applied sciences

MDPI

Article

MERLIN a Cognitive Architecture for Service Robots

Miguel Á. González-Santamarta *, Francisco J. Rodríguez-Lera, Claudia Álvarez-Aparicio, Ángel M. Guerrero-Higueras and Camino Fernández-Llamas

Grupo de Robótica, Universidad de León, Campus Vegazana, s/n, 24007 León, Spain; fjrodl@unileon.es (F.J.R.-L.); calvaa@unileon.es (C.Á.-A.); am.guerrero@unileon.es (Á.M.G.-H.); camino.fernandez@unileon.es (C.F.-L.)

* Correspondence: mgons@unileon.es; Tel.: +34-987-291-743

Received: 31 July 2020; Accepted: 25 August 2020; Published: 29 August 2020

Abstract: Many social robots deployed in public spaces hide hybrid cognitive architectures for dealing with daily tasks. Mostly, two main blocks sustain these hybrid architectures for robot behavior generation: deliberative and behavioral-based mechanisms. Robot Operating System offers different solutions for implementing these blocks, however, some issues arise when both are released in the robot. This paper presents a software engineering approach for normalizing the process of integrating them and presenting them as a fully cognitive architecture named MERLIN. Providing implementation details and diagrams for established the architecture, this research tests empirically the proposed solution using a variation from the challenge defined in the SciRoc @home competition. The results validate the usability of our approach and show MERLIN as a hybrid architecture ready for short and long-term tasks, showing better results than using a by default approach, particularly when it is deployed in highly interactive scenarios.

Keywords: cognitive robotics; robot planning and scheduling; finite state-machines; hybrid architecture; robotics competition; object oriented programming

1. Introduction

Historically, a considerable number of approaches [1] have been developed for performing robot behaviors. Paradigms such as deliberative, subsumption, three layers, or reactive architectures are just some of them. These architectures aim to provide a framework for managing and controlling a robot in a responsible and timely manner. However, when these paradigms are translated to a software engineering level, their abstract components are converted into software nodes interacting asynchronously in different conditions of uncertainty for creating robot behaviors.

This research presents the endeavor for testing current off-the-shelf solutions of the community, generally presenting in public repositories like GitHub, for releasing functional robotics architectures for real service robots. Thus, this paper analyzes not only the overall problem that many robotics researchers want to solve, generating long-term robot behaviors in real behaviors, but also the problem of how it is developed and tested in state-of-the-art platforms.

In order to address the classic conceptual issues, every time that a researcher tries to provide a cognitive architecture, it is necessary to understand the research papers, fix GitHub public versions, missing information, etc. In this manner, this paper provides a formal approach and how to example that we hope would be of interest to the community. This approach is required because, in the real world, robot behavior generation does not rely solely on abstract concepts of cognitive architectures, it also depends on immediate software development, software adaptation, version normalization, component integration and test.

This research proposes the architecture description from conceptual and software implementation perspectives. Accordingly, there is a formal specification of the implementation using Unified Modeling Language (UML) [2] considering the functional properties of each layer of the cognitive architecture. Using a top-down approach, from the deliberative to the behavioral layer for presenting a hybrid approach.

1.1. Research Questions

There are many interesting research questions surrounding the field of social and assistive robots for daily life tasks. When the complexity of these scenarios increases, by adding non-stationary furniture, humans and interaction with them, it becomes necessary to define a set of constraints at all levels for managing task planning, behavioral approaches and robot capabilities such as perception and dialogue. Thus, this research presents different research questions:

1. What Are the Elements for Providing Robot Behaviors in the Long-Term?
2. How We Can Integrate State-of-the-Art Components, or in This Case, Off-the-Shelf Software Components, for Creating a Cognitive Architecture?
3. What Are the Mechanisms for Successfully Managing Expected or Unexpected Requests Associated with a Given Robot Duty?

1.2. Contributions

The main contribution of this paper is to analyze the software mechanisms for dealing with the key aspects related to the decision making process and robot behavior generator in a real-world scenario using well-known software solutions. Having in mind the development of a hybrid architecture using the most famous components in the Robot Operating System (ROS) : State MACHine library (SMACH) and ROSPlan framework, this research proposes the next contributions:

1. We have integrated a set of ROS Based components (ROSPlan and SMACH) for performing tasks in a real-world environment.
2. We have established the formal mechanisms for explaining the deployment of an applied cognitive architecture using UML formalism.
3. We have identified the gaps for extended operation and are included in the architecture that we have called MERLIN.
4. We have integrated and evaluated the impact of our extensions and compared against the original solutions available in public repositories.

The remainder of this paper is organized as follows. Section 2 overviews the current state of the art in architecture for robotics from a cognitive and software point of view. Section 3 presents the approach proposed in this research for integrating the components for working together as a cognitive architecture with Deliberative and Behavioral systems. Section 4 illustrates the validation of our approach with a robot. Finally, Discussion (Section 5) and Conclusions (Section 6) review our results and conclude with the findings and lessons learned obtained during the research.

2. Background

Assisting people in daily life tasks is a complex duty for a service or assistive robot. Several challenges need to be fulfilled in order to find a service robot in a real-world scenario:

- The complexity and dynamics of known or unknown real-world scenarios human-oriented.
- Long-term operating periods.
- Managing multiple requests, internal and external, associated with robot tasks and duties.

For years, several European Projects have developed cognitive architectures so as to handle these issues adequately [3]. Robotics history presents classic approaches for producing effective robot behaviors. From a deliberative perspective, focus on Planning Domain Definition Language (PDDL) [4], which is a family of languages for automated planning, and planning approaches [5] to a reactive perspective, presented by researchers as Arkin [6]; behavioral approaches such as the subsumption architecture proposed by Brooks [7]. Besides, the robot community also presents several efforts for solving issues of both worlds through full integration in a single approach, as Gat approaches for his own in [8] or with a team in [9].

Nowadays, current approaches go beyond these classical approaches and present the robot architecture from a software perspective [10] and its perspectives: model-based, real-time, reconfiguration, etc. Notwithstanding, it is complicated to translate previous classical cognitive models to state-of-the-art software. Kotseruba and Tsotsos [11] review most of the cognitive architectures developed and presented in the literature in the last 40 years, showing if they were tested or not, or if the architecture involved real or simulated sensor information. Their work highlights the large gap in getting closer architecture and implementation perspectives in your robot.

CORTEX presented by Bustos et al. [12], presents an architecture supported on three key ideas: modularity, internal modeling and graph representations. As a result, they define agents, which are computational modules that can represent any functionality in the reactive-deliberative spectrum. They build everything on top of Robocomp [13] which is considered a middleware for robots. Although the community on RoboComp is growing up and supported under the Google Summer of Code framework, they are a work in progress and it is necessary a big effort for novel researchers to use and deploy in any robot.

Behavior-based Iterative Component Architecture (BICA) [14] is an architecture supported on ethological principles whose root ideas, extracted from [15], were defined in [16] aiming at the behavior generation in any robot. We have been successfully using this approach in different scenarios such as competitions and Proofs-Of-Concept for generating hybrid architectures based on motivational principles [17]. Thus, for years, several members of the ROS community and research suggested us to target ROSPlan and SMACH solution. Although the ROSPlan is integrated into BICA, we have decided to evaluate an approximation using SMACH as a Finite State Machine (FSM) generation and visualization instead of BICA.

Finally, given the massive acceptance of Robot Operating System (ROS), which provides the mechanism for easily deploying software in real platforms, we decided to test their main components for behavior generation at the software level, ROSPlan [18] and SMACH [19]. They emerged in the field of robot behavior generation as off-the-shelf solutions for integrating planning or hierarchical behaviors in real robots. There are new alternatives, such as the State Machine Asynchronous C++ library (SMACC), which provides extended characteristics to SMACH but is oriented to C++ development. Given our experience, we decided to start using SMACH with ROSPlan.

Nevertheless, when integrating both ROSPlan and SMACH is a solution, there is a set of limitations that need to be fulfilled. Lima and Ventura identified in their work [20] some of the limitations that are also presented in this research. Although they presented a great overview and approaches, given the time to understand the solution from scratch, we decided to provide not only the solutions but also a UML view of the approach along with the evaluation of the robot performance in these circumstances. They evaluated mainly the planner and here we evaluate the final robot performance.

The motivation for using Unified Modelling Language instead of block diagrams [21] or Systems Modeling Language (SysML) [22] was the time spent understanding the ROSPlan and SMACH mechanism to extend their core functionalities from a programmatically perspective. This approach is extended in the literature [23,24] and used for formally designing architectures in deliverables for European projects [25].

3. Proposed Architecture

The tools, methodology and experiments performed in this research are presented in this section.

3.1. Core Libraries

This research uses the Robot Operative System (ROS) [26], which is the most popular distributed framework for developing robotic solutions and has become the de facto standard for robotic software development. ROS includes a large set of libraries for controlling robot sensors and actuators. These libraries provide the abstractions, control engines, and inter-process communications for generating all kind of robot behaviors. This approach defines the ROS "Graph" layer that defines seven core elements: Nodes; Master; Parameters and Parameter Server; Topics; Services; Messages; and Bags. These elements are defined and explained in depth in ROS *ros_comm* [27]. Here, we would like to highlight the two main elements presented in this paper:

- Nodes: ROS computation takes place in processes called Nodes that interchange information through the ROS communication system.
- Communications: formally, it includes the topics, the services and the messages, however, we include the *actionlib Actions* concept in this list. Each time that a ROS node establishes a stream of messages using named buses based on publish/subscribe paradigm, we have topics. When a node needs to establish a remote procedure call that terminates quickly, it used a service. Finally, it is recommended to apply an action for those applications that move the robot for a known or unknown length of time and it is necessary to wait a period to terminate. All these communications run using different data structures called ROS messages.

Notwithstanding, when facing complex tasks that require the management of multiple sensors and actuators in order to generate robot behavior, it is necessary to somehow compose each robot's functionality in order to solve a task. Thus, for including the capability of long-term planning, we have designed an approach based on ROSPlan and for adding reactive behaviors based on FSMs, we have selected SMACH.

3.1.1. ROSPlan

ROSPlan is a framework based on ROS that provides a pool of tools to work with AI planning. It is also in charge of managing the knowledge that the robot has. This knowledge is written in PDDL, which is a family of languages for automated planning. There are two elements written in PDDL:

- The domain, which is composed of: the types of objects that can exist in the problem; the predicates, which are the properties that the objects can have; and the actions the robot can make to modify the objects.
- The problem, which is composed of: the objects that exist in the world of the robot; the propositions, which are the instances of the predicates; and the goals, which are the propositions that are wanted to be true.

ROSPlan is composed of the following components:

- Knowledge base: this component is in charge of storing the domain and the problem of the robot.
- Problem interface: this component creates the PDDL problem with the data from the knowledge base.
- Planner interface: this component creates the plan using the problem. A plan is a sequence of actions necessary to achieve specific objectives. There are many planners available: POPF, OPTIC, FF, LPG, TFD and TF.
- Parsing interface: this component is in charge of parsing the plan so that it can be executed.
- Plan dispatch: this component executes the plan by calling the actions that compose it. Each time an action is completed, the knowledge base can be changed by adding or removing propositions.

ROSPlan has several interfaces based on ROS to use these components. There are one ROS service and one ROS actionlib to execute each component, except the knowledge base which has several ROS services to query, edit and remove the knowledge.

3.1.2. SMACH

The aim of SMACH (that stands for State MACHine) is to create an architecture for developing a robust mid-level behavior generator. SMACH was released in 2010 [19]. It allows building hierarchical and concurrent state machines that communicate with the ROS system using its interfaces. As a result, it has the potential of generating robot behaviors for doing tasks. These behaviors can be controlled by high-level deliberative systems or provide their own mechanisms for having their own control engine.

SMACH provides two main interfaces: State and Container, the former describes the states of execution of the robot and all the possible outcomes; The latter is the collections of one or more states, and transitions that solve how to jump between robot states when there is a successful or unsuccessful robot action. As a result, we have a regular execution policy, also known as the classical State Machine, or concurrence execution policy, where different states are executed at the same time for generating complex robot behaviors.

Besides, SMACH provides an interface for simplifying the process of adding new behaviors using their own custom Python code. Moreover, it provides an interface with a set of parametrized state classes that eases the composition of SMACH solutions during the definition of new complex behaviors. The authors highlight the use of three states: ServiceState identifies the state that shows the execution of a ROS service call; MonitorState: those states that provide information from a ROS topic; and SimpleActionState: those states that represent the feedback of a ROS actionlib call.

3.2. MERLIN

This paper presents MERLIN (MachinEd Ros scheduLINg), a hybrid architecture for controlling and managing the behavior of a robot. MERLIN illustrated in Figure 1, is composed of a deliberative system, which is responsible for making decisions and creating plans, and a behavioral system, which has the actions the robot can make and the reactive systems. These systems consist of layers that are built on top of each other. The deliberative system is composed of Mission layer and Planning layer. The behavioral system is composed of the Executive layer and the Reactive layer.

The Mission layer is the layer in charge of creating the goals the robot has to achieve. It was designed so that other goals generation systems can be integrated easily. The main component of this layer is called Goal Dispatcher and it is communicated with the Executor from the Planning layer.

The Planning layer has to generate and execute the plans. Theses plans are sequences of actions to achieve goals. The main component of this layer is ROSPlan that provides tools to work with automatic pacification. Nevertheless, ROSPlan has limitations that are fixed by the other component of this layer, the Executor, which has to execute ROSPlan. Besides, ROSPlan is communicated with the actions of the Executive layer.

The Executive layer is composed of the actions that will be used to create the plans. Actions have been designed as FSMs to better control its execution. They can use the elements from the Reactive layer. In addition, actions can edit the knowledge the robot has.

The Reactive layer is composed of the elements which have to manage sensors and actuators. The reactive systems treated are navigation, object recognition, speech to text and text to speech.

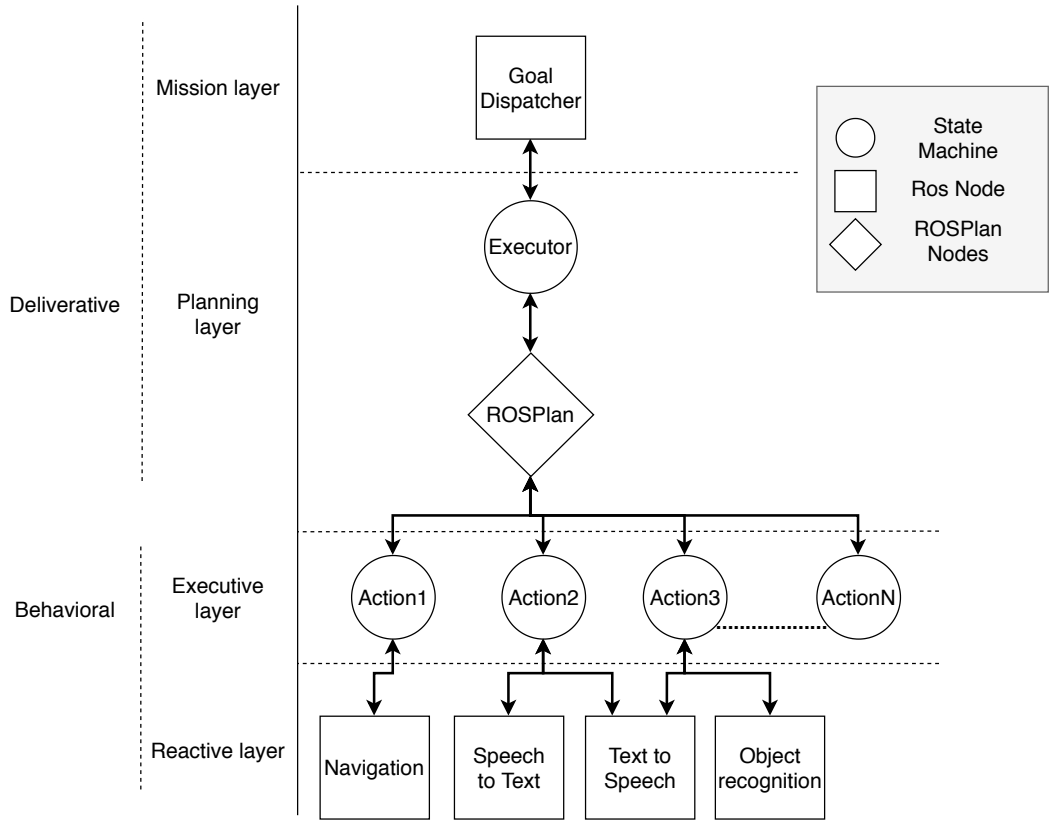


Figure 1. Proposed architecture.

3.2.1. MERLIN Core

At its core, MERLIN has a FSM capable of interacting with ROSPlan. As it is shown in Figure 2, this FSM is represented by the class MerlinSM. This class is based on class StateMachine, which is the class in green, and the class MerlinROSPlanAssistant. Thus, StateMachine belongs to SMACH and provides the state machine engine. On the other hand, MerlinROSPlanAssistant has a pool of functions to interact with the knowledge base of ROSPlan. Using these functions, the knowledge that the robot has from the scenario can be created, updated or deleted.

Finally, a factory has been created to manage the creation of states. This factory is represented in this class diagram by the class MerlinStateFactory, which has been developed using the factory software design pattern.

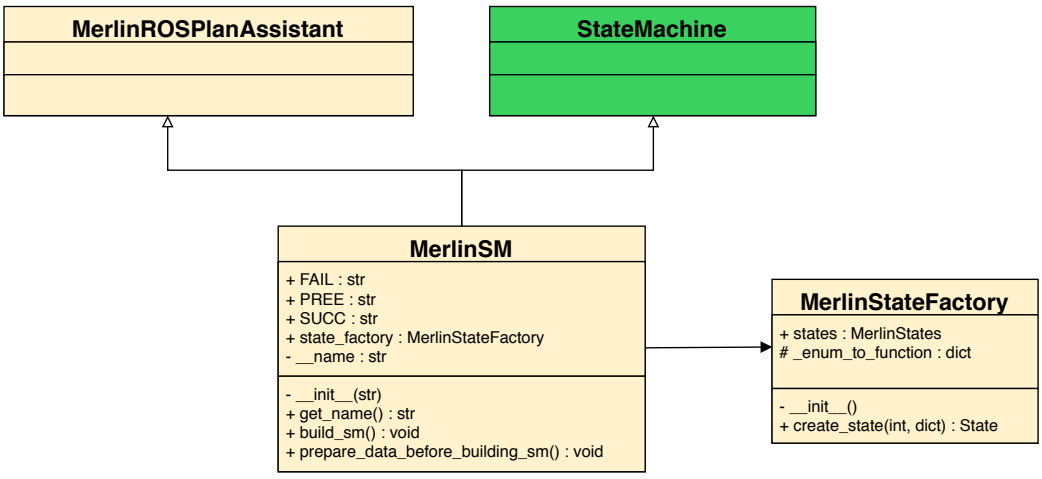


Figure 2. Class Diagram of MERLIN Finite-State Machine.

FSMs are used especially to create the Executive layer. In addition, they have been used in the Planning layer and the Mission layer. The FSM of the Executive layer and Planning layer has been created using the factory pattern, which is based on using a factory class to create the states of the FSM. The factory pattern will help us to arrange and create new states associated with robot basic capabilities such as navigation, object recognition. Besides, it provides the states for the planning layer.

3.2.2. MERLIN Deliberative System

The deliberative system has to control the behavior a robot needs to achieve certain goals. It is composed of two layers:

1: Mission layer: This layer has to manage the goals a robot needs to achieve. It can be implemented as a normal ROS node or as a FSM, as it is presented in Figure 3. Each state has several goals that are sent to the Executor in the next layer, the Planning layer. As it is shown in Figure 1, this component is called Goal Dispatcher.

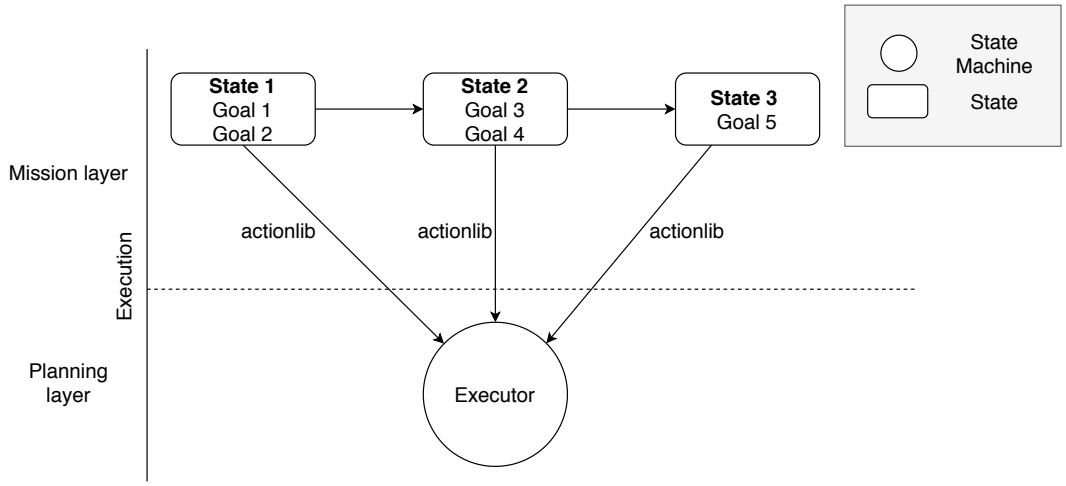


Figure 3. Communication diagram of Mission layer, designed as a Finite-State Machine and the Executor from the Planning layer.

Its main software component is MerlinExecutorClient. It is a Python class that uses a ROS actionlib client to communicate this layer with the Planning layer. Goals must be written in PDDL. This class provides a new functionality that allows canceling the current robot behavior. This is useful in assistive robotics where there are goals with different priorities, mainly oriented to user services.

2: Planning Layer: This layer is responsible for creating the plans, which are the sequences of actions needed to achieve certain goals. Besides, it manages the execution and monitoring of these actions.

From the software perspective, there are two main components: ROSPlan and the Executor. ROSPlan manages the knowledge about the environment and provides the engine for managing planning. The Executor is a ROS node implemented as a FSM. It is responsible for executing ROSPlan. It has a state to execute every ROSPlan component using ROS services. In addition, the Executor deletes the goals used and replans if ROSPlan fails.

The Executor is the element that communicates the Mission layer and the Planning layer. To do this, the Executor has an actionlib server that receives goals. When a goal is received, the Executor adds it to the knowledge base and starts ROSPlan.

3.2.3. MERLIN Behavioral System

The behavioral system provides the control engine for managing the robot behaviors of the actions. It is composed of an Executive Layer and a Reactive Layer.

3: Executive layer: This layer is composed of the actions that are used to create the plans. They have been implemented as a FSM using SMACH. The progress and state of the action can be monitored thanks to SMACH Viewer [28], a tool to visualize FSM created with SMACH.

This layer provides a crucial functionality to the system. It provides the mechanism for the integration of the SMACH FSM system with ROSPlan. There are two main classes created to do this:

- MerlinActionSM: this is the base class to create new MERLIN actions. As it is shown in Figure 4, MerlinActionSM inherits from MerlinSM, which is the MERLIN class to create FSM. These actions can be executed using an actionlib, which is created with the name of the action.
- MerlinAction: this class is the bridge between ROSPlan and the MERLIN actions. Inheriting from RPActionInterface, which is the class ROSPlan has to create new actions, this class can communicate with ROSPlan. RPActionInterface is the class in green shown in Figure 4. Besides, to communicate with MERLIN actions, actionlibs are used.

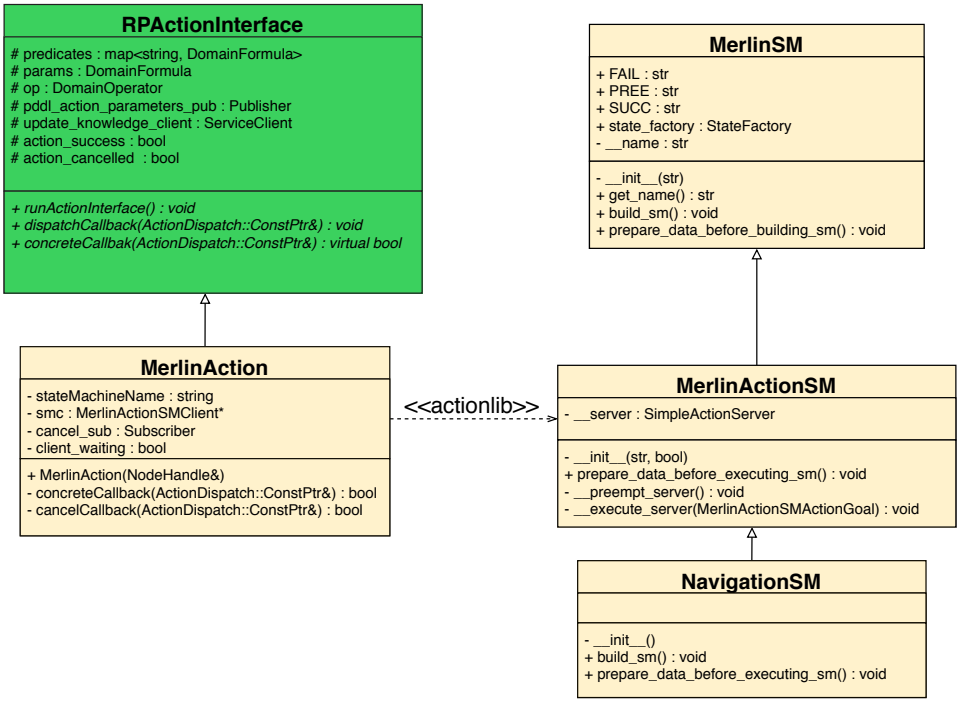


Figure 4. Class Diagram of the Executive layer.

Figure 5 shows the communication flow between ROSPlan and MERLIN actions. There are as many instances of MerlinAction as there are MERLIN actions. These instances are responsible for managing the MERLIN actions. In addition, they are communicated with ROSPlan using a topic, which is a communication channel to read and write data asynchronously. When ROSPlan wants to use an action, it writes in this topic the action name. Then, the MerlinAction instances read the name and the instance in charge of that action has to execute it.

The use of actionlib to control the execution of MERLIN actions allows canceling them. Thanks to this, when ROSPlan is canceled, MERLIN actions can be canceled too. In ROSPlan, actions are ignored by default when a plan is canceled. In fact, ROSPlan actions can read the cancel message, however, the handling of the messages that ROSPlan actions do is sequential, so this message is handled when the actions are completed.

The launch files are used to create MerlinAction instances and execute MERLIN actions. To create the MerlinAction instance, a launch has been created that takes the name of a MERLIN action and creates the MerlinAction instance associated with that action. The name of the action is used to create the actionlib, which is used to communicate each action with its corresponding MerlinAction instance.

Finally, each time a new MERLIN based application is developed, a new launch file has to be created to execute its MERLIN actions and to create the MerlinAction instances.

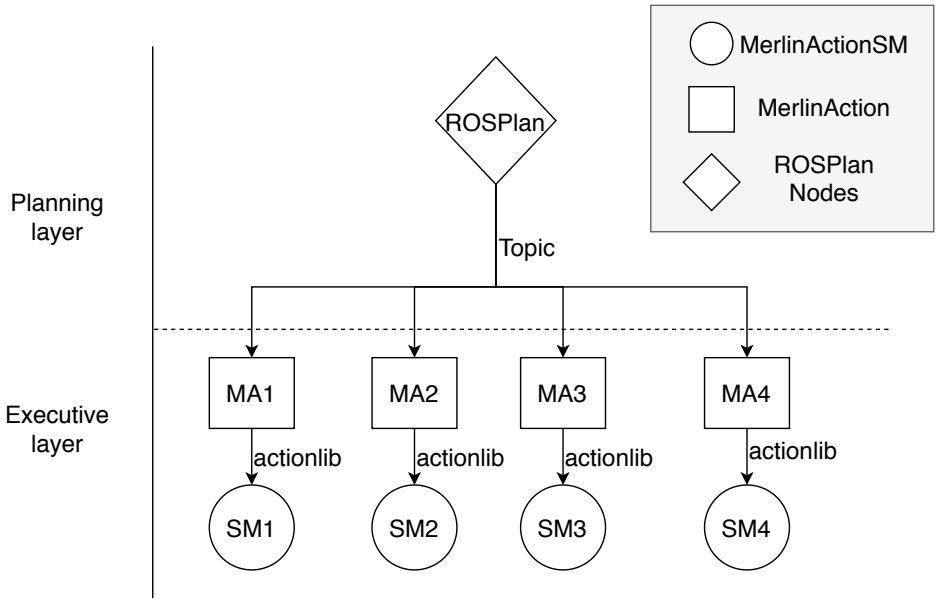


Figure 5. Communication diagram of Executive layer (Deliberative) and ROSPlan (Behavioral).

4: Reactive layer: This layer is composed of the reactive system the robot needs to create the actions. Theses systems are navigation, speech recognition, object recognition and speech synthesis. In order for the FSM to be able to use these systems, they must be accessible through services and actionlib. Especially, it is better to use actionlib since they provide more control over the execution. In addition, there is a state to call for each system. These states are created with the state factory shown in Figure 2.

3.3. MERLIN Example

In order to understand the issues associated when adding a new behavior to the architecture, we have illustrated the procedure using a formalized task [29]. Thus, we have selected the SCIROC restaurant challenge. This challenge is defined in the European Robotic Web Service rulebook [30]. The task consists of creating a robot that can serve the table of a restaurant. There are three stages in this challenge:

1. Checking the state of a table: the robot has to check the state of each table. The table's states are: needs serving, already served, needs cleaning and ready.
2. Serving an order: the robot has to serve an order to a table that needs serving. It also has to check if the barman has prepared the order properly.
3. Guiding a new client: the robot has to check if there is a new client at the entrance. Then, it has to guide the new client to a free table.

In order to create new behaviors using MERLIN, the flow of Figure 6 is followed. Thus, to include the behaviors and actions required for solving the restaurant challenge, we decompose the procedure into phases:

Our first phase faces the problem from the deliberative perspective. This phase is represented by the green elements of Figure 6. It is necessary to define the two main elements of our PDDL-based system, the domain and the problem. For instance, the code presented in Listing 1 is the PDDL problem. Then, they must be validated using the planner of ROSPlan. If they are not valid, they must be fixed.

Listing 1: Planning Domain Definition Language Problem associated to SCIROC restaurant challenge.

```
(define (problem restaurant_prb)
        (:domain restaurant)
        (:objects
                wp1 wp2 wp3 barman_wp wp0 - waypoint
                robot_waiting_wp person_waiting_wp - waypoint
                t1 t2 t3 - table
                barman - person
        )
        (:init
                (robot_at wp0)
                (robot_immobile)
                (is_wp_near_table wp1 t1)
                (is_wp_near_table wp2 t2)
                (is_wp_near_table wp3 t3)
                (person_at barman barman_wp)
                (is_robot_waiting_wp robot_waiting_wp)
                (is_person_waiting_wp person_waiting_wp)
        )
        (:goal   ())
)
```

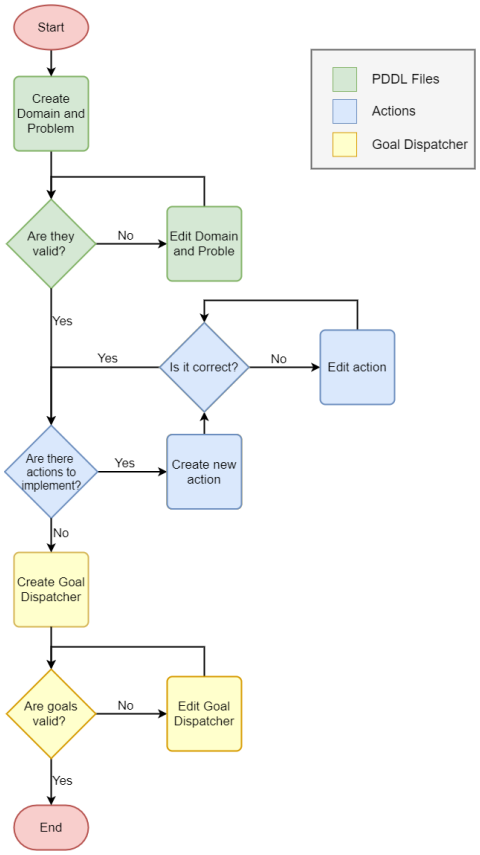


Figure 6. Programming flowchart to create new behaviors using MERLIN.

The second phase deals with the behavioral layer of the architecture. For using the SMACH library, it is necessary to split the restaurant challenge into subtasks given different complexities and

functionalities. These subtasks are equivalent to the actions that ROSPlan uses and that are defined in the domain. As a result, we will have hierarchical solutions as is presented in Figure 7. This phase is represented by the blue elements of Figure 6.

From a raw programming view, it is necessary a FSM for each new action. If all actions have already been implemented, the second phase can be ignored. In addition, the FSMs of the actions can have nested FSMs. Finally, it is necessary to create the launch file that executes the MERLIN action and creates a MerlinAction instance associated with that action. To do this, the launch file takes the name of an action.

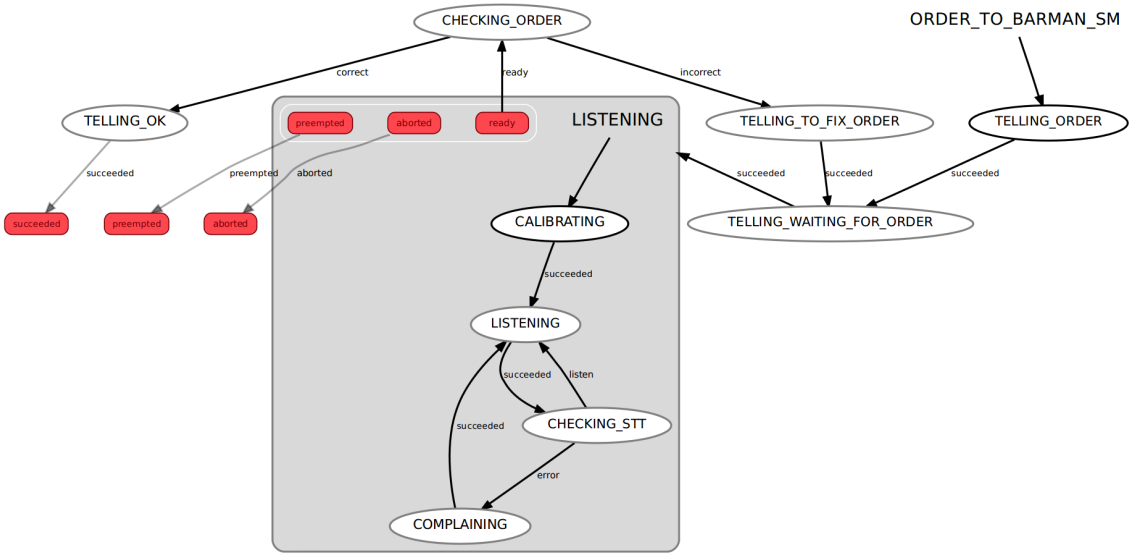


Figure 7. SMACH Viewer presentation of the FSM of the order to barman action.

Finally, it is necessary to create a software component that manages the goals that the robot must achieve. The goals must be validated using the planner of ROSPlan. This component, whose name is Goal Dispatcher, belongs to the Mission layer and it is created as a MERLIN FSM that will send the PDDL goals to the Executor using an actionlib. This phase is represented by the yellow elements of Figure 6.

4. Experiments

This research evaluates the performance of the proposed architecture using ROSPLan and SMACH, after the integration process. Besides, this research evaluates the robot performance when using the enhanced version of the architecture under the MERLIN proposed approach. The first approach is called the Naive approach and includes the original ROSPlan and SMACH components. The second approach deals with MERLIN. It is necessary to emphasize that it is necessary to make use of an executor component for being able to evaluate the overall performance. For this reason, the Executor developed for MERLIN is also used in the Naive approach.

4.1. Description

This experiment aims to validate the MERLIN approach for providing an architecture for a social robot. Besides, the experiment evaluates the usability and performance of our MERLIN approach and compares it with the Naive approach. Translating the restaurant challenge to an apartment, we have defined a set of four points of interest (PoI) to assist an individual: Dorm room, Bathroom, Kitchen and main door. These points are the way-points marked with red crosses in the map of Figure 8 and their positions are defined in the table of the same figure.

Then we have defined 3 different missions, where these PoIs are visited as a pool of tasks that the robot needs to check if a service is required (Does the user need something in point X_y?). To check these PoIs the actions used are navigation and dialogue. The navigation provides the movement functionality and the ROS navigation node is used. The dialogue is performed using the soundplay node also provided by ROS repositories. For each mission, we create a fixed list of PoIs that need to be visited using our pool of PoIs. In order to evaluate the impact and resilience of the architecture to manage missions, half of the tasks associated with the mission are canceled (for this reason the number of way-points is even).

4.2. Scenario and Metrics

Three different scenarios based on the Gazebo world and map of Figure 8 have been selected. These three scenarios are used with MERLIN and ROSPlan for the experiment. They differ in the number of tasks to be performed, that is, in the number of way-points to check. There are three different amounts of tasks for each mission: Mission 1:6; Mission 2:20; and Mission 3:120. These tasks (way-points to visit) are chosen randomly within the four points in the table of the map of Figure 8, so they will be repeated several times.

The following metrics were chosen for evaluating the performance of the architecture:

1. Time: we measure the seconds needed to complete a pool of tasks.
2. Traversed distance: we measure the meters used to complete a pool of tasks.

The election of these values allows us to measure the performance of a cognitive architecture. Firstly, the time metric measures the time needed by each mission, which means that it will cover the impact of moving from one task to another, which means to cancel planning and behavioral layers for dispatching a new goal and activating the new actions. Secondly, the traversed distance allows measuring the navigated distance for fulfilling a mission. A robot driving longer traversed distances implies higher battery consumption and less effective service time. At the same time, it means that the process of changing from one task to another has not only time impact but also performance implications.

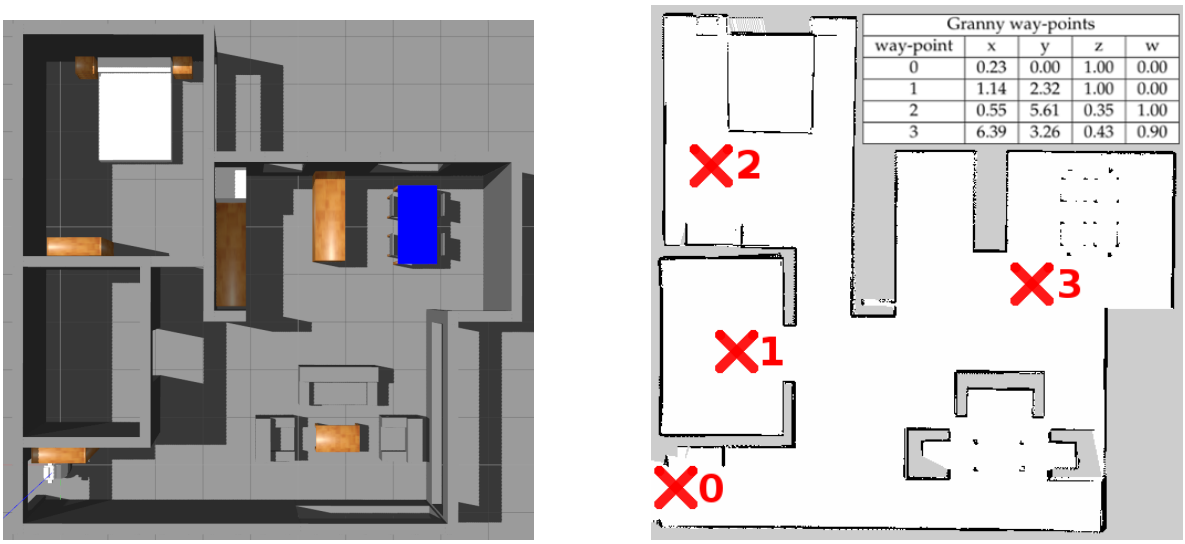


Figure 8. Gazebo zenith view and map deployed for the experimental validation.

4.3. Results

This section describes the results when performing the three proposed missions using Gazebo simulator and the RB1 robot platform manufactured by Robotnik [31]. We have illustrated (Figure 9) an example of Mission 2 experiment showing the paths followed by the RB1 robot using Naive and MERLIN.

Table 1 presents the results of Missions 1 and 2 after running it five times with different lists. It presents the robot performance when 6 or 20 points are established during the regular operating mode of the task. These values overview the significant difference between approaches. Graphically, Figure 10 shows the time difference when facing Mission 1 and Mission 2 given the tasks associated, notwithstanding, it is also clear the time needed for solving these missions is considerably less. The same behavior is also presented when measuring the traversal distance, the values depicted in Figure 11 also present better results when the robot deploys MERLIN architecture. When compared, MERLIN shows fewer fingerprints during robot traversal at home as seen in Figure 9.

Besides, the low differences between median and average time (arithmetic means) plus the evaluation of the coefficient of variation, which defines the ratio of dispersion of frequency distribution and presents values under 1, outlines that, at this stage, we can trust in the repeatability of our experiments.

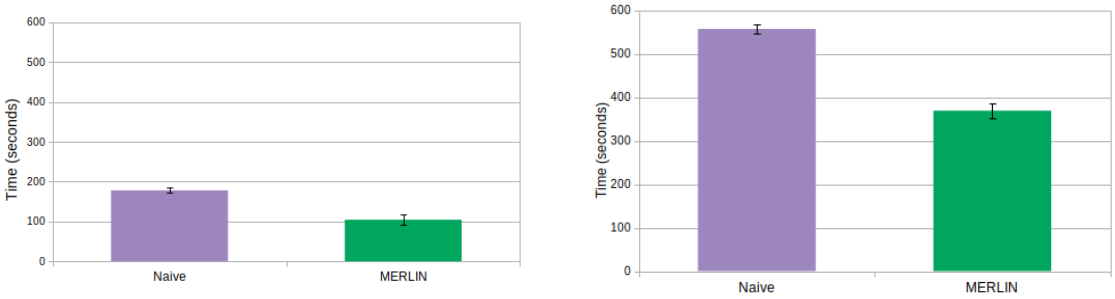
Table 1. Experimental results for each mission.

	Mission 1		**Mission 2**	
Approach	**Avg Time [Median] ± SD (Seconds)**	**Trav. Distance ± SD (Meters)**	**Avg Time (Seconds) ± SD (Seconds)**	**Trav. Distance ± SD (Meters)**
Naive	178.20 [178.19] ± 6.03	46.65 [46.92] ± 2.05	556.82 [557.54] ± 9.89	138.82 [139.20] ± 4.94
MERLIN	104.46 [110.66] ± 12.30	21.72 [23.64] ± 3.94	369.14 [373.31] ± 18.06	80.23 [78.50] ± 6.26

Finally, we perform an extensive experiment aiming to evaluate the robot performance with longer missions. This experiment runs around 120 PoIs, again canceling half of the requests. This scenario presents a duration of 2154 s when using MERLIN versus the 3379 s needed when the naive approach is presented. There is a difference in traversed distance of 369 m (MERLIN, 475 and Naive, 844) when using MERLIN.

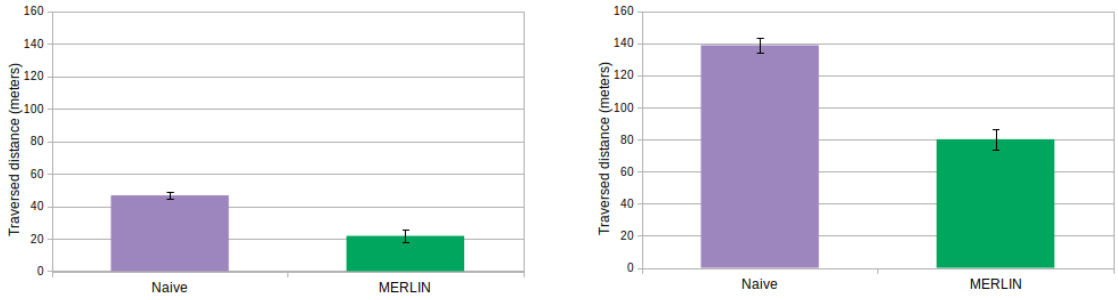
(**a**) Naive (**b**) MERLIN

Figure 9. Paths of experimental approach in Mission 2 with 20 PoIs.



(**a**) Mission 1 (6 PoIs) (**b**) Mission 2 (20 PoIs)

Figure 10. Time needed for fulfilling the missions using Naive and MERLIN approaches.



(**a**) Mission 1 (6 PoIs) (**b**) Mission 2 (20 PoIs)

Figure 11. Traversed distances for fulfilling the missions using Naive and MERLIN approaches.

5. Discussion

This paper is not intended to make a comparison between different cognitive architecture approaches, instead, it aims to provide the details for integrating and deploying one in a robot oriented to offer a service. Many factors determined our development and guided us to obtain MERLIN. As a result, the numbers of our experimental approaches show that although it is possible to deploy an architecture using OTS components, it is necessary to add some features in order to provide the expected results, at least for our motivation, long-term support.

For easily articulate the discussion section, we track the three research questions proposed at the beginning of our research:

5.1. RQ1: What Are the Elements for Providing Robot Behaviors in the Long-Term?

We have identified the "Executor" component as the crucial element for defining the complete architecture and being able to recover under canceled or blocked plans.

Besides, it is necessary to have in mind the good definition of the PDDL approach for the deliberative layer and the precise decomposition of the task into subtasks.

The experiment with 120 PoIs demonstrated that, even with this level of assistance, we have an architecture ready for working more than a punctual case. The results present a robot working for more almost one hour, and prove that MERLIN reduces the time in almost 15 min.

5.2. RQ2: How We Can Integrate State-of-the-Art Components, or in This Case, Off-the-Shelf Software Components, for Creating a Cognitive Architecture?

It is not a straightforward process that requires reading source code, their documentation and extract as much information as possible from source code. For instance, we obtained an initial version of packages dependencies using Doxygen software [32].

The authors believe that the architecture section presents an overview that clarifies the main components involved during the development of new robot behavior. Furthermore, it is included an example with programming details that provides an overview and a lesson learned for the scientific community.

5.3. RQ3: What Are the Mechanisms for Successfully Managing Expected or Unexpected Requests Associated with a Given Robot Duty?

We already defined the Executor and the Goal dispatcher as the main components for recovering after canceling the mission or a non-recovery state in the scenario. Thus, the testing scenarios present a fully usable architecture, and in addition, our proposal, MERLIN, reduces almost 40% of the time spent in the Naive approach, and around the 45% of enhancement when facing the covered distance. This is illustrated in Figures 10 and 11. These values allow us to say that MERLIN is a better approach for reducing the noise associated with a robot moving at home, such as physical noise (motors of the robot) or physiological disturbances (why the robot is doing that if it should be doing something else).

Besides, it is interesting to reduce the consumption when working in scenarios where requests could be easily canceled (this research proposes 50%) and it is necessary to manage a lot of unsuccessful tasks.

6. Conclusions

The target of this paper was to describe the integration of ROSPlan and SMACH to demonstrate the execution of a hybrid architecture in a social robot scenario. In order to do this, the inclusion of new software components at different levels such as the Executor, which has to control the ROSPlan execution, was necessary. In addition, it has to delete the goals from the knowledge base.

Another important component is the MERLIN actions of the Executive layer. The actions of MERLIN are FSM created with SMACH that ROSPlan can use to create plans. As a result, the Executive layer provides the mechanism for the integration of the SMACH FSM system with ROSPlan.

The future work is based on creating a new planning system that will replace ROSPlan. This new system will not have the limitations that have been fixed by MERLIN. In addition, a Web Application will be created to monitor the architecture. Instead of using the monitoring tools of SMACH and ROSPlan separately, this web application will have information about the knowledge, the progress of the plan, the state of the planner and the state of the FSM. Besides, thanks to being a web application, this information can be monitored independently of ROS and it is possible to use any device to access it. Finally, migrating all Python code to Python 3 is a must.

Besides, it is planned to add an ontological approach for managing the knowledge representation in our architecture given the good results presented by other researchers [33]. It will help us to enhance the flexibility and scalability to other scenarios.

Author Contributions: Data curation, F.J.R.-L.; Investigation, M.Á.G.-S., F.J.R.-L., C.Á.-A., Á.M.G.-H. and C.F.-L.; Project administration, M.Á.G.-S. and F.J.R.-L.; Software, M.Á.G.-S.; Supervision, Á.M.G.-H. and C.F.-L.; Validation, M.Á.G.-S. and F.J.R.-L.; Writing—original draft, M.Á.G.-S. and F.J.R.-L. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially funded by Ministerio de Ciencia, Innovación y Universidades through grant RTI2018-100683-B-I00.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

UML	Unified Modeling Language
SysML	Systems Modeling Language
PDDL	Planning Domain Definition Language
FSM	Finite State Machine
ROS	Robotic Operating System
SMACH	State MACHine library

References

1. Kortenkamp, D.; Simmons, R.; Brugali, D. Robotic systems architectures and programming. In *Springer Handbook of Robotics*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 283–306.
2. Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*; Addison-Wesley Professional: Boston, MA, USA, 2004.
3. Vitanza, A.; D'Onofrio, G.; Ricciardi, F.; Sancarlo, D.; Greco, A.; Giuliani, F., Assistive Robots for the Elderly: Innovative Tools to Gather Health Relevant Data. In *Data Science for Healthcare: Methodologies and Applications*; Consoli, S., Reforgiato Recupero, D., Petković, M., Eds.; Springer International Publishing: Cham, Swizerland, 2019; pp. 195–215. [CrossRef]
4. Fox, M.; Long, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.* **2003**, *20*, 61–124. [CrossRef]

5. Walker, N.; Jiang, Y.; Cakmak, M.; Stone, P. Desiderata for Planning Systems in General-Purpose Service Robots. *arXiv* **2019**, arXiv:1907.02300.
6. Arkin, R.C. *Reactive Robotic Systems*; MIT Press: Cambridge, MA, USA, 1995 .
7. Brooks, R. A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* **1986**, *2*, 14–23. [CrossRef]
8. Gat, E. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. *AAAi* **1992**, *1992*, 809.
9. Gat, E.; Bonnasso, R.P.; Murphy, R. On three-layer architectures. *Artif. Intell. Mob. Robot.* **1998**, *195*, 210.
10. Ahmad, A.; Babar, M.A. Software architectures for robotic systems: A systematic mapping study. *J. Syst. Softw.* **2016**, *122*, 16–39. [CrossRef]
11. Kotseruba, I.; Tsotsos, J.K. 40 years of cognitive architectures: Core cognitive abilities and practical applications. *Artif. Intell. Rev.* **2020**, *53*, 17–94. [CrossRef]
12. Bustos, P.; Manso, L.J.; Bandera, A.J.; Bandera, J.P.; Garcia-Varea, I.; Martinez-Gomez, J. The CORTEX cognitive robotics architecture: Use cases. *Cogn. Syst. Res.* **2019**, *55*, 107–123. [CrossRef]
13. Manso, L.; Bachiller, P.; Bustos, P.; Núnez, P.; Cintas, R.; Calderita, L. Robocomp: A tool-based robotics framework. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 251–262.
14. Agüero, C.E.; Canas, J.M.; Martın, F.; Perdices, E. Behavior-based iterative component architecture for soccer applications with the nao humanoid. In Proceedings of the 5th Workshop on Humanoids Soccer Robots, Nashville, TN, USA, 7 December 2010; Volume 127.
15. Firby, R.J. Building symbolic primitives with continuous control routines. In *Artificial Intelligence Planning Systems*; Elsevier: Amsterdam, The Netherlands, 1992; pp. 62–69.
16. Plaza, J.M.C.; Olivera, V.M. Integrating Behaviors for Mobile Robots An ethological Approach. In *Cutting Edge Robotics*; IntechOpen: London, UK, 2005; p. 311.
17. Rodríguez-Lera, F.J.; Matellán-Olivera, V.; Conde-González, M.Á.; Martín-Rico, F. HiMoP: A three-component architecture to create more human-acceptable social-assistive robots. *Cogn. Process.* **2018**, *19*, 233–244. [CrossRef] [PubMed]
18. Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; Carreras, M. Rosplan: Planning in the robot operating system. In Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, Jerusalem, Israel, 7–11 June 2015.
19. Bohren, J.; Cousins, S. The smach high-level executive [ros news]. *IEEE Robot. Autom. Mag.* **2010**, *17*, 18–20. [CrossRef]
20. Lima, O.; Ventura, R.; Awaad, I. Integrating Classical Planning and Real Robots in Industrial and Service Robotics Domains. In Proceedings of the PlanRob 2018—6th Workshop on Planning and Robotics, Held at ICAPS 2018, Delft, The Netherlands, 24–29 June 2018.
21. de Oliveira, R.W.; Bauchspiess, R.; Porto, L.H.; de Brito, C.G.; Figueredo, L.F.; Borges, G.A.; Ramos, G.N. A Robot Architecture for Outdoor Competitions. *J. Intell. Robot. Syst.* **2020**, *99*, 629–646. [CrossRef]
22. Corbato, C.H.; Milosevic, Z.; Olivares, C.; Rodriguez, G.; Rossi, C. Meta-control and Self-Awareness for the UX-1 Autonomous Underwater Robot. In *Iberian Robotics Conference*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 404–415.
23. Gogolla, M.; Vallecillo, A. (An Example for) Formally Modeling Robot Behavior with UML and OCL. In *Federation of International Conferences on Software Technologies: Applications and Foundations*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 232–246.
24. Estévez, E.; García, A.S.; García, J.G.; Ortega, J.G. Aproximación Basada en UML para el Diseño y Codificación Automática de Plataformas Robóticas Manipuladoras. *Rev. Iberoam. Autom. Inform. Ind. RIAI* **2017**, *14*, 82–93. [CrossRef]
25. Claudio, R.; Sergio, D.; Pascual, C.; André, D.; Alfredo, M.; Carlos, A.; José, A. *UX-1 Robot Software Architecture Report, UNEXMIN Deliberable D3.1*; Technical Report; Universidad Politécnica de Madrid (UPM): Madrid, Spain, 2017.
26. Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A.Y. ROS: An open-source Robot Operating System. In Proceedings of the ICRA Workshop on Open Source Software, Kobe, Japan, 12–17 May 2009; Volume 3, p. 5.
27. ros_comm. Available online: http://wiki.ros.org/ros_comm (accessed on 28 August 2020).

28. Bohren, J. SMACH Viewer. Available online: http://wiki.ros.org/smach_viewer (accessed on 28 August 2020).
29. Basiri, M.; Piazza, E.; Matteucci, M.; Lima, P. Benchmarking Functionalities of Domestic Service Robots Through Scientific Competitions. *KI-Künstliche Intell.* **2019**, *33*, 357–367. [CrossRef]
30. Basiri, M.; Piazza, E.; Matteucci, M.; Lima, P. *Rulebook of the European Robotic League for Consumer Service Robots*; euRobotics: Brussels, Belgium, 2018.
31. Robotnik. Available online: https://robotnik.eu/products/mobile-manipulators/rb-1/ (accessed on 28 August 2020).
32. Doxygen. Available online: https://www.doxygen.nl/index.html (accessed on 28 August 2020).
33. Umbrico, A.; Sorrentino, A.; Cavallo, F.; Fiorini, L.; Orlandini, A.; Cesta, A. Toward the Integration of Perception and Knowledge Reasoning: An Adaptive Rehabilitation Scenario. In *AI*AAL@AI*IA 2019*; CEUR-WS: Aachen, Germany, 2019; pp. 10–21.