EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mechanical Engineering
Control Systems Technology

4SC020 - Mobile Robot Control

Design Document

Supervisors:
dr. ir. M.J.G. van de Molengraft
ir. W. Houtman

Group 8:

Student name:	**Student number:**
J.J.W. Bevers	0904589
B.R. Herremans	0970880
J. Fölker	0952554
L.D. Nijland	0958546
A.J.C. Tiemessen	1026782
A.H.J. Waldus	0946642

Eindhoven, May 4, 2020

1 Requirements and Specifications

A distinction is made between general and competition specific requirements. Three stakeholders are considered: the hospital employees, patients and software engineers. In ascending order, the requirements are subdivided into environment requirements, border requirements, system requirements and function types. Five functions types are introduced: Localization(L), Detection(D), Mapping(M), Path Planning(PP) and Motion Control(MC). These are elaborated in the next section. The requirements are shown in Figure 1 and elaborated below. Note the legend located bottom right.

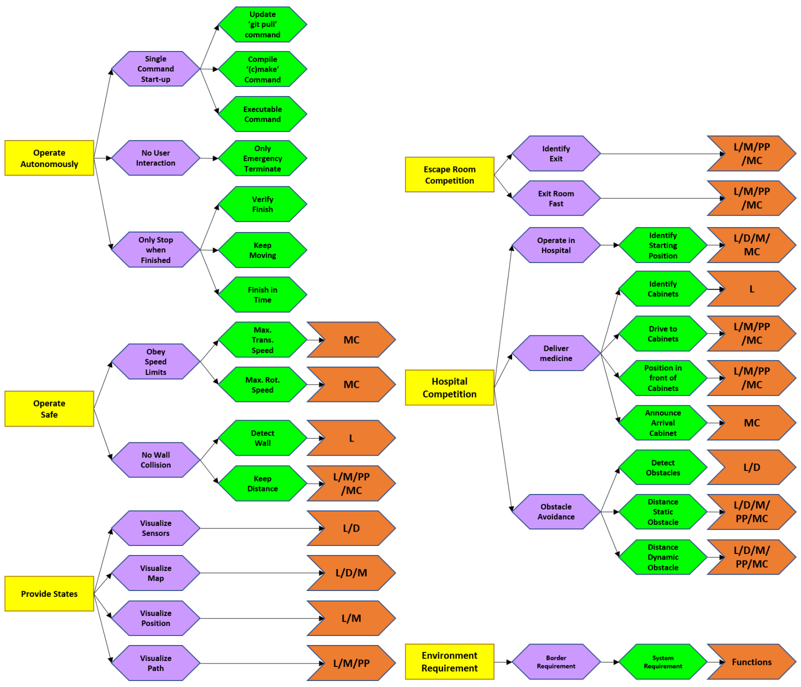

Figure 1: The general requirements (left), the competition specific requirements and legend (right).

PICO should operate autonomously. No interaction is allowed, unless an emergency stop is needed. To start the software, the 'git pull' command is used to update, the 'cmake' and 'make' commands are used for compiling and one executable is used to start execution. PICO can only stop executing when its task is finished. Therefore it must be able to verify when it is finished. Also, PICO cannot be stationary for 30 consecutive seconds and the task must be finished in time.

PICO should operate safely. In order to do so, it must obey speed limits. A maximum translational and rotational speed of 0.5 m/s and 1.2 rad/s, respectively, are set. PICO can not bump into walls. Therefore, PICO should be able to detect walls and keep a certain distance.

PICO should provide information on its state. Therefore, PICO should visualize its sensor data, produce and visualize a map and its current position and finally visualize its produced path.

PICO should finish the Escape Room Competition. To succeed, PICO must be able to identify the exit and it should exit as fast as possible. A strategy is explained in the next section.

PICO should finish the Hospital Competition. PICO should deliver medicine. Therefore, PICO must recognize cabinets, drive to them, position in front of them facing towards them and afterwards produce a sound signal. PICO should operate in the hospital environment. Therefore it must recognize its (initial) position in the provided map. Finally, PICO should not bump into static or dynamic objects. Thus PICO should detect an object within a 10 meter range and the distance between PICO and a static or dynamic object must be at least 0.2 m and 0.5 m respectively.

The hospital employees are involved in the first, second and fifth requirement, the patients in the second and fifth and the software engineers in the first, third and fourth.

2 Functions

The following finite state machine depicts the designed behaviour of PICO:

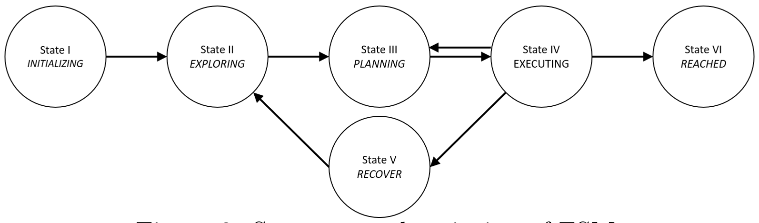

Figure 2: State space description of FSM.

State I - Initializing PICO performs an initial scan of it's surroundings and maps this data to find its position on the map. It is possible that insufficient information has been gathered to do this when, for instance, all objects are out of reach of the laser range finder.

State II - Exploring An exploration algorithm such as a potential field algorithm or a rapidly expanding random tree algorithm is used to explore the environment further. When PICO's position within the map is known and the exploration algorithm has been executed for T seconds, the transition to State III occurs.

State III - Planning After PICO has localized itself on the map, enough information has been gathered to compute a path towards a target. An algorithm is used to compute a state trajectory, that is then translated into control actions.

State IV - Executing During this state a control loop is executed which controls PICO towards the reference coordinates (x, y) by measuring position and actuating velocities. When PICO runs into an unexpected close proximity object, the state is transferred to State V. When PICO runs into an unexpected distant proximity object, the state is transferred to State III, such that a new path can be planned before PICO arrives at this object.

State V - Recover During the recover state, PICO has encountered a close proximity object which it did not expect. PICO then immediately stops and when completely stopped, it transits to State II, such that the exploration algorithm can move PICO away from this unexpected object for T seconds. The loop (State II, State III, State IV, State V) can be executed repeatedly and can be counted, such that T increases when more loops are encountered.

State VI - Reached After the path execution has been finished and PICO has reached the end of the path, PICO stops moving as it has reached its target location.

Table 1: Functions and their descriptions.

FUNCTION	**DESCRIPTION**
Localization	
InitialScan()	PICO makes a rotation of 190 [deg] to get a full 360 [deg] scan of the environment
GetMeasurement()	Read out the laser range finder sensor data
Detection	
ProximitySense()	Detects if an unexpected object is in close proximity
DistantSense()	Detects if an unexpected object is in distant proximity
Mapping	
UpdateMap()	Write measured data to Cartesian 2D world map used by path planner
UpdatePotField()	Write measured data into potential field map used by potential field algorithm
GetMap()	Retreive current 2D carthesian world map and (x, y) position of PICO
GetPotField()	Gets gradients from the potential field
Path Planning	
GetGoal()	Retrieve Cartesian coordinates of the goal (x_g, y_g) w.r.t. Map
PlanPath()	Path planner algorithm using the 2D world map to plan a state trajectory (x, y array)
ComputePotState()	Uses gradients from potential field as input, to compute a state reference $\dot{x}$, $\dot{y}$, $\dot{\theta}$
ReadPath()	Get current (x, y) goal of path array
Motion Control	
Move()	Send position reference (x, y, θ) to PICO. Contains position control loop.
StopMoving()	All movements of PICO are stopped: $\dot{x}$, $\dot{y}$, $\dot{\theta} = 0$
FullRotation()	PICO makes one full rotation
ExecutePath()	Translates (x, y) path reference into controller action for calling Move()
	Loop over all (x, y) to achieve position reference tracking

3 Components and Interfaces

The proposed information architecture is depicted in Figure 3. It shows a generic approach, so note that a specific challenge may require additional functionalities and interfaces. Between the software and mechanical components, information is shared via interfaces marked in green and labeled with an arbitrary number or letter. These interfaces dictate what information is set by which component and what information is shared, and amongst which components. In the central section, this schematic overview visualizes what capabilities are intended to be embedded in the software.

Firstly, **interface 10** represents the data received from the user, which may include a JSON file with a map and markers, depending on the challenge. On the right, **interface 1** between PICO (including the provided software layer) and the designed software shows the incoming sensor data, i.e. structs with distance data from the Laser Range Finder (LRF) and odometry data based on the omni-wheel encoders. This data is (possibly) processed and used to map PICO's environment (mapping), to estimate PICO's position and orientation (localization), and to detect static/dynamic obstacles (detection), via **interfaces 2, 3 and 5**. This knowledge is stored in the world model, serving as a central data-base. Via **interface 4**, a combination of the current map, PICO's pose and the current (user-provided) destination is sent, to plan an adequate path. This path is sent over **interface 8**, in the form of reference data for the (x, y, θ)-coordinates. The motion control component will then ensure that PICO follows this path by sending velocity inputs over **interface 9**. Simultaneously, the detection monitors any static/dynamics obstacles on the current path online. If this is the case, **interfaces 6 and 7** serve to feed this knowledge back to either the path planning component to adjust the path or directly to motion control to make a stop. On the left, **interface A** between the behavior scheme and the capabilities shows that the high-level finite state machine manages what functionality is to be executed and when. **Interfaces B** represent the data that is chosen to be visualized to support the design process, i.e. the visualization serves as a form of feedback for the designer. The information that is visualized depends heavily on the design choices later on in the process and, therefore, this will not be explained further.

3.1 Implementation

The components described above are intended to be explicitly embedded in the code through the use of classes. The functionality that belongs to a certain component will then be implemented as a method of that specific class. At this stage of the design process, the specific algorithms/methods that will be used for the functionalities are not set, but some possibilities will be mentioned here. Simultaneous mapping and localization (SLAM) algorithms, based on Kalman filters or particle filters, could be used for the mapping and localization components. Moreover, for path planning, Dijkstra's algorithm, the A* algorithm or the rapidly-exploring random tree algorithm can be used. For motion control, standard PI(D) control, possibly with feedforward, can be used to track the reference trajectory corresponding to the computed path. Additionally, for detection, artificial potential fields can be used to stay clear from obstacles.

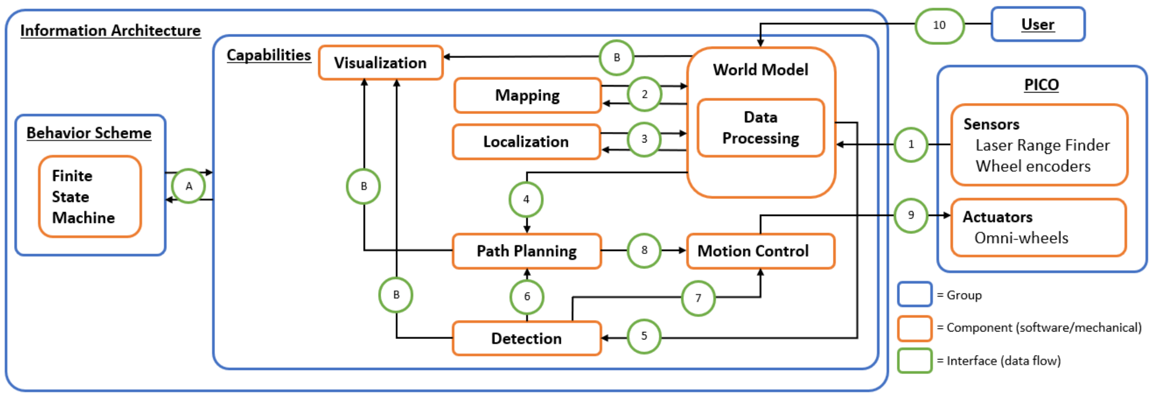

Figure 3: Schematic overview of the intended information architecture.