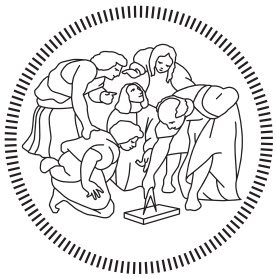
POLITECNICO DI MILANO

Facoltà di Ingegneria
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in
Computer Science and Engineering



A MODULAR ARCHITECTURE FOR AUTONOMOUS ROBOTS

Advisor: PROF. ANDREA BONARINI

Master Graduation Thesis by:

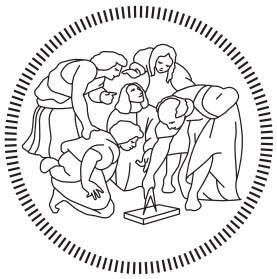
CESARE CONSONNI
Student Id n. 10476810

Academic Year 2020-2021

POLITECNICO DI MILANO

Facoltà di Ingegneria
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Computer Science and Engineering



UNA ARCHITETTURA MODULARE PER ROBOT AUTONOMI

Relatore: PROF. ANDREA BONARINI

Tesi di Laurea Magistrale di:

CESARE CONSONNI
Matricola n. 10476810

Anno Accademico 2020-2021

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

This template has been adapted by Emanuele Mason, Andrea Cominola and Daniela Anghileri: *A template for master thesis at DEIB*, June 2015. This version of the paper has been later adapted by Marco Cannici, March 2018.

ACKNOWLEDGMENTS

Vorrei ringraziare il Professore Andrea Bonarini: Lei è stata la mia guida in questo ultimo mio tratto di percorso, rispondendo a tutte le mie curiosità ed aiutandomi a fare al meglio questo lavoro.
Ora viene il momento "personale"... premetto sarò prolisso, ma credo che ognuno meriti il suo momento.
Il primo pensiero va a te amore mio: questa avventura non poteva avere un inzio migliore di te, scoppiata in lacrime di gioia, la sera precedente al "primo giorno di scuola". Ci sei stata sempre con le tue coccole e le tue premure (... e qualche calcio sul sedere quando serviva...) a sopportarmi, supportarmi e spronarmi ogni volta che anche solo minimamente vacillavo. Ti ho conosciuta il mio primo anno qui al "Poli" che eri una bellissima ragazza e ti trovo ora una Donna stupenda. Siamo cresciuti insieme, sostenendoci a vicenda e riempiendo le nostre giornate di risate, baci e taaaanta pazienza. Sei la mia musa ispiratrice, la mia pesce-debugger e voglio che tu sia l'ultima persona che vedrò ogni sera prima di chiudere gli occhi.
Il secondo pensiero va ai miei genitori: mi avete permesso di "guardare il mondo con i miei occhi" per trovare e scegliere la mia strada, qualunque essa fosse. Mi avete aiutato a crescere, mi avete insegnato quanto sia importante essere curioso, mi avete trasmesso passione e dedizione per il lavoro, mi siete d'esempio per l'amore che provate l'uno per l'altra. Ci avete aperto la porta di casa soprattutto in questi due anni di pandemia, offrendoci un rifugio e tantissimi momenti belli insieme, diventando veramente una famiglia unita e felice. So di essere un piccolo collerico cocciuto brontolone, ma spero di avervi reso fieri e orgogliosi per dove sono arrivato oggi, non solo accademicamente, ma soprattutto come persona.
L'ultimo pensiero va ai miei "vecchietti" che non ci sono più: Romana e Pietro. «Studìa studìa» mi dicevi sempre con affetto, nonna, quando ti venivo a trovare tra una storia e l'altra (che era sempre la prima volta che raccontavi...): ci tenevi così tanto che arrivassi qui, e mi fa piangere il cuore che per così poco tu non sia qui a festeggiare con noi. Mentre tu, nonno, "capostipite di questa dinastia di ingegneri" sei stato un po' la ragione, un po' la fortuna e un po' l'esempio che mi ha spinto a scegliere questa professione.

CONTENTS

Abstract xiii

1 INTRODUCTION 1
- 1.1 Brief Description of the Work . . . 1
- 1.2 Structure of the Thesis . . . 2

2 STATE OF THE ART 3
- 2.1 Autism spectrum disorder . . . 3
- 2.2 Triad of impairment . . . 3
- 2.3 Therapy with Social Robots . . . 5
- 2.4 Design features . . . 7
- 2.5 Target Behaviour . . . 8
- 2.6 Robot Operating System 2 . . . 8

3 METHODOLOGY 11
- 3.1 Script . . . 12
- 3.2 Architecture Overview . . . 16

4 IMPLEMENTATION 21
- 4.1 Map Message Structure . . . 21
- 4.2 Loop . . . 24
- 4.3 Rules and Their Grammar . . . 25
- 4.4 Json-Utility for Scripts . . . 28
- 4.5 Update Mechanism . . . 36

5 SIMULATION 39
- 5.1 Android App . . . 39
- 5.2 Gazebo Model . . . 47
- 5.3 Fake Actuator . . . 49

6 CONCLUSIONS 51
- 6.1 Conclusions . . . 51
- 6.2 Future Work . . . 52

BIBLIOGRAPHY 53

LIST OF FIGURES

Figure 2.1 Triad of impairments in Autism Spectrum Disorder. . . . 4
Figure 2.2 Nao robot. 5
Figure 2.3 Teo robot. 6
Figure 2.4 Tito robot. 6
Figure 2.5 ROS2 Graph 9
Figure 3.1 Architecture Overview 11
Figure 3.2 Sense Think Act paradigm implementation 12
Figure 3.3 Structure of a Script 13
Figure 3.4 Schematic representation of an instruction block 14
Figure 3.5 Example effects of jump vs. terminus 15
Figure 3.6 Map propagation diagram. 17
Figure 3.7 Model of the robot realized in Gazebo Simulator 19
Figure 4.1 Loop Schema 24
Figure 5.1 *Android App*: Main Activity 40
Figure 5.2 Get Ready Button 41
Figure 5.3 Start Button 41
Figure 5.4 Stop Button 42
Figure 5.5 Loading Button 42
Figure 5.6 Restart Button 43
Figure 5.7 Clear Button 43
Figure 5.8 Plus Button 44
Figure 5.9 Selection Activity Screen 44
Figure 5.10 Example for the Test Dance Activity 45
Figure 5.11 Example of an activity in current state 45
Figure 5.12 Example of a completed activity 46
Figure 5.13 Example of a not-yet-executed activity 46
Figure 5.14 Example of screenshot 47
Figure 5.15 The real robot and the modelled one 48
Figure 5.16 Highlighted parts of the model 48
Figure 5.17 XBOX360 Controller and Related Key Bounds 49
Figure 6.1 Nvidia Jetson Nano Board 52

LIST OF TABLES

Table 4.1 Update Guide 37
Table 4.2 Update Guide Cont'd 38

ABSTRACT

Autism Spectrum Disorders (ASD) are neurodevelopment conditions that involve aspects of social and communication skills. Each single case is characterized by different persistent challenges that the subject has to face every day [1]. Studies suggest that children affected by ASD exhibit positive social behaviors while interacting with robots that are not observed while interacting in other contexts, such as sessions with therapists or caregivers [3]. Under this light Social Assistive Robots (SAR), robots designed to help humans thought social interactions [2], when used under these circumstances should be user-adjustable, in order to match child's own unique challenges, and they should offer a certain degree of autonomy to allow multiple kind of interactions among the therapist, the subject and the robot itself.
The aim of this work is to provide a modular architecture to allow the execution of activities, established through a script-like language, combined with "on-the-fly" adaptation of them via rule triggering and joystick-based human intervention.

SOMMARIO

I Disturbi dello Spettro Autistico (DSA) sono condizioni legate al neurosviluppo che coinvolgono aspetti delle abilità comunicative e sociali. Ogni singolo caso è caratterizzato da differenti e persistenti sfide che il soggetto ha da affrontare ogni giorno [1]. Studi indicano che i bambini affetti da DSA esibiscono comportamenti sociali posisitivi durante l'interazione con i robot che non sono osservati in altri contesti, come sessioni con terapisti o caregiver [3]. Sotto questa luce i Social Assistive Robots (SAR), robot progettati per aiutare gli essere umani attraverso relazioni sociali [2], quando usati in queste circostanze devono essere personalizzabili, in maniera tale da adattarsi alle sfide uniche di ogni bambino, e offrire un determinato grado di autonomia per permettere multipli tipi di interazione tra terapista, soggetto e robot stesso.
Lo scopo di questo lavoro è quello di provvedere ad una architettura modulare che permetta l'esecuzione di attività, stabilite attraverso un linguaggio simil-script, combinate ad adattamenti "al volo" tramite l'attivazione di regole e l'intervento umano attraverso un joystick.

1 INTRODUCTION

Autism Spectrum Disorders (ASD) are neurodevelopment conditions that involve aspects of social and communication skills. Each single case is characterized by different persistent challenges that the subject has to face every day [1]. Studies suggest that children affected by ASD exhibit positive social behaviors while interacting with robots that are not observed while interacting in other contexts, such as sessions with therapists or caregivers [3].
The purpose of this work is to develop the skeleton where different activities can be scheduled and executed adapting to the environment and customized taking into consideration the involved subject.

1.1 BRIEF DESCRIPTION OF THE WORK

This thesis has been developed to try to propose a solution to core problems of the existing legacy system (the one proposed in [6] and [5]): Lagun/Oimi robot (respectively named) is based on a Python program executed upon a Raspberry Pi board. The need of allowing the development of a wide range of different applications in an easier way, not only to therapy purposes but above all to play[23], and the need of making space to sophisticated data analysis modules brought to the urge of provide a backbone on which build future works.
Inspired by the thesis "Design and Implementation of an Actor Robot for a Theatrical Play" by Lorenzo Bonetti, where a script-like language is used to implement a play [4], a modular architecture has been developed to prearrange a soil where activities, coded through Script(s), could be implemented and executed in an organic way.
A Script is a collection of goals, associated with the sequence of actions required to accomplish them, and a set of rules designed to guide, to correct and to make safe the execution of each moment of the interaction. Each rule is constituted by a body and a condition, to be satisfied in order to be executed, in terms of elements present in the "World Model" of the robot.
The "World Model" is updated by the information extracted, analyzed and elaborated by the sensorial part of the robot that will not be treated in this work and is currently under development. In this way it will be possible to monitor parameters like attention or other relevant feature to give a desired degree of autonomy to the robot.
A mobile Android application has been developed as interface between the robot and the therapist in order to select, to start and to stop and to modify

the flow of interactions with the child in a smooth way. In case of emergency or need, the architecture allows anytime the control of the robot through a Bluetooth controller, in a interdicting or non way.
Due to the pandemic situation, it was not possible to access the laboratory to test the whole work: for this reason, a model of the target robot was developed and linked to the architecture and a basic Generator was realized.

1.2 STRUCTURE OF THE THESIS

The document is structured as follows:

- **Chapter2** briefly describes ASD and gives a taste of existing robot used with child affected by autism, comparing some of them. Lastly, it introduces to ROS2 and why it was adopted as software to build the architecture.
- **Chapter3** gives a high level, general description about how the architecture works, which are its components and it explains the developed Script language.
- **Chapter4** describes in detail how rules, the world model, and Generator are intended to work, how they are handled and a guide on how to implement them.
- **Chapter5** treats the simulator, the software used to bind it to the architecture, the joystick interaction and the Android app functioning.
- **Chapter6** concludes the work with results and future works.

2 STATE OF THE ART

2.1 AUTISM SPECTRUM DISORDER

Autism Spectrum Disorder (ASD) is a complex neuro condition characterized by persistent impairments in social interaction, speech and non-verbal communication. This leads to repetitive patterns in behaviours and activities [7]. ASD can be usually diagnosed in childhood during the first two or three years of life; it is estimated, according to the CDC (Centers for Disease Control) that 1 out of 59 children in US have autism and it is more common in boys than girls [8]. Nowadays, the causes that lead to ASD are still unknown, even if some studies and researches let us think that there could be a link with a genetic aspect and with the environment [9]. The only sure thing is that, for now, there is no cure for the ASD. The only therapy consists in providing support in order to improve their quality of life and to reduce symptoms as much as is possible. The most serious problem is that the symptoms of ASD are different in each person, so that the possible therapies must be adapted to each one.

Given this, technology could help to create special therapies, in particular with the adoption of social robots. To do this, it is necessary to know the basis of ASD, so that it could be easier to design appropriate interventions with these robots.

2.2 TRIAD OF IMPAIRMENT

It is possible to divide the impairments of ASD in three different areas – as defined by the Diagnostic and Statistical manual of Mental Disorder (DSM) published in 1994 [10]:

1. *Social interaction*: people with ASD can have different behaviours and it is more clear looking at their social life. Some people are completely indifferent to others around them and they are not able to give affection for this reason. Others want to be surrounded by friends but they are not able to establish a solid friendship because of their wrong behaviours or way to express their feelings [11]. These could be the possible reasons for which the individual wants to stay alone, doing always the same things, without any progress.

2. *Social communication*: communication – verbal or non-verbal – is often a problem for people with ASD. Sometimes they are not able to establish

a discussion with other people because they cannot module their tone in an appropriate way or they are not able to understand the different social contexts so that they feel inadequate and they tend to isolate themselves. For what concerns the non-verbal communication, they must learn to use it because it is not innate for them. The experience is fundamental for these individuals, for this reason it is important to have people around them that can help them to increase their luggage of gestures and body language.

3. *Imagination*: the ASD leads individuals to a reduced ability of imagination and abstract thinking. For this reason, they usually do the same things and every change could be something really stressful for them. A change in their routine could also lead to apathy, in particular when they play or they have to eat. All this causes a difficulty in incidental learning and this could be a real problem not only for children, but also for their parents [12].

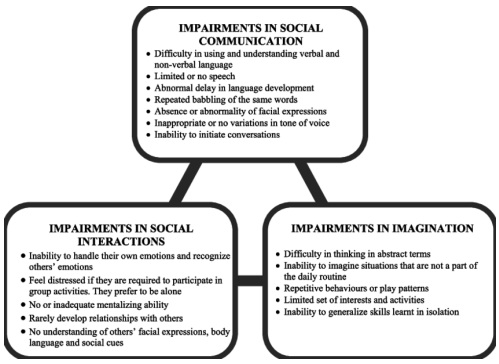


Figure 2.1: Triad of impairments in Autism Spectrum Disorder.

In 2013 has been published the fifth edition of the DSM in which the previous three points have been unified in two topics [7]. This caused the passage from the Triad of Impairments to the Dyad, composed by:

1. *Social communication skills*: this topic represents the union between social interaction and social communication. This is due to the fact that children with ASD have lots of problems in verbal and non-verbal communication and this causes social deficits.

2. *Repetitive patterns of behaviors*: this coincides with the imagination in the Triad of Impairments. Children with ASD usually do the same things, they are scared about changes in every aspect of life (food, social life, and so on).

This brief recap of the main topics of ASD is useful to understand how it is possible to use Social Robots for the therapy.

2.3 THERAPY WITH SOCIAL ROBOTS

Until now, several robots have been implemented to help children with ASD. Some examples are reported here below.

Nao: one of the most used robot for the therapy with children [13]. This robot (see Figure 2.2) has a human shape and is able to interact with the environment sharing information thanks to its sensors (microphones, camera, and touch sensors). It can speak and it is also able to use non-verbal communication in particular thanks to the LEDs in its eyes and body gesture.

Figure 2.2: Nao robot.

Teo: it is a robot designed and implemented in several versions at Politecnico di Milano, with the aim of playing interactive games with children, helping them to learn how to participate in a correct way [14]. It has the shape of 60 cm tall egg (see Figure 2.3) and it is soft so that children can embrace it without problems. It has got LEDs, a base that allows movements, the possibility to speak and emit sounds, distance and pressure sensors to know the position of the child, to get answers to questions, and to distinguish hugs, caresses and other kinds of tactile interaction.

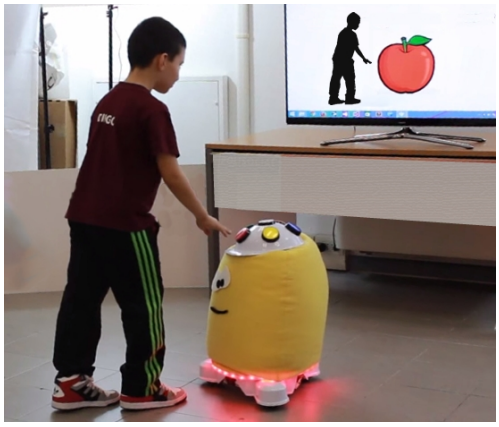


Figure 2.3: Teo robot.

Tito: it is a 60 cm tall robot (see Figure 2.4 made with a soft material. It has two arms (they can be moved up and down) and he can also turn his head. It has also two eyes and a mouth so that it can smile [15]. Tito is able to speak using prerecorded messages and it has also a microphone and a camera. It can be controlled by a wireless remote control.

Figure 2.4: Tito robot.

It is important to underline that children with ASD improve their skills when dealing with Robots rather than dealing with humans. Moreover, a higher level of stimulation achieved better results and positive effects of the therapy.

2.4 DESIGN FEATURES

Here below there is a pointed list with the characteristics that social robots should have in order to be more attractive for children:

1. *Appearance*: considering that children with ASD have a short concentration, the robot should catch their attention with its aspect [15]. For this reason, it should have bright colours but not too gaudy in order to not overstimulate children. For what concerns the size, the robot should not be too tall to avoid to intimidate the child. It should have, more or less, the same size as the child in therapy [20]. Taking into consideration the shape, it has been verified that robots with a human shape are preferred to others [21], however, for some children, the human shape may recall negative experiences with people and trigger undesired behaviors. In is important to underline that robots with a human shape must not be too similar to humans in order to avoid to threaten the child. There are also robots similar to animals or without any similarity with biological species [14]. In this last case, the robot must not be too mechanical because of the risk that the child could be interested only in the robot itself, without seeing it as a real support. In some cases, robots have a cartoon-like shape that can make the children more at ease, since recall them positive experiences with cartoons.

2. *Functionality*: during the therapy it is important to reward the children so that they can understand that what they are learning and doing is correct. The reward can be given with lights or sound that come from the robot, so that the children could feel satisfied about their actions. The most important thing is the movement: social robots must be able to move so that the children can play with it in different ways.

3. *Safety*: it is important to maintain a safe environment for the children during the therapy, because with children with ASD anything could happen. The robot must not have edges or do unexpected movements and there must be protections for its mechanical components so that the child is unable to touch them.

4. *Autonomy*: to avoid idle times – and the boredom of the child – the robot must have a high autonomy during therapy. However, the presence of the therapist during all the time is strictly required: he must have the control of the robot at any moment.

2.5 TARGET BEHAVIOUR

To make the therapy work, the fundamental thing is to provide the correct response to a certain behaviour of the child. To obtain improvements with children, it is important to make the robot able to propose some activities to the child, expecting a certain behaviour from him. These behaviours are:

- *Joint attention*: the act of sharing attention focus [22]. Robots must be used to keep high the attention of the child and it is possible to do this for example when two actors point through eye gaze or hand gestures the same target.
- *Imitation*: it happens when the child learns to copy a specific behaviour from the robot that can be useful, for example, in social life.
- *Eye contact*: this aspect is really important not only to keep the attention of the children, but also to teach them how to talk with people in everyday life. The eye contact is necessary to have a good interaction and to have a mutual comprehension between the child and another person.
- *Self-initiated interactions*: for children with ASD, it could be difficult to ask for something they need to someone and this can be a huge problem. For this reason, the robot provides a reward to the child when it tries to interact, encouraging this behaviour.
- *Turn-taking*: for children with ASD is often difficult to make a conversation without interrupting the person they are talking with. Social Robots may be able to teach them to wait for their turn in the conversation, waiting an answer from a person and saying something after he ended to talk.
- *Triadic interaction*: this interaction involves the child, the robot and a third person, so that the child can be able to interact with someone else that is not himself. This is usually done by exciting eye gaze toward the therapist, then providing a positive reward to the child.
- *Emotion recognition and expression*: the ASD leads to the inability to deliver emotions. Robots can show a set of emotions so that the child can learn how to share them with other people in social life.

2.6 ROBOT OPERATING SYSTEM 2

In the development of our system, we have adopted the ROS 2 framework [16]. The Robot Operating System (ROS) is a set of open source software libraries and tools for building robot applications. Released in 2007, originally it was

developed by Willow Garage for the PR2 robot by Willow Garage [18]. From its beginnings in the academic research community, ROS was implemented not only for a single robot, but with the aim to create a scalable, standard software, and its technology can be applied to several application areas, including industrial robotics [17]. All these new challenges brought ROS to its limits and the need to overcome them brought the developers to re-found it as ROS2, improving it and introducing new features.

Examples of these are [18]:

- Going from single robot to teams of multiple robots.
- Have real-time requirements.
- Enable the development on small embedded platforms.
- Take advantage of the opportunity to improve user-facing APIs.

The choice of just not expanding ROS was guided by the risk associated with the changes: may people relied on it, so it has to remain "as-is" [18]. ROS 2 was build as a set of separate, parallel and inter-operative packages that can be used alongside the legacy ones.

ROS 2 (such as ROS) is based on a message-passing communication, as shown in figure 2.5.

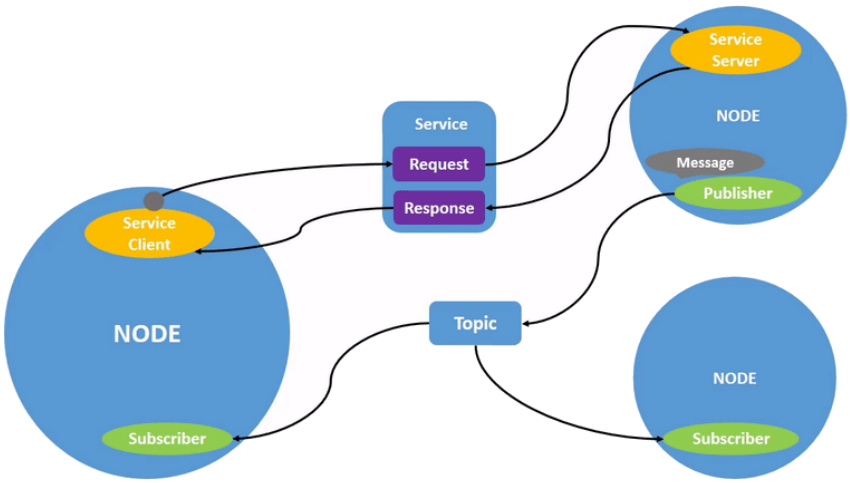


Figure 2.5: ROS2 Graph

Its elements are:

- *Nodes*: each "component" of the system can be designed as a distributed application called node (sensors systems, planners, controllers), that

can share or not the same process with other nodes. They don't require to be run on the same computer, but can be deployed on different units connected via network.

- *Communication Patterns*:

  - *Topic*: data produced by nodes can be shared by publishing them on a topic and collected by others through subscription. Data is shared in form of messages, specialized data structures.
  - *Service*: are the equivalent of remote method invocations.
  - *Action*: are used for long service calls, where action servers publish intermediate results until a goal is achieved.

Moreover, ROS 2 supports the quality of services (QOS) on the DDS [19].

Some Core Differences		
FEATURE	ROS	ROS2
Platforms	(Tested) Ubuntu, (Maintained) other Linux flavors as well as OS X	(Tested&Maintained) Ubuntu, OS X, Windows 10
C++	C++03	C++11 (planned C++14 & C++17)
Python	Python 2	>= Python 3.5
Middleware	Custom	Interface are based on the DDS standard
Threading model	Single-threaded execution or multi-threaded execution	More granular execution models are available and custom executors can be implemented easily
Multiple nodes	Not possible (per process)	Possible (per process)
roslaunch	Defined in XML with very limited capabilities	Written in Python

The work presented in this manuscript has been designed to be maintained in the years to come. For this reason, ROS2 appeared as a natural choice not only to get access to the middleware behind it, but also to allow the incorporation via Generators of edge technologies available to future developers.

3

METHODOLOGY

The whole architecture is shown in Figure 3.1.

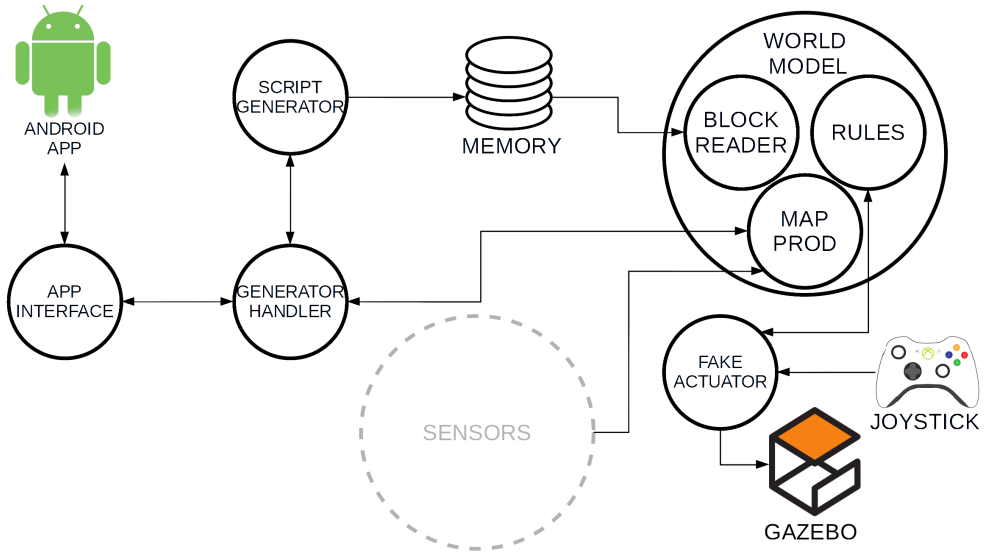


Figure 3.1: Architecture Overview

For sake of clearer exposition, first will be introduced the structure of a Script and how it works and then the various modules build to make them "alive". The language to produce *Scripts* will be presented in 4.4.

3.1 SCRIPT

A *Script* is the embodiment of an activity, defining what the robot has to do and why.

According to the Sense-Think-Act paradigm [25], the relationship between the three basic elements of the robotic paradigm is shown in Figure 3.2: the *Sensors* will capture the relevant aspects of the world, stored and harmonized by the *Map Prod(ucer)*. The map obtained by this last component is used as part of the knowledge base by *Rules* and *Script Generator* to, respectively, modify and to generate a *Script* containing a sequence of elementary actions (read by the *Block Reader* and issued to the actuators).

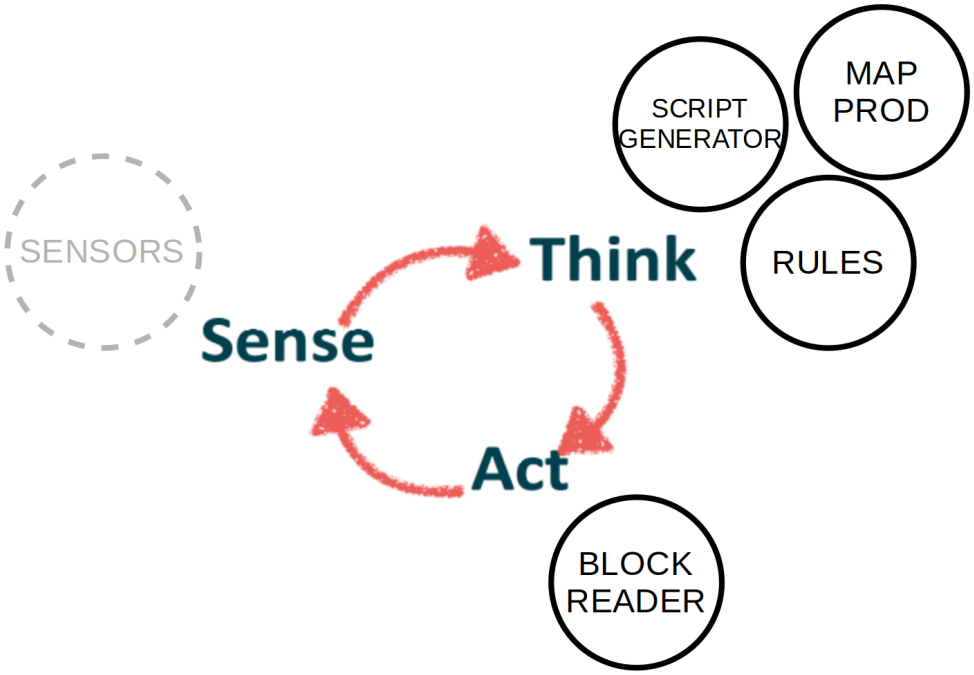


Figure 3.2: Sense Think Act paradigm implementation

The *Script* is a hierarchical structure organized as shown in Figure 3.3

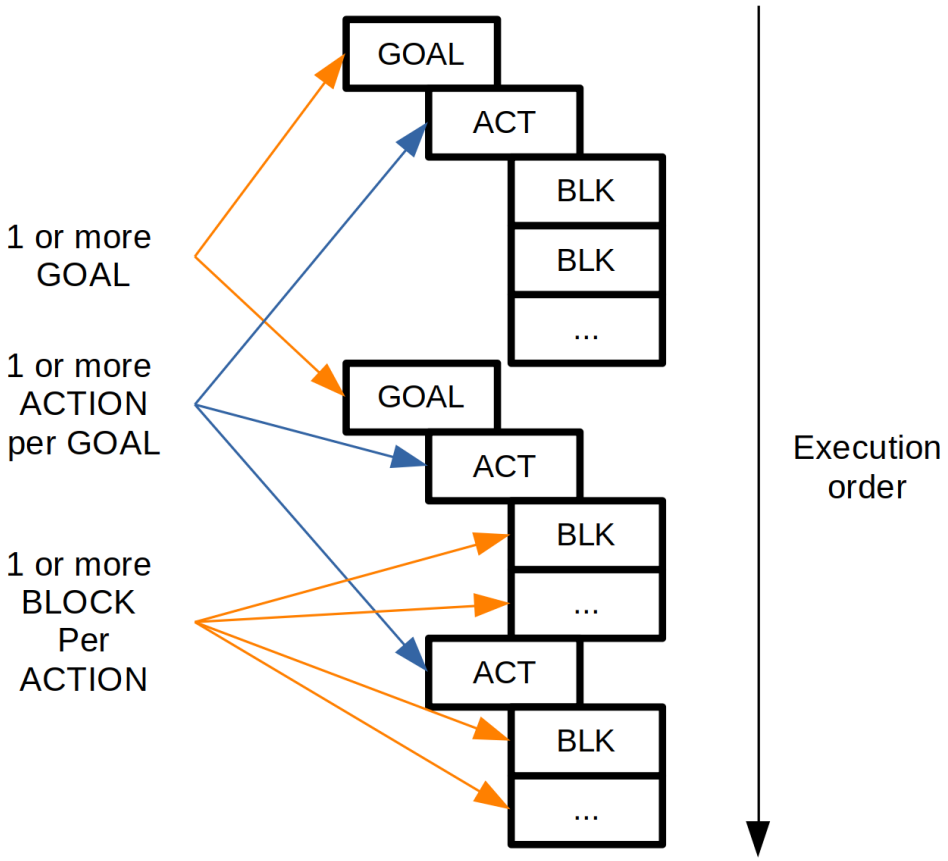


Figure 3.3: Structure of a Script

Script is executed goal by goal, for each goal action by action, for each action block by block in a sequential way (as can be seen in the following of this text).

However, the flow of execution can be manipulated both by *Rules* and by *Blocks*.

A **Goal** is a sub-target to be acquired during the execution, used to split the activity in steps to be performed. These milestones are reported also in the map maintained by the *Map Producer*, and are used to track the progress of the robot.

An **Action** is a subset of the elementary actions that share the same aim or contribution for the success of the interaction.

A **Block** is an atomic brick that constitutes the activity. In detail, it can shaped as one of the following structures:

- *instruction*: as represented in Figure 3.4, it defines what the actuators should do. Read from left to right, it is a sequence of timed atomic commands for each actuator.

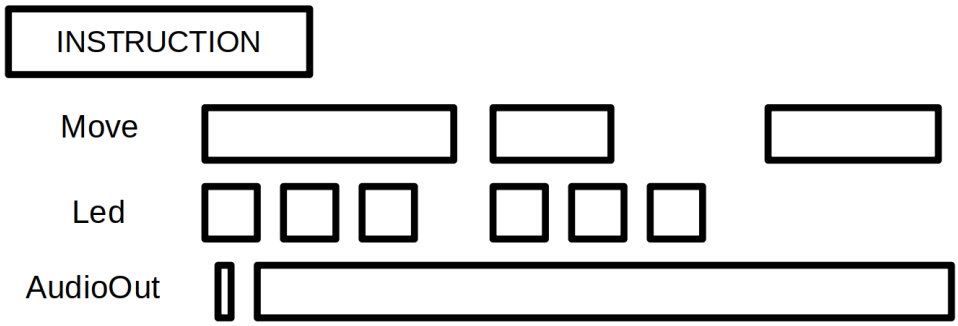


Figure 3.4: Schematic representation of an instruction block

Move(s) are issued to wheel actuators, in the form of:

- *x_speed*: speed along x axis to be applied to the robot, in m/s.
- *y_speed*: speed along y axis to be applied to the robot, in m/s.
- *theta_speed*: speed along theta axis to be applied to the robot, in rad/s.
- *wait_t*: ms to be waited before execute the current block.
- *duration*: ms in which execute the current block.

Once a move command has been consumed, the robot receives an order to stop its movement.
Led(s) are commands issued to the LEDs manager, in the form of:

- *r*: red amount in RGB code (ranges from 0 to 1).
- *g*: green amount in RGB code (ranges from 0 to 1).
- *b*: blue amount in RGB code (ranges from 0 to 1).
- *turnas*: 1 to turn on the leds, 0 to turn them off.
- *wait_t*: ms to be waited before execute the current block.
- *duration*: ms in which execute the current block.

Once a LED commend has been consumed, the robot receives an order to turn off the LEDs.
AudioOut(s) are issued to the speakers, in the form of:

- *track_path*: the path to the audio file to be reproduced.
- *wait_t*: ms to be waited before executing the current block.
- *duration*: ms dedicated to the current block.

Once an AudioOut command has been consumed, the robot receives an order to stop the current track.
An *instruction Block* is considered "consumed" when all its atomic commands have been executed, allowing the progression to the next one.

- *jump*: When reached, *jump block* imposes the evaluation of a condition, wrote in the *Rule Grammar* (see 4.3). Based on the returned truth value, a new *Block* inside the same *Script* will be analyzed (one for True, one for False). This is a way to influence the course of events.
- *Terminus*: A *terminus block* dictates the end of the current *Script*, no matter where it is and how it is reached. The next *Block* to be evaluated is the first *Block* of the Script that follows the current one (eventually idle_activity one).
  Example of how *jump(s)* and *terminus* are reported in Figure 3.5.

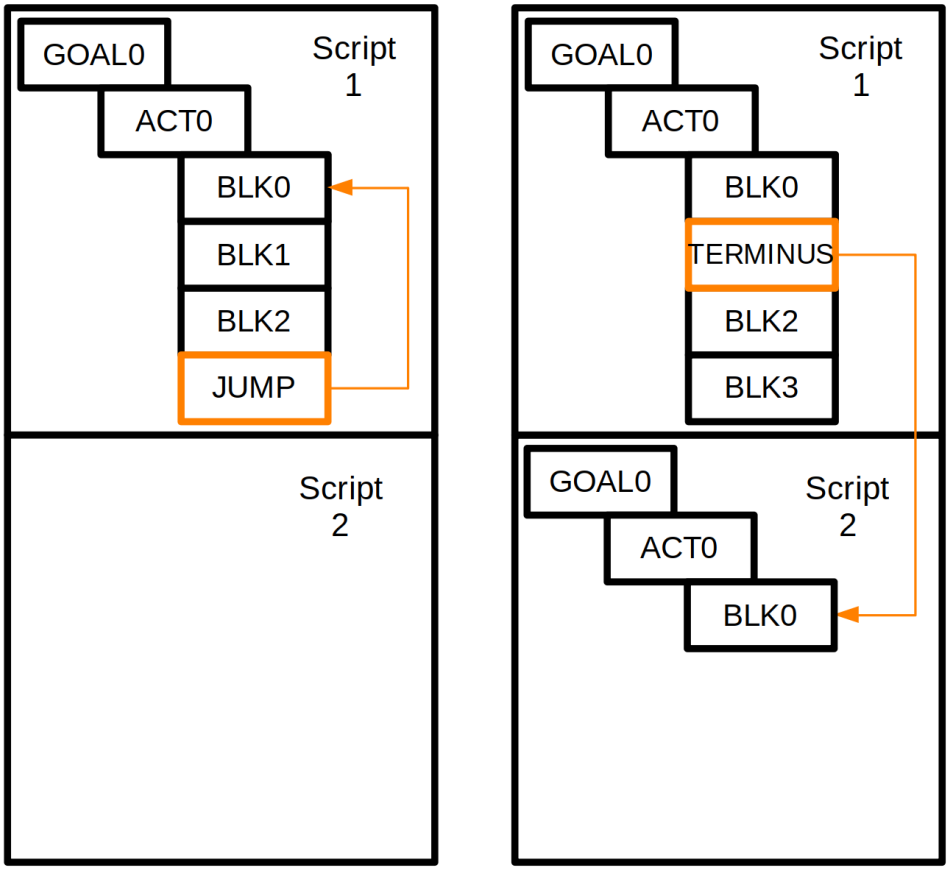


Figure 3.5: Example effects of jump vs. terminus

- *sensor*: They are the stub to allow the interaction with the sensors.

Combining these *Blocks* together it is possible to implement complex activities.

When a *Script* is loaded/unloaded (when it ends), it is responsible to launch and recall the *Rules* associated with it, throught the submission of a set of encoded (ROS2) messages stored, respectively, in *alfarulemessages* and *omegarulemessages* sections.

An utility has been developed to allow the production of a ".json" file containing the *Script*: this will be presented in Chapter 4.

3.2 ARCHITECTURE OVERVIEW

Under the light of how *Scripts* are made and read, it is possible to better understand what wraps them.

- *World Model*:
  - *Block Reader*: this component is what “physically” reads block by block each *Script*. *Scripts* are stored in a LIFO (Last In, First Out) queue: each time one has been consumed, the next will be loaded. A special "idle_activity" (an infinite, passive loop) is always provided in order to never let the queue empty. This embedded routine is provided in order to avoid that the robot, when not stimulated through the submission of *Scripts* from activities, will stall motionless and apathetic.
  - *Rules*: they are implemented as ROS nodes, linked in a "daisy chain way" together with the Map Producer (Figure 3.6).

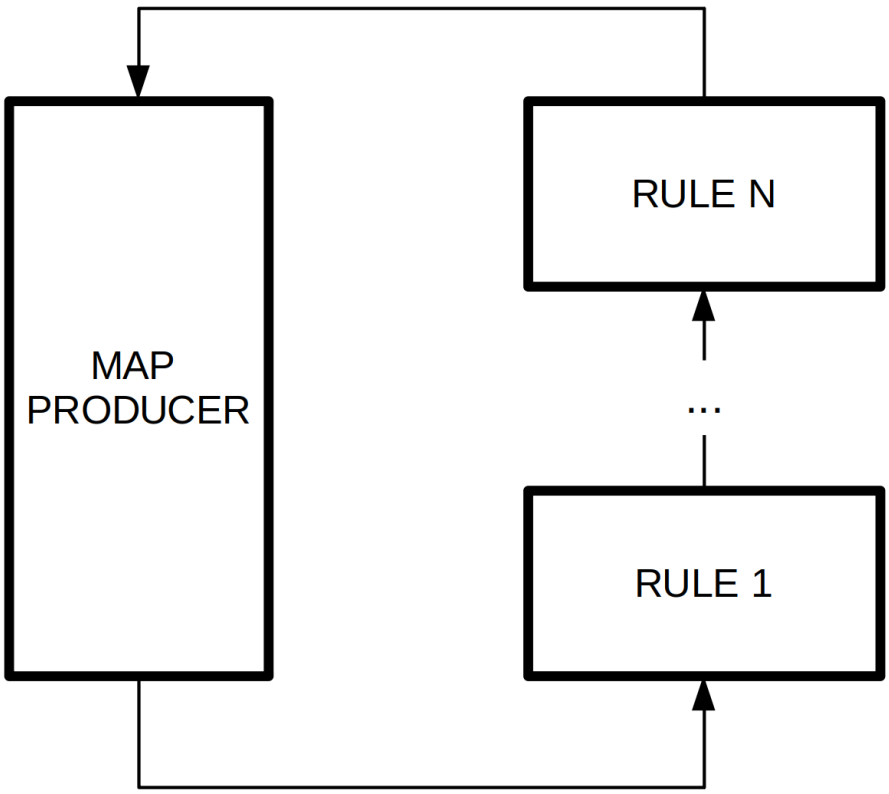


Figure 3.6: Map propagation diagram.

In short, the map is a ROS message passed through a zero copy mechanism from the *Map Producer*, responsible of updating it, to the *Rules*, that use them to their own computation, and back and forth. *Rules* are bounded to the *Scripts* that execute them, being loaded and unloaded following their life cycles. Scripts offer also the possibility to load complex or computationally heavy *Rules* ahead of schedule in a non-blocking manner, that ca be used also by other sibling *Scripts* that will come in future.

  - *Map Producer*: this part is in charge of handling the incoming updates from the Sensor part, merge them with the existing data and return the map to Rules. (Both rules and Map producer will be covered in a deeper way in Chapter 4).

- *Script Generator*: this is a ROS2 node, starting in a quiescent state, that should implement three commands: generate, activate and shutdown. Its role is to produce *Scripts* on demand, returning the address of the ".json" file containing it. *Script Generators* could be freely implemented by the activity designer, allowing also the possibility to intercept sensors

data, storing an internal state or listen to messages produced by their own linked *Rules*.

- *Generator Handler*: this module is in charge of executing the command issued by the user, managing the activation of *Script Generator(s)* and coordinating their outputs in an organic way. It also manages the updates of the LIFO queue of scripts that the *Block Reader* has to process.
- *App Interface*: it offers a socket interface for the *Android App*, transforming the incoming command to ROS2 messages and vice versa.
- *Android App*: this component is an Android Application developed in Java language designated to manage the order of the execution and selection of activities. It will be covered in detail in Chapter 5.
- *Sensors*: these modules are currently not implemented, designed to be covered in future when the application will run on the actual robot. For the moment, a Wizard-Of-Oz approach is used to simulate the desired behaviour.
- *Fake Actuator*: this node will be covered in detail in Chapter 5. Its goal is to consume the *instruction Blocks* issued to it and actuate the model of the robot in the Gazebo Simulation. In the final implementation, it will drive the actual robot.
- *Joystick*: an XBOX360 joystick (the one used for the performed test) is directly connected to the *Fake Actuator*: via this, is possible to move the virtual representation of the robot in a blocking way (freezing the progression of the *instruction block*) or not. A safety lock is implemented, to avoid unintentional commands.
- *Gazebo*: it is a 3D simulator, originally incorporated into ROS and then become stand alone [24]. Due to its bound with the chosen middleware, it was a natural choice to simulate the behavior of the robot. The model realized will be covered in Chapter 5 and it is shown in Figure 3.7.

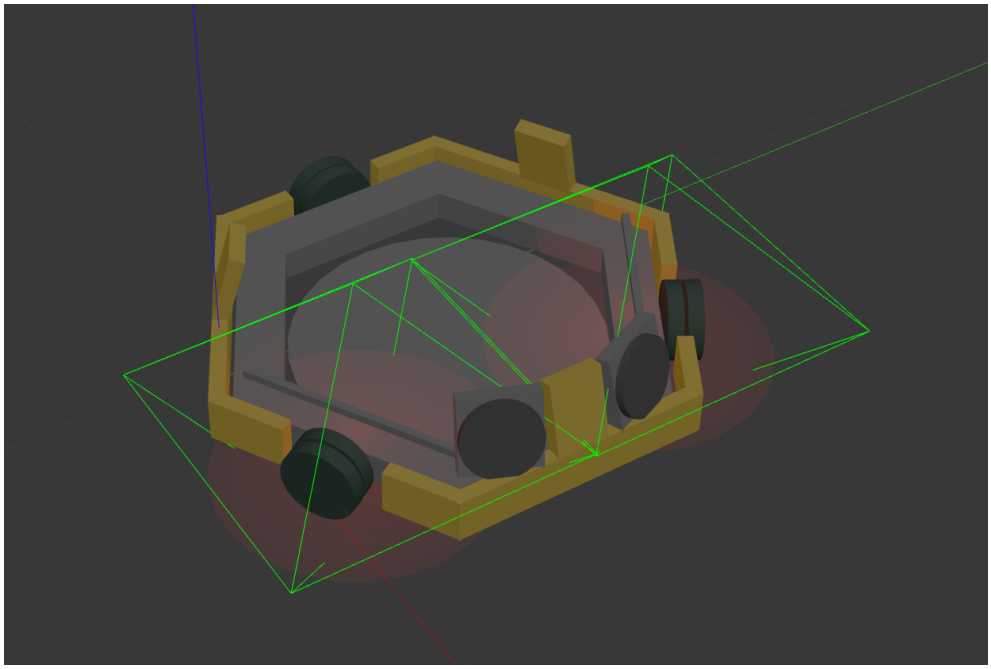


Figure 3.7: Model of the robot realized in Gazebo Simulator

4 IMPLEMENTATION

4.1 MAP MESSAGE STRUCTURE

The *Map* is a ROS2 message (as anticipated in section 3.2) with the following structure.

- *Script Info* field: all the information about the currently executed *Script* are stored here. It keeps track of the current *Goal*, *Action*, *Block* (and its content and completion point). A flag is em-placed in order to alert if the block has been modified by the *Rules* and it needs to be resubmitted to the actuators.
- *Last Update Time* field: it stores the last time the map was updated in ms.
- *Robot Status* field: it commits to memory all the progress of carrying out activities (including their goals), the inputs received by the sensors (namely touch-feedback, audio input-feedback).
  Touch and audio should be managed as the other inputs (camera and sonar). Touch has different positions and types of touch, audio input is still to be considered, but certainly will have a kind of management analogous to the other inputs.
  Lastly, it stores the history of last performed actions on actuators.
- *Camera Status* array field: it gathers together the information available on cameras and their statuses (the target robot is supposed to carry them on board).
- *Sonar Status* array field: it gathers together the information available on sonars and their statuses (the target robot is supposed to carry them on board).
- *Obstacles*, *Humans*, *Objects* array fields: entities in the world can be categorized as Obstacles (fixed and unmovable obstruction), Humans (human beings) and Objects (items with which the robot is supposed to interact). Each of these elements owns its unique attribute fields (freely assignable properties) and fixed features. Each entity is indexed by a fixed unique id, and each incoming update will be redirected upon it. If the id attached to an update doesn't match an existing one, a new entity will be allocated.

A detailed perspective of the map message structure is given in Figure 4.1.

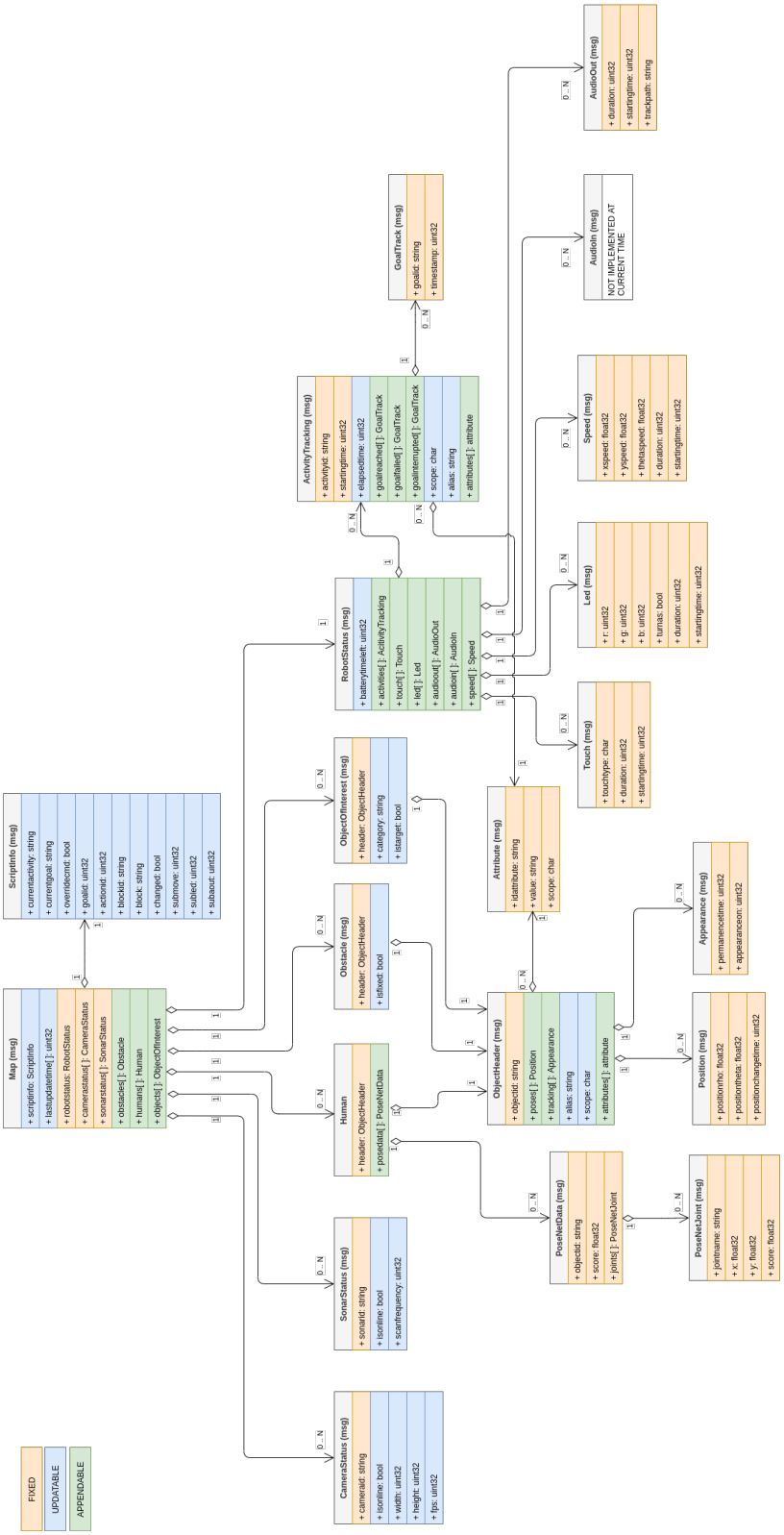


Figure 4.1: Structure of Map Message Class

The color code in 4.1 has the following meaning: orange stands for "fixed" (set upon the creation of the element and immutable), blue stands for "update-able" (so when a new value is supplied for that field, the older one will be discarded) and green stands for "append-able" (all incoming updates will be stored in a circular register).

It is important to mention that the *Script Block* contained in the *Script Info* field is the one that will be actually sent to the actuators. This gives the *Rules* the possibility to alter it to match their desired behaviour.

4.2 LOOP

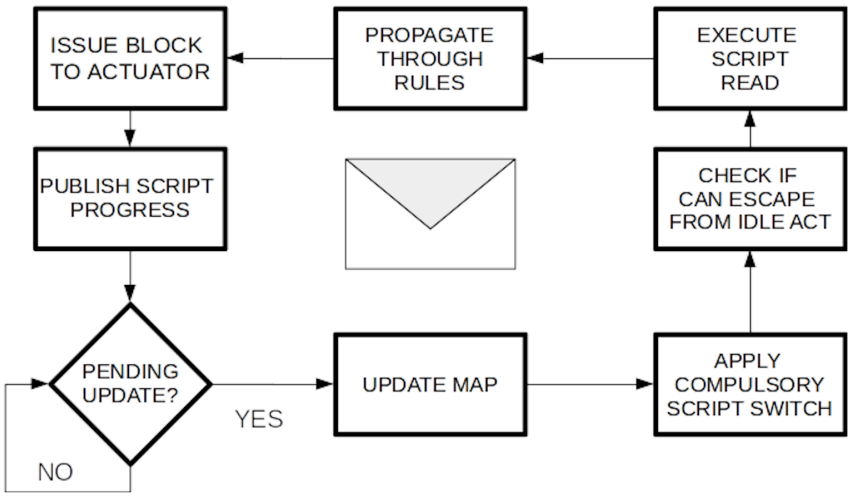


Figure 4.1: Loop Schema

The map propagation can be displayed as in Figure 4.1.

Laying in the same process, *Map Producer* and *Rules* could benefit of the intra-process communication offered by ROS2. Despite what happens in standard ROS messages, where when a message is published its subscribers receives a copy of that message, intra process publish/subscribe connection can result in zero-copy transport of messages when publishing and subscribing with std::unique_ptrs. Exploiting this possibility, the map is passed as a memory address, saving space and increasing the performance [26, 27].

The whole cycle of propagation starts from the top-right block of figure 4.1:

1. **Issue Block To Actuator**: the current *Script Block* contained in the map is sent to the actuators. A flag (*changed*, present in map's *Script Info* field) propagated together with the *Script Block* asserts if it has been modified or not, imposing the rejection of the currently executed one in favor of the new one, executed from the beginning.

2. **Publish Script Progress**: information on the “in the works” *Script Block* are published by the *Map Producer*.
3. **Pending Update?**: the *Map Producer* stalls until nothing in the world (or in the execution of the *Script Block*) has changed.
4. **Update Map**: When an update has been detected, the *Map Producer* consumes all available ones, unlocking the process.
5. **Apply Compulsory Script Switch**: the *Script Info* field in the map is read to check if a *Rules* has raised the *overridecmd* flag. If this happens, the *Block Reader* loads the (*Rule*-setted) *Script* reported in field *currentactivity* at *Goal*, *Action*, *Block* reported in *Script Info* sub-fields *goalid*, *actionid*, *blockid*.
6. **Check If Can Escape From idle Act**: if the *Script Info*'s *currentactivity* sub-field reports that the current activity is the idle one (see 3.2), the *Block Reader* checks if the LIFO queue containing *Scripts* has on top a "non-idle" entry. If it is true, that *Script* will be loaded.
7. **Execute Script Read**: the *Block Reader* first checks if the current *instruction Blocks* has been completed. In this case, it starts to read *Blocks* in a sequential way (executing *jumps*, *terminus* and sending *sensor* messages possibly found) until it lands on a new *instruction Block*. It loads it in the map and proceeds to publish the map.
8. **Propagate Through Rules**: *Rules*, linked in the "daisy chain", on the receiving of the map message elaborate their condition. If it is true, they can freely act on the map and doing what their designer has implemented in them before passing the map to the next *Rule* until the last in the chain passes the control to the *Map Producer* and the loop goes to its next iteration.

4.3 RULES AND THEIR GRAMMAR

Rules are defined by the following elements.

- *name*: Only for debugging purposes, it is used in the logger of the *Rule*.
- *id*: It is the unique id associated to the ROS node of the *Rule*.
- *input topic*: the ROS topic associated to the subscriber of the *Rule*.
- *output topic*: the ROS topic associated to the publisher of the *Rule*.
- *hard-coded condition*: a condition written according to the grammar (described in Section 4.3) that should be verified to allow the *Rule* to

act. The condition can be changed on run-time if the *Rule* itself judges it useful.

The **Rule's Grammar** is a "regular expression" characterized by the following structure:

(
({CLASS #*ID SCP* ?*ID* (**(**?THIS>&*FIELD* OP*x VAL* **|** B: *FUNCT* (*VAL****)**$^{+}$) } **)***
|
({VAR @*ID SCP* F: *FUNCT* (*VAL**)} **)***
|
({ATT ?*ENTITY* $*ID SCP* F: *FUNCT* (*VAL**)} **)***
)*
!
({E: *FUNCT* (*VAL**) } **)*** **|** { TRUE }
Note: ',' is used as separator when using * (repeat the element how many times as desired, even 0) or $^{+}$ (repeat the element how many times as desired, but at least 1).
Blue text stands for "regular expression operators" (**or**, **star** and **plus**), *Red* text stands for free text that can be used by the designer to define meaningful terms, and Black text is fixed and belongs to the grammar.
The meaning of the elements is described here below.

- **CLASS**: this condition requires to find an entity of #*ID* type (where *ID* can be "Human", "Obstacle", "Object", or "Activity") in the map, to assign to it within the scope *SCP* (where *SCP* can be "U", "L", "T", see below) the *alias* ?*ID* (*ID* can be any string without " "/"/"/")" chars, defined by the user) that:
  - Has its *FIELD* field (fixed element of map message, see 4.1) that compared with operator OP *x* (where *x* can be '=', '>', '<') to something (a field or an attribute) belonging to *VAL* (another *alias*) returns **true** value
    OR
  - The function *FUNCT* (a user-defined string associated to an implemented function in the parser of the grammar) of type B, which stands for "the family of functions returning boolean values", with the possibly needed arguments *VAL* returns **true** when it is evaluated on the entity.

  Each condition specified must be respected in order to accomplish the assignment.
  An *alias* is the way the grammar allows to make reference to entities in the world (alias is NOT the identifier of the entity in the map) or **VAR** (variables defined by the grammar, see next). In this way, the sensory

part and the *Rule*-related one are disjoint. To each entity, at the same time only one *alias* could be assigned, and, when more entities match the criteria, the older one will be chosen to break the tie. *Alias* are bound to entities according to their scope, in the following way:

  - "**U**": once bounded, the *alias* will be associated to the entity until the current *Script* remains in execution or a *Rule* removes it.
  - "**T**": once bounded, the *alias* will be associated to the entity until the map message is not returned in the hand of the *Map Producer* (that is when it is passed between rules).
  - "**L**": once bounded, the *alias* will be associated to the entity until the *Rule* that has assigned it through its condition has not passed the map messages to its successor.

- **VAR**: it requires to create a new **VAR** with the id @*ID* (with *ID* defined in the same way as for **CLASS**) with the scope *SCP* (same as for **CLASS**) with the value (a string) returned by the function *FUNCT* (a user-defined string associated with an implemented function in the parser of the grammar) of type F (which stands for "the family of functions returning string values") with the arguments *VAL*. A **VAR** is a support element that can be used within *Rules*.

- **ATT**: it requires to associate to the entity with *alias* ?*ENTITY* an attribute with id $*ID* (defined in the same way as $*ID* of **CLASS**) with the scope *SCP* (same as for **CLASS**) with the value (a string) returned by the function *FUNCT* (a user defined string associated with an implemented function in the parser of the grammar) of type F (which stands for "the family of functions returning string values"). **ATT** are used to bind to entities useful values that are needed during computation, overcoming the limits of the fixed fields present in the map. **ATT** can be only present in the right hand side of an operator in the **CLASS** assignment expression, due to the fact that they must be intended as bounded to the *alias* and not to the id of the entity. However, the user can specify a *FUNCT* to check them freely.

NOTE: It is responsability of who implements the functions to define proper casts of values stored in **VAR** or **ATT** when parsed by them. If used in a comparison with an entity field, they will be casted at the same time of the involved field.
!: is a separator between the left hand side of the rule (where assignments can be made) and the right hand side (the ones that will be computed in order to return the truth value of the whole condition). The right hand side is a conjunction (AND) of truth values returned by {E: *FUNCT* (*VAL**) }. Each of them is a function of type E, which stands for "the family of functions

depicted to evaluate the truth value of a condition returning boolean values") with the arguments *VAL*.
If the keyword **TRUE** is reached, it imposes the truth value of the condition as **true**, no matter what follows (can be used as stand alone to always activate the *Rule*).

Rules **must be read** as "there exist *alias* AND exist *alias* ... SO THAT *function* is true ... AND *function* is true ...": if an *alias*/**ATT** assignment fails, the truth function is automatically set as **false** and the rest of the condition (including the following assignments) will not be evaluated (it violates the "exist" condition), although keeping valid the preceding ones.

4.4 JSON-UTILITY FOR SCRIPTS

A *Script* is implemented as a "*.json*" file like the one in the following example.

Listing 4.1: "idle_activity.json"

```
{
  "activityname": "test_act",
  "alfarulemessages": [
    {
      "rulecode": 0,
      "id": "kick_starter",
      "intopic": "",
      "outtopic": "",
      "action": 67,
      "mutexcmd": 85
    }
  ],
  "omegarulemessages": [
    {
      "rulecode": 0,
      "id": "kick_starter",
      "intopic": "",
      "outtopic": "",
      "action": 82,
      "mutexcmd": 76
    }
  ],
  "goal0": {
    "goalid": "test_act_goal",
    "act0": {
      "blk0": {
```

```
        "type": "instruction",
        "flagged": "flag",
        "move": [
          {
            "v_x": "10",
            "v_y": "0",
            "v_theta": "0",
            "wait_t": 1000,
            "duration": 1000
          },
          {
            "v_x": "0",
            "v_y": "10",
            "v_theta": "0",
            "wait_t": 1000,
            "duration": 1000
          },
          {
            "v_x": "0",
            "v_y": "-10",
            "v_theta": "0",
            "wait_t": 1000,
            "duration": 1000
          },
          {
            "v_x": "-10",
            "v_y": "0",
            "v_theta": "0",
            "wait_t": 1000,
            "duration": 1000
          },
          {
            "v_x": "0",
            "v_y": "0",
            "v_theta": "50",
            "wait_t": 1000,
            "duration": 1000
          },
          {
            "v_x": "0",
            "v_y": "0",
            "v_theta": "-50",
            "wait_t": 1000,
```

```
            "duration": 1000
          }
        ],
        "led": [
          {
            "r": "1",
            "g": "0",
            "b": "0",
            "turnas": 1,
            "wait_t": 0,
            "duration": 2000
          },
          {
            "r": "0",
            "g": "1",
            "b": "0",
            "turnas": 1,
            "wait_t": 0,
            "duration": 2000
          },
          {
            "r": "0",
            "g": "0",
            "b": "1",
            "turnas": 1,
            "wait_t": 0,
            "duration": 2000
          },
          {
            "r": "1",
            "g": "0",
            "b": "0",
            "turnas": 1,
            "wait_t": 0,
            "duration": 2000
          },
          {
            "r": "0",
            "g": "1",
            "b": "0",
            "turnas": 1,
            "wait_t": 0,
            "duration": 2000
```

```
        },
        {
          "r": "0",
          "g": "0",
          "b": "1",
          "turnas": 1,
          "wait_t": 0,
          "duration": 2000
        }
      ],
      "audioout": [
        {
          "track_path": "Silence",
          "wait_t": 1,
          "duration": 1
        }
      ]
    },
    "blk1": {
      "type": "instruction",
      "flagged": "flag",
      "move": [
        {
          "v_x": "0",
          "v_y": "0",
          "v_theta": "0",
          "wait_t": 1000,
          "duration": 1000
        }
      ],
      "led": [
        {
          "r": "1",
          "g": "1",
          "b": "1",
          "turnas": 1,
          "wait_t": 1000,
          "duration": 1000
        }
      ],
      "audioout": [
        {
```

```
            "track_path": "plugins/audio_control/songs/
                sounds_Excited_R2D2.wav",
            "wait_t": 0,
            "duration": 4000
          }
        ]
      },
      "blk2": {
        "type": "jump",
        "toparse": "! {TRUE}",
        "truejump": "goal0/act0/blk4",
        "falsejump": ""
      },
      "blk3": {
        "type": "jump",
        "toparse": "! {TRUE}",
        "truejump": "goal0/act0/blk0",
        "falsejump": ""
      },
      "blk4": {
        "type": "terminus"
      }
    }
  }
}
```

However, letting the users write the *Script* could led to *"have a tough nut to crack"*: for this reason, a parser is put at the service of the user.
It takes a string with the following shape.
[*activityname*]
([alfa:*rulecode*, *ID*, *intopic*, *outtopic*, *action*, *mutexcmd*]**)*** % *Alpha_Message*%
([omega:*rulecode*, *ID*, *intopic*, *outtopic*, *action*, *mutexcmd*]**)*** % *Omega_Message*%
(
([goal:*goalid*]**)**$^{+}$% *New_Goal*%
([action]**)**$^{+}$% *New_Action*
(
[b#*label*#blk_inst:MV**(**(*move*)**)***)LD**(***led***)***)AO**(***audio***)***)]
|
[b#*label*#blk_terminus]
|
[b#*label*#blk_jump:CND(*condition*)TRJMP(*label*)FLSJMP(*label*)]
|
[b#*label*#blk_sens: *To Be Defined in Future*]

)+% *New_Block*%
)+
Elements in % *Green* % are comments and will be ignored during parsing. The *tokens* will be described here below, divided by % *Elements they belong to* %.
activityname is a string containing a reference name to be assigned to the current *Script* (used in the map message to instantiate a new ActivityTracking). *Alfa_Message* are messages used to set the *Rules* in a "proper way" when the *Script* is loaded: this mean that when the *Script* must creates its "daisy-chain" of *Rules* to work in the designed way. *Omega_Message* are messages used to restore the *Rules* in a safe, steady way when the *Script* is unloaded.
Going into the details:

- *Alfa_Message* is a string in the form of:
  "alfa:*rulecode*, *ID*, *intopic*, *outtopic*, *action*, *mutexcmd*"
  where:
  - *rulecode*: the corresponding code of the rule as defined in the *World Model* parser.
  - *ID*: the string that will be associated to the *name* field of the *Rule*.
  - *intopic*: the input ROS topic (see Section 2.6) of the *Rule*.
  - *outtopic*: the output ROS topic (see Section 2.6) of the *Rule*.
  - *action*: is a char and can be, referred to the rule called in the message:
    * A -> activate and add the rule to the active ones
    * R -> remove the rule
    * D -> activate and delay rule for a future usage
    * B -> bring back to the active ones the delayed rule
    * C -> recover a removed rule and put it among the active ones
  - *mutexcmd*: is a command char that blocks the propagation of the map message when the *daisy chain* of *Rules* is not ready. It can be:
    * L -> lock the map message propagation & issue this message
    * M -> issue this message
    * U -> issue this message & unlock the map message propagation

    To all *Alfa_Message* must be assigned 'M' except to the last one, where 'U' must be assigned. 'L' here **MUST** not be used. This command is in place to avoid that the map message will be lost due to operations on *Rules*.

- *Omega_Message* is a string with the shape describede here below.
  "omega:*rulecode*, *ID*, *intopic*, *outtopic*, *action*, *mutexcmd*"
  where:
  - *rulecode*: the corresponding code of the rule, as defined in the *World Model* parser.
  - *ID*: the string that will be associated to the *name* field of the *Rule*.
  - *intopic*: the input topic of the *Rule*.
  - *outtopic*: the output topic of the *Rule*.
  - *action*: is a char and can be, referred to the rule called in the message:
    * A -> activate and dd the rule to the active ones
    * R -> remove the rule
    * D -> activate and delay rule for a future usage
    * B -> bring back to the active ones the delayed rule
    * C -> recover a removed rule and put it in the active ones
  - *mutexcmd*: is a command char that blocks the propagation of the map message when the *daisy chain* of *Rules* is not ready. It can be:
    * L -> lock the map message propagation & issue this message
    * M -> issue this message
    * U -> issue this message & unlock the map message propagation

    To all *Omega_Message* must be assigned 'M' except to the first one, where 'L' must be assigned. 'U' here **MUST** not be used.

- *New_Goal*: is a string in the form of:
  "goal:*goalid*"
  It creates a new goal with ID *goalid* when found and assigns to it all the following actions until a new command of the same type is found.

- *New_Action*: is a string in the form of:
  "action"
  It creates a new action when found and assigns to it all the following blocks until a new command of the same type is found.

- *New_Block*: is a string in one of the following forms:

  - "b#*label*#blk_inst:MV((*move*)*)LD(*led*)*)AO(*audio*)*)"
    Producing an *instruction Block*, where:

    * *move* is a string in the form of:
      "*v_x*, *v_y*, *v_theta*, *wait_t*, *duration*"
      Containing the values to fill the atomic move.
    * *led* is a string in the form of:
      "*r*, *g*, *b*, *turnas*, *wait_t*, *duration*"
      Containing the values to fill the atomic led command.
    * *audio* is a string in the form of:
      "*track_path*, *wait_t*, *duration*"
      Containing the values to fill the audio order.

  – "b#*label*#blk_terminus"
    Producing an *terminus Block*.
  – "b#*label*#blk_sensor"
    Producing a *sensor Block*. It will be defined together with the sensor messages.
  – "b#*label*#blk_jump:CND(*condition*)TRJMP(*label*)FLSJMP(*label*)"
    Producing an *jump Block*, where
    * *condition* is a string in the Rule Grammar.
    * *label* are the labels of blocks, respectively for true jump and false jump.

  label(s) can be left empty if not needed.

NOTE: Black elements are mandatory, even if a field could be left empty. An example is:

```
[example_act]
[alfa:16,my_rule,"\myinputtopic","\myouttputtopic",A,M]
[alfa:0,kick_starter,"","",R,U]
[omega:0,kick_starter,"","",C,L]
[goal:blinking_and_stop]
[action]
[b##blk_inst:MV()LD("0, 0, 1, 1, 0, 100", "0, 0, 1, 1, 0, 100")AO()]
```

It will add *Rule* my_rule when the *Script* "example_act" is loaded, removing the "kick_starter" one.
When the *Script* is over, it will restore the previous situation doing the inverse of what it has done before.
The activity makes the robot's leds blink two time in blue and then it ends.

4.5 UPDATE MECHANISM

Updates are issued by the *Sensors* through MapUpdate message with the following fields:

- **objectid**: it is a string in the form *CLASS**ID*. *CLASS* is where to search the target entity with the name *ID*.
- **attributename**: is the name of the attribute to be updated.
- **policy**: can be *"R"* or *"A"* (not both options are always available given specified **attributename**). *"R"* deletes all non-already-processed updates that were issued before it, *"A"* appends the current udpate to the other non-already-processed ones.
- **jsonmsg**: is a string reporting the contents of the update.

To have a complete vision of the update messages, check the table 4.1.

CLASS	ATTNAME	POLICY	JSONMSG
Camera/ID	isonline	'R'	"s_true" or "s_false"
	width	'R'	"*int number*"
	height	'R'	"*int number*"
	fps	'R'	"*int number*"
Sonar/ID	isonline	'R'	"s_true" or "s_false"
	scanfrequency	'R'	"*int number*"
Human/ID	poses	'R', 'A	"{"positionrho": *float number*, "positiontheta": *float number*, "positionchangetime": *int number*}"
	tracking	'R', 'A'	"{"permanencetime": *int number*, "appearanceon": *int number*}"
Obstacle/ID	poses	'R', 'A	"{"positionrho": *float number*, "positiontheta": *float number*, "positionchangetime": *int number*}"
	tracking	'R', 'A'	"{"permanencetime": *int number*, "appearanceon": *int number*}"
	isfixed	'R'	"s_true" or "s_false"
Object/ID	poses	'R', 'A	"{"positionrho": *float number*, "positiontheta": *float number*, "positionchangetime": *int number*}"
	tracking	'R', 'A'	"{"permanencetime": *int number*, "appearanceon": *int number*}"
	istarget	'R'	"s_true" or "s_false"
	category	'R'	"*string*"
RobotStatus	batterytimeleft	'R'	"*int number*"
	touch	'R', 'A'	"{"touchtype": *char*, "duration": *int number*, "startingtime": *int number*}"
	led	'R', 'A'	"{"r": *int number*, "g": *int number*, "b": *int number*, "frequency": *int number*, "duration": *int number*, "startingtime": *int number*}"
	audioout	'R', 'A'	"{"duration": *int number*, "startingtime": *int number*, "trackpath": "*string*"}"
	audioin	'R', 'A'	"TBD"
	speed	'R', 'A'	"{"xspeed": *float number*, "yspeed": *float number*, "thetaspeed": *float number*, "duration": *int number*, "startingtime": 123}"

Table 4.1: Update Guide

CLASS	ATTNAME	POLICY	JSONMSG
RobotStatus/ID	elapsedtime	'R'	"*int number*"
	goalreached	'R', 'A'	"{"goalid": "*string*", "timestamp": *int number*}"
	goalfailed	'R', 'A'	"{"goalid": "*string*", "timestamp": *int number*}"
	goalinterrupted	'R', 'A'	"{"goalid": "*string*", "timestamp": *int number*}"
	startingtime	'R'	"*int number*"
Script	submove	'R'	"*int number*"
	subled	'R'	"*int number*"
	subaout	'R'	"*int number*"
	script_vect	'R'	"{"restart": "s_true" or "s_false", "update": ["*string*", ..., "*string*"]}"

Table 4.2: Update Guide Cont'd

It is important to notice that the update of *Script* LIFO is done as shown in the last row of table 4.2: the *string*(s) passed in the "update" field of "script_vect" (each of them should be a path) will be treated as how the LIFO queue should be.

5 SIMULATION

5.1 ANDROID APP

In this section, the *Android Application* will be presented together with the sequence of actions, triggered through its GUI, that will cause the *Script* to be produced and loaded. The aim of the Android Application is to allow a user-friendly control of the sequence of activities performed by the robot. The application is connected through WIFI connection to the robot and is thought up to be run on a phone or tablet under the control of an operator.
The main activity screen of the *Android Application* is shown in Figure 5.1.

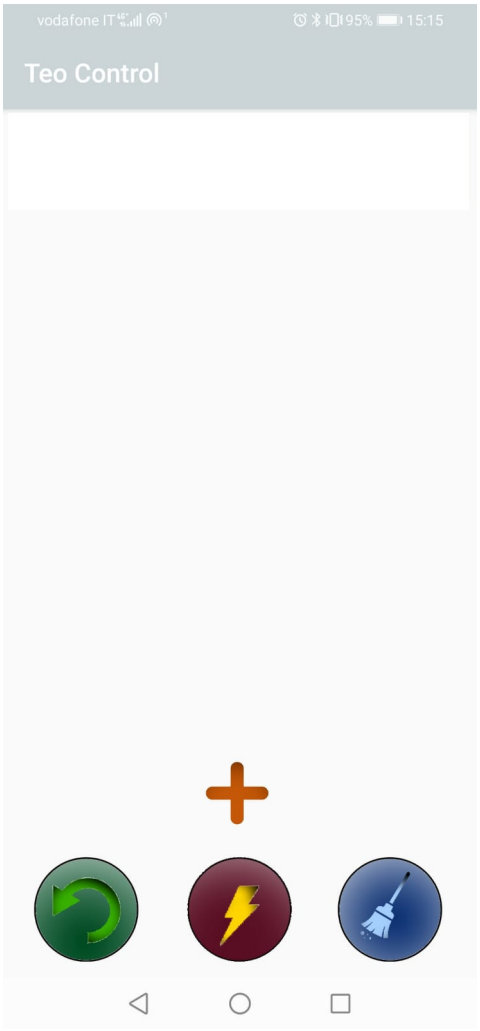


Figure 5.1: *Android App*: Main Activity

It offers the following options:

- **Activity Command Button**: it is a "shape-shifting" button that can take one of the following aspects:
  - **Get Ready Button**:

Figure 5.2: Get Ready Button

This is the default status: each time a *Script Generator* associated with a selected activity falls into the quiescent state (either because it has just been added or it is set so by a *Rule*), the **Activity Command Button** switches to this form. When pressed, it asks to all *Script Generators* to activate themselves (calling their "activate" command) and to produce a *Script* (calling their "generate" command).

– **Start Button**:

Figure 5.3: Start Button

When all *Script Generator*s are ready and have generated their *Script*, the **Activity Command Button** changes to this shape: when pressed, it orders the Script Generator Handler to fill the *Script* LIFO queue of the *Block Generator* (which will so leave the *idle activity*) with the *Scripts* of all non-"non yet completed" activities. Once done this, it switches to the **Stop** form. Each time an activity is completed, it updates its status (shown with different color, see next). Lastly, during execution, the current activity is highlighted in an unique color.

– **Stop Button**:

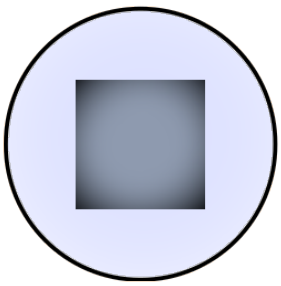


Figure 5.4: Stop Button

It allows the possibility to interrupt the current flow of activities in execution, setting the current one as "not yet completed" and keeping intact the others. Once done this, it switches back to the **Start** state.

- **Loading Button**:

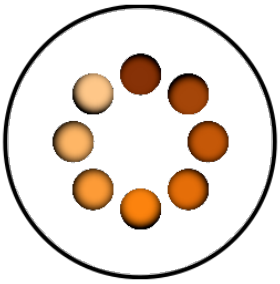


Figure 5.5: Loading Button

It is an inert button displayed doing transition between two of the following states.

- **Restart Button**:

Figure 5.6: Restart Button

If the current **Activity Command Button**'s state is not **Start**, it allows to re-set all activities to "not yet completed".

- **Clear Button**:

Figure 5.7: Clear Button

If the current state of the **Activity Command Button** is not **Start**, it allows to clear the list of selected activities, calling the shutdown method of each of them.

- **Plus Button**:

Figure 5.8: Plus Button

It allows to add a new activity to the list of selected ones. When pressed, the following screen will be opened (5.9):

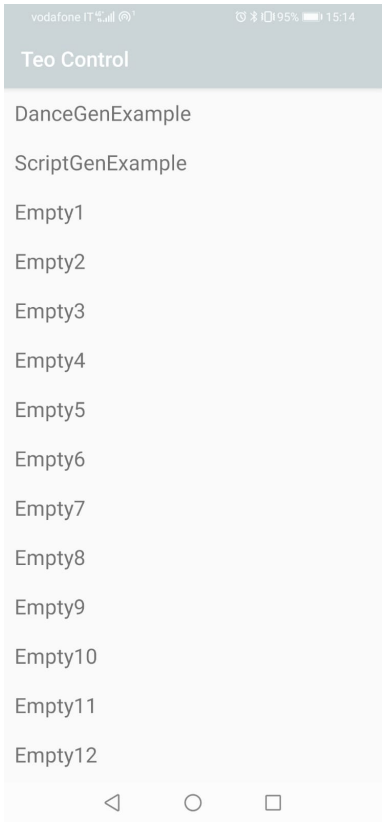


Figure 5.9: Selection Activity Screen

When an entry has been selected, it will be added to the list of selected activities unless it is not a duplicate (activities may be designed to share their parent generator).

Option screens can be displayed to set other features related to the chosen activity, as shown in Figure 5.10.

Figure 5.10: Example for the Test Dance Activity

As just displayed, activities in the selected list can be in one of the three following states:

- **current**:

Figure 5.11: Example of an activity in current state

It is the current activity in execution.

- **completed**:

Figure 5.12: Example of a completed activity

It is an activity that has been already completed. It will not be executed again until the **Restart** button is not pressed.

- **not-yet-completed**:

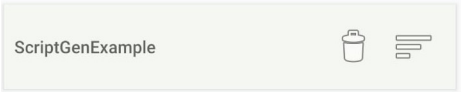


Figure 5.13: Example of a not-yet-executed activity

It is an activity that has to be completed. They will be executed in the specified order (from top to bottom).

Each element in the list of selected activities (an example is shown in Figure 5.13) offers two buttons: the "trash bin", which allows to delete the current activity from the list of selected ones, calling the shutdown method of its generator, or the button "with stripes", which allows drag-and-drop gestures to change the order of activities in the list. Example of a screenshot is reported in Figure 5.14.

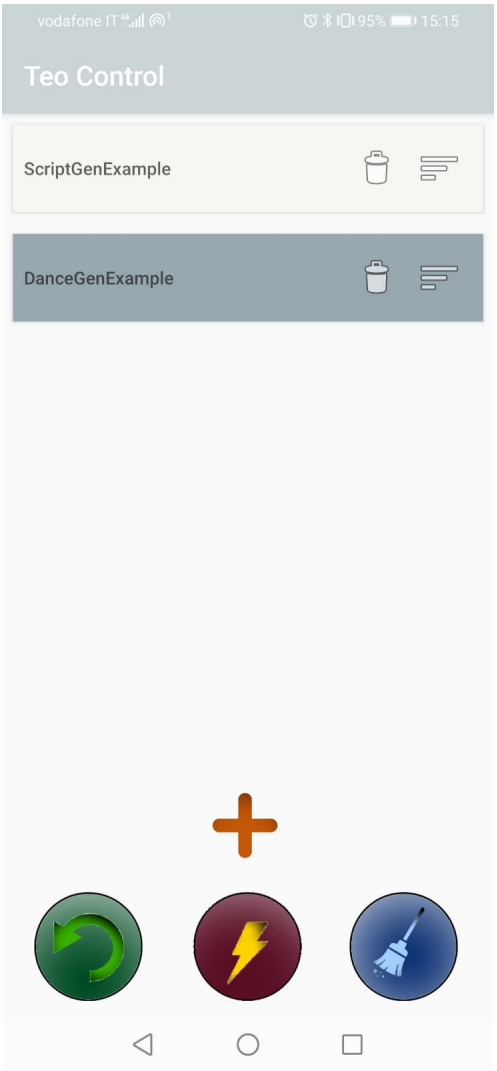


Figure 5.14: Example of screenshot

The *Android App* is connected through Socket [33] to an adapter (*App Interface*) that is in charge of translating the incoming TCP messages into ROS messages.

5.2 GAZEBO MODEL

The robot used in the simulator is designed taking inspiration from Teo/Oimi [5], as shown in Figure 5.16:

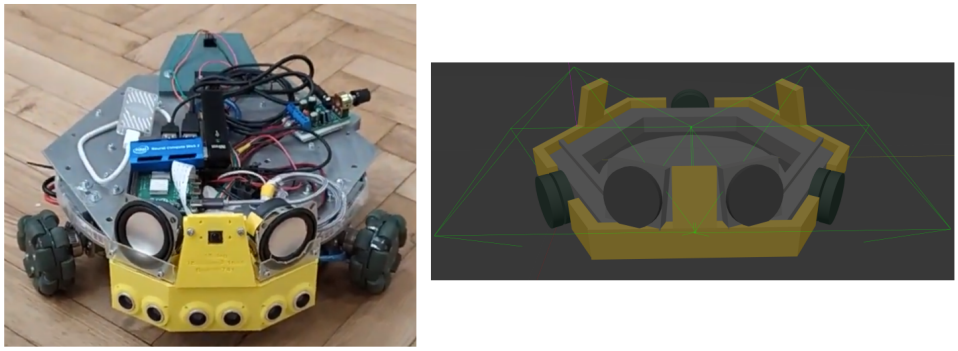


Figure 5.15: The real robot and the modelled one

It has been modelled as an omni-directional robot, equipped with Swedish wheels [29]; for sake of simplicity the model is characterized by a zero-friction constraint on each contact point (on the "wheels") and moved by applying force on its base (forces are applied for an infinitesimal time and then cancelled to avoid drifts or unwanted accelerations). Two LED strips are includes, implemented by Flashlight Plugin [30].

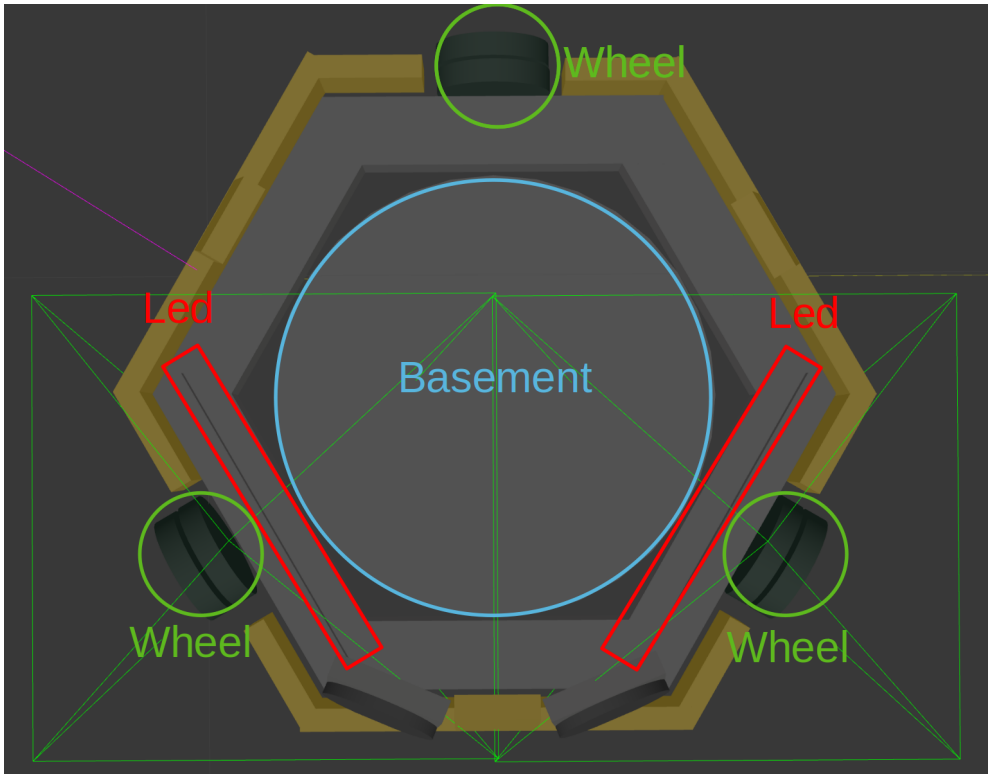


Figure 5.16: Highlighted parts of the model

To control the robot, three custom *Gazebo Plugins* [31] have been implemented:

- *audio_control*: implemented with SDL library [32], it allows the execution of an audio track upon the receiving of a ROS message. It also allows to stop, pause and restart the selected sound.
- *led_control*: it allows the turning on/off (and the setting of their colors) of the two led strips upon the receiving of a ROS message.
- *speed_control*: it allows the application of a force/torque to the basement of the robot upon the receiving of a ROS message, containing values to be applied on x, y and theta axes.

The simulator allows a formal verification of the architecture, both controlled through a test bench ROS2 node, both controlled by the *Android Application*.

5.3 FAKE ACTUATOR

A "fake actuator" has been implemented to capture the messages containing the *Script Block* (as shown in 4.2) and execute them, chunk by chunk and publishing ROS messages for the *Gazebo Plugins*.
The XBOX360 controller (as displayed in Figure 3.1) communicates directly with it, with key bounded as follow (figure 5.17):

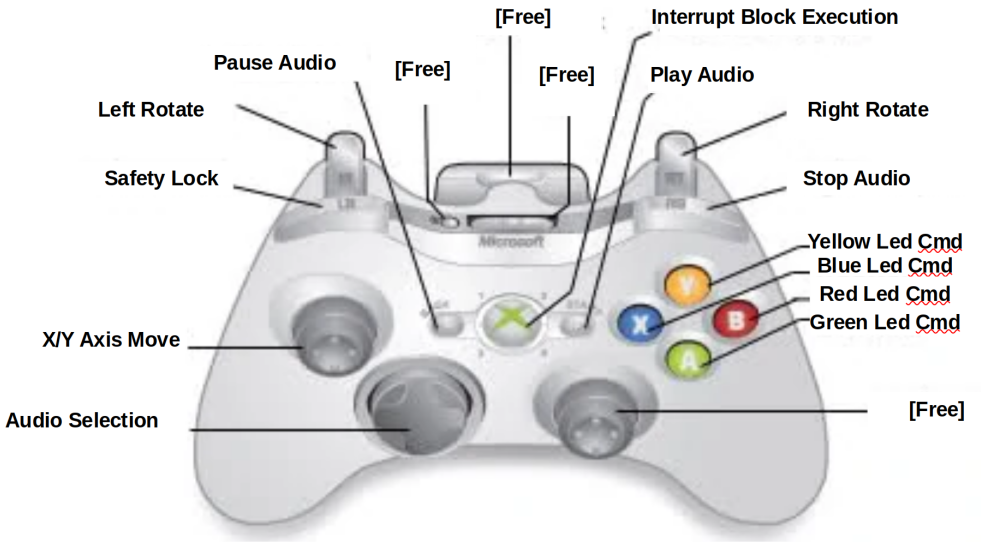


Figure 5.17: XBOX360 Controller and Related Key Bounds

Commands issued by the Joystick are ignored until the "**Safety Lock**" is not kept pressed. They will be executed together with the ones from the *Script Block*, overriding among themselves, cancelling the current one and standing out itself).

If the "**Interrupt Block Execution**" button is pressed, the execution of *Script Block* will be frozen until the next engagement.

6

CONCLUSIONS

6.1 CONCLUSIONS

The result of this work is a functional, modular architecture. It allows to add, to remove, to re-order, to start/stop/change activities in a simple way. The level of isolation and independence given to the embryo-activity used for the tests allows it to be copied and raised easily to match the designer's need. Furthermore, being it a fully functional ROS2 node, there is the possibility (as examples) to cherry-pick already implemented algorithms, to access to technologies as Cloud-Computing (transforming the *Script Generator* in a mere receiver of operations carried out on a more performing hardware) or to create intercommunication process between different activities. The map message can be extended if new sensors are added, but requests linked to a more expressive representations of entities can be satisfied by the use of their attributes field without any effort. Lastly and unintentionally, this architecture is born hardware independent, allowing it to be deployed on different machines with different capabilities and utilities.
This project ends with the hope of being used, crumpled, fixed, and expanded in future. While, at the moment, it remains only a skeleton, the intention is to see it running with real activities and real children.
This pandemic has brought us far away and the persons target of this work had lost what matters the most: the right to play and the right to be taken by the hand and helped to overcome a bit what challenges them everyday.
Quoting Charlie Chaplin [34] ".. Let us fight for a world of reason, a world where science and progress will lead to all men's happiness...": this is what Robotics and Artificial Intelligence should bring to us, even if this is not "trendy" or will lead to big earnings.
Even if not tangibly implemented the target robot for the architecture was conceived to be as cheap as possible, to become a part of the everyday life of these children.
Nevertheless, the flexibility offered by the architecture allows its usage also outside its cradle: empowered by the ROS2 middleware strength, with the right actuators under its command a wide range of tasks can be carried out. Both educational (thought the coding in the *Script Language*), and other goals for social robots could be achieved with the right activities implemented.

6.2 FUTURE WORK

The most urgent topic to deal with is to **deploy the architecture** on a real robot: the originally target platform is a Nvidia Jetson Nano [35] board (Figure 6.1).

Figure 6.1: Nvidia Jetson Nano Board

To do this, both **actuator drivers** and the **sensory part** should be implemented to interface the real robot to the real world.

Once done this, it's mandatory to **develop activities** to give a meaning to the work done.

Stubs for incoming audios and emotional reactions are left open: the possibility to do, even a simplified, **speech recognition** could enhance the interaction capabilities of the system, and **expressing custom emotions**, in particular with children affected by ASD, may create for them a more comfortable and effective participation.

Lastly, the target board (fig6.1) was selected to access to NVIDIA CUDA Cores [36]: this choice was made to allow the **usage of Artificial Intelligence** algorithms to improve the engagement with the robot and effective autonomous activities.

BIBLIOGRAPHY

[1] *What Is Autism Spectrum Disorder?* https://www.psychiatry.org/patients-families/autism/what-is-autism-spectrum-disorder (cit. on pp. xiii, xiv, 1).

[2] Sandra Bedaf, Gert Jan Gelderblom, and Luc de Witte. "Overview and Categorization of Robots Supporting Independent Living of Elderly People: What Activities Do They Support and How Far Have They Developed." In: *Assistive Technology* 27.2 (2015). PMID: 26132353, pp. 88–100. eprint: https://doi.org/10.1080/10400435.2014.978916 (cit. on pp. xiii, xiv).

[3] Daniel J. Ricks and Mark B. Colton. "Trends and considerations in robot-assisted autism therapy." In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 4354–4359 (cit. on pp. xiii, xiv, 1).

[4] L. Bonetti. *Design and Implementation of an Actor Robot for a Theatrical Play*. https://www.youtube.com/watch?v=v7QM1OyLGcA. 2019 (cit. on p. 1).

[5] D.M. Sortino. *Development of a Computer Vision based Social Robot for interaction and joint attention task delivery with Autism Spectrum Disorder children*. https://www.youtube.com/watch?v=sNeZHixv6Es&t=6s. 2020 (cit. on pp. 1, 47).

[6] A.S. Martın. *An Approach to Attract and Manitain Attention of People with Interactive Mobile Robots*. 2019 (cit. on p. 1).

[7] American Pyschiatric Association. *Diagnostic and statistical manual of mental disorders (DSM-5R)*. Arlington, 2013 (cit. on pp. 3, 4).

[8] Matthew Maenner, Kelly Shaw, Jon Baio, Anita Washington, Mary Patrick, Monica DiRienzo, Deborah Christensen, Lisa Wiggins, Sydney Pettygrove, Jennifer Andrews, Maya Lopez, Allison Hudson, Thaer Baroud, Yvette Schwenk, Tiffany White, Cordelia Rosenberg, Li-Ching Lee, R. Harrington, Margaret Huston, and Patricia Dietz. "Prevalence of Autism Spectrum Disorder Among Children Aged 8 Years — Autism and Developmental Disabilities Monitoring Network, 11 Sites, United States, 2016." In: *MMWR. Surveillance Summaries* 69 (Mar. 2020), pp. 1–12 (cit. on p. 3).

[9] Julie Daniels, Ulla Forssén, Christina M. Hultman, Sven Cnattingius, David Savitz, Maria Feychting, and Pär Sparén. "Parental Psychiatric Disorders Associated With Autism Spectrum Disorders in the Offspring." In: *Pediatrics* 121 (June 2008), e1357–62 (cit. on p. 3).

[10] American Pyschiatric Association. *Diagnostic and statistical manual of mental disorders*. 4th ed. Arlington, 1994 (cit. on p. 3).

[11] Sandra Costa, Cristina Santos, Filomena Soares, Manuel Ferreira, and Fátima Moreira. "Promoting interaction amongst autistic adolescents using robots." In: *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*. 2010, pp. 3856–3859 (cit. on p. 3).

[12] K. Wall. *Autism and Early Years Practice*. SAGE Publications, 2009. ISBN: 9781446205860 (cit. on p. 4).

[13] Esubalew Bekele, Julie A Crittendon, Amy Swanson, Nilanjan Sarkar, and Zachary E Warren. "Pilot clinical application of an adaptive robotic system for young children with autism." In: *Autism* 18.5 (2014). PMID: 24104517, pp. 598–608. eprint: https://doi.org/10.1177/1362361313479454 (cit. on p. 5).

[14] Andrea Brivio, Ksenia Rogacheva, Matteo Lucchelli, and Andrea Bonarini. "A soft, mobile, autonomous robot to develop skills through play in autistic children." In: *Paladyn, Journal of Behavioral Robotics* 12.1 (2021), pp. 187–198 (cit. on pp. 5, 7).

[15] Audrey Duquette, François Michaud, and Henri Mercier. "Exploring the use of a mobile robot as an imitation agent with children with low-functioning autism." In: *Auton. Robots* 24 (Feb. 2008), pp. 147–157 (cit. on pp. 6, 7).

[16] *ROS 2 Documentation*. https://docs.ros.org/en/foxy/index.html (cit. on p. 8).

[17] *ROS-INDUSTRIAL*. https://rosindustrial.org/about/description (cit. on p. 9).

[18] *Generation Robots - ROS vs ROS2*. https://blog.generationrobots.com/en/ros-vs-ros2/ (cit. on p. 9).

[19] Huseyin Kutluca. *Robot Operating System 2 (ROS 2) Architecture*. https: https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776 (cit. on p. 10).

[20] B. Robins, K. Dautenhahn, R. Te Boekhorst, and A. Billard. "Robotic assistants in therapy and education of children with autism: can a small humanoid robot help encourage social interaction skills?" In: *Universal Access in the Information Society* 4.2 (2005), pp. 105–120. ISSN: 1615-5297 (cit. on p. 7).

[21] Salvatore Maria Anzalone, Elodie Tilmont, Sofiane Boucenna, Jean Xavier, Anne-Lise Jouen, Nicolas Bodeau, Koushik Maharatna, Mohamed Chetouani, and David Cohen. "How children with autism spectrum disorder behave and explore the 4-dimensional (spatial 3D+time) environment during a joint attention induction task with a robot." In: *Research in Autism Spectrum Disorders* 8.7 (2014), pp. 814–826. ISSN: 1750-9467 (cit. on p. 7).

[22] Hideki Kozima and Jordan Zlatev. "An epigenetic approach to human-robot communication." In: Feb. 2000, pp. 346 –351. ISBN: 0-7803-6273-X (cit. on p. 8).

[23] *LUDI: Play for Children with Disabilities*. https://www.ludi-network.eu/about-ludi/ (cit. on p. 1).

[24] *Gazebo Simulator*. http://www.gazebosim.org/ (cit. on p. 18).

[25] J.-J. E. Slotine H. Asada. *Robot analysis and control*. J. Wiley, 1986 (cit. on p. 12).

[26] *ROS2: Efficient intra-process communication*. https://docs.ros.org/en/foxy/Tutorials/Intra-Process-Communication.html (cit. on p. 24).

[27] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the performance of ROS2." In: Oct. 2016, pp. 1–10 (cit. on p. 24).

[28] *UML definition*. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/.

[29] *Mobile Robot Kinematics - Carnegie Mellon University*. http://www.cs.cmu.edu/ rasc/-Download/AMRobots3.pdf (cit. on p. 48).

[30] *Gazebo Flash Light Plugins*. http://gazebosim.org/tutorials?tut=flashlight_plugin&cat=plugins (cit. on p. 48).

[31] *Gazebo ROS Plugins*. http://gazebosim.org/tutorials?tut=ros_gzplugins (cit. on p. 48).

[32] *SDL Audio Library*. https://wiki.libsdl.org/SDL_OpenAudioDevice (cit. on p. 49).

[33] *java.net Class Socket*. https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html (cit. on p. 47).

[34] *The Great Dictator*. https://en.wikipedia.org/wiki/The_Great_Dictator (cit. on p. 51).

[35] *NVIDIA JETSON NANO*. https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-nano/ (cit. on p. 52).

[36] *CUDA FAQ (NVIDIA)*. https://developer.nvidia.com/cuda-faq (cit. on p. 52).