

Trabajo Fin de Máster

Implementation of Algorithms for Navigation of an Autonomous Mobile Robot by using software component based frameworks

AUTOR: João Miguel Marques Melo
DIRECTORES: Dr. D. Francisco José Ortiz Zaragoza
Dr. D.Diego Alonso Cáceres

Index

1	Introduction	1
1.1	Introduction	1
1.2	Goals and Objectives	2
1.3	Structure of the document	2
2	Robotic Subsystems and Architectures	4
2.1	Robotic Subsystems	4
2.2	Robotic Architectures	7
2.2.1	Hierarchical Architecture	8
2.2.2	Reactive Architecture	9
2.2.3	Hybrid Architecture	11
2.3	Perception	12
2.3.1	Sensors	14
2.3.2	Sensor fusion	18
2.4	Navigation	20
2.5	Local navigation	21
2.5.1	Vector Field Histogram	22
2.5.2	Dynamic Window Approach	23
2.6	Global navigation	24
2.6.1	Localization	25
2.6.1.1	Single hypothesis belief	27
2.6.1.2	Multiple hypothesis belief	28
2.6.1.2.1	Markov	29
2.6.1.2.2	Kalman	30
2.6.2	Mapping	32
2.6.2.1	Continuous	33
2.6.2.2	Discrete	34
2.6.2.2.1	Metric	35
2.6.2.2.1.1	Spatial	36
2.6.2.2.1.2	Geometric	40
2.6.2.2.2	Topological	42
2.6.3	SLAM	43
2.6.4	Path Planning	43
2.6.4.1	A* Algorithm	44
2.6.4.2	D* Algorithm	47
2.6.4.3	Potential fields	48
2.7	Motion Control	51
2.7.1	Path Execution	52
2.7.1.1	Pure Pursuit Algorithm	52
2.7.2	Acting	56
2.7.2.1	Siegwart Equations	56
3	Software Engineering for Robotics	59
3.1	Object-Oriented Frameworks	60
3.1.1	Robotic Object-Oriented Frameworks and Middleware	64
3.1.1.1	Player (http://playerstage.sourceforge.net/)	64
3.1.1.2	Webots API (http://www.cyberbotics.com/)	65
3.1.1.3	ARIA (http://robots.mobilerobots.com/wiki/ARIA)	66
3.2	Component-Based Frameworks	67
3.2.1	Types of users and requirements of a component-based framework	70
3.2.2	Robotics Component-Based Frameworks	72
3.2.2.1	OROCOS (http://www.oroocos.org/)	72
3.2.2.2	Orca (http://orca-robotics.sourceforge.net/)	73

3.2.2.3	Marie (http://marie.sourceforge.net/)	74
3.2.2.4	RoboComp (http://robocomp.sourceforge.net/)	76
3.2.2.5	Microsoft Robotics Studio (http://www.microsoft.com/robotics/)	76
3.3	Model-Driven Engineering Frameworks	77
4	Robotic Frameworks used on this project	80
4.1	Smartsoft	80
4.1.1	Smartsoft Component-Based Framework	80
4.1.2	Smartsoft MDSD	83
4.2	MinFr	85
4.2.1	MinFr Component-Based Framework	85
4.2.2	V3 CMM	94
4.2.3	MinFr from the point of view of an end-user	96
5	Application Design	100
5.1	Robotic Platform Pioneer 3-AT	100
5.2	Requirements	103
5.3	Components	103
5.4	Algorithms	106
5.4.1	CommonFunctions	106
5.4.2	MapOperator	107
5.4.3	A*	109
5.4.4	VFH	112
5.4.5	To Goal	113
5.4.6	Pure Pursuit	115
6	Application Implementation	118
6.1	Smartsoft	118
6.1.1	Implementation with the Player/Stage simulator	118
6.1.1.1	Communication objects	119
6.1.1.2	Components	119
6.1.1.2.1	UPCTcompConsole	119
6.1.1.2.2	UPCTcompMissionPlanner	121
6.1.1.2.3	UPCTcompExecutor	123
6.1.1.2.4	UPCTcompAstar	125
6.1.1.2.5	UPCTcompVFH	126
6.1.1.2.6	UPCTcompPurePursuit	127
6.1.1.2.7	UPCTcompToGoal	128
6.1.1.2.8	UPCTcompLocalization	130
6.1.1.3	Deployment	131
6.1.1.4	How to run the application	134
6.1.2	Implementation with the robot Pioneer 3-AT	136
6.2	MinFr	137
6.2.1	Tutorial	139
6.2.1.1	Components Libraries	140
6.2.1.1.1	C_Console	141
6.2.1.1.2	C_ToGoal	144
6.2.1.1.3	C_RobotBasePioneerHAL	149
6.2.1.1.4	C_MobileSimCon	149
6.2.1.2	Application Structure	150
6.2.1.3	Application Distribution	152
6.2.1.4	Activities	157
6.2.1.4.1	C_Console	159
6.2.1.4.1.1	A_Console	159
6.2.1.4.1.2	A_ReportingGoalStatus	161
6.2.1.4.2	C_ToGoal	163
6.2.1.4.2.1	A_AtGoal	163
6.2.1.4.2.2	A_MovingToGoal	165

6.2.1.4.3	C_RobotBasePioneerHAL	169
6.2.1.4.4	C_MobileSimCon	169
6.2.1.5	How to run the application	170
6.2.2	Hybrid Architecture	175
6.2.2.1	Component Libraries	177
6.2.2.1.1	C_Console	177
6.2.2.1.2	C_MissionPlanner	177
6.2.2.1.3	C_ObstaclesDetector	178
6.2.2.1.4	C_Executor	179
6.2.2.1.5	C_Astar	180
6.2.2.1.6	C_VFH	180
6.2.2.1.7	C_PurePursuit	181
6.2.2.1.8	C_ToGoal	181
6.2.2.1.9	C_Localization	182
6.2.2.2	Application Structure	182
6.2.2.3	Application Distribution	182
6.2.2.4	Activities	183
6.2.2.4.1	C_Console	184
6.2.2.4.2	C_MissionPlanner	184
6.2.2.4.3	C_Executor	185
6.2.2.4.4	C_ObstaclesDetector	186
6.2.2.4.5	C_Astar	186
6.2.2.4.6	C_VFH	186
6.2.2.4.7	C_PurePursuit	187
6.2.2.4.8	C_ToGoal	187
6.2.2.4.9	C_Localization	187
6.2.2.5	How to run the application	188
7	Conclusions and Future Work	194
8	References	196
9	Appendices	198
9.1	Appendix A: Diagram of how to work with MinFr	198
9.2	Appendix B: MobileSim laser packet	199

1 Introduction

1.1 Introduction

When speaking about robots, the main question most people have is “*What is a robot?*”.

A robot is a machine, with both mechanic and electronic components, that can perform actions automatically or by remote control, which are based on both computer and electronic programs.

These two types of programs are composed by functions, which basically are a portion of code or a group of electronic components, respectively, that take in inputs, process them and produce outputs.

The process part of these two types of programs consists on performing a task, which can vary from very simple, like summing two inputs and as output put the result, to very complex, like performing an action on a robot.

Also, functions can use other functions to perform more complex tasks, which mean that there are different types/categories of functions.

On this document, the only type of robots that is going to be mentioned is autonomous mobile robots (AMR), which are robots that are able to move from one position to another in an autonomous way, which means that there is no use of remote control.

According to Siegwart and Nourbakhsh [Siegwart04], every time an AMR wants to move, there are three key factors that need to be accomplished:

- “*Where am I?*” – The robot needs to know its location in the environment, its initial position;
- “*Where am I going?*” – The robot needs to know where to go to, its desired final position;
- “*How do I get there?*” – The robot needs to calculate a path to be able to move from its initial position to the desired final position.

In order for an AMR to be able to accomplish these three factors, it needs to have some specific types of subsystems, which can be classified according to their function.

The two most known methods of classification of subsystems of an AMR are robotic components [Siegwart04] and robotic primitives [Murphy00].

The way the subsystems of an AMR are organized in terms of the interactions between them, define the intelligence and behavior of an AMR. There are three different ways of organizing the subsystems of a robot, also known as robotic architectures, which are hierarchical, reactive and hybrid.

With all that was mentioned until now, it is clear that when one wants to built an application for an AMR, it needs not only to build the components but also to define how they will work together, since with same components it is possible to get different behaviours depending on the interactions between them.

On this project, it is described in detail each robot subsystem, and how to organize them to generate different behaviours. Also, it is described the design of an application that simulates an hybrid behaviour, and its respective implementation on two different robotic frameworks, MinFr and Smartsoft.

1.2 Goals and Objectives

This project is divided into five main goals/objectives:

- Definition of a classification method for the various robotic subsystems used for navigation of an autonomous mobile robot. Together with this goal, it comes the analysis of the state of the art in terms of algorithms for local and global navigation of an autonomous mobile robot.
- Implementation on C++ of the algorithms A* and Vector Field Histogram, for global and local navigation, respectively.
- Design of an application that performs an hybrid behaviour in terms of navigation of an autonomous mobile robot. As main feature of this application, there is a decision-making component that makes the robot able to choose between both algorithms depending on environment around it. If the environment is not known, this component must request the local navigation algorithm to guide the robot since on this situation, the main goal is to avoid the unknown obstacles on the environment, and if the environment is known, this component must request the global navigation algorithm to guide the robot since on this case, the main goal is to go through the shortest path.
- Analysis of the state of the art in terms of software engineering for robotics.
- Implementation of the designed hybrid application and respective components on two different robotic frameworks, MinFr and Smartsoft.

1.3 Structure of the document

Excluding this first chapter, this document is divided into eight chapters:

2. Robotic Subsystems and Architecture – The first section of this chapter (2.1) describes the classification method robotic components, and how it was adapted to generate the method of classification used on this project for the various robotic subsystems used for navigation of an autonomous mobile robot.

On section 2.2, it is described each of the three robotic architectures, and then, on the following sections of this chapter it is explained in detail each robotic subsystem, as well as, an analysis of the state of art in terms of the most used algorithms for navigation.

3. Software Engineering for Robotics – This chapter consists on a state of the art in terms of software engineering for robotics. It is described the different types of frameworks that exist on the field of robotics, as well as, some of the most known frameworks of each type.

4. Robotic Frameworks used on this project – This chapter is an extension of the previous one, since it consists on a detailed description of the two robotic frameworks used on the implementation part of this project, MinFr and Smartsoft.

5. Application Design – On this chapter, it is described how it was designed the application that simulates the hybrid behaviour on a robot, in terms of requirements that need to be fulfilled by the application, components that were needed to develop and the relationship between them.

6. Application Implementation – On this chapter, it is described on a step by step way how it was implemented the application explained on chapter 5 on two different robotic frameworks, MinFr and Smartsoft. Also, it is here described a tutorial of how to work with MinFr.

7. Conclusion and Future Works – Here it is mentioned the conclusions taken after finishing this project, as well as, some possible ideas that can be used as future projects.

8. References – Here, it is mentioned all the details about the references used on this project.

9. Appendices – It includes a detailed diagram of how to work with the robotic framework MinFr, and also, how the laser packet of the simulator *MobileSim* is structured.

2 Robotic Subsystems and Architectures

2.1 Robotic Subsystems

The process of navigation in an autonomous mobile robot (AMR) is divided in subsystems, in a way that when a robot has to perform an action each subsystem is responsible for a part of the whole process.

According to Siegwart and Nourbakhsh [Siegwart04], the control scheme of a mobile robot can be divided in four components/subsystems:

- **Perception** – Consists on the components of the robot that are responsible for getting knowledge about its surroundings, its “world”, which usually are sensors.
- **Localization** – It is the component that answers to the question “Where am I?”. In this component the robot calculates its absolute position in the world, and also, its relative position to moving objects, such as a person. Mapping is part of this component.
- **Planning** – It is the component responsible for calculating the path that will make the robot to reach its goal.
- **Motion Control** – It is the component responsible for translating the path given by the planning component into motor inputs, in a way that the robot achieves what is desired.

On Fig. 1 it is shown these components and the relationships between them. This figure also shows that there is a sequence that is followed every time the robot wants to execute a movement.

To explain that sequence in step by step way, consider the situation when a robot wants to go from its actual position to a goal position:

1st step: The perception component of the robot senses the environment around the robot with the sensors.

2nd step: The localization component builds a map with those perception readings.

3rd step: Considering a map, which can be the one built on step 2 or a given one, and some cognitive knowledge, the localization component localizes the robot on the environment.

4th step: The planning component determines the path that is needed to follow to go from the actual robot's position to the goal.

5th step: The motion control component executes the movements that are needed for the robot to follow the path.

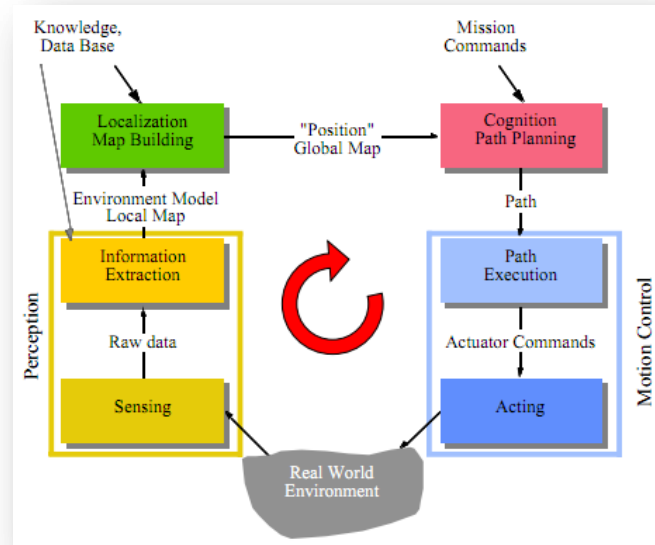


Fig. 1 – Control scheme of a mobile robot considered by Siegwart and Nourbakhsh [Siegwart04]

The method used on this document for the classification of subsystems of a robot is this one but with two changes.

The first change is that mapping is considered as another robotic subsystem and not part of localization, since both localization and planning use the map.

The second change is the addition of a new subsystem called navigation, which according to Fox [Fox97] is divided in two types:

- **Local navigation** - When using this type of navigation, a robot acts based on reactive behaviours, which main priority is to avoid collisions with the obstacles on the environment around the robot but also to reach a goal. To be able to do so, the robot needs real-time exteroceptive sensors readings. On other words, the robot decides how to behave based on a real-time local point of view of what is around it.
- **Global navigation** – When using this type of navigation, a robot needs a map representation of the whole environment, which can be given or built by the mapping component, in order to localize itself and plan a trajectory to reach a goal on the environment around it. On other words, the robot localizes itself and decides the trajectory to reach a goal based on a global point of view of what is around it.

Both types of navigation are goal oriented, the main difference between them is that local navigation is based only on the actual environment around the robot while global is based on the whole environment around the robot.

Making an analogy with situations from the day by day of a human, the local navigation can be when a person wants to cross a street, since it needs to have real-time information about the cars that are coming when it comes up to decide to cross or not, while the global navigation is when a person wants to go from one side of a city to another, and for that it needs first to locate itself by using a map of the city and then plan the trajectory it needs to do to go to the other side.

These two types of navigation can be used together in order to improve the navigation performance of an autonomous mobile robot.

As conclusion for this section, the resulting structure and control scheme that are used on this document to classify the different subsystem of an autonomous mobile robot are the ones shown on Fig. 2 and Fig. 3, respectively.

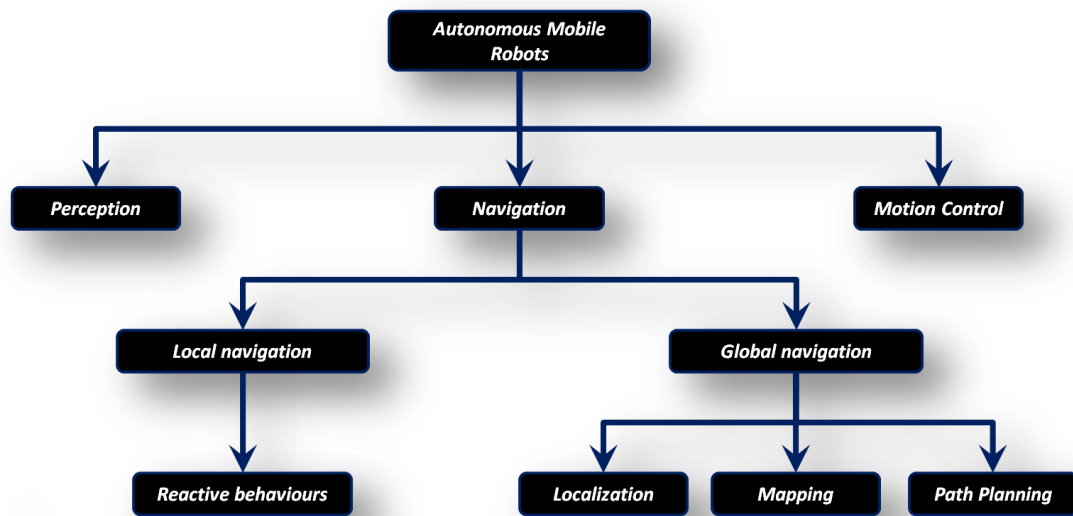


Fig. 2 – Structure of an AMR

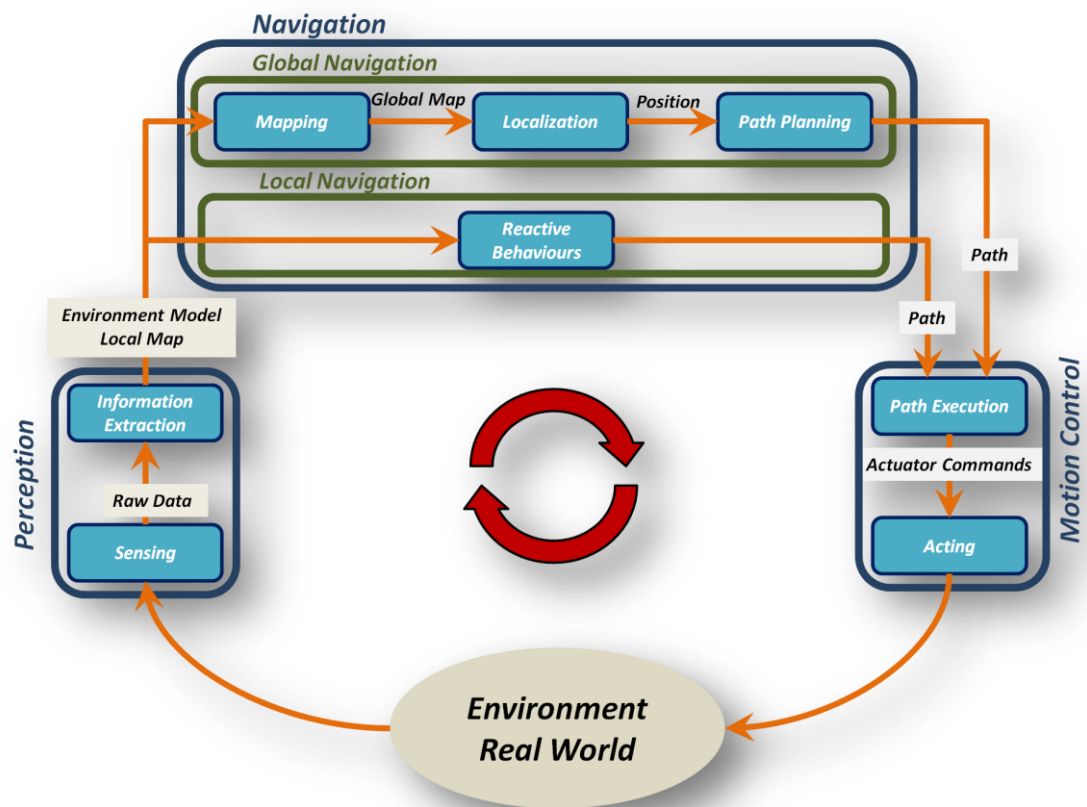


Fig. 3 - Control scheme of an AMR

2.2 Robotic Architectures

According to Murphy [Murphy00], who refers to architectures as paradigms, a paradigm is a set of assumptions and/or techniques which characterize an approach to a class of problems.

In other words, architecture is an approach, which is composed by some techniques/modules, to solve a type of problems. Then, when it is said that we have two architectures for a problem, it means that there are two different ways of solving that problem.

Considering robotics, the problem is how to organize the intelligence of the robots (how is the robot going to behave) and the solutions/architectures are described in terms of the two following factors:

- The relationship between the three robotic primitives;
- The way sensory data is processed and distributed through the system, which can be local or global.

Robotic primitives are categories which define the different functions of a robot in terms of their inputs and outputs (see also Fig. 4):

- **SENSE** – functions that take as input information from the robot's sensors and produce as output sensed information, which is later used by other functions;
- **PLAN** – functions that take as input information (can be sensed or its own knowledge) and produces as output directives, which are tasks for the robot to perform;
- **ACT** – functions that take as input information sensed information or directives, and produces as output commands to motor actuators.

ROBOT PRIMITIVES	INPUT	OUTPUT
SENSE	Sensor data	Sensed information
PLAN	Information (sensed and/or cognitive)	Directives
ACT	Sensed information or directives	Actuator commands

Fig. 4 - Robotic primitives [Murphy00]

Making an analogy with the robotic subsystems mentioned on the previous section, 2.1, the sense primitive is the perception component, the plan primitive is the navigation component and the act primitive is the motion control component.

The sensed information produced by the functions of the category SENSE is called world model, which consists on what the robot can see/sense, and there are two types of it:

- **Local** – when the sensed information is restricted to be used in a specific way for each function of the robot, and so the processing is local to each function;

- Global – when the sensed information is processed first into one global world model by the sense functions and then distributed in subsets to other functions as it is needed.

In terms of the relationships between these three robotic primitives, they can change depending on the design of the chosen architecture, but there are two dependencies that always need to be respected:

- PLAN and ACT always depend on SENSE, because without the sensed information the robot is blind, which means that it doesn't know where it is neither it is able to avoid obstacles.
- ACT depends or not on PLAN, depending on what the robot has to do, because with the sensed information but without the planning part the robot knows where it is but it doesn't know where to go and so, the robot can wonder around but can't be given a task to do since it can't plan how to do it.

Following this theory, there are three different architectures, hierarchical, reactive and hybrid, which are explained on the three following sub sections.

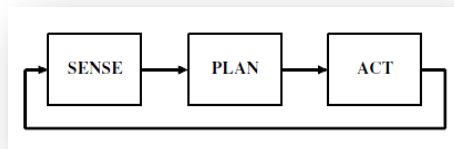
2.2.1 Hierarchical Architecture

This robotic architecture is the oldest one and it's a type of architecture where before each action the robot needs to sense the surroundings, plan the actions and then do it, which means that the robot always does the same sequence before each action.

In terms of behaviors between the robotic primitives, a robot working with this architecture does the following sequence every time it wants to move:

- First, senses its surroundings by using its SENSE functions and creates a global world model with all the sensed information it has gathered;
- Second, with the sensed information sent by the SENSE functions plus its own knowledge, the robot plans the next action (PLAN);
- Finally, with the directives sent by the PLAN functions, the robot actuates on the motors (ACT) in order to execute the planned move.

This architecture works in a loop mode, which means that after each move the robot repeats this sequence in order to plan the next move (see also Fig. 5).



ROBOT PRIMITIVES	INPUT	OUTPUT
SENSE	Sensor data	Sensed information
PLAN	Information (sensed and/or cognitive)	Directives
ACT	Directives	Actuator commands

Fig. 5 - Robotic primitives in hierarchical architecture [Murphy00]

In this architecture, the global world model created on the SENSE functions, it's composed not only by what it senses but can be also composed by a previously acquired representation of the “world”, a map, and some previous knowledge.

The big disadvantage of this architecture is that it can't run concurrent actions at the same time and so, when the environment has objects moving the robot can collide with them.

The resulting control scheme of an AMR with this architecture is the one shown on Fig.

6.

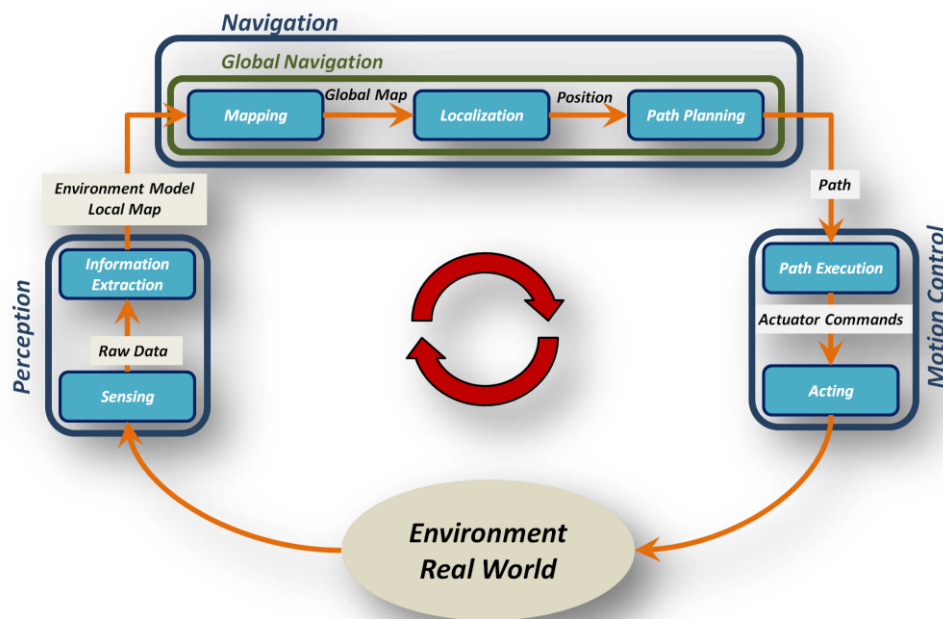


Fig. 6 - Control scheme of an AMR with hierarchical architecture

2.2.2 Reactive Architecture

This robotic architecture only works with two primitives, SENSE and ACT, which means that the input of the ACT functions are not directives but sensed information (see Fig. 7).

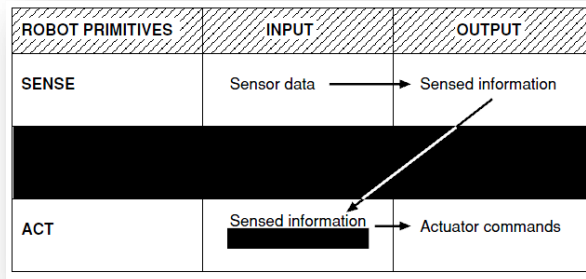
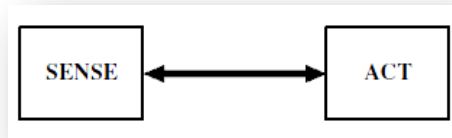


Fig. 7 - Robotic primitives in reactive architecture [Murphy00]

The idea of this architecture is the robot to have a behavior similar to an animal, which behaves in a stimulus-response way and combines concurrent behaviors happening at the same time.

In order to accomplish such behavior in a robot, this architecture permits the existence of concurrent processes of SENSE-ACT, also known as behaviors, running at the same time in a way that there are different ACT functions processing sensed information and acting on the motors at the same time. This architecture also permits the combination of commands by the robot when there are two ACT functions sending different commands to the motors at the same time.

To understand better this feature there is the following example:

When there is a command saying “move forward 10 meters” (ACT on drive motors) to reach a goal (SENSE goal) and another saying “turn 90 degrees” (ACT on steer motors) to avoid a obstacle (SENSE obstacles, the robot will go forward with an angle of 45 degrees, which is a combination of both commands, since no command said to go at 45 degrees, that makes it possible to go towards the goal and avoid the obstacle at the same time.

The two big advantages of this architecture are the fast execution of commands, since there is no planning, and the possibility of having concurrent tasks and combine them. On the other hand, it has a big disadvantage that is not being able to work with human intelligence since there is no planning.

The resulting control scheme of an AMR with this architecture is the one shown on Fig. 8.

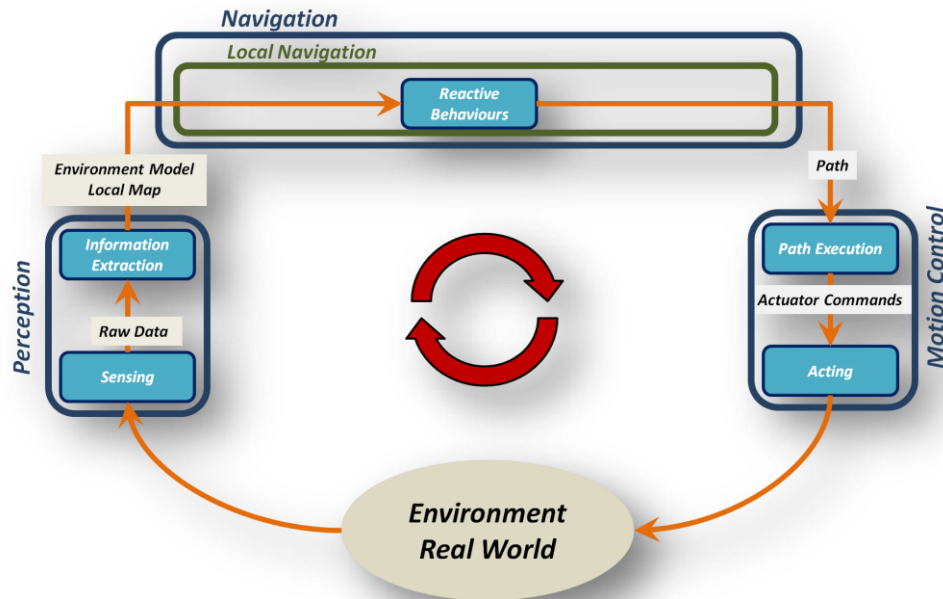


Fig. 8 - Control scheme of an AMR with reactive architecture

2.2.3 Hybrid Architecture

This robotic architecture is a combination of the two previously mentioned architectures which means, that basically it is a reactive architecture with planning, making it possible for the robot to react in real-time but also to plan the actions that it needs to perform (see also Fig. 9).

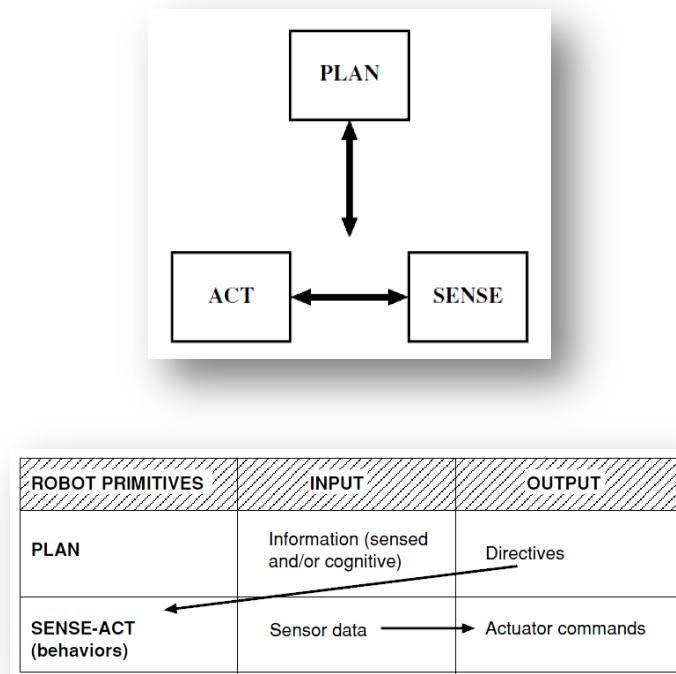


Fig. 9 - Robotic primitives in hybrid architecture [Murphy00]

The idea of this architecture is first, to plan a task for the robot to accomplish, and then execute it by decomposing it on a set of behaviors.

In terms of robotic primitives, this architecture can be thought of as PLAN and then SENSE-ACT.

With this, it means that planning is out of the reactive part of the architecture, but it's also true that planning needs sensing to get knowledge about the environment and so, SENSE is not only used by ACT but also by PLAN, which makes SENSE hybrid since it works with both PLAN and ACT.

So, the main feature of this architecture is SENSE, since at the same time it gives a global world model to PLAN functions, so they can plan the tasks with all the information of the environment around it, and local world models to ACT functions, so they can avoid collisions.

Clearly, this architecture solves the disadvantages of the reactive architecture, being then the best architecture (out of this three) to choose for an autonomous mobile robot.

The resulting control scheme of an AMR with this architecture is the one shown on Fig. 10.

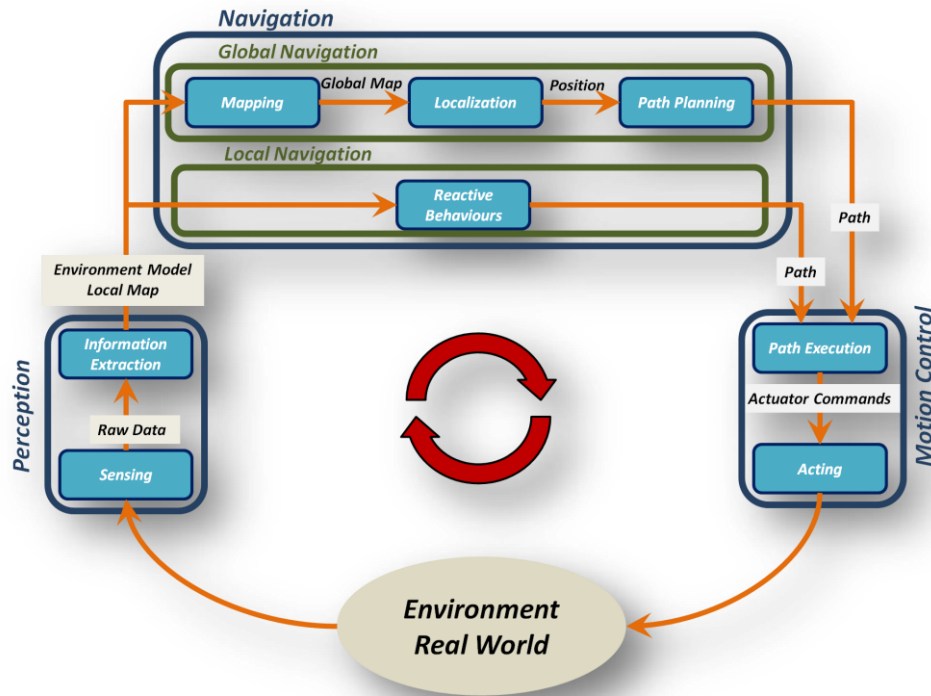


Fig. 10 - Control scheme of an AMR with hybrid architecture

2.3 Perception

One of the most important tasks of an autonomous system of any kind is to acquire knowledge about its environment. This is done by taking measurements using various sensors and then extracting meaningful information from those measurements [Siegwart04].

Perception, consists on the components of the robot that are responsible for getting knowledge about its surroundings, its “world”, which usually are sensors.

According to Murphy [00] there are three types of sensors:

- **Proprioceptive sensors**, measure internal values of the robots such as motor speed, wheel rotation, acceleration and orientation of the robot, etc.
- **Exteroceptive sensors**, get data from the environment around the robot such as distance to features, light intensity, sound amplitude, etc.
- **Exproprioceptive sensors**, get data about the robot's position on the environment by communication between sensors that are on the robot and others that are on the environment.

These three types of sensors can be passive or active, depending on what is supposed to measure.

Passive sensors measure environmental energy such as temperature, sound, image, etc., while active sensors emit some kind of energy into the environment such as light, sound, etc., and then measure the environment reaction to it. Examples of passive sensors are thermometers, microphones or cameras, and of active sensors are lasers, sonars or infrareds.

Considering the three mentioned types of sensors, a robot can get two different types of measurements, absolute position measurements and relative position measurements, like shown on Fig. 11.

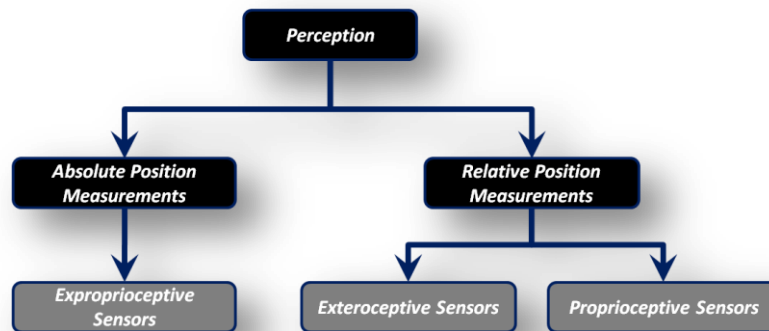


Fig. 11 - The two types of measurements a robot can get with perception

Absolute position measurements are gotten by using the *exproprioceptive* sensors. The measurements of this type of sensors are gotten by communication between sensors on the robot and on the environment, which gives an estimation of the absolute position of the robot on the environment, since it is calculated according to the environment frame of reference.

Relative position measurements are gotten by using *proprioceptive* and/or *exteroceptive* sensors. The measurements of these two types of sensors are gotten by only using sensors on the robot, which gives a estimation of the relative robot position on the environment, since it is calculated according to the robot frame of reference.

It is also possible to get an estimation of the absolute position of the robot on the environment by matching the *exteroceptive* sensors data with a localization algorithm and a map of the environment. This happens because *exteroceptive* sensors measure data from the environment and so, by localizing the sensed data on a given map of the environment, it is possible to get an estimation of the absolute position of the robot on it.

A general diagram with how to get the two estimations of the robot position, absolute and relative, with perception is shown on Fig. 12.

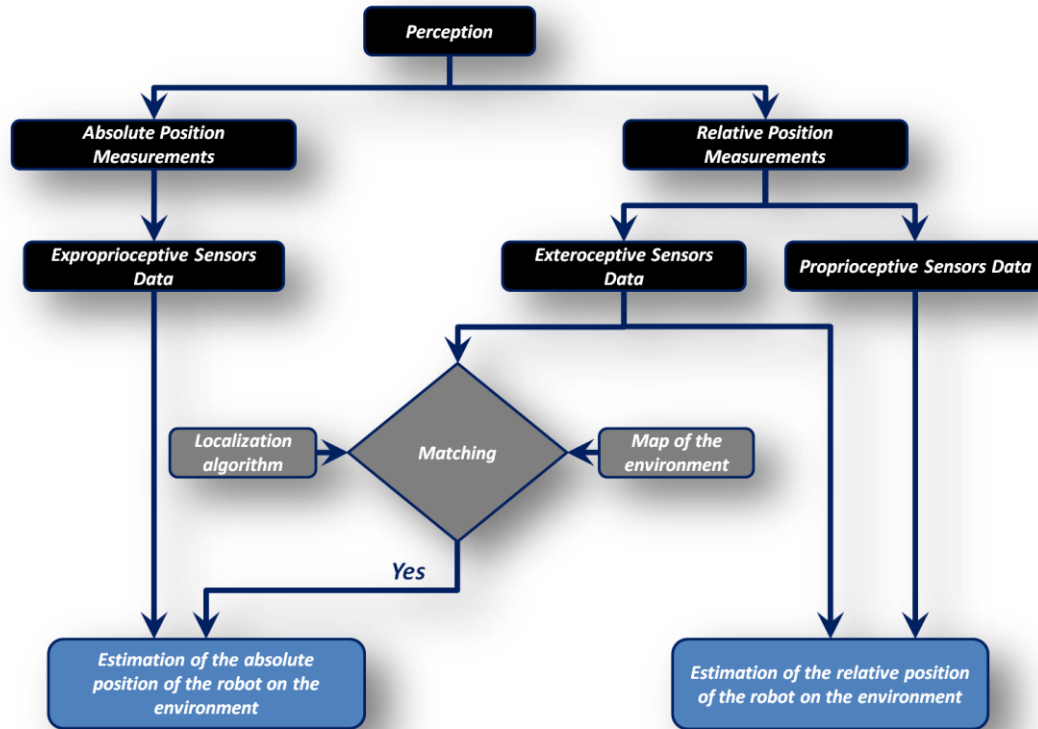


Fig. 12 -How to get the estimations of the absolute and relative position of the robot on the environment with perception

Comparing both types of measurements, when a robot uses relative position measurements, the estimation of the position of the robot has a bigger error than when using absolute position measurements, since they use as reference the robot itself and not the environment. On the other hand, relative position measurements are always possible to get, the sensors are self-contained and can always provide the robot with an estimation of the relative position of the robot, while the absolute position measurements availability depends on the environment where the robot is and on the type of sensors that it has.

Most autonomous mobile robots use the three types of sensors, *proprioceptive*, *exteroceptive* and *exproprioceptive*, in order to get both types of measurements and also the best estimation of the position of the robot in all the situations.

Different sensors and methods of the three types of sensors are shown and classified on the sub section 2.3.1.

A technique that combines measurements of different types of sensors, called sensor fusion, is explained on the sub section 2.3.2.

2.3.1 Sensors

On autonomous mobile robots, sensors play a crucial role since it is needed the data of the sensors in order to make any type of movement. The configuration of each robot in terms of sensors depends on its needs.

Like mentioned before, on perception there are two types of measurements, absolute and relative position measurements, and to get them it is needed to use different types of sensors.

To get absolute position measurements, it is needed to use *exproprioceptive* sensors. There are two types of *exproprioceptive* sensors, active beacons and global computer vision.

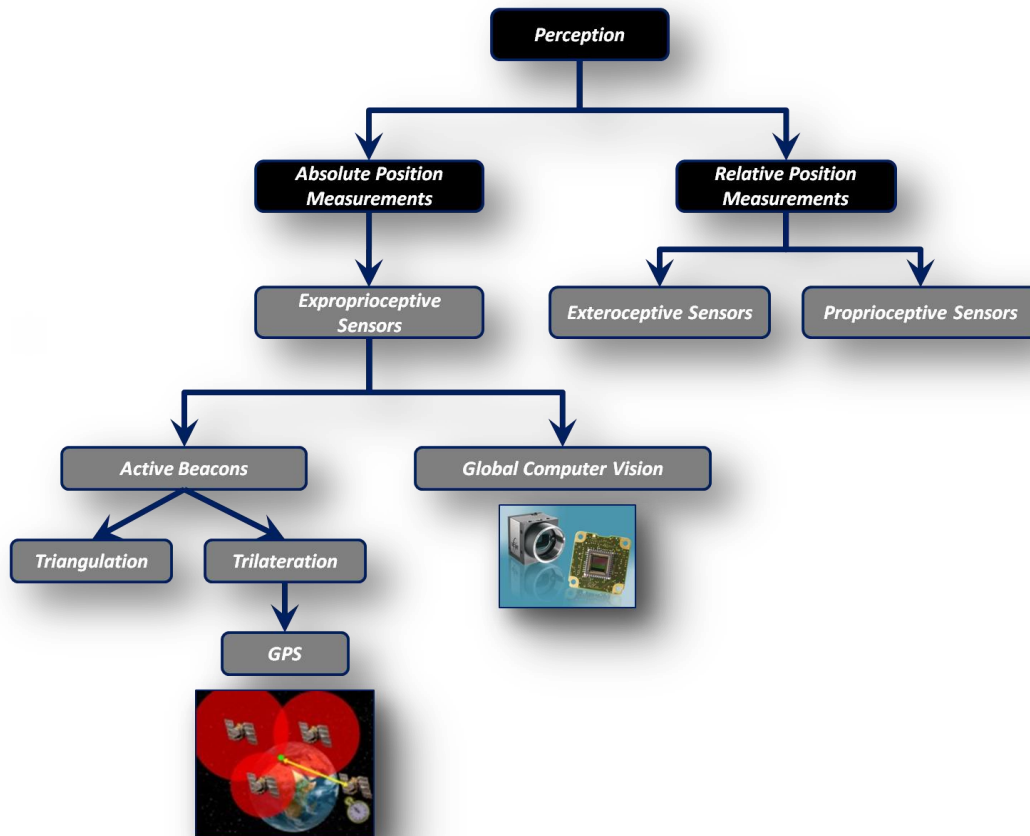


Fig. 13 – Different sensor methods that measure the absolute position of the robot on the environment

- **Active beacons** is a method that calculates the absolute position of the robot by measuring the direction of incidence of three or more actively transmitted beacons. The transmitters usually transmit light or radio frequencies and need to be located at known locations on the environment. There are two implementations of active beacons, triangulation and trilateration:
 - Triangulation is an implementation of active beacons, where there are three or more active transmitters placed at known locations on the environment and a rotating sensor on the robot.
 With the rotating sensor, the robot registers the angles at which it “sees” each of the transmitters relative to the robot’s longitudinal axis and computes its absolute position on the environment based on them.
 - Trilateration is another implementation of active beacons that places three or more active transmitters at known locations on the environment and one sensor on the robot, or the opposite, three or more receivers at known locations on the environment and one transmitter on the robot.
 This method computes the robot’s absolute position on the environment by using the time-of-flight data of the transmitted signals to calculate the distances between the transmitter and the receiver or, between the receivers and the transmitter, depending on the configuration. GPS is an example of trilateration.
- **Global computer vision** is a method that uses cameras placed at a fixed location on the environment and computes the absolute position of the robot on the environment by

detecting the robot on the images provided by the cameras. This method is computer based since it requires image processing to detect the robot and to calculate its position on the environment.

A diagram with the mentioned sensors methods of getting absolute position measurements is shown on Fig. 13.

To get relative position measurements, it is needed to use *exteroceptive* and/or *proprioceptive* sensors.

There are two types of *exteroceptive* sensors, ranging and computer based sensors.

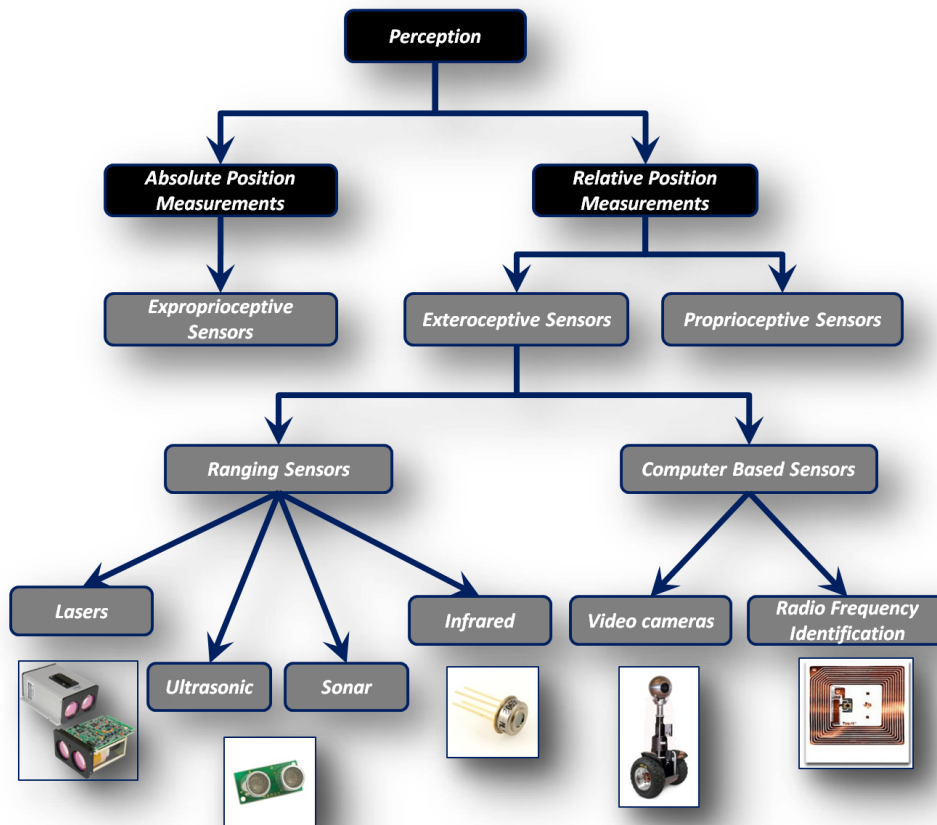


Fig. 14 - Types of exteroceptive sensors

- **Ranging sensors**, measure the proximity between the robot and features on the environment. To do so, this type of sensors emit some type of energy (light, sound) to the environment and, by measuring the environment reaction to them (reflectivity, time-of-flight) the robot calculates its relative position on the environment. Lasers, ultrasonics, infrareds and sonars are examples of ranging sensors.
- **Computer based sensors**, are a type of sensors that require some computer intelligence in order to process the data that they measure from the environment around the robot, and to calculate the relative position of the robot on the environment based on that data. This happens because when sensing the environment, this type of sensors provide an enormous amount of data, where not all is important for the robot.
The measurements that can be gotten with this type of sensors are detection and distances from landmarks.

Landmarks are features that are distinct on the environment and that can be recognized from the sensors data by using techniques of image processing. There are two types of landmarks, natural and artificial:

- Natural landmarks are those features that are already on the environment and have a function other than for robot navigation. The selection of which features are considered as natural landmarks is very important since it determines the complexity of the algorithms of detection. Examples of natural landmarks are doors, wall junctions and corners.
- Artificial landmarks are specially designed objects or markers that are placed on the environment with the purpose of helping on robot navigation. Usually, the size and shape of this type of landmarks is known in advance by the robot. An example of an artificial landmarks is retro reflective barcodes.

Two examples of computer based sensors are video cameras and radio frequency Identificators.

A diagram with the two mentioned types of *exteroceptive* sensors and respective examples is shown on Fig. 14.

Finally, about *proprioceptive* sensors there are also two types, dead reckoning and heading sensors.

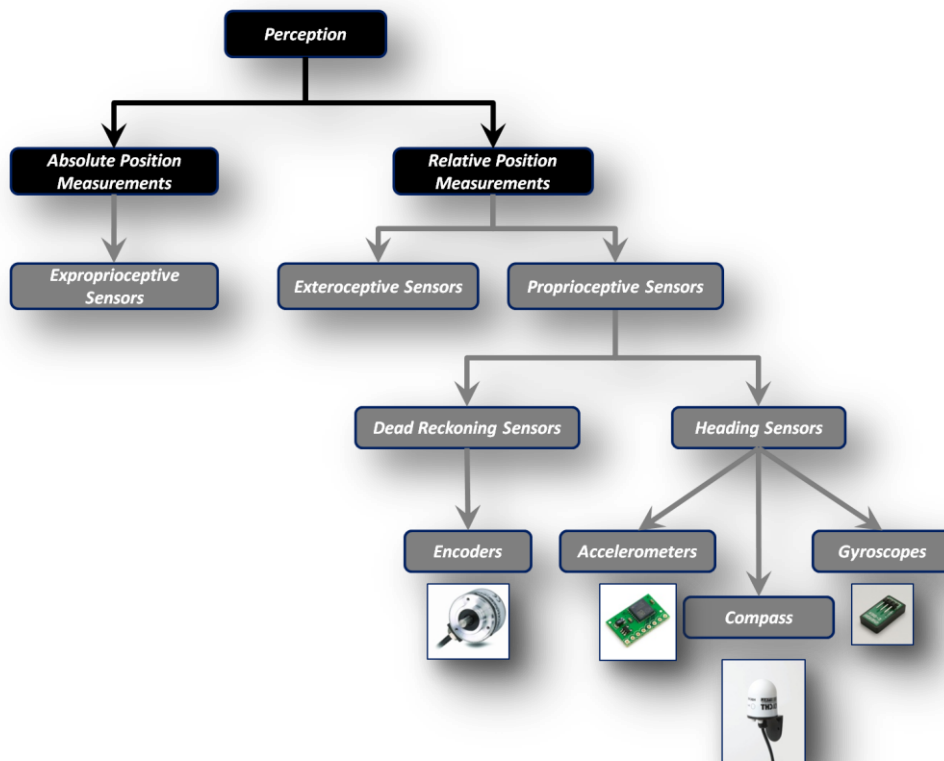


Fig. 15 - Types of proprioceptive sensors

- **Dead reckoning sensors**, also known as odometry, measure wheel rotation, motor speed and/or steering orientation of the robot while moving through the environment. By combining these measurements, the robot is able to calculate how much and how it moved

through the environment and compute its relative position on the environment. Encoders are an example of dead reckoning sensors.

- **Heading sensors**, measure the orientation of the robot in relation to the robot frame of reference. To do so, this type of sensors measure the rate of rotation and acceleration of the robot and by combining them, the robot is able to calculate how much and how it moved through the environment and compute its relative position on the environment. Accelerometers, compass and gyroscopes are examples of heading sensors.

A diagram with the two mentioned types of *proprioceptive* sensors and respective examples is shown on Fig. 15.

The accuracy of the position calculated by using *proprioceptive* sensors data depends only on the sensors noise and limitation but, when using *exteroceptive* sensors, the accuracy also depends on the distance and angle between the robot and the landmarks.

With the sensor data of the *proprioceptive* and *exteroceptive* sensors, it is possible to calculate the relative position of the robot on the environment since what they measure are internal values from the robot and distances between the robot and features on the environment, respectively.

Also, like mentioned on the previous section, it is also possible to calculate the absolute position of the robot by matching the sensor data of the *exteroceptive* sensors with a localization algorithm and a map of the environment. On the case of computer based sensors, the absolute position of the robot on the environment can also be calculated if a database with the description of the landmarks and their location on the environment is given *a priori* to the robot.

A diagram with the all the mentioned sensors and sensing methods of getting absolute and relative position measurements is shown on Fig .16.

2.3.2 Sensor fusion

Sensor fusion in this context is the process of integrating data from distinctly different sensors for detecting objects, and for estimating parameters and states needed for robot self-location, map making, path computing, motion planning, and motion execution [Kam97].

On other words, an autonomous mobile robot gets information about the environment that is around it by using exteroceptive sensors and gets an estimate of its position according to a fixed reference frame by using the proprioceptive sensors readings.

In order to get better estimates of the position of the robot and what is around it, the data of both types of sensors can be integrated. This process of matching data from different sensors is called sensor fusion.

According to Martinez [Martinez01], all the sensors provide readings with a degree of uncertainty depending on their characteristics and so, in many applications, by matching inaccurate information coming from various sensors with sensor fusion, the uncertainty of the global sensor information can get reduced.

There are many techniques of sensor fusion. On the bibliography [Martinez01], [Kam97], [Luo89] and [Crowley] can be seen many different techniques that are usually used for fusion of different sensors readings in a way that, the result of that fusion becomes useful for the other components of the robot.

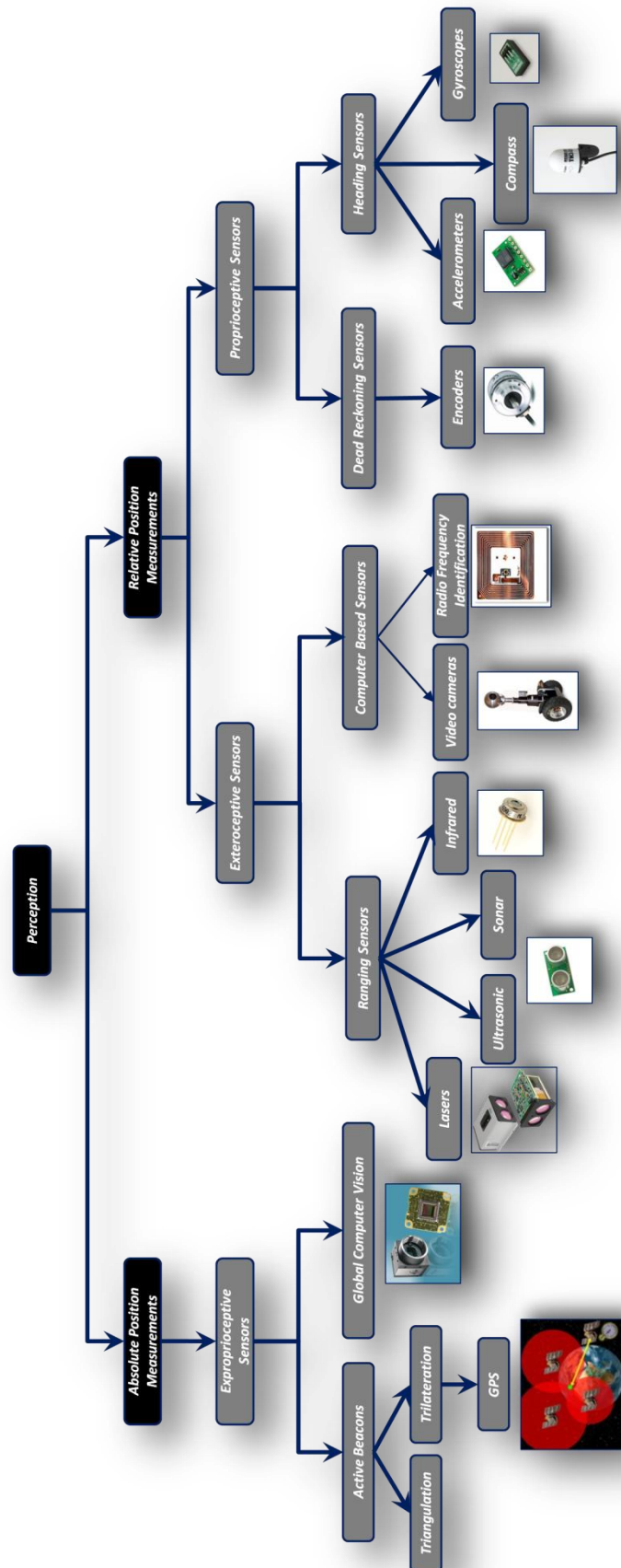


Fig. 16 - Types and examples of sensors used to get absolute and relative position measurements

2.4 Navigation

According to Siegwart and Nourbakhsh [Siegwart04], given partial knowledge about its environment and a goal position or series of positions, navigation encompasses the ability of the robot to act based on its knowledge and sensor values so as to reach its goal positions as efficiently and as reliably as possible.

An autonomous mobile robot must be able to calculate the best path that will make it reach its goal but also have a reactive attitude where it must be able to react in real-time to avoid collisions. For that to be possible, the robot must be given real-time sensors readings so it can realize about possible collisions, calculate a new path to avoid those collisions and still arrive to the goal.

With this, it's means that the navigation component of an AMR includes two competences:

- Given a map and a goal location, planning involves identifying a trajectory that will cause the robot to reach the goal location when executed;
- Given real-time sensor readings, obstacle avoidance means modulating the trajectory of the robot to avoid collisions but still reach the goal.

Considering that and according to Fox [Fox97], it is than clear that there are two types of navigation:

- **Local navigation** – When a robot wants to avoid obstacles but also reach a goal. On this type of navigation, the robot needs real-time exteroceptive sensors readings in order to know what is around it, so it can decide how to behave, in a way that it needs to avoid collisions with the obstacles but also reach the goal;
- **Global navigation** – When a robot wants to plan a path to reach a goal. On this type of navigation the robot needs to localize itself on the environment and plan the trajectory to reach the goal. To achieve these two tasks, it needs a map that can be already given or built by the robot by using the readings of its exteroceptive sensors.



Fig. 17 - The two types of navigation

Both types of navigation can be matched to improve the robot's performance, since without reacting a robot can follow a path to its goal but can't avoid collisions with moving objects, but also with reacting and without path planning, the robot will be able to avoid collisions but will take a lot more time to arrive to the goal.

The resulting control scheme of an autonomous mobile robot with this two types of navigation is shown on Fig. 18.

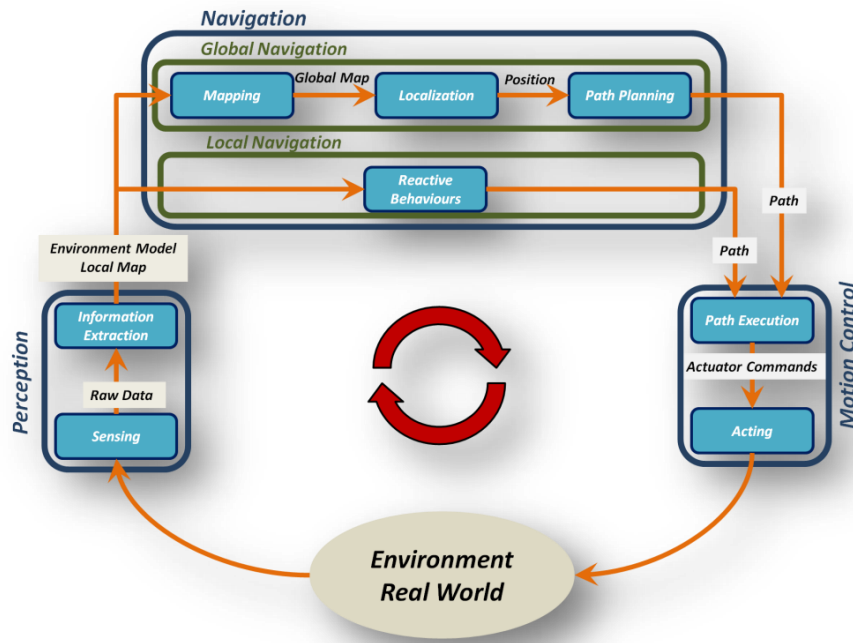


Fig. 18 - Control scheme of an autonomous mobile robot with the two types of navigation

These two types of navigation and their components are explained in detail on the following two sections (2.5 and 2.6).

2.5 Local navigation

Local navigation is the type of navigation that an AMR needs to do when its main priority is to avoid obstacles on the environment around it but also to reach a goal.

In order to do so, the AMR needs real-time exteroceptive sensors readings so it can know what is around it on every second and be able to react according to the reactive behavior it's using. These behaviors are called **reactive behaviors**, since what they do is to react to a stimulus of the robot, which are the real-time sensors readings.

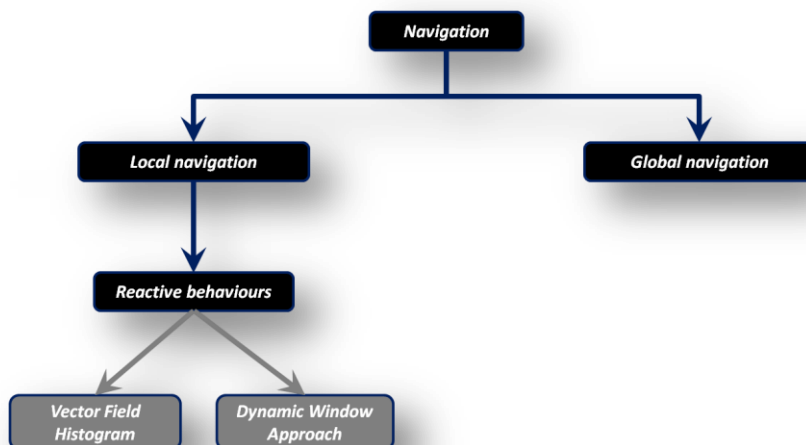


Fig. 19 - Diagram with the two mentioned methods of reactive behaviours

There are many algorithms that implement reactive behaviours. The two most known are the Vector Field Histogram and the Dynamic Window Approach, which are explained on the two following sub sections, respectively.

2.5.1 Vector Field Histogram

This method, named the vector field histogram (VFH), permits the detection of unknown obstacles and avoids collisions while simultaneously steering the mobile robot toward the target.

The VFH method uses a two-dimensional Cartesian histogram grid as a world model. This world model is updated continuously with range data sampled by on-board range sensors. The VFH method subsequently employs a two-stage data reduction process in order to compute the desired control commands for the vehicle [Borenstein91].

On other words, the VFH method is composed by three main components:

- A local map of the environment around the robot. A small occupancy grid is created with the real-time sensors readings;
- A polar histogram, which considers only the map positions closest to the robot;
- A cost function, which is applied to the openings that are large enough for the robot to pass through.

On the polar histogram, the horizontal axis shows the angle at which that position is according to the actual direction of the robot, and the vertical axis represents the probability that there is an obstacle in that direction, which is calculated based on the value of that cell.

To calculate the openings that are large enough for the robot to pass through, a threshold is used on the histogram so that, when there are enough consecutive positions with their cell values below the threshold, that is considered as an opening. An example of a polar histogram with a threshold is shown on the picture below, Fig. 20.

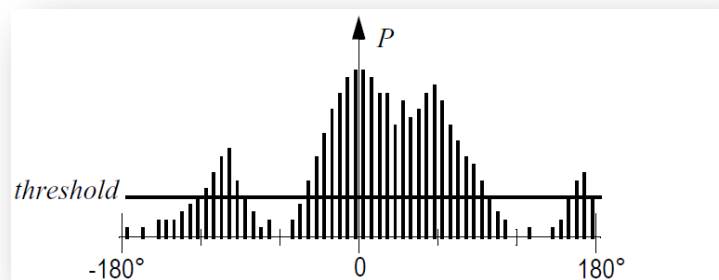


Fig. 20 - Example of a polar histogram with a threshold

The quantity of consecutive positions that is needed to be considered as an opening depends on the dimensions of the robot.

The cost function G that is applied to those openings has the following form,

$$G = A * \text{target_direction} + B * \text{wheel_orientation} + C * \text{previous_direction}$$

target_direction is the direction of the goal considering the actual position of the robot.

wheel orientation is the difference between the new direction and the current wheel orientation.

previous direction is the difference between the previous direction and the new direction.

The terms A, B and C are tuned according to the wanted behaviour for the robot. If the behaviour of the robot is supposed to be goal oriented then the A term must have a large value.

After applying this equation to all the openings, the one with the lowest cost is the one selected for the robot to go through and the robot will avoid collisions as fast as it can under the constraints imposed by the terms, while making progress towards its goal.

2.5.2 Dynamic Window Approach

In this method, the search for commands controlling the robot is carried out directly in the space of velocities. The dynamics of the robot is incorporated into the method by reducing the search space to those velocities which are reachable under the dynamic constraints.

In addition to this restriction only velocities are considered which are safe with respect to the obstacles. This pruning of the search space is done in the first step of the algorithm. In the second step the velocity maximizing the objective function is chosen from the remaining velocities [Fox97].

Given the current robot speed, this method process is divided in two phases.

The first phase corresponds to the process of building the dynamic window, which is done in three steps:

1st step – Considering the actual sensors readings, a dynamic window is created with the circular trajectories (curvatures), which are pairs of translational velocity (v) and angular velocity (ω).

2nd step – The dynamic window is reduced by keeping only those pairs of velocity and angular velocity that ensure that the robot is able to stop before it reaches the closest obstacle on the corresponding curvature. The remaining pairs of velocities are called admissible velocities.

3rd step – The admissible velocities are restricted to only those that can be reached within a short time interval. To do so, this method takes into account the acceleration capabilities of the robot and the duration of the interval.

The second phase corresponds to the process of applying an objective function to all the remaining velocities, in order to get the new direction of the robot.

The objective function O has the following form,

$$O(v, \omega) = A * \text{heading}(v, \omega) + B * \text{velocity}(v, \omega) + C * \text{distance}(v, \omega)$$

heading is the measure of progress towards the goal location with the selected curvature v, ω . It is maximal if the robot moves directly towards the goal.

velocity is the translational velocity of the robot, v . It evaluates the progress of the robot on the corresponding trajectory of the selected curvature v, ω , by checking the translational velocity v .

distance is the distance to the closest obstacle that intersects with the trajectory of the selected curvature v, ω . The smaller the distance to an obstacle is, the higher is the robot's desire to move around it.

The terms A, B and C are used to normalize the three components, heading, velocity and distance, to [0,1] and are tuned according to the wanted behaviour for the robot.

It is necessary to have all the three components, heading, velocity and distance. By maximizing solely the distance and the velocity, the robot would always travel into free space but there would be no incentive to move towards a goal location. By solely maximizing the target heading the robot quickly would get stopped by the first obstacle that blocks its way, unable to move around it.

By combining all three components, the robot circumvents collisions as fast as it can under the constraints imposed by the terms, while still making progress towards reaching its goal.

On the picture below, Fig. 21, it is shown an example of a dynamic window. It is easy to detect the different steps of the two phases of this method. The window V_s (the whole image), is the dynamic window that results from the first step of the first phase. The window V_a (all the light gray parts of the image) is the resulting window of the second step of the first phase. The window V_r (in white) is the resulting window of the third step of the first phase. Finally, the *actual velocity* is selected by applying the objective function O to the window V_r .

Note that the vertical axis refers to the translational velocity v , and the horizontal axis refers to the angular velocity ω .

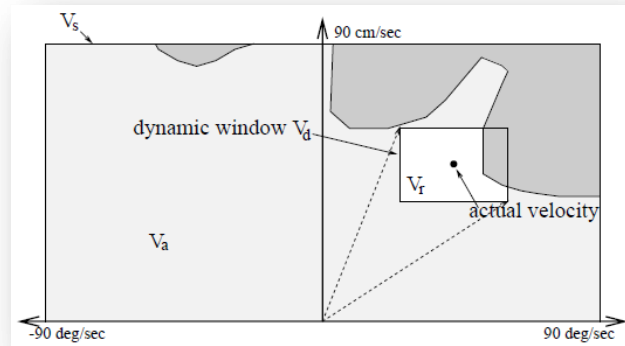


Fig. 21 - Example of a dynamic window approach

2.6 Global navigation

Global navigation is the type of navigation that an AMR needs to do when it wants to plan how to go from a position to another or reach a goal. To do so, an AMR needs to localize itself on the environment by using a map of it and then plan the trajectory that it is needed to follow to arrive at the goal location.

With this, it is clear that this type of navigation is divided in three components, localization, mapping and path planning.

- Localization is the component responsible for localizing the robot on the environment. In other words, it calculates the robot's position, which can be done by matching the sensors readings with a map;
- Mapping is the component responsible by building the map of the environment around the robot. To do so, it makes uses of the exteroceptive sensors readings;
- Path planning is the component responsible for finding the path to go from the robot's actual position until the goal. It does so by using search algorithms on a map of the environment.

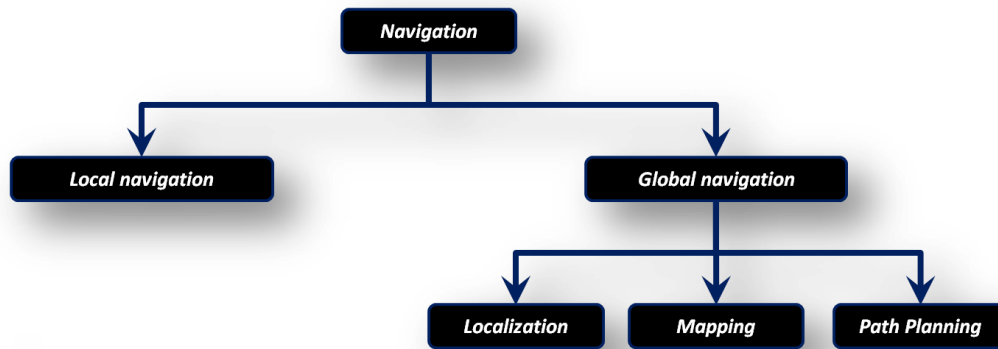


Fig. 22 - The three components of global navigation

These three components are explained in detail on the sub sections 2.6.1, 2.6.2 and 2.6.4, respectively.

2.6.1 Localization

Localization is the component that answers to the question “*Where am I?*”. By saying so, it looks that what this component basically does is to localize the robot in the world by using the information sent by the perception components combined with some cognitive knowledge but in fact, it is more complex than that.

Localization calculates the robot’s absolute position in the world and/or the relative position of the robot to the environment, depending on the architecture of the robot.

To calculate the absolute position it uses the proprioceptive sensors readings, which give information about all the movements the robot has done to go from the initial to the final position, while to calculate the relative position to the environment it makes use of the exteroceptive sensors readings, which give information about what is around the robot, together with a map from the environment.

The environmental map is build by another component called mapping, which creates a map of what the robot knows about the world based on the exteroceptive sensors readings or given information. This map is also used to guide the robot when it wants to go to some location.

Note that the robot can be also be given the environmental map which means that in these cases, the robot will not need to build the map but only to localize itself on it.

The way a robot calculates its relative position to the environment is similar to the behavior of a person when human in a way that, when a person arrives to a place and wants to localize itself, it compares what it is watching(perception) with what we have in memory(map).

In order to get the most possible accurate position, a robot can calculate both absolute and relative positioning. The picture below (Fig. 23) shows the process of matching both measurements.

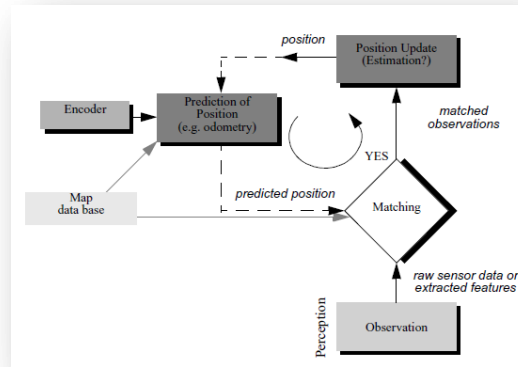


Fig. 23 - Process of localizing a robot based on both types of sensors readings

From this picture, it's clear that there are three very important factors when localizing a robot based on the combination between both relative and absolute measurements:

- Perception, since the robot needs the sensors readings to localize itself and/or to build the map;
- The map of the environment;
- The algorithms that are used to combine all the perception readings and localize the robot on the map.

Considering all that was mentioned until now, it can be concluded that the performance of localization in robotics depends on the following factors:

- Sensor's noise – The sensors readings will lack some environmental features due to the noise and limitations of the sensors;
- Sensor's aliasing – The same sensors reading can appear in different situations. The amount of information given by the sensors in one reading is not enough to identify the robot's position in every situation;
- Precision of the map – How precise is the approximation of the map comparing with the real environment. It depends on the two factors previously mentioned. The precision of the map influences directly the accuracy of the localization of the robot in a way that, when a map is not too precise the localization becomes also less accurate.

Despite the fact that there are some techniques that can reduce the sensor's noise and aliasing, such as taking multiple readings, it's not possible to remove them at all.

With this, it means that it is not possible for a robot to determine the precise position of the robot in every situation but a belief of what can be its actual position.

Beliefs are associated to probabilities, which mean that when calculating the robot's actual position, each position of the map is assigned some probability depending on the previous positions and on the matching between the sensor readings and the map.

Considering how much information is taken into account when determining the belief of a robot's actual position, it is possible to divide localization into two methods:

- Single hypothesis belief – The belief is only composed by one possible position on the map;

- Multiple hypothesis belief – The belief is composed by more than one possibility for the robot's current position. This method requires some algorithms for decision-making.

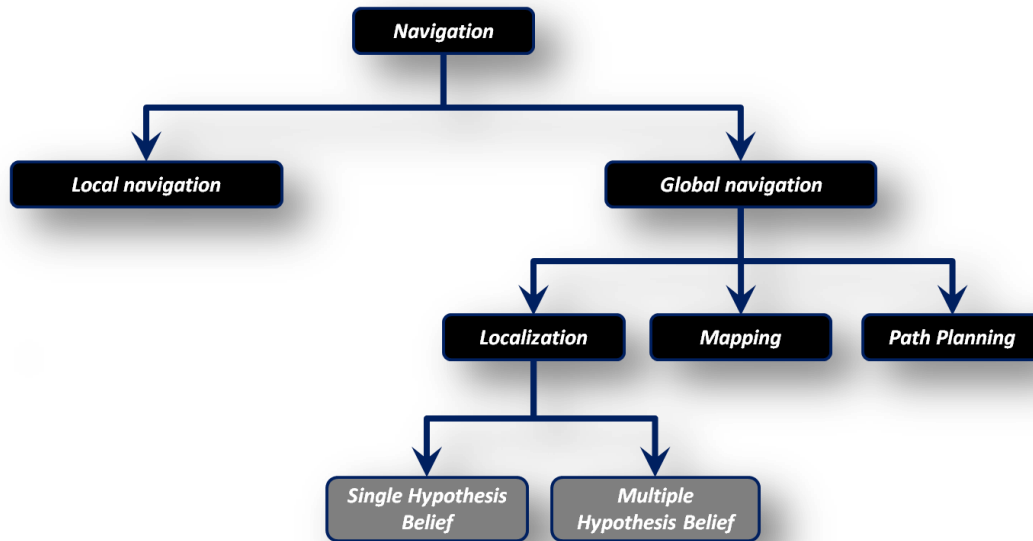


Fig. 24 - The two types of localization

On the two following sub sections it is described in detail these two different types of localization.

2.6.1.1 Single hypothesis belief

As mentioned before, a single hypothesis belief of the robot's position is composed by only one position of the map.

An example of this method is shown on Fig. 25, where it is shown an environment that is decomposed into an occupancy grid and where there is only one hypothesis for the belief actual position of the robot.

The big advantage of using this technique is that is not needed to use any algorithm of decision making to decide the actual position of the robot, which means that it doesn't require much computational processing.

On the other hand, the big disadvantage of this technique is the position uncertainty by selecting only one possible position, since there is always sensor's noise which creates a big uncertainty.

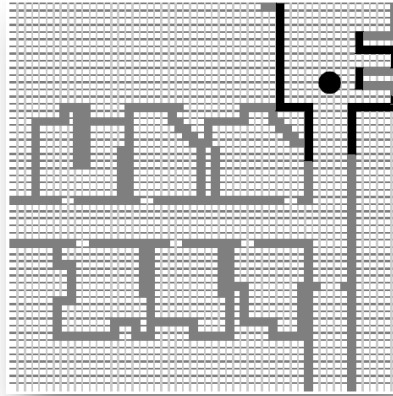


Fig. 25 - Example of a single hypothesis belief on an occupancy grid

2.6.1.2 Multiple hypothesis belief

When a robot uses a multiple hypothesis belief to represent its actual position, the belief is not composed only by one single possible position but by a set of possible positions.

This set of possible positions is ordered according to their probability of being the robot's actual position, which is calculated according to the robot previous positions and the matching between the sensors readings and the map.

The construction of a belief state is divided into two phases, action update and perception update:

- Action update consists on combining the proprioceptive sensors measurements with the former belief state to create the new belief state.
- Perception update consists on combining the exteroceptive sensors measurements with the new belief state in order to improve it, since encoders usually have a big error.

There are many method of multiple hypothesis belief localization. The two most known are Markov and Kalman:

- **Markov** is a method that determines the robot's position by using an arbitrary probability density function to represent the robot's position. The probabilities are calculated by taking in account the robot's last known position and the matching between the sensor values and the map of the environment;
- **Kalman** is a method that determines the robot's position by first, combining various sensor readings for feature extraction and then by matching that combination result with a map of the environment.

The two following sections (2.6.1.2.1 and 2.6.1.2.2) describe in detail these two methods of multiple hypothesis belief localization.

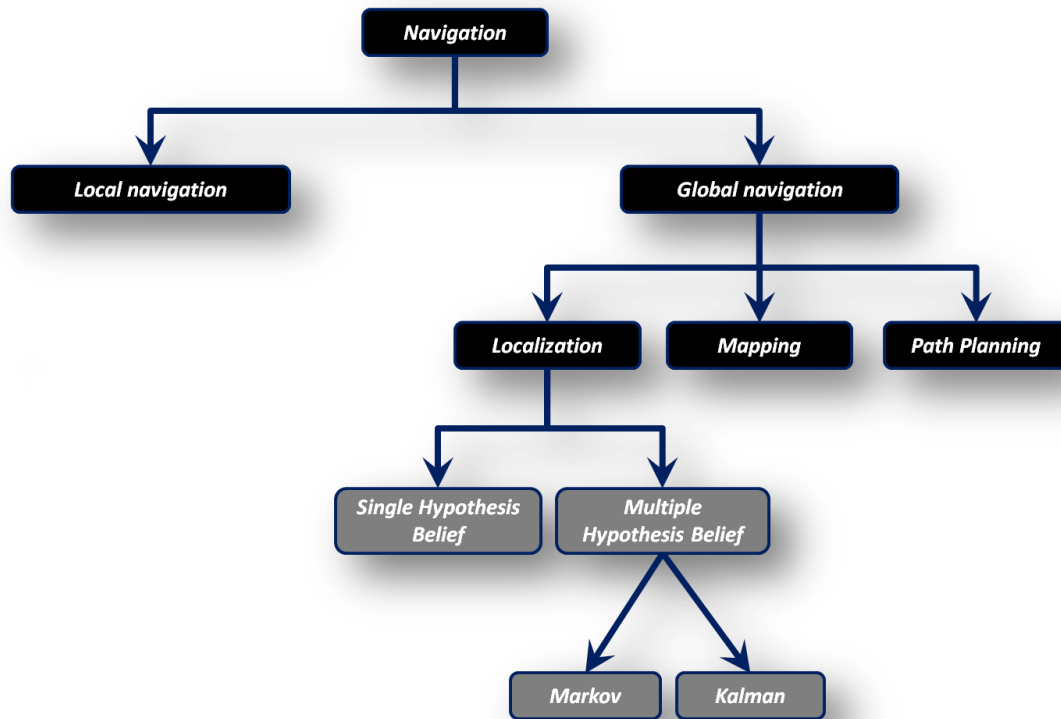


Fig. 26 - The two mentioned methods of multiple hypothesis belief localization

2.6.1.2.1 Markov

The Markov method tracks the robot's belief state by using an arbitrary probability density function to represent the robot's position.

In other words, this method assigns to each position on the map a probability of that to be the actual position of the robot, which depends on the sensor readings and on the robot's last position, its former belief state.

All the probabilities together form the actual belief state of the robot, which can be defined as $p(r=l)$, denoting the probability p that the robot r is at position l for all the possible robot positions.

On the Markov method, each phase of the construction of a new belief state has a different way of being calculated:

- On the action phase, the belief state is denoted as $p(l|o)$, the probability p that the robot is at position l given the encoder measurements o .
To compute the robot's new belief state as a function of its proprioceptive sensors measurements and its former belief state it's needed to integrate over all the possible ways in which the robot have reached l according to the potential positions expressed in the former belief state.
This is done because the same location l can be reached from multiple source locations with the same proprioceptive sensors measurements and also because this measurements are uncertain.
- On the perception phase, the belief state is denoted as the probability p that the robot is at the position l given the sensor outputs I , $p(l|i)$.
To compute the robot's new belief state as a function of its exteroceptive sensor readings and its former belief state it's needed to use the Bayes rule, which is mentioned below,

$$p(l|i) = \frac{p(i|l)p(l)}{p(i)}$$

On this equation $p(i)$ can be ignored since it doesn't depend on l and so, it never varies.

The key for the calculation of this belief state is the factor $p(i|l)$, the probability p of a sensor input i at each robot position l , which is computed by checking the robot's map and matching the sensor readings with each possible map position.

The results of both belief states can be combined in order to get a more accurate estimation of the robot's position.

Also, the two mentioned ways of calculating the belief state respect the Markov assumption, which says that their output needs to be a function only of the robot previous state and its most recent actions and sensor inputs.

The big disadvantage of the Markov method is that it only takes in account the previous state and not the previous states. On the other hand, a big advantage of this method is that it allows the robot to locate itself from any starting unknown position.

2.6.1.2.2 Kalman

The Kalman filter is a mathematical mechanism for producing an optimal estimate of the system state based on the knowledge of the system and the measuring device, the description of the system noise and measurement errors and the uncertainty in the dynamic models [Siegwart04].

This method works similar to the Markov one with the difference that on the perception update phase, instead of treating the exteroceptive sensors measurements as a whole like on the Markov method, it treats them as a set of extracted features from the environment, as long as the error characteristics of the mentioned sensors can be approximated to unimodal zero-mean Gaussian noise.

Those extracted features result from a fusion between the different exteroceptive sensor readings, and the Kalman method later determines the robot location by making a fusion between the distances estimates from each environment feature detected and the expected distances from environment features, which are based on the robot position estimate gotten by combining the proprioceptive sensors measurements with the robot's old location and a map of the environment.

A scheme with the process of determining the robot location with the Kalman method is shown on Fig. 27.

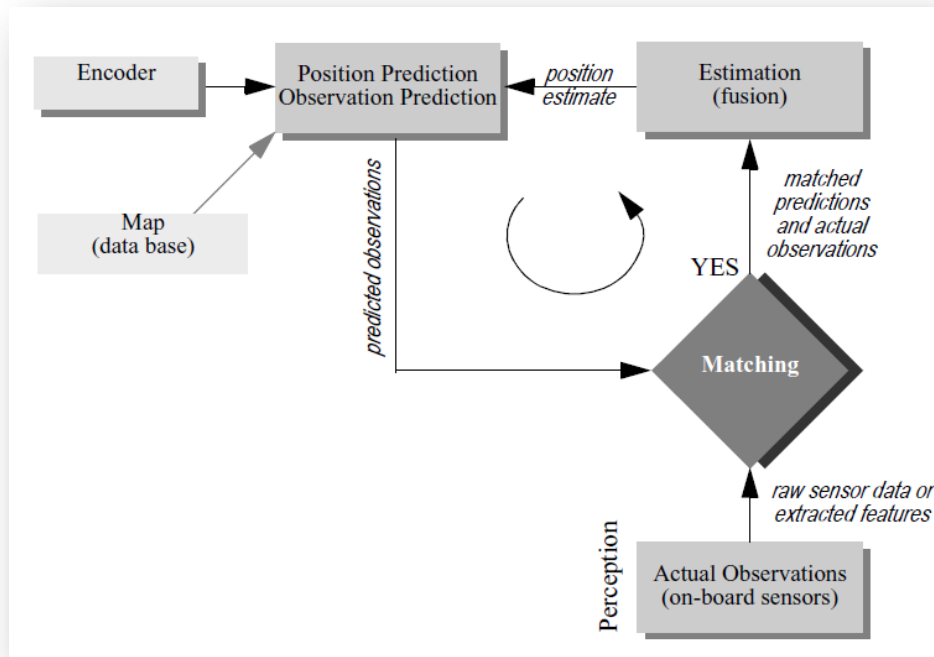


Fig. 27 - Scheme of how to determine the robot location with the Kalman method

With all this, it is then clear that the process of determining the robot location with the Kalman method is divided in five steps:

1st step: Position Prediction. This step consists on the action update phase, which estimates the robot actual position based on the proprioceptive sensors measurements and its old location.

2nd step: Observation. This step consists on obtaining the exteroceptive sensors measurements and extracting the appropriate features from the environment from those measurements.

3rd step: Measurement Prediction. This step consists on identifying on the map from the environment, which features the robot expects to find and at which distance and orientation from them.

4th step: Matching. This step consists on identifying the best pairings between the features extracted during the 2nd step (Observation) and the expected features that result from the 3rd step (Measurement Prediction).

5th step: Estimation. This step consists on determining the best estimate of the robot actual position based on the position prediction that result from the 1st step (Position Prediction) and on the results from the matching between the observations and the expected features that result from the 4th step (Matching).

The two big disadvantages of the Kalman method is that if the uncertainty of the sensors becomes too large it can become lost and also, that it needs the robot to start from a known position. On the other hand, a big advantage of this method is that it requires less computer power and processing than the Markov method.

2.6.2 Mapping

The most natural representation of the robot's environment is a map [Dudek00].

A map in robotics is the representation of the environment around the robot in terms of the exteroceptive sensors readings. The most used exteroceptive sensors in robotics are lasers and sonars, which means that a map constructed by the readings of these two types of sensors shows where the obstacles and free spaces of the environment around the robot are.

Mapping, also known as map representation, is a very important part of navigation since every time the robot wants to move on the environment around it, it uses the map to localize itself on the environment and to guide itself through it. With this, it means that credibility of the map influences the credibility of the localization and movements of the robot.

According to Siegwart and Nourbakhsh [Siegwart04], there are three key factors that must be taken in account when deciding which type of map representation to use:

- The precision of the map must appropriately match the precision with which the robot needs to achieve its goals;
- The precision of the map and the type of features represented must match the precision and data types returned by the robot's sensors;
- The complexity of the map representation has direct impact on the computational complexity of reasoning about mapping, localization and navigation.

In addition to representing places in an environment, a map can include other information such as, reflectance properties of objects, regions that are unsafe or difficult to traverse, or information of prior experiences.

To choose which mapping method to use in robotics, it is needed to take two factors in account:

- Which type of exteroceptive sensors the robot has;
- The required precision required in terms of representation of the environment.

Considering these two factors, there are two mapping methods, continuous and discrete.

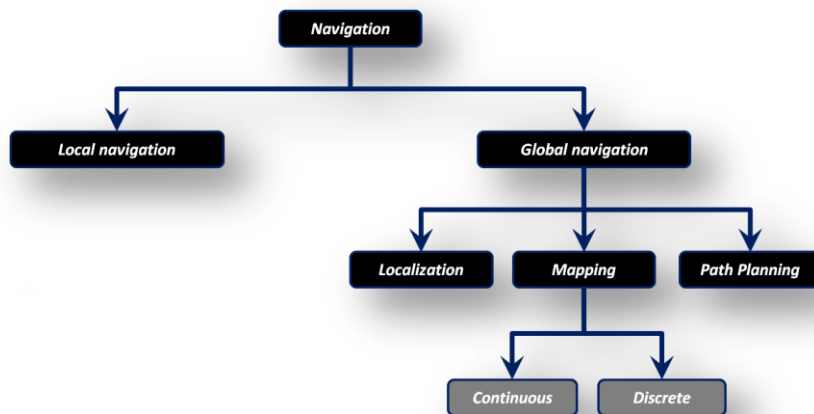


Fig. 28 – The two types of mapping

Continuous mapping basically make a map of what the robot sees, the sensor readings, while discrete mapping make a discretization/decomposition of the sensor readings and the resulting values make the map.

The following sub sections describe these two different types of mapping, reasons to use them and how to build them.

2.6.2.1 Continuous

A continuous map in robotics is a representation of the environment around the robot as it is, without any use of discretization, based only on the sensors readings. Since there is no discretization, this type of maps has a very high precision on the position of the robot and the environmental features.

Considering that the most used types of sensors are lasers and sonars, which measure the distance from the robot to the features on the environment, when using this type of mapping, the resulting continuous map of the environment is basically constituted by continuous lines that represent the shape of the features on the environment.

With this it means that the size of the map in terms of memory depends on the density of features of the environment where the robot is moving. A continuous map of an environment with a high density of features will be big in terms of memory while a map of an environment with a low density of features will be low in terms of memory.

In order to reduce the computational processing and memory size of the continuous maps, there are some techniques which select only aspects of interest from the environment in terms of localization and planning and then represent the environmental features by using geometric approximations of their shape.

An example of one of those techniques like is shown on the picture below (Fig. 29) where it is shown a continuous map of an environment that uses polygons to represent the environmental features.

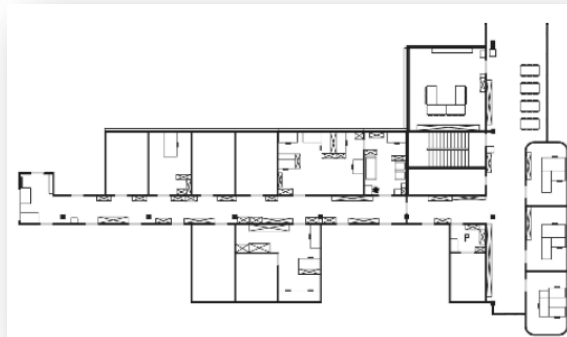


Fig. 29 - Example of a continuous map where the environmental features' shape is approximated by polygons

Adding to this, there is another technique that is commonly used to reduce even more the memory size of the maps which is called line extraction. This technique selects the most important lines from all the sensor readings in a way that the lines represent approximately the shape of the environmental features.

An example of how to implement this technique is shown on the picture below (Fig. 30), where it is shown on the left (a) a continuous map approximated by polygons, and then on the right (b) the same map but approximated by using the line extraction technique.

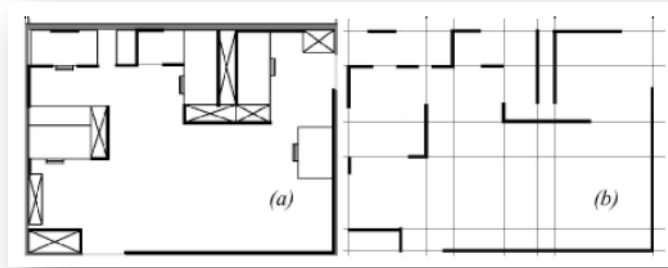


Fig. 30 - Example of a continuous map where the environmental features are approximated by using line extraction

In summary, all these techniques that reduce the complexity and size of the maps do some kind of decomposition of the environment by selecting only more relevant information, which means that when using these techniques the precision of a continuous map doesn't depend only on the sensor's precision but also on the type and how much information the techniques use.

As conclusion about continuous mapping it can be said that there is a big advantage and two disadvantages.

The advantage is that when representing the environment of a robot as a continuous map there is a very high precision in terms of robot and environmental features position since it respects always the environment architecture and no decomposition method is used.

The two disadvantages are the map's size and the computational cost in terms of processing, which can be both reduced by using some techniques that select only the most relevant information from the sensor readings by reducing the precision of the map.

2.6.2.2 Discrete

Considering the environment around the robot, what this mapping method does is a general decomposition/discretization of the environment and then a selection of features according to the needs of the robot. For that reason, this method is also called as abstraction since what it does is an abstraction from the real world in terms of map representation.

The big advantage of this method is the computational cost since with a map of this type, the localization and planning become easy to compute.

The big disadvantage is the loss of fidelity between the map and the real world because of the discretization and abstraction.

Inside the discrete mapping method there are two main groups:

- Metric – Represents the environment as a discrete map that is based on the absolute positions of the environmental features according to the robot's coordinate's frame;
- Topological – Represents the environment as nodes and connections, where each node represents an environmental feature and the connections represent the geometric relationship between them.

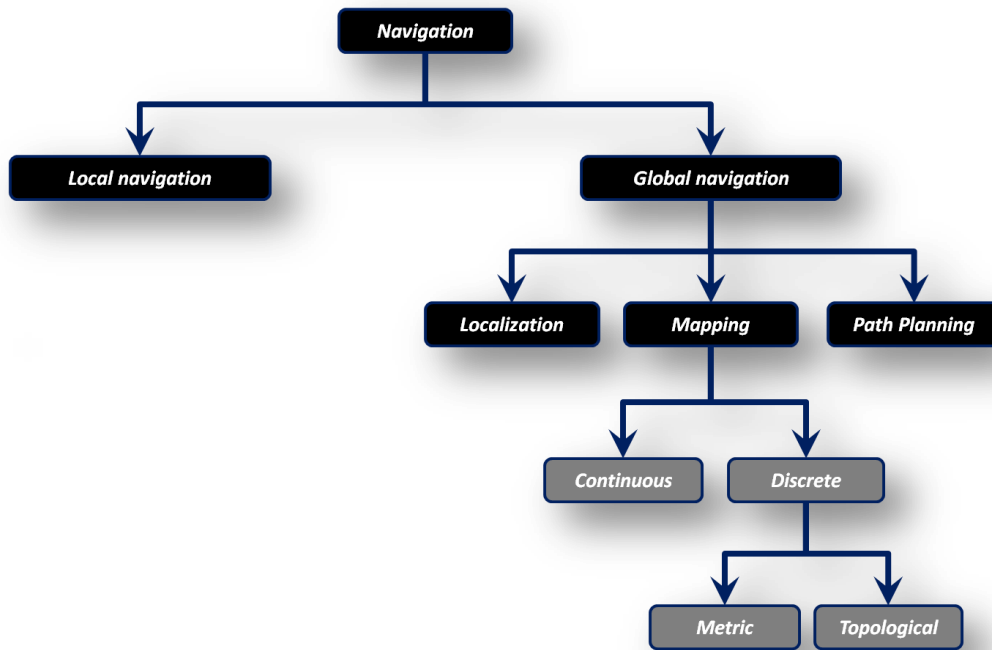


Fig. 31 - The two types of discrete mapping

The following two sub sections describe in detail these two different types of discrete mapping.

2.6.2.2.1 Metric

As mentioned before, this type of maps represents the absolute position of the environmental features according to the robot's coordinate's frame.

In other words, this type of discrete mapping creates a discrete map of the environment based on the distances between the different environmental features and the robot, which means that when using this type of maps it's possible to know the robot's absolute position and also to calculate distances and angles between the various environmental features and the robot.

There are two key factors that need to be taken into account when generating a map with this technique:

- Precision, which depends on how accurate the measurements of the distances that represented in the map are;
- Computational cost, which depends on how precise the map is, in a way that the more precise it is, the more costly it becomes in terms of computation.

The key point of this type of mapping is the way the decomposition of the environment around the robot is made. The two most known techniques of decomposition that are used to build metric maps are:

- Spatial – Decomposes the environment into areas, also known as cells. All the cells together form a grid and each cell corresponds to a part of the environment;
- Geometric – Decomposes the environment into lines by using the geometric properties of the environmental features in a way that those lines combined form the map.

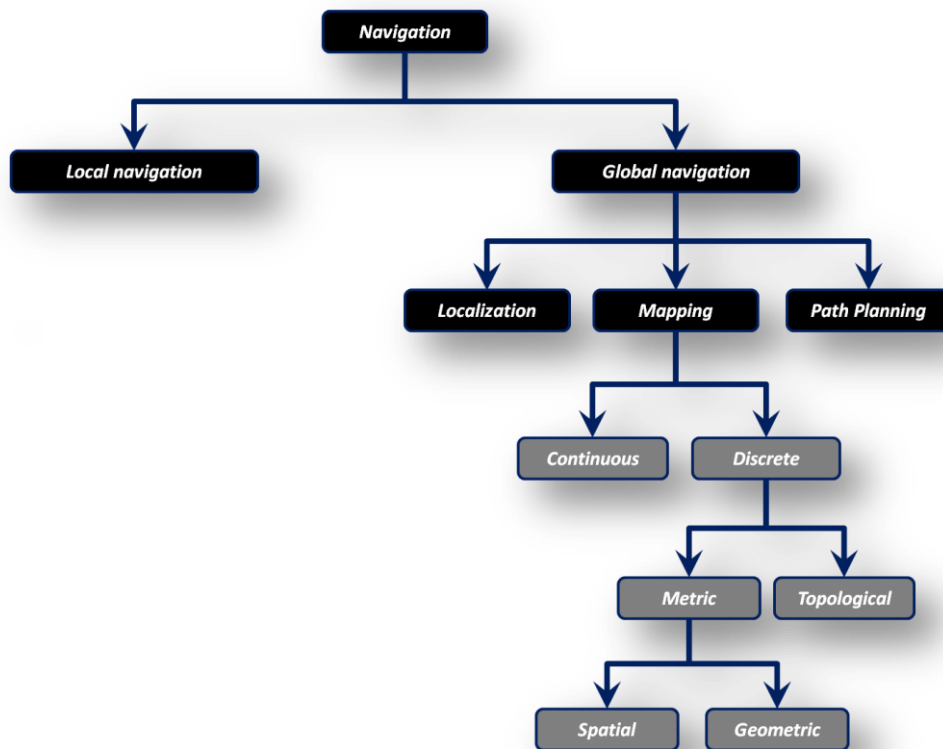


Fig. 32 - The two types of discrete metric mapping

The following two sub sections describe in detail these two different types of discrete metric mapping.

2.6.2.2.1.1 Spatial

Considering a robot's environment, this technique discretizes it by creating a grid where each cell corresponds to a portion of that environment that is homogeneous or that respects some imposed limitations.

With this, it's clear that the precision of a map generated by this technique depends not only on the number of cells that it has but also on the criteria used to build the cells.

Based on this, it's possible to define three spatial discretization techniques which are commonly used:

- Exact cell decomposition – The environment is discretized into areas/cells of free space. Each cell is considered as a node and has adjacencies with other cells/nodes;
- Fixed cell decomposition – The environment is discretized into cells with all of them having the same size, creating then a grid;
- Adaptive cell decomposition – The environment is discretized into cells with each cell's size depending on how close they are to an obstacle in a way, that as closer to an obstacle as smaller the cells are.

▪ Exact cell decomposition

This technique discretizes the environment into cells of free space, also called exact cells, in a way that each exact cell needs to have a polygonal shape and to correspond to a part of the environment that has no obstacles.

The exact cells shape is defined in function of the environment obstacles and free space.

The final representation that comes from using this technique is a graph that is constituted by all the exact cells and their adjacencies with the other exact cells.

The main idea of this technique is that it's not important to know the exact position of the robot but in which exact cell it is and how it is able of going to other exact cells.

According to this and also to what was said before about topological maps, the maps generated by this technique can be considered as topological maps that are created by making a metric discretization.

An example of this technique is shown on the picture below (Fig. 33), where it is shown an environment with three polygonal obstacles that is decomposed into exact cells resulting in 18 exact cells with different polygonal shapes.

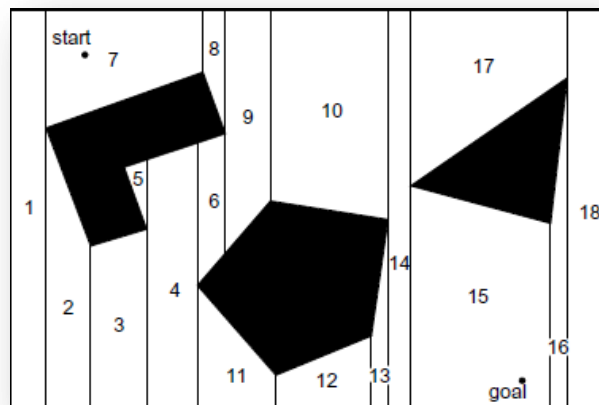


Fig. 33 – Example of exact cell decomposition

The big disadvantage of using this technique is that it's not possible to know the exact position of the robot, which makes this technique not appropriate for the cases that it's needed such information.

▪ Fixed cell decomposition

This technique discretizes the environment into a grid where all the cells have the same fixed size.

Here all the cells have the same shape, squared, and each cell value depends only on what it represents from the environment.

The most known technique of fixed cell decomposition is called occupancy grid and what it does is representing the environment as a grid where each cell can only have one out of two possible values, free or obstacle.

When a cell has the value free it means that the robot can go through it since it's not an obstacle and, when it has the value obstacle it means that the position is an obstacle and so the robot can't go through it.

An example of this technique is shown on the picture below (Fig. 34), where it is shown an environment with three polygonal obstacles that is decomposed into an occupancy grid. The

white cells represent free space, the black ones represent the borders of the obstacles and the gray cells the “body” of the obstacles.

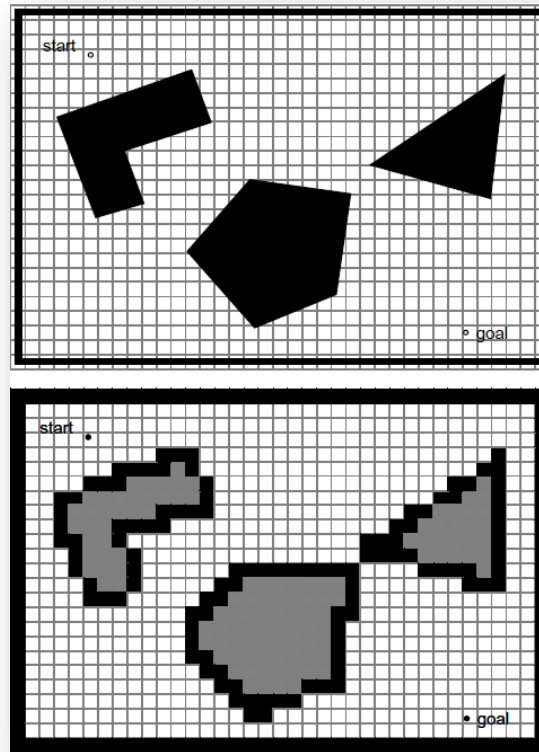


Fig. 34 - Example of fixed cell decomposition

When using this technique, the precision of the grid/map grows with the increasing of the resolution of the grid. With this, it means that the bigger the resolution is, the less information each cells represents and so the more precise the map is about the environment it represents.

This statement is true for all the cases but not always is needed a big resolution and so, when defining the size of the cells is important to consider three very important factors:

- Robot's size – It's important that the cell size is smaller than the robot's size so the exact position can be estimated;
- Environmental features' size – If the environment contains big features it's not needed to have such a big resolution as when it contains many small details;
- Type of information the robot needs to know – For example, if a robot needs to be as fast as possible on his movements a higher resolution can be used in order to calculate the fastest path by going closer to the obstacles but, if it only matters about avoiding the obstacles then a small resolution can be enough.

In general, this technique works for all the cases since it can give the exact robot position but, it has three big disadvantages:

- The size of the grid/map grows with the size of the environment it represents and with the size of the cells in a way that, as smaller the cells are and as bigger the environment is, the more memory is needed to store the map;
 - The geometric shape it gives to all the features of the environment by discretizing it in squares, which makes it a bad approximation in the cases that the environment is not mainly composed by geometric features;
 - The vanishing of narrow passages by having a cell size that is smaller than the size of some passages between obstacles.
- **Adaptive cell decomposition**

This technique is similar to the previous one, fixed cell decomposition, in a way that it also discretizes the environment into a grid but all the rest is different since with this technique each cell has a size that depends on how close it is to an obstacle way.

In other words, when using this cell decomposition technique, the cell's size adapts to the distance that it is from the obstacles. To do so, it needs to follow a procedure which is explained below.

Considering the environment around the robot, the first thing that this technique does is to divide the environment into two big cells.

Then, it uses a procedure of recursive decomposition of the environment where for each cell,

- If the values inside it are homogenous it remains with the size that it has;
- Otherwise, if the values inside it are not homogeneous, the cell is divided into four cells all with the same size.

This procedure is done until all the cells within the grid have all homogenous values inside them or until some predefined resolution size is obtained.

The considered values on this technique are obstacle and free like on technique before which means, that in the end of this procedure all the grid is composed by cells of various sizes but where all of them can only have one out of two possible values, free or obstacle, and within them they need to be homogeneously an obstacle or free space.

An example of this technique is shown on the picture below (Fig. 35), where it is shown an environment with three polygonal obstacles that is decomposed into cells of different sizes by using the adaptive cell decomposition. The white cells represent free space, the gray ones represent the borders of the obstacles and the black cells the "body" of the obstacles.

Note that on this example some predefined resolution was used in a way that the minimum allowed size for a cell is the one of the gray cells.

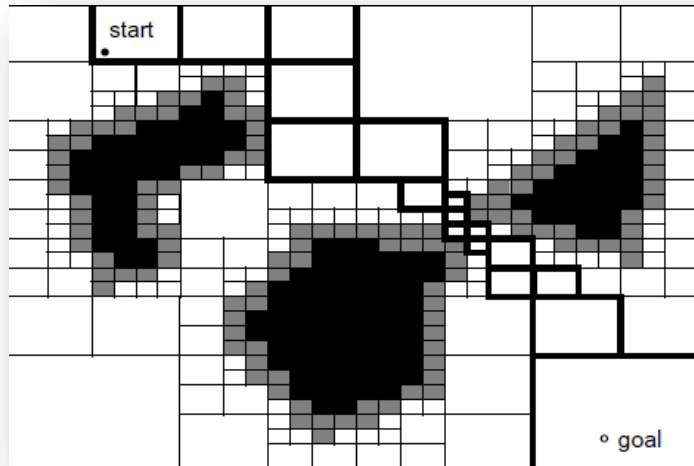


Fig. 35 - Example of adaptive cell decomposition

In summary, this technique is quite similar to the previous one, fixed cell decomposition, but with the advantage of solving the problem of the narrow passages due to its adaptive size for the cells.

2.6.2.2.1.2 Geometric

Considering a robot's environment, this technique of discrete metric mapping discretizes the environment into lines by using geometric properties of the environmental features in a way that those lines combined form the map.

The geometric properties used to build the lines and the ways those lines are combined depend on the type of geometric discretization technique used.

The two most known geometric discretization techniques are:

- Meadow maps – This technique builds lines between the boundaries and corners of the environmental features and combines them as convex polygons. The areas of the convex polygons represent the free areas of the environment;
- Generalized Voronoi graphs – This technique builds lines that are equidistant from all the obstacles and combines them in vertices. If the robot follows the lines it will always avoid obstacles since it's always staying in the middle between them.

▪ **Meadow maps**

As mentioned before, this technique generates meadow maps, which is a representation of the environment that uses convex polygons to represent the free areas between obstacles. Those convex polygons are constructed by combining lines that connect features of the environment.

To succeed on creating a meadow map there are three very important steps that always need to be accomplished in the following order:

- Construction of lines between pairs of features such as corners and objects boundaries;
- Combination of those lines with the lines that represent the walls in order to construct convex algorithms;

- Removal of the lines that are not being used by any polygon.

The final representation that comes from using this technique is a graph that is constituted by all the polygons and their adjacencies with the other polygons.

With this representation of polygons it's possible to know where the free areas are but it is not possible to plan a path. To solve this problem there are some techniques that can be used.

The most known technique calculates the middle point of each line that makes part of the borders of the polygons and with those points creates a graph that shows the adjacencies between middle points.

An example of a meadow map is shown on the picture below (Fig. 36), where it is shown an environment that is decomposed into lines between corners on the top figure, and on the bottom figure it is shown the middle points of all the lines that compose the polygons.

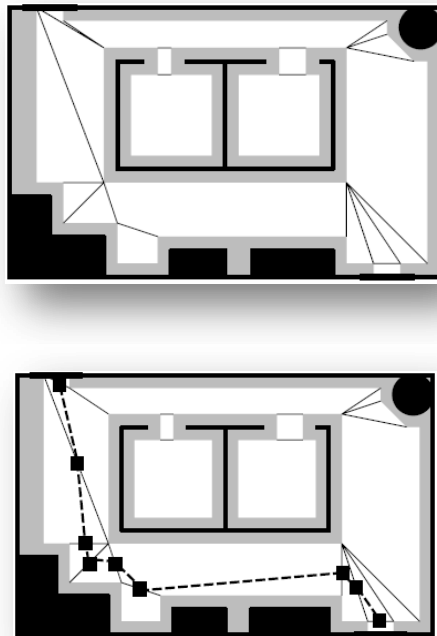


Fig. 36 - Example of a meadow map

This technique has a very big disadvantage which is the fact of being computationally complex and costly to determine the features of the environment and to generate the polygons.

▪ **Generalized Voronoi graphs**

As mentioned before, this technique builds lines that are equidistant from all the obstacles/features on the environment and combines them in vertices. If the robot follows the lines it will always avoid obstacles since it's always staying in the middle between them.

In other words, this technique is somehow a mixture between metric and topological representation since it makes uses of metric to build the lines but the final representation is a graph with vertices and lines connecting them.

The obstacles and features that are selected to build the lines depend on the needs of the robot in terms of localization and planning.

An example of this technique is shown on the picture below (Fig. 37), where it is shown an environment that is decomposed into lines that are equidistant between environmental features. As it can be seen, those lines are combined in vertices in order to build a graph with the purpose of the robot to follow it and with this being avoiding all the possible obstacles.

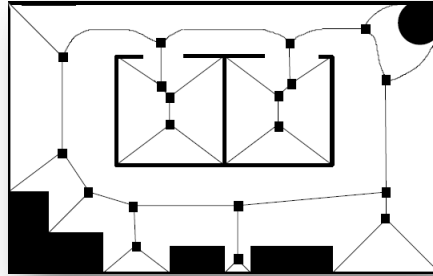


Fig. 37 - Example of a generalized Voronoi graph

This technique has a very big advantage that is the fact of the final representation to be a graph since it turns easier the planning, but also a very big disadvantage that is the fact of being computationally costly to identify the features and calculate the equidistant points between them.

2.6.2.2.2 Topological

As mentioned before, the representation that is generated by this type of mapping method is a topological graph.

A topological graph is composed by nodes and connections, where each node represents an environmental feature or an area and, the connections represent the adjacency between the features/nodes. A connection can only connect two nodes and it represents that it's possible to go from one of the nodes to the other without crossing any other node.

The main idea of this technique is that it's not important to know the absolute position of the robot but in which exact node it is and how it is able of going from one to another node. With this, it means that when a robot uses this method it needs to be able to detect its position in terms of nodes.

In contrast with the metric maps, this mapping method doesn't calculate the absolute position of the robot but the relative position of the map in terms of nodes.

As mentioned before, the nodes represent environmental features and so, the types of features that are considered for the nodes when building a map by using this method depend on these two factors:

- Which features are more relevant for the robot in terms of localization and planning;
- Capability of detection of the robot which depends on the type of sensors it has.

An example of this method is shown on the picture below (Fig. 38), where it is shown an environment that is decomposed into a topological graph. On this situation the type of features considered for the nodes are corners, doors, rooms and corridors.

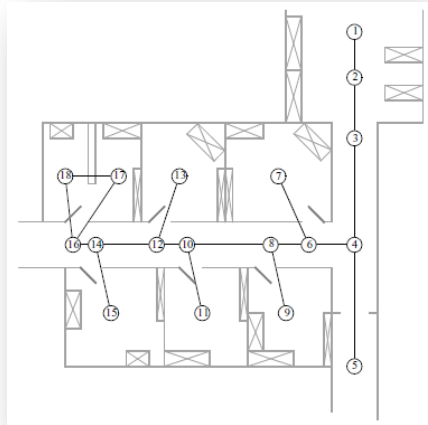


Fig. 38 - Example of a decomposition of an environment into a topological graph

As conclusion about this technique, it's possible to say that it has a big disadvantage that is the fact of not being able to precisely represent the position of the robot but only relative to features, and the big advantage of not being computationally costly since the resulting map is a graph that only depends on environmental features and not on measurements.

2.6.3 SLAM

Simultaneous Localization and Mapping (SLAM), is a technique that concerns with the problem of building a map of an unknown environment by a mobile robot, while at the same time navigating through the environment using that map [Riisgaard05].

On other words, SLAM is a technique which builds a map of an unknown environment or updates a map within a known environment, while navigating through it.

To do so, SLAM works as an iterative process with both mapping and localization together in a way that by sending the feedback of one component to the other, it enhances the results of both components. This means, that both components are separated from each other but are together in a closed loop where they exchange feedbacks with the purpose of enhancing the performance of both.

The main purpose of this technique is then to reduce the error of mapping and localization, which is caused by the sensors errors and noise, by putting both components working on a closed loop to improve the performance of both.

There are many techniques of SLAM. The most known one is Extended Kalman Filter (EKF) which uses the exteroceptive sensors readings to correct the position of the robot. To do so, it re-observes the features from the environment while the robot is moving, in order to get different perspectives and with this update the estimation of the position of the robot to a more accurate one. The features considered by this technique are usually landmarks.

2.6.4 Path Planning

The basic path planning problem refers to determining a path in configuration space between an initial configuration of the robot and a final configuration, such that the robot does not collide with any obstacle in the environment, and that the planned motion is consistent with the kinematic constraints of the vehicle [Dudek00].

In other words, path planning is the component responsible for calculating the path that will make the robot to reach its goal. It's the decision-making component since, given the environmental map and the position of the goal, this component calculates a path, usually the best possible, to arrive to the goal.

To calculate the best path, this component uses search algorithms on the map representation of the environment.

There are two types of algorithms for path planning:

- Discrete state space search algorithms – Calculate the path in a discrete way, which means that the path is decomposed into intermediate goals which the robot needs to reach in order to arrive to the final goal.
- Continuous state space search algorithms – Calculate the path in a continuous way, which means that, as the robot is moving, this type of algorithms calculate the path from its current position to the goal.

On the three next sub sections it is described three different algorithms of these two types of planning. A* and Potential Fields are two discrete state space search algorithms and D* is a continuous state space search algorithm.

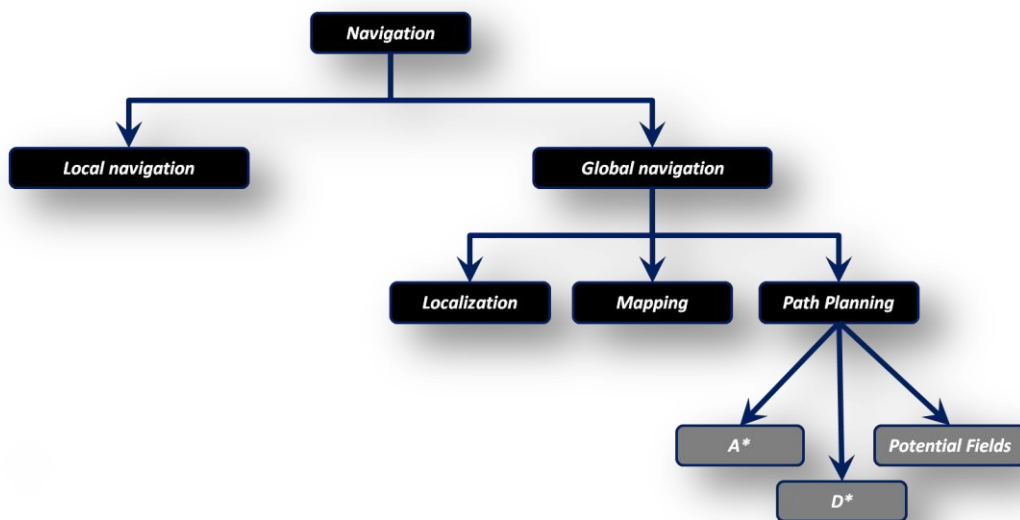


Fig. 39 - Diagram with the three mentioned methods of path planning

2.6.4.1 A* Algorithm

The A* algorithm is an extension of Dijkstra's algorithm, which is a systematic search algorithm that finds the optimal shortest paths from an initial node to all the other nodes in a graph.

Since the A* algorithm is very similar to the Dijkstra's algorithm, before explaining the A*, it will be explained how the Dijkstra's work. To explain it, it will be considered a situation where it is needed to determine a path for a robot from an initial to a final position.

Considering a discrete map of the environment around the robot, the Dijkstra's algorithm sets for each position of the map three fields. The first one shows the status of the position, if visited or unvisited, the second shows the distance that it takes to go from the initial position to that one through the shortest path possible, and the third and last field shows what is the previous position on the shortest path from the initial position to that one.

Before starting the algorithm all these fields are not filled except the status one, which is set as visited for the initial position of the robot and as unvisited for all the other positions of the map.

This algorithm is iterative and at each iteration it follows the same procedure, which is divided in four steps:

1st step: Pick up all the positions that respect the three following conditions:

- Not obstacle;
- Status set as unvisited;
- Have at least one position adjacent to it that has the status set as visited.

2nd step: For each of the considered positions, calculate the shortest path possible from that position to the initial one by only considering positions that are not obstacles and with status set as visited;

3rd step: For each of the considered positions, set the distance and the previous positions fields, with the length of the correspondent shortest path from the initial position that was just calculated and with the previous position that is considered on that shortest path, respectively.

4th step: Go through all the considered positions and pick up the one that has the shortest path, in terms of length, to the initial node and set its status as visited.

This procedure is repeated until all the positions on the map that are not obstacle have their status set as visited.

The result of this algorithm is a map where all the positions that are not obstacle have the shortest path from the initial position of the robot calculated. To know what the trajectory, in terms of positions, of shortest path from the initial position to any other position on the map is, it's needed to check the previous position field of all the positions that are part of the shortest path of that position.

To understand better the Dijkstra's algorithm, consider the situation where there is a map that is a matrix of 6 rows and 6 columns, with all the positions set as not obstacles, and where the initial position is the position (0,0) and the goal position is the position (5,5).

After implementing the Dijkstra's algorithm on the mentioned map, the resulting map with the lengths of the shortest path from each position of the map to the initial position is the one shown on the picture below, Fig. 40.

The positions in green are the positions that compose the shortest path from the goal position to the initial position.

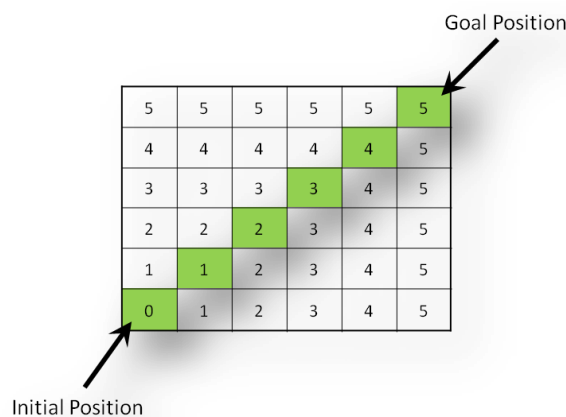


Fig. 40 - Example of an implementation of the Dijkstra's algorithm on a map of 6 rows and 6 columns where all the positions are not obstacles

According to LaValle[LaValle06], the A* is an extension of the Dijkstra's that tries to reduce the total number of states explored by incorporating a heuristic estimate of the cost to get to the goal from a given position.

To do so, this algorithm uses the euclidean distance from the actual to the goal position summed with the length of the shortest path to the initial position, as factor to change the status of a position from not visited to visited. This means, that basically both algorithms are the same except for the 4th step of the iteration procedure.

Considering the same situation that was used to describe the behaviour of the Dijkstra's algorithm, determination of the best path for a robot to go from an initial to a final position, the procedure of each iteration of the A* algorithm looks like this:

1st step: Pick up all the positions that respect the three following conditions:

- Not obstacle;
- Status set as unvisited;
- Have at least one position adjacent to it that has the status set as visited.

2nd step: For each of the considered positions, calculate the shortest path possible from that position to the initial one by only considering positions that are not obstacles and with status set as visited;

3rd step: For each of the considered positions, set the distance and the previous positions fields, with the length of the correspondent shortest path from the initial position that was just calculated and with the previous position that is considered on that shortest path, respectively.

4th step: Go through all the considered positions and pick up the one that has the smallest value from the sum between the euclidean distance from that position to the goal position and the length of the shortest path from that position to the initial position.

While on the Dijkstra's algorithm the procedure of iteration is repeated until all the positions on the map that are not obstacle have their status set as visited, on the algorithm A* the procedure of iteration is repeated until the algorithm arrives to the goal position.

To understand better the A* algorithm, consider the same situation that was used to explain the Dijkstra's, a map that is a matrix of 6 rows and 6 columns, with all the positions set as not obstacles, and where the initial position is the position (0,0) and the goal position is the position (5,5).

After implementing the A* algorithm on the mentioned map, the resulting map with the results from the sum between the length of the shortest path from each position of the map to the initial position and the euclidean distance from that position to the goal position, is the one shown on the picture below, Fig. 41.

The positions in green are the positions that compose the shortest path from the goal position to the initial position.

The positions that have as result of the sum *inf.* are the ones that were not considered by the algorithm when calculating the path, which means that have their status set as not visited. For a position to be considered by the algorithm it needs to not be a obstacle and have at least one neighbour position with status set as visited.

Note that all the positions that have been considered by the algorithm have a sum inside them, where the first value corresponds to the length of the shortest path from that position to the initial one, and the second value to the euclidean distance between that position and the goal one.

At each iteration, the algorithm goes through all the positions that have status set as not visited and at least one neighbour position with status set as visited, and pickup the one with the smallest value from the sum of the two values that it has inside (length from that position to the initial one and euclidean distance from that position to the goal one, respectively).

The selected position later has its status changed to visited and the algorithm goes to the next iteration until the selected position is the goal one.

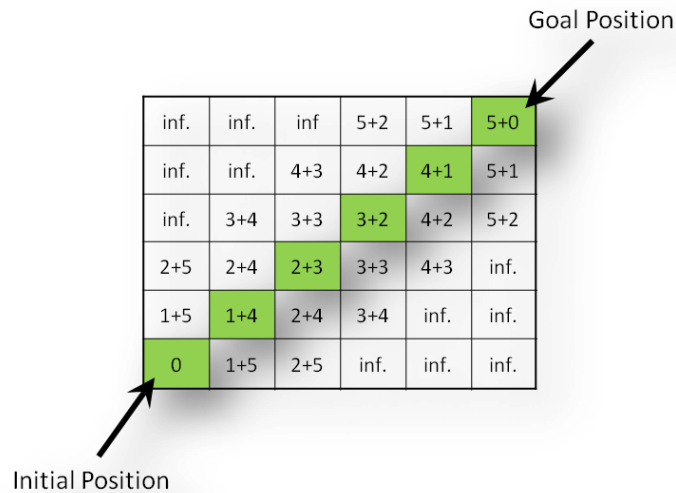


Fig. 41 - Example of an implementation of the A* algorithm on a map of 6 rows and 6 columns where all the positions are not obstacles

Both algorithms, Dijkstra's and A*, work almost the same but, in cases that it is needed to calculate the shortest path from an initial to a final position, the A* has the big advantage of considering less states, which means that less calculations are done and that the complexity of the planning is smaller.

The worst case of the A* in terms of complexity and number of considered states is the Dijkstra's algorithm, which considers all the states possible.

On Fig. 42, it's shown a situation where it's calculated a path from an initial position to a goal position with both the Dijkstra's and A* algorithms. It can be seen that the resulting shortest path got by both algorithms have the same length but A* considered less positions when determining it.

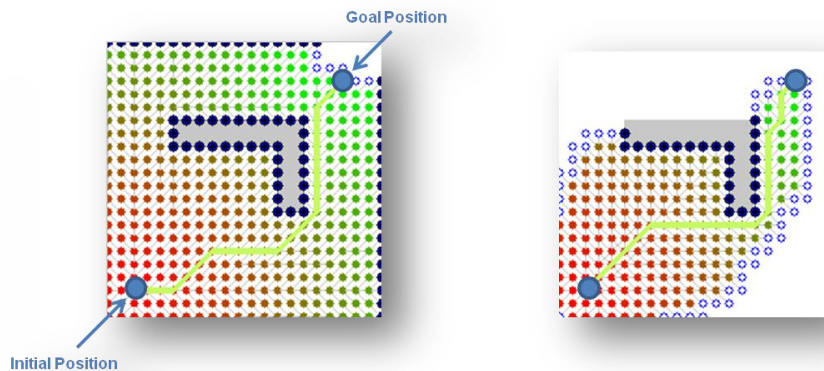


Fig. 42 - Result of a calculation of the shortest path from an initial to a goal position by using the Dijkstra's algorithm (picture on the left) and the A* algorithm (picture on the right)

2.6.4.2 D* Algorithm

The D* algorithm (Dynamic A*) plans optimal traverses in real-time by incrementally repairing paths to the robot's state as new information is discovered. [Stentz95].

When a robot is moving while using this algorithm, it may discover that some parts were easier to traverse than it originally thought. In other cases, it might realize that some

direction it was intending to go is impassable due to a large bolder or a ravine. If the goal is to arrive at some specified coordinates, this problem can be viewed as a navigation problem in an unknown environment [LaValle06].

In summary, this algorithm works similar to the A*, but with more features. When working with the list of positions in order to select the next position to be set as visited, this algorithm has two more states, RAISE and LOWER.

RAISE is when the cost of a path increases due to a position that was known as free space to be detected as an obstacle and LOWER is when a position that was known as an obstacle to be detected as free space, which can make the cost of a path to decrease.

These states propagate the position increase or decrease through all the position that in the region of the position so that the algorithm can compute new optimal paths.

To get more details about the way this algorithm works, check the documents referred here as [Stentz95] and [LaValle06].

2.6.4.3 Potential fields

Potential field path planning is an algorithm that creates a potential field across the map of the environment, which is able to move the robot towards the goal position from all the other positions of the map, if there is a possible path between that position and the goal one.

To do so, on a potential field the goal position acts as an attractive force and the obstacles as repulsive forces. The sum of all the forces is applied on the robot so that, the robot moves towards the goal position while avoiding the obstacles that are represented on the map.

The basic idea behind the potential field algorithm is that the robot is attracted towards the goal, while being repulsed by obstacles that are known in advance. If new obstacles appear while the robot is moving, the robot can update its potential field in order to integrate this new information [Siegwart04].

In other words, this algorithm works like a charged particle navigating through a marble rolling down a hill. The difference is that the behaviour of the marble depends on the shape of the hill, while the potential fields depends on the environment around the robot, obstacles and free space.

According to Goodrich, the designer of a potential field has three tasks:

1. Create multiple behaviors, each assigned a particular task or function;
2. Represents each of those behaviors as a potential field, attractive or repulsive;
3. Combine all those behaviors to produce the robot's motion.

Each potential field consists on action vectors, one per position of the map, which correspond to the speed and orientation of the robot on that position in order to act according to the potential field. The orientation and length of the action vector corresponds to the orientation and speed that the robot must have to act according to the behaviour of the potential field on that position.

The two most known behaviours in terms of potential fields are attractive and repulsive.

Starting by the attractive potential field, it is composed by vectors that point the robot towards the goal while moving on it. Its behaviour causes the robot to be attracted towards the goal.

An example of an attractive field is shown on Fig.43, where the goal is the blue circle on the middle and all the action vectors point towards the goal.

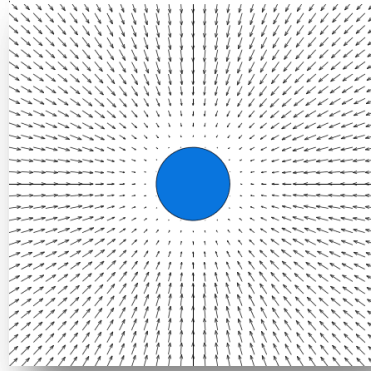


Fig. 43 - Example of an attractive potential field

Note that, all the actions vectors point to the goal according to the orientation of its positions according to the goal but also, that the length of the action vectors gets smaller as getting closer to the goal so it doesn't collide.

Now, about the repulsive potential field, it is composed by vectors that point the robot away from the obstacle while moving on it. Its behaviour causes the robot to be repulsed by the obstacle.

An example of a repulsive field is shown on Fig.44, where the obstacle is the pink circle on the middle and all the action vectors point away from the obstacle.

Note that, all the actions vectors point away from the obstacle according to the orientation of its positions according to the obstacle but also, that the length of the action vectors gets smaller as getting away from the obstacle since the robot is more assured that while avoid the collision with it.

On the obstacle, the actions vector has infinite length.

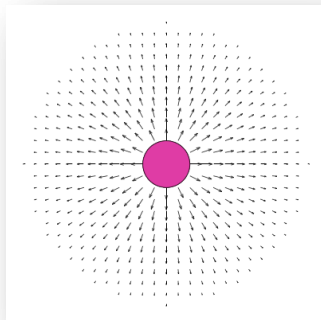


Fig. 44 - Example of a repulsive potential field

Note that, all the actions vectors point away from the obstacle according to the orientation of its positions according to the obstacle but also, that the length of the action vectors gets smaller as getting away from the obstacle since the robot is more assured that while avoid the collision with it.

On the obstacle, the actions vector has infinite length.

The main feature of this algorithm, potential fields, is the combination between different behaviours, potential fields. This is done by adding the action vectors of all the potential fields considered on each position of the map.

An example of a potential field generated by combining an attractive and a repulsive potential fields is shown on Fig.45, where the goal is the blue circle on the top right corner and the obstacle is the brown circle on the bottom left corner.

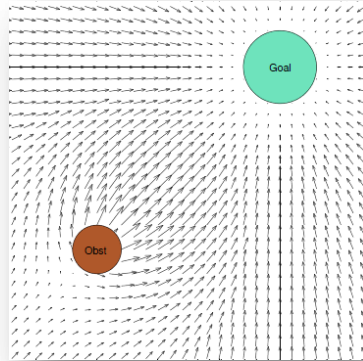


Fig. 45 - Example of a potential field that was generated by combining an attractive and a repulsive potential fields

To make this combination, the corresponding action vector of both potential fields, attractive and repulsive, of each position of the map are summed so that when the robot is moving through this potential field, it is able to avoid the collision with the obstacle and at the same time move towards the goal.

The resulting path that the robot will take when moving through the potential field generated by the combination of both behaviours is shown on Fig. 46, where the goal is the pink circle, the obstacle is the light blue circle and the path is the black line.

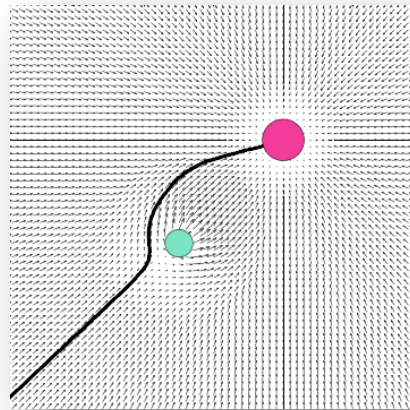


Fig. 46 - Path that results from the robot moving through a potential field that is a combination of an attractive and a repulsive potential fields

There are many other behaviours that can be implemented to potential fields. The five most used ones are the already mentioned attractive and repulsive and the perpendicular, uniform/parallel and tangential ones. An example of each of this five mentioned potential fields is shown on Fig. 47.

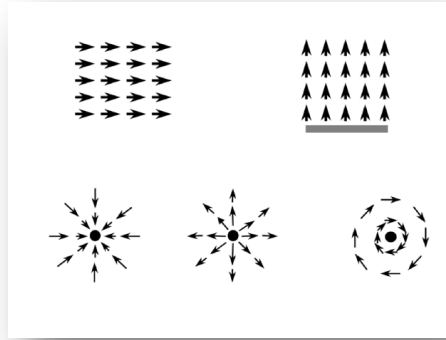


Fig. 47 - The five most used behaviours for potential fields (From the left top corner to the bottom right: uniform/parallel, perpendicular, attractive, repulsive, tangential)

As conclusion for this algorithm, a big disadvantage it has is the local minima, which happens when the sum of all the actions vectors on some position is 0, which makes the robot to stop on that position and cancel the behaviour of all the potential fields acting on the robot.

2.7 Motion Control

In the structure of an autonomous mobile robot, motion control is the component responsible for translating the path given by the planning component into motor inputs, in a way that the robot achieves what is desired by following the path provided by the navigation component.

Apart from translating the given path into motor inputs, this component also takes into account the robot kinematics when calculating the commands to send to the motors, in a way that the mechanic constraints of the robot are also respected.

The motor inputs considered in robotics are usually the translational and angular speed, v and ω , respectively, and two examples of mechanic constraints that are usually considered in mobile robots are the type of wheels and the size of the robot.

With all this that was mentioned, it can be concluded that the motion control component of an AMR includes two competences:

- **Path Execution** – The real-time translation of the path into an horizontal and a vertical displacements, which depends on the actual robot position and on the algorithm used;
- **Acting** – The calculation of the motor inputs based on the horizontal and vertical displacements provided by the path execution competence, and on the robot kinematics.

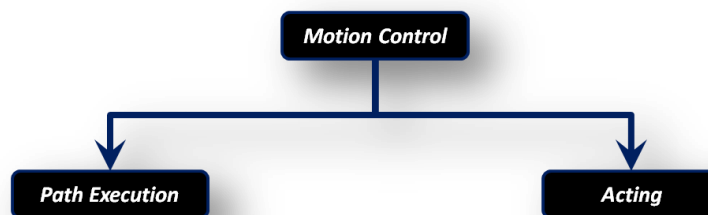


Fig. 48 - The two competences of the motion control component of an autonomous mobile robot

These two competences and one example of each are explained in detail on the following two sub sections, 2.7.1 and 2.7.2, respectively.

2.7.1 Path Execution

Path execution is responsible for the real-time translation of the path into an horizontal and a vertical displacements.

This two displacements are calculated by an algorithm that does so by taking into account the robot actual position and the map positions mentioned on the path.

This is done so that the robot, instead of executing the set of positions mentioned on the path one by one, executes all of them in a continuously way without needing to go strictly through all of them neither to stop on all of them.

The level of strictness that the path execution algorithm uses to calculate the displacement is provided by the user in a way that, if the user sees that the environment has almost no obstacles the level of strictness can be low but, if the environment has a lot of obstacles the level of strictness already needs to be high so the robot respects more the positions mentioned on the path making it not to collide with the obstacles.

An example of a path execution algorithm is the pure pursuit algorithm, which is explained in detail on the next sub section, 2.7.1.1.

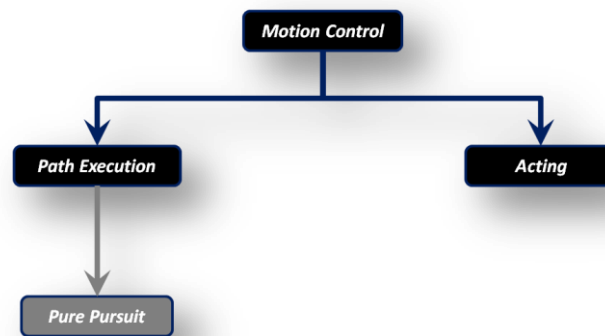


Fig. 49 - The mentioned algorithm of path execution, Pure Pursuit

2.7.1.1 Pure Pursuit Algorithm

According to Coulter [Coulter92], pure pursuit is a tracking algorithm that works by calculating the curvature that will move a vehicle from its current position to some goal position.

The basic idea of this algorithm is to choose a position that is some distance ahead of the robot on the path. The name pure pursuit comes from the fact that when a robot is using this algorithm, it is always pursuing a point on the path that is some distance ahead from it.

The distance considered by this algorithm to choose the positions is called lookahead distance and it is set by the user of the robot in order to control the level of strictness that the robot uses to respect the path.

This algorithm receives as input a path and the real-time robot position and as output it gives an horizontal and a vertical displacement which is sent to the actuator.

It is an iterative algorithm and at each iteration it follows the same procedure, which can be divided in four steps:

Before starting the explanation of each of the steps in detail there are two things that need to be taken into account:

- On this explanation, instead of considering a curvature when calculating the displacements, it is considered the value of the euclidean distance between the position of the robot and the positions on the path;
- On the first iteration of this algorithm, the variable *Position2* points to the first position on the path.

1st step: This step consists on the calculation of the euclidean distance D from the real-time robot position, which is here considered as *Position1*, to the position of the path that is pointed by the variable *Position2*, by applying the following equation:

$$D = \sqrt{(\text{Position2}(x) - \text{Position1}(x))^2 + (\text{Position2}(y) - \text{Position1}(y))^2}$$

2nd step: This step consists on checking if the sum of all the values of euclidean distances D calculated until now on this iteration, is smaller or bigger than the predefined lookahead distance, and act according to it:

- If it is smaller, it is needed to consider the next position on the path. To do so, it is needed to calculate the euclidean distance D between the path position that was considered as *Position2* on the last calculation of D, which becomes now considered as *Position1* on this calculation, and the next position on the path, which becomes now considered on this calculation as *Position2*.

The result of this calculation of D is summed with all the other values of euclidean distances D calculated until now on this iteration.

If the value of that sum is still smaller than the value of the predefined lookahead distance, repeat this step.

- If it is bigger, go to the next step.

3rd step: This step consists on the calculation of the displacement needed to do to do to make the lookahead position to be with a distance equal to the predefined lookahead distance from the actual robot position.

To do so, first it is needed to calculate the angle between the two positions that were considered as *Position1* and *Position2* on the last calculation of D.

$$\text{Angle} = \tan^{-1} \frac{(\text{Position2}(y) - \text{Position1}(y))}{(\text{Position2}(x) - \text{Position1}(x))}$$

Then, by applying the equations that are used to convert from polar to trigonometric coordinates, we are able to get the displacement needed to do on both axis, vertical and horizontal, from the position considered on the last calculation of D as *Position1*.

This is done by multiplying the distance that is needed to make the exact displacement from the position considered as *Position1* on the last calculation of D equal to the predefined lookahead distance, by the sine and cosine of the calculated angle.

To get the needed distance, *DisNeeded*, it is needed to apply the following equation:

$$\text{DisNeeded} = \text{LookAhead distance} - (\text{Sum of all Ds} - \text{Last D})$$

To get the needed vertical and horizontal displacements, it is needed to apply the two following equations:

$$\text{Displacement needed on vertical axis} = \text{DisNeeded} * \cos(\text{Angle})$$

$$\text{Displacement needed on horizontal axis} = \text{DisNeeded} * \sin(\text{Angle})$$

4th step: This step consists on getting the coordinates of the lookahead position, which is the position from the path that has as value of euclidean distance from the robot actual position equal to the predefined lookahead distance.

To do so, it is needed to sum the vertical and horizontal displacements calculated previously to the vertical and horizontal coordinates, respectively, of the position considered as *Position1* on the last calculation of D, like shown on the equations below:

$$\text{LookAhead Position}(x) = \text{Position1}(x) + \text{Displacement needed on horizontal axis}$$

$$\text{LookAhead Position}(y) = \text{Position1}(y) + \text{Displacement needed on vertical axis}$$

Finally, these are the two displacements, horizontal and vertical, that will be sent to the actuator, and the algorithm goes to the next iteration.

Note that, on the next iteration, the variable *Position2* points to same position that was pointed by that variable on the last calculation of D done on this iteration.

This algorithm is executed until the last position on path is considered as *Position1* for a calculation of D. When this happens, the algorithm finishes and sends as vertical and horizontal displacements to the actuator, the exact vertical and horizontal displacements from the robot actual position to the final path position, since there are no more positions on the path to consider.

To understand better the pure pursuit algorithm, consider the situation that was used to explain the Dijkstra's algorithm on section 2.6.4.1, where there is a map that is a matrix of 6 rows and 6 columns, with all the positions set as not obstacles, where the initial position is the position (0; 0) and the goal position is the position (5; 5), and where the shortest path from the initial to the goal positions that results from the implementation from the Dijkstra's algorithm on the given map are the positions in green (Fig. 50).

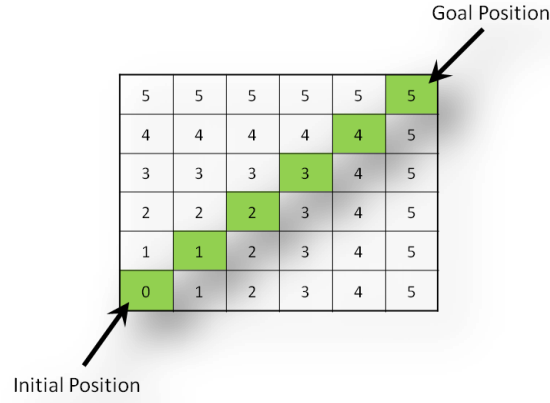


Fig. 50 - Example of an implementation of the Dijkstra's algorithm on a map of 6 rows and 6 columns where all the positions are not obstacles and where the positions in green corresponds to the shortest path from the position (0; 0) to the position (5; 5)

Considering then, that the robot is at the position (0; 0), the next position on the path is the position (1; 1) and that the last position on the path is the position (5; 5), by applying the pure pursuit algorithm with a lookahead distance of 2, the lookahead position for the first iteration would be calculated by the doing the following way:

1st step: Calculation of D, considering that the actual robot position is the position (0; 0) and the first position on the path is the position (1; 1):

$$D = \sqrt{(1 - 0)^2 + (1 - 0)^2} = \sqrt{2} = 1,4$$

2nd step: The sum of all the values D calculated until now on this iteration, which is only one and is equal to 1,4, is smaller than the predefined value of lookahead distance, 2, so, it is needed to consider the position that was considered as *Position2* on the last calculation of D, the position (1; 1), as *Position1* and, to consider the next position on the path, the position (2; 2), as *Position2*, and calculate the euclidean distance between them.

$$D = \sqrt{(2 - 1)^2 + (2 - 1)^2} = \sqrt{2} = 1,4$$

By summing this value of D with the ones previously calculated on this iteration, which is only one and has as value 1,4, the sums of all the distances D is equal to 2,8.

This value is bigger than the predefined lookahead distance, so we can already go to the next step.

3rd step: The two positions considered as *Position1* and *Position2* on the last calculation of D are the positions (1; 1) and (2; 2), respectively, and so the angle between them is:

$$\tan^{-1} \frac{(2 - 1)}{(2 - 1)} = 45^\circ$$

The displacement needed to make the exact displacement from the position considered as *Position1* on the last calculation of D, the position (1; 1), equal to the predefined lookahead distance is:

$$\text{DisNeeded} = 2 - (2,8 - 1,4) = 0,6$$

Finally, the vertical and horizontal displacements needed from the position considered as *Position1* on the last calculation of D, the position (1; 1), to make the lookahead position to be with an euclidean distance from the robot actual position equal to the predefined lookahead distance are:

$$\text{Displacement needed on vertical axis} = 0,6 * \cos(45^\circ) = 0,42$$

$$\text{Displacement needed on horizontal axis} = 0,6 * \sin(45^\circ) = 0,42$$

4th step: The position considered as *Position1* on the last calculation of D is the position (1; 1) and the horizontal and vertical displacements needed are 0,42 both.

Then, the position on the path that has euclidean distance equal to 2 from the robot actual position is the position (1,42; 1,42), which corresponds to the two displacements that will be sent to the actuator

On the next iteration, the variable *Position2* would keep pointing to same position that was pointed by that variable on the last calculation of D done on this iteration, the position (2; 2).

The algorithm would be executed until the position (5; 5), last position on the path, would be considered as *Position1* for the calculation of D.

2.7.2 Acting

Acting is responsible for the calculation of the motor inputs based on the horizontal and vertical displacements provided by the path execution competence, and on the robot kinematics.

2.7.2.1 Siegwart Equations

The algorithm explained on this section correspond to the control equations explained by Siegwart on the chapter 3.6 of the book referred here as [Siegwart04].

The mentioned control equations implement a real-state feed-back controller, where the inputs are the actual robot position and the goal position, and the outputs are the calculated translational and angular velocities

The situation considered by these equations to calculate the translational and angular velocities is shown on the image below, Fig. 51, which refers to a robot with an arbitrary position and orientation and a predefined goal position and orientation.

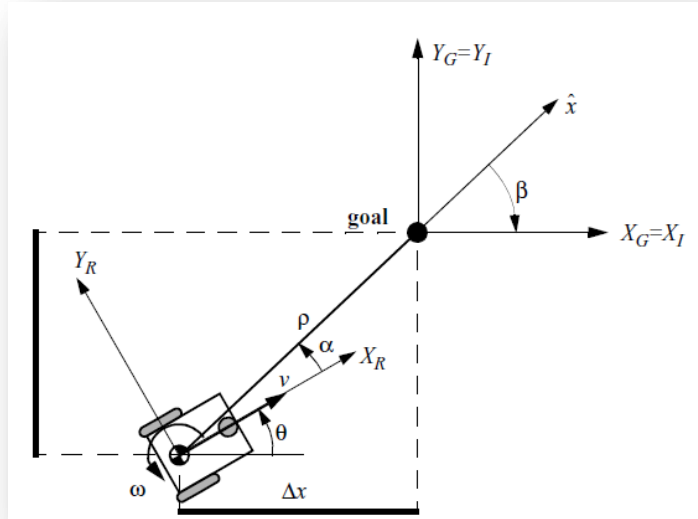


Fig. 51 - Robot kinematics and its frame of interests [Siegwart04]

In order to explain the equations, considering the following variables:

- X_R , Y_R and θ_R as the coordinates of the actual position of the robot;
- X_G , Y_G and θ_G as the coordinates of the goal position;
- ρ as the distance from the robot actual position to the goal position.
- α as the difference between the actual robot angle and the angle at which is the goal according to the robot actual position;
- β as the difference between the angle at which is the goal according to the robot actual position and the goal angle;
- v and ω as the translational and angular velocities of the robot.

The explanation of how to calculate the robot velocities by using the mentioned equations is divided in four steps:

1st step – Determine the distance ρ , which is done by applying the Pythagoras' theorem:

$$\rho = \sqrt{(\Delta X)^2 + (\Delta Y)^2}$$

where ΔX corresponds to the difference between the actual robot X coordinate and the goal X coordinate:

$$\Delta X = X_R - X_G$$

and ΔY to the difference between the actual robot Y coordinate and the goal Y coordinate:

$$\Delta Y = Y_R - Y_G$$

2nd step – Determine the angle α , which is done by the following way:

$$\alpha = \left(\tan^{-1} \left(\frac{\Delta Y}{\Delta X} \right) \right) - \theta_R$$

where ΔX and ΔY correspond to the same values that were calculated on the first step of this calculation.

3rd step – Determine the angle β , which is done by the following way:

$$\beta = -\theta_R - \alpha$$

4th step – Calculate the velocities of the robot by applying the values determined on the previous three steps on the equations below:

$$v = k_\rho * \rho$$

$$\omega = k_\alpha * \alpha + k_\beta * \beta$$

where k_ρ , k_α and k_β correspond to the control parameters, which need to be calculated according to the robot kinematics.

In order for the control system to be stable, these three control parameters need to respect the following rules:

- $k_\rho > 0$;
- $k_\beta < 0$;
- $k_\alpha - k_\rho > 0$.

As conclusion for this algorithm, there are two things that need to be taken into account when using these equations.

The first one is that the control parameters mentioned on the last step need to be calculated according the kinematics of the robot, which can be done by determining its control laws or by testing different values and watch the behaviour of the robot.

The second one is that the units of the calculated speeds depend on the units of the inputs of the equation.

3 Software Engineering for Robotics

In order to implement the different subsystems of an AMR it is needed to use a defined paradigm of implementation. Like mentioned before on section 2.2, a paradigm is a set of assumptions and/or techniques which characterize an approach to a class of problems.

In this situation, the problem is how to program the different subsystems of an AMR, and the paradigms are the different methods of programming that can be used to solve that problem.

The most known programming methods/paradigms are:

- **Imperative Programming** – Describes how the program should do what it is supposed to do in terms of sequences of actions that change the program state. Basically, this paradigm of programming describes how to do something in terms of sequences of actions for the computer to perform.

When using this programming paradigm, the same sequence of actions can result in different values at different times, depending on the initial state of the program.

C, *Basic* and *Pascal* are examples of imperative programming languages.

- **Declarative Programming** – Describes what the program should do without describing how to do it in terms of changes on the program state.

There are two types of declarative programming.

- **Functional Programming** – Describes what the program should do in terms of mathematical functions.

Functions are a set of commands that receive some arguments as input, process them and as output puts the result of that process.

When using this programming paradigm, the output value of the functions only depends on the input arguments, which means that calling a function at different times with the same arguments as input will produce the same output.

Haskell and *Scheme* are two examples of functional programming languages.

- **Logic Programming** – Describes what the program should do in terms of mathematical logic.

Prolog is an example of logic programming languages.

- **Object-Oriented Programming** – Describes a program or an application in terms of objects, which are data structures that have data fields and methods, and their interactions.

C++, *Java* and *Smalltalk* are examples of object-oriented programming languages.

- **Component-Based Programming** – Describes a program or an application in terms of components with well-defined communication interfaces, and the communication links between those components.

Components produce and consume events, which are changes on their states.

UML and *Smartsoft* are two examples of component-based programming languages.

- **Model-Driven Engineering** – Describes a program or an application in terms of models.

Models are representations of the real world that only show the aspects that are of interest to the application.

There is not a paradigm that can solve all the problems in an easy and efficient way.

Also, there are some programming languages that use more than one paradigm. Two examples of that are *C++* and *Visual Basic*, which are programming languages that combine the imperative and the object-oriented programming paradigms.

All the programming languages are based on one or more paradigms of programming with the intention that the developers of applications can choose the most adequate paradigm according to its situation.

To implement the different subsystems of an AMR based on any of the mentioned programming paradigms it is needed to use software frameworks, which are reusable software platforms used to develop and implement applications.

Software frameworks include support programs, compilers, code libraries and an application programming interface.

On the following sections is explained in detail the state of art in terms of software frameworks for robotics.

3.1 Object-Oriented Frameworks

The paradigm of Object-Oriented Programming created new possibilities about the concept of reusing modules of programming, due to the production of generic modules, which are easy to integrate, distributed and independent of the platform where they were developed.

On the area of robotics, there are libraries that are used by some subroutines with the intention of maximizing the reuse of software.

A **library** is a group of programming resources, such as data structures, classes, functions, etc., that everyone can use in their own program.

By including a library in a program, the user is increasing the group of resources that he has access to by just adding some new components, which have no underlying logic or hierarchy.

The use of frameworks of programming is one more step in the reuse of software.

A **framework** defines a standard, in terms of concepts, applications and criteria, which is used to solve a problem in particular, but that can also work as reference when it comes up to solve similar problems.

In general, a framework uses as support, programs, libraries and a programming language, in order to develop the different components of a project. It also represents a software architecture that defines the relationships between all the entities of its domain and provides a structure and a working methodology, which extends or uses its applications.

The biggest advantages of using a framework are the cost reductions in the development of software applications for specific domains, and the increase on the quality of the final product (Fayad and Schmidt, 1997).

According to Gamma [Gamma95], the framework defines the architecture of the application. This is an interesting approach since the framework is the one that defines the general structure, in terms of classes and objects, the role of each of them and also the relationships between them.

All these parameters are defined by the framework so that the user doesn't need to worry about them and just focus his attention on the details of his application.

Also according to Gamma [Gamma95], the framework includes the design decisions that are common to the application domain, which means that the framework promotes not only the reuse of code but also of the design.

A library and a framework are made for different types of users. A library can be used by anyone that needs some type of objects or algorithms that it provides as resources, while a framework is used by anyone that wants to develop an application in the domain of the framework. Another way to define both concepts is to think about a library as a big packet of pieces (resources), which the user puts together and makes a puzzle (its application), existing many ways of doing it.

On the other hand, a framework starts by identifying the parts of the application that vary from another of the same domain (*Hot Spots*), and after defining its architecture and interface, it puts together the pieces of the puzzle in a way that the user only needs to worry about the implementation of the *Hot Spots*, which are the pieces where it is defined the behaviour of the application.

When using a framework to develop an application, the user is using its architecture, which means that the same architecture is re-used from one application to another.

It is very important to note that when someone wants to use a framework it is required to know in detail the architecture of the framework, which means that it is needed to read in detail the documentation about the framework.

The use of the framework is recommended to someone that wants to develop software in a specific area, since the design solves almost all the problems of the application that the user wants to create.

Once it's clear what is a framework and its advantages when it comes to build an application, it is time to describe which types of frameworks there are.

There are two types of frameworks:

- **Vertical Frameworks** – Frameworks that are developed for a specific domain of an application.
This type of frameworks covers many domains such as telecommunications, manufacturing, advanced telematic services, etc.
The main problem of this type of frameworks is that they are hard to develop since they are very specific and so require very precise knowledge in the domain of the wanted application.
- **Horizontal Frameworks** – Frameworks that are developed not for a specific domain of an application, which means that they can be used in different domains.
Examples of this type of frameworks are communications infrastructures user interfaces and visual environments.

As example of a object-oriented framework there is ACE, which is used to simplify network programming.

Adaptive Communication Environment (ACE), is an open source object-oriented framework, which implements many of the most important features used in network programming.

ACE is independent of the platform, which means that the same code will work on most operating systems.

ACE is implemented in C++ and it is oriented to applications developers and to network and real time high performance services. ACE separates the operative system from the application by using a layer called Adaptation to the Operating System.

In other words, ACE is a socket implementation which is independent from the final platform of execution (the sockets are used in different ways in Linux and Windows). Apart from the fact that ACE provides a different implementation for each platform, the way to use ACE is always the same.

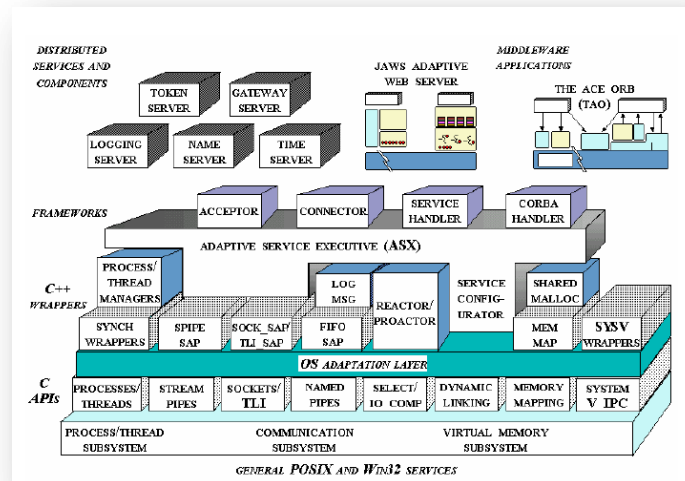


Fig. 52 - ACE and its components

ACE was initially developed by Douglas C. Schmidt during his graduate work at the University of California, Irvine. Nowadays, most of the development work on ACE is done on the Institute for Software Integrated Systems (ISIS) at Vanderbilt University.

Because of the need of integrating the networks on the frameworks, it was developed the Middleware Application Frameworks, which was designed to integrate modules and software applications on distributed environments, allowing then a high level of modularity and reutilization when developing new applications, and at the same time isolating almost all the technical and conceptual difficulties that appear when developing a distributed application.

Middleware is a standard communication protocol which allows the communication between different software applications. With this it means that when developing software applications, middleware must be implemented so that it makes it easier for software developers to perform communication between different applications. Middleware is usually described as *software glue*.

With middleware, applications that are being ran on different or on the same processors can communicate between them.

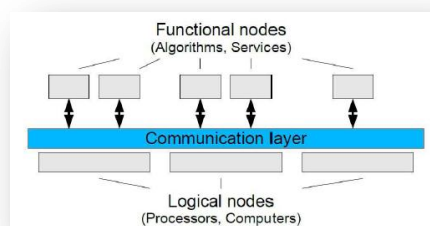


Fig. 53 - Communication layer between software applications (Middleware)

Considering all that was mentioned until now, it is time to speak about **CORBA (Common Object Request Broker Architecture)**, which is a standard defined by the Object Management Group (OMG) that enables different software applications written in different languages and running on different computers to work with each other like a single application or set of services.

To be able to do so, CORBA, for each piece of code written in a different language, creates a packet that contains additional information about the capabilities of that piece of code and also about how to call its methods.

In order to specify the interfaces that each object offers, CORBA uses a interface definition language (IDL). Since each object can be implemented in a different language, CORBA also specifies a kind of mapping/translation from IDL to the implementation language.

On the image below , Fig. 54 , it is shown the model that is used by CORBA and OMG to define an architecture of an application.

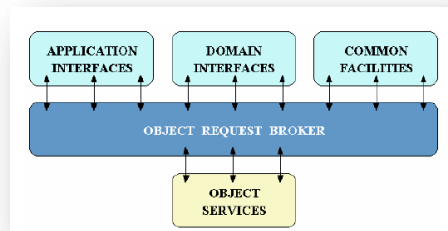


Fig. 54 - Model used by OMG to define an architecture

Object Request Broker (ORB) is the software layer that allows each object to interact with another objects that can be situated on different machines. On other words, it is middleware software that controls the transfer of data between different objects in a way that the data is understandable by both of them.

To be able to do so, ORB uses a standard of transformation/translation of data structures from a byte sequence, conserving the order of the bytes on that sequence, which is called marshalling. The opposite action is called unmarshalling.

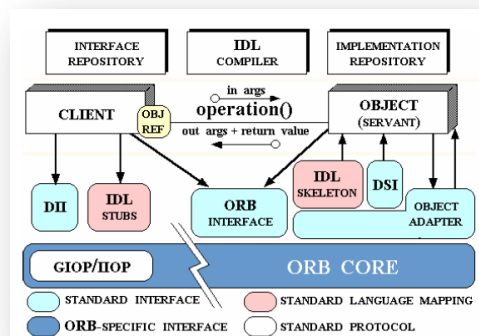


Fig. 55 - CORBA ORB structure

Like mentioned before, CORBA is a standard, from which there are many different implementations. On the next paragraphs, it is explained two different implementations of CORBA, TAO and ICE.

TAO (The ACE ORB) is an open source implementation of CORBA that uses ACE to access the sockets. Its goal is to provide high-performance and efficient QoS (Quality of Service) to real time distributed applications.

TAO allows its clients to work with different objects without needing to worry about the localizations of those objects as well as the language it is programmed on, the operative system it uses and the communication protocols it uses.

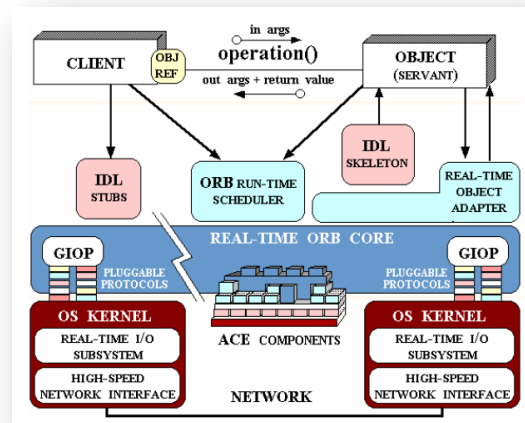


Fig. 56 - The Ace Orb (TAO) structure

ICE (Internet Communications Engine) is an object-oriented middleware that can be used for internet applications without the need to use the HTTP protocol and is also capable of going through firewalls unlike most of the other middleware.

ICE is a set of CORBA that includes components for object-oriented remote-invocation, replication, grid-computing, failover, load-balancing, firewall-traversals and publish-subscribe services. To access to those services, applications need to be linked to a library called slice.

ICE has been developed by ZeroC and it is compatible with C++, Java, C#, Python, PHP, etc. on most major operating systems such as Linux, Solaris, Windows and Mac OS X.

3.1.1 Robotic Object-Oriented Frameworks and Middleware

On this section it will be described some examples of object-oriented frameworks and middleware that were designed specifically to work on the field of robotics.

3.1.1.1 Player (<http://playerstage.sourceforge.net/>)

Player is an open-source middleware network server for robot control, which provides an abstract interface to robot devices, such as mobile robot bases, sensors, etc.

By running player on a robot, Player provides a clean and simple interface to the robot's sensors and actuators over the IP network. To do so, Player communicates with the robot devices by using the devices' drivers but provides to its clients a standard device interface.

The client program communicates with Player over a TCP socket, reading data from sensors, writing commands to actuators and configuring devices of the robot.

The client program can be run on any machine that has a network connection with the robot, and it can be written in any language that supports TCP sockets.

With this it means that Player makes no assumptions about the structure of the robot control programs, being then independent of the platform and language.

In terms of robot hardware, Player is compatible with a big variety. The original Player platform is the ActiveMedia Pioneer2 family, but several other robots and sensors are supported.

Player's modular architecture makes it very easy to add support for new hardware, which makes it possible for developer communities to contribute with new drivers.

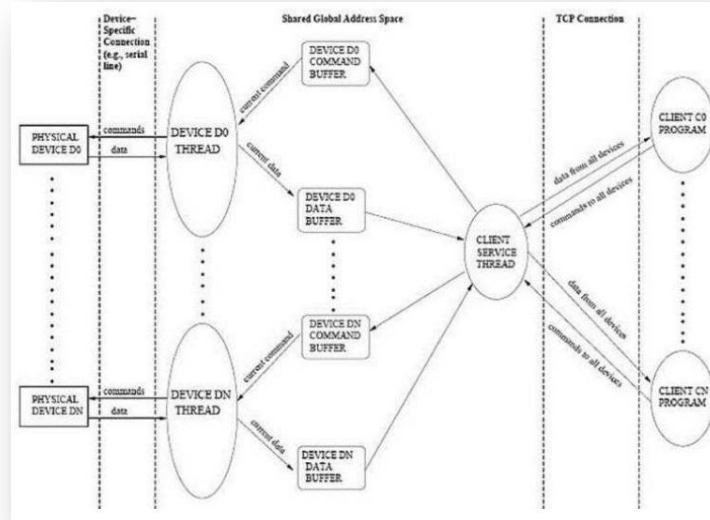


Fig. 57 - Threads distribution inside Player

3.1.1.2 Webots API (<http://www.cyberbotics.com/>)

Webots is an open-source development environment used to model, program and simulate mobile robots.

This application is organized as a programmable object-oriented interface in order to control the movements of the robot. The objects correspond to devices of the robot such as wheels, camera, sensors, etc.

For each object there are methods, which can be simple like activating or deactivating the devices or complex as processing the readings of that device and sending it to another object.

On the image below, Fig. 58, it shown a small example of how the devices and robot could be structured when using Webots.

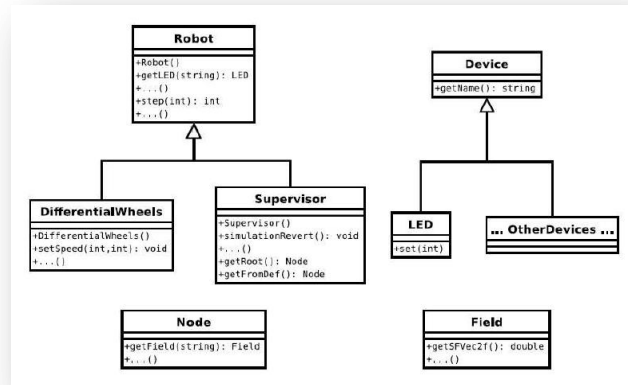


Fig. 58 - Structure of a small example of how the devices and robot could be structure with Webots

This object-oriented API is available in different languages, including C, C++, Java and Python. Webots API also has a TCP/IP interface (programmed in C) that allows the user to program its robot from another program that is compatible with TCP/IP and still use the Webots API (ex. Matlab with the toolbox of TCP/IP, Labview, Lisp, etc.).

On Fig. 59 it is shown how Webots API is structured to work with different programs.

The communication protocol used between the API and the robot uses three different technologies, which depend on what the user uses the API Webots for, if for simulation, remote control or on a compiled application.

Webots API includes a big set of sensors and actuators frequently used in robotic experiments. Still, Webots API is totally documented, making it possible to easily adapt it to new robots and devices.

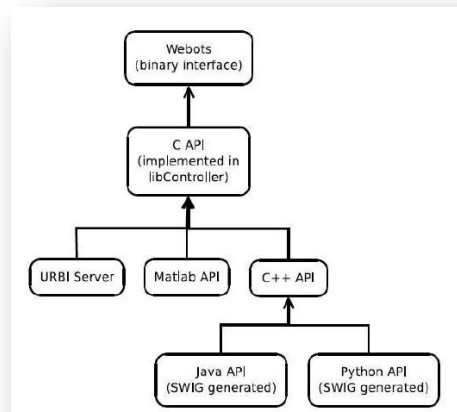


Fig. 59 - Structure of Webots API

3.1.1.3 ARIA (<http://robots.mobilerobots.com/wiki/ARIA>)

Advanced Robot Interface for Applications (ARIA) is a C++ library for all MobileRobots/ActivMedia platforms.

It includes a library called ARNetworking, which implements an extensible infrastructure for easy remote network operations for your robots, user interfaces, and other networked services.

ARNetworking provides the critical layer for the TCP/IP based communication between the robot and the clients that are connected to the same network. ARNetworking is responsible for the low-level interaction between the client and the server, such as, the commands packet processing, multithreading, etc.

ARIA is supported by Linux and Win32 OS, which means that an application that is developed on its API can work in robots with one of these two operating systems.

On the low-level, ARIA works on a client-server environment. The robot microcontroller is the server, and it establishes a dialog through its serial port with the application that was developed on ARIA API, which is client.

It is on that dialog client-server, that the ultrasonic, laser and odometry readings are sent to the client, and also, where the commands for the motors are sent to the server.

In terms of the hardware access, ARIA offers a group of classes to configure the API, which are shown on the image below, Fig. 60.

Comparing with other object-oriented frameworks, ARIA objects are not distributed, since they are all on the machine that connects to the robot.

Indeed, ARIA allows the user to program distributed application by using ARNetworking, which controls the remote communications.

About multitasking, ARIA provides a great flexibility since the applications can be programmed as one thread or multithread.

ARIA contains basic navigation behaviours like obstacle avoidance, but it doesn't include other functionalities, such as localization or map building.

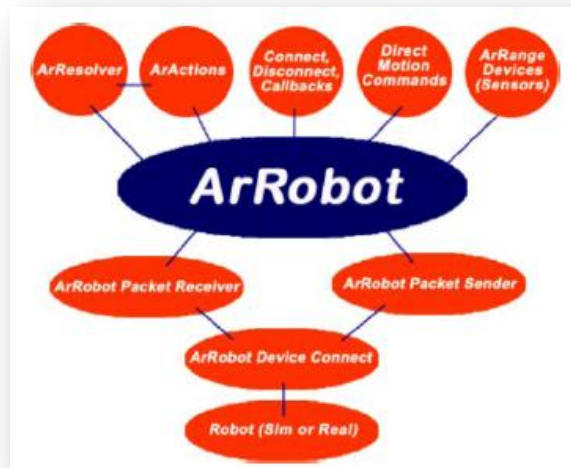


Fig. 60 - Structure of ARIA API

3.2 Component-Based Frameworks

The paradigm of Object-Oriented Programming (OOP) was the support of software engineering during many years. Indeed, it was shown that it was not possible to apply its techniques on the development of applications for extendable environments.

On other words, the OOP:

- Doesn't mention the difference between the computational and compositional aspects of an application;
- Uses an object-vision instead of a modular one, where the modules correspond to compositional units that are application independent;
- Doesn't take into account the real world needs, like distribution, acquisition and incorporation of those modules on the systems.

Because of these reasons and by going higher on the level of abstraction, it was born the Component-Based Programming (CBP) paradigm.

The main goal of this new paradigm was to build a global market of software components, where the users are application developers that need to use components that already exist, in order to build their own application in a faster and more robust way.

On other words, this paradigm purposed the development and utilization of reusable components in a global software market. On the other hand, the availability of components is not enough if we are not able to reuse it, and the reuse of a component doesn't mean to use it more than one time but the ability of using that component in different contexts from the one it was designed for.

With all this, one of the biggest dreams of software engineering was to have a global market of components, like the one that exists for integrated circuits, where it is possible to buy and integrate components easily in new designs when and only when it is clearly known the interfaces of both.

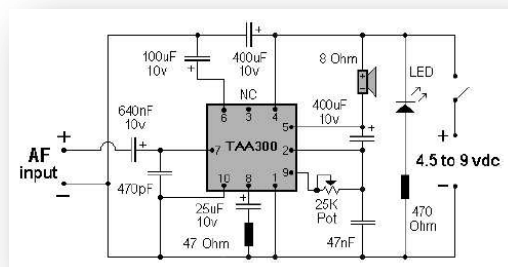


Fig. 61 – Example of an integrated circuit

On the CBP paradigm, the basic entities are the components, which are “black boxes” that incorporate some functionalities that were designed to be part of the global market of components, without knowing when, how and who will use them.

The only information that the users have about the components is the services that the components offer them by the interfaces and its requirements, without knowing its implementation.

According to Szyperski [Szyperski98], a **software component** is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.

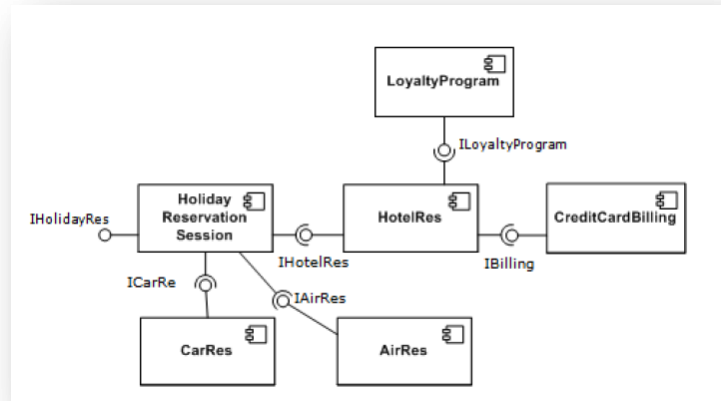


Fig. 62 - Example of a component-based system for a holidays reservation system

Once it is clear the concept of software component, a component model defines the way the interfaces and mechanisms of a component communicate between them. The coordination and interaction between components requires a standard that sets the required framework and the design rules and restrictions.

The most known component models are part of COM+/Microsoft, Enterprise JavaBeans/Sun and the CORBA component model [Heineman01].

The COM+/Microsoft component model is not portable between operating systems, the JavaBeans/Sun one is only available on Java operating systems and the CORBA one required reliable and mature implementations [Brooks05].

In summary, the component models are complex and require a very good knowledge in programming since they offer any help or advices about how to define the components interfaces, which means that to understand them it is needed some good experience.

In fact, this type of information about how to work with component models should be available so that, the newcomers would understand it faster and better, which would make them to contribute more and better to the field of robotics, without getting disappointed by how complex is the software.

An **interface** defines the set of operations that a component can perform, being these operations also known as services or responsibilities of the component. The interfaces refer to a mechanism of interaction between components and that control the dependencies between them.

In order to define the aspect of the interfaces of the components, it was born the concept of communication patterns.

The first time that communication patterns were used with the intention of defining the diversity of interfaces of components was by Schlegel and Wörz in 1999 [Schlegel99]. Later, and after this concept to become more mature by being used in various projects, it was better described by Schlegel in 2004 [Schlegel04]. As result of all this work was then born Smartsoft.

Nowadays, the development on the field of CBP is mainly made on several specific areas such as:

- Adaptation of both the programming languages and component models to these concepts;
- Design of new programming languages and component models;

- Creation of new tools and frameworks for the development of components.

In terms of programming languages, there are few that include enough concepts to be able to perform component-based programming, such as, Oberon, Java, Ada95, Modula-3 and Component Pascal. Indeed, none of them includes all the concepts but only some of them.

Once defined the component-based programming paradigm as well as the component models, it is now possible to describe the requirements that are needed to respect when creating a component-based framework on the field of robotics.

Also, it is possible to distinguish the different types of users that will use the framework.

3.2.1 Types of users and requirements of a component-based framework

There are different types of users of component-based frameworks, and the difference between them is on the approach that they make over the framework when managing the integration on robotics:

- **End users** are the type of users that use an application that is based on the user interface that was given to them. This type of users focus their attention on the functionalities of their application and make use of a system to accomplish the required tasks. They don't matter about how the application was implemented but about its reliability.
- **Applications developers** are the type of users that create applications based on the appropriate reusable components. They are responsible for adapting the application parameters according to the used components. They expect the framework to guarantee them clearly structured and coherent components interfaces in order to make it easier for them to put all the components together.
- **Components developers** are the type of users that focus on the implementation and specification of the components. They expect the framework to provide the needed infrastructure to get the maximum advantage possible from the implementation of a component in a way that it is compatible with other components and without being limited to its internal components.
In summary, the components developers want to focus their attention on the algorithms and functionalities of the components without needing to worry about the problems of integration with other components.
- **Frameworks developers** are the type of users that design and implement the framework in a way that it coincides with the multiple requirements by the best way possible, allowing the other types of users to focus only on their respective tasks.

On the field of robotics, there are some important characteristics that are required furthermore than the standard component-based software. Some of those functional and non-functional requirements are:

- The use of a **dynamic wiring pattern** that allows the dynamic configuration of the connections between the services of the different components during the running time of the applications.
To make the control flow as well as the data flow configurable from the outside part of the component is the key for the composition of the abilities, and it is needed in almost all the robotic architectures.
The dynamic cabling pattern interacts strictly with the communication primitives, and produces one of the most important differences when compared with other approaches.
- **Asynchronicity** is a very powerful concept that is used to divide the activities and to reduce the latency by making use of the concurrency between components as much as possible.
Decoupling is especially important when working with components, in order to avoid the time dependencies during the transmission between components.
A robotic framework must make use of the asynchronism as much as possible without requesting its users.
- **Components interfaces** need to be defined with a reasonable level of granularity. With this, it is possible to avoid interaction between fine-grained components and give support to the components that are weakly coupled, but with a strict and standard interface semantic.
- The **internal structures of the components** can follow completely different designs, which will make the components developers to request as less restrictions as possible.
Then, the frameworks need to allow different internal architectures for the components and at the same time guarantee the interoperability between them, by helping on the process of structuring and implementing a component.
- A framework needs to provide a certain level of **clarity** when it comes up to hide the details in order to reduce the complexity of an application or component.
On the other hand, it is not recommended to hide completely all the aspects of the distribution since it reduces the efficiency but also prevents from predicting the required time for the communication and use of the resources of the system.
- The **easy usage** of the framework allows the users to focus on robotics and at the same time allows the use of available recent software technology without needing an expert on robotics to convert himself on an expert on software engineering.
An example of one of the subjects that needs to be improved, is the clear localization of the components and respective services, and the concepts of concurrency that include the synchronization and security of the threads.

There are some hard subjects that need to be approached such as the clarity of the localization of the components and its services, or the concepts of concurrency such as synchronization, mutual exclusion and the security of the threads, without ignoring the different bandwidth requirements.

A framework is more valuable if its use is easy than if all the other requirements are respected.

A framework will be accepted by all the experts on robotics if:

- Offers something more than the others that are already on the market;
- It is easy to learn and to use;

- It is a software platform and not a robotic architecture;
- Approaches the middleware and synchronization problems;
- Provides a big quantity of devices controllers and components of the sensors and platforms that are most frequently used.

There are already many robotic software frameworks on the market. The majority are object-oriented without having an explicit components model (Montemerlo, 2003, Vaughan, 2003, Utz, 2002). Others, impose a particular internal component structure (Mallet, 2002) or even determine a specific robotic architecture (Konolige, 1997).

Anyway, none of them helps the components developers with the definition of adequate mechanisms for the interaction between components.

On the next sections it is described some robotics component-based frameworks, such as OROCOS, Orca2, Marie, RoboComp, Microsoft Robotics Studio and Smartsoft.

3.2.2 Robotics Component-Based Frameworks

3.2.2.1 OROCOS (<http://www.oroocos.org/>)

OROCOS (Open Robot Control Software) is a framework that was created with the intention of unifying all the methodologies used for the development of software architectures for robotic systems.

It is divided into four C++ libraries:

- Real-Time Toolkit (RTT) – Provides the structure and functionalities needed to develop components on real-time.
- Kinematics and Dynamics Library (KDL) – Develops an application independent framework for modeling and computation of kinematic chains.
- Bayesian Filtering Library (BFL) – Provides an application independent framework for inference in Dynamic Bayesian Networks.
- OROCOS Components Library (OCL) – Provides some of the components that are ready for use.

The OROCOS components are all built with RTT but it is also possible to use functionalities that are offered by other libraries.

The users can interact with the components through a predefined set of interfaces. On the image below, Fig. 63, it is shown the five different ways of interacting with a component.

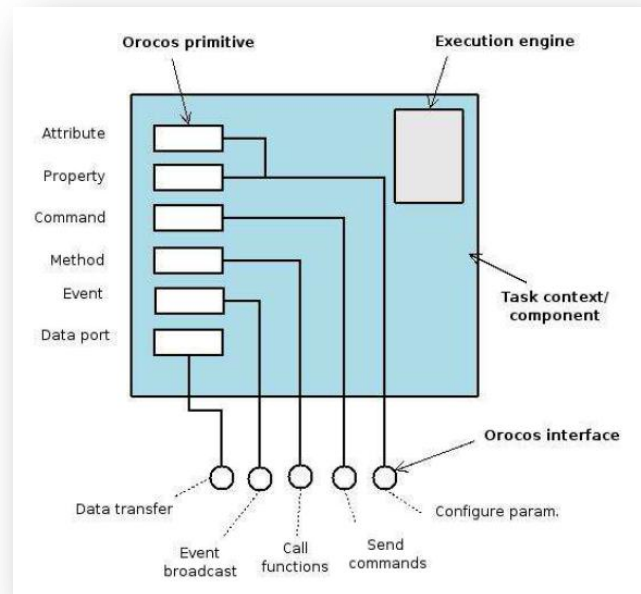


Fig. 63 - An OROCOS component

Apart from defining the components communication mechanisms and interface, OROCOS allows the applications developers to implement hierarchical state machines by using those mechanisms. Those state machines can be ran in any of the components during its respective running time.

All the OROCOS components are implemented by using the class *TaskContext*, which defines the context to run the component task. The most important units of an OROCOS component are:

- Commands – Are sent asynchronously from a component (client) to another components (servers), in order to send some instructions and achieve a particular goal.
- Methods – Are calls made to some particular components in order to calculate something.
- Properties – Are the components parameters that can be changed during its running time, which are stored in XML format.
- Events – Are run by a component when there is a particular change on it.
- Data ports – Are the communication mechanisms used to communicate the data buffers between different components.

For distributed applications, OROCOS uses TAO as transport mechanism. Each tasks defines its ports with an unique name, data type that it transmits and the method used to transmit.

3.2.2.2 Orca (<http://orca-robotics.sourceforge.net/>)

Orca is an 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.

During the running time, each component is an independent process. Orca doesn't impose any predefined architecture to the components neither to the whole system.

On Orca, all the communications between components are managed by ICE, which required that all the interfaces of all the components are defined on an unique language called Slice. The definition of each interface describe the component offered and required services.

ICE can be configured to use TCP, UDL or SLL as transport mechanisms.

On the image below, Fig. 64, it is shown an example of a system implemented with Orca which uses the Player/Stage simulator.

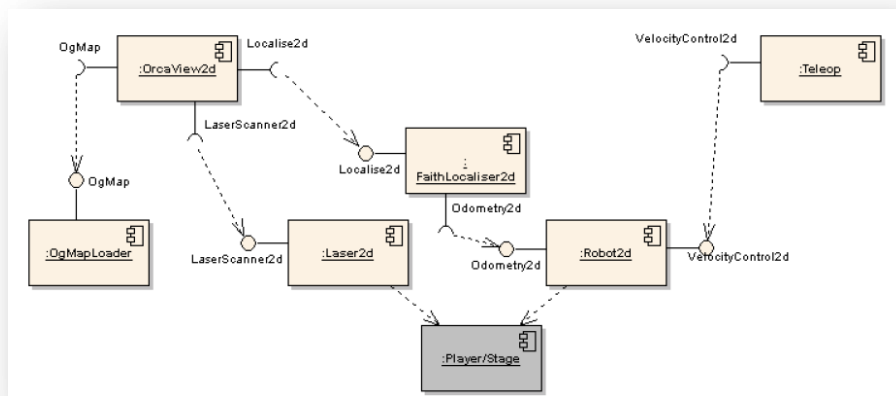


Fig. 64 - Example of system implemented with Orca2

Orca older versions used other transport mechanisms such as CORBA ACE/TAO and two personalized communication libraries. But, according to the development team of Orca, the reason for using now ICE instead of those mechanisms were the difficulties to use and limitations brought by CORBA and the two personalized communication libraries, respectively.

Orca deployment is carried out through *CMake*, which is a construction system, and ICE. It is possible to work with Orca on Linux, MacOS X and some versions of Windows. The programming languages supported by Orca are C++, Java, C#, Pything etc.

3.2.2.3 Marie (<http://marie.sourceforge.net/>)

Marie is a free software tool using a component-based approach to built robotics software systems by integrating previously-existing and new software components. Marie's initiative is based on a high level of reuse by integrating many different libraries and packets such as Player/Stage/Gazebo, FlowDesigner/RobotFlow, CARMEN/PMAP, ACE, MPICH2, ARIA/ARNL, libxml2, etc.

Marie supports multiple sets of concepts and abstractions by adopting a layered software architecture, defining different levels of abstraction into the global middleware framework:

- Core Layer – Consists of tools for communication, data handling, distribute computing and low-level operating system functions (e.g., memory, threads and processes, I/O control).

- Component Layer – Specifies and implements useful frameworks to add components and to support domain-specific concepts.
- Application Layer – Contains useful tools to build and manage integrated applications using available components, to craft robotic systems.

From the architecture perspective, Marie uses the MDP (Mediator Design Pattern) as the design base-concept. MDP creates a centralized control unit (named Mediator) which interacts with each other component independently and that also coordinates the global interactions between the components in order to make the desired system.

In order to implement the design units, the developers of Marie identify four main units (the Mediator is referred below as Mediator Interoperability Layer (MIL):

- Application Adapter (AA) – Responsible for sending and receiving the services and communication requests from the MIL to the applications and vice-versa.
- Communication Adapter (CA) – Responsible for converting the data between different communication methods: Mailbox (sends asynchronously), Shared Map (sends the received data through many out ports), Splitter (push-in / pull-out structures) and Switch (only one in port for each out port).
- Communication Manager (CM) – Responsible for creating and managing the communication links between the members of an AA that need to be connected. On a distributed system it is needed a CM on the processing node.
- Applications Manager (AM) – Responsible for controlling and managing all the system, by coordinating the system states. It is also responsible for configuring and controlling all the available components of the systems. On a distributed system, the processing node needs a AM.

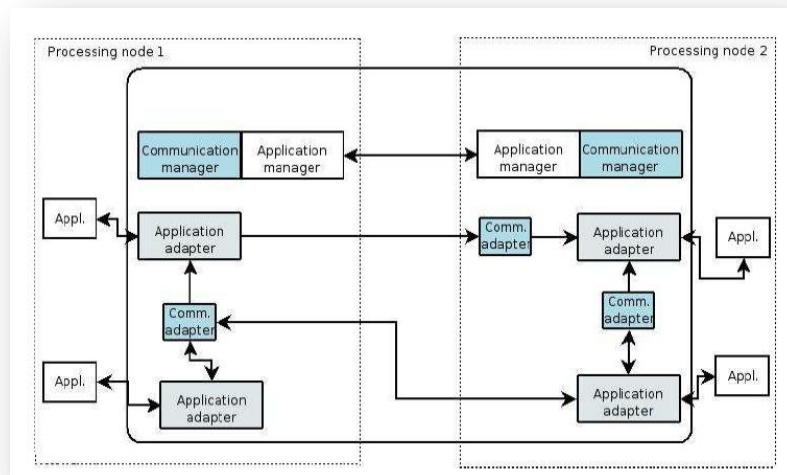


Fig. 65 - Marie centralized control unit (Mediator)

From the point of view of the middleware, Marie doesn't work with a specific communication mechanism but tries to support as many as possible. This is possible because of

the abstraction level of the framework, which provides the communication protocols and connections between components.

Marie uses ACE to implement the communications, which means that it is possible to use different types of communications (mostly through sockets that are based on the TCP protocol).

3.2.2.4 RoboComp (<http://robocomp.sourceforge.net/>)

RoboComp is an open-source robotics framework that provides different software components for the development of robotics software and the necessary tools to use them and to create new ones.

When running the components on RoboComp, they form a graph of processes that can be distributed over different cores or CPU's by using software component technology.

RoboComp doesn't impose any restrictions on the architecture of the components neither of the whole system.

The communications between components are handled by ICE, like on Orca.

RoboComp uses in some of its components other tools or libraries such as CMake, Qt4, IPP, OpenSceneGraph and OpenGL.

RoboComp can be implemented in any system that supports ICE and Python. The programming languages that are supported are the ones supported by ICE: C++, Java, Python, C#, Ruby, PHP and Objective-C.

It is possible to work with RoboComp on Linux, MacOS X and Windows.

3.2.2.5 Microsoft Robotics Studio (<http://www.microsoft.com/robotics/>)

Microsoft Robotics Developer Studio (MRDS) is a windows-based environment for developing robotics applications.

MRDS is a tool that allows the user to implement the software routines that will control its robot in any of the programming languages supported by the .NET environment, such as Visual Basic, C or C#.

The main features of MRDS are:

- A visual programming tool called Microsoft Visual Programming Language (VPL), that allows the user to intuitively create and debug robotics applications;
- A 3D visual simulation environment (VSE);
- Easy access to robot's sensors and actuators;
- Runtime support that manages the asynchronous input/output communications and the concurrency and distributions of services.

This last main feature, runtime support, consist of two components that make possible the development, supervision, deployment and functionality of a big range of applications. Those two components are CCR (Concurrency and Coordination Runtime) and DSS (Decentralized Software Services).

- CCR is a .NET-based concurrent library implementation for asynchronous parallel tasks.
It allows the concurrent and asynchronous coordination of the running flow, avoiding then the user to use threads, semaphores or other low level techniques to ensure the mutual exclusion or to prevent deadlocks.
- DSS allows the orchestration of multiple services to achieve complex behaviours.
It combines the traditional web architecture (HTTP) with units from new web-services oriented architectures (SOAP).
The resulting architecture is then completely based on services that coordinate themselves between them in order to create distributed applications. DSS uses the HTTP and DSSP (a protocol provided by DSS for sending messages between different services) protocols.

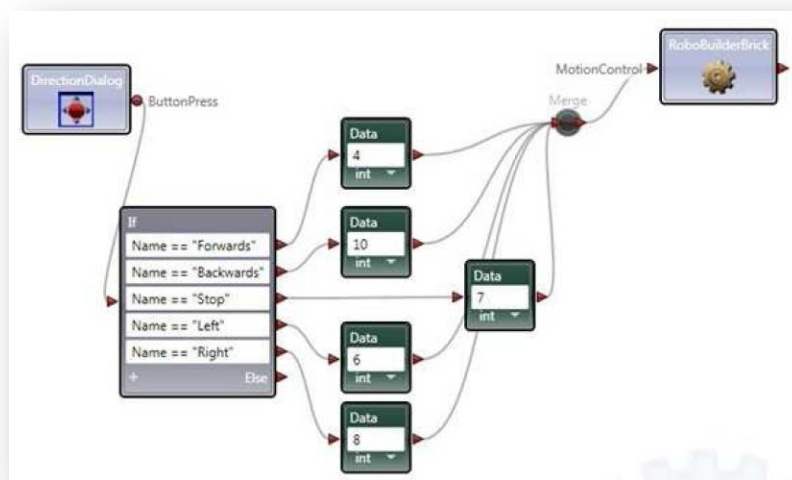


Fig. 66 - Example of a robotic application created with VPL

3.3 Model-Driven Engineering Frameworks

Model-Driven Engineering (MDE) is a step higher on the abstraction level when compared with component-based programming. MDE proposes the use of models as main concept for software development.

A **model** is a simplified representation of the reality that only shown the aspects that are interesting for our application.

The model-driven paradigm has called the attention of the robotic community because of the results it got on other domains like automotive, aviation, electronics, etc., in such a way that it improved the levels of reuse and quality of the software, and at the same time reduced the development time.

MDE allows the developers to focus on the concepts of the domain of the application, rather than on the implementation details.

A model is considered effective if it makes sense from the point of view of the user and if it can serve as basis for the implementation of systems.

Models are developed through extensive communication between all the members of the development team. As the models approach completion, they enable the development of software and systems based on them.

Models are formally defined as meta-models, which consist of the concepts related with the domain of a particular application, the syntactic relationships between those concepts, and the rules that determine when a model is correctly built.

On other words, a meta-model defines the abstract syntax of a modeling language.

The infinite set of valid models that can be developed by using a meta-model as starting point are known as modeling language.

Due to the fact that a system can be described by different models on the different levels of abstraction, the models transformations are one of the key points of this approach.

The models transformations define how the models can be understood and translated. There are two models transformations: model to model (M2M) and model to text (M2T).

Apart from providing a formal basis that improves the process of software development, MDE doesn't offer the most appropriate way to select or design modeling languages neither to perform the models transformations.

For those reasons, MDA (Model-Driven Architectures) was developed, which is a software design approach for the development of software systems that was launched by OMG (Object Management Group).

Like it can be seen on the image below, Fig. 67, which shows the structure of MDA, the MDA models can be classified in three different levels of abstraction: Computational-Independent Models (CIM), Platform-Independent Models (PIM) or Platform-Specific Models (PSM).

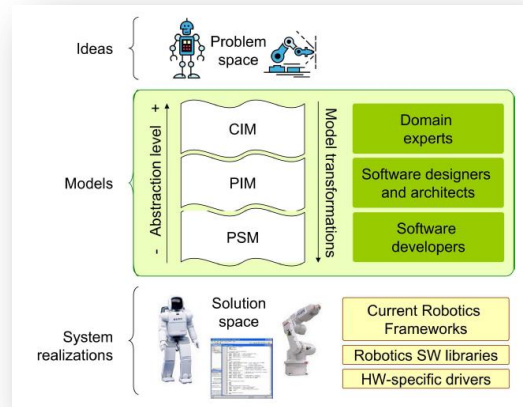


Fig. 67 - Structure of MDA

Given a PIM, a MDA can translate it into a PSM in order to start its correspondent implementation by using domain specific languages or general languages like Java or C++.

The translation from PIM to PSM is a M2M transformation, while the implementation of a PSM is a M2T transformation since it translates the PSM into code.



Fig. 68 - MDA transformations from PIM to Code

The separation between the architecture design and the building technologies is one of the main features of MDA, since it allows the design and the architecture to be changed independently. MDA assures that the model is independent from the platform where it was developed, and also that it “survives” to all the changes that can be made on the fabrication technologies and on the software architectures.

During the last years, the MDE approach has shown that it reduces many limitations that existed on complex systems, which need a multi-disciplined development and that require a big reliability and robustness. Examples of that are the domains of integrated systems, automotive and home-automation.

On the domain of robotics, the BRICS project is a specific approach for the development of MDE as a way that requires less effort to develop robotic engineering systems, making that the best solutions are easily reusable.

Two robotics model-driven frameworks are Smartsoft and MinFr, which are both explained in detail on chapter since they are used on this project, in order to implement some applications.

An example of an application that uses the MDE approach and that is not from the domain of robotics is *Webratio* (<http://www.webratio.com>).

Webratio is a tool developed by *WebModels S.L.R.* that provides a model-based development environment to model web applications, allowing the automatic code generation for the J2EE platform. The generated applications are Java standard applications and can be deployed in any Java application server such as Tomcat, JBoss, Resin, IBM Websphere, BEA WebLogic, etc.

Webratio reduces the time needed for development of web applications, makes it easier to develop functional prototypes, and also reduces the costs related with the development, implementation and maintenance. For this reasons, *Webratio* increases the ability of the developers and the efficiency of the Java solutions that are deployed through web.

Webratio provides a diagrams-editor named WebML, which is shown on Fig. 69, that allows the user to express visually all the requirements of his Web application.

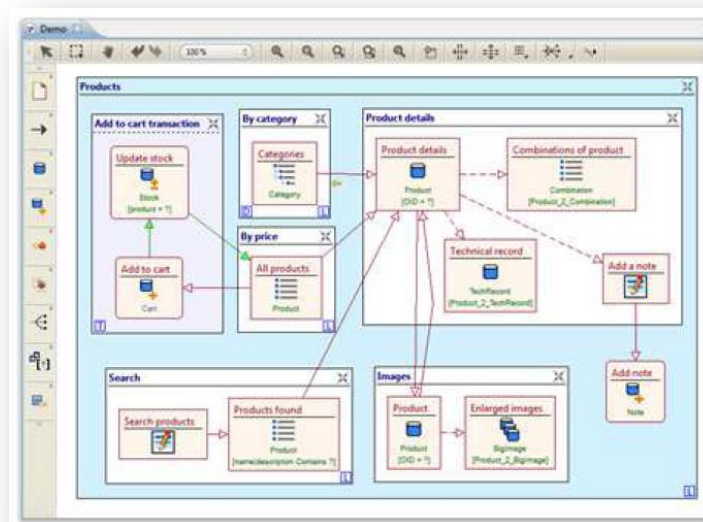


Fig. 69 – WebML, the diagram editor of Webratio

4 Robotic Frameworks used on this project

4.1 Smartsoft

Smartsoft is a component-based framework for robotics software based on communication patterns as core of a robotics component model.

For the development of robotics applications, Smartsoft provides a model-driven tool chain named *Smartsoft MDSD* and a framework that provides support to run the applications.

The Smartsoft component-based framework will be explained on section 4.1.1, while the model-driven tool chain *Smartsoft MDSD* will be explained in detail on the section 4.1.2.

4.1.1 Smartsoft Component-Based Framework

The Smartsoft framework follows the specification of the middleware CORBA and uses ACE/TAO for the communication between components.

In order to achieve the requirements of the component-based frameworks, it is suggested that all the interactions between components must be done through a standardized set of communication patterns. This assures that the separation between the components and its related elements from the appropriate abstraction levels.

Smartsoft assists the component developer, the application developer and the end user in developing and using distributed components in such a way that the semantics of the interface of components is defined by a standardized set of communication patterns, irrespective of where they are applied.

Smartsoft is divided into four abstraction levels:

- **Components** – Are processes that contain threads and interacts with other components by making use of communication patterns.
- **Communication Patterns** – Define the communication mode, provides predefined access methods to the data and hides all the communication and synchronization issues. It always consists of two complementary parts named service requestor and service provider (client and server).
- **Communication Objects** – Represent the content to be transmitted via a communication pattern. They are always transmitted *by value*. Communication objects are ordinary objects decorated with additional member functions for use by the framework.
- **Services** – Each instantiation of a communication pattern provides a service. Basically, a service comprises the communication mode as defined by the communication pattern and the content as defined by the communication object.

Mastering the inter-component communication is considered as key to master component dependencies and to ensure uniform component interfaces.

The approach taken by Smartsoft therefore selects inter-component communication as a suitable starting point. The basic idea is to provide a small set of communication patterns which can transmit objects between components and then squeeze every component interaction into

those predefined patterns. As shown in Fig. 70, where the components interact only through those patterns.

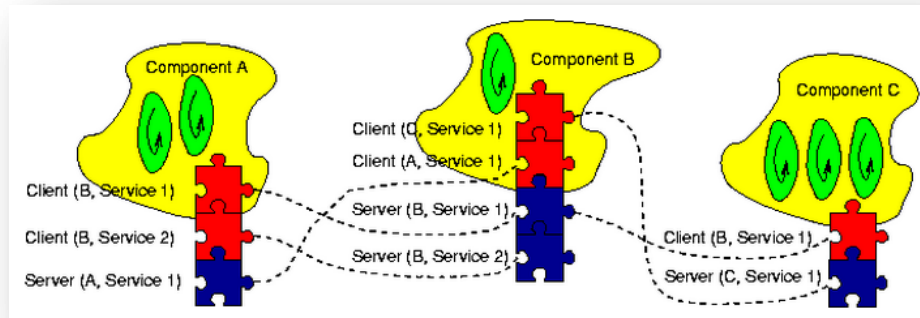


Fig. 70 - Overview of the Smartsoft component-based approach

Since all the interactions between components are done by using standardized communication patterns, the components interfaces consist of a known model with a precise and know semantics. This, allows the separation of the internal implementation from the external behaviours of a component, obtaining then coupled and distributed systems that are based on standard components, which interaction is changed according to the needs.

Predefined member functions of the communication patterns provide access modes like synchronous and asynchronous service invocation or provide a handler based request handling. This no longer leaves it to the component developer to decide on whether to invoke a remote service synchronously or asynchronously but already defines an easy to use and fixed set of access modes with every communication pattern.

Communication patterns hide the underlying middleware and do not expect the framework user to deal with CORBA details.

Compared to distributed objects, one can neither expose arbitrary member functions as component interface nor can one dilute the precise interface semantic.

Since communication objects are always transmitted by communication patterns and since member functions of communication objects are not exposed outside a component, usage of communication objects and implementing user member functions is completely free from cumbersome and demanding details of inter-component communication and distributed object mechanisms.

Communication patterns ensure decoupling of components since both parts always interact asynchronously irrespective of the access mode used by the user.

Each component can provide and use as many services as needed. Apart from that, the use of dynamic wiring during the running time is supported by a pattern that interacts with the communication primitives.

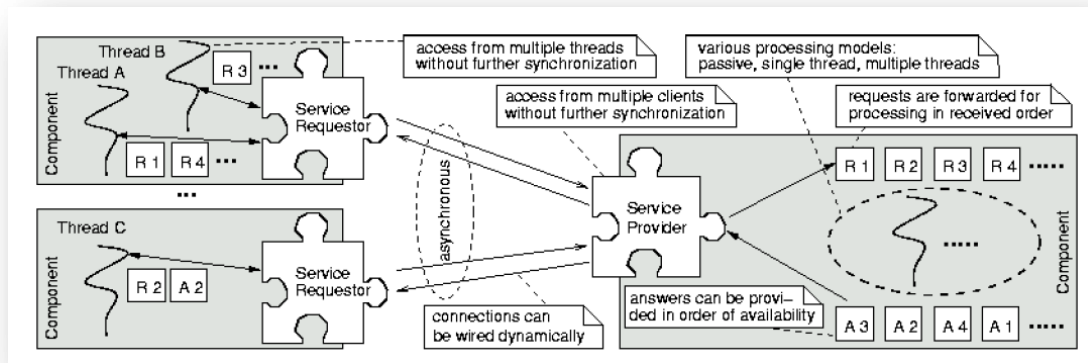


Fig. 71 - Implementation of the query communication pattern with two clients (requestor) and one server (provider)

As it can be seen on Fig. 71, which shows an example of implementation of the query communication pattern with two requestors and one provider, the component on the right side (provider) provides a service through a communication port, while the other two components on the left side (requestors) have a port that requests that exact service.

The communication objects are sent via the links that are connecting the ports of the three components. Each Smartsoft component has an arbitrary number of running threads, which are responsible for using and managing the communication ports of that component.

Once the components and its respective communication objects are built, it is needed to implement the communications between the components, which is done on the Smartsoft deployments, generating then the robotic application.

The Smartsoft deployment is responsible for converting the components into binary files so that the used middleware (CORBA/ACE) can perform the communications between them.

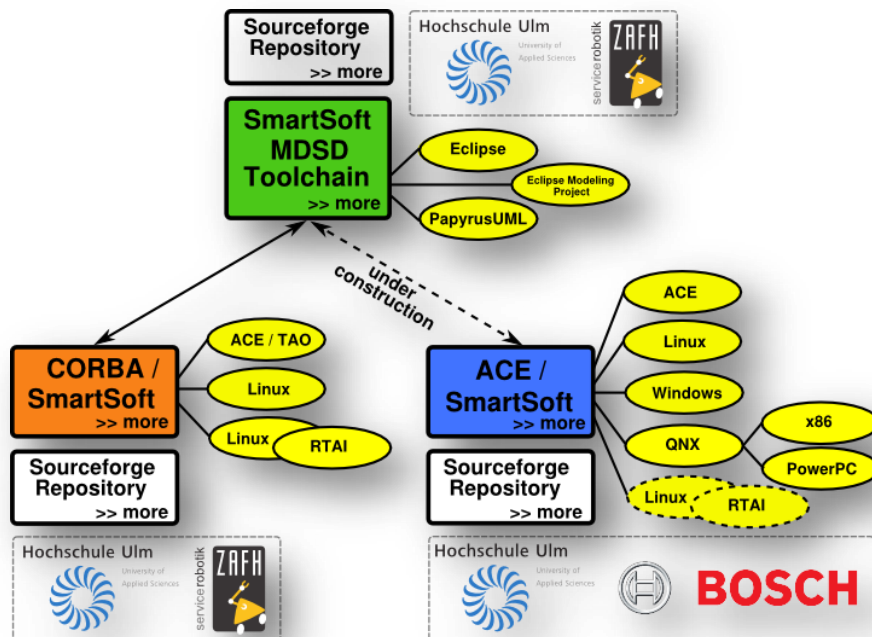


Fig. 72 - Structure of Smartsoft

On the image above, Fig. 72, it is shown how Smartsoft is structured. The toolkit *Smartsoft MDSD*, which imposes the components models that define the communication patterns, is independent from the two different implementations of the Smartsoft framework, CORBA/Smartsoft and ACE/Smartsoft.

CORBA/Smartsoft is an implementation of Smartsoft framework that uses TAO as middleware, which is a implementation of CORBA by making use of the framework ACE to access the sockets, while ACE/Smartsoft doesn't use CORBA neither TAO but only ACE to perform all the managing of the communications.

From the point of view of the end user, there is no difference between both implementations since both use the toolkit *Smartsoft MDSD* and the only difference they have is on the running platform they use, which influences only on low level details of the communications between components.

The web pages of each of the implementations are:

- **CORBA/Smartsoft** – <http://smart-robotics.sourceforge.net/corbaSmartSoft/>
- **ACE/Smartsoft** – <http://smart-robotics.sourceforge.net/aceSmartSoft/>

When developing an application with a Smartsoft framework, the components must be connected between themselves on the deployments, so that they are able to send and receive communication objects through their ports.

In summary, from the point of view of the end-user, Smartsoft is dividing into four units:

- **Communication Objects** – It is the data that is transmitted by the components through the communication ports.
- **Components** – Are the reusable units that interact between themselves by interchanging communication objects through their communication ports, offering and requesting services.
- **Communication Ports** – Help the components and applications developer to create and use components in a way that the semantic of the interfaces is defined by the communication patterns.
Each communication port is composed by two complementary parts named service requestor and server provider, which are implemented at different components.
- **Deployment** – It is a project where the application is designed in a way that it makes use of the components and implements the communication links between them to make the application work.

On the section 6.1 there is explained in detail a Smartsoft application that implements the hybrid architecture.

4.1.2 Smartsoft MDSD

Smartsoft MDSD is the modeling tool chain that imposes the components model when implementing an application with Smartsoft component-based framework.

Smartsoft MDSD is based on the Eclipse Modeling Framework, and with it, is possible to define and change models, in a way that the generated code associated to those models is then executed by the framework.

On the image below, Fig. 73, it shown all the process needed to obtain the source-code (PSI) of a Smartsoft model.

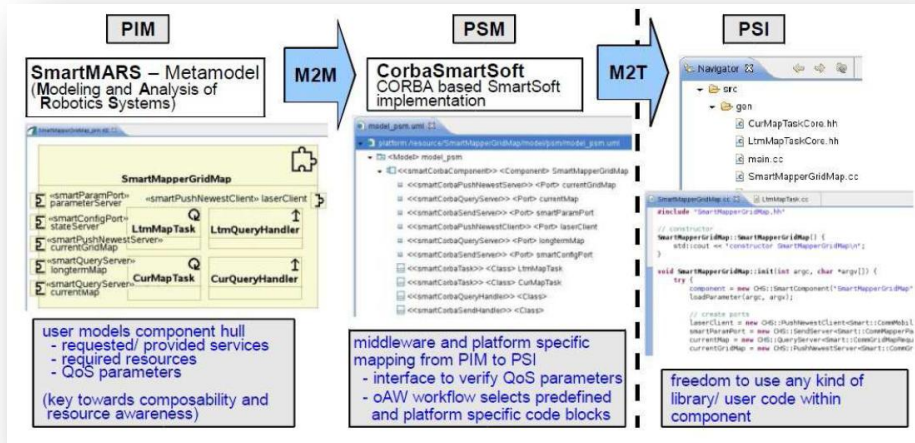


Fig. 73 - Smartsoft MDSD design process

On *Smartsoft MDSD*, the meta-model implemented with *Papyrus* (PIM) is denominated as *SmartMARS* (Smart Modeling and Analysis of Robotics Systems).

After performing the M2M transformation and the correspondent mapping process over *SmartMARS* it is obtained the respective PSM, which on Smartsoft corresponds to the CORBA-based Smartsoft implementation.

At last, after performing the M2T transformation over the obtained PSM, it is obtained the source-code where it is implemented the PSM model, which corresponds to the Platform Specific Implementation (PSI).

4.2 MinFr

MinFr is a component-based framework that uses a model-driven tool chain to integrate the framework real-time issues, as well and also to impose the components models.

It is being developed by the DSIE development team of the Technical University of Cartagena.

Initially, on the first version of MinFr, the development approach of an application started by modeling the CB application architecture with the modeling language V3 CMM and then, use a M2M model transformation to generate the UML implementation mode, from which the executable code was generated by using a M2T model transformation.

In a second version, the development approach of an application with MinFr became slightly different. To ease the generation of executable code from the input V3 CMM model, MinFr used a OO framework as the target for a model transformation.

That OO framework provided the required properties for the final application as well as it could be changed, substituted or improved easier than a model transformation.

Nowadays, the executable code of each model is not anymore generated but the models are interpreted by a *Java* application that instantiates from a MinFr library all the MinFr classes that are used by each model.

The MinFr library that includes all the structure of MinFr is explained on section 4.2.1. On section 4.2.2 it is explained the V3 CMM modeling language used to model the components, and on section 4.2.3 it is explained how does MinFr looks like from the point of view of an end-user.

4.2.1 MinFr Component-Based Framework

The main drawback of CB (Component-Based) frameworks, is that despite being a CB approach, their designers must develop, integrate and connect components by using OO (Object-Oriented) technology. This is a big problem since CB approach requires more abstractions and tool support than OO technology can offer.

According to that, the design of a CB framework implementation should overcome, among other, the following problems:

- The definition of a component language for modeling domain application in a way that designers could work with CBSD (Component-Based Software Development) abstractions rather than with OO ones.
- The translation of the resulting models to executable code and to analysis models, which can be later injected to tools in order to analyze both the CB application and the resulting code properties.

On MinFr, components are not a unit with well defined interfaces and explicit context of use but, architectural units that communicate only through their ports, in line with the software architecture discipline and ADLs (Architecture Description Languages).

Those architectural components are used to model robotic applications and to translate the CBSD concepts to OO concepts to enable compliance with the real-time requirements of the applications and the components distribution.

The main ADs (Architectural Drivers) that guide the design of the MinFr component-based framework are:

- **AD1** – User control over concurrency policy: thread number, thread spawning (static vs. dynamic policies), scheduling policy (fixed priority schedulers vs. dynamic priority schedulers), etc.
- **AD2** – User control over the allocation of activities to threads, which means that the user can control the computational load assigned to each thread. V3 CMM considers the activity associated to a state as the minimum computational unit and the framework allows allocating all the activities to a single thread, every activity to its own thread, or a combination of both policies.
- **AD3** – Facilitate the instantiation of the framework from the input CBSD model.
- **AD4** – Control over the communication mechanism between components, if synchronous or asynchronous.
- **AD5** – Control over the component distribution and deployment.

In order to describe the design of the framework it will be used the design patterns that were used to implement it. On Fig. 74 it is shown the pattern sequence used to meet the architectural drivers mentioned before, and on Fig. 75 it is shown the classes that fulfill the roles defined by the selected patterns.

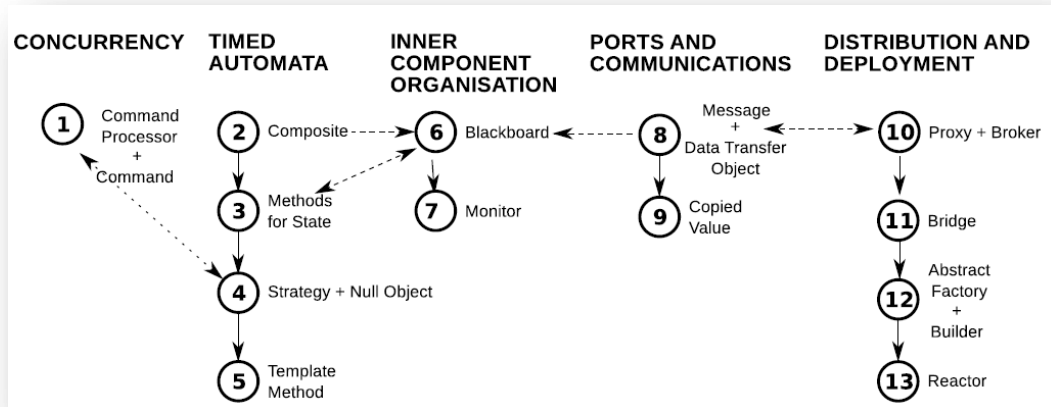


Fig. 74 - Sequence of design patterns considered for the framework development

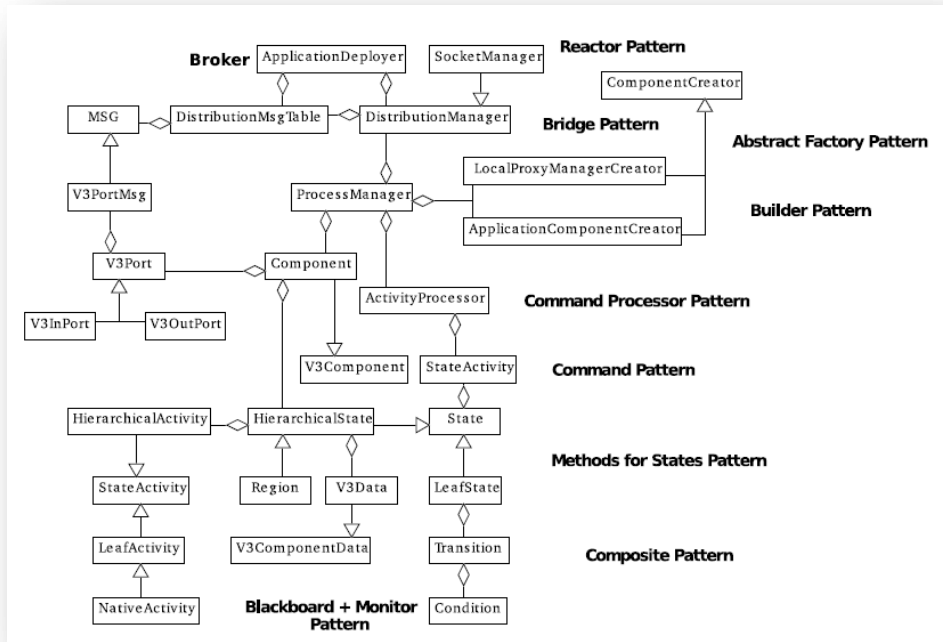


Fig. 75 - Class diagram considered for the framework development

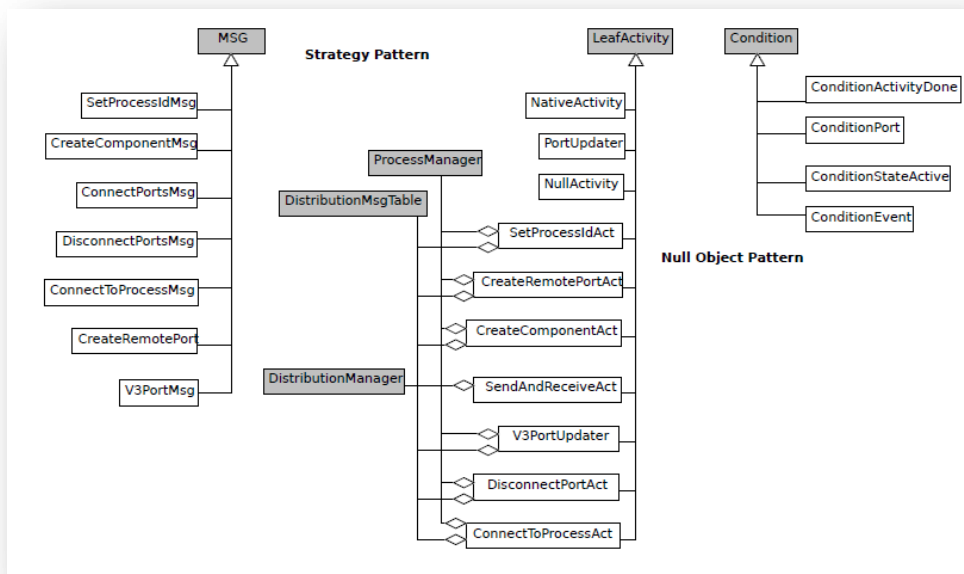


Fig. 76 - Classes hierarchy that complement the previous image diagram (Fig. 76), where the classes in grey correspond to classes from that image.

Among the aforementioned drivers, the main one is the ability to define any number of threads and control their computational load (architectural drivers AD1 and AD2). This computational load is mainly determined by the activities associated to the states of the timed automata.

In order to achieve this goal, the **Command Processor** architectural pattern and its highly coupled **Command** pattern have been selected, and they were the firsts to be applied in the framework design (see Fig. 74).

The **Command Processor** pattern separates service requests from their execution by defining a thread (the **command processor**) where the requests are managed as independent objects (the **commands**). Those patterns provide the required level of flexibility, since they impose no constraints over command subscription to threads, number of commands, concurrency scheme, etc.

The roles defined by these two patterns are executed by the classes ActivityProcessor and StateActivity, respectively (see Fig. 75).

Another key aspect, related to AD4, is to provide an OO implementation of the timed automata compatible with the selected patterns for concurrency control, in order to integrate it in the scheme defined by the aforementioned **Command Processor** pattern. This aspect also has a great influence on the whole design, since timed automatas model the behaviour of the components.

To do so, regions and states are treated homogeneously, and thus a simplified version of the **Composite** pattern was selected. The timed automata is managed following the **Methods for States** pattern, and the instances of the classes representing it are stored in a hash table. The roles defined by this pattern are executed by the classes State, HierarchicalState, Region and LeafState (see Fig. 75).

Each activity associated to a state of the timed automata is implemented as an object, following again the **Command** pattern, represented by the class StateActivity.

In this way, activities can be allocated to any **command processor**. This constitutes the link between concurrency control and timed automata implementation, since activities play roles in both patterns.

In order to distinct between states and regions, two hierarchies of StateActivity were defined, which were implemented following the Strategy pattern:

- Activities that are associated to leaf states (represented by the root class LeafActivity);
- Activities that are associated to regions (represented by the class HierarchicalActivity).

The second hierarchy is aimed at managing the region states and transitions, and thus is provided as part of the framework.

The first hierarchy, which is shown in Fig. 76, is related to:

- Activities that are defined in the V3 CMM models, represented by NativeActivity subclasses;
- Activities that manage the flow of data and control among a component through their ports.

As it can be seen on Fig. 75, conditions, transitions and events are modeled as separate classes. This is done to facilitate the framework instantiation from the input V3CMM model (AD4).

Condition is an abstract class used to model transitions conditions. It provides an abstract method to evaluate the condition. Concrete subclasses of the class Condition are:

- ConditionStateActive, which tests whether a specific state is active;
- ConditionActivityDone, which tests whether an activity is finished;
- ConditionPort, which tests if a message has been received by a specific port.

On the other hand, the class *Transition* include the source and target states, and groups a set of conditions vectors that must be evaluated to determine if the transition should be executed.

To end with timed automata implementation, it is worth mentioning two additional patterns. The *Null Object* pattern is used for smoothly integrating states that have no associated activity, while the *Template Method* pattern is related to defining the activity in charge of region management.

The next challenge on the implementation of this framework was then how to store and manage the component internal data, including all the states and activities mentioned above, the data received or that must be sent to other components, the transitions among states, event queues, etc.

All these data is organized following the *Blackboard* pattern. The idea behind this pattern is that a collection of different threads can work cooperatively on a common data structure. In this case, the threads are the command processors mentioned above.

The main liabilities of the *Blackboard* pattern (i.e. difficulties for controlling and testing, as well as synchronization issues in concurrent threads) are mitigated by the fact that each component has its own blackboard, which maintains a relatively small amount of data. Besides that, the data is organized in small hash tables with different access policies (monitors, 1-writer/n-readers, etc.).

The roles defined by this pattern are realized by the classes *V3Data* and *V3ComponentData*.

As shown in Fig. 74, the *Blackboard* pattern serves as a joint point between timed automata and the input/output messages sent by components through their ports.

Component ports and messages exchanged between them are modeled as separate classes. The classes representing these entities are the classes *V3Port* and *V3Port Msg*, respectively.

The communication mechanism implemented by default in the framework is the asynchronous without reply scheme, based on the exchange of messages following the *Message* pattern.

To prevent the exchange of many small messages, it is used the *Data Transfer Object* pattern in order to encapsulate in a single message all state information associated to a port interface, which is later serialized and sent through the port.

Finally, because components encapsulate their inner state, it used the *Copied Value* pattern to send copies of the relevant state information in each message.

The framework adds extra regions to manage the flow of messages through ports, the internal memory of the component, and each region of the component's timed automata.

The activities that perform such a duty are predefined in the framework, and they shall be assigned to threads in a similar way as the user does with the activities of the input V3 CMM model.

It is also worth highlighting that this design facilitates schedulability analysis, since no code is "hidden" in the framework implementation, but it must be explicitly allocated to a particular thread.

While the temporal characteristics of the input model activities are specified in the model itself, the characteristics of those added by the framework cannot be set arbitrarily, but must be derived from the model, although currently this is done manually.

For example, the period of the activity that manages a region must be less than the period of the most frequent activity defined inside the region.

On the image below, Fig. 77, it is shown a scenario that illustrates these features.

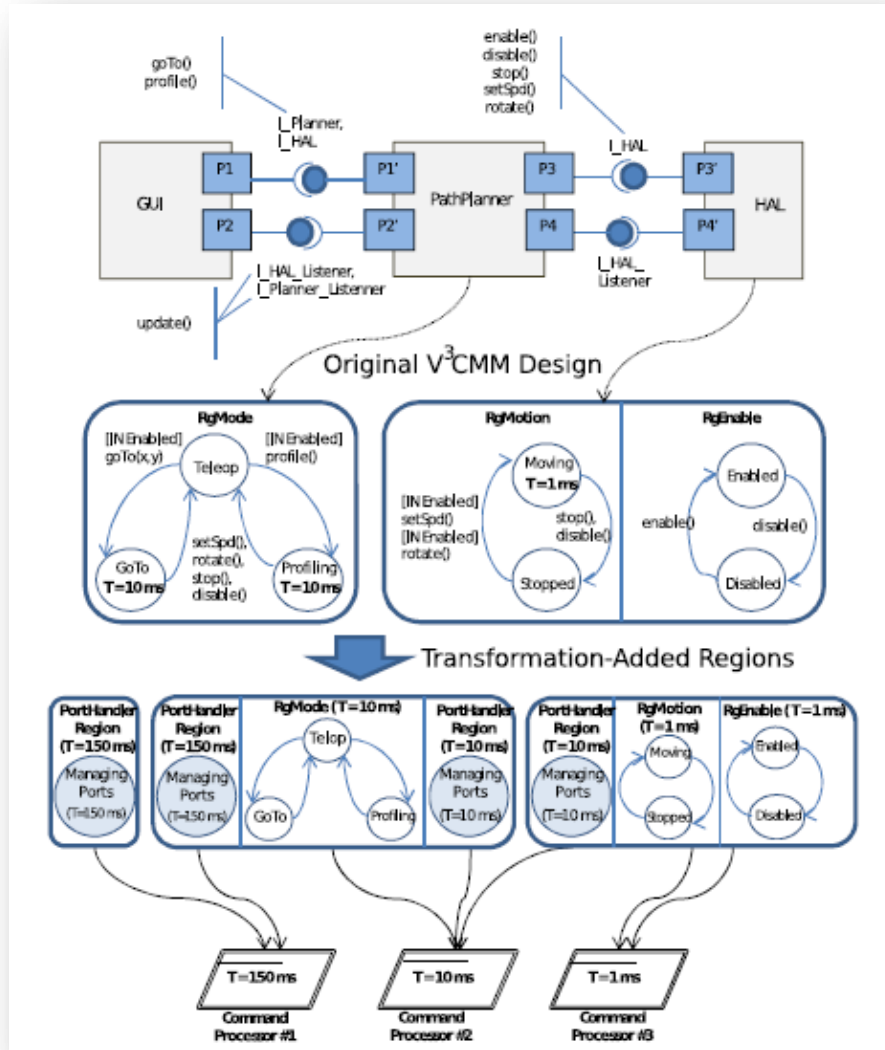


Fig. 77 - An example of the use of the framework. Definition of thread number and allocation of activities to them.

The original V3 CMM timed automata is extended with regions for managing the component state and the messages flow through ports.

Notice that, regions have marks indicating the period of the activity managing it. For instance, region RgMotion is managed every 10 ms.

Also notice that new regions have been added to the original model in order to manage communications among ports.

The framework allows designers to select the number of threads (three in this case) and the way activities are allocated to them (in this case, all the region activities are allocated to the same thread, according to their periods).

In any case, the framework user can select any other combination of threads and activities once the framework has been instantiated from the input V3CMM model.

The framework does not provide any guidance on the number of threads to be created, or how to make the assignment of activities to threads, but provides the mechanisms needed for the users to make such assignment according to some heuristics.

In summary, the code that implements these architectural drivers is organized into three groups with clearly defined interfaces:

- **C1** – The code that provides the runtime support (modeling threads);
- **C2** – The code that provides an OO interpretation of the CBSD concepts and the framework 'hot-spots';
- **C3** – The application-specific code that supplement the 'hot-spots' of the framework to create a specific application.

With this, it is then possible to provide an alternative interpretation of CBSD concepts (C2) using the same run-time support (C1), as well as to reuse the same application (C3) in a different run-time (C1), provided that C2 is not changed.

Being able to distribute components among nodes should be one of the main characteristics of a framework.

This is true due to the following reasons:

- By the very nature of robotic applications, which normally include several computing nodes;
- Isolation of components with hard RT requirements from those with soft RT requirements. In this case, this characteristic can be achieved by separating both types of components and assigning them to different nodes;
- The ability to modify the components deployment depending on the available computing resources and the state of the environment.

In fact, each and every available robotic framework include distribution in one way or another, being the use of a middleware technology, like CORBA, the most common solution used all of them.

On MinFr, however, it was develop an ad-hoc middleware for carrying out component distribution for the following reasons:

- The users of commercial middleware technologies normally lose the control over the execution of the application (the "inversion of control" problem), as well as some RT characteristics (like number of threads, computing time, etc.) that must be taken into account if RT analysis is required;
- Commercial middleware normally target OO applications, while MinFr approach uses components for modeling the application architecture. Thus, it would be awkward to combine components with objects when designing the application architecture;
- Though it would be possible to achieve a certain degree of control over the middleware characteristics, the changes needed for introducing these modifications would require a deep knowledge of the middleware implementation (and its source code).
- The overall design approach to CBSD applications that was followed for the implementation of MinFr does not need all the distribution services normally required by distributed applications and provided by middleware technologies. On MinFr, the components that make the application architecture up and the

connections among them are defined in the input models, and therefore services like naming, registering, searching, etc. are not needed.

Because of these reasons, it was considered that the best choice was to develop an ad-hoc middleware with the minimum services needed for managing component distribution.

With this, it means that were only considered services for component allocation and de-allocation inside processes, component and application start and stop services, and connection and disconnection of compatible ports of components allocated in the same or in different processes.

All these processes can be executed in different nodes, and thus, the framework considers two levels of component distribution:

- Components are allocated to a process, which means that all the activities belonging to the component must be assigned to threads belonging to such process;
- Processes are assigned to different computational nodes.

According to what has been stated, in order to make the distribution possible and feasible, two artifacts have been defined:

- One belonging to the CBSD domain (the LocalProxyManager component);
- The other one is a class added to the framework (the ApplicationDeployer class).

These elements are in charge of realizing the deployment of the application components, and of managing the messages sent among the nodes.

It is worth highlighting that the implementation of these two elements did not require modifying the original framework structure, but they only instantiate the base classes provided by the framework.

The objectives of the LocalProxyManager component are:

- Create and connect component instances in the node that manages them;
- Act as a proxy of the ports of remote components.

This component is not meant to be directly added by the application developer, but, for each deployment process, one of such components is automatically added to the application architecture, and then created by the framework distribution services.

The framework distribution services are mainly provided by the ApplicationDeployer class, which acts as the master node for application deployment, and thus, it must be executed in its own process before the application can be deployed.

This class performs the application deployment according to the specification of a configuration file.

Internally, the LocalProxyManager contains the same constructive elements (regions, states, transitions and activities) as any other component that has to be instantiated in the framework.

Also, like the rest of the components, it allows their internal activities to be assigned to different threads in any arbitrary fashion.

The same structure was maintained so that it ensures that the overhead added by this component is completely measurable and afterwards analyzable.

Although from the point of view of the framework user is irrelevant how the LocalProxyManager component works internally, it is worth mentioning that the current implementation is based on the use of the **Reactor** pattern (see Fig. 75).

On the other hand, the ApplicationDeployer can be considered as a mini Broker, without explicit register nor look-up services.

These services are not explicitly invoked because all the information required to deploy and connect the application components is read from the application configuration file, which is used by the ApplicationDeployer to communicate with LocalProxyManagers, which components should be created, and how to connect their ports.

If any of these ports belong to a component that runs in a separate process, the ApplicationDeployer sends to the LocalProxyManager a proxy of such port, so that the LocalProxyManager can make a local connection.

The creation of the concrete application components, which also involves the creation of its ports, states, transitions, etc., is carried out by the classes LocalProxyManagerCreator and ApplicationComponentCreator, which play the roles defined in the **Abstract Factory** and **Builder** patterns (see Fig. 74 and Fig. 75)

The communication protocol currently used is TCP. The decoupling of ApplicationDeployer and LocalProxyManager from the communication infrastructure is achieved by means of the Bridge pattern.

Both artifacts use the communication services defined in the abstract class DistributionManager. Currently, there is only one concrete subclass of this class, SocketManager, which relies on the sockets libraries, but, this design allows an easy extension of the framework with new classes for changing the underlying communication protocol.

In the current implementation, based on TCP sockets, the ApplicationDeployer is also in charge of sending to each LocalProxyManager the port and IP address of the rest of the nodes the application is going to be deployed into, in order for them to establish and manage communication among them.

Ports are taken starting from number 50.000 to avoid possible collisions with other protocols.

Messages exchanged among components (both intra and inter nodes) as well as those related to components deployment are marshaled as ASCII string characters, where each character has the size of a byte.

These messages use labels to separate and identify their fields, and to separate consecutive messages.

Many of the subclasses shown in Fig. 76 are related to component distribution. Specifically, those derived from the class MSG define the kind of messages exchanged by the ApplicationDeployer and the LocalProxyManagers, while the ones deriving from LeafActivity define how the services requested by such messages are carried out.

Newly, the activities added by the framework to achieve distribution are treated as “normal” activities, and thus have to be allocated to threads just like the rest of the component activities.

The artifacts in charge of managing component distribution and deployment are considered “normal” components, in the sense that they use the same elements and behave exactly like any other component in the application.

This allows the inclusion of communication overhead in the RT analysis, provided that transmission times are known and can be incorporated to the execution time associated to the activities that manage communications.

In order to illustrate the whole process, the feasibility of the approach, and its main advantages, it will be used a simple example consisting of three components deployed in two nodes.

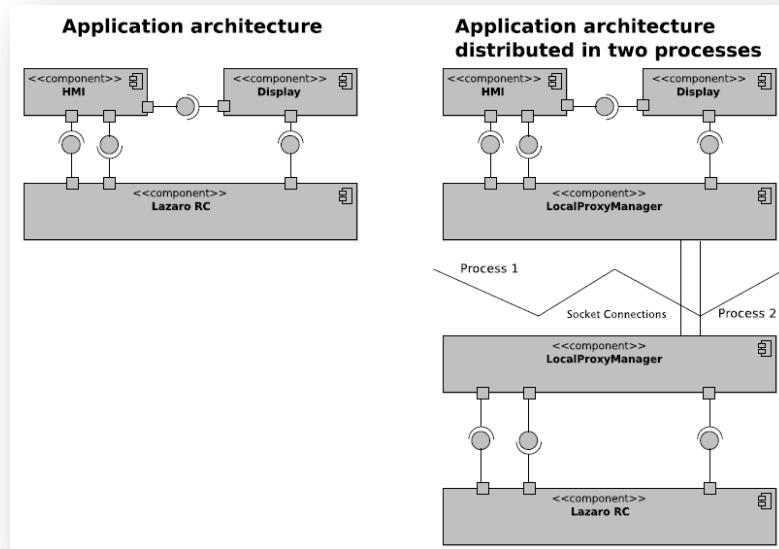


Fig. 78 – Example of an architecture with three components where they are assigned to one process (left side) and to two processes (right side)

The selected example revolves around an application for tele-operating a mobile robot. The architecture of the application, which is shown in the left side of Fig. 78, consists of three components:

- A Human-Machine Interface (HMI) component that sends the movement and control commands issued by the user to the robot;
- A Display component that shows the information obtained from the robot sensors and the commands sent by the user;
- A controller component, entitled LazaroRC, that acts as a hardware abstraction layer that facilitates the sending and receiving of messages to the robot.

The figure only shows the structural view of the application, since the timed automatas that describe component behaviour are not relevant in this example.

The right side of the figure shows the same application distributed in two process, which requires the addition of two LocalProxyManager.

4.2.2 V3 CMM

V3 CMM (3 View Component Meta-Model) is a component meta-model that was developed by the DSIE development team of the Technical University of Cartagena.

V3 CMM language provides three complementary views that allows application and component developers to define and connect software components:

- Architectural view – Used to define components (interfaces, ports, requested and offered services, composite components, etc.);
- Coordination View – Used to specify the component behaviour;
- Algorithmic view – Used to express the sequence of actions executed by a component according to its current state, based on activity diagrams. This view considers periodic and non-periodic activities with the additional restriction that infinite loops are forbidden on periodic activities.

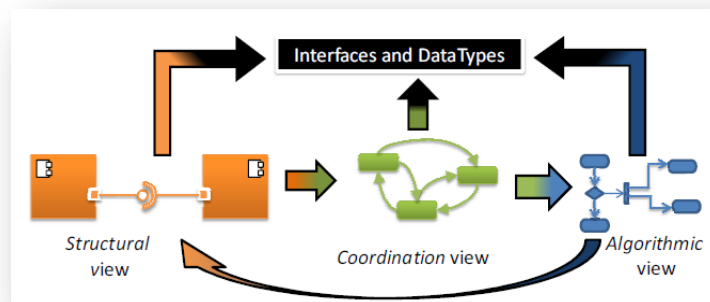


Fig. 79 - Schematic representation of the V3 CMM views

In order to ease the generation of executable code from the input V3 CMM model, an OO (Object-Oriented) framework was designed and implemented. Such framework provides an OO interpretation of the CBSD (Component-Based Software Development) concepts that allows translating the CB designs into OO applications.

With V3 CMM it is possible to describe the architecture of an application by the basis of its components and also by the behaviours and algorithms that are ran by those components, since V3 CMM defines a minimum set of elements that are needed to describe the applications architecture and ignores all the others that are unnecessary.

On the image below, Fig. 80, there is a schematic that shows the process that is done when developing robotic applications with V3 CMM.

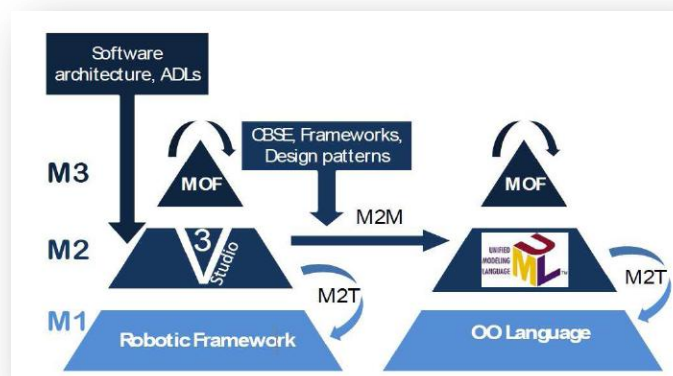


Fig. 80 - Process of developing a model with V3 Studio

Each level of the pyramid correspond to a model that forms the meta-model located on the correspondent higher level. The cycle finishes on the meta-model MOF (Meta Object Facility defined by OMG), which adjusts itself.

On the pyramid on the left, V3 CMM corresponds to the meta-model where it is built the elements that define the software architectures. Once the model V3 CMM is generated, it is possible to translate it directly to code by performing a M2T transformation or to a object-oriented model by performing a M2M transformation.

4.2.3 MinFr from the point of view of an end-user

Considering all that has been said on the previous sections, from the point of view of an end-user, a MinFr application is divided into three units:

- **Component Library** – Where it is defined the structure of each component used by the application, in terms of ports, regions, states, events and transitions. Each component has its own component library file, which is defined on a *.minfr* file on the folder *MinFrProjects*.
- **Application** – Where it is defined the application structure, in terms of the connections between the ports of the different components used by the application. The application structure file is defined on a *.minfr* file on the folder *MinFrProjects*.
- **Distribution** – Where it is defined the distribution of the all the regions of all the components used by the application into threads. The application distribution file is defined on a *.minfr* file on the folder *MinFrProjects*.

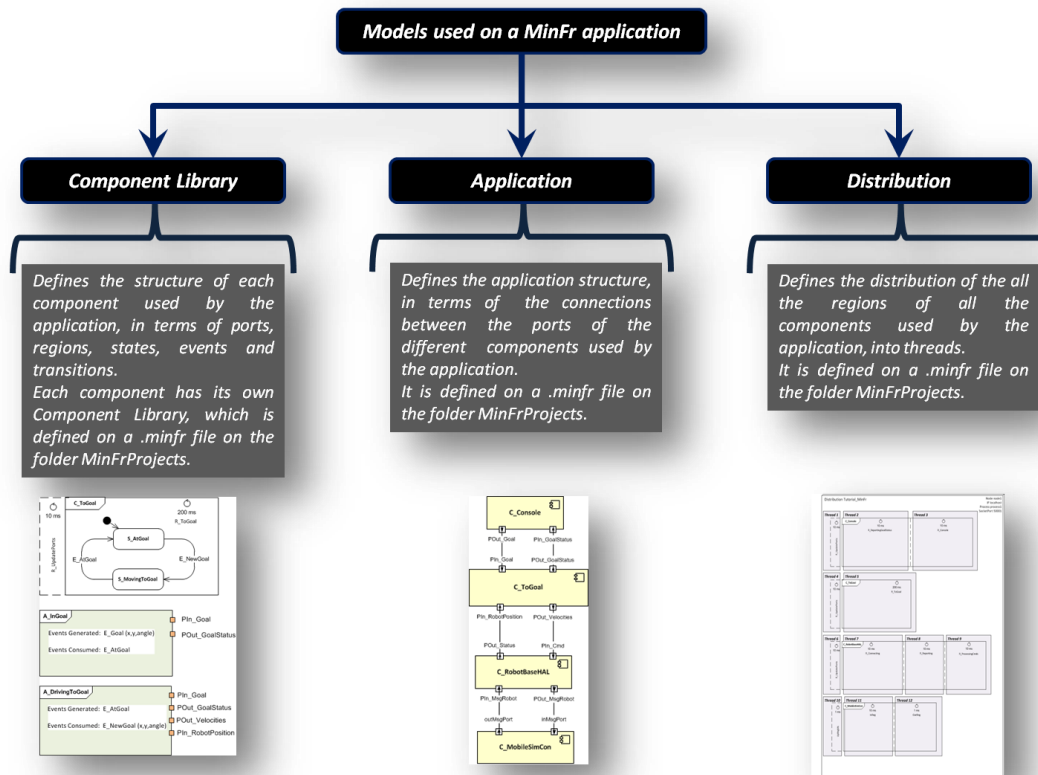


Fig. 81 - The models used on a MinFr application, from the point of view of an end-user

Each component on MinFr is composed by ports, regions, states, activities, events and transitions.

Ports are the interfaces of the components through which they send and receive data from other components.

Regions are orthogonal parts of the component that contain states and transitions. With orthogonal, it means that are concurrent parts of the same component, not depending on each other when it comes up to process them.

States are situations during which some invariant condition holds. This invariant condition can represents two types of situations:

- Static situation – Which correspond to situations such as an object waiting for some external event to occur;
- Dynamic situation – Which corresponds to situations such as the process of performing some action.

On MinFr, each state can only have one activity and needs to mention which events it generates and consumes.

Activity corresponds to the code that is processed when the correspondent state is active.

On MinFr, there is only one activity per state, and each activity is programmed on C++.

Activities can access the ports of the component to read and sent data through them, and are structured as a class that has four methods:

- *init* – Corresponds to the code sequence that is processed when the application is started, no matter if the state is active or not.
- *onEntry* – Corresponds to the code sequence that is processed every time the state becomes active.
- *tick* – Corresponds to the code sequence that is processed at each system iteration while the state is active.
- *onExit* – Corresponds to the code sequence that is processed every time the state becomes inactive.

It is on these methods that a MinFr user specifies what happens in each state in terms:

- Which data is read from the ports it has access to;
- How that data or other internal or external state data is processed;
- Which data is sent through the ports it has access to.

Events correspond to something that happens that affects the system. It only refers to the type of occurrence and can include some information regarding that occurrence. Events are consumed and generated by states in order to perform some action or to change from one state to another inside the same region.

For example, Keystroke is an event for the keyboard, but each press of a key is not an event but a concrete instance of the Keystroke event, which includes information about which key was pressed.

Transitions correspond to switching from one state to another because of an instance of an event. The events that cause transition from one state to another are called triggers.

On MinFr, each transition corresponds to a switch from a state to another and has a field named *ConditionsList*, where the user mentions the events that need to occur for that switch to happen.

A switch from a state to another means that the state that generated the event becomes inactive and the event that consumes it becomes active.

Now that is clear the structure of a MinFr application, it is time to explain the sequence that must be taken when it comes up to create a MinFr application, which is shown on the picture below, Fig. 82.

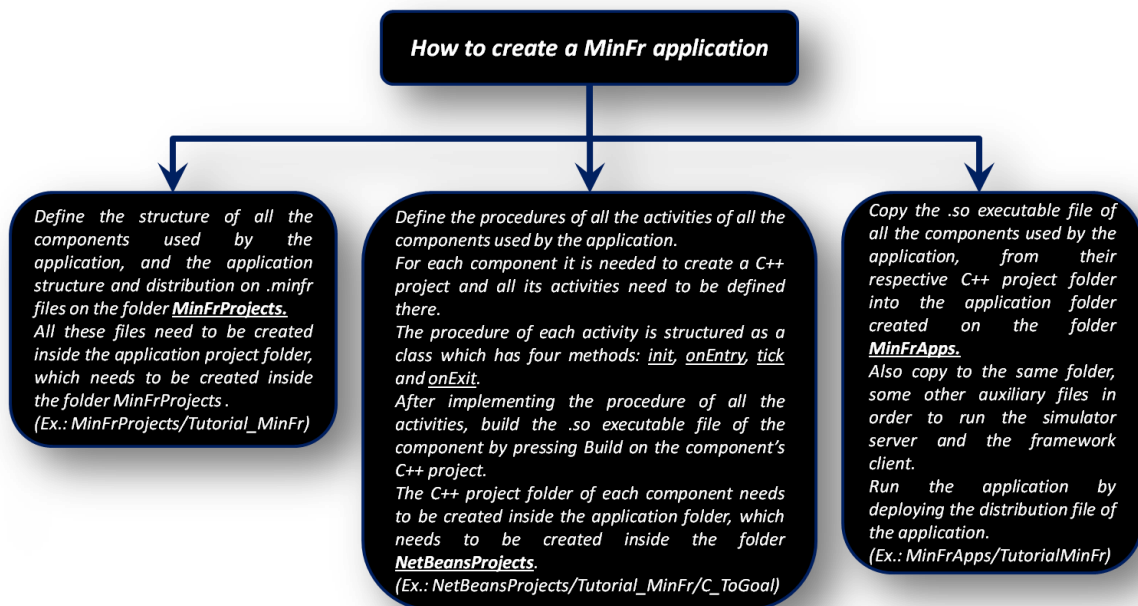


Fig. 82 - Diagram of how to create a MinFr application

The first step of the implementation of a MinFr application consist on three parts:

- 1st – Definition of the structure of all the components that are used on the application in components libraries, where each component has one and only one component library;
- 2nd – Definition of the application structure;
- 3rd – Definition of the application distribution.

As mentioned previously, all these definitions need to be implemented on .minfr files by using the program *Eclipse*, and all the resulting files need to be stored inside the folder *MinFrProjects*.

The second step consists on defining the procedures of all the activities of all the components that are used on the application by using the program *NetBeans*.

For each component, it is needed to create a C++ project and all the activities of that component need to be implemented there.

For each activity it is needed to have two files:

- A *.h* (header file) where it is declared the structure of the activity, which is the same for all the activities.
The activity structure consists of a class with four methods: *init*, *onEntry*, *tick* and *onExit*.
- A *.cpp* (source code file) where it is declared the body/procedure of each of those methods.

After defining the procedures of all the activities, of each component used by the application, it is needed to build the C++ project of each component in order to generate its *.so* executable file.

The third and last step consists of two parts:

1st – Copying the *.so* executable files of all the components from their respective C++ project folder into the application folder created on the folder *MinFrApps*.

2nd – Run the application by deploying the distribution file of the application.

On the section 6.2 there are explained in detail two MinFr applications: one which implements a simple architecture and that is used as tutorial of how to create an application with MinFr, and another that implements the hybrid architecture on a robot.

5 Application Design

This chapter describes the design of an application that implements the hybrid robotic architecture.

Like mentioned before on section 2.2.3, the hybrid robotic architecture is a combination of the hierarchical and reactive robotic architectures, which means, that basically it is a reactive architecture with planning, making it possible for the robot to react in real-time but also to plan the actions that it needs to perform.

On section 5.1, it is described the robotic platform considered as basis for the implementation of this application.

On section 5.2, it is described the requirements that the application must respect in order to work as expected.

On section 5.3, there are described the components that are needed to implement in order for the application to follow the requirements.

Finally, on section 5.4, there are described the auxiliary C++ libraries that are used by some components.

5.1 Robotic Platform Pioneer 3-AT

Pioneer is a well-known robots family, which are mostly used for education and research purposes.

The Pioneer robots consists on small two-wheel and four-wheel robots, such as, Pioneer 1, Pioneer AT, Pioneer 2-DX, -DXE, -DXf, -CE, -AT, Pioneer 2-DX8/DX8 Plus and – AT8/AT8 Plus, and the mobiles robots Pioneer 3-DX and Pioneer 3-AT.



Fig. 83 - Pioneer robots family

These robots share an architecture and software basis with the rest of the MobileRobots platforms, such as *AmigoBot*, *PeopleBot V1*, *Performance PeopleBot* and *PowerBot*.

The MobileRobots platforms establish standards for the mobile intelligent robots that contain all the basic components that allow them to move through the real world.

Nowadays, MobileRobots is used as a reference by a big variety of development projects.

All the MobileRobots robots contain a mobile basis with two motors, ultrasonic sensors, odometry, and some other optionally components such as, contact sensors and lasers. All these components are controlled by a microcontroller and the respective mobile robot server software. Apart from the microcontroller, each MobileRobots platform includes a host with some programming interfaces to control the robot, and some frameworks to develop robotic applications.

One of the programming interfaces developed by MobileRobots to sensor and control their various robot platforms is Advanced Robotics Interface for Application (ARIA), which is explained in detail on section 3.1.1.3.

The robot Pioneer 3-AT with embedded computer is an autonomous mobile robot. One of the advantages of this robot is its size, which allows it to navigation in small environments.

The robot used on this project as basis for the development of application is a Pioneer 3-AT with a laser, and its correspondent configuration programmed on the programming interface Advanced Robotics Navigation and Localization (ARNL).



Fig. 84 - Pioneer 3-AT

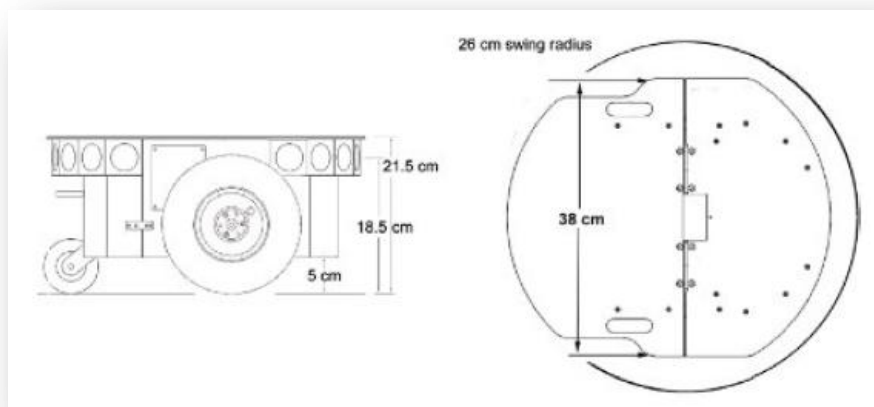


Fig. 85 - Pioneer 3-AT dimensions

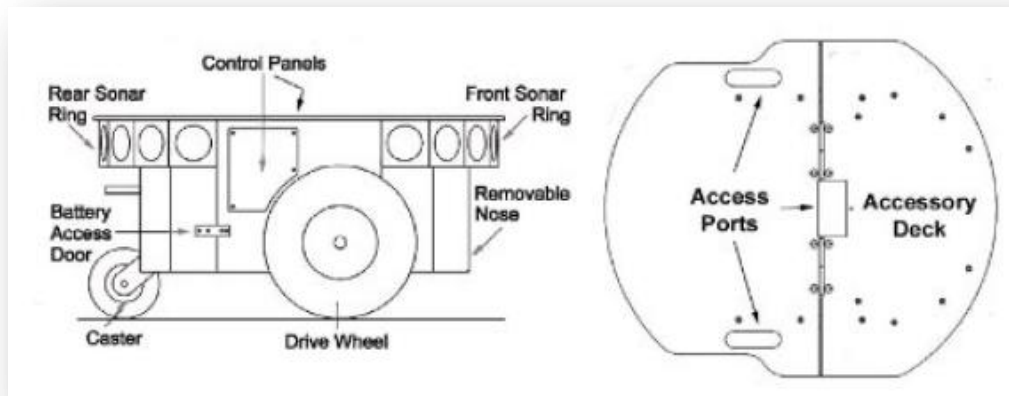


Fig. 86 - Pioneer 3-AT components

The MobileRobots platforms work on a client-server environment, where all the MobileRobots platforms perform the role of servers.

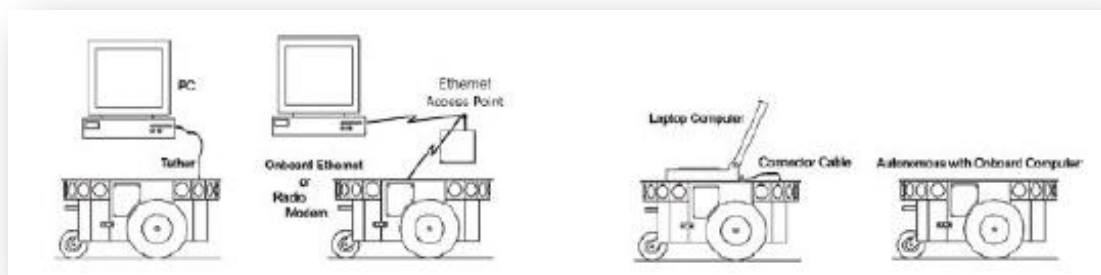


Fig. 87 - MobileRobots Client-Server environment

The MobileRobots platforms microcontroller controls all the mobile robotics low-level details, such as, maintaining the robot velocities and direction, acquisition of the sensors readings, and control of some other features such as a robot arm.

In order to complete the client-server environment, the MobileRobots platforms require a PC connection through a serial port. It is on the PC, where it is executed the software that allows the high-level control of the robot intelligence, like obstacle avoidance, path planning, localization, etc.

The big advantage of this client-server environment is that a different client can control various robots with the same software. Also, it is possible for various clients to control the same robot, which allows the distribution of tasks.

For more information about this robot, visit the webpage <http://robots.mobilerobots.com/wiki/Manuals> or check the manual of the robot.

5.2 Requirements

From the point of view of the application developer, the requirements that this application must respect in order to work as expected are:

- **R1** – The user must have a user interface where it is able to interact with the robot by ordering it to execute some missions, where each mission corresponds to a goal position (x, y and angle coordinates);
- **R2** – The user must be able to order as many missions as possible, in such a way that, they are all stored on a FIFO queue and executed one by one.
- **R3** – When getting sensor readings, the robot must compare them with the map it has of the environment, in order to check if new obstacles are detected and to update it;
- **R4** – The robot must have a localization system, based on user inputs or on the sensors readings, that corrects the readings sent by the encoders;
- **R5** – The robot must have a local navigation algorithm to guide the robot through the environment by avoiding the obstacles with the help of the sensors readings, and at the same time being able to reach the goal position;
- **R6** – The robot must have a global navigation algorithm to calculate the shortest path between the robot and the goal position with the help of a given map of the environment;
- **R7** – When executing a mission, if the robot detects new obstacles, it must use the local navigation algorithm, and if it does not detect new obstacles, it must use the global navigation algorithm;
- **R8** – The robot must have a path execution algorithm in order to process the calculated paths on the most efficient and fastest way possible;
- **R9** – In order to move the robot from one position to another, the robot must have an actuator to translate the distance from the robot to the goal position into velocities for the robot motors;

5.3 Components

In order to fulfill the application requirements, the following components need to be implemented by the component developers:

- **Console** – This component consists on the user interface, where the user can add new missions to queue, check the actual position of the robot, make emergency stops, and change the actual localization of the robot to a new one inserted by him. Here **R1** and **R4** are fulfilled, because the user can insert new missions and also the actual position of the robot, which corrects the one sent by the encoders;
- **Mission Planner** – This component consists on the implementation of the FIFO queue used for the missions. It receives new missions from the console and stores them on the queue, as well as, if there are more to do, it sends a new mission to the **Executor** once the previous one that was sent is completed. Here **R2** is fulfilled;

- **Obstacles Detector** – This is the component that detects if new obstacles are detected on the environment according to the given map of the environment and the sensor readings that it receives from the robot sensors.
If new obstacles are detected, it updates the map of the environment with those new obstacles. Here **R3** is fulfilled;
- **Executor** – This is the decision-making component, since with the sensor readings it receives from the robot sensors, it checks if there are new obstacles, updates the map of the environment with those new obstacles, and decides which navigation algorithm to use, if global or local. The output of both navigation algorithms are paths which must be directly sent to the path execution component. Here **R7** is fulfilled;
- **Astar** – This component consists on the implementation of the A* global navigation algorithm, which calculates the shortest path between two points of a given map. Here **R6** is fulfilled;
- **VFH** – This component consists on the implementation of the VFH local navigation algorithm, which calculates a path from the robot to the goal position by avoiding the obstacles on the environment. Here **R5** is fulfilled;
- **Pure Pursuit** – This component consists on the implementation of the Pure Pursuit path execution algorithm, which uses a lookahead distance factor to select the next position on the path to send to the robot actuators in order to be executed. Here **R8** is fulfilled;
- **To Goal** – This component consists on the implementation of the Siegwart equations acting algorithm, which translates the distance from the robot to the goal position into velocities to the robot motors by considering its control constants and constraints. Here **R9** is fulfilled;
- **Localization** – This component is responsible for sending the updated position of the robot to all the needed components. Once the user inserts a updated position by console, this component calculates the difference between that position and the one sent by the encoders. For the following positions sent by the encoders, that difference will be summed in order to correct the position and the resulting position will be sent to all the needed components.

The way these components must be connected and the how they must interact between themselves is shown on the image below, Fig. 89, which shows the application structure in terms of the components and connections between them.

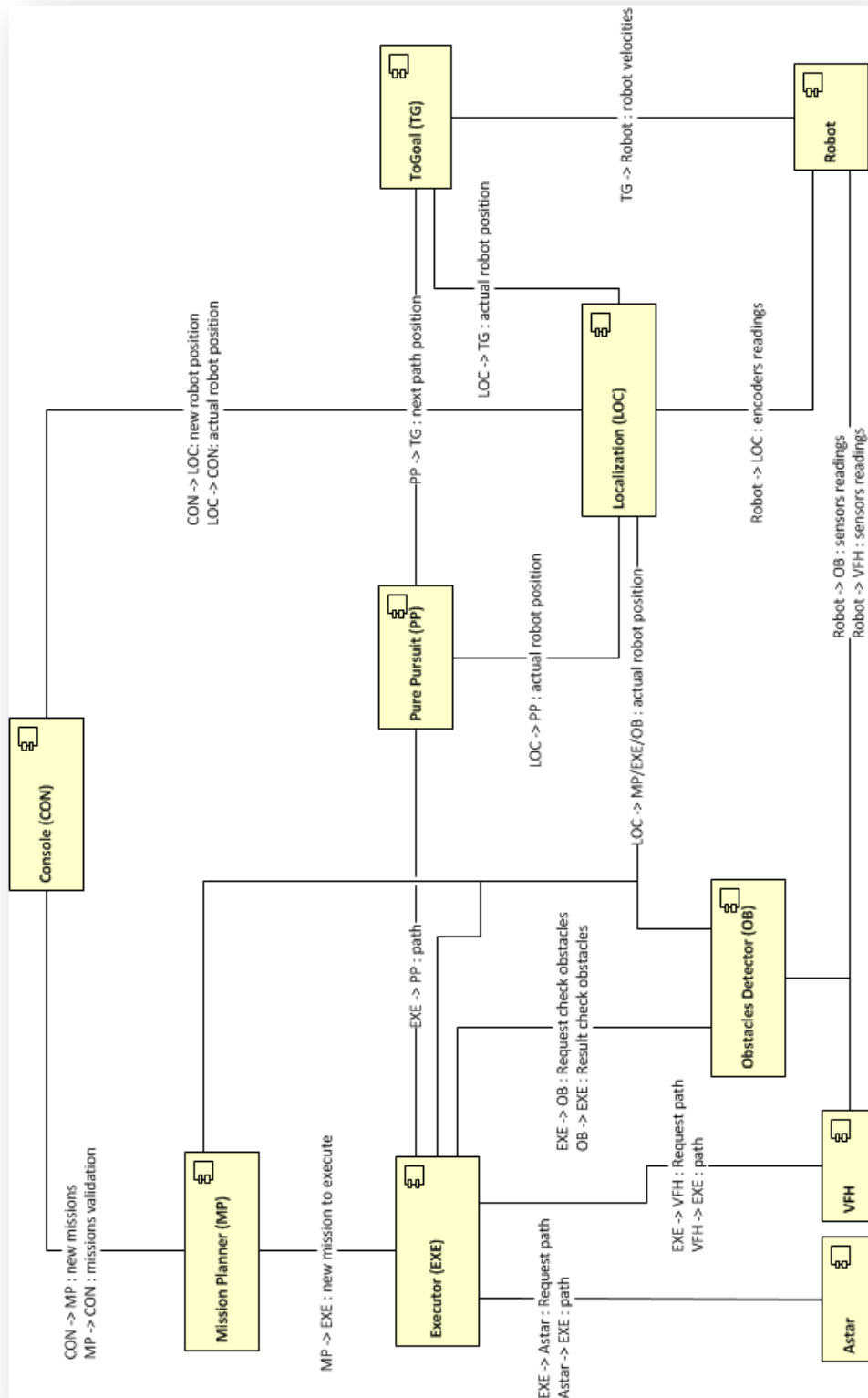


Fig. 88 - Structure of the application in terms of components and connections between them

5.4 Algorithms

In order to implement the components that were mentioned on the previous section, it is needed to implement some auxiliary libraries that will perform the tasks of some robotic subsystems.

On this chapter it will be explained their implementations.

All the auxiliary libraries are implemented on C++.

5.4.1 CommonFunctions

This library includes some required data structures and data conversion functions.

Its implementation consists on the following structure:

```
double Gra2Rad(double deg);

double Rad2Gra(double rad);

struct positionCoords{
    int x;
    int y;
    int angle;
};

struct velocities{
    double v;
    double w;
};

struct robotData{
    double Kv;
    double Kw;
    double Vmax;
    double Wmax;
    double DistanceThresholdError;
    double AngleThresholdError;
};

struct laserData{
    std::string Readings;
    int MaxAngle;
    int MinAngle;
    double Threshold;
};
```

The functions *Gra2Rad* and *Rad2Gra* are used to convert from degrees to radians and vice-versa, respectively.

The data structure *positionCoords* is used to store the coordinates of a position in terms of x, y and angle.

The data structure *velocities* is used to store the two types of velocities that are sent to the robot, translational (v) and angular (w).

The data structure *robotData* is used to store all the information related with the constants and constraints used by the algorithm *ToGoal* when it comes up to calculate the robot velocities according to the goal and the robot actual positions.

The data structure *laserData* is used to store all the information related with the laser such as its readings, the maximum and minimum angles of those readings, and the maximum possible value of a reading.

5.4.2 MapOperator

This library includes all the functions that are needed in order to work with a given map.

It has many methods that read, change and write maps from and to *.txt* files, where all the positions values are between 0 and 255. Here, a position is considered obstacle if its value is higher or equal to 128 and not obstacle if it is smaller.

Its implementation mainly consists on the following C++ class:

```
class MapOperator{  
  
private:  
    unsigned int NrRows;  
    unsigned int NrColumns;  
    double ** Map;  
    double ** DilatedMap;  
    unsigned int NrCellsDilation;  
  
public:  
    void OpenOriginalMap();  
    void OpenDilatedMap();  
    void ReadMapSizeFile();  
    void ReadOriginalMap();  
    void ReadDilatedMap();  
    unsigned int MaximumRow();  
    unsigned int MaximumColumn();  
    unsigned int MinimumRow();  
    unsigned int MinimumColumn();  
  
    void SaveMaps();  
    void WriteOriginalMap();  
    void WriteDilatedMap();  
  
    void SetMapPositionValue(unsigned int, unsigned int, double, unsigned int);  
    bool PositionIsObstacle(unsigned int, unsigned int, unsigned int);  
  
    void GenerateDilatedMap(unsigned int);  
    void DilateObstacle(unsigned int, unsigned int, unsigned int);  
    void DilateObstacleMap(unsigned int, unsigned int, unsigned int);  
  
    bool NewObstacles(positionCoords, laserData, double);  
    positionCoords GetCoords(double, double, positionCoords);  
  
    void PrintMap(unsigned int);  
  
};  
  
positionCoords RobotCoordsToMapPosConverter(positionCoords coords);  
positionCoords MapPosToRobotCoordsConverter(positionCoords pos);  
bool CoordsValidation(positionCoords coords, double coordsThreshold);
```

The methods and variables that are on the public part of the class are the ones the user can access to. The rest, the ones on the private part, are only used by the class on the calculations and the user can't access their values neither change them.

Here, it will only be explained the public part of the class, which consists of the following methods:

- *void OpenOriginalMap()* – It calls the two other methods *ReadMapSizeFile()* and *ReadOriginalMap()*, in order to read the size of the original map and the map itself from two *.txt* files to the correspondent variables;
- *void OpenDilatedMap()* – It calls the two other methods *ReadMapSizeFile()* and *ReadDilatedMap()*, in order to read the size of the dilated map and the map itself from two *.txt* files to the correspondent variables;
- *void ReadMapSizeFile()* – It reads the size of the maps (which is the same for both the original and dilated map), in terms of number of rows and columns, from a *.txt* file named *MapSize* to the correspondent variables *NrRows* and *NrColumns*;

- *void ReadOriginalMap()* – It reads the original map, which size is read by the previously mentioned method *ReadMapSizeFile()*, from a *.txt* file named *MapData* to the variable *Map*;
- *void ReadDilatedMap()* – It reads the dilated map, which size is read by the previously mentioned method *ReadMapSizeFile()*, from a *.txt* file named *DilatedMap* to the variable *DilatedMap*;
- *unsigned int MaximumRow()* – It returns the maximum size the maps have in terms of rows;
- *unsigned int MaximumColumn()* – It returns the maximum size the maps have in terms of columns;
- *unsigned int MinimumRow()* – It returns the minimum size the maps have in terms of rows, which usually is 0;
- *unsigned int MinimumColumn()* – It returns the minimum size the maps have in terms of columns, which usually is 0;
- *void SaveMaps()* – It calls the two other methods *WriteOriginalMap()* and *WriteDilatedMap()*, in order to save the values of the original and dilated maps that are stored on its correspondent variables on the two correspondent *.txt* files;
- *void WriteOriginalMap()* – It saves the values of the original map that are stored on the variable *Map* on the *.txt* file named *MapData*;
- *void WriteDilatedMap()* – It saves the values of the dilated map that are stored on the variable *DilatedMap* on the *.txt* file named *DilatedMap*;
- *void SetMapPositionValue(unsigned int, unsigned int, double, unsigned int)* – It changes the value of a position on one of the maps.
The first two input values correspond to the x and y coordinates of the position, the third input value corresponds to the new value of that position, and the fourth and last input value correspond to the map where the position value needs to be changed, 1 if it is on the original map or 2 if it is on the dilated map.
If it is to change a position value from not obstacle to obstacle on the dilated map, also the method *DilateObstacle()* is called;
- *bool PositionIsObstacle(unsigned int, unsigned int, unsigned int)* – It checks the value of a position on one of the maps and it returns true if the position is an obstacle or false if it is not.
The first two input values correspond to the x and y coordinates of the position, and the third and last input value correspond to the map where the position value needs to be checked, 1 if it is on the original map or 2 if it is on the dilated map;
- *void GenerateDilatedMap(unsigned int)* – It generates a map from the original map, where all the obstacles are dilated by a given dilation factor, which is given as input.
To dilate each obstacle, it is called the method *DilatedObstacleMap()*;
- *void DilateObstacle(unsigned int, unsigned int, unsigned int)* – It dilates an obstacle on the dilated map by a given dilation factor, in a way that all the positions that have a row or column difference from the obstacle position equal or smaller than the factor are also set as obstacles.

The first two input values correspond to the x and y coordinates of the obstacle position, and the third and last input value correspond to the dilation factor;

- *void DilateObstacleMap(unsigned int, unsigned int, unsigned int)* – It dilates an obstacle on the dilated map by a given dilation factor, in a way that all the positions that have a row or column difference from the obstacle position equal or smaller than the factor are also set as obstacles.

The first two input values correspond to the row and column coordinates of the obstacle position, and the third and last input value correspond to the dilation factor;

- *bool NewObstacles(positionCoords, laserData, double, double)* – It checks if new obstacles are detected from the ones mentioned on the dilate map. It returns true if new obstacles were detected and false if not.

It does so by checking the laser readings according to the robot position. If from those laser readings new obstacles are detected, the position value of those obstacles is changed to obstacle and the dilation operation is also performed.

The first input value corresponds to a structure with the x, y and angle coordinates of the robot position, the second input value corresponds to other structure with the laser readings and information, and the third and fourth input values correspond to the laser and coordinates error threshold, respectively.

Those two last values, laser and coordinates error threshold, are used in order to remove the invalid laser readings;

- *positionCoords GetCoords(double, double, positionCoords)* – It calculates and returns the coordinates of the laser reading according to the robot position.

The first input value corresponds to the angle of the laser reading, the second input value corresponds to the laser reading, and the third and last input value corresponds to a structure with the x, y and angle coordinates of the robot position;

- *void PrintMap(unsigned int)* – It prints one of the maps on the screen.

The input value corresponds to the map to be printed, 1 if it is the original map or 2 if it is the dilated map;

On this auxiliary library there are also two functions named *RobotCoordsToMapPosConverter()* and *MapPosToRobotCoordsConverter()*, which convert the position given by the robot to a map position, and vice versa, respectively.

Finally, there is another function named *CoordsValidation()*, which receives as input a position and a value that corresponds to the coordinates laser threshold, mentioned before on the explanation of the method *NewObstacles()*, that checks if the position that was considered as the coordinates of the laser reading is valid or not according to that value of threshold.

This auxiliary library is used by all the algorithms and components that need to work with a map, such as the algorithm A* and the components MissionPlanner, ObstaclesDetector and Executor, which will be later mentioned on the chapter Implemented Applications.

Note that, some methods of this auxiliary library are implemented on different ways depending on the components where they are used. For example, the method *bool NewObstacles()* is implemented correctly on the version of this auxiliary library used by the component ObstaclesDetector.

5.4.3 A*

This library corresponds to the algorithm that is explained on section 2.6.4.1. It is an algorithm that is used for calculating the shortest path from one point to another on a given map.

As input, this algorithm must receive the robot and the goal positions, as well as a map, which needs to be on a *.txt* format, and as output, it gives a string with the calculated shortest path, in terms of map positions, from the robot to the goal position.

Its implementation consists on the following C++ class:

```
class Astar{
private:
    MapOperator Map;
    unsigned int IndexQueue;
    unsigned int IndexPathArray;
    positionCoords robot;

    positionCoords goal;

    positionCoords initialPosition;
    positionCoords finalPosition;
    bool GoalReached;
    bool InvalidRobotPosition;
    bool InvalidGoalPosition;
    bool InvalidPath;
    double ** Queue;
    double ** PathArray;

    std::string PathString;
    void ResizeQueue();
    void ResizePathArray();
    void DeleteDynamicVariables();
    double NrPosition(double, double);
    double NrVisitedPositions();
    void CalculateDistanceFromNeighbours(int);
    void SelectMinimumDistance();
    void CalculatePath();

public:
    Astar();
    int Main(positionCoords, positionCoords);
    std::string Path();
};
```

The methods and variables that are on the public part of the class are the ones the user can access to. The rest, the ones on the private part, are only used by the class on the calculations and the user can't access their values neither change them.

Here, it will only be explained the public part of the class, which consists of three methods only:

- Astar() – It is the constructor of the class Astar, which is automatically called every time an object of this class is created.
Its process consists only on calling the function OpenDilatedMap() from the object *Map* from the class *MapOperator* in order to read the map from its *.txt* file into a 2D array.
- int Main(positionCoords, positionCoords) – It receives as input the robot and goal positions, respectively, and starts by verifying if both are valid positions or not. This validation consists on verifying if they are positions of the map and if they are not obstacles.
If both are valid, then it starts the algorithm process, which consists on calculating the shortest distance to the robot position from the positions that have at least one

adjacent neighbor position with its status set as visited, and selecting from those, the position that has the smallest value from the sum between that value and the euclidean distance from that to the goal position.

When a position is selected, its status changes to visited.

This algorithm process starts with only one position with its status set as visited, the robot position, and finishes when the goal position has its status changed to visited.

When the algorithm finishes, the path from the robot to the goal position is calculated.

The output of this method is of type integer and it returns the result of the path calculation, if it was or not possible to calculate.

- *string Path()* – It only returns the calculated path on a string, where the first field corresponds to the size of that path in terms of map positions and the rest are those map positions.

Every time an user wants to calculate the shortest path from an initial to a final position of the map with this algorithm, it must start by creating an object of this class followed by a call to the method *Main()*, where it puts as input the initial and final positions, and finally, call the method *Path()* to obtain the calculated shortest path between those two positions.

In summary, the process of calculation of the shortest path is controlled by the method *Main*, but the process itself is not all done there but divided between many functions that are declared on the private part of the class. On the image below, Fig. 89, it is shown the state chart diagram of the implemented A* algorithm.

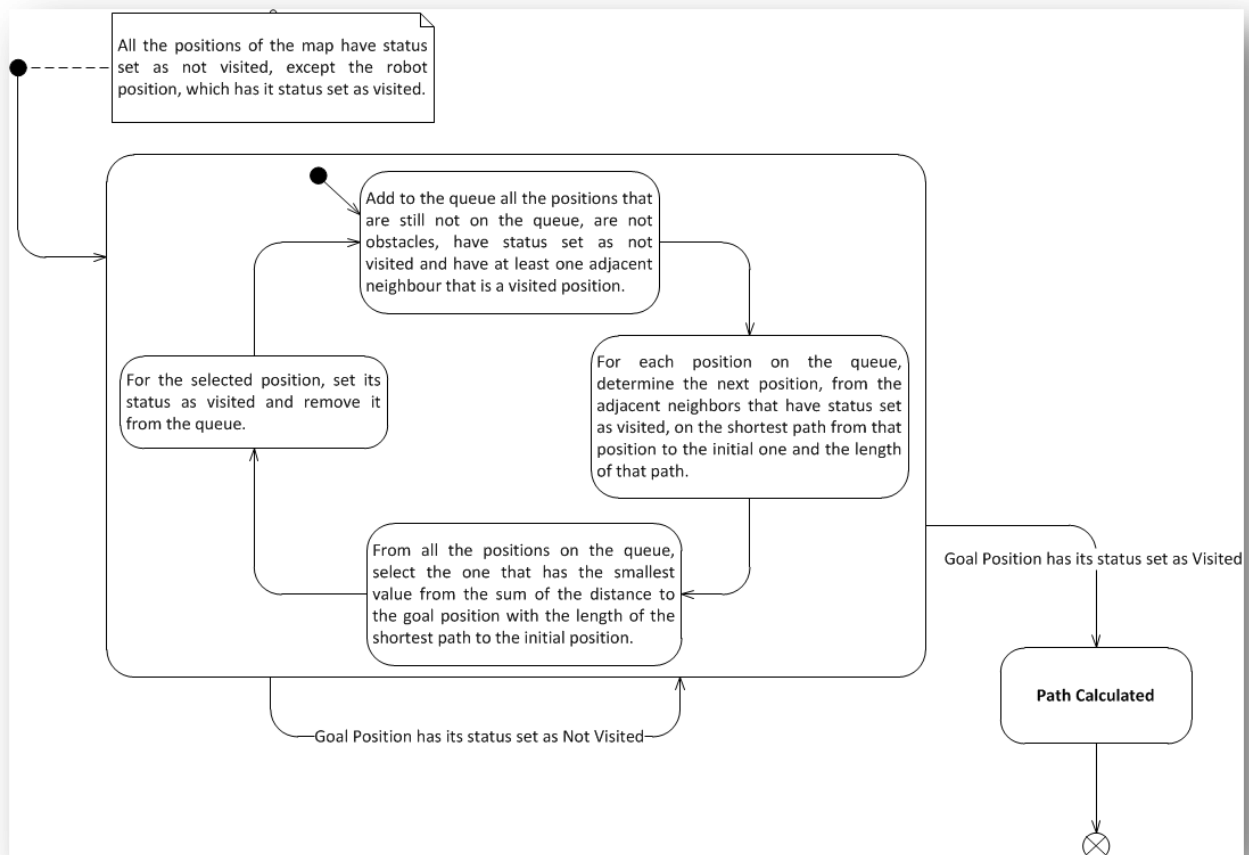


Fig. 89 - Statechart of the implemented A* algorithm

5.4.4 VFH

This library corresponds to the algorithm that is explained on section 2.5.1. It is an algorithm that is used for calculating the path from the robot to the goal positions, by avoiding unknown obstacles on the environment.

As input, this algorithm must receive the robot and the goal positions, as well as the laser readings, in order to know what is around it.

As output, it gives a string with the calculated path, from the robot position to the goal position.

Its implementation consists on the following C++ class:

```
class VFHAlgorithm{

public:
    VFHAlgorithm(positionCoords, positionCoords);
    VFHAlgorithm(positionCoords, positionCoords, laserData);
    ~VFHAlgorithm();
    std::string FindPath();
    laserData ReadLaser(int, int, int, int);
    int GetStatus();

private:
    int** DetectGaps();
    int** OrganizeGaps();
    std::string ChoosePath();
    void ResizeArraySub();
    void ResizeArrayGap();
    void ResizeNewPath();
    void ResizePath(int, int);
    void DeleteDynamic();
    void ReadMap();
    void PrintMap();
    int* getXy(double, double);

    positionCoords Goal, AbsGoal, Robot;
    int Ndist;
    double Thres, MaxAng, MinAng, MinGap;
    int *Dist, **sub, **gap, **path;
    int n_sub, n_gap, n_path, n_nodes;
    int Status;
    std::string LDR, Solut;

    unsigned int NrRows;

    unsigned int NrColumns;
    double ** Map;

};
```

The methods and variables that are on the public part of the class are the ones the user can access to. The rest, the ones on the private part, are only used by the class on the calculations and the user can't access their values neither change them.

The way this algorithm is implemented is not exactly the same as the one explained on section 2.5.1, which uses a threshold value to detect the gaps for the robot to go through.

Here, to detect gaps, it is used derivatives, in such a way that when the laser values start to increase it is considered an ascent, and when the laser values start to decrease it is considered a descent. Each gap then needs to be formed by an ascent and a descent.

To detect the derivatives, this algorithm looks on the histogram of the laser readings for the level changes between adjacent readings.

On the image below, Fig. 90, it is shown the state chart diagram of the implemented ToGoal algorithm.

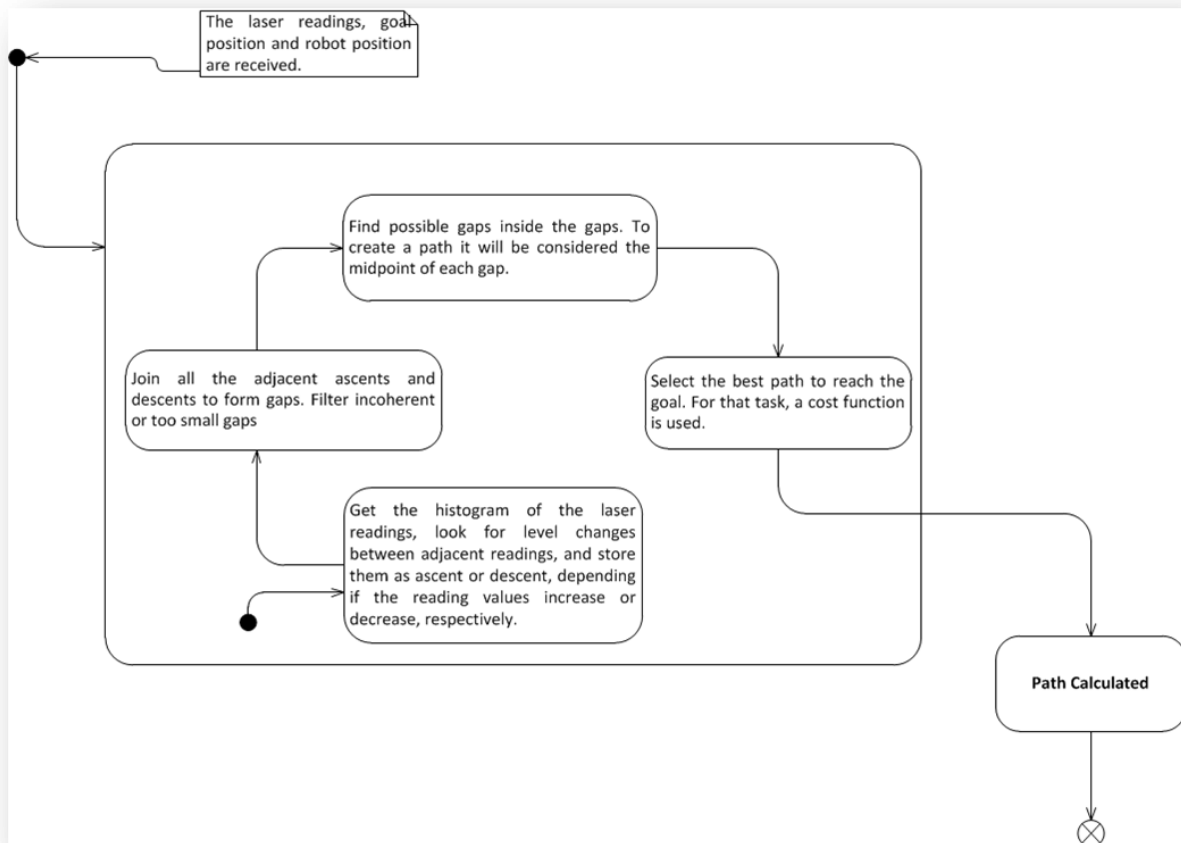


Fig. 90 - Statechart of the implemented VFH algorithm

5.4.5 To Goal

This library corresponds to the algorithm that is explained on section 2.7.2.1. It is an algorithm that is used for controlling the robot motors by setting both translational and angular speeds of the robot at each instant.

As input, this algorithm must receive the robot current position, the robot control constraints and constants, and the goal position that is to move the robot to, and as output, it gives the velocities that must be sent to the robot motors to perform such movement.

This is an algorithm that must be called periodically so that it updates the robot velocities according to the robot current position.

Its implementation consists on the following C++ class:

```

class ToGoal{

private:
    robotData RobotConstraints;
    bool robotStopped;
    bool goalReached;
    double angleGoal(double, double);

public:
    ToGoal(robotData);
    bool RobotAtPosition(positionCoords, positionCoords);
    velocities CalculateVelocities(positionCoords, positionCoords);

};

```

The methods and variables that are on the public part of the class are the ones the user can access to. The rest, the ones on the private part, are only used by the class on the calculations and the user can't access their values neither change them.

Here, it will only be explained the public part of the class, which consists of three methods only:

- ToGoal(robotData) – It is the constructor of the class ToGoal, which is automatically called every time an object of this class is created.
It must have as input value a structure with the robot control constraints and constants.
- bool RobotAtPosition(positionCoords, positionCoords) – It checks if the robot is already at the desired goal position. It returns true if is yes and false if not.
As input values it must have two structures with the robot current position and the goal position.
- velocities CalculateVelocities(positionCoords, positionCoords) – It receives as input the robot current position and the goal position, and it calculates the translational and angular velocities that are to send to the robot according to the current distance from the robot to the goal.
The output of this method is a structure with those two velocities.
If previously the robot was stopped, this algorithm starts by only returning angular velocities, in order for the robot to be facing the goal position before moving to it.
If previously the robot was already moving towards some position, the algorithm returns the angular and translational speeds needed for the robot to move to the current goal position.
The robot is considered as moving when it is moving with translational speed towards some goal position.

In summary, every time an user wants to control a robot with this algorithm, it must start by creating an object of this class, where it puts as input the robot control constraints and constants, and then, make periodic calls to the method RobotAtPosition() to know if the robot is already at the current goal. If yes, it is not needed to do anything. If not, call the method CalculateVelocities() in order to calculate the robot velocities.

On the image below, Fig. 91, it is shown the state chart diagram of the implemented ToGoal algorithm.

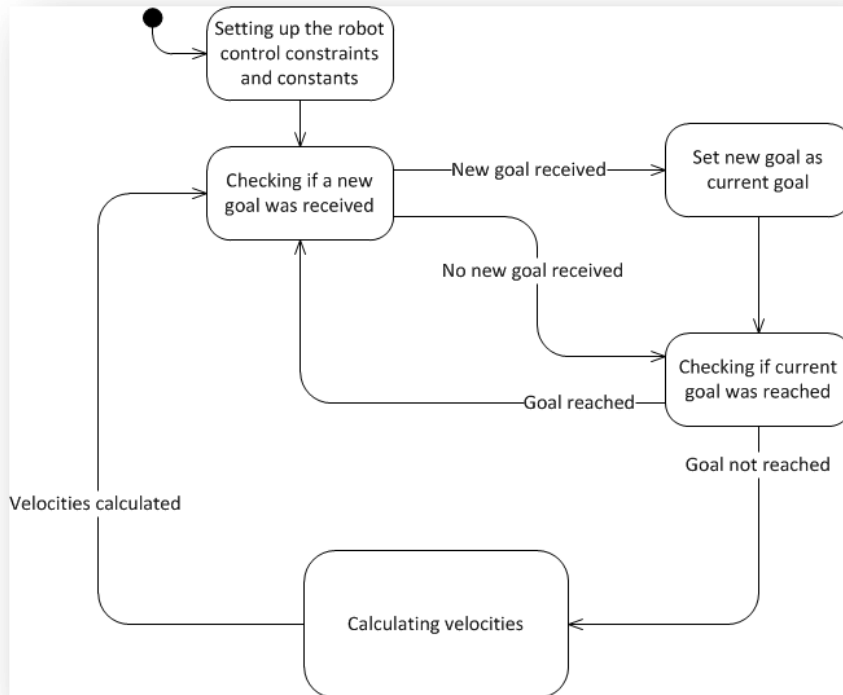


Fig. 91 - Statechart of the implemented ToGoal algorithm

5.4.6 Pure Pursuit

This library corresponds to the algorithm that is explained on section 2.7.1.1. It is an algorithm that is used for path execution in a way that it selects a position from the given path that is at a predefined lookahead distance from the robot actual position.

As input, this algorithm must receive the robot position, the path and the predefined lookahead distance that is to be considered by the algorithm when calculating the next target position of the robot.

As output, it gives a position that belongs to the path received as input, and that is at a predefined lookahead distance from the robot actual position.

Its implementation consists on the following C++ class:

```

class PurePursuit{
private:
    double MaxLookAheadDistance;
    int IndexConsideredPosition;
    bool GoalReached;
    bool LookAheadGoalFound;

    double LookAheadDistance;

    int SizePath;
    double (*Path)[2];
    double FinalAngle;
public:
    PurePursuit(std::string, double);
    bool RobotAtGoal();
    positionCoords CalculateNextPosition(positionCoords);
    ~PurePursuit();
};

```

The methods and variables that are on the public part of the class are the ones the user can access to. The rest, the ones on the private part, are only used by the class on the calculations and the user can't access their values neither change them.

Here, it will only be explained the public part of the class, which consists of three methods only:

- *PurePursuit(string, double)* – It is the constructor of the class PurePursuit, which is automatically called every time an object of this class is created.

It must have as input values the path that is to be processed and the predefined lookahead distance.

The format of the path must be a string, in which the value it contains at its first position is the size of the path in terms of positions, and the rest are the respective positions, x and y coordinates, that compose the path from the robot initial position to the goal. The last position of the path, the goal, also includes the final orientation of the robot.

This method process consists only on copying the path positions from the string to a 2D array and on setting all the initial values of the algorithm.

- *bool RobotAtGoal()* – It returns as output the current status of the algorithm, if the path final goal was already reached or not.
- *positionCoords CalculateNextPosition(positionCoords)* – It receives as input the robot current position, and it calculates the next position on the path that is at distance equal to the predefined lookahead distance from the robot current position.

The output of this method is that next path position. Once the distance from the robot current position to the last position of the path, the goal, is smaller than the predefined lookahead distance, this method returns the last position of the path.

- *~PurePursuit()* – It is the destructor of the class PurePursuit, which is called automatically every time an object of this class is deleted.

Its process consists on deleting the dynamic variables that were used by the object.

In summary, every time an user wants to execute a path with this algorithm, it must start by creating an object of this class, where it puts as input the path and the predefined lookahead

distance, and then, make periodic calls to the method *CalculateNextPosition()* to get the next position on the path that the robot must go to, until the robot arrives at the goal position.

On the image below, Fig. 92, it is shown the state chart diagram of the implemented Pure Pursuit algorithm.

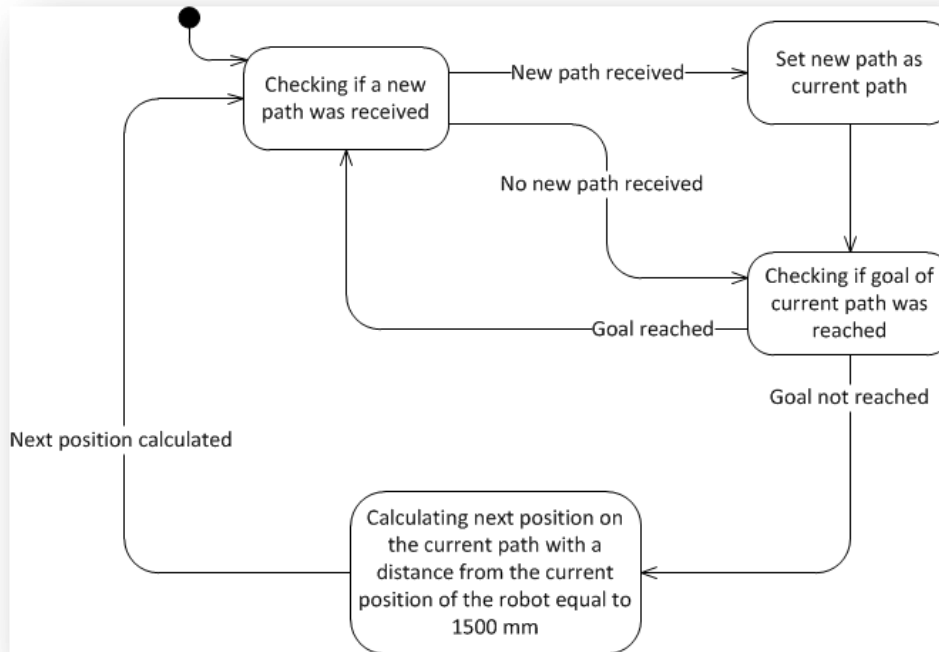


Fig. 92 - Statechart of the implemented Pure Pursuit algorithm

6 Application Implementation

On this chapter, it is explained how to implement the hybrid architecture application described on chapter 5 is implemented on the two robotic frameworks described on chapter 4, Smartsoft and MinFr.

6.1 Smartsoft

To be able to work with Smartsoft in order to know how to install it, and create an application and run it on both the simulator and real robot, it was used as guide the project *Programación de un robot Pioneer utilizando el framework SmartSoft* [Ardil12].

Like mentioned before, on section 4.1.1, from the point of view of the end-user, Smartsoft is divided into four units:

- **Communication Objects** – It is the data that is transmitted by the components through the communication ports.
- **Components** – Are the reusable units that interact between themselves by interchanging communication objects through their communication ports, offering and requesting services.
- **Communication Ports** – Help the components and applications developer to create and use components in a way that the semantic of the interfaces is defined by the communication patterns.
Each communication port is composed by two complementary parts named service requestor and server provider, which are implemented at different components.
- **Deployment** – It is a project where the application is designed in a way that it makes use of the components and implements the communication links between them to make the application work.

On the development of an application with Smartsoft there are then three tasks/phases:

- Implementation of the communication objects;
- Implementation of the components and its communication ports;
- Implementation of the deployment.

On the following sections it will be explained how to implement the application that is described on chapter 5, by following this approach.

The workspace that was used to implement the application was ws-mdsd.

6.1.1 Implementation with the Player/Stage simulator

On this section it will be described how it was implemented the hybrid architecture application in order for it to work with the robot simulator Player/Stage.

The goal of this application is to fulfill the requirements mentioned on the previous chapter by implementing an application with the also mentioned components.

On the following subsections it will be explained which communication objects and components were implemented, as well as how to create the deployment and how to run the application.

6.1.1.1 Communication objects

On this subsection it will be explained which communication objects were implemented for the hybrid architecture application.

Following what was implemented on the section 6.1.1 of [Ardil12], some communication objects were implemented in order to send the needed data between the different components of the hybrid architecture application.

On the image below, Fig. 93, it is shown the implemented communication objects.

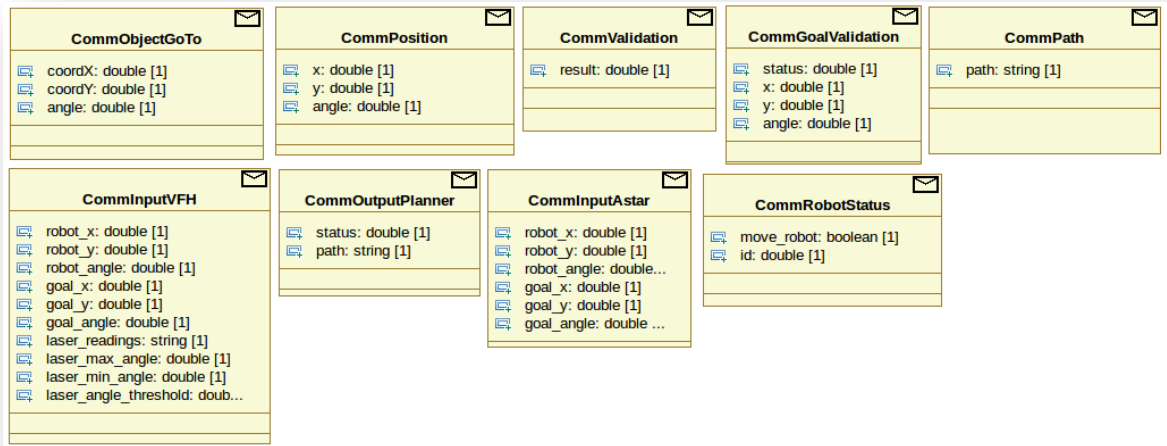


Fig. 93 - Implemented communication objects for the hybrid architecture application

The functionality of each communication will be explained on the next section with the explanation of the components where they are used, respectively.

The implementation of those communication objects followed the same procedure that is described on the section 6.1.1 of [Ardil12], which explains how the communication object *CommObjectGoTo* was implemented, with the only difference that the names and type of data is different for each communication object implemented here.

6.1.1.2 Components

Like mentioned before on chapter 5, in order to fulfill the requirements of this application, some components need to be implemented.

On this subsection, it is explained how each of those components is implemented on Smartsoft. The code of all the components is attached to this project on the chapter 8.

6.1.1.2.1 UPCTcompConsole

This Smartsoft component corresponds to the Smartsoft implementation of the Console component mentioned on chapter 5.

On the image below, Fig. 94, it is shown its structure.

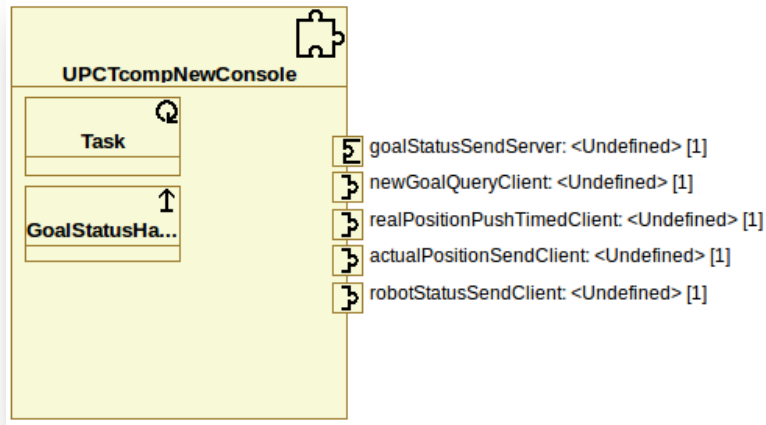


Fig. 94 - Structure of the Smartsoft component *UPCTcompConsole*

This component procedure is mainly controlled by the *SmartTask Task*, which corresponds to the cyclic task that includes all the code that is processed at each cycle of this component.

Since this component corresponds to the user interface, its mainly consists of a menu with various options, as it is shown below.

```
cout << endl << "                *** MENU ***                " << endl << endl;
cout << " > 1 - Add a mission to the missions queue." << endl;
cout << " > 2 - Check the actual position of the robot." << endl;
cout << " > 3 - Insert manually the actual position of the robot." << endl;
cout << " > 4 - Emergency stop." << endl;
cout << " > 0 - Exit the program." << endl;
cout << endl << " > Choose one option: ";
```

Following, it is explained the procedure of each menu option:

- The first menu option corresponds to the addition of a new mission to the missions queue, which is done by requesting the mission goal coordinates to the user and then sending it to the **UPCTcompMissionPlanner** component through the port **newGoalQueryClient**.
newGoalQueryClient is a port of type query, a bidirectional communication pattern, and it is the client part of this communication, which means that the task needs to wait for an answer after sending each goal request, where that answer corresponds to the goal validation value. If the goal validation value is equal to 1, the goal is valid and was added to the queue, if not, the goal is invalid.
The communication object used by the port **newGoalQueryClient** for the requests is *CommPosition* and the one used for the answers is *CommValidation*.
- The second menu option corresponds to the verification of the actual robot position, which is done by reading the last value that is at the communication port **realPositionPushTimedClient**.

That port is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, it is the client part of this communication, and the data that is received through it comes from the **UPCTcompLocalization** component and corresponds to the robot actual position.

The communication object used by the port *realPositionPushTimedClient* is *CommPosition*.

- The third menu option corresponds to the correction of the robot actual position by adding it manually.

It starts by showing the current robot position, which is done by reading the last value that is at the communication port *realPositionPushTimedClient*, and then it asks the user to insert the correct robot actual position coordinates, which are sent to the component **UPCTcompLocalization** through the port *actualPositionSendClient*.

The port *actualPositionSendClient* is of type send, a unidirectional communication pattern, it is the client part of this communication, and the communication object that it uses to send the correct actual position coordinates is *CommPosition*.

Another feature of this menu option is that when the user chooses this option, a message is sent to the component **UPCTcompActuator** through the port *robotStatusSendClient*, in order to stop the motors, so that the application can recalculate the path to the goal position that is being currently considered, by considering the corrected actual robot position.

After the user inserting the correct robot actual position, another message is sent through the same port to reactivate the robot motors.

The port *robotStatusSendClient* is of type send, unidirectional communication, it is the client part of this communication, and the communication object that it uses is *CommRobotStatus*.

- The fourth menu option corresponds to the emergency stop of the robot. When the user chooses this option, a message is sent to the component **UPCTcompActuator** through the port *robotStatusSendClient* in order to perform the emergency stop, which is done by stopping the motors.

The task then waits the user to press any key followed by an Enter, in order to reactive the robot motors by sending another message through the same motor.

- The last menu option should perform the exit of the application but it is not implemented yet.

This Smartsoft component contains an extra feature that is the port *goalStatusSendServer*.

This port is of type send, a unidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data.

This handler is named *CurrentGoalStatusHandler*, and it contains the code that prints the messages that it receives, such as, the currently considered goal was already reached, the path to the currently considered goal was successfully calculated or not, etc.

This port uses *CommGoalValidation* as communication object.

6.1.1.2.2 UPCTcompMissionPlanner

This Smartsoft component corresponds to the Smartsoft implementation of the Mission Planner component mentioned on chapter 5.

On the image below, Fig. 95, it is shown its structure.

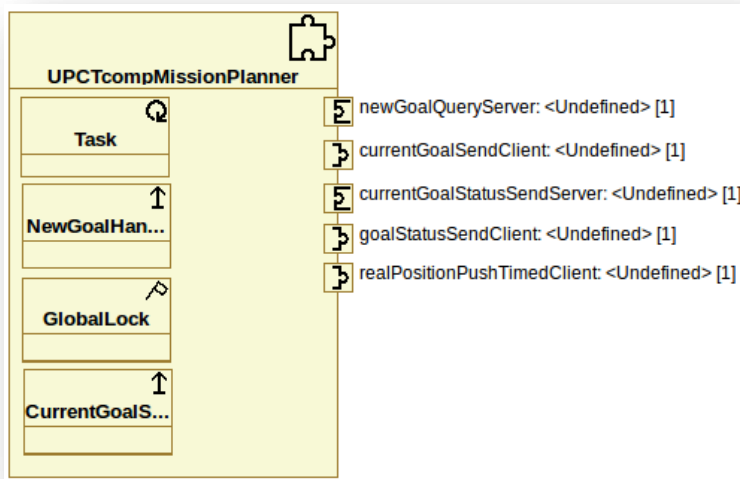


Fig. 95 - Structure of the Smartsoft component *UPCTcompMissionPlanner*

This component procedure consists of various different features:

- Receives new goals from the component ***UPCTcompConsole*** through the port ***newGoalQueryServer*** and processes them.

This communication port is of type query, a bidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data and to send an answer.

This handler is named *NewGoalHandler*, and it contains the code that validates the received goal or not, depending if it is a valid position of the map and if it is not a obstacle, and that sends the validation result through the same port as answer.

If the received goal is valid, it is sent to the *SmartTask Task*, by changing the values of the component global variables that correspond to the received goal, in order to add it to the missions queue.

The communication object used by the port ***newGoalQueryServer*** for the requests is *CommPosition* and the one used for the answers is *CommValidation*.

- On the *SmartTask Task*, this component adds the valid received goals to the missions queue, and also, sends by order of reception, the goals that are stored on the missions queue to the component ***UPCTcompExecutor*** through the port ***currentGoalSendClient***.

A goal is sent only when the robot has reached, or it is not possible to reach, the current goal, which is the one from the missions queue that was previously sent.

To know if a robot has already reached the current goal or not, it checks the current robot position by reading the last value that is at the communication port ***realPositionPushTimedClient***.

The port ***currentGoalSendClient*** is of type send, a unidirectional communication pattern, it is the client part of this communication, and the communication object it uses is *CommPosition*.

The port ***realPositionPushTimedClient*** is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, it is the client part of this communication, and the data that is received through it comes from the ***UPCTcompLocalization*** component and corresponds to the robot actual position.

The communication object used by this port is *CommPosition*.

- Receives the current goal status from the component **UPCTcompExecutor** through the port **currentGoalStatusSendServer**.

This communication port is of type send, a unidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data.

This handler is named *CurrentGoalStatusHandler*, and it contains the code that depending on what it receives, it concludes if it is possible to reach the current goal or not.

This port uses *CommGoalValidation* as communication object.

- Sends the current goal status to the component **UPCTcompConsole** through the port **goalStatusSendClient**.

This communication port is of type send, a unidirectional communication pattern, and it is the client part of this communication.

This port uses *CommGoalValidation* as communication object, and sends messages such as, the currently considered goal was already reached, it is not possible to reach the currently considered goal, the path to the currently considered goal was successfully calculated or not, etc.

6.1.1.2.3 UPCTcompExecutor

This Smartsoft component corresponds to the Smartsoft implementation of the Executor and Obstacles Detector components mentioned on chapter 5.

The reason why on this implementation both components were put together is because by the time this application was implemented, they were not considered as two separate components but one.

On the image below, Fig. 96, it is shown its structure.

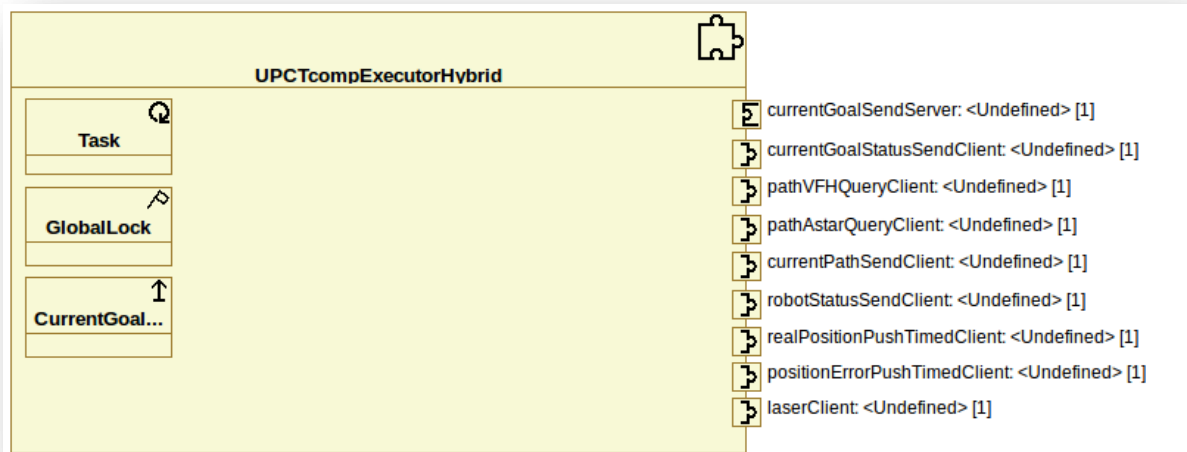


Fig. 96 - Structure of the Smartsoft component *UPCTcompExecutor*

This component procedure consists of various different features:

- Receives new goals from the component **UPCTcompMissionPlanner** through the port **currentGoalSendServer**, and processes them.

This communication port is of type send, a unidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data.

This handler is named *CurrentGoalHandler*, and it contains the code that receives the goal and sends it to the *SmartTask Task*, by changing the values of the component global variables that correspond to the current goal.

The port *currentGoalSendServer* uses *CommPosition* as communication object.

- On the *SmartTask Task*, this component tries to execute the current goal sent by the component **UPCTcompMissionPlanner**.

It starts by checking the current robot position by reading the last value that is at the communication port *realPositionPushTimedClient*, and then by getting the last laser readings by reading the last value that is at the communication port *laserClient*.

With this data, this component is then able to check if new obstacles were detected on the environment around the robot, apart from those that are mentioned on the given map. It does so by using the method *NewObstacles* of the implemented library *MapOperator*, mentioned on section 4.2. This method gives two possible answers, true if a new obstacle was detected or false if no new obstacles were detected.

If a new obstacle is detected, this component sends a message to the component **UPCTcompVFH** through the port *pathVFHQueryClient*, with the laser readings, the robot and current goal positions, in order to request the component to calculate the path to the goal with the implemented VFH algorithm, mentioned on section 4.4.

pathVFHQueryClient is a port of type query, a bidirectional communication pattern, and it is the client part of this communication, which means that the task needs to wait for an answer after sending each message. The answer corresponds to the path calculation result.

The communication object used by the port *pathVFHQueryClient* for the requests is *CommInputVFH* and the one used for the answers is *CommOutputPlanner*.

If the path validation result is equal to 4 or 6, the calculated path is sent to the component **UPCTcompPurePursuit** through the port *currentPathSendClient*, in order to execute it.

If no new obstacles are detected, this component sends a message to the component **UPCTcompAstar** through the port *pathAstarQueryClient*, with the robot and current goal positions, in order to request the component to calculate the path to the goal with the implemented A* algorithm, mentioned on section 4.3.

pathAstarQueryClient is a port of type query, a bidirectional communication pattern, and it is the client part of this communication, which means that the task needs to wait for an answer after sending each message. The answer corresponds to the path calculation result.

The communication object used by the port *pathAstarQueryClient* for the requests is *CommInputAstar* and the one used for the answers is *CommOutputPlanner*.

If the path validation result is equal to 4, the calculated path is sent to the component **UPCTcompPurePursuit** through the port *currentPathSendClient*, in order to execute it. If the path validation result is equal to 5, it is not possible to calculate the path to the current goal because the goal position is an obstacle or because it is unreachable.

Note that, the path validation results of the correspondent case is always sent to the component **UPCTcompMissionPlanner** through the port *currentGoalStatusSendClient*.

Since the method that detects new obstacles on the environment around the robot is called inside the *SmartTask* named *Task*, which corresponds to the cyclic task that includes all the code that is processed at each cycle of this component, it is then verified at each component cycle if new obstacles are detected or not.

If it is not detected new obstacles more than once in a row, a new request is not sent to the component UPCTcompAstar, since it is already being executed the shortest path to the current goal.

On the other hand, every time new obstacles are detected, a new request is sent to the component UPCTcompVFH, in order to calculate a path that avoids the obstacles on the environment and that reaches the goal.

The port realPositionPushTimedClient is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, it is the client part of this communication, and the data that is received through it comes from the UPCTcompLocalization component and corresponds to the robot actual position.

The communication object used by this port is *CommPosition*.

The port laserClient is of type push newest, a unidirectional communication pattern where it is sent updated data from the server when there is new data to send, it is the client part of this communication, and the data that is received through it comes from the SmartPlayerStageSimulator component and corresponds to the last laser readings.

The communication object used by this port is one that comes already implemented with Smartsoft, named *CommMobileLaserScan*.

The port currentPathSendClient is of type send, a unidirectional communication pattern, it is the client part of this communication, and the communication object it uses is *CommPath*.

The port currentGoalStatusSendClient is of type send, a unidirectional communication pattern, it is the client part of this communication, and the communication object it uses is *CommGoalValidation*.

- Also on the *SmartTask Task*, this component verifies when the robot actual position is changed manually.

It does so by checking the last value that is at the communication port positionErrorPushTimedClient, which corresponds to the error factors that are summed to each robot position coordinate that is read from the encoders.

When those received coordinates error factors are different from the previously received ones, a message is sent to the component UPCTcompActuator through the port robotStatusSendClient, in order to stop the motors, so that the application can recalculate the path to the goal position that is being currently considered, by considering the corrected actual robot position.

After so, the *SmartTask Task* is then moved to its initial point, where no path was calculated previously to the currently considered goal, in order to recalculate the path according to the new robot position.

After the path being calculated by the correspondent algorithm and sent to the component UPCTcompPurePursuit, another message is sent through the same port to reactivate the robot motors.

The port positionErrorPushTimedClient is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, it is the client part of this communication, and the data that is received through it comes from the UPCTcompLocalization component and corresponds to the robot position error factors.

The communication object used by this port is *CommPosition*.

The port robotStatusSentClient is of type send, a unidirectional communication pattern, it is the client part of this communication, and the communication object that it uses is *CommRobotStatus*.

6.1.1.2.4 UPCTcompAstar

This Smartsoft component corresponds to the Smartsoft implementation of the Astar component mentioned on chapter 5.

On the image below, Fig. 97, it is shown its structure.

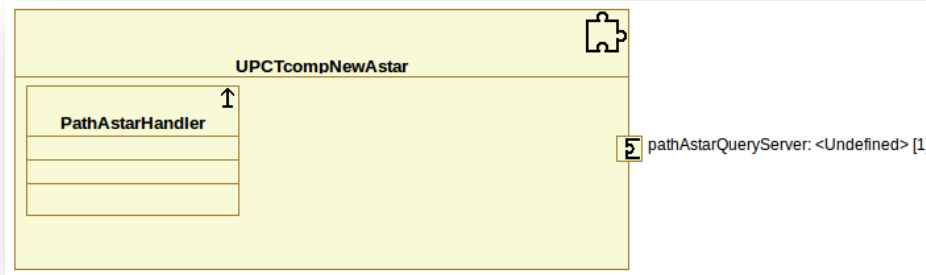


Fig. 97 - Structure of the Smartsoft component *UPCTcompAstar*

The procedure of this component consists on processing the answer to the request messages received from the component **UPCTcompExecutor** through the port **pathAstarQueryServer**.

pathAstarQueryServer is a port of type query, a bidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data and to send an answer.

This handler is named ***PathAstarHandler***, and it contains the code that creates an object of the class Astar, mentioned on section 4.3, and calls all the methods that are needed to calculate the shortest path from the received robot position to the received goal position.

If it is possible to calculate the path from one position to another with the Astar algorithm, this component sends the value 4 and the calculated path as answer to the component **UPCTcompExecutor**, through the port **pathAstarQueryServer**.

If it is not possible to calculate the path from one position to another with the Astar algorithm, which can happen due to the goal position being unreachable or because the goal position corresponds to an obstacle position on the given map, this component sends the value 5 as answer to the component **UPCTcompExecutor**, through the port **pathAstarQueryServer**.

The communication object used by the port **pathAstarQueryServer** for the requests is ***CommInputAstar*** and the one used for the answers is ***CommOutputPlanner***.

6.1.1.2.5 UPCTcompVFH

This Smartsoft component corresponds to the Smartsoft implementation of the VFH component mentioned on chapter 5.

On the image below, Fig. 98, it is shown its structure.

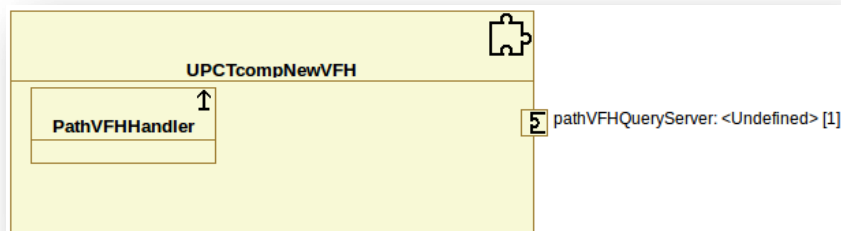


Fig. 98 - Structure of the Smartsoft component *UPCTcompVFH*

The procedure of this component consists on processing the answer to the request messages received from the component **UPCTcompExecutor** through the port **pathVFHQueryServer**.

pathVFHQueryServer is a port of type query, a bidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data and to send an answer.

This handler is named ***PathVFHHandler***, and it contains the code that creates an object of the class VFH, mentioned on section 4.4, and calls all the methods that are needed to calculate the path from the received robot position to the received goal position by avoiding the obstacles on the environment that were detected with the also received laser readings.

If it is possible to calculate the path from one position to another with the VFH algorithm, this component sends the value 4 and the calculated path as answer to the component **UPCTcompExecutor**, through the port **pathVFHQueryServer**.

If it is not possible to calculate the path from one position to another with the VFH algorithm, but a path to another position that is on the direction of the goal, this component sends the value 6 and the calculated path as answer to the component **UPCTcompExecutor**, through the port **pathVFHQueryServer**.

The communication object used by the port **pathVFHQueryServer** for the requests is ***CommInputVFH*** and the one used for the answers is ***CommOutputPlanner***.

6.1.1.2.6 UPCTcompPurePursuit

This Smartsoft component corresponds to the Smartsoft implementation of the Pure Pursuit component mentioned on chapter 5.

On the image below, Fig. 99, it is shown its structure.

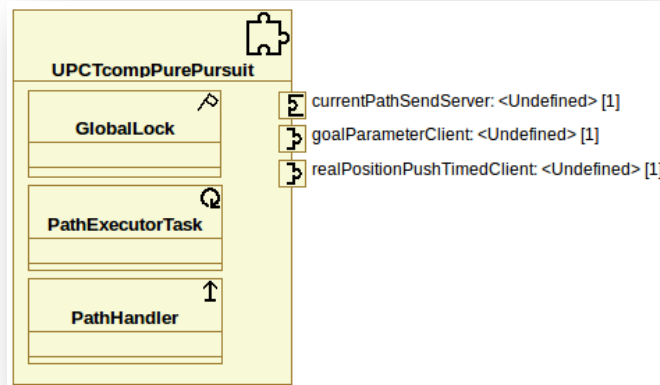


Fig. 99 - Structure of the Smartsoft component ***UPCTcompPurePursuit***

This component procedure consists of various different features:

- Receives a new path to be executed from the component **UPCTcompExecutor** through the port **currentPathSendServer** and processes them.

This communication port is of type send, a unidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data.

This handler is named ***PathHandler***, and it contains the code that receives the new path to be executed and sends it to the ***SmartTask PathExecutorTask***, by changing the values of the component global variables that correspond to the current path to be executed.

The port *currentPathSendServer* uses *CommPath* as communication object.

- On the *SmartTask PathExecutorTask*, this component executes the current path sent by the component ***UPCTcompExecutor***.

It starts by creating an object of the class *PurePursuit*, mentioned on section 4.6, and then, it checks the current robot position by reading the last value that is at the communication port *realPositionPushTimedClient*, and calls all the methods that are needed to calculate the next position on the current path with a distance from the robot position equal to the lookahead distance.

The calculated position is then sent to the component ***UPCTcompToGoal***, through the port *goalParameterClient*, in order to be processed by the robot.

The process of checking the current robot position and calling the methods used to calculate the next position on the current path that is to be processed by the component ***UPCTcompToGoal***, are called periodically at each component cycle, until the calculated next position corresponds to the final goal of the path.

If a new path is received while another is being processed, the object of the class *PurePursuit* that is processing the current one is deleted and a new one is created to process the new path.

The port *goalParameterClient* is of type send, a unidirectional communication pattern, it is the client part of this communication, and the communication object it uses is *CommObjectGoTo*.

The port *realPositionPushTimedClient* is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, it is the client part of this communication, and the data that is received through it comes from the ***UPCTcompLocalization*** component and corresponds to the robot actual position.

The communication object used by this port is *CommPosition*.

6.1.1.2.7 UPCTcompToGoal

This Smartsoft component corresponds to the Smartsoft implementation of the ToGoal component mentioned on chapter 5.

On the image below, Fig. 100, it is shown its structure.

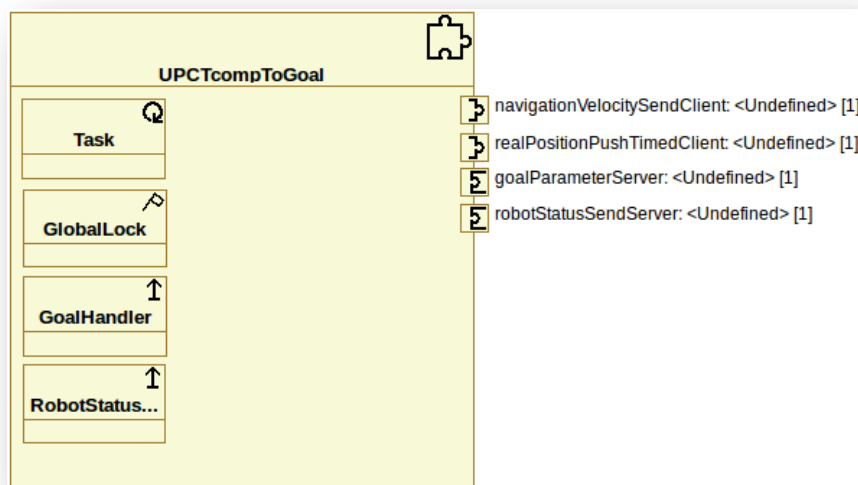


Fig. 100 - Structure of the Smartsoft component *UPCTcompToGoal*

This component procedure consists of various different features:

- Receives new goals to be executed from the component **UPCTcompPurePursuit**, through the port **goalParameterServer**

This communication port is of type send, a unidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data.

This handler is named *GoalHandler*, and it contains the code that receives the new goal to be executed and sends it to the *SmartTask Task*, by changing the values of the component global variables that correspond to the current goal to be executed.

The port **goalParameterServer** uses *CommObjectGoTo* as communication object.

- On the *SmartTask Task*, this component executes the current goal sent by the component **UPCTcompPurePursuit**.

It starts by creating an object of the class ToGoal, mentioned on section 4.5, where it puts as input the robot control constraints and constants, then, it checks the current robot position by reading the last value that is at the communication port **realPositionPushTimedClient**, and calls all the methods of the class ToGoal that are needed to calculate the robot velocities according to the distance from the robot position to the current goal position.

The calculated velocities are then sent to the component **SmartPlayerStageSimulator**, through the port **navigationVelocitySendClient**, in order to be processed by the robot motors.

The process of checking the current robot position and calling the methods used to calculate the robot velocities is executed at each component cycle, until the robot arrives to the current goal position.

If a new goal is received while another is being processed, the same process is kept, with the only change that the current goal that is being considered to calculate the velocities changes.

The port **navigationVelocitySendClient** is of type send, a unidirectional communication pattern, it is the client part of this communication, and the communication object it uses is *CommObjectGoTo*.

The port **realPositionPushTimedClient** is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, it is the client part of this communication, and the data that is received through it comes from the **UPCTcompLocalization** component and corresponds to the robot actual position.

The communication object used by this port is *CommPosition*.

- Also on the *SmartTask Task*, this component receives messages from the components **UPCTcompConsole** and **UPCTcompExecutor**, through the communication port **robotStatusSendServer**, which allow or not the robot to move.

This communication port is of type send, a unidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data.

This handler is named *RobotStatusHandler*, and it contains the code that receives the messages and send the value of the status field of each of them to the *SmartTask Task*, by changing the values of the component global variables that correspond to the current status of the console and executor, respectively.

The port **robotStatusSendServer** uses *CommRobotStatus* as communication object.

On the *SmartTask Task*, the robot only moves when both components allow it at the same time. With this, it means that, in order to move the robot, the last messages

sent by each of both components need to have their status field set to true. If the last of one of them has its status field set to false, the robot is not allowed to move.

6.1.1.2.8 UPCTcompLocalization

This Smartsoft component corresponds to the Smartsoft implementation of the Localization component mentioned on chapter 5.

On the image below, Fig. 101, it is shown its structure.

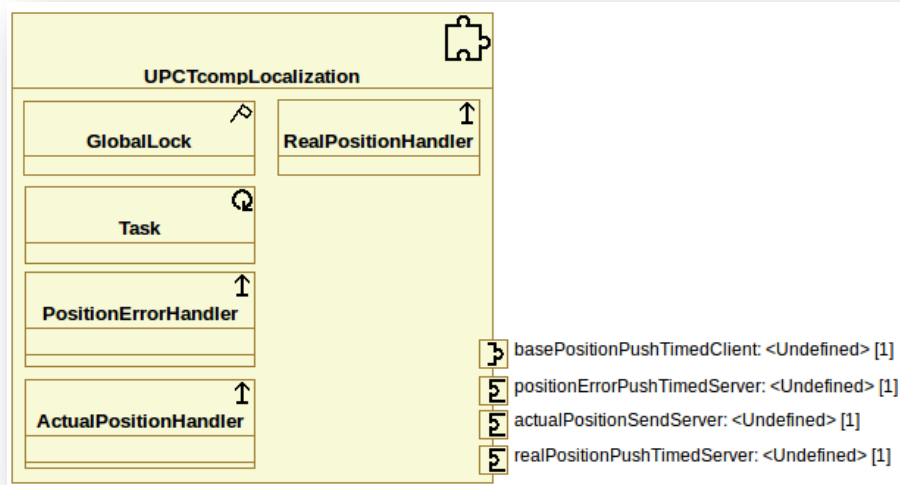


Fig. 101 - Structure of the Smartsoft component UPCTcompLocalization

This component procedure consists of various different features:

- Receives the actual robot position coordinates inserted manually by the user on the component **UPCTcompConsole**, from that component, through the port actualPositionSendServer
This communication port is of type send, a unidirectional communication pattern, and it is the server part of this communication, which means that it needs to have an handler to process the received data.
This handler is named *ActualPositionHandler*, and it contains the code that receives those robot position coordinates and sends them to the *SmartTask Task*, by changing the values of the component global variables that correspond to the actual robot position coordinates.
The port actualPositionSendServer uses *CommPosition* as communication object.
- On the *SmartTask Task*, this component starts by checking the current robot position sent by the encoders, which is done by reading the last value that is at the communication port basePositionPushTimedClient, and then, if a new actual robot position was received from the component **UPCTcompConsole**, it calculates the coordinates error factors by subtracting both positions respective coordinates.
At last, it sends both the actual robot position sent by the encoders and the coordinates error factors to the *Handler RealPositionHandler*, which is done by changing the values of the component global variables that correspond to the robot position sent by the encoders and the coordinates error factors, in order for that

handler to send the real robot position to every component of the application that needs it.

This process is executed at each component cycle.

If it was not received any robot position from the component through the port *actualPositionSendServer*, this task sends to the *Handler RealPositionHandler*, the actual robot position sent by the encoders, and the last calculated coordinates error factors.

Also, every time new coordinates error factors are calculated, they are sent to the *Handler PositionErrorHandler*, by changing the values of the component global variables that correspond to the coordinates actual error factors, in order for that handler to send a message with the new errors that were calculated to every component of the application that needs them.

The port *basePositionPushTimedClient* is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate (50 miliseconds), it is the client part of this communication, and the data that is received through it comes from the *SmartPlayerStageSimulator* component and corresponds to the robot actual position read by the encoders.

The communication object used by this port is *CommPosition*.

- The port *realPositionPushTimedServer* is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, and it is the server part of this communication, which means that it needs to have an handler to process the data that is to be sent.

This handler is named *RealPositionHandler*, and it contains the code that receives the data sent by the *SmartTask Task* through the global variables, the robot position sent by the encoder and the coordinates actual error factors, sums them in order to get the real robot position, and sends that position to all the components that have a port that is client of this handler.

The timed rate at which this process is repeated is 50 miliseconds.

The communication object used by this port is *CommPosition*.

- The port *positionErrorPushTimedServer* is of type push timed, a unidirectional communication pattern where it is sent updated data from the server at a predefined timed rate, and it is the server part of this communication, which means that it needs to have an handler to process the data that is to be sent.

This handler is named *PositionErrorHandler*, and it contains the code that receives the data sent by the *SmartTask Task* through the global variables, the coordinates error factors, and that sends them to all the components that have a port that is client of this handler.

If no new data is received from the *SmartTask Task*, the last calculated coordinates error factors are sent to all the components that have a port that is client of this handler.

The timed rate at which this process is repeated is 50 miliseconds.

The communication object used by this port is *CommPosition*.

After implementing all these components it is then needed to build their binary files one by one, which is done by pressing build on each component project.

6.1.1.3 Deployment

Finally, like mentioned before on section 6.1.1, after implementing all the components it is needed to implement the deployment, which consists on connecting all the components ports.

The first thing to do is to import all the implemented components to the deployment project and then, to connect all the components ports to their respective pairs.

On Fig. 102 it is shown the implemented deployment with all the connections between the implemented components and the **SmartPlayerStageSimulator** component, which is the component that performs the connections between the simulator and the framework.

All the data that needs to be sent to the simulator, such as the linear and angular velocities, needs to be sent to the **SmartPlayerStageSimulator** component, which processes the data to send by putting it with the correct format for the simulator to be able to understand it, and then, sends that formatted data to the simulator.

On the inverse way, all the data that is sent by the simulator, such as the encoders and laser readings, is sent to **SmartPlayerStageSimulator**, which processes the received data by putting it with the correct format for the other Smartsoft components to be able to understand it.

After connecting all the components ports, it is then needed to deploy the application deployment project in order to generate the .sh file that later is to be run as the application executable file.

Since it is needed to run the simulator with this application it is needed to change some part of the code of the generated application deployment .sh file.

On the protected region for the start up code it is needed to activate the simulator and open a map for it, by adding the following code lines that are in bold:

```
### protected region for startup code
# PROTECTED REGION ID(CustomUserStartupCode) ENABLED START
# put your custom startup code here

xterm -e "cd $SMART_ROOT/src/components/SmartPlayerStageSimulator/src/player_stage;
robot-player smart_cave.cfg" &
sleep 4

# PROTECTED REGION END
```

On the protected region for the shut down code it is needed to deactivate the simulator, by adding the following code lines that are in bold:

```
### protected region for shutdown code
# PROTECTED REGION ID(CustomUserShutdownCode) ENABLED START
# put your custom shutdown code here

killall robot-player

# PROTECTED REGION END
```


6.1.1.4 How to run the application

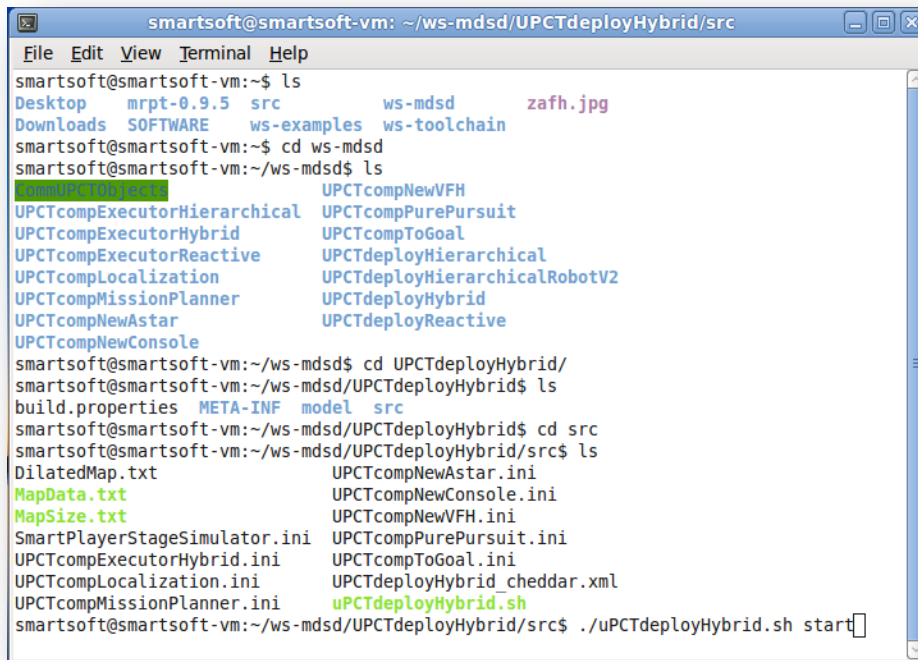
The situation used as test for this application consists on requesting the robot to go to a goal position, on an environment where all the obstacles are unknown by the robot. Since there are some obstacles on the way to the goal position, the robot will need to detect them and calculate paths with both navigation algorithms, in order to avoid the obstacles and arrive as fast as possible to the goal position.

In order to run the implemented application, it is needed to do the following procedure:

- Put the map files inside the *src* folder that is inside the application folder directory. The map files correspond to two *.txt* files:
 - *MapSize.txt*, which must have inside it two values that correspond to the environment map size in terms of rows and columns;
 - *MapData.txt*, which must have inside it the environment map positions values, where each map positions must have a value between 0 and 255. On this situation, the value of all the positions is 0 since the robot knows nothing about the environment.
- Open a terminal window;
- Through the terminal window, go to the *src* folder that is inside the application deployment folder directory, and insert the following code line in order to run the application:

```
smartsoft@smartsoft-vm:~/ws-mdsd/UPCTdeployHybrid/src$ ./uPCTdeployHybrid.sh start
```

The terminal window must now look like the image below, Fig. 103.



```
smartsoft@smartsoft-vm: ~/ws-mdsd/UPCTdeployHybrid/src
File Edit View Terminal Help
smartsoft@smartsoft-vm:~$ ls
Desktop  mrpt-0.9.5  src          ws-mdsd      zafh.jpg
Downloads SOFTWARE  ws-examples ws-toolchain
smartsoft@smartsoft-vm:~$ cd ws-mdsd
smartsoft@smartsoft-vm:~/ws-mdsd$ ls
CommandUPCTdeployHybrid.sh  UPCTcompNewVFH
UPCTcompExecutorHierarchical  UPCTcompPurePursuit
UPCTcompExecutorHybrid        UPCTcompToGoal
UPCTcompExecutorReactive      UPCTdeployHierarchical
UPCTcompLocalization          UPCTdeployHierarchicalRobotV2
UPCTcompMissionPlanner        UPCTdeployHybrid
UPCTcompNewAstar              UPCTdeployReactive
UPCTcompNewConsole
smartsoft@smartsoft-vm:~/ws-mdsd$ cd UPCTdeployHybrid/
smartsoft@smartsoft-vm:~/ws-mdsd/UPCTdeployHybrid$ ls
build.properties META-INF model src
smartsoft@smartsoft-vm:~/ws-mdsd/UPCTdeployHybrid$ cd src
smartsoft@smartsoft-vm:~/ws-mdsd/UPCTdeployHybrid/src$ ls
DilatedMap.txt          UPCTcompNewAstar.ini
MapData.txt             UPCTcompNewConsole.ini
MapSize.txt             UPCTcompNewVFH.ini
SmartPlayerStageSimulator.ini  UPCTcompPurePursuit.ini
UPCTcompExecutorHybrid.ini    UPCTcompToGoal.ini
UPCTcompLocalization.ini     UPCTdeployHybrid_cheddar.xml
UPCTcompMissionPlanner.ini   uPCTdeployHybrid.sh
smartsoft@smartsoft-vm:~/ws-mdsd/UPCTdeployHybrid/src$ ./uPCTdeployHybrid.sh start
```

Fig. 103 - Terminal window with the commands to run a Smartsoft application

- After pressing Enter, all the components are instantiated and the connections between them established;

- In order to work with the application, go to the **UPCTcompConsole** component window, which corresponds to the user interface.

There is the menu with which you can interact with the robot;

- To execute a mission with the robot, choose the option 1 and insert the coordinates of the mission goal in millimeters and degrees.

On the image below, Fig. 104, it is shown an example of how to add a mission, which goal corresponds to the position 8000, 8000 with a final orientation of 120°, and also it is shown that the robot arrived at it successfully.

In order to verify if the robot really arrived at the correct final position, the menu option 2 can be used, which prints on the screen the real robot position.

The red trail corresponds to the trajectory that the robot took from its initial position 0, 0 to the final position 8000, 8000.

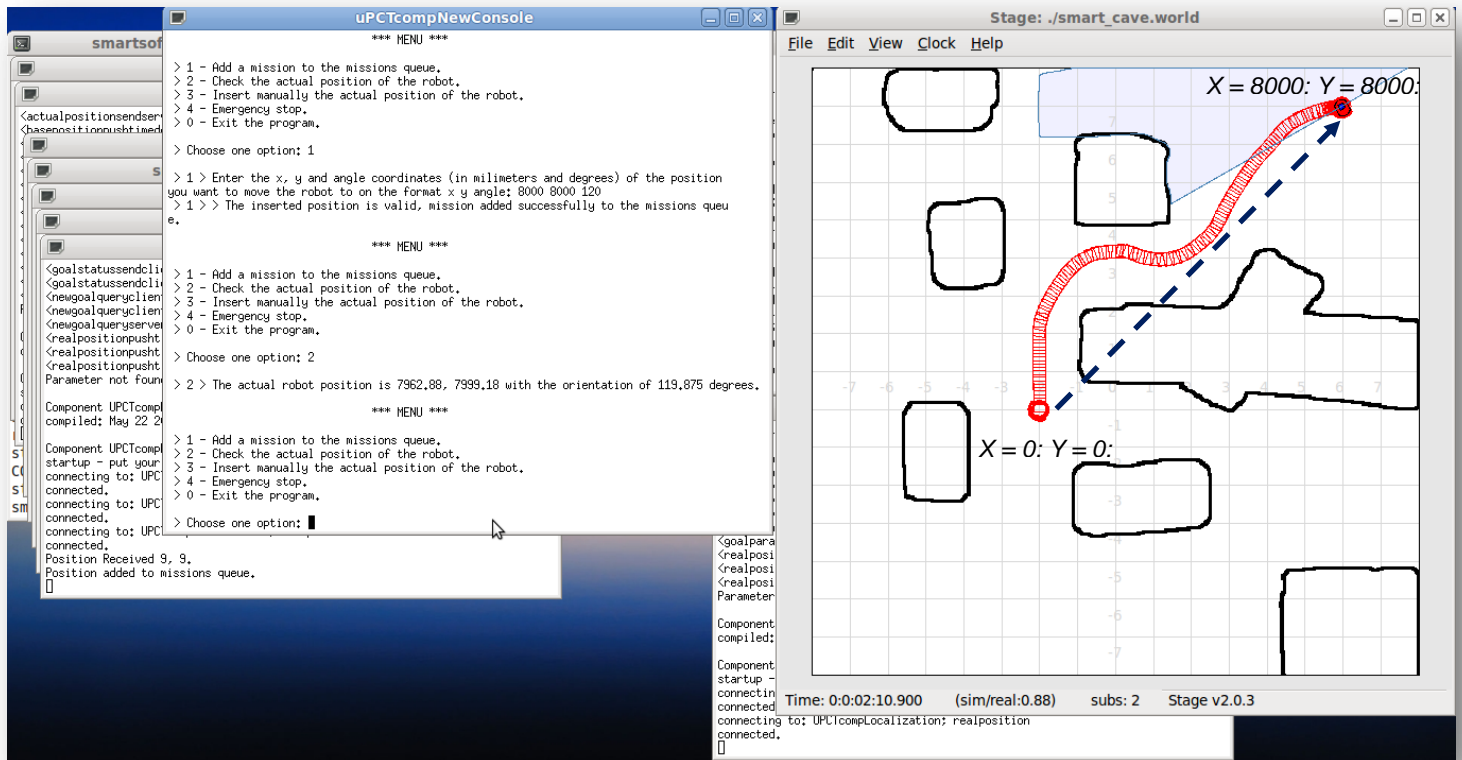


Fig. 104 - Example of how to add a new mission to the robot

- Finally, in order to shut down the application, insert the following code line on the terminal window:

```
smartsoft@smartsoft-vm:~/ws-mdsd/UPCTdeployHybrid/src$ ./uPCTdeployHybrid.sh stop
```


6.1.2 Implementation with the robot Pioneer 3-AT

The implementation with the robot Pioneer 3-AT consists on a simpler application that performs the hierarchical architecture, since it is not possible to work with the robot laser.

The components of this application are **Console**, **Astar**, **PurePursuit** and **ToGoal**, and the application procedure consists on inserting goals by using the **Console**, calculating the shortest path from the robot actual position to that goal position by using the component **Astar** and execute the path by using both components **PurePursuit** and **ToGoal**.

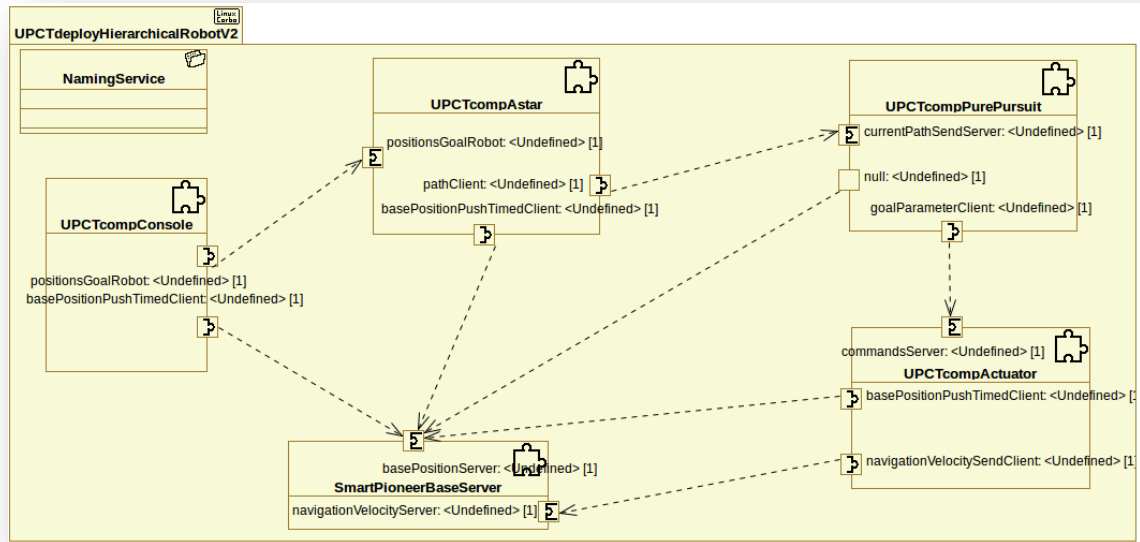


Fig. 105 - Deployment of the implemented hierarchical architecture application with the robot Pioneer 3-AT

As it can be seen on the image above, Fig. 105, it is not used the component **SmartPlayerStageSimulator**, which is the component that perform the connections between the simulator and the framework, but the component **SmartPioneerBaseServer**, which is the component that makes the connections between the real robot and the framework.

Also, it is needed to copy the application from the platform where it was implemented to the real robot and run it there.

The instructions of how to do all these steps are mentioned on the section 6.2.3 of [Ardil12].

6.2 MinFr

This section describes the implementation of two applications with the robotic framework MinFr.

Like mentioned before, on section 4.2.3, from the point of view of the end-user, a MinFr application is divided into three units:

- **Component Library** – Where it is defined the structure of each component used by the application, in terms of ports, regions, states, events and transitions. Each component has its own component library file, which is defined on a *.minfr* file on the folder *MinFrProjects*.
- **Application** – Where it is defined the application structure, in terms of the connections between the ports of the different components used by the application. The application structure file is defined on a *.minfr* file on the folder *MinFrProjects*.
- **Distribution** – Where it is defined the distribution of the all the regions of all the components used by the application into threads. The application distribution file is defined on a *.minfr* file on the folder *MinFrProjects*.

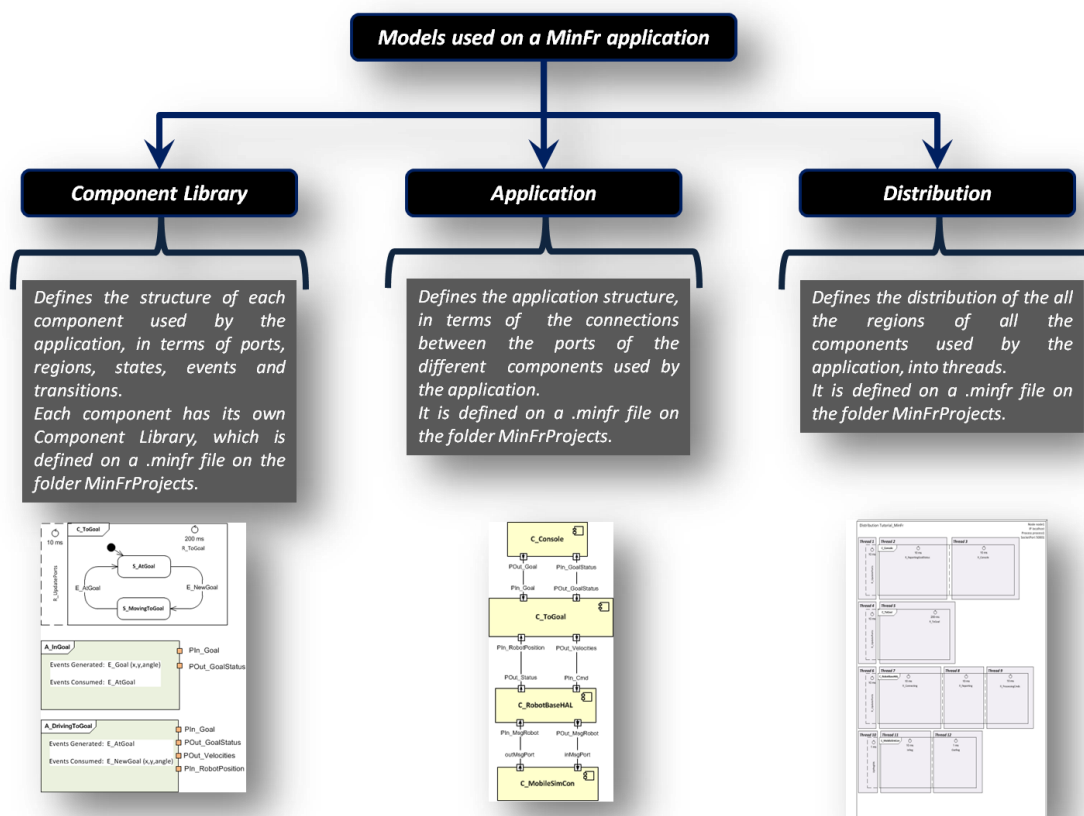


Fig. 106 - The models used on a MinFr application, from the point of view of an end-user

When it comes up to create a MinFr application, there is a sequence that must be taken, which is shown on the picture below, Fig. 107.

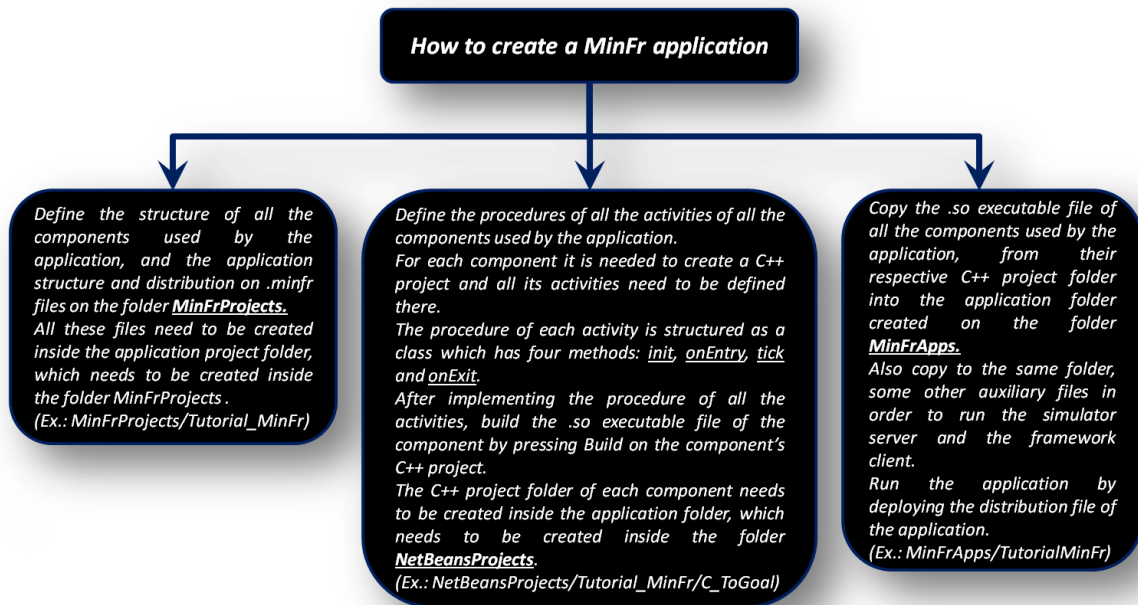


Fig. 107 - Diagram of how to create a MinFr application

The first step of the implementation of a MinFr application consist on three parts:

- 1st – Definition of the structure of all the components that are used on the application in components libraries, where each component has one and only one component library;
- 2nd – Definition of the application structure;
- 3rd – Definition of the application distribution.

As mentioned previously, all these definitions need to be implemented on *.minfr* files by using the program *Eclipse*, and all the resulting files need to be stored inside the folder *MinFrProjects*.

The second step consists on defining the procedures of all the activities of all the components that are used on the application by using the program *NetBeans*.

For each component, it is needed to create a C++ project and all the activities of that component need to be implemented there.

For each activity it is needed to have two files:

- A *.h* (header file) where it is declared the structure of the activity, which is the same for all the activities.
The activity structure consists of a class with four methods: *init*, *onEntry*, *tick* and *onExit*.
- A *.cpp* (source code file) where it is declared the body/procedure of each of those methods.

After defining the procedures of all the activities, of each component used by the application, it is needed to build the C++ project of each component in order to generate its *.so* executable file.

The third and last step consists of two parts:

1st – Copying the .so executable files of all the components from their respective C++ project folder into the application folder created on the folder *MinFrApps*.

2nd – Run the application by deploying the distribution file of the application.

On the following two sections it will be explained how to implement two applications with MinFr by following this approach.

On section 6.2.1, it is described the implementation of a simple robotic application, which is used here as tutorial for how to work and create an application with this robotic framework.

On section 6.2.2, it is described the MinFr implementation of the application described on chapter 5.

6.2.1 Tutorial

On this section it will be described the implementation of a simple robotic application, which is used here as tutorial for how to work and create an application with this robotic framework.

This simple robotic application consists on an application where the user inserts some goal positions, and the robot moves towards them. The structure of this application in terms of components and connections between them is shown below, on Fig. 108.

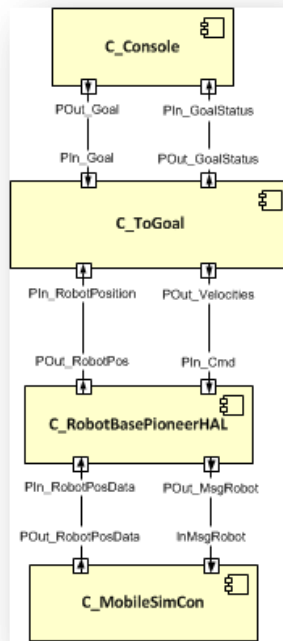


Fig. 108 - Structure of the tutorial application

This application uses four components:

- **C_Console** – This is the user interface component where the user inserts the goals coordinates. While the robot is moving towards the goal, this component reports the current robot position at each instant. If the user inserts a new goal while the robot is moving, the robot will start moving towards the new goal.
- **C_ToGoal** – This component corresponds to the MinFr implementation of the ToGoal component mentioned on chapter 5.

- **C RobotBasePioneerHAL** – This component is the *Hardware Abstraction Layer* (HAL) between the robot and the application components. Whenever any component wants to get information about the robot or send some command to the robot, it needs to access to the ports of this component.
- **C MobileSimCon** – This component is responsible for performing the connection between the simulator and the framework.
 All the data that is to be sent from the framework to the simulator, is sent to this component, which processes it by putting it with the correct format for the simulator to be able to understand it, and then, sends that formatted data to the simulator.
 On the inverse way, all the data that is to be sent by the simulator to the framework, is sent to this component, which processes it by putting it with the correct format for the framework components to be able to understand it.

In order to implement the components libraries, application structure and distribution, which are all defined on *.minfr* files, it is needed to use the program *Eclipse*, open the workspace *PROGRAMA/MinFrProjects*, and then create a new project for this application named *Tutorial_MinFr*.

To the create the application project with the program *Eclipse*, it is needed to do the following procedure:

- Go the tab *File*;
- Select the option *New* and then the sub option *Project*;
- On the window that appears to select the type of project that you want to create, select the type *General* and then the sub type *Project*, and press *Next*;
- Finally, on the same window, insert the project name, on this case *Tutorial_MinFr*, check the box that says *User Default Location*, uncheck the box that says *Add Project to Working Sets*, and press *Finish*.

After completing this procedure, the project is then created successfully on the workspace folder (*PROGRAMA/MinFrProjects*). All the files with the extension *.minfr* that belong to this application need to be created inside this project.

On the following sections it will be explained how to implement this application with MinFr by following the approach mentioned on section 6.2.

An important thing to take in account when implementing the components library, or the application structure or distribution files, is that if you press *Ctrl + Space* while inserting code, it gives you suggestions of what should be the code that can be introduced on that code line according to the current file structure.

6.2.1.1 Components Libraries

It is on the component library, where it is defined the structure of each component that is used by the application in terms of ports, regions, states, events and transitions.

Each component has its own component library, which is defined on a *.minfr* file on the folder *MinFrProjects*.

To create each component library, it is needed to do the following procedure on the program *Eclipse* by using the workspace *PROGRAMA/MinFrProjects*:

- Go the tab *File*;
- Select the option *New* and then select the sub option *File*;
- On the window that appears to select the project where you want to create the file, select the project *Tutorial_MinFr*, insert on the field *Filename* the name of the component library with extension *.minfr*, for example *C_ConsoleLib.minfr*, and press *Finish*.

After completing this procedure, the component library is then created successfully on the application folder (*PROGRAMA/MinFrProjects/Tutorial_MinFr*).

On this section it will be explained how to implement the component libraries of the four components that are used on this application.

For the first two components, *C_Console* and *C_ToGoal*, it will be explained in detail the procedure of implementing their respective component libraries by following a eight steps procedure.

For the other two components, *C_RobotBasePioneerHAL* and *C_MobileSimCon*, it will be only shown their structure, since their implementation follows the same basic concepts explained on the detail for the other two.

6.2.1.1.1 C_Console

In order to implement the component structure, the first to do is to create the component library file, which is named *C_Console_Lib.minfr*, inside the project *Tutorial_MinFr*, by following the steps mentioned previously on this section.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 109.

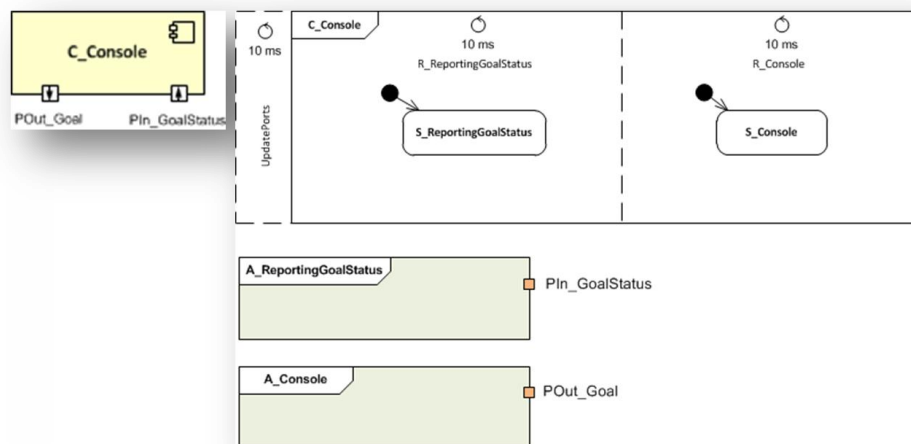


Fig. 109 - Structure of the component *C_Console*

The process of defining a component structure on its correspondent component library is divided in eight steps:

- 1st step – Define the component library name;
The component library name is *C_Console_Lib*, and so, after the first step, the component library file looks like this:

```
Library C_Console_Lib {
}
```

- 2nd step – Define the component name;
As it can be seen on Fig. 109, the component name is *C_Console*, and so, after the second step, the component library file looks like this:

```
Library C_Console_Lib {
  Component C_Console {
  }
}
```

- 3rd step – Define the component ports;
As it can be seen on Fig. 109, this component has two ports, one is of type *OutPort*, *POut_Goal*, and the other is of type *InPort*, *PIn_GoalStatus*, and so, after the third step, the component library file looks like this:

```
Library C_Console_Lib {
  Component C_Console {
    OutPort POut_Goal;
    InPort PIn_GoalStatus;
  }
}
```

- 4th step – Define the component update region, and correspondent period of repetition.

This update region is responsible for updating the data that the *OutPorts* have as their output and send to the ports that are connected to them. The period consists on the time that there is between two updates.

As it can be seen on Fig. 109, the name of the update region is *R_UpdatePorts*, and the repetition period is 10000 microseconds (µs), and so, after the fourth step, the component library file looks like this:

```
Library C_Console_Lib {
  Component C_Console {
    OutPort POut_Goal;
    InPort PIn_GoalStatus;
    UpdateRegion UpdatePorts Period 10000;
  }
}
```

- 5th step – Define the component regions, and correspondent period of repetition.

Each region corresponds to a concurrent part of the same component, and their periods consist on the time that there is between two executions of the same region procedure.

As it can be seen on Fig. 109, this component has two regions, *R_Console* and *R_ReportingGoalStatus*, and the repetition time of both of them is 10000 microseconds (µs), and so, after the fifth step, the component library file looks like this:

```
Library C_Console_Lib {
```

```

Component C_Console {
    OutPort POut_Goal;
    InPort PIn_GoalStatus;
    UpdateRegion UpdatePorts Period 10000;
    Region R_Console {
        Period 10000;

    }
    Region R_ReportingGoalStatus {
        Period 10000;

    }
}

```

- 6th step – Define the states of each region.

As it can be seen on Fig. 109, both regions have only one state, *S_Console* for the region *R_Console*, and, *S_ReportingGoalStatus* for the region *R_ReportingGoalStatus*, and so, after the sixth step, the component library file looks like this:

```

Library C_Console_Lib {
    Component C_Console {
        OutPort POut_Goal;
        InPort PIn_GoalStatus;
        UpdateRegion UpdatePorts Period 10000;
        Region R_Console {
            Period 10000;
            State S_Console {

            }
        }
        Region R_ReportingGoalStatus {
            Period 10000;
            State S_ReportingGoalStatus {

            }
        }
    }
}

```

- 7th step – Define the activity of each state.

Each state can only have one activity, and here it is defined its name, path and port accesses.

The path corresponds to the place where is defined the state activity procedure, which is done in C++, and the port accesses correspond to the ports that each state accedes from the ones that the component has. If needed the port name can be changed inside a state for its procedure.

As it can be seen on Fig. 109, the state *S_Console* activity is named *A_Console*, it only accedes to the port *POut_Goal*, and for its procedure it uses the same port name.

The state *S_ReportingGoalStatus* activity is named *A_ReportingGoalStatus*, it only accedes to the port *PIn_GoalStatus*, and for its procedure it uses the same port name.

The path of the activity *A_Console* is *libC_Console/A_Console*, and the path of the activity *A_ReportingGoalStatus* is *libC_Console/A_ReportingGoalStatus*.

Then, after the seventh step, the component library file looks like this:

```

Library C_Console_Lib {
    Component C_Console {

```



```

OutPort POut_Goal;
InPort PIn_GoalStatus;
UpdateRegion UpdatePorts Period 10000;
Region R_Console {
    Period 10000;
    State S_Console {
        Activity A_Console {
            Path "libC_Console/A_Console";
            PortAccess POut_Goal Port POut_Goal;
        }
    }
}
Region R_ReportingGoalStatus {
    Period 10000;
    State S_ReportingGoalStatus {
        Activity A_ReportingGoalStatus {
            Path "libC_Console/A_ReportingGoalStatus";
            PortAccess PIn_GoalStatus Port PIn_GoalStatus;
        }
    }
}
}
}

```

- 8th step – Define the events that each state consumes and generates, and the transitions that happen between states of the same region.

To make a transition from one state to another is needed an event. The event that is active needs to generate it, and the one that is to change to, needs to consume it.

As it can be seen on Fig. 109, there are no events neither transitions, since each region only have one state each, which means that it is not needed to make any changes to the component library file.

So, after the eight and last step, the component library file looks like this:

```

Library C_Console_Lib {
    Component C_Console {
        OutPort POut_Goal;
        InPort PIn_GoalStatus;
        UpdateRegion UpdatePorts Period 10000;
        Region R_Console {
            Period 10000;
            State S_Console {
                Activity A_Console {
                    Path "libC_Console/A_Console";
                    PortAccess POut_Goal Port POut_Goal;
                }
            }
        }
        Region R_ReportingGoalStatus {
            Period 10000;
            State S_ReportingGoalStatus {
                Activity A_ReportingGoalStatus {
                    Path "libC_Console/A_ReportingGoalStatus";
                    PortAccess PIn_GoalStatus Port PIn_GoalStatus;
                }
            }
        }
    }
}
}

```

6.2.1.1.2 C_ToGoal

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 110.

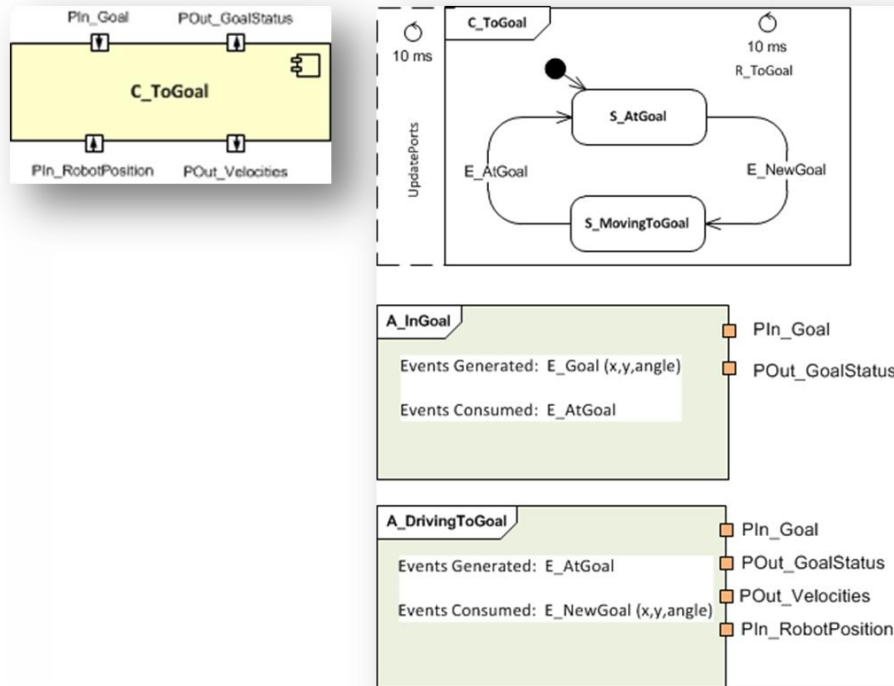


Fig. 110 - Structure of the component *C_ToGoal*

In order to implement the component structure, the first to do is to create the component library file, which is named *C_ToGoal_Lib.minfr*, inside the project *Tutorial_MinFr*, by following the steps mentioned previously on this section.

The process of defining a component structure on its correspondent component library is divided in eighth steps:

- 1st step – Define the component library name;
The component library name is *C_ToGoal_Lib*, and so, after the first step, the component library file looks like this:

```
Library C_ToGoal_Lib {
}
```

- 2nd step – Define the component name;
As it can be seen on Fig. 110, the component name is *C_ToGoal*, and so, after the second step, the component library file looks like this:

```
Library C_ToGoal_Lib {
  Component C_ToGoal {
  }
}
```

- 3rd step – Define the component ports;
As it can be seen on Fig. 110, this component has four ports, two of type *OutPort*, *POut_GoalStatus* and *POut_Velocities*, and two other of type *InPort*, *Pin_Goal* and *Pin_RobotPosition*, and so, after the third step, the component library file looks like this:

```
Library C_ToGoal_Lib {
  Component C_ToGoal {
```

```

        OutPort POut_GoalStatus;
        InPort PIn_Goal;
        OutPort POut_Velocities;
        InPort PIn_RobotPosition;
    }
}

```

- 4th step – Define the component update region, and correspondent period of repetition.

This update region is responsible for updating the data that the *OutPorts* have as their output and send to the ports that are connected to them. The period consists on the time that there is between two updates.

As it can be seen on Fig. 110, the name of the update region is *R_UpdatePorts*, and the repetition period is 10000 us, and so, after the fourth step, the component library file looks like this:

```

Library C_ToGoal_Lib {
    Component C_ToGoal {
        OutPort POut_GoalStatus;
        InPort PIn_Goal;
        OutPort POut_Velocities;
        InPort PIn_RobotPosition;
        UpdateRegion UpdatePorts Period 10000;
    }
}

```

- 5th step – Define the component regions, and correspondent period of repetition.

Each region corresponds to a concurrent part of the same component, and their periods consist on the time that there is between two executions of the same region procedure.

As it can be seen on Fig. 110, this component has only one region, *R_ToGoal*, and its repetition is 10000 us, and so, after the fifth step, the component library file looks like this:

```

Library C_ToGoal_Lib {
    Component C_ToGoal {
        OutPort POut_GoalStatus;
        InPort PIn_Goal;
        OutPort POut_Velocities;
        InPort PIn_RobotPosition;
        UpdateRegion UpdatePorts Period 10000;
        Region R_ToGoal {
            Period 10000;
        }
    }
}

```

- 6th step – Define the states of each region.

As it can be seen on Fig. 110, the only region of this component has two states, *S_AtGoal* and *S_MovingToGoal*.

When declaring the region states, the initial state of the region state machine must be the one declared first and so, after the sixth step, the component library file looks like this:

```

Library C_ToGoal_Lib {
    Component C_ToGoal {
        OutPort POut_GoalStatus;
        InPort PIn_Goal;
        OutPort POut_Velocities;

```

```

InPort PIn_RobotPosition;
UpdateRegion UpdatePorts Period 10000;
Region R_ToGoal {
    Period 10000;
    State S_AtGoal{

    }
    State S_MovingToGoal{

    }
}
}

```

- 7th step – Define the activity of each state.

Each state can only have one activity, and here it is defined its name, path and port accesses.

The path corresponds to the place where is defined the state activity procedure, which is done in C++, and the port accesses correspond to the ports that each state uses from the ones that the component has. If needed the port name can be changed inside a state for its procedure.

As it can be seen on Fig. 110, the state *S_AtGoal* activity is named *A_AtGoal*, it accesses to the ports *PIn_Goal* and *POut_GoalStatus*, and for its procedure it uses the same ports names.

The state *S_MovingToGoal* activity is named *A_MovingToGoal*, it accesses to the ports *PIn_Goal*, *POut_GoalStatus*, *POut_Velocities* and *PIn_RobotPosition*, and for its procedure it uses the same ports names.

The path of the activity *A_AtGoal* is *libC_ToGoal/A_ToGoal*, and the path of the activity *A_MovingToGoal* is *libC_ToGoal/A_MovingToGoal*.

Then, after the seventh step, the component library file looks like this:

```

Library C_ToGoal_Lib {
    Component C_ToGoal {
        OutPort POut_GoalStatus;
        InPort PIn_Goal;
        OutPort POut_Velocities;
        InPort PIn_RobotPosition;
        UpdateRegion UpdatePorts Period 10000;
        Region R_ToGoal {
            Period 10000;
            State S_AtGoal {
                Activity A_AtGoal {
                    Path "libC_ToGoal/A_AtGoal";
                    PortAccess PIn_Goal Port PIn_Goal;
                    PortAccess POut_GoalStatus Port POut_GoalStatus;
                }
            }
            State S_MovingToGoal {
                Activity A_MovingToGoal {
                    Path "libC_ToGoal/A_MovingToGoal";
                    PortAccess PIn_Goal Port PIn_Goal;
                    PortAccess POut_GoalStatus Port POut_GoalStatus;
                    PortAccess POut_Velocities Port POut_Velocities;
                    PortAccess PIn_RobotPosition Port
PIn_RobotPosition;
                }
            }
        }
    }
}

```

- 8th step – Define the events that each state consumes and generates, and the transitions that happen between states of the same region.

To make a transition from one state to another it is needed a condition, which can be an instance of an event or an arrival of data at a port. About the transitions with events, the state that is active needs to generate an instance of the event, and the state that is to change to, needs to consume it.

As it can be seen on Fig. 110, there are two transitions.

One of them is from the state *S_AtGoal* to the state *S_MovingToGoal*, and it needs an instance of the event *E_NewGoal*, which is an event that is generated by the state *S_AtGoal* and consumed by the state *S_MovingToGoal*.

The other one is from the state *S_MovingToGoal* to the state *S_AtGoal*, and it needs an instance of the event *E_AtGoal*, which is an event that is generated by the state *S_MovingToGoal* and consumed by the state *S_AtGoal*.

These two transitions are called *Transition1* and *Transition2*, respectively.

So, after the eight and last step, the component library file looks like this:

```
Library C_ToGoal_Lib {
  Component C_ToGoal {
    OutPort POut_GoalStatus;
    InPort PIn_Goal;
    OutPort POut_Velocities;
    InPort PIn_RobotPosition;
    UpdateRegion UpdatePorts Period 10000;
    Region R_ToGoal {
      Period 10000;
      State S_AtGoal {
        Activity A_AtGoal {
          Path "libC_ToGoal/A_AtGoal";
          PortAccess PIn_Goal Port PIn_Goal;
          PortAccess POut_GoalStatus Port POut_GoalStatus;
          GeneratesEvent E_NewGoal Event E_NewGoal;
          ConsumeEvent E_AtGoal Event
S_MovingToGoal.A_MovingToGoal.E_AtGoal;
        }
      }
      State S_MovingToGoal {
        Activity A_MovingToGoal {
          Path "libC_ToGoal/A_MovingToGoal";
          PortAccess PIn_Goal Port PIn_Goal;
          PortAccess POut_GoalStatus Port POut_GoalStatus;
          PortAccess POut_Velocities Port POut_Velocities;
          PortAccess PIn_RobotPosition Port
PIn_RobotPosition;
          GeneratesEvent E_AtGoal Event E_AtGoal;
          ConsumeEvent E_NewGoal Event
S_AtGoal.A_AtGoal.E_NewGoal;
        }
      }
      Transition Transition1 {
        from S_AtGoal to S_MovingToGoal;
        ConditionsList {
          ConditionActivityEvent Cond1 Event
S_AtGoal.A_AtGoal.E_NewGoal;
        }
      }
      Transition Transition2 {
        from S_MovingToGoal to S_AtGoal;
        ConditionsList {
          ConditionActivityEvent Cond1 Event
S_MovingToGoal.A_MovingToGoal.E_AtGoal;
        }
      }
    }
  }
}
```

6.2.1.1.3 C_RobotBasePioneerHAL

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on Fig. 111.

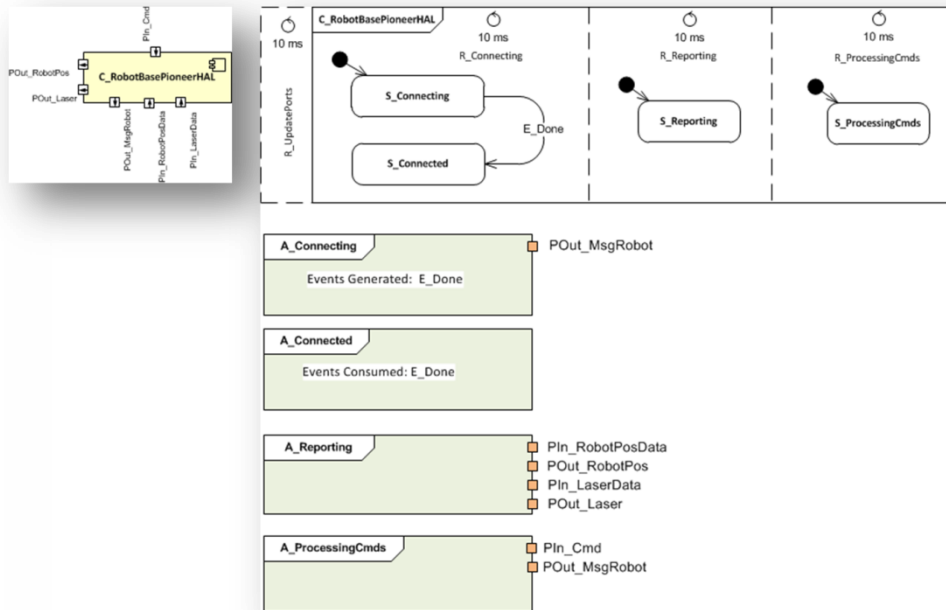


Fig. 111 - Structure of the component C_RobotBasePioneerHAL

In order to implement the component structure, the first to do is to create the component library file, which is named *C_RobotBasePioneerHAL_Lib.minfr*, inside the project *Tutorial_MinFr*, by following the steps mentioned previously on this section.

The implementation of this component library is not explained here, since it follows the same basic concepts explained in detail for the two other components, *C_Console* and *C_ToGoal*.

6.2.1.1.4 C_MobileSimCon

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on Fig. 112.

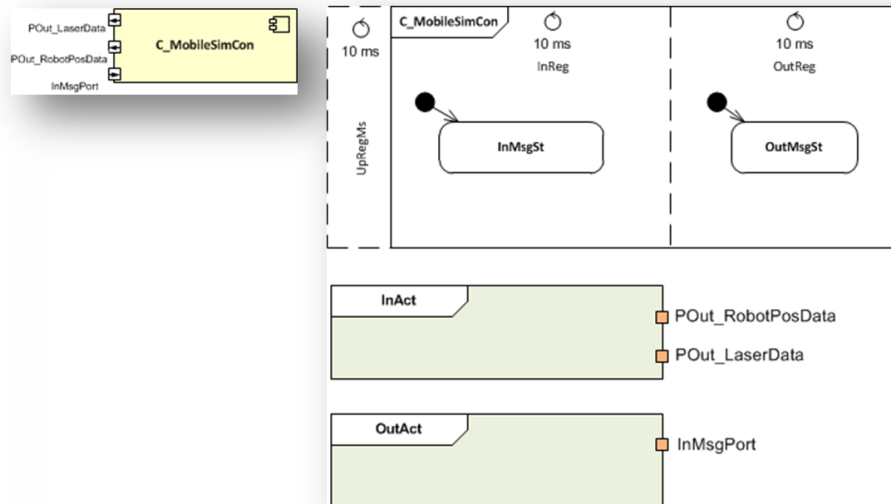


Fig. 112 - Structure of the component *C_MobileSimCon*

In order to implement the component structure, the first to do is to create the component library file, which is named *C_MobileSimCon_Lib.minfr*, inside the project *Tutorial_MinFr*, by following the steps mentioned previously on this section.

The implementation of this component library is not explained here, since it follows the same basic concepts explained in detail for the two other components, *C_Console* and *C_ToGoal*.

6.2.1.2 Application Structure

It is on the application structure where it is defined the structure of the application, in terms of the connections between the ports of the components.

The application structure is defined on a *.minfr* file on the folder *MinFrProjects*.

To create a application structure file, it is needed to do the following procedure on the program *Eclipse* by using the workspace *PROGRAMA/MinFrProjects*:

- Go the tab *File*;
- Select the option *New* and then select the sub option *File*;
- On the window that appears to select the project where you want to create the file, select the project *Tutorial_MinFr*, insert on the field *Filename* the name of the application structure with extension *.minfr*, for example *Tutorial_App.minfr*, and press *Finish*.

After completing this procedure, the application structure file is then created successfully on the application folder (*PROGRAMA/MinFrProjects/Tutorial_MinFr*).

On this section it will be explained how to implement the application structure file of this simple application.

The structure of the application, in terms of connections between the ports of the components, is shown on the image below, Fig. 113.

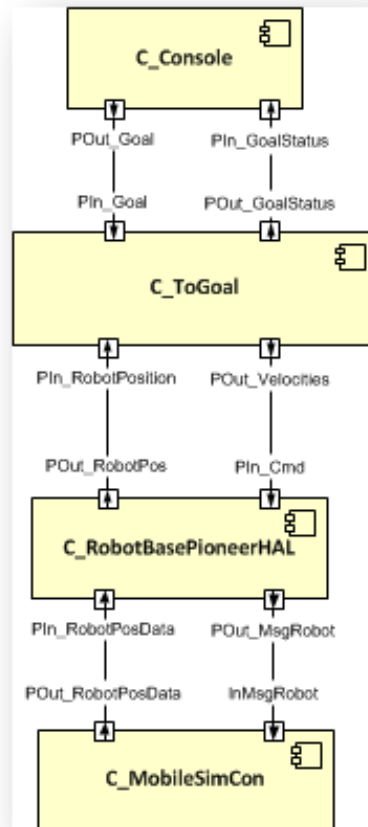


Fig. 113 - Structure of the tutorial application

The process of defining an application structure on its correspondent application structure file is divided in four steps:

- 1st step – Import the component library of each component that form the application;

As it can be seen on Fig. 113, this application is formed by four components, *C_Console*, *C_ToGoal*, *C_RobotBasePioneerHAL* and *C_MobileSimCon*, which means that the component libraries of each of them need to be imported, and so, after the first step, the application structure file looks like this:

```
Import "C_Console_Lib.minfr" ;
Import "C_ToGoal_Lib.minfr" ;
Import "C_RobotBasePioneerHAL_Lib.minfr";
Import "C_MobileSimCon_Lib.minfr";
```

- 2nd step – Define the application architecture name;
The application architecture name is *Tutorial_App*, and so, after the second step, the application structure file looks like this:

```
Import "C_Console_Lib.minfr" ;
Import "C_ToGoal_Lib.minfr" ;
Import "C_RobotBasePioneerHAL_Lib.minfr";
Import "C_MobileSimCon_Lib.minfr";

Application Tutorial_App{
}
```


- 3rd step – Define the components names that will be considered by the application when referring to the components implemented on the component libraries that were imported;

As it can be seen on Fig. 113, this application is formed by four components, *C_Console*, *C_ToGoal*, *C_RobotBasePioneerHAL* and *C_MobileSimCon*, and the components names that will be considered by the application are *C_CON*, *C_TG*, *C_Robot* and *C_MobSim*, respectively, and so, after the third step, the application structure file looks like this:

```
Import "C_Console_Lib.minfr" ;
Import "C_ToGoal_Lib.minfr" ;
Import "C_RobotBasePioneerHAL_Lib.minfr";
Import "C_MobileSimCon_Lib.minfr";

Application Tutorial_App{
    C_Console_Lib.C_Console C_CON;
    C_ToGoal_Lib.C_ToGoal C_TG;
    C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL C_Robot;
    C_MobileSimCon_Lib.C_MobileSimCon C_MobSim;
}
```

- 4th step – Define the connections between the ports of the components.

The connections between the ports of the components that need to be defined are the ones shown on Fig. 113, and so, after the fourth and last step, the application structure file looks like this:

```
Import "C_Console_Lib.minfr" ;
Import "C_ToGoal_Lib.minfr" ;
Import "C_RobotBasePioneerHAL_Lib.minfr";
Import "C_MobileSimCon_Lib.minfr";

Application Tutorial_App{
    C_Console_Lib.C_Console C_CON;
    C_ToGoal_Lib.C_ToGoal C_TG;
    C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL C_Robot;
    C_MobileSimCon_Lib.C_MobileSimCon C_MobSim;

    Connect C_CON-C_Console_Lib.C_Console.POut_Goal and C_TG-
C_ToGoal_Lib.C_ToGoal.PIn_Goal;
    Connect C_TG-C_ToGoal_Lib.C_ToGoal.POut_GoalStatus and C_CON-
C_Console_Lib.C_Console.PIn_GoalStatus;
    Connect C_Robot-C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.POut_MsgRobot and
C_MobSim-C_MobileSimCon_Lib.C_MobileSimCon.InMsgPort;
    Connect C_Robot-C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.POut_RobotPos and
C_TG-C_ToGoal_Lib.C_ToGoal.PIn_RobotPosition;
    Connect C_Robot-C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.PIn_RobotPosData
and C_MobSim-C_MobileSimCon_Lib.C_MobileSimCon.POut_RobotPosData;
    Connect C_Robot-C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.PIn_LaserData and
C_MobSim-C_MobileSimCon_Lib.C_MobileSimCon.POut_LaserData;
    Connect C_TG-C_ToGoal_Lib.C_ToGoal.POut_Velocities and C_Robot-
C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.PIn_Cmd;
}
```

6.2.1.3 Application Distribution

It is on the application distribution where it is defined the distribution of the all the regions of all the components used by the application into threads.

The application distribution is defined on a *.minfr* file on the folder *MinFrProjects*.

To create a application distribution file, it is needed to do the following procedure on the program *Eclipse* by using the workspace *PROGRAMA/MinFrProjects*:

- Go the tab *File*;
- Select the option *New* and then select the sub option *File*;
- On the window that appears to select the project where you want to create the file, select the project *Tutorial_MinFr*, insert on the field *Filename* the name of the application distribution with extension *.minfr*, for this case *Tutorial_Dist.minfr*, and press *Finish*.

After completing this procedure, the application distribution file is then created successfully on the application folder (*PROGRAMA/MinFrProjects/Tutorial_MinFr*).

On this section it will be explained how to implement the application distribution file of this simple application.

The structure of the application, in terms of distribution of all the regions of all the components used by the application in threads, is shown on Fig. 114.

The process of defining an application distribution on its correspondent application distribution file is divided in seven steps:

- 1st step – Import the application structure;
The application structure of this application is the one implemented on the previous section, which is named *Tutorial_App.minfr*, and so, after the first step, the application distribution file looks like this:

```
Import "Tutorial_App.minfr";
```

- 2nd step – Define the application distribution name;
The application distribution name is *D_Tutorial*, and so, after the second step, the application distribution file looks like this:

```
Import "Tutorial_App.minfr";
```

```
Distribution D_Tutorial {  
  
}
```

- 3rd step – Define the nodes that are going to be used by the distribution.

Here, nodes correspond to CPUs, since concurrent parts of an application can be processed at different CPUs.

As it can be seen on Fig. 114, this application only uses one node named *Node1*, and so, after the third step, the application distribution file looks like this:

```
Import "Tutorial_App.minfr";
```

```
Distribution D_Tutorial {  
    Node Node1{  
  
    }  
}
```

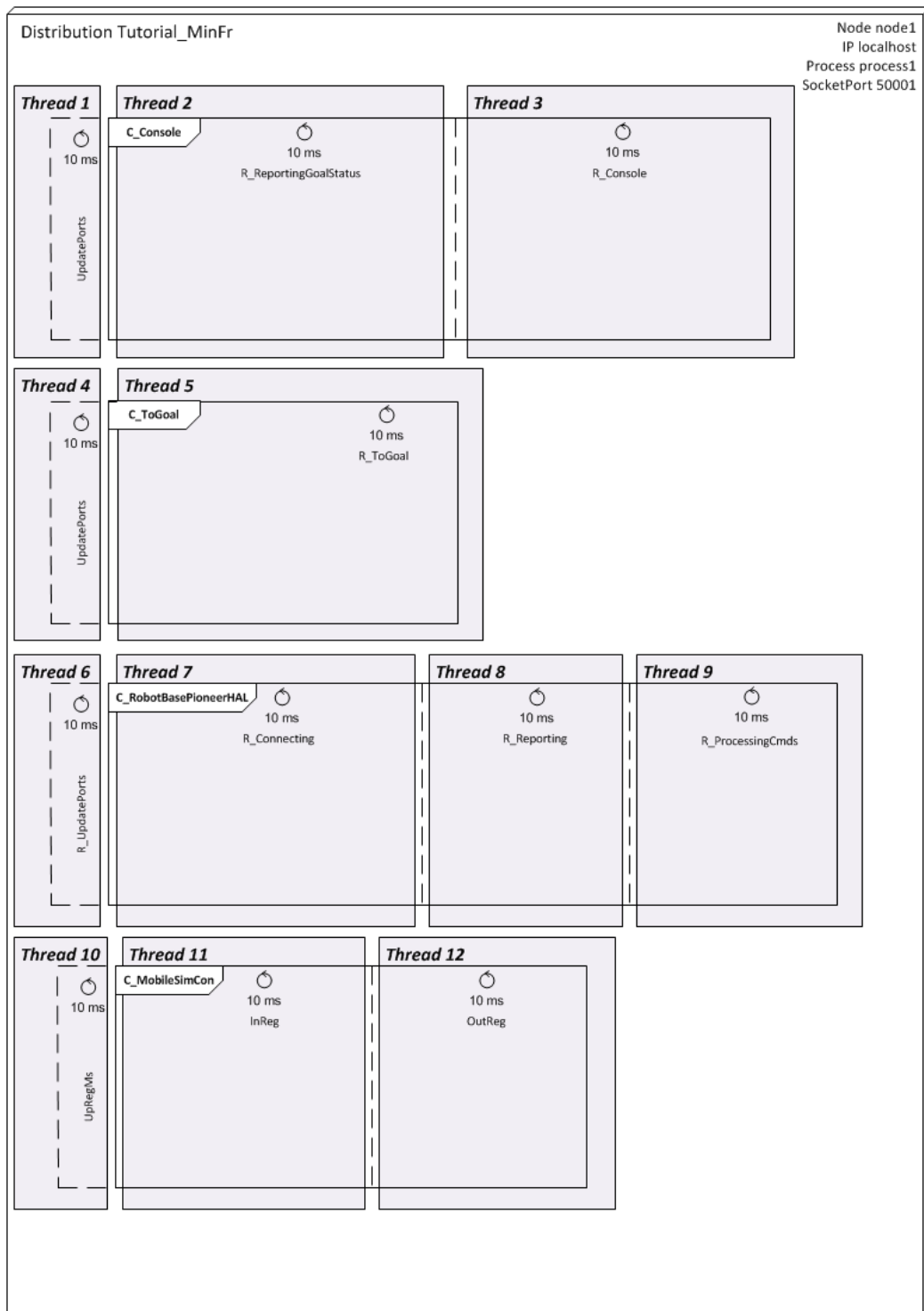


Fig. 114 - Distribution of the tutorial application

- 4th step – Define the IP address of each node used by the distribution.

As it can be seen on Fig. 114, the IP address of the only node that this application uses is *localhost*, and so, after the fourth step, the application distribution file looks like this:

```
Import "Tutorial_App.minfr";
```

```
Distribution D_Tutorial {
    Node Node1{
        IP localhost ;
    }
}
```

- 5th step – Define the processes that are processed by each node.

As it can be seen on Fig. 114, the *Node1* only processes one process, which is named *Process1*, and so, after the fifth step, the application distribution file looks like this:

```
Import "Tutorial_App.minfr";

Distribution D_Tutorial {
    Node Node1{
        IP localhost ;
        Process Process1 {

        }
    }
}
```

- 6th step – Define the socket port of each process as well as, the components that are there instantiated and the threads that are needed to process the regions of those components.

As it can be seen on Fig. 114, the socket port of the only process is *50001*, and on that process are instantiated all the processes and are needed twelve threads in order to process the regions of all the components, and so, after the sixth step, the application distribution file looks like this:

```
Import "Tutorial_App.minfr";

Distribution D_Tutorial {
    Node Node1{
        IP localhost ;
        Process Process1 {
            SocketPort 50001 ;
            ComponentInstances Tutorial_App.C_CON, Tutorial_App.C_TG,
Tutorial_App.C_Robot, Tutorial_App.C_MobSim;
            Thread thread1{

            }

            Thread thread2{

            }

            Thread thread3{

            }

            Thread thread4{

            }

            Thread thread5{

            }

            Thread thread6{

            }

            Thread thread7{

            }

            Thread thread8{

            }

            Thread thread9{

            }

            Thread thread10{

            }
        }
    }
}
```

```

Thread thread1{

}

Thread thread12{

}

}
}

```

- 7th step – Define the regions that are processed at each thread.

As it can be seen on Fig. 114, each region of each component is assigned to a different thread.

This region distribution approach was taken in order for the application to be as efficient as possible, but, it doesn't mean it is the best one since this is using more CPU resources.

It is possible to have more than one region, and they can even be from different components. The only rule that needs to be respected when there is more than one region per thread, is that those regions that belong to the same thread can't have any blocking function, since each thread goes through each of its regions on a cyclic way, and if it would have a blocking one it could not process the others.

Then, according to Fig. 114, after the seventh and last step, the application distribution file looks like this:

```

Import "Tutorial_App.minfr";

Distribution D_Tutorial {
    Node Node1 {
        IP localhost ;
        Process Process1 {
            SocketPort 50001 ;
            ComponentInstances Tutorial_App.C_CON, Tutorial_App.C_TG,
Tutorial_App.C_Robot, Tutorial_App.C_MobSim;
            Thread thread1{
                Tutorial_App.C_CON-
C_Console_Lib.C_Console.UpdatePorts;
            }
            Thread thread2{
                Tutorial_App.C_CON-
C_Console_Lib.C_Console.R_ReportingGoalStatus;
            }
            Thread thread3{
                Tutorial_App.C_CON-
C_Console_Lib.C_Console.R_Console;
            }
            Thread thread4{
                Tutorial_App.C_TG-
C_ToGoal_Lib.C_ToGoal.UpdatePorts;
            }
            Thread thread5{
                Tutorial_App.C_TG-C_ToGoal_Lib.C_ToGoal.R_ToGoal;
            }
            Thread thread6{
                Tutorial_App.C_Robot-
C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.R_UpdatePorts;
            }
            Thread thread7{
                Tutorial_App.C_Robot-
C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.R_Connecting;
            }
            Thread thread8{
                Tutorial_App.C_Robot-
C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.R_Reporting;
            }
            Thread thread9{
                Tutorial_App.C_Robot-
C_RobotBasePioneerHAL_Lib.C_RobotBasePioneerHAL.R_ProcessingCmds;
            }
        }
    }
}

```

```

        Thread thread10{
            Tutorial_App.C_MobSim-
C_MobileSimCon_Lib.C_MobileSimCon.UpRegMs;
        }
        Thread thread11{
            Tutorial_App.C_MobSim-
C_MobileSimCon_Lib.C_MobileSimCon.InReg;
        }
        Thread thread12{
            Tutorial_App.C_MobSim-
C_MobileSimCon_Lib.C_MobileSimCon.OutReg;
        }
    }
}

```

6.2.1.4 Activities

Each component activity corresponds to the place where it is defined the cyclic procedure that is executed while the correspondent state is active.

For each component, it is needed to have a C++ project, where it is be implemented all its activities procedures.

Anyway, before creating those projects, it is needed to go to the folder *NetBeansProjects* and create inside it a folder with the application name, *Tutorial_MinFr*, in order to save there all the application components projects.

Once that is done, to the create each component C++ project, it is needed to do the following procedure on the program *NetBeans*:

- Go the tab *File*;
- Select the option *New Project*;
- On the window that appears to select the type of project that you want to create, select the type *C/C++ Dynamic Library*, and press *Next*;
- Finally, on the same window, insert the project name, which must be the same as the component, for example *C_Console*, set as project location the application folder that is created inside the folder *NetBeans Projects*, and press *Finish*.

After completing this procedure, the project is then created successfully on the directory (*NetBeans Projects/Tutorial_MinFr*). All the components projects that belong to this application need to be created on this directory.

After creating all the components C++ projects, in order for each component project to work with the *MinFr* library it is needed to add it to the project. To do so for each component project, it is needed to do the following procedure on the program *NetBeans*:

- Go to the view *Projects*, press with the right button on the component project and then, press on the option *Properties*;
- On the window *Project Properties* that now appears, go to the category *Build* and then to the sub category *C++ Compiler*;
- There, go to the option *Include Directories*, and press on the three dots that appear on the right side of that option;

- On the window *Include Directories* that now appears, press *Add*, navigate until the folder *Headers* that is on the directory *PROGRAMA/MinFrLib*, select it and press *OK*;
- Finally, back to the window *Project Properties*, press *Apply* and *OK*.

After doing these steps for all the components projects, they are all ready to work with the state activities files.

Like mentioned before, each state activity procedure consists of two *C++* files:

- A *.h* (header file) where it is declared the structure of the state activity, which is the same for all the activities.
The structure consists of a class with four methods: *init*, *onEntry*, *tick* and *onExit*.
- A *.cpp* (source code file) where it is declared the body/procedure of each of those methods.

For each component, their correspondent states activities files can be easily generated. To generate them, for each component project, it is needed to do the following procedure on the program *Eclipse* by using the workspace *PROGRAMA/MinFrProjects*:

- Open the program *Eclipse*;
- On the view *Project Explorer*, press with the right button on the component library file, go to the option *Generate*, and then press the sub option *ActivityCode*;

After completing this procedure, the state activities files of all the state activities of each component are generated onto a folder with the component name, which is automatically created inside the folder *activity-gen*.

The *activity-gen* folder is also automatically created, and is inside the application project folder that is inside the folder *MinFrProjects*.

After generating the state activities files of all the components, it is needed to move them from where they were generated to their respective component *C++* project folder.

Finally, once all the files are moved to their respective *C++* projects folder, it is needed to import them to their respective *C++* projects, which is done by doing the following procedure on the program *NetBeans*:

- Go to the view *Projects*, press with the right button on the folder *Header Files* of the component project and press the option *Add Existing Item*;
- On the window *Select Item* that now appears, go to the correspondent component project folder, select all the state activities *.h* files that are there and press *Select*;
- Again on the view *Projects*, press with the right button on the folder *Source Files* of the component project and press the option *Add Existing Item*;
- On the window *Select Item* that now appears, go to the component project folder, select all the state activities *.cpp* files that are there and press *Select*;

After completing this procedure for all the components, it is finally possible to implement the activities procedure.

Like mentioned before, activities are structured as a class that has four methods:

- ***init*** – Corresponds to the procedure that is executed when the application is started, no matter if the state is active or not.
- ***onEntry*** – Corresponds to the procedure that is executed every time the state becomes active.
- ***tick*** – Corresponds to the procedure that is executed at each system cycle while the state is active.
- ***onExit*** – Corresponds to the procedure that is executed every time the state becomes inactive.

It is here that the user defines the procedure of the activity.

After defining the procedures of all the activities of each component, it is needed to build the correspondent C++ project of each component in order to generate their respective .so executable file.

On the following sections it will be explained per component, how to implement the procedure of each activity used on this simple application.

For the first two components, *C_Console* and *C_ToGoal*, it will be explained in detail the procedure of implementing each of their respective activities by following a two steps procedure.

For the other two components, *C_RobotBasePioneerHAL* and *C_MobileSimCon*, it will be only explained their basic procedure, since their implementation follows the same basic concepts explained on the detail for the other two components activities.

6.2.1.4.1 C_Console

According to what was mentioned before on section 6.2.1.1.1, the component *C_Console* has two activities, *A_Console* and *A_ReportingGoalStatus*.

Each of these two activities belongs to a different region.

Inside the same component, each region is concurrent from each other, which means that the state machine of each of them is independent.

Since none of the regions have more than one state, it is not needed to generate neither to consume events.

6.2.1.4.1.1 A_Console

The activity *A_Console* corresponds to the state *S_Console*, which corresponds to the user interface.

The procedure of this activity, consists on waiting for the user to insert the goal coordinates, which are then sent through the port *POut_Goal*, in order for the robot to move towards it.

If a new goal is inserted while the robot is already moving towards a goal position, the new goal is sent and the robot will start moving towards the new goal.

The process of implementing this procedure on the two generated activity files is divided in two steps:

1st step – On the header file, define the namespaces, auxiliary libraries and global variables used on this activity procedure.

The header file of this activity is *A_Console.h* and there needs to be defined the namespace *std*, which corresponds to the C++ standard library entities, and the *stdio.h* auxiliary library. This activity doesn't need any global variables.

So, after the first step, the header file *A_Console.h* looks like this (in bold there are the implementations made on this step and in normal there is the activity generated code):

```
#ifndef A_CONSOLE_H
#define A_CONSOLE_H

#include "minfr.h"
#include "stdio.h"

using namespace std;
using namespace minfr;

class A_Console:Activity {
public:
    virtual void init();
    virtual void onEntry();
    virtual void tick();
    virtual void onExit();
};

#endif
```

2nd step – On the source code file, define the procedure of each of the four activity class methods: *init*, *onEntry*, *tick* and *onExit*.

The functionality of each of these four methods is explained before on the section 6.2.1.4.

The source code file of this activity is *A_Console.cpp*, and for this activity it is only needed to define a procedure for the method *tick*, which corresponds to the user interface procedure.

So, after the second and last step, the source code file *A_Console.cpp* looks like this (in bold there is the implemented procedure on this step and in normal there is the activity generated code):

```
#include "A_Console.h"

void A_Console::init() {

}

void A_Console::onEntry() {

}

void A_Console::tick() {
    int goal_x, goal_y, goal_angle;
    cout << endl << " > Enter the x, y and angle coordinates (in milimeters and degrees)
of the position you want to move the robot to on the format x y angle: "
    cin >> goal_x >> goal_y >> goal_angle;

    MinfrMsg MsgGoal("GOAL");

    MsgGoal.setValue("x", Primitive<int>::toString(goal_x));
    MsgGoal.setValue("y", Primitive<int>::toString(goal_y));
    MsgGoal.setValue("angle", Primitive<int>::toString(goal_angle));

    getCompData() ->pushOutPortMsg("POut_Goal",MsgGoal);

}

void A_Console::onExit() {
```

```

}

extern "C" A_Console* create_A_Console(){
    return new A_Console;
}

```

On the method *tick*, the implemented code corresponds to the following procedure:

- Starts by requesting the user to insert the goal position, where the goal position is composed by three coordinates, x, y and angle;
- After the user inserting the goal position coordinates, it sends them on a *MinFrMsg* named *GOAL* to the component *C_ToGoal* through the port *POut_Goal*.

Note that, when it is sent something between components or states, all the data is converted to string.

6.2.1.4.1.2 A_ReportingGoalStatus

The activity *A_ReportingGoalStatus* corresponds to the state *S_ReportingGoalStatus*, which reports the current goal position while it is moving towards the goal.

The procedure of this activity, consists on reporting the current robot position, which is received through the port *PIn_GoalStatus*.

The process of implementing this procedure on the two generated activity files is divided in two steps:

1st step – On the header file, define the namespaces, auxiliary libraries and global variables used on this activity procedure.

The header file of this activity is *A_ReportingGoalStatus.h* and there needs to be defined the namespace *std*, which corresponds to the C++ standard library entities, the *stdio.h* auxiliary library, and the implemented *CommonFunctions.h* auxiliary library, which correspondent files need to be copied to this component project folder. This activity doesn't need any global variables.

So, after the first step, the header file *A_ReportingGoalStatus.h* looks like this (in bold there are the implementations made on this step and in normal there is the activity generated code):

```

#ifndef A_REPORTINGGOALSTATUS_H
#define A_REPORTINGGOALSTATUS_H

#include "minfr.h"
#include "CommonFunctions.h"
#include "stdio.h"

using namespace std;
using namespace minfr;

class A_ReportingGoalStatus:Activity {
public:
    virtual void init();
    virtual void onEntry();
    virtual void tick();
    virtual void onExit();
};

#endif

```

2nd step – On the source code file, define the procedure of each of the four activity class methods: *init*, *onEntry*, *tick* and *onExit*.

The functionality of each of these four methods is explained before on the section 6.2.1.4.

The source code file of this activity is *A_ReportingGoalStatus.cpp*, and for this activity it is only needed to define a procedure for the method *tick*, which corresponds to the procedure that reads the robot position sent from the component *C_ToGoal* and prints it.

So, after the second and last step, the source code file *A_ReportingGoalStatus.cpp* looks like this (in bold there is the implemented procedure on this step and in normal there is the activity generated code):

```
#include "A_ReportingGoalStatus.h"

void A_ReportingGoalStatus::init() {

}

void A_ReportingGoalStatus::onEntry() {

}

void A_ReportingGoalStatus::tick() {

    if (getCompData() -> hasInPortMsg("PIn_GoalStatus")) {

        MinFrMsg inPortMsg = getCompData() -> getLastInPortMsg("PIn_GoalStatus");

        if ("GOAL_STATUS" == inPortMsg.getName()) {

            positionCoords robot;
            int goal_status;

            goal_status = Primitive<int>::toPrimitive(inPortMsg.getValue("status"));
            robot.x = Primitive<double>::toPrimitive(inPortMsg.getValue("x"));
            robot.y = Primitive<double>::toPrimitive(inPortMsg.getValue("y"));
            robot.angle = Primitive<double>::toPrimitive(inPortMsg.getValue("angle"));

            if(goal_status == 0) {
                cout << "Robot is at position "<< robot.x << ", " << robot.y << ", " << "
with an orientation of " << robot.angle << " degrees." << endl;
            }
            if(goal_status == 1) {
                cout << "Robot has arrived at goal "<< robot.x << ", " << robot.y << ", "
<< " with an orientation of " << robot.angle << " degrees." << endl;
            }

        }

    }

}

void A_ReportingGoalStatus::onExit() {

}

extern "C" A_ReportingGoalStatus* create_A_ReportingGoalStatus() {
    return new A_ReportingGoalStatus;
}
```

On the method *tick*, the implemented code corresponds to the following procedure:

- Starts by checking if new *MinFrMsgs* arrived at the port *PIn_GoalStatus*, since the last time it was checked;

- If there are new *MinFrMsg*, it gets the newest and checks if its name is *GOAL_STATUS*;
- If the newest *MinFrMsg* name is *GOAL_STATUS*, it reads the *MinFrMsg* data, which corresponds to the robot position, and prints it.

6.2.1.4.2 C_ToGoal

According to what was mentioned before on section 6.2.1.1.2, the component *C_ToGoal* has two activities, *A_AtGoal* and *A_MovingToGoal*.

Both activities belong to the same region, which means that they can't be active both at the same time, but only one of them.

When a region has more than one state, it is needed to generate and consume events, in order to change from one state to another.

6.2.1.4.2.1 A_AtGoal

The activity *A_AtGoal* corresponds to the state *S_AtGoal*, which is active while the robot is stopped.

The procedure of this activity, consists on waiting to receive a new *MinFrMsg* through the port *Pln_Goal* with a new goal position. When it receives a new *MinFrMsg* with a new goal position, it generates an event *E_NewGoal* that makes this state inactive and the state *S_MovingToGoal* active.

When this state is inactive, it will only become active again when the state *S_MovingToGoal* generates an event *E_Done*.

The process of implementing this procedure on the two generated activity files is divided in two steps:

1st step – On the header file, define the namespaces, auxiliary libraries and global variables used on this activity procedure.

The header file of this activity is *A_AtGoal.h* and there needs to be defined the namespace *std*, which corresponds to the C++ standard library entities, the *stdio.h* auxiliary library, and the implemented *CommonFunctions.h* auxiliary library. This activity doesn't need any global variables.

So, after the first step, the header file *A_AtGoal.h* looks like this (in bold there are the implementations made on this step and in normal there is the activity generated code):

```
#ifndef A_ATGOAL_H
#define A_ATGOAL_H

#include "minfr.h"
#include "CommonFunctions.h"
#include "stdio.h"

using namespace std;
using namespace minfr;

class A_AtGoal:Activity {
public:
    virtual void init();
    virtual void onEntry();
    virtual void tick();
    virtual void onExit();
};

#endif
```

2nd step – On the source code file, define the procedure of each of the four activity class methods: *init*, *onEntry*, *tick* and *onExit*.

The functionality of each of these four methods is explained before on the section 6.2.1.4.

The source code file of this activity is *A_AtGoal.cpp*, and for this activity it is only needed to define a procedure for the method *tick*, which corresponds to the user interface procedure.

So, after the second and last step, the source code file *A_AtGoal.cpp* looks like this (in bold there is the implemented procedure on this step and in normal there is the activity generated code):

```
#include "A_AtGoal.h"

void A_AtGoal::init(){
}

void A_AtGoal::onEntry(){
}

void A_AtGoal::tick(){
    if (getCompData() -> hasInPortMsg("PIn_Goal")) {

        MinFrMsg MsgNewGoal = getCompData() -> getLastInPortMsg("PIn_Goal");

        if ("GOAL" == MsgNewGoal.getName()){

            positionCoords goal;

            goal.x = Primitive<double>::toPrimitive(MsgNewGoal.getValue("x"));
            goal.y = Primitive<double>::toPrimitive(MsgNewGoal.getValue("y"));
            goal.angle = Primitive<double>::toPrimitive(MsgNewGoal.getValue("angle"));

            MinFrMsg eMsg("E_NewGoal");

            eMsg.setValue("x", Primitive<double>::toString(goal.x));
            eMsg.setValue("y", Primitive<double>::toString(goal.y));
            eMsg.setValue("angle", Primitive<double>::toString(goal.angle));

            getCompData()->generateEvent("E_NewGoal", eMsg);

        }

    }

}

void A_AtGoal::onExit(){
}

extern "C" A_AtGoal* create_A_AtGoal(){
    return new A_AtGoal;
}
```

On the method *tick*, the implemented code corresponds to the following procedure:

- Starts by checking if, since the last time it was checked, new *MinFrMsgs* arrived at the port *PIn_Goal*;
- If there are new *MinFrMsgs*, it gets the newest and checks if its name is *GOAL*;

- If the newest *MinFrMsg* name is *GOAL*, it reads the *MinFrMsg* data, which corresponds to the coordinates of a new goal position to be executed, and generates an event *E_NewGoal*, where it sends those new goal position coordinates on a *MinFrMsg* named *E_NewGoal* to the state *S_MovingToGoal*;

By generating the event *E_NewGoal*, this state becomes inactive and the state *S_MovingToGoal* active.

6.2.1.4.2.2 A_MovingToGoal

The activity *A_MovingToGoal* corresponds to the state *S_MovingToGoal*, which is active while the robot is moving towards a goal position.

The procedure of this activity consists on calculating the velocities of the robot with the implemented algorithm *ToGoal*, while the robot still didn't arrive at the current goal position. Those velocities are sent to the robot on a *MinFrMsg*, through the port *POut_Velocities*.

If a new goal position is received through the port *PIn_Goal* while the robot is already moving towards a goal position, the robot will start moving towards the new goal and ignore the old one.

Once the robot arrives at the current goal position, an event *E_AtGoal* is generated, which makes this state inactive and the state *S_AtGoal* active.

When this state is inactive, it will only become active again when the state *S_AtGoal* generates an event *E_NewGoal*.

The process of implementing this procedure on the two generated activity files is divided in two steps:

1st step – On the header file, define the namespaces, auxiliary libraries and global variables used on this activity procedure.

The header file of this activity is *A_MovingToGoal.h* and there needs to be defined the namespace *std*, which corresponds to the C++ standard library entities, the *stdio.h* auxiliary library, and the two implemented auxiliary libraries *CommonFunctions.h* and *ToGoal.h*. This activity has some global variables, which are declared on the private part of the class *A_MovingToGoal*.

This header file also includes some *defines*, which correspond to the robot control constants and constraints values that are used by the algorithm *ToGoal* to calculate the robot velocities.

Then, after the first step, the header file *A_AtGoal.h* looks like this (in bold there are the implementations made on this step and in normal there is the activity generated code):

```
#ifndef A_MOVINGTOGOAL_H
#define A_MOVINGTOGOAL_H

#include "minfr.h"
#include "CommonFunctions.h"
#include "stdio.h"
#include "ToGoal.h"

#define distanceThreshold 100 // Distance Threshold Error Tolerance (in mms)
#define angleThreshold 5 // Angle Threshold Error Tolerance (in degrees)
#define unitV 0.001 // Unit of the translational speed (v), 0.001 means 1 mm/sec
#define K_V 0.3 // Constant of conversion from distance (mm) to translational speed (mms/sec)
#define K_W 0.01*30 // Constant of conversion from alpha (degrees) to rotational speed (deg/sec)
```

```

#define V_MAX 300 // Maximum translational speed (v) permitted
#define W_MAX 0.2*10 // Maximum rotational speed (w) permitted

using namespace std;
using namespace minfr;

class A_MovingToGoal:Activity {
public:
    virtual void init();
    virtual void onEntry();
    virtual void tick();
    virtual void onExit();
private:
    robotData robotInfo;
    positionCoords goal, robot;
    velocities OutputVelocities;
    bool GoalReached;
    ToGoal *TG;
};

#endif

```

2nd step – On the source code file, define the procedure of each of the four activity class methods: *init*, *onEntry*, *tick* and *onExit*.

The functionality of each of these four methods is explained before on the section 6.2.1.4.

The source code file of this activity is *A_MovingToGoal.cpp*, and for this activity it is needed to define procedures for the method *init*, *onEntry* and *tick*.

Then, after the second and last step, the source code file *A_MovingToGoal.cpp* looks like this (in bold there is the implemented procedure on this step and in normal there is the activity generated code):

```

#include "A_MovingToGoal.h"

void A_MovingToGoal::init() {

    robotInfo.Kv = K_V;
    robotInfo.Kw = K_W;
    robotInfo.Vmax = V_MAX;
    robotInfo.Wmax = W_MAX;
    robotInfo.DistanceThresholdError = distanceThreshold;
    robotInfo.AngleThresholdError = angleThreshold;

    GoalReached = true;

}

void A_MovingToGoal::onEntry() {

    MinfrMsg eMsg = getCompData()->getLastEventMsg("E_NewGoal");

    goal.x = Primitive<double>::toPrimitive(eMsg.getValue("x"));
    goal.y = Primitive<double>::toPrimitive(eMsg.getValue("y"));
    goal.angle = Primitive<double>::toPrimitive(eMsg.getValue("angle"));

    GoalReached = false;

    TG = new ToGoal(robotInfo);

}

void A_MovingToGoal::tick() {

    if (getCompData() -> hasInPortMsg("PIn_Goal")) {

        MinfrMsg MsgNewGoal = getCompData() -> getLastInPortMsg("PIn_Goal");

        if ("GOAL" == MsgNewGoal.getName()) {

            goal.x = Primitive<double>::toPrimitive(MsgNewGoal.getValue("x"));

```

```

goal.y = Primitive<double>::toPrimitive(MsgNewGoal.getValue("y"));
goal.angle = Primitive<double>::toPrimitive(MsgNewGoal.getValue("angle"));

delete TG;

TG = new ToGoal(robotInfo);

}

}

if (GoalReached == false) {

    if (getCompData() -> hasInPortMsg("PIn_RobotPosition")) {

        MinfrMsg MsgRobotPos = getCompData() ->
getLastInPortMsg("PIn_RobotPosition");

        if ("ROBOT_POS" == MsgRobotPos.getName()){

            robot.x = Primitive<double>::toPrimitive(MsgRobotPos.getValue("x"));
            robot.y = Primitive<double>::toPrimitive(MsgRobotPos.getValue("y"));
            robot.angle =
Primitive<double>::toPrimitive(MsgRobotPos.getValue("angle"));

        }

    }

    if (TG->RobotAtPosition(robot,goal) == false){

        OutputVelocities = TG->CalculateVelocities(robot,goal);

        int V_INT = round(OutputVelocities.v);
        int W_INT = round(OutputVelocities.w);

        MinfrMsg MsgVelocities("NAV_VEL");

        MsgVelocities.setValue("v", Primitive<double>::toString(V_INT));
        MsgVelocities.setValue("w", Primitive<double>::toString(W_INT));

        getCompData()->pushOutPortMsg("POut_Velocities",MsgVelocities);

        MinfrMsg MsgGoalStatus("GOAL_STATUS");

        MsgGoalStatus.setValue("status", Primitive<int>::toString(0));
        MsgGoalStatus.setValue("x", Primitive<int>::toString(robot.x));
        MsgGoalStatus.setValue("y", Primitive<int>::toString(robot.y));
        MsgGoalStatus.setValue("angle", Primitive<int>::toString(robot.angle));

        getCompData()->pushOutPortMsg("POut_GoalStatus",MsgGoalStatus);

    } else {

        MinfrMsg MsgVelocities("NAV_VEL");

        MsgVelocities.setValue("v", Primitive<double>::toString(0));
        MsgVelocities.setValue("w", Primitive<double>::toString(0));

        getCompData()->pushOutPortMsg("POut_Velocities",MsgVelocities);

        MinfrMsg MsgGoalStatus("GOAL_STATUS");

        MsgGoalStatus.setValue("status", Primitive<int>::toString(1));
        MsgGoalStatus.setValue("x", Primitive<int>::toString(robot.x));
        MsgGoalStatus.setValue("y", Primitive<int>::toString(robot.y));
        MsgGoalStatus.setValue("angle", Primitive<int>::toString(robot.angle));

        getCompData()->pushOutPortMsg("POut_GoalStatus",MsgGoalStatus);

        GoalReached = true;

        MinfrMsg eMsg("E_AtGoal");

        getCompData()->generateEvent("E_AtGoal", eMsg);

```



```

    }

}

void A_MovingToGoal::onExit() {

}

extern "C" A_MovingToGoal* create_A_MovingToGoal() {
    return new A_MovingToGoal;
}

```

On the method *init*, the implemented code reads all the robot control constant and constraint values to the data structure *robotInfo* and sets the value of the global variable *GoalReached* to true.

On the method *onEntry*, the implemented code gets the last event *E_NewGoal* *MinFrMsg* that was generated by the state *S_AtGoal*, reads its data, which corresponds to the new goal position coordinates, sets those coordinates as the current goal position coordinates, sets the value of the global variable *GoalReached* to false, and creates an object of the class *ToGoal*, by giving as input the data structure *robotInfo*.

On the method *tick*, the implemented code corresponds to the following procedure:

- It starts by checking if, since the last time it was checked, new *MinFrMsgs* arrived at the port *PIn_Goal*;
 - If there are new *MinFrMsgs*, it gets the last and checks if its name is *GOAL*;
 - If the name of that *MinFrMsg* is *GOAL*, it reads the *MinFrMsg* data, which corresponds to the coordinates of a new goal position to be executed, and sets those coordinates as the current goal position coordinates;
- After that, it checks if, since the last time it was check, new *MinFrMsgs* arrived at the port *PIn_RobotPosition*;
 - If there are new *MinFrMsgs*, it gets the last and checks if its name is *ROBOT_POS*;
 - If the name of that *MinFrMsg* is *ROBOT_POS*, it reads the *MinFrMsg* data, which corresponds to the coordinates of the current robot position, and sets those coordinates as the current robot position coordinates;
- Then, it checks if, according to the current robot and goal position, the robot is already at the current goal position or not, by calling the method *RobotAtPosition* of the class *ToGoal*;
 - If the robot is not already at the current goal position, it calculates the robot velocities according to the distance from the current robot position to the current goal positions, sends them on a *MinFrMsg* named *NAV_VEL* through the port *POut_Velocities*, and sends the coordinates of the current robot position on a *MinFrMsg* named *GOAL_STATUS* through the port *POut_GoalStatus*.

- If the robot is already at the current goal position, it sends two null robot velocities on a *MinFrMsg* named *NAV_VEL* through the port *POut_Velocities*, in order to stop the robot, sends the coordinates of the current robot position on a *MinFrMsg* named *GOAL_STATUS* through the port *POut_GoalStatus*, sets the value of the global variable *GoalReached* to true, and generates an event *E_AtGoal*, where it sends nothing on a *MinFrMsg* named *E_AtGoal* to the state *S_AtGoal*.

By generating the event *E_AtGoal*, this state becomes inactive and the state *S_AtGoal* active.

6.2.1.4.3 C_RobotBasePioneerHAL

According to what was mentioned before on section 6.2.1.1.3, the component *C_RobotBasePioneerHAL* has four activities, *A_Connected*, *A_Connecting*, *A_ProcessingCmds* and *A_Reporting*.

The implementation of this four activities is not explained in detail, since it follows the same basic concepts explained in detail for the activities of the two other components, *C_Console* and *C_ToGoal*.

The two first activities, *A_Connecting* and *A_Connected*, belong to the same region, which means that they can't be active both at the same time, but only one of them.

The activity *A_Connecting* corresponds to the state *S_Connecting*, which is active while the application is trying to establish connection with the simulator.

The procedure of this activity consists on sending a *MinFrMsg* through the port *POut_MsgRobot*, to the component *C_MobileSimCon*, in order for it to establish the connection with the simulator. After that, it generates an event *E_Done* that makes this state inactive and the state *S_Connected* active.

The activity *A_Connected* corresponds to the state *S_Connected*, which is active when the application becomes connected to the simulator. This activity has no procedure.

Each of the two activities *A_ProcessingCmds* and *A_Reporting* belong to a different region.

The activity *A_ProcessingCmds* corresponds to the state *S_ProcessingCmds*, which procedure consists on receiving *MinFrMsgs* with the commands for the robot, through the port *PIn_Cmd* from other application components, and then, send those commands on *MinFrMsgs* through the port *POut_MsgRobot*, to the component *C_MobileSimCon*.

The activity *A_Reporting* corresponds to the state *S_Reporting*, which procedure consists on receiving *MinFrMsgs* from the component *C_MobileSimCon*, with the robot actual position or with the laser readings, through the ports *PIn_RobotPosData* and *PIn_LaserData*, respectively, and then, send that position or the laser readings, to other application components, on *MinFrMsgs* through the ports *POut_RobotPos* and *POut_Laser*, respectively.

6.2.1.4.4 C_MobileSimCon

According to what was mentioned before on section 6.2.1.1.4, the component *C_MobileSimCon* has two activities, *InputActivity* and *OutputActivity*.

The implementation of this two activities is not explained in detail, since it follows the same basic concepts explained for the activities of the two other components, *C_Console* and *C_ToGoal*.

Each of the two activities *InputActivity* and *OutputActivity* belongs to a different regions.

The activity *InputActivity* corresponds to the state *InMsgSt*, which procedure consists on receiving the robot actual position and the laser readings, through sockets from the simulator,

and then send that position and the laser readings to the component *C_RobotBasePioneerHAL*, on *MinFrMsgs* through the ports *POut_RobotPosData* and *POut_LaserData*, respectively.

The activity *OutputActivity* corresponds to the state *OutMsgSt*, which procedure consists on receiving *MinFrMsgs* with the commands for the robot, through the port *inPort* from the component *C_RobotBasePioneerHAL*, and then put those commands with the correct format, in order for the simulator to be able to process them, and send them through sockets to the simulator.

6.2.1.5 How to run the application

Before running the application, it is needed to put all the files on their correct places in order to be able to run the application. To do so, it is needed to do the following three steps:

- Go to the folder *MinFrApps* and create inside it a folder with the application name, *Tutorial_MinFr*, in order to save there all the application executable files;
- Copy the *.so* files of each component used by the application from the folder *dist* that is inside of each correspondent component C++ project, to the folder that was just now created (*MinFrApps/Tutorial_MinFr*).
- Copy also to the same folder, the *MinFr* server executable file and its respective libraries, as well as, the server executable file that establishes the connection between the application with the simulator;
 - The *MinFr* server executable file is named *minfrhost* and its respective libraries are:
 - *libminfr.so*;
 - *libtbb.so*;
 - *libtbb.so.2*.
 - The server executable file that establishes the connection between the application with the simulator is named *servidor_hmi_2*.

(The files mentioned on this last step can be copied from other application folder that is inside the folder *MinFrApps*.)

Now that the application is ready to be run, in order to run it, it is needed to do the following procedure, which is divided in three phases:

1st phase:

- Open a new terminal window, which will be called as *terminal1* during this explanation;
- Through *terminal1*, go to the *Tutorial_MinFr* folder that is inside the folder *MinFrApps*, and insert the following code line, in order to run the simulator:

```
MobileSim &
```

After inserting this code line and pressing *Enter*, a window will appear asking if you want to load a map for the simulation or not.

Here, press the button *No Map*.

- After appearing the simulator window, go back to *terminal1*, press *Enter*, insert the following code line, in order to run the server that establishes the connection between the application with the simulator, and press *Enter*:

```
./servidor_hmi_2
```

By now, *terminal1* and the simulator should look like the image below, Fig. 115.

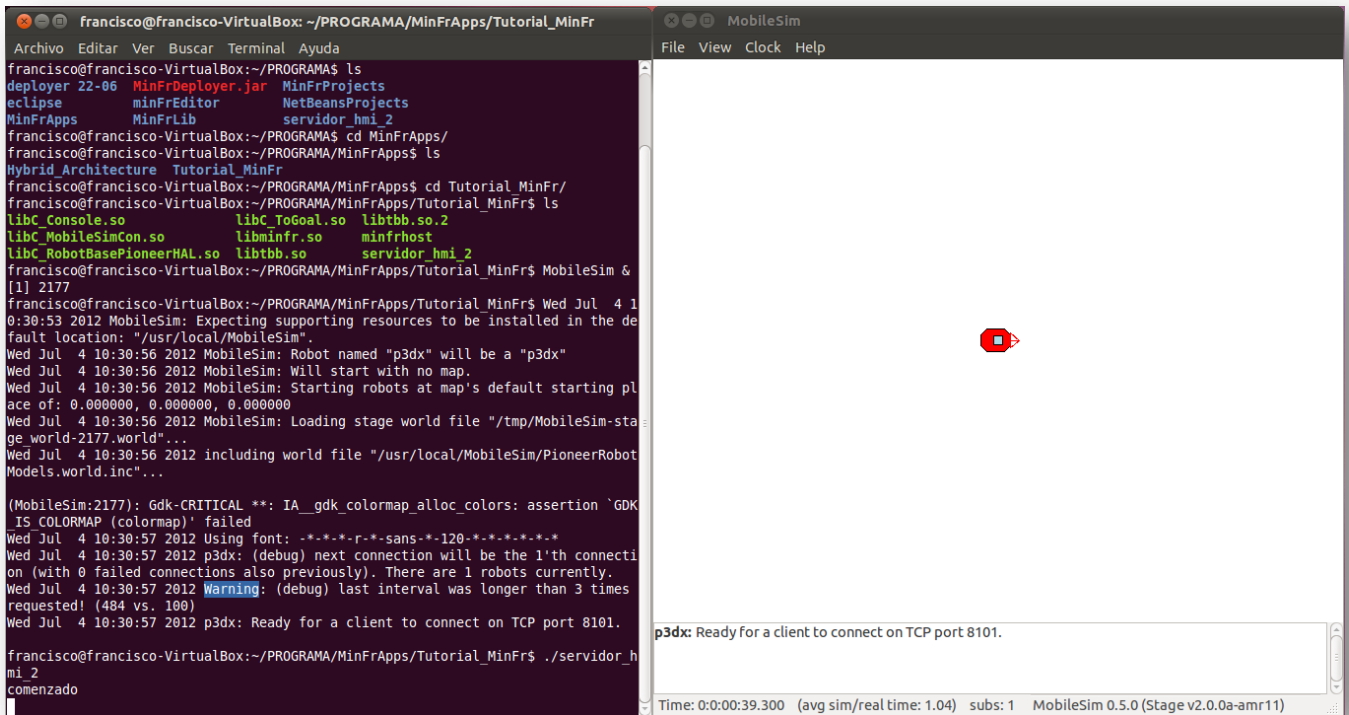


Fig. 115 - *terminal1* and simulator window after phase 1

2nd phase:

- Open a new terminal window, which will be called as *terminal2* for this explanation;
- Through *terminal2*, go to the *Tutorial_MinFr* folder that is inside the folder *MinFrApps*, insert the following code line, in order export the path to this folder as *LD_LIBRARY_PATH*, and press *Enter*:

```
export LD_LIBRARY_PATH=home/francisco/PROGRAMA/MinFrApps/Tutorial_MinFr:$LD_LIBRARY_PATH
```

- Also on *terminal2*, insert the following code line, in order to run the *MinFr* server executable file, and press *Enter*:

```
./minfrhost 50001
```

By now *terminal2* should look like the image below, Fig. 116.

```

francisco@francisco-VirtualBox: ~/PROGRAMA/MinFrApps/Tutorial_MinFr
Archivo Editar Ver Buscar Terminal Ayuda
francisco@francisco-VirtualBox:~$ ls
COMPILER_ADA  Dropbox      Imágenes      Plantillas  Videos
Descargas     Escritorio   Música         PROGRAMA
Documentos    examples.desktop netbeans-7.0.1 Público
francisco@francisco-VirtualBox:~$ cd PROGRAMA/
francisco@francisco-VirtualBox:~/PROGRAMA$ ls
deployer 22-06 MinFrDeployer.jar MinFrProjects
eclipse  minFrEditor NetBeansProjects
MinFrApps MinFrLib      servidor_hmi_2
francisco@francisco-VirtualBox:~/PROGRAMA$ cd MinFrApps/
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps$ ls
Hybrid_Architecture Tutorial_MinFr
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps$ cd Tutorial_MinFr/
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Tutorial_MinFr$ ls
libC_Console.so      libC_ToGoal.so  libtbb.so.2
libC_MobileSimCon.so libminfr.so     minfrhost
libC_RobotBasePioneerHAL.so libtbb.so      servidor_hmi_2
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Tutorial_MinFr$ export LD_LIBRARY_PATH=home/francisco/PROGRAMA/MinFrApps/Tutorial_MinFr:$LD_LIBRARY_PATH
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Tutorial_MinFr$ ./minfrhost
50001
After deployment completes, press enter to start the execution

```

Fig. 116 - *terminal2* after phase 2

3rd phase:

- Open a new terminal window, which will be called as *terminal3* for this explanation;
- Through *terminal3*, go to the *Tutorial_MinFr* folder that is inside the folder *MinFrProjects*, and insert the following code line, in order to deploy the application distribution:

```
java -jar ../../MinFrDeployer.jar 50000 Tutorial_Dist.minfr
```

By now *terminal3* should look like the image below, Fig. 117.

```

francisco@francisco-VirtualBox: ~/PROGRAMA/MinFrProjects/Tutorial_MinFr
Archivo Editar Ver Buscar Terminal Ayuda
francisco@francisco-VirtualBox:~$ ls
COMPILER_ADA  Dropbox      Imágenes      Plantillas  Videos
Descargas     Escritorio   Música         PROGRAMA
Documentos    examples.desktop netbeans-7.0.1 Público
francisco@francisco-VirtualBox:~$ cd PROGRAMA/
francisco@francisco-VirtualBox:~/PROGRAMA$ ls
deployer 22-06 MinFrDeployer.jar MinFrProjects
eclipse  minFrEditor NetBeansProjects
MinFrApps MinFrLib      servidor_hmi_2
francisco@francisco-VirtualBox:~/PROGRAMA$ cd MinFrProjects/
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrProjects$ ls
Hybrid_Architecture Tutorial_MinFr
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrProjects$ cd Tutorial_MinFr/
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrProjects/Tutorial_MinFr$ ls
activity-gen          C_RobotBasePioneerHAL_Lib.minfr Tutorial_Dist.minfr
C_Console_Lib.minfr   C_ToGoal_Lib.minfr
C_MobileSimCon_Lib.minfr Tutorial_App.minfr
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrProjects/Tutorial_MinFr$ java -jar
r ../../MinFrDeployer.jar 50000 Tutorial_Dist.minfr

```

Fig. 117 - *terminal3* after phase 3

- Wait until appears *Deployment complete* on *terminal3*, and when it does, go back to *terminal2* and press *Enter* , in order to run the application.

By now, *terminal3* and *terminal2* should look like the images below, Fig. 118 and 119, respectively.

```

francisco@francisco-VirtualBox: ~/PROGRAMA/MinFrProjects/Tutorial_MinFr
Archivo Editar Ver Buscar Terminal Ayuda
gionId" DataType="String">1002004</Label></Msg>
Response: OK
Destination: Process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">
6</Label><Label Name="AckCode" DataType="String">18</Label><Label Name="UniqueRe
gionId" DataType="String">1002001</Label></Msg>
Response: OK
Destination: Process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">
5</Label><Label Name="AckCode" DataType="String">19</Label><Label Name="UniqueRe
gionId" DataType="String">1002000</Label></Msg>
Response: OK
Destination: Process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">
4</Label><Label Name="AckCode" DataType="String">20</Label><Label Name="UniqueRe
gionId" DataType="String">1001001</Label></Msg>
Response: OK
Destination: Process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">
3</Label><Label Name="AckCode" DataType="String">21</Label><Label Name="UniqueRe
gionId" DataType="String">1001000</Label></Msg>
Response: OK
Destination: Process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">
2</Label><Label Name="AckCode" DataType="String">22</Label><Label Name="UniqueRe
gionId" DataType="String">1000001</Label></Msg>
Response: OK
Destination: Process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">
1</Label><Label Name="AckCode" DataType="String">23</Label><Label Name="UniqueRe
gionId" DataType="String">1000003</Label></Msg>
Response: OK
Destination: Process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">
0</Label><Label Name="AckCode" DataType="String">24</Label><Label Name="UniqueRe
gionId" DataType="String">1000000</Label></Msg>
Response: OK
Deployment complete
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrProjects/Tutorial_MinFr$

```

Fig. 118 - terminal3 when the deployment is complete

```

francisco@francisco-VirtualBox: ~/PROGRAMA/MinFrApps/Tutorial_MinFr
Archivo Editar Ver Buscar Terminal Ayuda
francisco@francisco-VirtualBox:~$ ls
COMPILER_ADA  Dropbox      Imágenes      Plantillas  Videos
Descargas     Escritorio   Música         PROGRAMA
Documentos    examples.desktop netbeans-7.0.1 Público
francisco@francisco-VirtualBox:~$ cd PROGRAMA/
francisco@francisco-VirtualBox:~/PROGRAMA$ ls
deployer 22-06 MinFrDeployer.jar MinFrProjects
eclipse  minFrEditor NetBeansProjects
MinFrApps MinFrLib      servidor hmi_2
francisco@francisco-VirtualBox:~/PROGRAMA$ cd MinFrApps/
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps$ ls
Hybrid Architecture Tutorial_MinFr
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps$ cd Tutorial_MinFr/
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Tutorial_MinFr$ ls
libC Console.so libC ToGoal.so libtbb.so.2
libC MobileSimCon.so libminfr.so minfrhost
libC RobotBasePioneerHAL.so libtbb.so servidor hmi_2
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Tutorial_MinFr$ export LD LI
BRARY_PATH=home/francisco/PROGRAMA/MinFrApps/Tutorial_MinFr:$LD_LIBRARY_PATH
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Tutorial_MinFr$ ./minfrhost
50001
After deployment completes, press enter to start the execution
Realizada la primera conexion a 127.0.0.1 : 8102
error on select(): 9 Bad file descriptor

Process Started
Realizada la primera conexion a 127.0.0.1 : 8103

> Enter the x, y and angle coordinates (in milimeters and degrees) of the posit
ion you want to move the robot to on the format x y angle:

```

Fig. 119 - terminal2 after pressing *Enter*, when the deployment is complete

Now that the application is running, in order to move the robot insert the coordinates of the goal in millimeters and degrees on *terminal2*, which is the terminal that it is used, when it is needed to interact with the application.

On the image below, Fig. 120, it is shown an example of how to insert a goal position coordinates, which goal corresponds to the position 1000, 1000 with a final orientation of 0°, and also it is shown that the robot arrived at it successfully.

In order to verify if the robot really arrived at the correct final position, the robot position is printed on *terminal2* at each second while it is moving.

The red trail corresponds to the trajectory that the robot took from its initial position 0, 0 to the final position 1000, 1000.

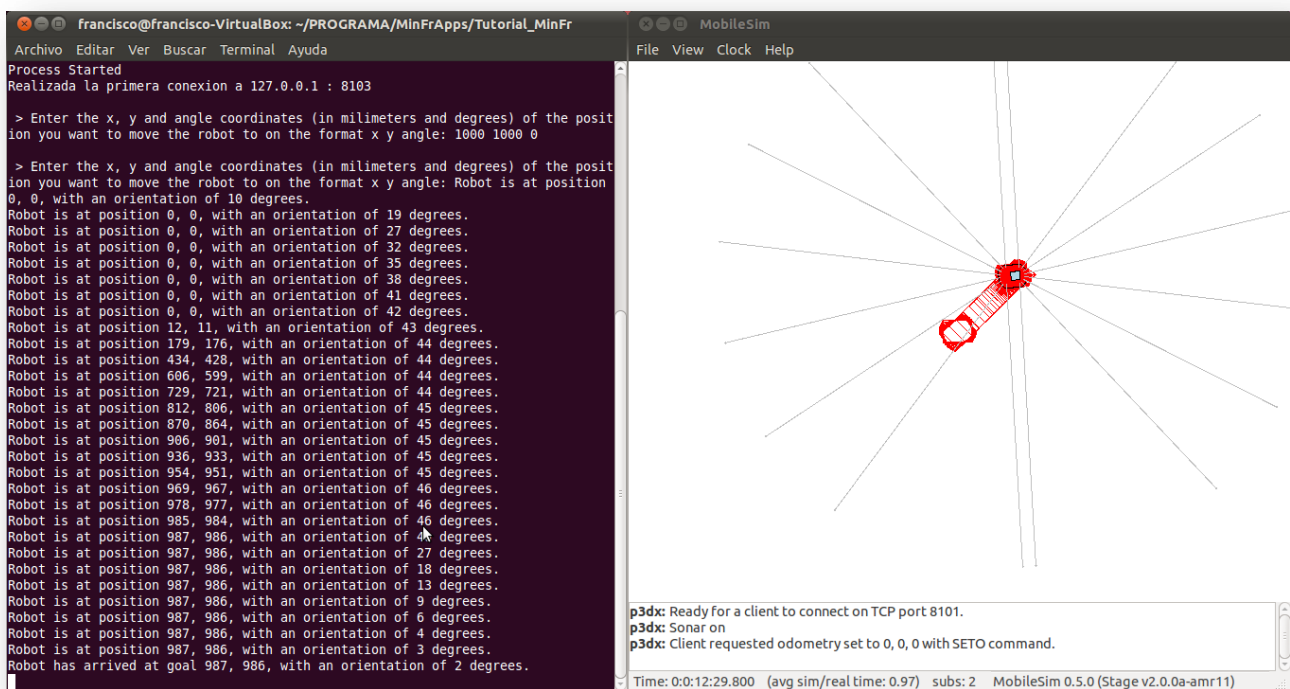


Fig. 120 - Application running (on the left, terminal2, which it the terminal used to when it is needed to interact with the application, and on the right, the simulator)

6.2.2 Hybrid Architecture

On this section it will be described the implementation of the application described on chapter 5.

The goal of this application is to fulfill the requirements mentioned on chapter 5, by implementing an application with the also mentioned components.

The structure of this application in terms of components and connections between them is shown below, on Fig. 121.

In order to implement the components libraries, application structure and distribution, which are all defined on *.minfr* files, it is needed to use the program *Eclipse*, open the workspace *PROGRAMA/MinFrProjects*, and then create a new project for this application named *Hybrid_Architecture*.

After completing this procedure, the project is then created successfully on the workspace folder (*PROGRAMA/MinFrProjects*). All the files with the extension *.minfr* that belong to this application need to be created inside this project.

On the following sub sections it is explained how are implemented the component libraries of each component used by the application, the application structure and distribution, and the activities of each component used by the application.

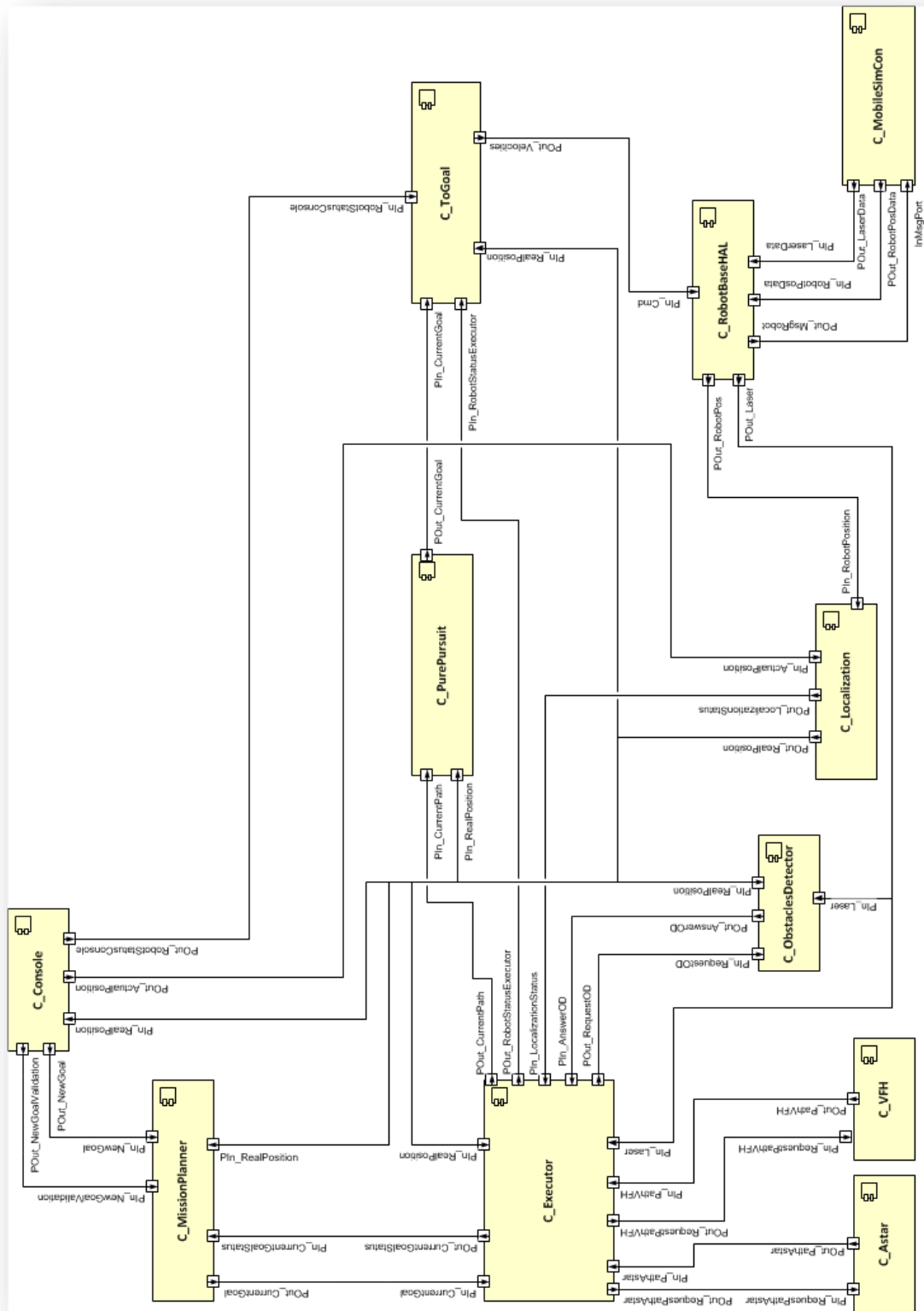


Fig. 121 - Structure of the hybrid application

6.2.2.1 Component Libraries

Like mentioned before, it is on the component library, where it is defined the structure of each component that is used by the application in terms of ports, regions, states, events and transitions.

Each component has its own component library, which is defined on a *.minfr* file on the folder *MinFrProjects*.

To create each component library, it is needed to do the follow the procedure mentioned on the section 6.2.1.1.

For each component of this application, it is only mentioned its structure, since the implementation of their correspondent component libraries follows the same basic concepts explained on the detail on section 6.2.1.1 for the tutorial application components libraries.

6.2.2.1.1 C_Console

This component corresponds to the MinFr implementation of the Console component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 122.

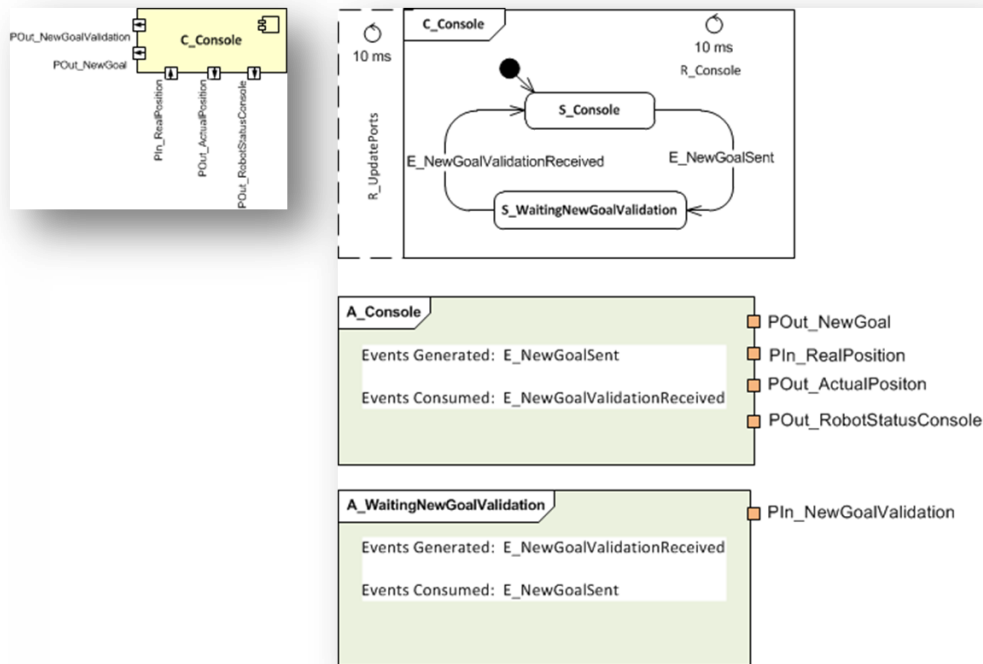


Fig. 122 - Structure of the component *C_Console*

6.2.2.1.2 C_MissionPlanner

This component corresponds to the MinFr implementation of the Mission Planner component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 123.

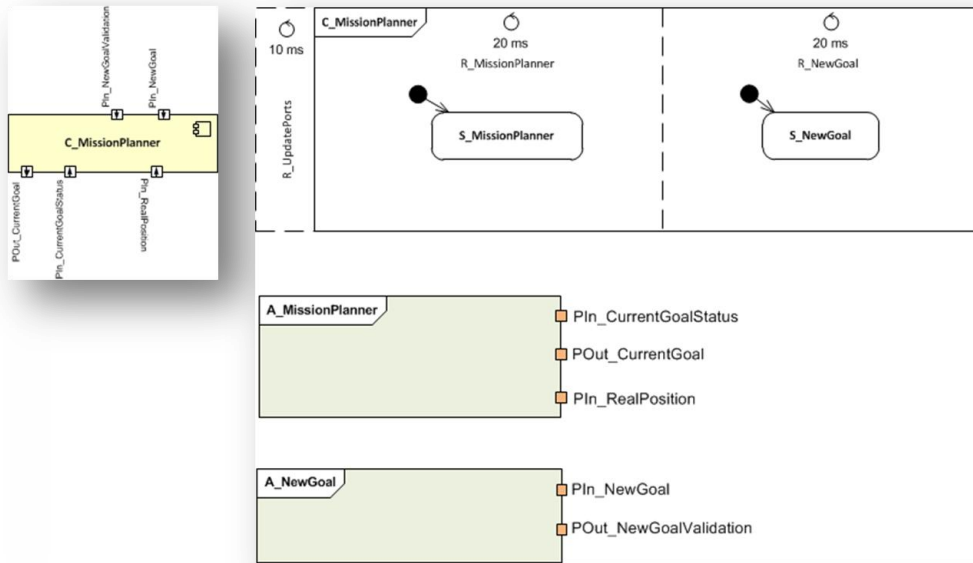


Fig. 123 - Structure of the component *C_MissionPlanner*

6.2.2.1.3 C_ObstaclesDetector

This component corresponds to the MinFr implementation of the Obstacles Detector component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 124.

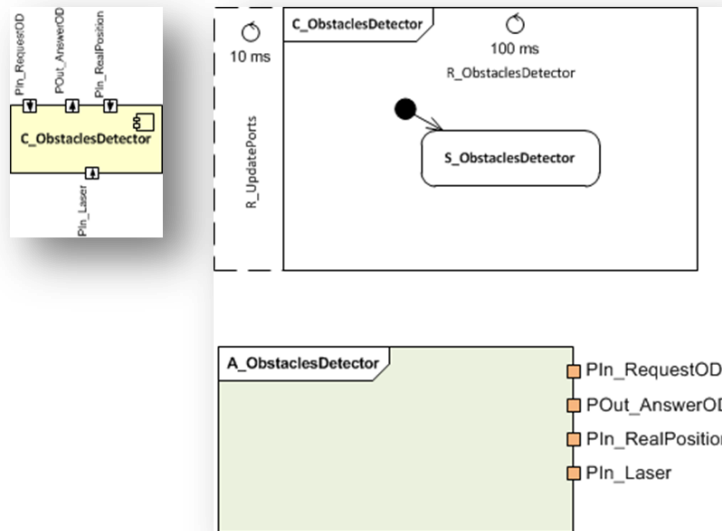


Fig. 124 - Structure of the component *C_ObstaclesDetector*

6.2.2.1.4 C_Executor

This component corresponds to the MinFr implementation of the Executor component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 125.

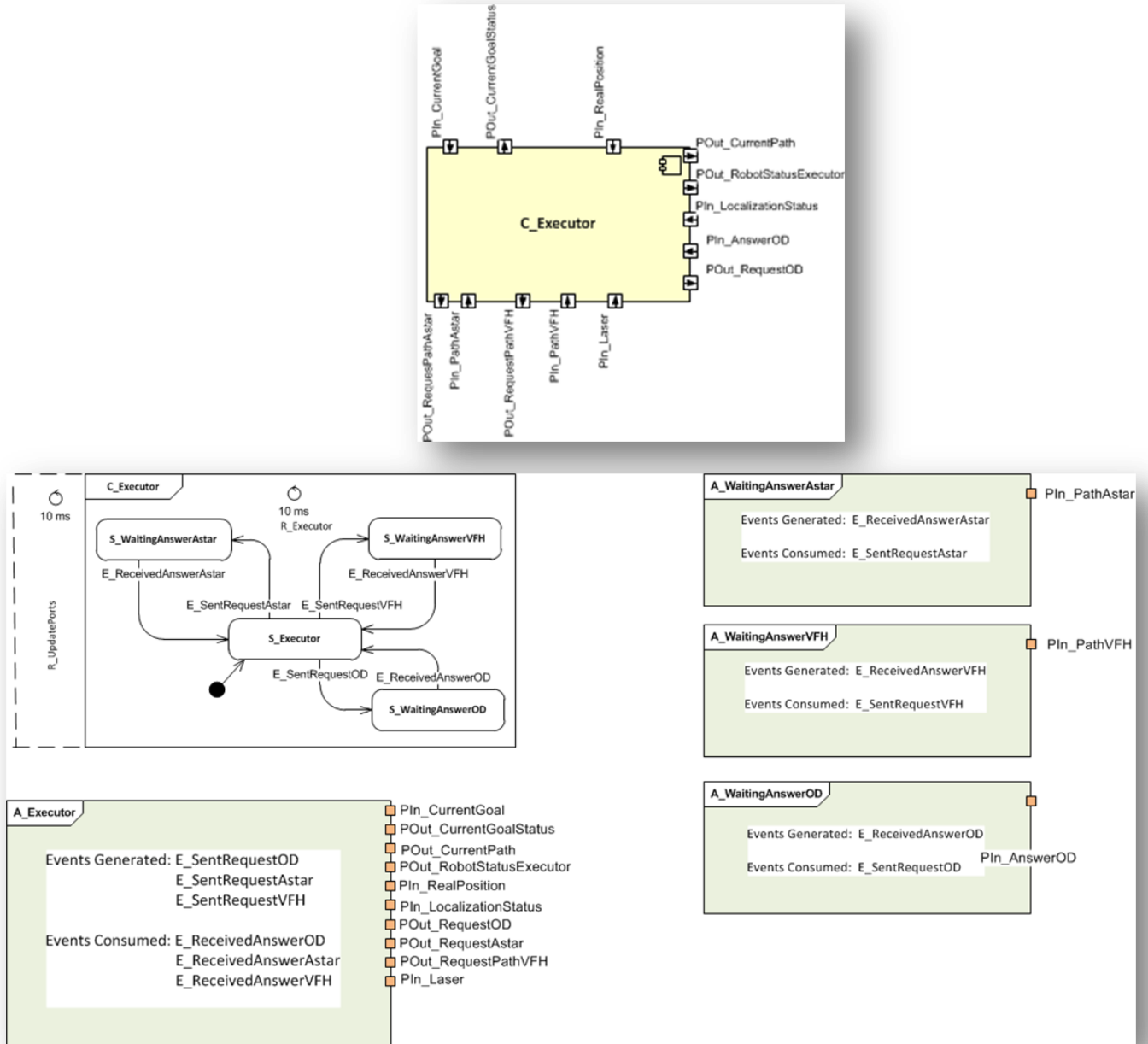


Fig. 125 - Structure of the component C_Executor

6.2.2.1.5 C_Astar

This component corresponds to the MinFr implementation of the Astar component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 126.

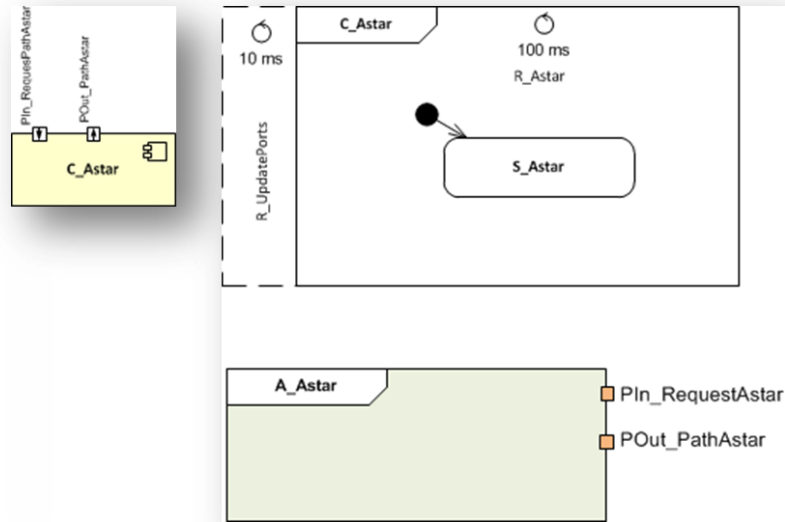


Fig. 126 - Structure of the component C_Astar

6.2.2.1.6 C_VFH

This component corresponds to the MinFr implementation of the VFH component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 127.

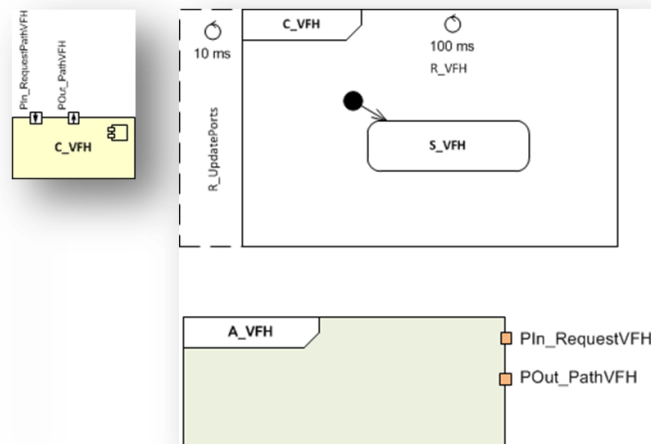


Fig. 127 - Structure of the component C_VFH

6.2.2.1.7 C_PurePursuit

This component corresponds to the MinFr implementation of the Pure Pursuit component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 128.

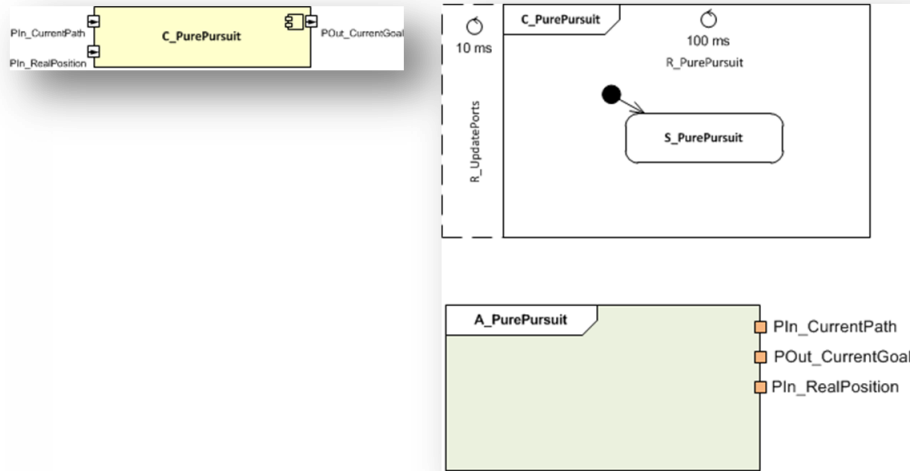


Fig. 128 - Structure of the component C_PurePursuit

6.2.2.1.8 C_ToGoal

This component corresponds to the MinFr implementation of the ToGoal component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 129.

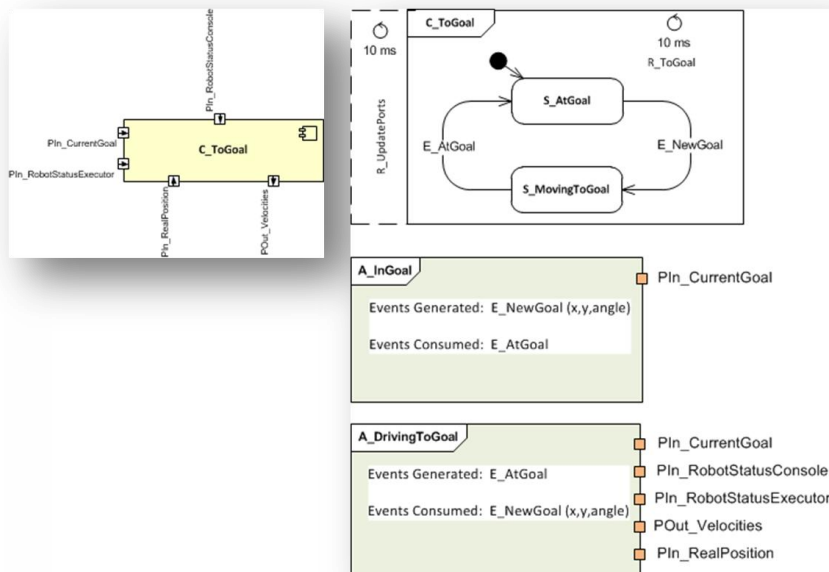


Fig. 129 - Structure of the component C_ToGoal

6.2.2.1.9 C_Localization

This component corresponds to the MinFr implementation of the Localization component mentioned on chapter 5.

The structure of this component, in terms of ports, regions, states, events and transitions, is shown on the image below, Fig. 130.

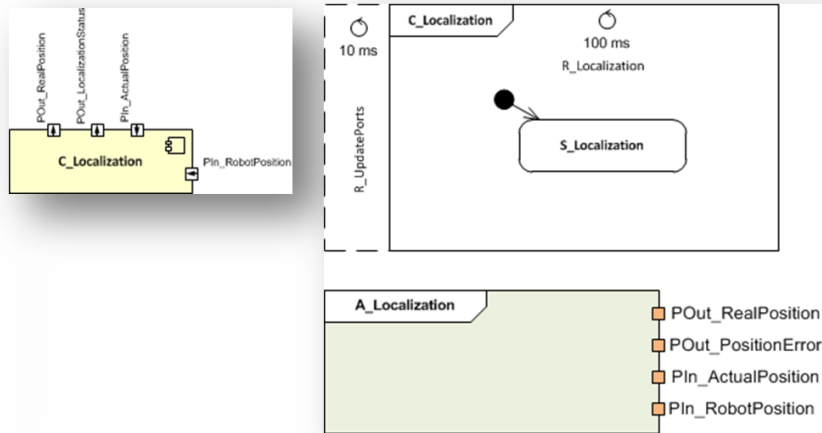


Fig. 130 - Structure of the component `C_Localization`

6.2.2.2 Application Structure

Like mentioned before, it is on the application structure where it is defined the structure of the application, in terms of the connections between the ports of the components.

The application structure is defined on a `.minfr` file on the folder `MinFrProjects`.

To create the application structure file, it is needed to do the follow the procedure mentioned on the section 6.2.1.2.

Here it is not explained the implementation of this application structure file, since it follows the same basic concepts explained on the detail on section 6.2.1.2 for the application structure file of the tutorial application.

The structure of this application, in terms of connections between the ports of the components is shown on Fig. 121.

6.2.2.3 Application Distribution

Like mentioned before, it is on the application distribution where it is defined the distribution of the all the regions of all the components used by the application into threads.

The application distribution is defined on a `.minfr` file on the folder `MinFrProjects`.

To create the application distribution file, it is needed to do the follow the procedure mentioned on the section 6.2.1.3.

Here it is not explained the implementation of this application distribution file, since it follows the same basic concepts explained on the detail on section 6.2.1.3 for the application distribution file of the tutorial application.

The structure of this application, in terms of distribution of all the regions of all the components used by the application in threads is shown on the image below, Fig. 131.

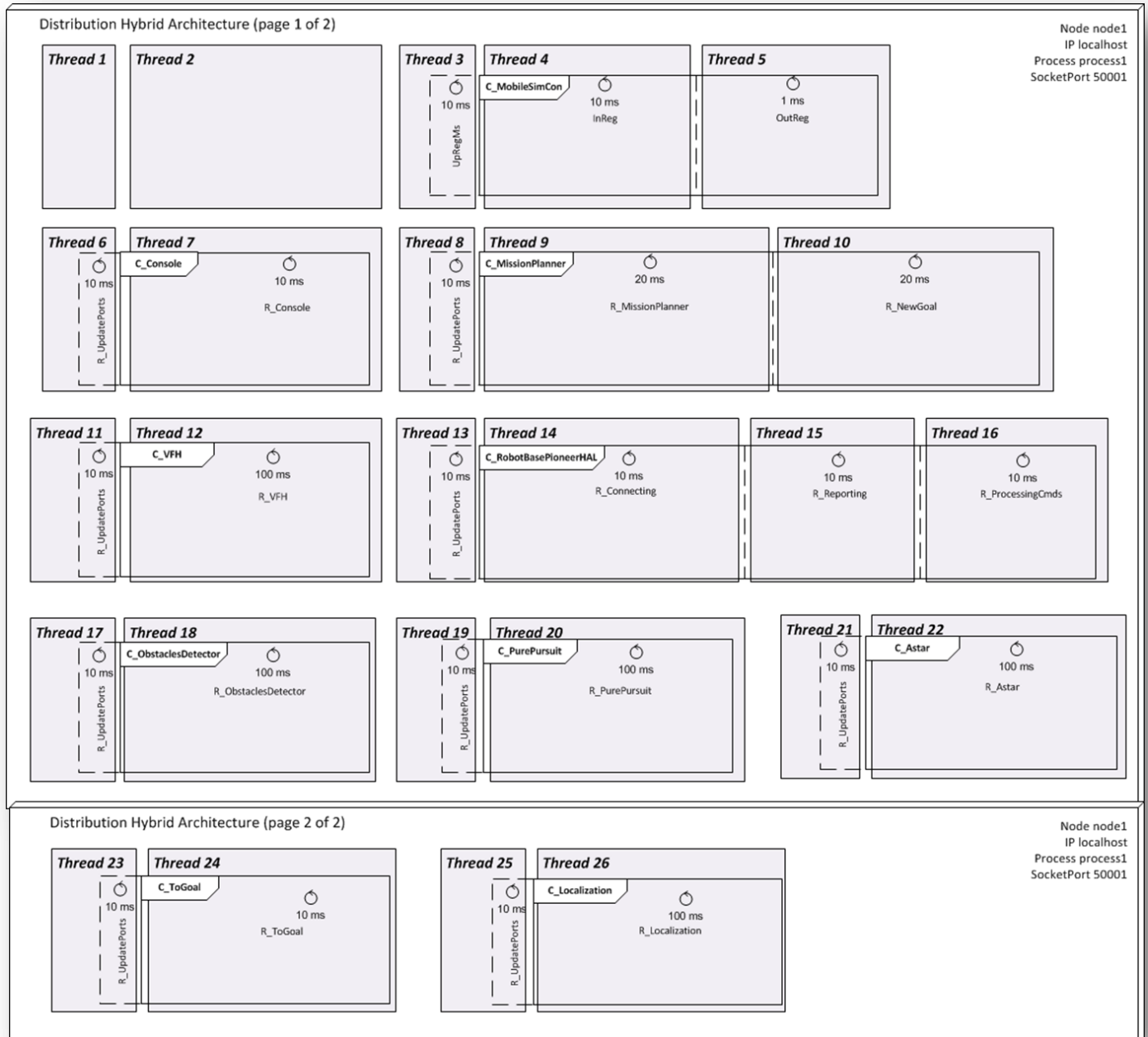


Fig. 131 - Distribution of the hybrid application

6.2.2.4 Activities

Like was mentioned before, each component activity corresponds to the place where it is defined the cyclic procedure that is executed while the correspondent state is active.

For each component, it is needed to have a C++ project, where it is implemented all its activities procedures.

To create each component C++ project and to generate each component activities files, it is needed to do the follow the procedures mentioned on the section 6.2.1.4.

On this section it will not be explained the implementation of the components activities but only their main procedure, since their implementation follow the same basic concepts explained on section 6.2.1.4 for the tutorial application components activities.

6.2.2.4.1 C_Console

According to section 6.2.2.1.1, the component *C_Console* has one region, *R_Console*, with two states, *S_Console* and *S_WaitingNewGoalValidation*.

The procedure of the state *S_Console* activity consists on the user interface menu, which menu and its options are shown below.

```
cout << endl << "                *** MENU ***                " << endl << endl;
cout << " > 1 - Add a mission to the missions queue." << endl;
cout << " > 2 - Check the actual position of the robot." << endl;
cout << " > 3 - Insert manually the actual position of the robot." << endl;
cout << " > 4 - Emergency stop." << endl;
cout << " > 0 - Exit the program." << endl;
cout << endl << " > Choose one option: ";
```

The procedure of the state *S_WaitingNewGoalValidation* activity consist on waiting for the new missions validation result and prints it on the screen.

Everytime a new mission is inserted by the user, the state *S_Console* activity generates an event *E_NewGoalSent*, which changes from the state *S_Console* to the state *S_WaitingNewGoalValidation*, in order for the state *S_WaitingNewGoalValidation* activity to wait until it receives the validation result of that new mission.

When the validation result of the new mission is received, the state *S_WaitingNewGoalValidation* activity prints it on the screen and generates an event *E_NewGoalValidationReceived*, in order to change back from the state *S_WaitingNewGoalValidation* to the state *S_Console*.

6.2.2.4.2 C_MissionPlanner

According to section 6.2.2.1.2, the component *C_MissionPlanner* has two regions, *R_MissionPlanner* and *R_NewGoal*, where each of them has one state, *S_MissionPlanner* and *S_NewGoal*, respectively.

The procedure of the state *S_MissionPlanner* activity consists on sending the next mission on the missions queue to the component **C_Executor**, once the previous one that was sent is completed and if there are more to do.

The procedure of the state *S_NewGoal* activity consists on validating or not the missions received from the component **C_Console**, depending if the mission goal position is valid or not, stores the validated missions on the queue, and sends that validation result to the component **C_Console**.

6.2.2.4.3 C_Executor

According to section 6.2.2.1.4, the component *C_Executor* has one region, *R_Executor*, with four states, *S_Executor*, *S_WaitingAnswerObstaclesDetector*, *S_WaitingAnswerAstar* and *S_WaitingAnswerVFH*.

The state *S_Executor* is the decision-making state of this component. Its activity procedure consists on receiving new missions from the component **C_MissionPlanner**, and decide which type of navigation algorithm to use, **C_Astar** or **C_VFH**, according to the matching result between the laser readings and the map of the environment, which is calculated by the component **C_ObstaclesDetector**.

While the robot is not at the mission goal position, this activity asks the component **C_ObstaclesDetector** if new obstacles are detected or not.

To do so, it sends a message to the component **C_ObstaclesDetector**, in order for it to check if new obstacles are detected or not, and then it generates an event *E_SentRequestOD*, which is to be consumed by the state *S_WaitingAnswerObstaclesDetector*.

This state, *S_Executor*, will only become active again when the state *S_WaitingAnswerObstaclesDetector* generates an event *E_ReceivedAnswerOD*, with the answer to the request made to the component **C_ObstaclesDetector**.

Once this state becomes active again by receiving the answer to that request, one out of two things can happen:

- If the answer received is that no new obstacles were detected, this activity, *A_Executor*, asks the component **C_Astar** to calculate the shortest path from the robot current position to the mission goal position.

To do so, it sends a message to the component **C_Astar**, in order for it to calculate the shortest path from the robot current position to the mission goal position, and then it generates an event *E_SentRequestAstar*, which is to be consumed by the state *S_WaitingAnswerAstar*.

This state, *S_Executor*, will only become active again when the state *S_WaitingAnswerAstar* generates an event *E_ReceivedAnswerAstar*, with the result of the calculation made by the component **C_Astar**.

- If the answer received is that new obstacles were detected, this activity, *A_Executor*, asks the component **C_VFH** to calculate a path from the robot current position to the mission goal position by avoiding all the obstacles on the way.

To do so, it sends a message to the component **C_VFH**, in order for it to calculate a path from the robot current position to the mission goal position by avoiding all the obstacles on the way, and then it generates an event *E_SentRequestVFH*, which is to be consumed by the state *S_WaitingAnswerVFH*.

This state, *S_Executor*, will only become active again when the state *S_WaitingAnswerVFH* generates an event *E_ReceivedAnswerVFH*, with the result of the calculation made by the component **C_VFH**.

Also, while the robot is not at the mission goal position, this activity checks if the localization error factors changed, by checking the values received at the port *PIn_LocalizationStatus*.

If they changed according to the previous ones received, this activity sends a message to the component **C_ToGoal**, in order for it to stop the robot, and then, it recalculates the path to the mission goal position by considering the new robot position.

After receiving a path from the components **C_Astar** or **C_VFH**, this activity sends a message to the component **C_ToGoal**, in order for it to allow the robot to move, and then, it sends that path to the component **C_PurePursuit**, for it to be processed.

The procedure of the state *S_WaitingAnswerObstaclesDetector* activity consists on waiting for a message from the component **C_ObstaclesDetector**, with the result of the matching between the laser readings and the map of the environment, which was requested previously by the state *S_Executor*.

After receiving that message, this activity generates an event *E_ReceivedAnswerOD* with that result, which is to be consumed by the state *S_Executor*.

The procedure of the state *S_WaitingAnswerAstar* activity consists on waiting for a message from the component **C_Astar**, with the result of the calculation of the shortest path from the robot current position to the mission goal position, which was requested previously by the state *S_Executor*.

After receiving that message, this activity generates an event *E_ReceivedAnswerAstar* with that result, which is to be consumed by the state *S_Executor*.

The procedure of the state *S_WaitingAnswerVFH* activity consists on waiting for a message from the component **C_VFH**, with the result of the calculation of a path from the robot current position to the mission goal position by avoiding all the obstacles on the way, which was requested previously by the state *S_Executor*.

After receiving that message, this activity generates an event *E_ReceivedAnswerVFH* with that result, which is to be consumed by the state *S_Executor*.

6.2.2.4.4 C_ObstaclesDetector

According to section 6.2.2.1.3, the component *C_ObstaclesDetector* has one region, *R_ObstaclesDetector*, with one state, *S_ObstaclesDetector*.

The procedure of the state *S_ObstaclesDetector* activity starts by waiting a message from the component **C_Executor**, with a request to check if new obstacles are detected or not.

After receiving that message, this state matches the laser readings with the map of the environment by using some methods of the implemented library *MapOperator*, and sends a message to the component **C_Executor** with the result of that matching, which can be new obstacles detected or not.

6.2.2.4.5 C_Astar

According to section 6.2.2.1.5, the component *C_Astar* has one region, *R_Astar*, with one state, *S_Astar*.

The procedure of the state *S_Astar* activity starts by waiting a message from the component **C_Executor**, with a request to calculate the shortest path between the robot current position and the mission goal position.

After receiving that message, this state calculates the shortest path between those two positions, by using the map of the environment and some methods of the implemented library *Astar*, and sends a message to the component **C_Executor** with the result of that calculation, which can be the path in the case that both positions are valid and if it is possible to arrive from one to another.

6.2.2.4.6 C_VFH

According to section 6.2.2.1.6, the component *C_VFH* has one region, *R_VFH*, with one state, *S_VFH*.

The procedure of the state *S_VFH* activity starts by waiting a message from the component **C_Executor**, with a request to calculate a path between the robot current position and the mission goal position by avoiding all the obstacles on the way.

After receiving that message, this state calculates a path between those two positions, by using the current laser readings and some methods of the implemented library *VFH*, and sends a message to the component **C_Executor** with the result of that calculation, which can be a path to the mission goal position or a path to another position on the environment, which is on the way to the mission goal position.

6.2.2.4.7 C_PurePursuit

According to section 6.2.2.1.7, the component *C_PurePursuit* has one region, *R_PurePursuit*, with one state, *S_PurePursuit*.

The procedure of the state *S_PurePursuit* activity starts by waiting a message from the component **C_Executor**, with a path to be executed.

After receiving that message, this state activity calculates the next position on the path that the robot must go to, by using the robot current position and some methods of the implemented library *PurePursuit*, and sends a message to the component **C_ToGoal** with that calculated next position.

If a new path is received while one is being executed, this state ignores the current one and starts executing the new one.

6.2.2.4.8 C_ToGoal

According to section 6.2.2.1.8, the component *C_ToGoal* has one region, *R_ToGoal*, with two states, *S_AtGoal* and *S_MovingToGoal*.

The procedures of these two states activities is very similar to the ones described on the section 6.2.1.4.2, with the only difference that here the state *S_MovingToGoal* only calculates the robot velocities when it is allowed by both components **C_Console** and **C_Executor**.

6.2.2.4.9 C_Localization

According to section 6.2.2.1.9, the component *C_Localization* has one region, *R_Localization*, with one state, *S_Localization*.

The procedure of the state *S_Localization* activity consists on calculating the real robot current position, which is done by adding the current position coordinates error factors to the robot current position coordinates sent by the encoders.

The position coordinates error factors are calculated when the user inserts the robot current position through the component **C_Console**. The calculation procedure of the position coordinates error factors consists on subtracting the robot current position inserted by the user to the robot current position sent by the encoders.

6.2.2.5 How to run the application

The situation used as test for this application consists of a room with three tables and a rubbish bin, where all this data is known a priori by the robot, by mentioning it on the file *MapData.txt*.

In order to check if this application works correctly, three new obstacles are added to the room, but not told to the robot, being then unknown and needing the robot to detect them with the laser.

As test for this application, a mission is given to the robot, in order for it to go to a goal position, which was of easy access before adding the unknown obstacles, but that now, after adding them, became of more difficult access, needing then the robot to detect them and calculate the paths to the goal position by taking them into account.

The room situation known a priori by the robot is shown on the image below, Fig. 132.

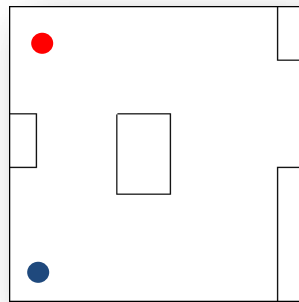


Fig. 132 - Room situation known by the robot

The room situation with the unknown obstacles is shown on the image below, Fig. 133.

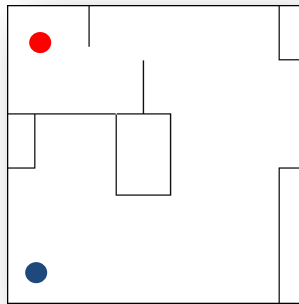


Fig. 133 - Real room situation

On both images, the blue point corresponds to the robot initial position, and the red point to the mission goal position.

The procedure to run this application is very similar to the one explained for the tutorial application on section 6.2.1.5, with the difference that the folder and files used are not the ones of the tutorial application, but the ones of this application.

With this, the procedure to run this application is the following one:

- Go to the folder *MinFrApps* and create inside it a folder with the application name, *Hybrid_Architecture*, in order to save there all the application executable files;

- Copy the *.so* files of each component used by the application from the folder *dist* that is inside of each correspondent component C++ project, to the folder that was just now created (*MinFrApps/ Hybrid_Architecture*).
- Copy also to the same folder, the *MinFr* server executable file and its respective libraries, as well as, the server executable file that establishes the connection between the application with the simulator;
 - The *MinFr* server executable file is named *minfrhost* and its respective libraries are:
 - *libminfr.so*;
 - *libtbb.so*;
 - *libtbb.so.2*.
 - The server executable file that establishes the connection between the application with the simulator is named *servidor_hmi_2*.

(The files mentioned on this last step can be copied from other application folder that is inside the folder *MinFrApps*.)

Now that the application is ready to be run, in order to run it, it is needed to do the following procedure, which is divided in three phases:

1st phase:

- Open a new terminal window, which will be called as *terminal1* during this explanation;
- Through *terminal1*, go to the *Hybrid_Architecture* folder that is inside the folder *MinFrApps*, and insert the following code line, in order to run the simulator:

```
MobileSim &
```

After inserting this code line and pressing *Enter*, a window will appear asking if you want to load a map for the simulation or not.

Here, go the folder *MinFrApps* (or the folder where you have stored the map), select the map *RoomWithUnknownObstacles.map* and press the button *Load Map*.

- After appearing the simulator window, go back to *terminal1*, press *Enter*, insert the following code line, in order to run the server that establishes the connection between the application and the simulator, and press *Enter*:

```
./servidor_hmi_2
```

By now, *terminal1* and the simulator should look like the image below, Fig. 134.

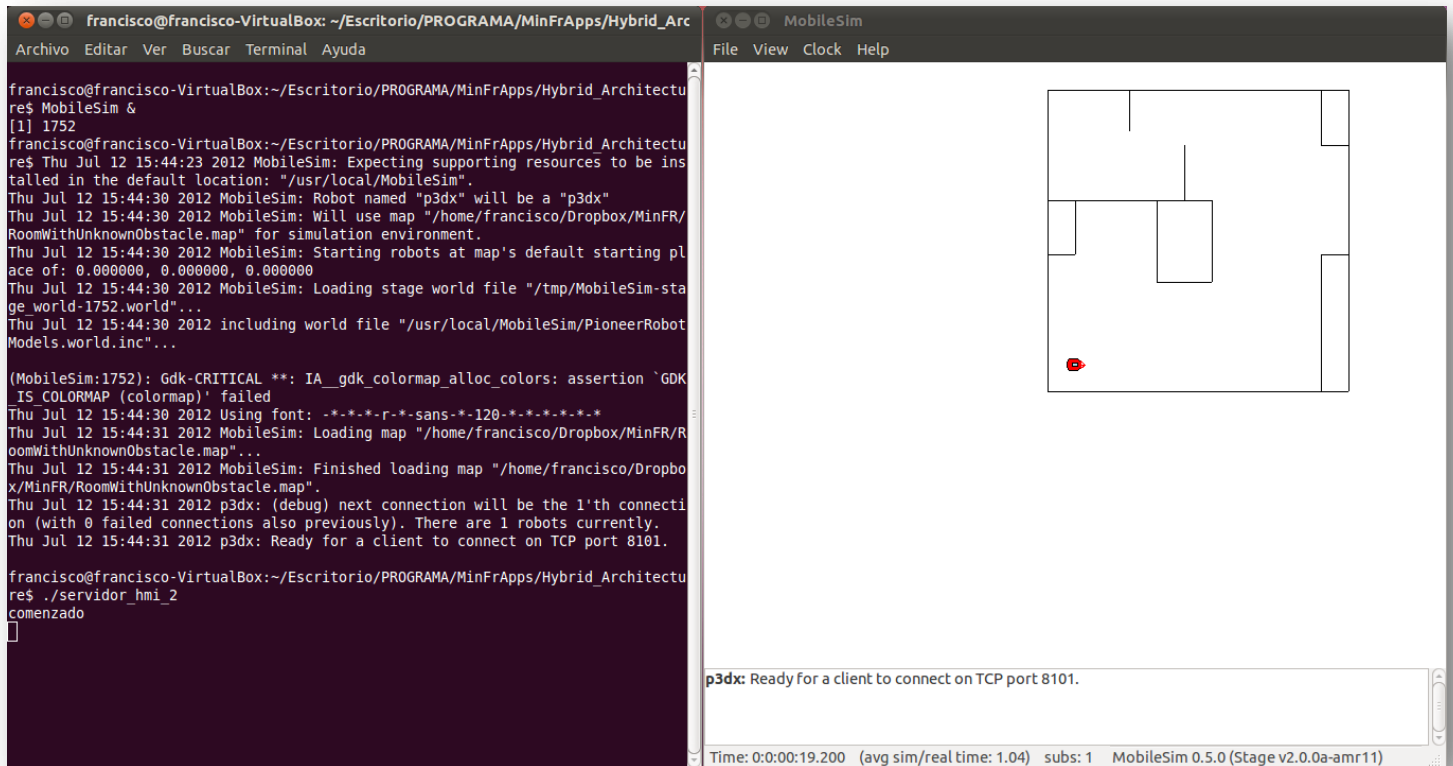


Fig. 134 - *terminal1* and simulator window after phase 1

2nd phase:

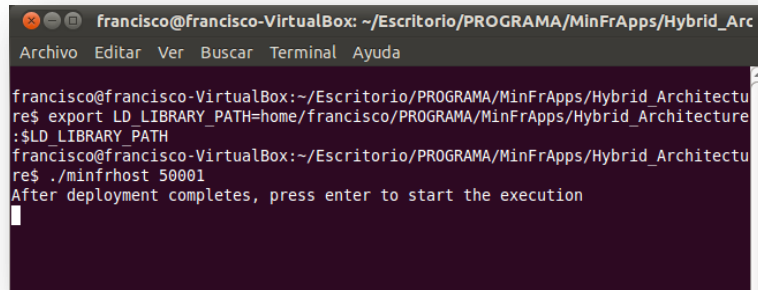
- Open a new terminal window, which will be called as *terminal2* for this explanation;
- Through *terminal2*, go to the *Hybrid_Architecture* folder that is inside the folder *MinFrApps*, insert the following code line, in order export the path of this folder as *LD_LIBRARY_PATH*, and press *Enter*:

```
export LD_LIBRARY_PATH=home/francisco/PROGRAMA/MinFrApps/Hybrid_Architecture:$LD_LIBRARY_PATH
```

- Also on *terminal2*, insert the following code line, in order to run the *MinFr* server executable file, and press *Enter*:

```
./minfrhost 50001
```

By now *terminal2* should look like the image below, Fig. 135.



```
francisco@francisco-VirtualBox: ~/Escritorio/PROGRAMA/MinFrApps/Hybrid_Arc
Archivo Editar Ver Buscar Terminal Ayuda

francisco@francisco-VirtualBox:~/Escritorio/PROGRAMA/MinFrApps/Hybrid_Architectu
re$ export LD_LIBRARY_PATH=home/francisco/PROGRAMA/MinFrApps/Hybrid_Architecture
:$LD_LIBRARY_PATH
francisco@francisco-VirtualBox:~/Escritorio/PROGRAMA/MinFrApps/Hybrid_Architectu
re$ ./minfrhost 50001
After deployment completes, press enter to start the execution
█
```

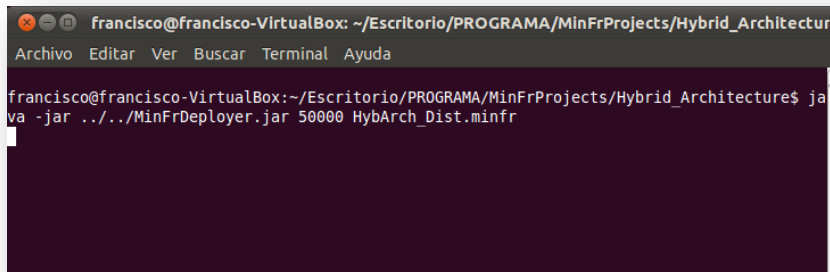
Fig. 135 - terminal2 after phase 2

3rd phase:

- Open a new terminal window, which will be called as *terminal3* for this explanation;
- Through *terminal3*, go to the *Hybrid_Architecture* folder that is inside the folder *MinFrProjects*, and insert the following code line, in order to deploy the application distribution:

```
java -jar ../../MinFrDeployer.jar 50000 HybArch_Dist.minfr
```

By now *terminal3* should look like the image below, Fig. 136.



```
francisco@francisco-VirtualBox: ~/Escritorio/PROGRAMA/MinFrProjects/Hybrid_Architectur
Archivo Editar Ver Buscar Terminal Ayuda

francisco@francisco-VirtualBox:~/Escritorio/PROGRAMA/MinFrProjects/Hybrid_Architectur
e$ java -jar ../../MinFrDeployer.jar 50000 HybArch_Dist.minfr
```

Fig. 136 - terminal3 after phase 3

- Wait until appears *Deployment complete* on *terminal3*, and when it does, go back to *terminal2* and press *Enter* , in order to run the application.

By now *terminal3* and *terminal2* should look like the images below, Fig. 137 and 138, respectively.


```

francisco@francisco-VirtualBox: ~/Escritorio/PROGRAMA/MinFrProjects/Hybrid_Architectur
Archivo Editar Ver Buscar Terminal Ayuda
Label Name="AckCode" DataType="String">63</Label><Label Name="UniqueRegionId" DataType="St
ring">1001001</Label></Msg>
Response: OK
Destination: process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">3</Label><
Label Name="AckCode" DataType="String">64</Label><Label Name="UniqueRegionId" DataType="St
ring">1001000</Label></Msg>
Response: OK
Destination: process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">2</Label><
Label Name="AckCode" DataType="String">65</Label><Label Name="UniqueRegionId" DataType="St
ring">1000001</Label></Msg>
Response: OK
Destination: process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">1</Label><
Label Name="AckCode" DataType="String">66</Label><Label Name="UniqueRegionId" DataType="St
ring">1000000</Label></Msg>
Response: OK
Destination: process1
Message: <Msg Type="RegionToThreadMsg"><Label Name="ThreadId" DataType="String">0</Label><
Label Name="AckCode" DataType="String">67</Label><Label Name="UniqueRegionId" DataType="St
ring">1004001</Label></Msg>
Response: OK
Deployment complete
francisco@francisco-VirtualBox: ~/Escritorio/PROGRAMA/MinFrProjects/Hybrid_Architectur

```

Fig. 137 - terminal3 when the deployment is complete

```

francisco@francisco-VirtualBox: ~/PROGRAMA/MinFrApps/Hybrid_Architecture
Archivo Editar Ver Buscar Terminal Ayuda
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Hybrid_Architecture$ export
LD_LIBRARY_PATH=home/francisco/PROGRAMA/MinFrApps/Hybrid_Architecture:$LD_LIBRARY
PATH
francisco@francisco-VirtualBox:~/PROGRAMA/MinFrApps/Hybrid_Architecture$ ./minfr
host 50001
After deployment completes, press enter to start the execution
Realizada la primera conexion a 127.0.0.1 : 8102
error on select(): 9 Bad file descriptor

Process Started

*** MENU ***

> 1 - Add a mission to the missions queue.
> 2 - Check the actual position of the robot.
> 3 - Insert manually the actual position of the robot.
> 4 - Emergency stop.
> 0 - Exit the program.

> Choose one option: Realizada la primera conexion a 127.0.0.1 : 8103

```

Fig. 138 - terminal2 after pressing Enter, when the deployment is complete

Now that the application is running, in order to execute a mission with the robot, choose option 1, and insert the coordinates of the mission goal in millimeters and degrees on *terminal2*, which is the terminal that it is used, when it is needed to interact with the application.

On the image below, Fig. 139, it is shown how to execute the situation explained on the beginning of this section, which was to order the robot to go to mission goal position that before adding the unknown obstacles was of easy access , but that now, after adding them, became of more difficult access, needing then the robot to detect them and calculate the path to the goal position by taking them into account. The mission goal position corresponds to the position 0, 9000 with a final orientation of 0°.

In order to verify if the robot really arrived at the correct final position, the menu option 2 can be used, which prints on the screen the real robot position.

The red trail corresponds to the trajectory that the robot took from its initial position 0, 0 to the final position 0, 9000.

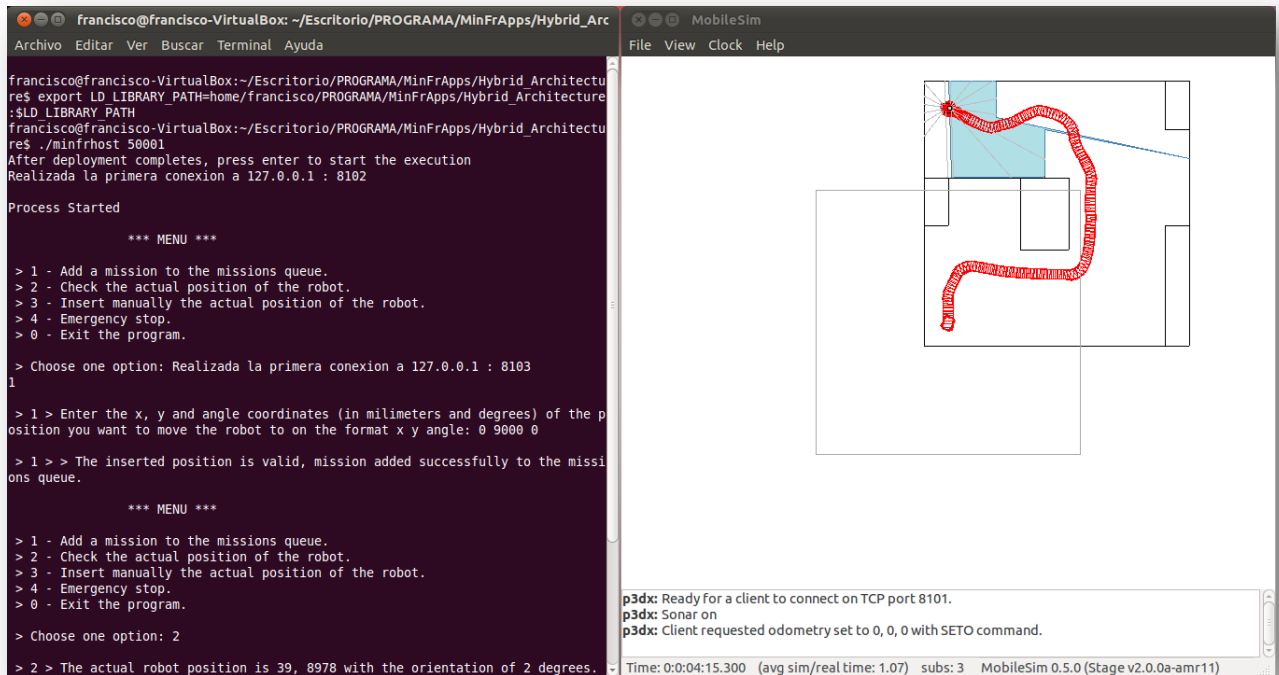


Fig. 139 - Application running with the map with the unknown obstacles (on the left, terminal2, which is the terminal used to interact with the application, and on the right, the simulator)

In order to show that without the unknown obstacles the path taken by the robot would be shorter, it is shown on the image below, Fig.140, the execution of the same mission but by using the room map that doesn't include the unknown obstacles, *Room.map*.

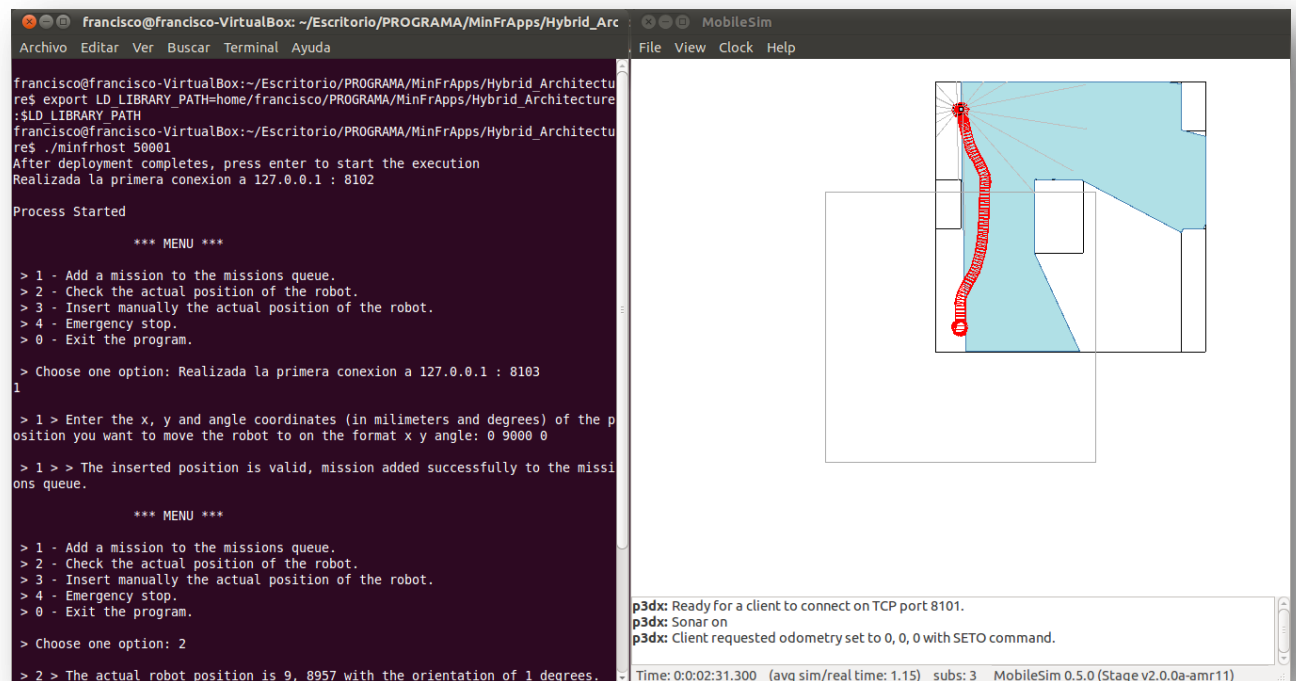


Fig. 140 - Application running with the map without the unknown obstacles (on the left, terminal2, which is the terminal used to interact with the application, and on the right, the simulator)

7 Conclusions and Future Work

As main conclusion for this project, it must be referred that all the defined objectives were achieved, which makes of this project a very complete one on the field of robotics.

Such conclusion is taken due to the fact that through the different chapters of this project, it is clearly described what is an autonomous mobile robot (AMR), its components and the way they can interact between themselves, as well as, how to create and run applications for an AMR on two different robotic frameworks.

Considering all that is mentioned on this project, there are two parts that have more importance than all the others, which are, the theoretical foundation about autonomous mobile robots, chapter 2, and the implementation and design of the hybrid application, chapters 5 and 6, and.

The reason for giving such importance to chapter 2 is the extensive work that was made in terms of bibliography, since the purpose was to define a clear and simple classification method for the different components of an autonomous mobile robot, which would be based on some existing classification methods so that, no new concepts would be created.

About chapters 5 and 6, the reason for their importance is more obvious, since due to the design and implementation of the hybrid application on the two different frameworks, it was gotten a good example of how to implement an application that allows the robot to move in an autonomous way, with both frameworks.

During the implementation part of this project it was detected some problems and limitations on both frameworks.

About Smartsoft, the only problem that was detected is related with the deployment of the application. In specific, the problem that occurred was that after deploying the deployment diagram, when it came up to run the application, some components were trying to connect to themselves, instead of connecting to the components specified on the deployment diagram.

The solution found for this problem is efficient but takes some time, since it consists on going through the components *.ini* files that are inside the deployment folder, and edit them one by one with the correct connections.

About MinFr, the only problem/limitation that was found, was in terms of the number of transitions and events a region can have, since during the initial implementation of the hybrid application, the executor component had 7 states and 11 associated transitions, and when it came up to run it, some transitions were not executed.

Since until that time, a solution for the problem was not found, the adopted solution for this case was to reduce the number of states of that component to 5 and the transitions to 9.

About the overall performance of the two frameworks, both work good, but have some limitations, which is explained with the fact of both being still in development.

As disadvantages of Smartsoft there is the graphic tool used to build diagrams, which is not of easy use, and can be the reason behind the problem mentioned previously.

As disadvantages of MinFr there is the fact of not including a graphic tool to build diagrams, which can also be considered as an advantage when compared with Smartsoft, since it is easier to define the components and applications structure on MinFr than on Smartsoft, and also the fact of only having one communication pattern, the communication pattern send.

Another disadvantage of MinFr is what it takes to run an application, since it is needed to copy the correspondent component files to the application location every time some activity is changed, and then, work with 3 terminals at the same time.

About future works that can follow what was done here, there are some possibilities such as, development of different robotic applications with the same frameworks or

implementation of better algorithms and components that replace the ones implemented on this project.

In terms of implementation of algorithms, there are two in specific that were thought as possibilities for this project, but that because of their complexity in terms of implementation were discarded. One of them is the Iterative Closest Point (ICP), which performs simultaneous localization and mapping (SLAM), and that would make the designed application a lot more efficient and robust in terms of localization, mapping and path planning.

The other algorithm is the D* algorithm, which is a very robust path planning algorithm that is able to perform both local and global navigation.

In terms of development of different robotic applications, a possible future work is to adapt the implemented application to other mobile robots, since the applications implemented on both frameworks were only tested on the robot Pioneer 3-AT.

Finally, as personal conclusion, I would like to mention that I am very proud of the final result of this project, since in the end, after 10 months of intensive work, it was possible to implement and run the hybrid application on both frameworks, which proved sometimes to be harder than expected due to the limitations of each framework but also due to my lack of knowledge on this field.

Also I must mention that thanks to this project, I obtained an enormous quantity of knowledge about robotics and programming, which in fact is the main goal of a project, to learn about what exists on the field, and develop according to the needs.

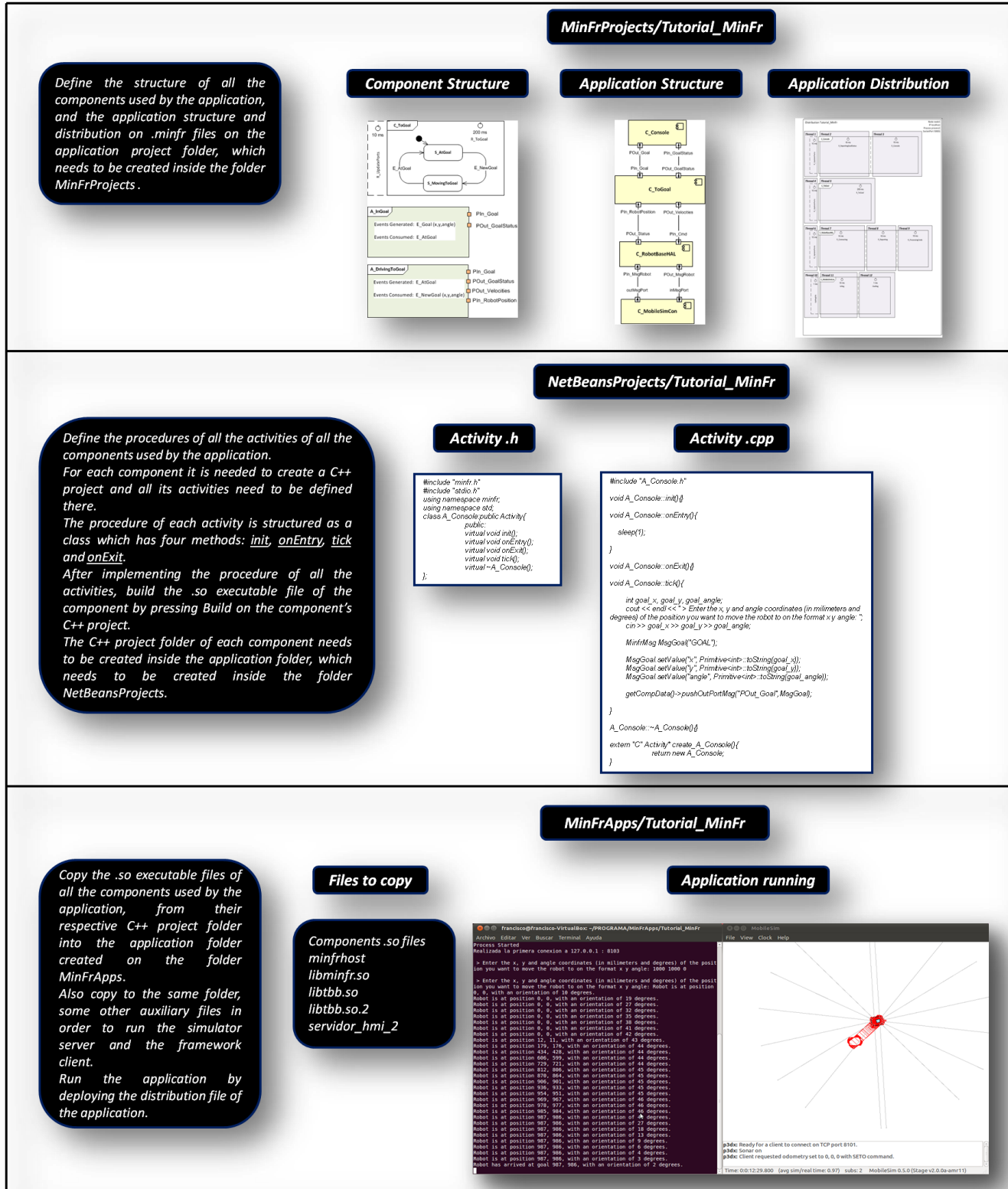
8 References

- [Ardil12] *THESIS*. Ardil González, José, “**Programación de un robot Pioneer utilizando el framework Smartsoft**”, Proyecto Fin de Carrera, Universidad Politécnica de Cartagena, 2012.
- [Borenstein91] *PAPER*. Borenstein, J. and Koren, Y., “**The Vector Field Histogram-Fast Obstacle Avoidance for Mobile Robots**”, IEEE Transactions on Robotics and Automation, Vol. 7, No. 3, June 1991.
- [Borenstein96] *BOOK*. Borenstein, J.; Everett, H. R. and Feng, L., “**Where am I? Sensors and Methods for Mobile Robot Positioning**”, The University of Michigan, 1996.
- [Brooks05] *PAPER*. Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S. and Orebäck, A., “**Towards Component-Based Robotics**”, In proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Canada, 2005.
- [Cañas08] *COURSE*. Cañas J.M., “**Apuntes de Robótica Móvil**”, Máster oficial en Sistemas Electrónicos Avanzados. Sistemas Inteligentes. Universidad Rey Juan Carlos, Alcalá, 2008.
- [Coulter92] *PAPER*. Coulter, R. C., “**Implementation of the Pure Pursuit Path 'Planning Algorithm**”, Carnegie Mellon University, January 1992.
- [Crowley] *PAPER*. Crowley, J.L. and Demazeau, Y., “**Principles and Techniques for Sensor Data Fusion**”, LIFIA(IMAG), (<http://www-prima.inrialpes.fr/jlc/papers/SigProc-Fusion.pdf>).
- [Dudek00] *BOOK*. Dudek, G. & Jenkin, M., “**Computational Principles of Mobile Robotics**”, Cambridge University Press, 2000.
- [Fox97] *PAPER*. Fox, D., Burgard, W. and Thrun, S., “**The Dynamic Window Approach to Collision Avoidance**”, IEEE Robotics and Automation, 1997.
- [Gamma95] *BOOK*. Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J., “**Design Patterns: Elements of Reusable Object-Oriented Software**”, Addison-Wesley Professional Computer Series, 1995.
- [Goodrich04] *PAPER*. Goodrich, M. A., “**Potential Fields Tutorial**”, (<http://students.cs.byu.edu/~cs470ta/goodrich/fall2004/lectures/Pfields.pdf>).
- [Goodwin08] *THESIS*. Goodwin, Richard. “**A unified design framework for mobile robot systems**”, PhD Thesis, Bristol Institute of Technology, University of the West of England, 2008.
- [Heineman01] *BOOK*. Heineman, G. T. and Councill, W. T., “**Component based software engineering: putting the pieces together**”, Addison-Wesley, 2001.
- [Jimenez09] *THESIS*. Jiménez, Ismael Martínez, “**Desarrollo de una aplicación de teleoperación para el robot Pioneer 3-AT**”, Proyecto Fin de Carrera, Universidad Politécnica de Cartagena, 2009.
- [Kam97] *PAPER*. Kam, M.; Zhu, X. and Kalata, P., “**Sensor Fusion for Mobile Robot Navigation**”, Proceedings of the IEEE, 1997.
- [LaValle06] *BOOK*. LaValle, Steve M., “**Planning Algorithms**”, Cambridge University Press, 2006.
- [Luo89] *PAPER*. Luo, R.C. and Kay, M.G., “**Multisensor Integration and Fusion in Intelligent Systems**”, IEEE Transaction on Systems, Man and Cybernetics, 1989.

- [Martínez01] *THESIS*. Martínez-Barberá, Humberto, “**Una Arquitectura Distribuida para el Control de Robots Autónomos Móviles**”, PhD Thesis, Dpto. Ingeniería de la Información y las Comunicaciones, Universidad de Murcia, 2001.
- [Murphy00] *BOOK*. Murphy, R., “**Introduction to AI Robotics**”, ISBN 0-262-13383-0, The MIT Press, 2000.
- [Ortiz08] *COURSE*. Ortiz, A. “**Control de Robots Móviles**”, Dpto. Matemáticas e Informàtica, Universitat de les Illes Balears, 2008.
- [Parker08] *COURSE*. Parker, Lynne E., “**CS494/594: Autonomous Mobile Robots**” . Dept. of Electrical Engineering and Computer Science, The University of Tennessee. Available on-line: <http://web.eecs.utk.edu/~parker/Courses/CS594-fall08/Schedule.html>, 2008.
- [Riisgaard05] *PAPER*. Riisgaard, S. and Blas, M. R., “**SLAM for DUMMIES**”, http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf, 2005.
- [Schlegel99] *PAPER*. Schlegel, C. and Wörz, R., “**Software framework SmartSoft for implementing sensorimotor systems**”, In proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 1610-1616, Korea, 1999.
- [Schlegel04] *THESIS*. Schlegel, C. “**Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach**”, PhD Thesis, Faculty of Computer Science, Univ. of Ulm, 2004.
- [Siegwart04] *BOOK*. Siegwart, R. and Nourbakhsh, I., “**Introduction to Autonomous Mobile Robots**”, ISBN 0-262-19502-X. Slides and exercises available on <http://www.mobilerobots.org>, The MIT press, 2004.
- [Stentz95] *PAPER*. Stentz, A., “**The Focused D* Algorithm for Real-time Replanning**”, In Proceedings of the International Joint Conference on Artificial Intelligence, August 1995.
- [Szyperski98] *BOOK*. Szyperski, C., “**Component Software - Beyond Object-Oriented Programming**”, Addison-Wesley Longman, 1998.

9 Appendices

9.1 Appendix A: Diagram of how to work with MinFr

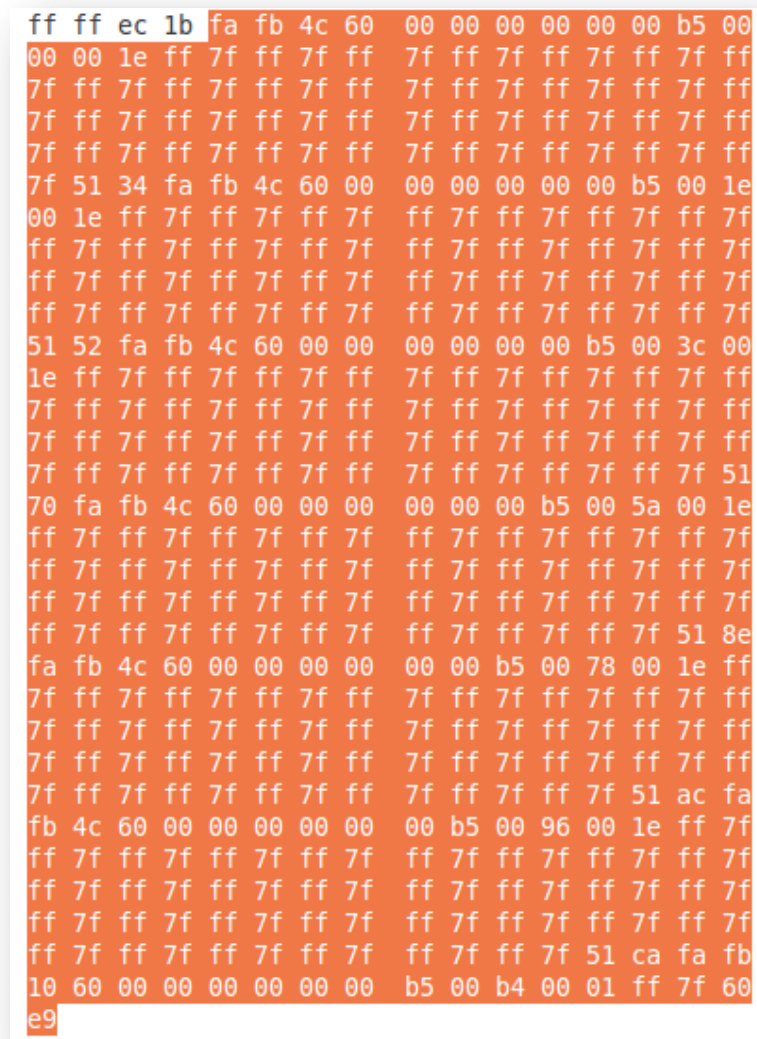


9.2 Appendix B: MobileSim laser packet

While implementing the hybrid application with MinFr, it was needed to change the code of the server that connects the simulator with the framework, *servidor_hmi_2*, since the laser packet that was sent by the simulator *MobileSim* was not being correctly processed at that server.

In order to know how to send it correctly, it was analyzed with the program *Wireshark*, which packets are sent from the simulator to the server with the laser information.

The data field of the laser packet sent by the simulator to the server *servidor_hmi_2* is shown on image below, Fig. 141.



ff	ff	ec	1b	fa	fb	4c	60	00	00	00	00	00	00	b5	00
00	00	1e	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	51	34	fa	fb	4c	60	00	00	00	00	00	b5	00	1e	
00	1e	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
51	52	fa	fb	4c	60	00	00	00	00	00	00	b5	00	3c	00
1e	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	51
70	fa	fb	4c	60	00	00	00	00	00	00	b5	00	5a	00	1e
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	51	8e
fa	fb	4c	60	00	00	00	00	00	00	b5	00	78	00	1e	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff
7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	51	ac	fa
fb	4c	60	00	00	00	00	00	00	b5	00	96	00	1e	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f
ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	ff	7f	51	ca	fa	fb
10	60	00	00	00	00	00	00	b5	00	b4	00	01	ff	7f	60
e9															

Fig. 141 - MobileSim laser packet

As it can be seen, the simulator sends the data field divided into sub packets, where each sub packet can only have in the maximum, 30 laser readings. Each sub packet has two parts: header and data.

The header of each sub packet is formed by 4 bytes, where the first two bytes are always the hexadecimal numbers *fa* and *fb*, respectively, the third byte is the sub packet size and the fourth and last byte is the packet ID, on this case always 60.

fa fb 4c 60

Fig. 142 – MobileSim laser sub packet header

The data part of each sub packet is divided in two parts: the robot position coordinates and the laser readings.

The robot position coordinates corresponds to the 6 bytes that come after the sub packet header. The first 2 bytes of this part are the x coordinate, the second 2 bytes are the y coordinate and the last 2 bytes are the robot orientation in angles.

00 00 00 00 00 00

Fig. 143 - MobileSim laser sub packet robot position

The laser readings correspond to the rest of the sub packet. The first 2 bytes of this part are the number of laser readings that come on the whole laser packet, the second 2 bytes are the angle that it corresponds the first laser reading of this sub packet, the following byte corresponds to the number of readings that comes on this sub packet, and the following groups of two bytes are the laser readings.

b5 00 1e
00 1e ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f
ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f
ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f
ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f

Fig. 144 - MobileSim laser sub packet laser readings

On the end of each sub packet, there are two more bytes that corresponds to the CRC.

