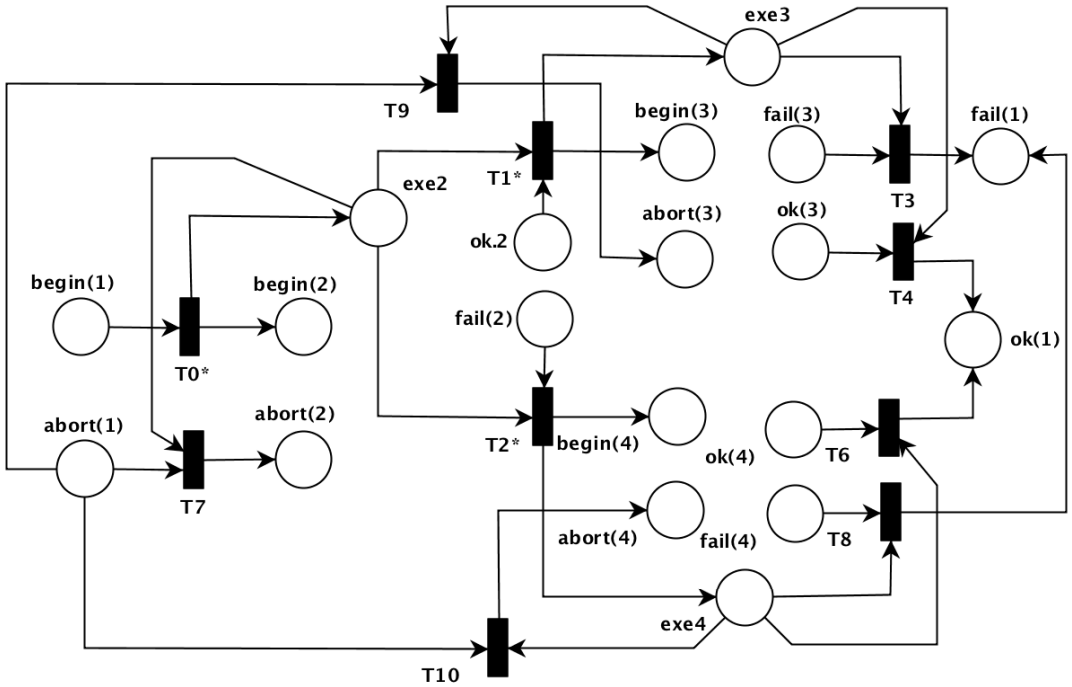


(b)

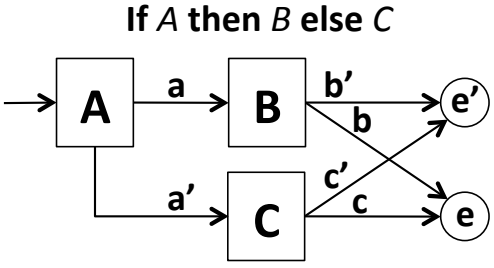
Figure B.3: (a) Petri net and (b) schematic for the control structure if-then.

Table B.4: If-Then-Else.

id:	If-Then-Else
internal interfaces:	I_2, I_3, I_4
MCL:	if(A) { B } then { C }



(a)

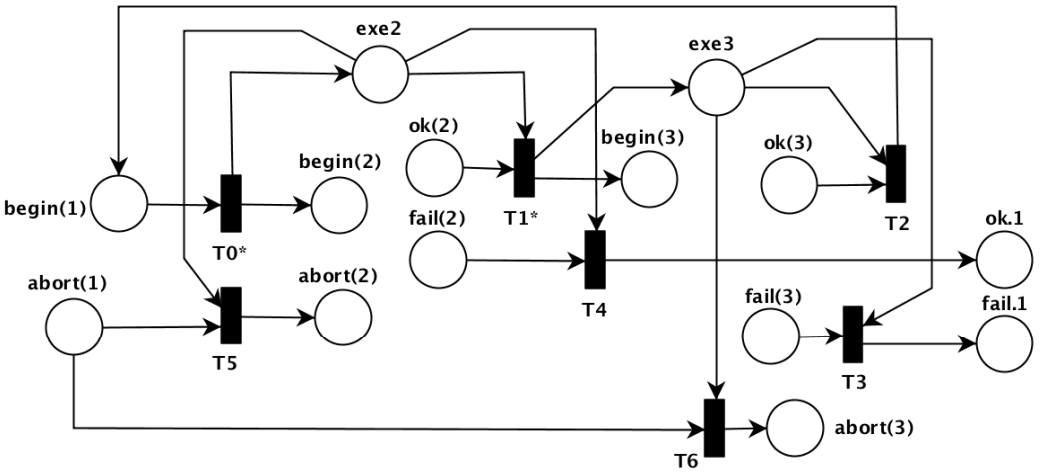


(b)

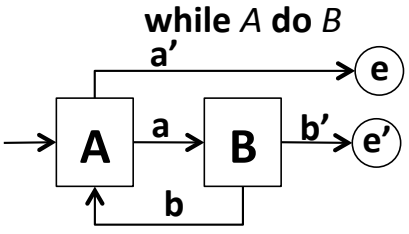
Figure B.4: (a) Petri net and (b) schematic for the control structure if-then-else.

Table B.5: While-Do.

id:	While-Do
internal interfaces:	I_2, I_3
MCL:	while(A) { B }



(a)

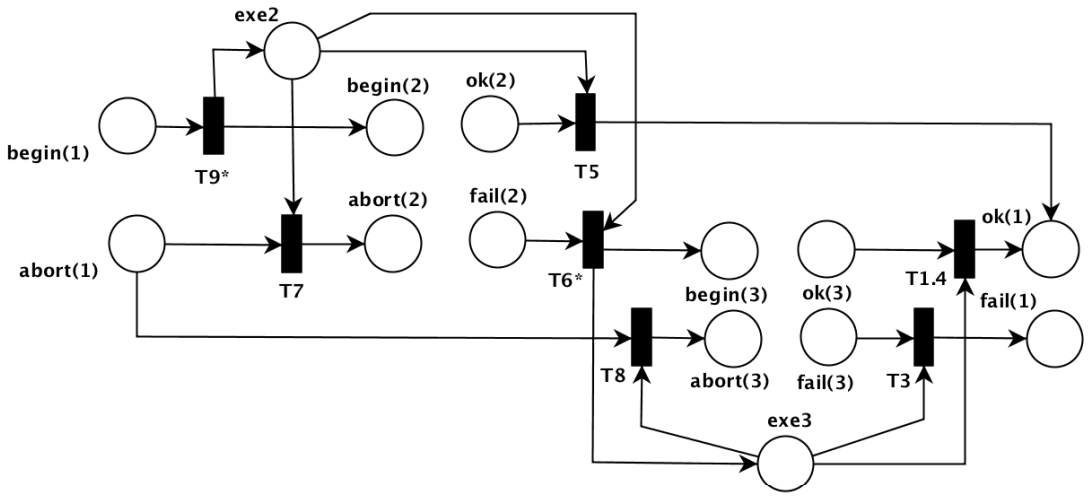


(b)

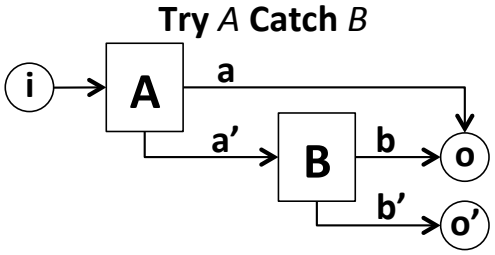
Figure B.5: (a) Petri net and (b) schematic for the control structure if-then.

Table B.6: Try-Catch.

id:	Try-Catch
internal interfaces:	I_2, I_3
MCL:	try{ A } catch{ B }



(a)

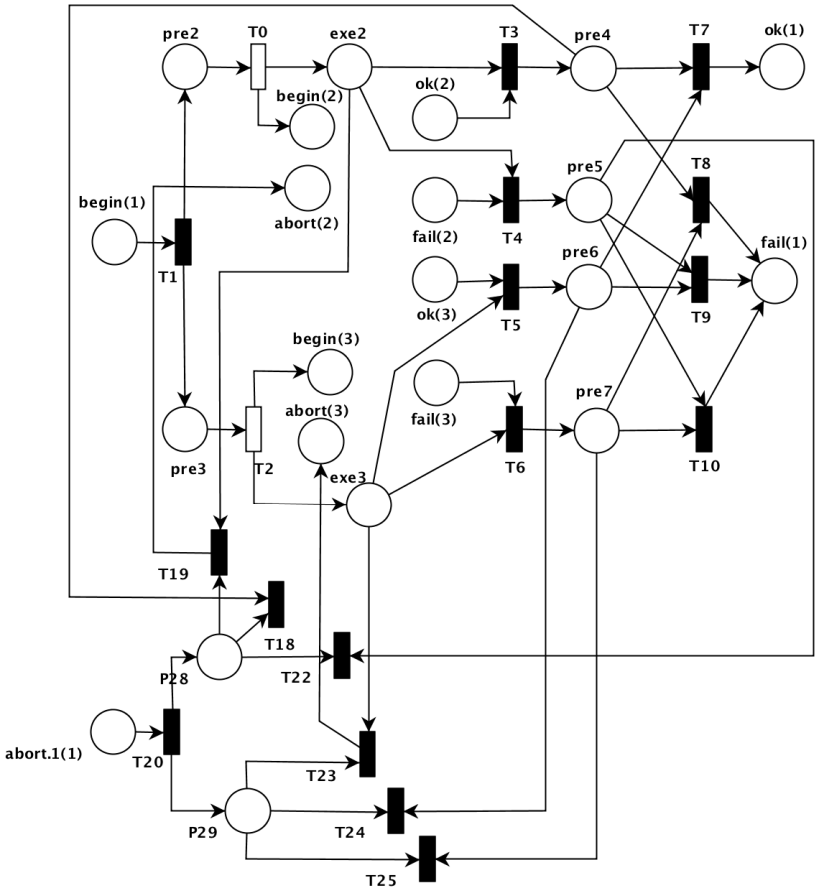


(b)

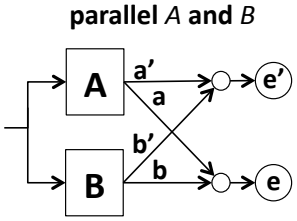
Figure B.6: (a) Petri net and (b) schematic for the control structure try-catch.

Table B.7: Parallel-And.

id:	Parllel-And
internal interfaces:	I_2, I_3
MCL:	parallel{ A } and{ B } and{ C} ...



(a)

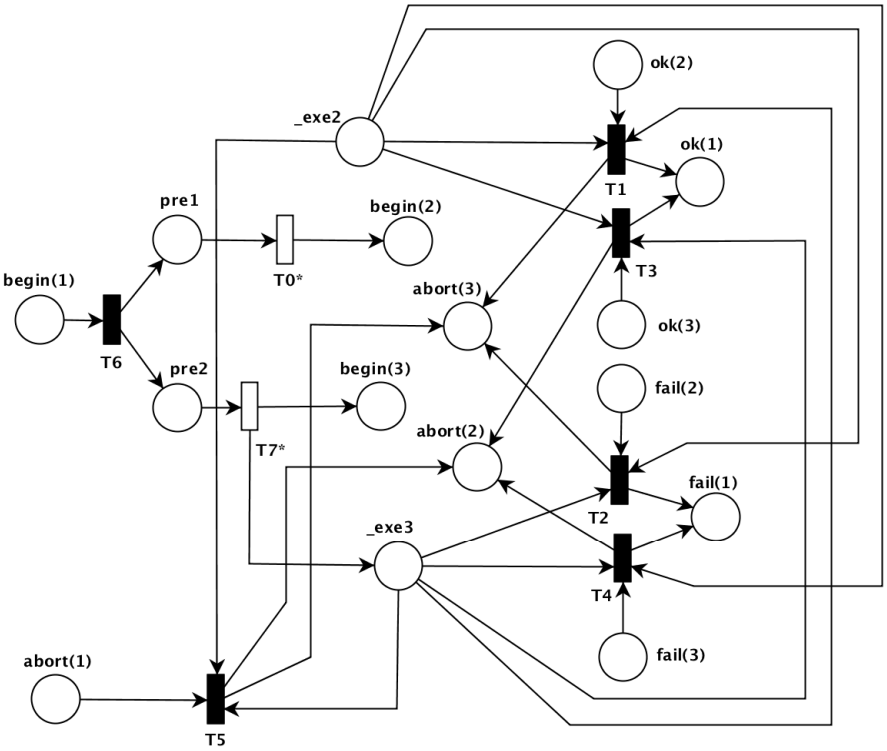


(b)

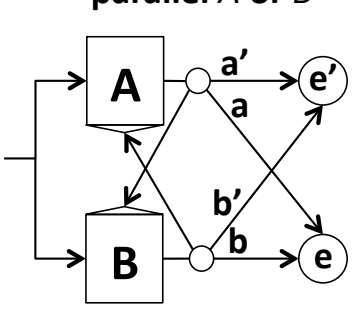
Figure B.7: (a) Petri net and (b) schematic for the control structure parallel-and.

Table B.8: Parallel-Or.

id:	Parllel-Or
internal interfaces:	I_2, I_3
MCL:	parallel{ A } or{ B } or{ C} ...



(a)

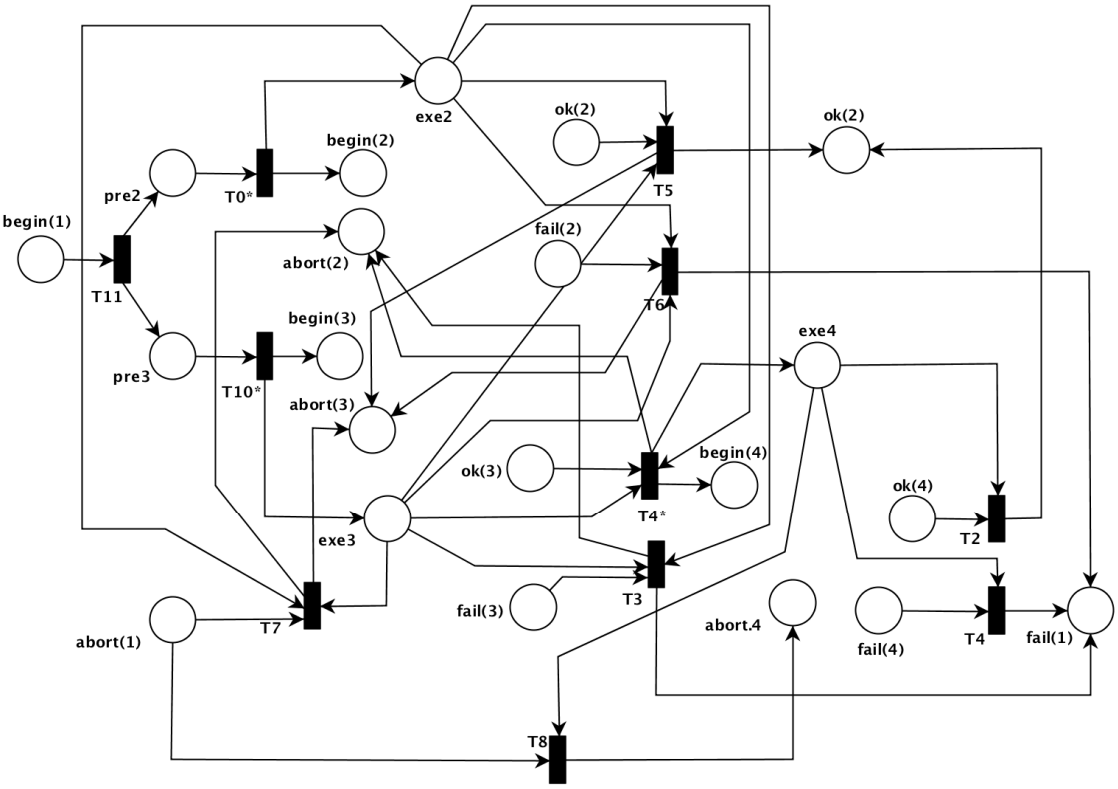


(b)

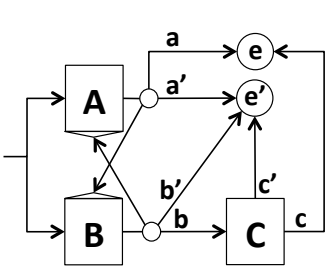
Figure B.8: (a) Petri net and (b) schematic for the control structure parallel-or.

Table B.9: Monitor-Condition-Do.

id:	Monitor-Condition-Do
internal interfaces:	I_2, I_3, I_4
MCL:	monitor{ A } condition(B) do{ C} ...



(a)

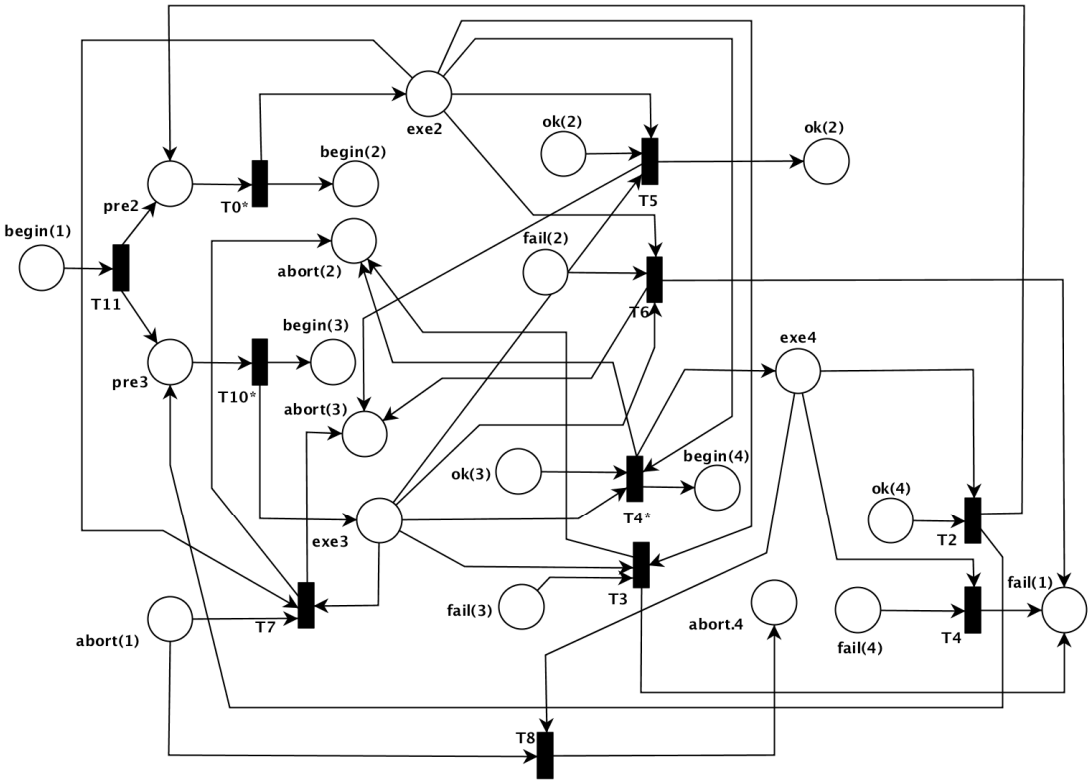


(b)

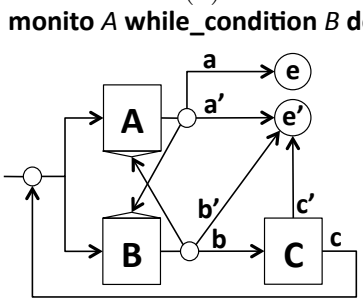
Figure B.9: (a) Petri net and (b) schematic for the control structure monitor-condition-do.

Table B.10: Monitor-While_Condition-Do.

id:	Monitor-While_Condition-Do
internal interfaces:	I_2, I_3, I_4
MCL:	monitor{ A } while_condition(B) do{ C} ...



(a)



(b)

Figure B.10: (a) Petri net and (b) schematic for the control structure monitor-while_condition-do.

Appendix C

Mission Control Language grammar

This appendix contains the context-free grammar for the Mission Control Language (MCL) introduced in Chapter 5. It is described with a variation of the standard Backus Normal Form (BNF) notation technique used by the ANTLR [Parr, 2010] software tool.

⟨*global*⟩ → ⟨*actionsBlock*⟩ ⟨*eventsBlock*⟩ ⟨*patternBlock*⟩ ⟨*tasksBlock*⟩ (⟨*missionBlock*⟩)+ (⟨*constraintsBlock*⟩)? EOF

⟨*actionsBlock*⟩ → **'actions'** **'{'** (⟨*actionsDef*⟩)* **'}'**

⟨*actionsDef*⟩ → ID **'='** ⟨*actionPrimitive*⟩ **';'**

⟨*actionPrimitive*⟩ → ID **'('** ⟨*command*⟩ ⟨*actionVarList*⟩ **')'**

⟨*command*⟩ → **'c'** **':'** ID

⟨*actionVarList*⟩ → (**','** ⟨*actionVar*⟩)*

⟨*actionVar*⟩ → (**'v'** **':'** (ID | NUM))

⟨*eventsBlock*⟩ → **'events'** **'{'** (⟨*eventsDef*⟩)* **'}'**

⟨*eventsDef*⟩ → ID **';'**

⟨*patternBlock*⟩ → **'pattern'** **'{'** (⟨*pattern*⟩)* **'}'**

⟨*pattern*⟩ → P_ID (**':'** ID **'.xml'** **';'** | **'{'** ⟨*patternDef*⟩ **'}'**)

⟨*patternDef*⟩ → ⟨*patternPlaces*⟩ ⟨*patternTrans*⟩ ⟨*patternArcs*⟩

⟨*patternPlaces*⟩ → **'places'** **'{'** (⟨*patternPlacesDefs*⟩)+ **'}'**

⟨*patternPlacesDefs*⟩ → ID (**'.'** NUM)? (**'('** NUM **')'**)? **';'**

⟨*patternTrans*⟩ → **'transitions'** **'{'** (⟨*patternTransDefs*⟩)+ **'}'**

⟨*patternTransDefs*⟩ → ID (**'.'** NUM)? (**'('** NUM **')'**)? **';'**

⟨*patternArcs*⟩ → **'arcs'** **'{'** (⟨*patternArcsDefs*⟩)+ **'}'**

⟨*patternArcsDefs*⟩ → ID (**'.'** NUM)? **'->'** ID (**'.'** NUM)? **';'**

⟨*tasksBlock*⟩ → **'tasks'** **'{'** (⟨*tasks*⟩)* **'}'**

⟨*tasks*⟩ → ID **'('** (⟨*listId*⟩)? **')'** **':'** P_ID **'{'** ⟨*taskDef*⟩ **'}'**

⟨*taskDef*⟩ → (⟨*taskActions*⟩ | ⟨*taskEvents*⟩) *

⟨*taskActions*⟩ → **'a'** **':'** ID **'->'** listId **';'**

⟨*taskEvents*⟩ → **'e'** **':'** ID **'->'** listId **';'**

⟨*missionBlock*⟩ → **'mission'** (**':'** ID (**'['** ID **'at'** NUM **']'**))? **'{'** ⟨*statement*⟩ **'}'**

⟨*statement*⟩ → ⟨*call*⟩ (**';'** ⟨*statement*⟩)?
'while' **'('** ⟨*statement*⟩ **')'** **'{'** ⟨*statement*⟩ **'}'** (**';'** ⟨*statement*⟩)?
'if' **'('** ⟨*statement*⟩ **')'** **'{'** ⟨*statement*⟩ **'}'** (**'else'** **'{'** ⟨*statement*⟩ **'}'**)? (**';'** ⟨*statement*⟩)?
'parallel' **'{'** ⟨*statement*⟩ **'}'** ((**'and'** | **'or'**) **'}'** ⟨*statement*⟩ **'}'**)+ (**';'** ⟨*statement*⟩)?
'try' **'{'** ⟨*statement*⟩ **'}'** **'catch'** **'{'** ⟨*statement*⟩ **'}'** (**';'** ⟨*statement*⟩)?
'not' **'('** ⟨*statement*⟩ **')'** (**';'** ⟨*statement*⟩)?
'monitor' **'{'** ⟨*statement*⟩ **'}'** ((**'condition'** | **'while_condition'**) **'('** ⟨*statement*⟩ **')'** **'do'** **'{'** ⟨*statement*⟩ **'}'**)+ (**';'** ⟨*statement*⟩)?

⟨*call*⟩ → ID **‘(’** (⟨*listId*⟩)? **‘)’** (**‘:’** ID)?

⟨*constraintsBlock*⟩ → **‘constraints’** **‘{’** ⟨*constraint*⟩ **‘}’**
⟨*constraint*⟩ → (⟨*mutex*⟩ | ⟨*order*⟩ | ⟨*sync*⟩)*
⟨*mutex*⟩ → **‘_mutex_’** **‘{’** ⟨*listId*⟩ **‘}’** **‘=’** NUM
⟨*order*⟩ → **‘_order_’** **‘{’** ID **‘:’** ID **‘,’** ID **‘:’** ID **‘}’**
⟨*sync*⟩ → **‘_sync_’** **‘{’** ⟨*listId*⟩ **‘}’**

⟨*listId*⟩ → (ID (**‘,’** ID) *)
⟨*P_ID*⟩ → (**‘P_’**) ([A-Z] | [0-9] | **‘_’** | **‘-’**)+
⟨*ID*⟩ → (**‘$’** | [a-z] | [A-Z] | **‘_’**)(**‘$’** | [A-Z] | [a-z] | [0-9] | **‘/’** | **‘_’** | **‘-’**)*
⟨*NUM*⟩ → (**‘-’**)? (**‘0’** | [1-9] ([0-9])*) (**‘.’** ([0-9])+)?

References

Petri net marked language reference, November 2010. URL http://www.pnml.org. 93

Internet communications engine, 2011. URL http://www.zeroc.com/ice.html. 52

LEGO mindstorms, 2011. URL http://mindstorms.lego.com/. 48

The ace orb, 2011. URL http://www1.cse.wustl.edu/~schmidt/TAO.html. 52

Webots, 2011. URL http://www.cyberbotics.com. 49

EPFL education robot, 2011. URL http://www.e-puck.org/. 48

Gavia AUV, 2011. URL http://www.gavia.is. 14, 22

iRobot create, 2011. URL http://www.irobot.com/. 48

OROCOS, 2011. URL http://www.orocos.org/. 47

Urbi forge, 2011. URL http://www.urbiforge.org/. 15, 23, 49

P.E. Agre and D. Chapman. Pengi: an implementation of a theory of activity. In *6th Annual Meeting of the American Association for Artificial Intelligence*, pages 268–272, Seattle, Washington, 1987. 10

R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotic Research*, 17(4):315 – 337, 1998. 30

J.S. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):473–509, May-June 1991. 10

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. 22

J. Amat, J. Batlle, A. Casals, and J. Forest. GARBI: a low cost ROV, constrains and solutions. In *6ème Seminaire IARP en robotique sous-marine*, pages 1 – 22, Toulon-La Seyne, France, 1996. 39

N.M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 113 – 120, 1996. 130

B. Anderson and J. Crowell. Workhorse AUV- a cost-sensible new autonomous underwater vehicle for surveys/soundings, search & rescue, and research. In *IEEE/MTS OCEANS*, Jan 2005. 13, 23

N.A. Anisimov, A.A. Kovalenko, G.V. Tarasov, A.V. Inzartsev, and A.Ph. Scherbatyuk. A graphical environment for auv mission programming and verification. In *10th International Symposium on Unmaned Untethered Submersible Technology*, pages 394 – 405, 1997. 28

R.C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research, August 1989*, 8(4):92–112, 1989. 11

R.C. Arkin and T. Balch. AuRA: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence*, 9:175–189, 1997. 12, 45

K. Asakawa, J. Kojima, Y. Kato, S. Matsumoto, N. Kato, T. Asai, and T. Iso. Design concept and experimental results of the autonomous underwater vehicle AQUA EXPLORER 2 for the inspection of underwater cables. *Advanced Robotics*, 16(1):27–42, 2002. 176

A. Balasuriya and T. Ura. Vision based underwater cable detection and following using AUVs. In *MTS/IEEE Oceans*, Biloxi, Mississippi, October 2002. 176

M. Barbier, J. Lemaire, and N. Toumelin. Procedures planner for an AUV. In *International Symposium on Unmanned Untethered Submersible Technology*, 2001. 15, 32, 37, 57

D. Barrett, M. Grosenbaugh, and M. Triantafyllou. The optimal control of a flexible hull robotic undersea vehicle propelled by an oscillating foil. In *Autonomous Underwater Vehicle Technology, 1996. AUV '96., Proceedings of the 1996 Symposium on*, pages 1 – 9, jun 1996. 12

C. Barrouil and J. Lemaire. Advanced real-time mission management for an AUV. In *SCI NATO RESTRICTED Symposium on Advanced Mission Management and System Integration Technologies for Improved Tactical Operations*, Florence, Italy, September 1999. 15, 32, 37

J. Batlle, P. Ridao, R. Garcia, M. Carreras, X. Cufi, A. El-Fakdi, D. Ribas, T. Nicosevici, and E. Batlle. *URIS: Underwater Robotic Intelligent System*, chapter 11, pages 177 – 203. Instituto de Automatica Industrial, Consejo Superior de Investigaciones Cientificas,, 1st edition, 2004. 40

M. Beetz, T. Arbuckle, T. Belker, A. Cremers, D. Schulz, M. Bennewitz, W. Burgard, D. Hahnel, D. Fox, and H. Grosskreutz. Integrated, plan-based control of autonomous robot in human environments. *Intelligent Systems, IEEE*, 16 (5):56 – 65, 2001. doi: 10.1109/5254.956082. 3

M Berkelaar, J Dirks, K Eikland, P Notebaert, and J Ebert. Mixed integer linear programming solver, November 2010. URL `http://lpsolve.sourceforge.net/5.5/`. 128

H. Berliner. The B*-Tree search algorithm - a best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979. 143

M. Bibuli, R. Bono, G. Bruzzone, and M. Caccia. Event handling towards mission control for unmanned marine vehicles. In *IFAC Conference on Control Applications in Marine Systems*, 2007. viii, 31, 77

G. Biggs, T. Collett, B. Gerkey, A. Howard, N. Koenig, J. Polo, R. Rusu, and R. Vaughan. Player and stage, 2010. URL `http://playerstage.sourceforge.net/`. 48, 50

A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997. 132

Jonathan Bohren. SMACH, 2011. URL http://www.ros.org/wiki/smach_ros. 15, 26

P. Bonet, C.M. Llado, R. Puijaner, and W.J. Knottenbelt. Platform independent Petri net editor 2, November 2010. URL http://pipe2.sourceforge.net. 97

F. Boussinot and R. de Simone. The ESTEREL language. In *Another Look at Real Time Programming IEEE*, volume 79, pages 1293–1304, 1991. vii, 15, 28, 29

R.A. Brooks. A robust layered control system for a mobile robot. *IEEE J. Robot. and Auto.*, 2(3):14–23, 1986. 10, 45

R.A. Brooks. A robot that walks; emergent behaviours from a carefully evolved network. *Neural Computation, 1989*, 1(2):253–262, 1989. 19

W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2):3–55, 1999. 3

M. Caccia, P. Coletta, G. Bruzzone, and G. Veruggio. Execution control of robotic tasks: a Petri net-based approach. *Control Engineering Practice*, 13(8):959 – 971, 2005. vii, 15, 30, 31, 37, 57, 66

A. Caiti, G. Casalino, E. Lorenzi, A. Turetta, and R. Viviani. Distributed adaptive environmental sampling with AUVs: Cooperation and team coordination through minimum-spanning-tree graph searching algorithms. In *IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles*, 2008. 107

M. Carreras, J. Batlle, and P. Ridao. Hybrid coordination of reinforcement learning-based behaviors for AUV control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 1410 – 1415, 2001. 45, 56

M. Carreras, P. Ridao, and A. El-Fakdi. Semi-online neural-qlearning for real-time robot learning. In *IEEE/RSJ IEEE/RSJ Conference on Intelligent Robots and Systems*, pages 662 – 667, 2003. 12, 56

M. Carreras, N. Palomeras, P. Ridao, and D. Ribas. Design of a mission control system for an AUV. *International Journal of Control*, 80(7):993 – 1007, 2007. 153, 192

C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2007. 65, 202

J. Champeau, P. Dhaussy, and L. Latreille. Mission control with the UML and SDL formalisms. In *OCEANS 2000 MTS/IEEE Conference and Exhibition*, volume 3, pages 1639 – 1645, 2000. 32

Z. Chang, X. Bian, and X. Shi. Autonomous underwater vehicle: Petri net based hybrid control of mission and motion. In *International Conference on Machine Learning and Cybernetics*, volume 2, pages 1113 – 1118, 2004. 17

R. Chatila and J. Laumond. Position referencing and consistent world modelling for mobile robots. In *IEEE International Conference on Robotics and Automation, ICRA*, pages 138–170, 1985. 10

K. Cheung, H. So, W. Ma, and Y. Chan. Least squares algorithms for time-of-arrival-based mobile location. *IEEE Transactions on Signal Processing*, 52(4): 1121 – 1130, 2004. 170

S. Chien, B. Smith, G. Rabideau, N. Muscettola, and K. Rajan. Automated planning and scheduling for goal-based autonomous spacecraft. In *IEEE Intelligent Systems and their Applications*, volume 13, pages 50 – 55, 1998. doi: 10.1109/5254.722362. 138

J.H. Connell. SSS: A hybrid architecture applied to robot navigation. In *IEEE International Conference on Robotics and Automation, ICRA*, pages 2719–2724, 1992. 11

G. Conte, S.M. Zanoli, and D. Scaradozzi. A feedback scheme for missions managing in underwater archeology. In *The 7th IFAC Symposium on Intelligent Autonomous Vehicles*, 2010. 2

H. Costelha and P. Lima and. Modelling, analysis and execution of robotic tasks using petri nets. In *IEEE International Conference on Intelligent Robots and System*, 2007. 15, 37

J. Cote and J. Lavallee. Augmented reality graphic interface for upstream dam inspection. In *Proceedings of SPIE*, 1995. 157

X. Cufí, R. Garcia, and P. Ridao. An approach to vision-based station keeping for an unmanned underwater vehicle. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 799 – 804, 2002. 156

D. Davis. *Precision maneuvering and control of the Phoenix Autonomous Underwater Vehicle for entering a recovery tube*. PhD thesis, Naval Postgraduate School, Monterey, California, September 1996. 2

M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960. 132

B. Dawes, D. Abrahams, and R. Rivera. Boost C++ libraries, 2010. URL `http://www.boost.org/`. 60

A. Deshpande and J. B. de Sousa. Real-time multi-agent coordination using DIADEM: Applications to automobile and submarine control. In *IEEE Conference on Systems, Man and Cybernetics*, 1997. 31

D.B. Edwards, T.A. Bean, D.L. Odell, and M.J. Anderson. A leader-follower algorithm for multiple AUV formations. In *Autonomous Underwater Vehicles, 2004 IEEE/OES*, pages 40 – 46, 2004. 107, 191

D. Eickstedt and S. Sideleau. The backseat control architecture for autonomous robotic vehicles: A case study with the iver2 AUV. In *OCEANS 2009, MTS/IEEE Biloxi - Marine Technology for Our Future: Global and Local Challenges*, pages 1 – 8, 2009. 26

A. El-Fakdi, M. Carreras, and E. Galceran. Two steps natural actor critic learning for underwater cable tracking. In *IEEE International Conference on Robotics and Automation ICRA*, Anchorage, Alaska, USA, May 2010. 12, 42, 56, 154, 176, 178

Applied Informatics Software Engineering. Poco C++ libraries, 2010. URL `http://pocoproject.org/`. 60

J. Evans, Y. Petillot, P. Redmond, M. Wilson, and D. Lane. AUTOTRACKER: AUV embedded control architecture for autonomous pipeline and cable tracking. In *MTS/IEEE Oceans*, pages 2651–2658, San Diego, California, September 2003. 176

J. Evans, C. Sotzing, P. Patrón, and D. Lane. Cooperative planning architectures for multi-vehicle autonomous operations. Technical report, Systems Engineering for Autonomous Systems Defence Technology Centre, Jul 2006. 17, 130

D. Ferguson, M. Likhachev, and A. Stentz. A guide to heuristic-based path planning. In *International Conference on Automated Planning and Scheduling*, 2005. 130

R.E. Fikes and N.J. Nilsson. "STRPS": A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189 – 208, 1971. 10

R Firby. An investigation into reactive planning in complex domains. In *AAAI*, pages 201 – 206, Jan 1987. 14, 18

R.J. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, New Haven, Connecticut, 1989. 12, 45

R.J. Firby, R.E. Kahn, P.N. Prokopowic, and M.J. Swain. An architecture for vision and action. In *Fourteenth International Joint Conference on Artificial Intelligence*, pages 72 – 79, 1995. 18

R. Garcia, X. Cufí, and J. Batlle. Detection of matchings in a sequence of underwater images throughtexture analysis. In *International Conference on Image Processing*, 2001. 158, 169

GaryBradski. Open source computer vision library, 2010. URL `http://opencv.willowgarage.com/wiki/`. 50, 60

E. Gat. *Reliable Goal-directed Reactive Control for Real-World Autonomous Mobile Robots*. PhD thesis, Virginia Polytechnic and State University, Blacksburg, Virginia, 1991. 12

M. Ghallab, D. Nau, and P. Traverso. *Automated Planning*. Morgan Kaufmann, Elsevier, 2004. 130, 131, 132, 138, 144

M. Goden and A. Pascoal. Model of an autonomous ocean vehicle. Master's thesis, Dynamical Systems and Ocean Robotics Lab in the Instituto Superior Tecnico of Lisbon, 2001. 153, 173

G. Griffiths, N. Millard, S. McPhail, P. Stevenson, J. Perrett, M. Peabody, A. Webb, and D. Meldrum. Towards environmental monitoring with the Autosub autonomous underwater vehicle. In *International Symposium on Underwater Technology*, pages 121 – 125, 1998. 13

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305 – 1320, Sep 1991. 29

P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Systems Science and Cybernetics*, volume 4, pages 100 – 107, 1968. doi: 10.1109/TSSC.1968.300136. 143

A. Healey, D. Marco, P. Oliveira, and A. Pascoal. Strategic level mission control - an evaluation of CORAL and PROLOG implementations for mission control specifications. In *Symposium on Autonomous Underwater Vehicle Technology*, 1996. 57

A.J. Healey. *Advances in Unmanned Marine Vehicles*, chapter Guidance laws, obstacle avoidance and artificial potential functions, pages 43–66. Number 3. The Istitution of Electrical Engineers, 2006. 155

M. Henning and S. Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co. Inc., 1999. 52

M. Herman, T. Hong, S. Swetz, D. Oskard, and M. Rosol. Planning and world modeling for autonomous undersea vehicles. In *IEEE International Symposium on Intelligent Control*, pages 370 – 375, August 1988. doi: 10.1109/ISIC.1988.65459. 145

E. Hernandez, P. Ridao, M. Carreras, D. Ribas, N. Palomeras, A. El-Fakdi, and F. Chung. Ictineu AUV, un robot per a competir. In *Congrs Catal d'Intelligncia Artificial*, 2006. 192

I. Horswill. Polly: A vision-based artificial agent. In *IEEE National Conference on Artificial Intelligence, AAAI*, 1993. 11

H.M. Huang. An architecture and a methodology for intelligent control. *IEEE Expert: Intelligent Systems and their applications*, 11(2):46–55, 1996. 10

N. Hurtos, A. Mallios, S. Carreno, R. Campos, C. Lee, X. Fuster, S. Cusi, E. Galceran, A. Carrera, M. Villanueva, N.Palomeras, D. Ribas, and M. Carreras. Sparus, the university of Gironas entry for SAUC-E 2010. *International Journal of Maritime Engineering*, 2010. 4, 153, 157

R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. *Lua 5.1 Reference Manual*. Lua.org, August 2006. 24

Silicon Graphics Internationa. Standard template library programmer's guide, 2010. URL `http://www.sgi.com/tech/stl/`. 60

M. Iordache and P. Antsaklis. Synthesis of supervisors enforcing general linear vector constraints in petri nets. *in Proceedings of the American Control Conference*, 1:154 – 159, Apr 2002. 115, 116, 207

M. Iordache and P. Antsaklis. *Supervisory control of concurrent systems*. Birkhäuser Boston, 2006a. 71

M. Iordache and P. Antsaklis. Decentralized supervision of petri nets. In *IEEE Transactions on Automatic Control*, volume 51 (2), pages 376 – 381, 2006b. 124, 125, 190, 193

M. Iordache, J. Moody, and P. Antsaklis. Automated synthesis of deadlock prevention supervisors using petri nets. Technical report, ISIS Group at the University of Notre Dame, 2002a. 121

M. Iordache, J. Moody, and P. Antsaklis. Synthesis of deadlock prevention supervisors using petri nets. *IEEE Transactions on Robotics and Automation*, 18 (1):59 – 68, 2002b. 114

Y. Ito, N. Kato, J. Kojima, S. Takagi, K. Asakawa, and Y. Shirasaki. Cable tracking for autonomous underwater vehicle. In *IEEE Symposium on AUV technology*, pages 218–224, 1994. 176

M.D. Iwanowski. Surveillance unmanned underwater vehicle. In *IEEE Oceans*, pages 1116–1119, 1994. 176

R. Simmonsand D. James. *Inter-Process Communication, A Reference Manual*, April 2011. Version 3.9. 52

Kyle Johns and Trevor Taylor. *Professional Microsoft Robotics Developer Studio*. Wrox, May 2008. 15, 24, 49

M. Kao, G. Weitzel, X. Zheng, and M. Black. A simple approach to planning and executing complex AUV missions. In *Symposium on Autonomous Underwater Vehicle Technology*, pages 95 – 102, May 1992. doi: 10.1109/AUV.1992.225188. vii, 13, 15, 19, 20, 26

W. Kazmi, P. Ridao, and D. Ribas. Dam wall detection and tracking using a mechanically scanned imaging sonar. In *International Conference on Robotics and Automation*, page 3595, May 2009. 159

T.W. Kim and J. Yuh. Task description language for underwater robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003. 15, 21

R. Tinosch R. Kolagheichi-Ganjineh. *Ein hybrider Verhaltensansatz zur Steuerung autonomer Fahrzeuge in Echtzeitumgebungen im Kontext der DARPA Urban Grand Challenge*. PhD thesis, Freie Universität Berlin Institut für Informatik, 2008. 47

D. Kopec, T.A. Marsland, and J.L. Cox. *Computer Science Handbook*, chapter Search. Chapman and Hall/CRC, 2nd ed. edition, 2004. 140

D. Kortenkamp and R. Simmons. *Handbook of Robotics*, chapter Robotic Systems Architectures and Programming, pages 187 – 206. Number 8. Springer-Verlag, 2008. 2, 154

C. Kunz, C. Murphy, R. Camilli, H. Singh, J. Bailey, R. Eustice, M. Jakuba, K. Nakamura, C. Roman, T. Sato, R.A. Sohn, and C. Willis. Deep sea underwater robotic exploration in the ice-covered arctic ocean with AUVs. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 3654 – 3660, 2008. 13

J.E. Laird and P.S. Rosenbloom. Integrating, execution, planning, and learning in Soar for external environments. In T. S. W. Dietterich, editor, *8th Annual Meeting of the American Association for Artificial Intelligence, AAAI*, pages 1022–1029, Hynes Convention Centre, July–August 1990. MIT Press. 10

K. Lautenbach and H. Ridder. The linear algebra of deadlock avoidance - a petri net approach. Technical report, Institute for Computer Science, University of Koblenz, Germany, 1996. 121

D. Lefebvre and G. Saridis. A computer architecture for intelligent machines. In *IEEE International Conference on Robotics and Automation, ICRA*, pages 245–250, Nice, France, 1992. 10

D. Lyons. Planning, reactive. In *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, New York, 1992. 12

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. 14, 20

D. McDermott. Transformational planning of reactive behavior. Technical report, YALEU/DCS/RR-941, Yale University, 1994. 18

K.L. McMillian. A technique of a state space search based on unfolding. In *Formal Methods in System-Design*, pages 45 – 65, 1995. 31

S. McPhail and M. Pebody. Autosub-1. a distributed approach to navigation and control of an autonomous underwater vehicle. In *7th International Conference on Electronic Engineering in Oceanography. Technology Transfer from Research to Industry.*, pages 16 – 22, 1997. vii, 20, 21

J. Moody and P. Antsaklis. *Petri net supervisors for discrete event systems*. PhD thesis, University of Notre Dame, 1998. 31, 207

T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 – 580, 1989. 37, 197, 201

J. Murillo, V. Munoz, D. Busquets, and B. López. Coordinating agents schedules through auction mechanisms. In *CAEPIA's workshop on Planning, Scheduling and Constraint Satisfaction*, pages 105 – 114, 2007. 107

K. Nagahashi, T. Ura, A. Asada, T. Obara, T. Sakamaki, K. Kim, and K. Okamura. Underwater volcano observation by autonomous underwater vehicle "r2D4". In *OCEANS 2005*, 2005. 15, 21

L. Nana, L. Marce, J. Opderbecke, M. Pettier, and V. Rigaud. Investigation of safety mechanisms for oceanographic AUV missions programming. In *Oceans 2005 - Europe*, volume 2, pages 906 – 913, 2005. 28

P.M. Newman. *MOOS - Mission Orientated Operating Suite*, 2005. vii, 26, 28, 48, 57

T. Niemueller, A. Ferrein, and G. Lakemeyer. A lua-based behavior engine for controlling the humanoid robot nao. In *Proc. of RoboCup Symposium 2009*, Graz, Austria, 2009. 25

N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980. 9

N.J. Nilsson. Shakey the robot. Technical Report 323, SRI International, Menlo Park, California, 1984. 10

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):1065 – 1080, 1998. vii, 15, 29, 30, 37, 57, 66

R. Oliveira and C. Silvestre. Supervisão e controlo da missão de veículos autónomos. Master's thesis, Dynamical Systems and Ocean Robotics Lab in the Instituto Superior Tecnico of Lisbon, 2003. 30, 81

A. Ortiz, J. Antich, and B. Oliver. A particle filter-based approach for tracking undersea narrow telecommunication cables. *International Journal of Machine Vision and Applications*, 2009. 176

N. Palomeras, M. Carreras, P. Ridao, and E. Hernandez. Mission control system for dam inspection with an AUV. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2551 – 2556, 2006a. 37, 153, 192

N. Palomeras, P. Ridao, M. Carreras, and E. Hernandez. Design of a mission controller for an autonomous underwater robot. In *Workshop de Agentes Físicos*, pages 167–174, 2006b. 153, 192

N. Palomeras, P. Ridao, and M. Carreras. Defining a mission control language. In *Congrs Internacional sobre Tecnologia Marina*, 2007a. 192

N. Palomeras, P. Ridao, M. Carreras, and J. Batlle. MCL: A mission control language for AUVs. In *Control Applications in Marine Systems*, 2007b. 192

N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre. Towards a mission control language for AUVs. In *17th IFAC World Congress*, pages 15028 – 15033, 2008. 86, 192

N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre. Mission control system for an autonomous vehicle: Application study of a dam inspection using an AUV. In *8th IFAC International Conference on Manoeuvring and Control of Marine Craft*, 2009a. 13, 165, 192

N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre. Using petri nets to specify and execute missions for autonomous underwater vehicles. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009b. 192

N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre. Mission control system for an autonomous vehicle: Application study of a dam inspection using an AUV. In *International Conference on Manoeuvring and Control of Marine Craft*, 2009c. 153

N Palomeras, J C Garcia, M Prats, J J Fernandez, P J Sanz, and P Ridao. A distributed architecture for enabling autonomous underwater intervention missions. In *IEEE Systems Conference*, pages 159 – 164, 2010a. 45, 193

N. Palomeras, P. Ridao, M. Carreras, and C. Silvestre. Towards a deliberative mission control system for an AUV. In *7th IFAC Symposium on Intelligent Autonomous Vehicles*, September 2010b. 154, 193

N. Palomeras, P. Ridao, C. Silvestre, and A. El-fakdi. Multiple vehicles mission coordination using petri nets. In *IEEE International Conference on Robotics and Automation*, pages 3531 – 3536, May 2010c. 154, 193

N. Palomeras, A. El-Fakdi, M. Carreras, and P. Ridao. COLA2: A control architecture forAUVs. *Journal of Oceanic Enginering*, submitted. 193

S. Pang, J.A. Farrell, R.M. Arrieta, and W. Li. AUV reactive planning: deepest point. In *OCEANS 2003. Proceedings*, volume 4, pages 2222–2226, 2003. 22

Terence Parr. Another tool for language recognition, 2010. URL `http://www.antlr.org/`. 60, 225

P. Patrón, J. Evans, and D. Lane. Mission plan recovery for increasing vehicle autonomy. Technical report, Systems Engineering for Autonomous Systems Defence Technology Centre, Mar 2007. 14, 17

P. Patrón, E. Miguelañez, Y.R. Petillot, D.M. Lane, and J. Salvi. Adaptive mission plan diagnosis and repair for fault recovery in autonomous underwater vehicles. In *IEEE Oceans*, Sep 2008. 130, 138

J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley, 1984. 143

J. Pearl and R. E. Korf. Search techniques. *Annual Review of Computer Science*, 2(1):451–467, 1987. doi: 10.1146/annurev.cs.02.060187.002315. 139

J.R. Perrett and M. Pebody. Autosub-1. implications of using distributed system architectures in AUV development. In *International Conference on Electronic Engineering in Oceanography*, 1997. 15, 20

J. Peters. *Machine Learning for Motor Skills for Robotics*. PhD thesis, Department of Computer Science, University os Southern California, 2007. 12

C.A. Petri. Kommunikation mit automaten. *Schriften des Institutes für Instrumentelle Mathematik*, Jan 1962. 197

M. Poupart, P. Benefice, and M. Plutarque. Subacuatic inspections of EDF (electricite de france) dams. In *MTS/IEEE Conference and Exhibition OCEANS*, volume 2, pages 939 – 942, September 2000. 158

M. Prats, D. Ribas, N. Palomeras, J. C. García, V. Nannen, J. J. Fernández, J. P. Beltrán, R. Campos, P. Ridao, P. J. Sanz, G. Oliver, M. Carreras, N. Gracias, R. Marín, and A. Ortiz. Reconfigurable AUV for intervention missions: A case

study on underwater object recovery. *Journal of Intelligent Service Robotics*, Submitted. 193

K. Rajan, C. McGann, F. Py, and H. Thomas. Robust mission planning using deliberative autonomy for autonomous underwater vehicles. In *ICRA Workshop on Robotics in challenging and hazardous environments*, pages 21 – 25, 2007. 14, 18, 129, 130

R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling (Intervale)*, pages 191–233, 1982. 133

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*, pages 151 – 156, 2007. 4, 40, 56, 153, 157, 192

D. Ribas, P. Ridao, and J. Neira. Underwater SLAM for structured environments using an imaging sonar. *Springer Tracts in Advanced Robotics*, 2010. 56

D. Ribas, P. Ridao, LL. Magí, N. Palomeras, and M. Carreras. The Girona 500, a multipurpose autonomous underwater vehicle. In *Proceedings of the Oceans IEEE*, Santander, Spain, June 2011. 193

D. Ribas, N. Palomeras, P. Ridao, M. Carreras, and A. Mallios. Girona 500 AUV, from survey to intervention. *Transactions on Mechatronics*, Submitted. 193

P. Ridao, J. Yuh, J. Batlle, and K. Sugiharat. On AUV control architecture. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 855 – 860, 2000. 13

P. Ridao, J. Batlle, and M. Carreras. *o2ca2*, a new object oriented control architecture for autonomy: the reactive layer. *Control Engineering Practice*, 10(8): 857–873, 2002. 3, 45, 55

P. Ridao, E. Batlle, D. Ribas, and M. Carreras. Neptune: a HIL simulator for multiple UUVs. In *MTTS/IEEE OCEANS*, volume 1, pages 524 – 531, 2004a. 56, 153, 173, 184

P. Ridao, A. Tiano, A. El-Fakdi, M. Carreras, and A. Zirilli. On the identification of non-linear models of unmanned underwater vehicles. *Control Engineering Practice*, 12:1483–1499, 2004b. 153

P. Ridao, E. Batlle, and N. Palomeras. First steps in remote experimentation with UUVs. In *Workshop International en Telerobtica y Realidad Aumentada para Teleoperacin*, 2005. 193

P. Ridao, E. Hernandez, N. Palomeras, and M. Carreras. Remote training in AUV control using HIL simulators. In *Manoeuvring and Control of Marine Craft*, 2006. 193

P. Ridao, M. Carreras, E. Hernandez, and N. Palomeras. *Advances in Telerobotics*, volume 31, chapter Underwater Telerobotics for Collaborative Research, pages 347–359. Advances in Telerobotics, 2007. 193

P. Ridao, M. Carreras, D. Ribas, and R. Garcia. Visual inspection of hydroelectric dams using an autonomous underwater vehicle. *Journal of Field Robotics*, 27 (6):759 – 778, November - December 2010. 42, 153, 165

D. Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zürich, 2000. 46

S.J. Rosenschein and L.P. Kaelbling. The synthesis of digital machines with provable epistemic properties. *TARK: Theoretical Aspects of Reasoning about Knowledge*, pages 83–98, 1986. 10

G. Sandini, G. Metta, and D. Vernon. RobotCub: An open framework for research in embodied cognition. In *IEEE-RAS/RSJ International Conference on Humanoid Robots*, pages 13–32, 2004. 48

M. Saptharishi, C.S. Oliver, C. Diehl, K. Bhat, J. Dolan, A. Trebi-Ollennu, and P. Khosla. Distributed surveillance and reconnaissance using multiple autonomous ATVs: Cyberscout. *Robotics and Automation, IEEE Transactions on*, 18(5):826 – 836, 2002. doi: 10.1109/TRA.2002.804501. 3

M. Schmiing, P. Afonso, F. Tempera, and R. Santos. Integrating recent and future marine technology in the design of marine protected areas - the azores as case study. In *OCEANS-EUROPE*, pages 1 – 7, 2009. 154, 169

T. Schneider and H. Schmidt. Unified command and control for heterogeneous marine sensing networks. *Journal of Field Robotics*, 27(6), 2010. 26

ScienceEncyclopedia. Underwater exploration - history, oceanography, instrumentation, diving tools and techniques, deep-sea submersible vessels, key findings in underwater exploration - deep-sea pioneers, 2011. URL http://science.jrank.org/pages/7100/Underwater-Exploration.html. 1

J. Silva, A. Martins, and F.L Pereira. A reconfigurable mission control system for underwater vehicles. In *OCEANS '99 MTS/IEEE. Riding the Crest into the 21st Century*, volume 3, pages 1088 – 1092, 1999. doi: 10.1109/OCEANS.1999.800141. 15, 31

R. Simmons and D. Apfelbaum. A task description language for robot control. In *IEEE/RSJ International Conference Intelligent Robots and Systems*, volume 3, pages 1931–1937, 1998. URL 10.1109/IROS.1998.724883. 25

R.M. Turner. Orca: Intelligent adaptive reasoning for autonomous underwater vehicle control. In *Proceedings of the FLAIRS - 95 International Workshop on Intelligent Adaptive Systems*, pages 52–62, Melbourne, Florida, 1995. vii, 14, 16, 17

R.M. Turner. Intelligent mission planning and control of autonomous underwater vehicles. In *International Conference on Automated Planning and Scheduling*, 2005. 60, 129, 130

K.P. Valavanis, D. Gracanin, M. Matijasevic, R. Kolluru, and G. A. Demetriou. Control architectures for autonomous underwater vehicles. *Control Systems, IEEE*, 17(6):48 – 64, 1997. 13

F. Vanni, A.P. Aguiar, and A.M. Pascoal. Cooperative path-following of underactuated autonomous marine vehicles with logic-based communication. In *IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles*, 2008. 107, 191

D.S. Weld. An introduction to least commitment planning. *AI magazine*, 15(4): 27, 1994. 138

L.L. Whitcomb. Underwater robotics: out of the research laboratory and into the field. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 1, pages 709 – 716, 2000. 13

WillowGarage. Robot operating system, 2010. URL http://www.ros.org. 49

D. Yoerger, M. Jakuba, A. Bradley, and B. Bingham. *Techniques for Deep Sea Near Bottom Survey Using an Autonomous Underwater Vehicle*, volume 28 of *Springer Tracts in Advanced Robotics*. Springer Berlin / Heidelberg, 2007. 13

V.A. Ziparo and L. Iocchi. Petri net plans. In *ATPN/ACSD 4th International Workshop on Modelling of Objects, Components, and Agents*, 2006. 37