POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell’Informazione
Master’s degree in Automation and Control Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Improving the Code Generation Mechanism for the PuRSUE Framework

Supervisor: Prof. Matteo Rossi

Co-supervisor: Dr. Marcello M. Bersani

Master’s Thesis of:
Valentina Bonetti, Matr. 898311

Academic year 2019 - 2020

Contents

List of Figures

Abstract

The PuRSUE (Planner for RobotS in Uncontrollable Environments) Framework's purpose is to make the programming and reprogramming of robot applications easier. It also aims to deal with the programming of complex real scenarios, like the ones that can include multiple mobile robots or the presence of not controllable agents. Furthermore, another goal of the PuRSUE framework is to enlarge the number of people able to get in touch with robotic applications even without having the specific background.
A new high-level language PuRSUE-ML (Modelling Language), with the aim to model complexity, was developed in the previous work [1, 2]. The PuRSUE framework, through several translations, can automatically synthesize the control strategy and generate the code ready to be deployed into one robot. All the formal rules that define the new language and the parsing phase are well explained in the previous work [1, 2]. Moreover, the PuRSUE framework already presents features that make it suitable in some simple scenarios, for example, a situation in which a single mobile robot is acting in a well-known environment.
To know where to start the improvement, a complete analysis of the existing code was done. The main processes are presented in Appendix B.
An improvement of the framework was studied and tested. This improvement aims to make the framework suitable for distributed applications. Simple scenarios with at least two robots are taken into consideration. In simulations, these robots can be coordinate without knowing the complete control strategy. So, the main idea behind the partition of the control strategy and the algorithms used to make it feasible will be shown. The main results obtained from the tests are also reported to show the actual behavior once the distributed control strategy is applied. In the end, a more complex scenario was taken into account to show the robots' behavior in presence of a non-controllable agent.

Sommario

Il framework PuRSUE (Planner for RobotS in Uncontrollable Environments) si propone di semplificare la programmazione e riprogrammazione di robot. Cerca inoltre di ampliare il numero di persone che possono affrontare la sintesi di strategie di controllo, anche per scenari complessi. Questi scenari possono comprendere la presenza di più robot, che necessitano di coordinamento, e di agenti non controllabili. Nonostante sia un framework ancora in via di sviluppo presenta notevoli potenzialità che una volta implementate potrebbero portare ad un utilizzo di robot in nuovi scenari.
Un nuovo linguaggio ad alto livello PuRSUE-ML (Modelling Language), si propone di a modellare tali complessità, é stato sviluppato nel lavoro di tesi precedente [1, 2]. Il framework, a seguito di varie traduzioni, riesce a generare automaticamente il codice di controllo da implementare sul robot. Nella tesi su cui si basa questo lavoro [1, 2], sono specificate tutte le regole su cui il linguaggio è modellato e su cui si basano le traduzioni successive. Sono state implementate alcune funzioni che lo rendono già utilizzabile per semplici applicazioni che presentano un singolo robot controllabile che agisce in un ambiente conosciuto.
Per poter sviluppare ulteriormente questo framework si è resa necessaria un'analisi completa ed approfondita del codice esistente, riportata nell' Appendice B. Un miglioramento del framework é stato studiato. Questo miglioramento mira a renderlo distribuito e quindi implementabile su un sistema che presenta più agenti controllabili. Permetterà quindi il coordinamento di più robot mobili rendendoli capaci di perseguire un obbiettivo comune nonostante ogni robot abbia una conoscenza parziale della strategia di controllo.
L'idea alla base della separazione della strategia del controllo e tutti gli algoritmi volti a realizzare tale separazione sono presentati. Sono riportati inoltre i principali risultati ottenuti dai test fatti. Questi test mostrano il comportamento simulato in scenari che presentano più di un robot controllabile, con complessità crescente. Infine è stato preso in considerazione uno scenario più complesso che presenta anche un agente non controllabile.

Chapter 1

Introduction

Since the second industrial revolution, people began to use machinery to accomplish heavy and repetitive tasks. These machines became gradually more and more complex and sophisticated. With the addition of electronics and software a new category was born: robots. Robots can handle heterogeneous jobs but,with them, new challenges appeared. One of the most complicated challenges to deal with turned out to be how to handle situations with uncontrollable and unpredictable agents or time dependent actions.
We could better describe these general subjects as an increasing need to solve coordination and communication issues among robots. Other difficulties can emerge, for example, from the complexity of writing programs in low-level languages. Although powerful, these languages has some drawbacks: they are error-drown, make the robots not re-programmable, and always need a well-trained programmer.
Some examples of these possible scenarios could be the so called Drug Delivery Problem[1], the Catch the Thief Problem, or any other application of mobile robots in a protected environment (as in an airport). In this kind of situations, it could be necessary to have a complete and real-time knowledge of the entire system state and an overall control strategy.
People with less defined tasks have the possibility of being uncontrollable agents in already present real circumstances. One typical real-life scenario could be found in the residences for elderly, with robots acting as automated medical carts carrying drugs. Driven by a nurse, who knows the medical aspects but does not need to know any informatics nor programming languages. At the airport, for luggage transportation, logistic and transport coordination. A reprogrammable robot provides advantages in terms of flexibility and

[1]For more information see Appendix B

reduced total cost: it could complete different duties under different needs or different time frames. A robot working on internal luggage delivery could be temporary engaged in porterage service for travelers during rush hours, or dedicated to people with disabilities.
As an agricultural application: a robots' distributed swarm which seeds or plows, even in many different and distant fields. Actually, all the scenarios presenting swarm structure are well suited to have reprogrammable features.
An ecological application: an EcoRobot [1, 2] able to dispose of the waste in the right containers, managing priorities and different kinds of trash.
In all these cases, we could count on real economic advantages. Furthermore, it is a reprogrammable solution and it is also able to manage priorities, e.g., if an automated medical cart is carrying drugs following a routine schedule, it can suddenly leave its mission to provide help if an emergency occurs.
To create these no longer classic applications, robots have to be reprogrammed by non-programming experts, as employees, nurses, doctors, patients, farmers, etc. In this vision, being able to describe high-level scenarios becomes helpful.
A previous work [1, 2], on which the current thesis is based on, has designed a Domain Specific Language, named Pursue-ML [1, 2], to specifically address these problems and to describe complex scenarios in an easy way. Together with the language, a framework has been created as well, to read and translate these high-level descriptions into programs.
To better understand the context in which this work was carried out, let us briefly introduce the PuRSUE Framework. The framework translates scenarios described in the PuRSUE-ML language into the Timed Game Automata formalism. This operation is possible through the use of already existing technologies and, through the use of UPPAAL-TiGA [25] [24] [23], the Framework is able to synthesize a control strategy, or plan, containing the descriptions of the desired behavior of every controllable agent. The complete control strategy is able to take into account also the decisions taken by uncontrollable agents and, so, the behavior of the controllable one changes accordingly. To make the strategy feasible, the intermediate .txt file, output coming from TiGA and representing the control strategy has to be translated again, this time in python, to make it actually runnable on the scenario's robots. The final output, i.e. the executable code, has then to be deployed on each robot, to allow it to communicate both with its lower-level control system, and with other robots.
In this work, we initially deeply examined the PuRSUE Framework in all its main aspects, as well as the logic on which it is based, all the parts that make it work, and how it works, to build a basis for subsequent improvements (the results of this analysis are included in Appendix B).

The two major limitations of the current version of the framework that emerged from this study are the sending of *a priori* commands and the generation of a single, centralized controller. The goal of this work is to overcome these limitations. In particular, the designed and implemented changes aim to improve the PuRSUE framework and make it able to handle also distributed system scenarios.
We focus in particular on making the framework more efficient and more intuitive. Even though the PuRSUE framework is at its very beginning, it shows a great potential and it could even change some aspects of our future lives. In this work, we will take into account only mobile robot scenarios, even if PuRSUE could be used for other types of situations as well. To illustrate the context in which this work was developed, next we present a significant scenario.

The Catch the Thief Scenario

The Catch the Thief scenario is one of the most meaningful examples of a possible application of the PuRSUE Framework.
Figure 1.1 reports a simplified map of a generic floor of a building. In this

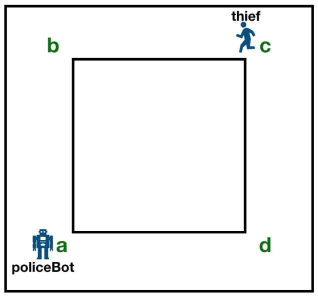


Figure 1.1: Catch the Thief simple map

situation, there is an uncontrollable agent, the thief (we can think of a real human being), who wants to escape from the aforementioned floor, and a policeBot that has to catch him.
To accomplish its task, the policeBot has to immobilize the thief with a given

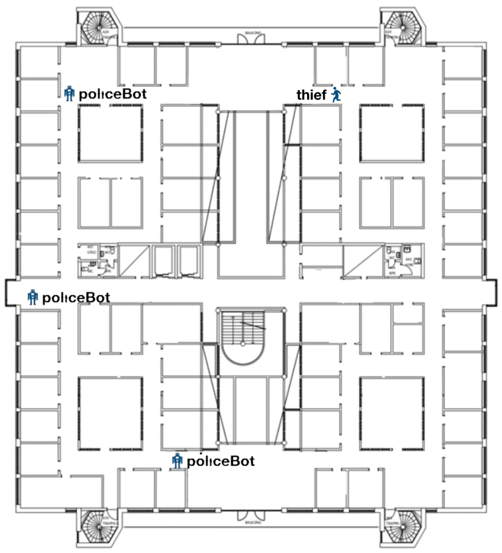


Figure 1.2: Catch the Thief complex map

tool. This operation will be modeled as a collaborative event in PuRSUE-ML[2].
The policeBot, to operate, has to know the environment in which it moves. For this purpose, we can assume that the policeBot is provided with a map and/or perception and mapping algorithms. The low-level robot software (e.g., for controlling the speed of the robot) is not significant for the framework and we assume the robots will be provided with all the required low-level control functions.
In PuRSUE-ML the environment is described through the Points Of Interest (POI) and the distances between them and, through this information, the framework can synthesize the control strategy which will bring the policeBot to catch the thief, if possible. In this case, the environment is a floor where we identify four POIs labeled as "a", "b", "c", "d". PuRSUE-ML is also able to model other variables, such as the speed of the two agents or the unfeasible robot movements.
This framework is able to describe and solve many variations of this simple situation as well. Consider, for example, Figure 1.2 where the environment is more complex and more policeBots are present.
It is also possible to consider the complete reverse situation, where the Bot

[2]For more information see [1, 2].

is the thief and policemen are real people.

Structure of the thesis

The thesis is structured as follows:
Chapter 2 summarizes the state of the art of literature most closely relayed to the present work, and gives a brief description of the main tools used for developing and test our improvements.
Chapter 3 provides some details of the PuRSUE framework developed in [1,2], i.e. the work on which this thesis is based.
Chapter 4 presents the main upgrades introduced in the PuRSUE Framework to make it distributed and more efficient.
Chapter 5 evaluates the effectiveness of the introduced modifications through experiments on four scenarios of increasing complexity, and also on a simple Catch the Thief scenario.
Chapter 6 concludes with a summary of the main results obtained and a discussion of the main limits that emerged from the experiments, and it proposes some future works.
The appendices provide a brief user-guide of the PuRSUE framework and an exhaustive description of the code developed in [1,2].

Chapter 2

Background

This section treats two main topics. In the first part, we present the state of the art of some relevant issues for this work, in particular, the literature related to the creation of plans through Timed Game Automata formalism and the code generation starting from formal models. In addition, we present the technical tools and the technologies used to achieve the robots Run-Time infrastructure, in particular Ros and Docker (and Uppaal TiGA).

2.1 State of the art

Here a brief description of the state of the art related to the code generation from a Timed Game Automata (TGA) formalism is presented.
The research starts with [11], where a "semidecision procedure for synthesizing controllers for systems modelled as linear hybrid automata" is discussed. This procedure is implemented and tested. The control problem is seen as a two-person game, in order to win the plant, for its continuous variables, is allowed to make uncontrollable discrete jumps or just follow the flow. Definitions, theorems, lemmas, and proofs are provided, together with the definition of the control problem. The supervisor control theory is extended and the algorithm for discrete event system is generalized for a fully observable systems. For systems with partial observability, the controller may be not found even if it exists. A sufficient condition that affirms the requirements for an LHA to be control-divergent is given, but this condition is not necessary.
The aim of [8] is to describe how to find the behavior that satisfies specific properties. It is added the possible choice, for the automata, between taking an action or just leave the time pass. The basic system's algorithm is extended with the quantitative timing information and also the game theory

is added to the TA formalism with the propose to find a winning strategy. The definitions of Timed Automata, winning strategy, and safety are given in order to define these entities. In the beginning, only the controller synthesis for Automata is taken into consideration, together with the algorithm used to find it. Then the results are expanded to considering the Timed Automata. In the end, a `run` is defined as the sequence of joint steps. Together with this definition theorems and corollaries are announced and proofs are presented.
In [4] it is introduced the characteristic of Dense Real-Time into a supervisor control. It provides two main results, the "condition on the existence of a controller" and the statement that the problem of the synthesis of the supervisor in Timed Automata has exponential complexity.
This kind of complexity can generate problems for Timed Automata in a target platform with restricted memory. So, in [10], it is exposed how to generate code in a restricted memory condition. The timing specifications are translated into a compilable C-code. The workflow is based on UPPAAL [25] and presents two different types of synchronization through two different types of channels. In the end, the results are compared with a traditional thread-based implementation.
In [6] the problem of the controller synthesis for Metric Temporal Logic is faced. Metric Temporal Logic is a Linear Temporal Logic with a timed extension, and [6] asserts that their controller synthesis is undecidable. To avoid this problem it describes how to translate the MTL into a non-deterministic Timed Büchi Automata passing through a Transition-based Timed Büchi Automata and using an over and under-approximations of this last one.
A strategy to synthesize a decentralized control for a discrete-event system is described in [3], with a focus on the possibility to present a priority structure. To do that, the transitions related to an automaton can be dependent on other systems. The overall system time is the same for every automaton in order to maintain synchronism.
Referring to the "Model Interpreter For Timed Automata" document [9], the model is directly executed without passing through a model-to-code translation. The model interpreter can be used to model and verify real-time systems, making the model verifiable and executable. While the code generation can present problems, the document explains how a virtual machine can read and run the model avoiding such issues. The transitions are classified in different types and the two main phases of the interpreter are described. These two phases are the "Executable model generation" and the "Model Execution". All the used algorithms are presented and tested.
Close work to PuRSUE is found in [5], where a network of interacting automata for model multi-robotic vehicles is described and the motion planning

problems are solved at a high-level. Different approaches are suggested and threaded Petri nets are used to simulate the concurrent robot behavior. The hypothesis on the robot was that they have the feedback controllers already implemented. The overall output is a sequence of control locations, and the Computational Tree Logic (CTL) formalism is interpreted by the local trajectories generator. The model, through synchronization channels, is able to consider the environment, the robot, and the control. Anyway, for exemplification, it is assumed that the robots can move only on a Cartesian grid, and clocks are used to represent the motion time. All the proposed properties are verified with UPPAAL.
Another interesting framework is presented in [7], the "framework for the synthesis of robust and optimal energy-aware controllers". Here is defined the Energy Timed Automata (ETA), on which the control problem is based, with the adding of the timing constraints and the variable energy rates to the Automata definition. The propose is to present an optimization on solving the identification of the minimal upper bound. Theorems are enunciated and some examples are presented to prove the improvement. Then also the Energy Timed Automata with the addiction of uncertainties is considered. The solution of this last further complexity is provided through the translation of the ETA into a first-order linear arithmetic expression that is then simplified with a quantifier eliminator and formula simplifier.
The [12] begins with a comparison, where are reported some of the limits related to the programming languages, the Petri Nets and the temporal logic. Therefore, a framework based on the Time Transition Model (TTM) and Real-Time Temporal Logic (RTTL) is presented. The features of the TTM and RTTL are exposed and it is verified that the generated controller, form the framework, satisfies the required specification. The semantics is defined and all the necessary definitions are provided together with examples. The parallel composition of TTMs is discussed and, at the and, the framework's problems are displayed together with its good qualities. One of the most meaningful aspects is that this framework can give an expression of qualitative and quantitative properties and model the complex features of discrete event systems.

A fast overview was made also on the following topics, to have a more complete.
The article [16] shows the main aspects of a multi-robot system: the model on which their behavior is based (biological inspirations), the communication, the internal architecture, the algorithms of localization, mapping and exploration, the object transport and manipulation and the motion coordination.

Distributed intelligence was treated in [14], where is shown how it can be applied on a multi-robot system through three different paradigms: the bio-inspired paradigm, the organizational and social paradigm, taking into account some variability, e.g. the robot can be similar or specialized in some specific functions.
As a communication system, the use of the neural network is theorized in [13].
An infrastructure Genom3 [15, 17, 18], integrated with the Bip engine, can manage all the execution and the synchronization of the inner robot functions. In the future, can be developed an interface able to allow the communication between the PuRSUE framework and Genom3This can be possible because Genom3 takes as input, the command addressed to the robot, and, through the fact that is middleware independent, is able to work with Ros. The [15,17,18] explains also how the automata are used to follow the evolution of the robot situation.

2.2 Used Tools and Framework

The most important technical tools and the technologies used to achieve the robots Run-Time infrastructure are here shortly introduced.

2.2.1 Uppaal-TiGA

The University of Uppsala (Sweden) and the University of Aalborg (Denmark) cooperate to develop the Uppaal integrated tool environment. It aims to help in the management of a timed automata's network.These automata, that model real-time systems, are submitted to validation and verification processes [25].
The Uppaal integrated tool environment was upgraded with the TiGA extension in order to operate also on Timed Game Automata. An efficient on-the-fly algorithm was added for solving games. A solution can be found under the conditions of the satisfaction of reachability and safety properties. [23].
This tool is the basement of the framework. The scenario is, indeed, translate into a Timed-Game-Automata to be processed from Uppaal-TiGA which gives, as output, the complete control strategy. In addition, the satisfaction of reachability and safety properties is really important otherwise, the control strategy does not exist and the PuRSUE framework becomes worthless.
The version of the executable code can vary with the release, if problems in the parsing phase arise, please check if a little change in the java code or in

the grammar's file[1] is needed.

2.2.2 Ros

Ros, the acronym of Robot Operating System, is maybe one of the most famous open-source middleware for robot application [28]. It is provided with the implementation of commonly-used functionality, with tools and libraries that can manage low-level tasks and also the communication across multiple devices [29].
In the PuRSUE Framework, the Ros middleware is widely used, especially in the Run-Time phase. It can put in communication all the agents acting, both in the base framework and in the distributed proposal. Lots of tutorials and documentation are available, the official one is [29].

2.2.3 Docker

Docker is the last tool used, but not for importance.
The main function of the Docker tool is to make a container, a simple isolate part of the code. The containers can be seen as a lighter version of virtual machines, they are based on the host operating system and, with the host, can share only the OS with fewer additions. Once you operate inside a container, aside from the OS, just the files inside a linked folder can be used or the Dockers Images code as well. The Dockers Images provide a code package over the given OS, in this case, we use the image of Ros-melodic [33].

[1]Generally needs a change from a space to a tab in a "when "condition in the `UP2CO.g4` file (the grammar of the controller's translation)

Chapter 3

PuRSUE Background

In this chapter the structure of the PuRSUE Framework developed in [1] is presented.
As anticipated, to achieve the purposes of this work the first thing was to fully understand the functioning of the code generation of the current PuRSUE Framework. We did a detailed reverse engineering of the framework both to build a basis for its future development and to highlight the points on which we will intervene to apply the improvements announced in the Introduction 1.
The macro operation is here explained, further details are given in Appendix B.

This framework aims to convert a complex situation described in a high-level language into a usable code that can be deployed on a robot to achieve the mission. All the formal rules are well described in the previous thesis [1, 2].
The PuRSUE (Planner for RobotS in Uncontrollable Environments) framework is developed with the purpose to make the synthesis of the controller code easy and automatic to allow anyone to develop a robotic application without getting in touch with the difficulties of coding at a deep level.
The framework allows us to model a wide range of scenarios, it can support Robotic Applications with Uncontrollable Agent(RAUA) and applications that require real-time control. The most interesting use of PuRSUE is on modeling scenarios that involve a multi-mobile-robot system in presence of one or more uncontrollable agents.

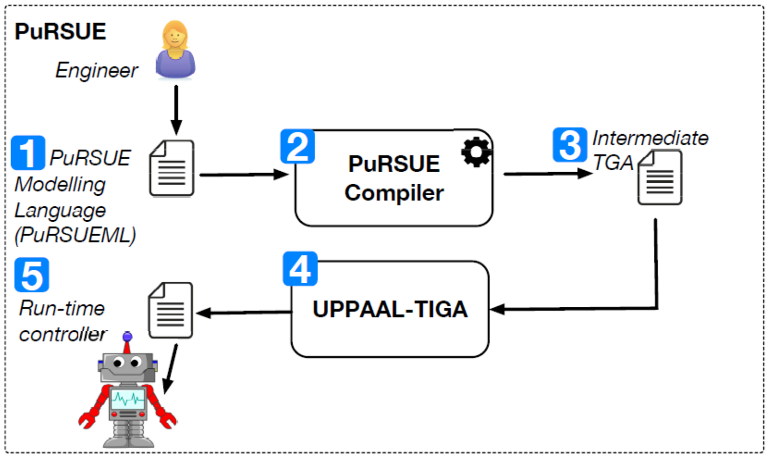


Figure 3.1: Basic Idea scheme

3.1 Overview

The flow of the transformation of information is well exemplified in Figure3.1 taken from [1].
Everything begins with the PuRSUE Modelling Language (PuRSUE-ML), a Domain Specific Language in which the scenario can be modeled. The file written in PuRSUE-ML is then passed into a compiler that translates it into a Time Game Automata (TGA). The rules of this translation are well defined in the previous thesis [1, 2] and the result is a TGA that completely models the scenario saving it into two files: one with extension `.xml` that contain all the features of the TGA and another one with extension `.q` where is saved the goal that the controllable agent(s) has(have) to reach.
To synthesise the control strategy the UPPAAL-TiGA tool is used, having as input the complete TGA. As output UPPAALL-TiGA gives a file containing all the instruction that the controllable agents have to follow to achieve the goal. This file is written in an unusual computer language, so another translation is needed to have a complete usable code. So the framework parses the previous file with extension `.xml` and the last generated file with extension `.txt` to have a python written code.
To be more precise, from the file with extension `.xml` is generated a so-called `Observer.py`; the Observer is in charged to keep marked the overall state of the system. From the file with extension `.txt` is generated the `Controller.py`; the Controller, once it knows the overall update system state, can identify the correct action that has to be taken and notified it to

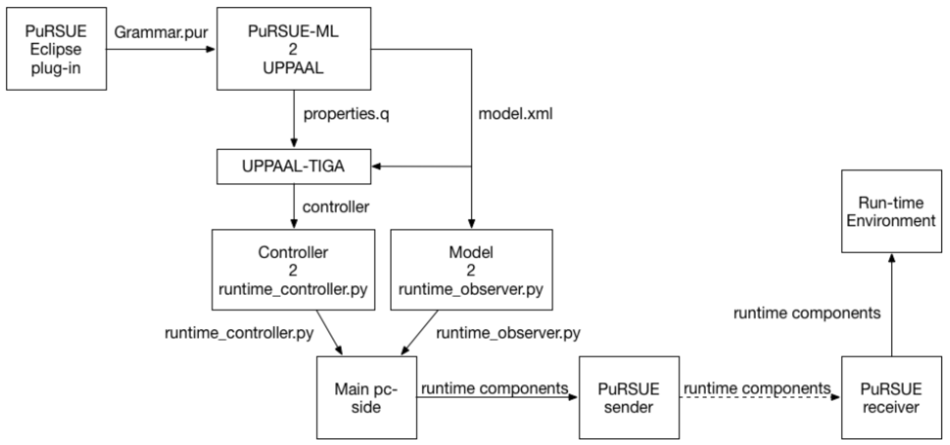


Figure 3.2: Code flow scheme

the robot. All these passages can be seen in Figure 3.2, taken from [1,2], where is well highlighted the bifurcation of the parsing operation that converge at the end of the `main_pc-side`. In Figure 3.2 the main_pc-side is shown as the last operation but is the program that coordinates all the translation processes.
Once the `Observer.py` and the `Controller.py` were obtained, they are sent to a completely different context. Now the generated code is part of a complex ROS environment that it is able to consider all the agents and the actions that they can perform and also all the event that can happen. Despite it is not largely represented in the Figure 3.2, the ROS environment phase is important because is where it is seen the real contribution of the framework (the robot moves!).
To avoid confusion between the three main phases, from now on, they are indicated with: Design-Time, Rest2Ros and Run-Time respectively.

3.2 PuRSUE Implementation

The separation of the three phases is also reflected in the code's folders organization. Once the framework is downloaded it presents two main folders that separate the Design-Time code from the Run-Time code, while Rest2Ros has a mechanism that makes its code dislocated.
In Figure 3.3 is shown the separation between the three phases applied to the scheme of the code flow. Before going into a deeper description of the Framework, a schematic summary is given:

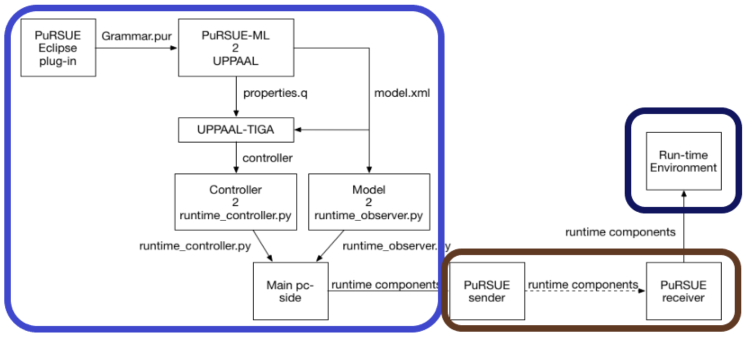


Figure 3.3: How the Code Flow is split in the three main phases

- **Design-Time:** The light blue rectangular, in this phase all the translations take parts, as described before. The input is the PuRSUE-ML written file (.pur) and the outputs are the Controller and the Observer written in python.
- **Rest2Ros:** The brown rectangular, is composed of handwritten functions [39] [40] that allow the communication between the Design-Time and the Run-Time. It takes the generated `Controller.py` and the `Observer.py` and sends them in the correct folder inside the robot's software.
- **Run-Time:** The blue square, once the `Controller.py` and the `Observer.py` are obtained and correctly placed, a prepared Ros environment is ready to run. In this way, the Observer_node knows whatever is happening and the Controller_node can give the exact command to the robot. It is assumed that the robot already has an internal code that allows itself to complete the task.

In the following three sections is given a little more detailed description, for deeper information see Appendix B.

3.2.1 Design_Time

Inside the Design-Time folder are present some subfolders, as shown in Figure 3.4, that completely reflects the parsing stages. Even if the number of the folders can mislead, only the first four folders contain the code used for the translation. The fifth and sixth ones contain particular files to develop other structures, for this reason, these folders are left aside and briefly described

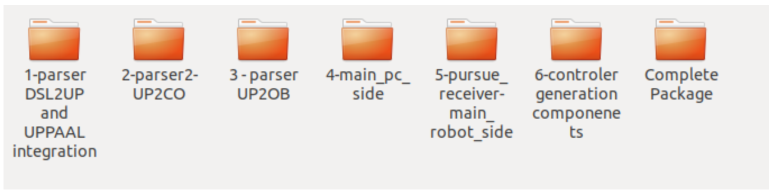

1-parser DSL2UP and UPPAAL integration
2-parser2-UP2CO
3 - parser UP2OB
4-main_pc_side
5-pursue_receiver-main_robot_side
6-controler generation componenets
Complete Package

Figure 3.4: Design-Time Subfolders

in Section B.5, in the Appendix B. In the end, the seventh folder is again relevant for the design phase and will be analyzed. This last one is the result of the previous four and the real one to be used for the parsing. Actually, the folders from 1 to 4 are there to be written and tested, but to be able to work their code has to be compressed and added into `CompletePackage`. For more information on how use it see Appendix A. As said in the above paragraph, here from the file written in PuRSUE-ML are generated the `Controller.py` and the `Observer.py`. Each folder is in charge of one translation and is developed separately for interacting only once in the `CompletePackage`.
Each subfolder has a complex structure since tools as Xtext [26, 27], Maven

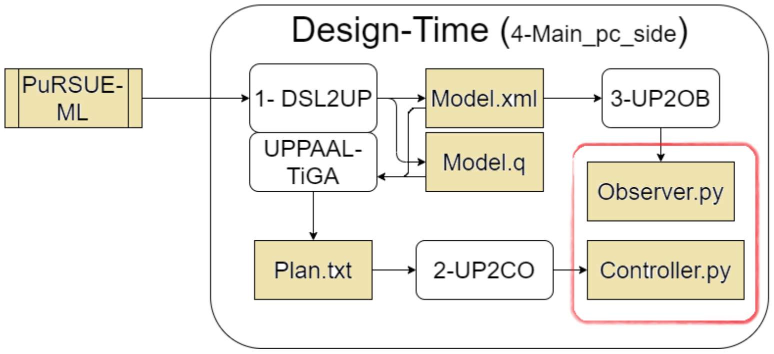


Figure 3.5: The Design-Time translation flow with the respective fold number

[19, 20] and Antlr [21, 22] are used. These tools generate files for inner use and testing.
The first folder is the most articulate, but beside it, all of them present more or less the same arrangement. The code of interest is contained in the `src` folder, and some other encapsulation till arrived at the `Main.java`.
Taking as reference the Figure 3.5 where is shown the correspondence between the idea previously described and what is done: the created files are in the slightly colored box, and the parsers in the white ones. The output files are

grouped in the red rectangle.

- **1-parser DSL2UP and UPPAAL integration:** Translate the PuRSUE-ML input file into the equivalent Timed Game Automata. After that the executable program of *verifytga* [23–25] is launched and the control strategy is generated in the `.txt` extension.
- **2-parser2-UP2CO:** the generated control strategy is translated into a python file thought a java parser. Here are defined all the classes used to define the `Controller.py`, so here it is decided how the Controller will be.
- **3-parser UP2OB:** from the file with the extension `.xml` who described how the Automata is, it generates the `bserver.py` file. The same strategy of `2-parser2-UP2CO` is used, and here all the classes used for the parsing are defined.
- **4-main_pc_side:** here the sequence of parsing is managed. The results are then sent to the next phases, invoking the MissionSender.

3.2.2 Rest2Ros

The Design-Time phase and the real implementation in Run-Time phase occur in different moments and different environments. While the first one takes place on a computer that can be anywhere, the Run-Time implementation needs to be inside a specific structure with parts on the robot and has to perform in the real scenario. To put the generated file into the correct folder, an *ad hoc* code was written [39, 40]. This code allows the sending the part of interest via internet into the physical network, where the robot interacts. In Figure 3.6 the part of the complete flow of information that is treated, exactly the brown box is shown. In the blue box is represented the real Run-Time environment described in the next paragraph. What is shown in Figure 3.7 is a complex scheme of what this phase has to do. Starting from the code generated in the previous section, they are sent via REST protocol, so with the function GET and POST. This mechanism has to convert the function of Rest into messages in ROS protocol, so in a mechanism of Publish and Subscribe. To well understand this, it is suggested to have minimal confidence with both, Rest and Ros.
The first agent is **Mission_sender**, it is invoked at the end of the Main.java of `4-main_pc_side` and it forwards the `Controller.py` and the `Observer.py`, one at a time, to a dedicated server. The IP address and the port are specified with the invocation of the method. Once arrived on the server, the

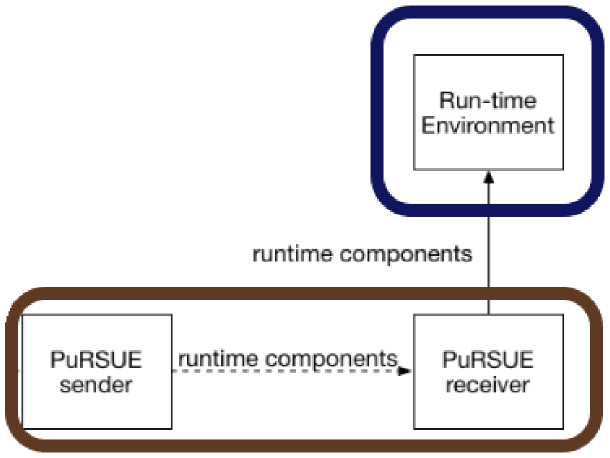


Figure 3.6: Communication flow scheme

`mission` will found the **Communication_manager** working on the given port. The Communication_manager is the second agent, converts the obtained information from the Http protocol to a message that will be published on a topic, following the Ros rules. As the last step, the Pursue_reader reads the messages published form the Communication_manager and places them into the correct folder. To do that, three Ros topics are defined: two on which the Communication_manager is subscribed (`mission_action` and `mission_location`) and one on which the Communication_manager can publish, `local_mission`.

3.2.3 Run_Time

In Figure 3.8 the Run-Time scheme is shown, it is composed of only one box because it strongly depends on the situation. In this section the main agents and the main structure, to make the description more general as possible, are presented. In Figure 3.9, from [1], the architecture of the Run-Time phase is represented. It takes place into the Ros workspace and uses the main Ros tools for delivering messages. There are four main actors and four main communication channels.
The PuRSUE_UI is used to notified external events and can publish on the topic `pursue_event`.
The Observer_node, based on the `Observer.py`, keeps updated the system state and the internal clocks. To do that, it listens on the `pursue/events`

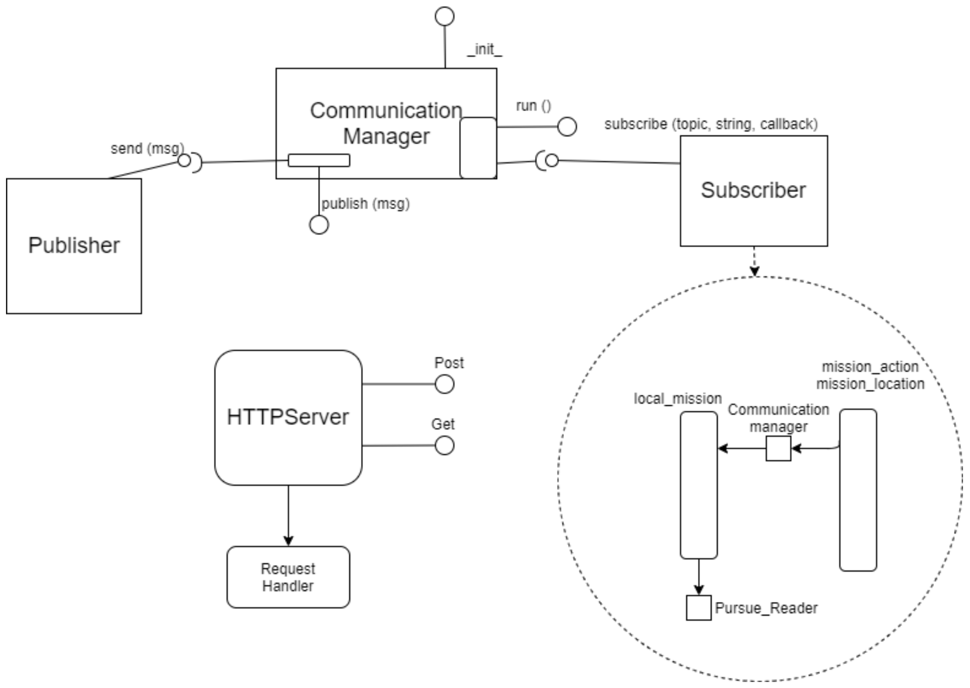


Figure 3.7: Rest2Ros scheme

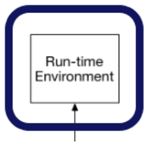


Figure 3.8: Run-Time scheme

topic, where the Controller_node and the PuRSUE_UI are publishing. The Observer_node can also publish on `pursue\ system_state`.
The third agent is the Controller_node, based on `Controller.py` an others files. It is in charge to select the correct command and notify it to the robot. The Controller_node reads from `pursue\ system_state` and publishes on `move_base_simple\goal`, `pursue\action` and `pursue\events`.
The last agent is the robot itself, it can read the command from `move_base_simple\goal` and `pursue\action` and execute command forwarded on them.

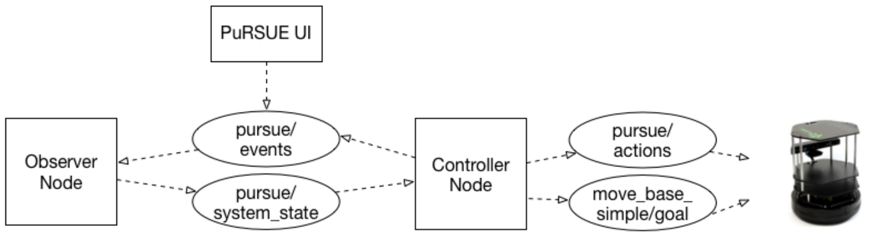


Figure 3.9: Run-Time Ros scheme

Complex case: Controller_node

The Controller_node is one of the main agents of the Run-Time phase and, for sure, is the most complex. As the main part derived from the control strategy generated at Design-Time, some functions are not performed directly from the `Controller.py` code. For this reason, the structure reported in Figure 3.10, from [1] was implemented. The `Controller.py` file can read the

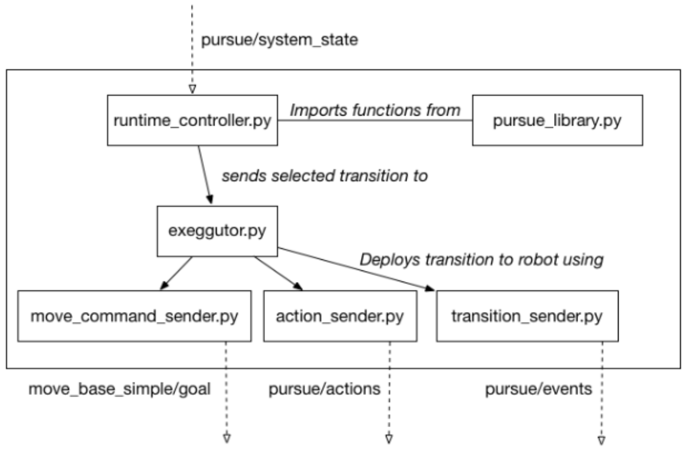


Figure 3.10: Controller_node's composition

messages provided by the Observer_node, but relies on five other files to select and forward the correct command.

In `pursue_library.py` the functions used to compute the optimal waiting time are contained.
The duty to select the correct topic, on which forwarding, is committed to the `Executor.py`.
To send correctly the messages other three files are created, each one can publish on a specific topic: `move_command_sender.py` on `move_base_simple\goal`, `action_sender.py` on `pursue\action` and `transition_sender` on `pursue\events`.

Run-Time interactions

It is shown how the actions take place. Once the agents are correctly placed and started, they began to work. In Figure 3.11 is represented the Run-Time sequence diagram between them, taken from [1]. The PURSUE_UI (the

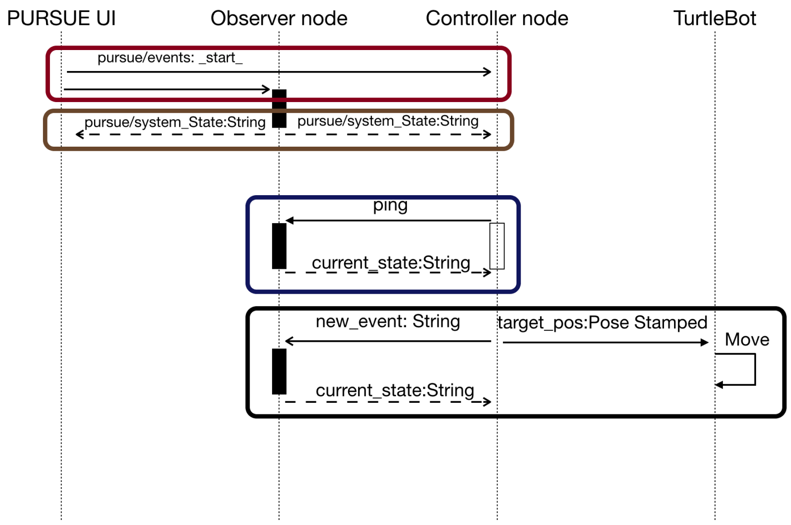


Figure 3.11: Run-Time sequence diagram

Interface) launches the start signal, in this way the Controller_node and the Observer_node are activated. This action is shown in the Figure 3.11 inside the red box.
After that, as shown in the brown box, the Observer_node computes and publishes the system state initial conditions. The initial conditions are seen from both, the PURSUE_UI and the Controller_node.
Once the Controller_node has received the information, it begins the computation of the corresponding command related to the current system's state.

After it has found it, the Controller_node computes the optimal waiting time and waits for it. Before sending the corresponding command, the Controller_node asks the Observer_node to republish the updated system state (Blue box).
The Observer_node re-publishes it and if the system state is not changed the Controller_node sends the command to the robot through the specific topic and on `pursue\events` (Black box).
The robot will execute the movement or the action, while the Observer_node will update the system state. The updated data will be published, and the Controller_node will begin the new research for the next instruction.
As an example, in Figure 3.12 is reported the complete Run-Time Ros structure of the ecoBot problem solved in [1].

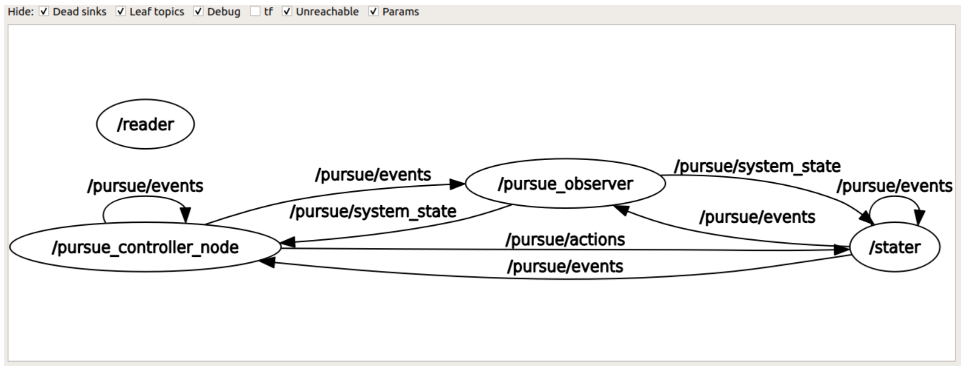


Figure 3.12: Run-Time Ros scheme for the EcoBot scenario

3.3 Considerations

The PuRSUE Framework already implements the features for a single robot application and it is already tested with the use of TurtleBot [36]. This robot provides free software for the low-level robot control (as speed, path planning, obstacle avoidance, and so on), acting on the EcoBot scenario.
The Figure 3.13 shows the UML scheme related to the current composition of the Run-Time phase. Each component is associated with a class, the scheme also reports the main relationships between the classes themselves and expands the composition of the more complex ones. Anyway, the framework presents two important limitations.
The first one is that the control strategy, and so the Controller_node, provides also the information of the end of a movement or an action. While this

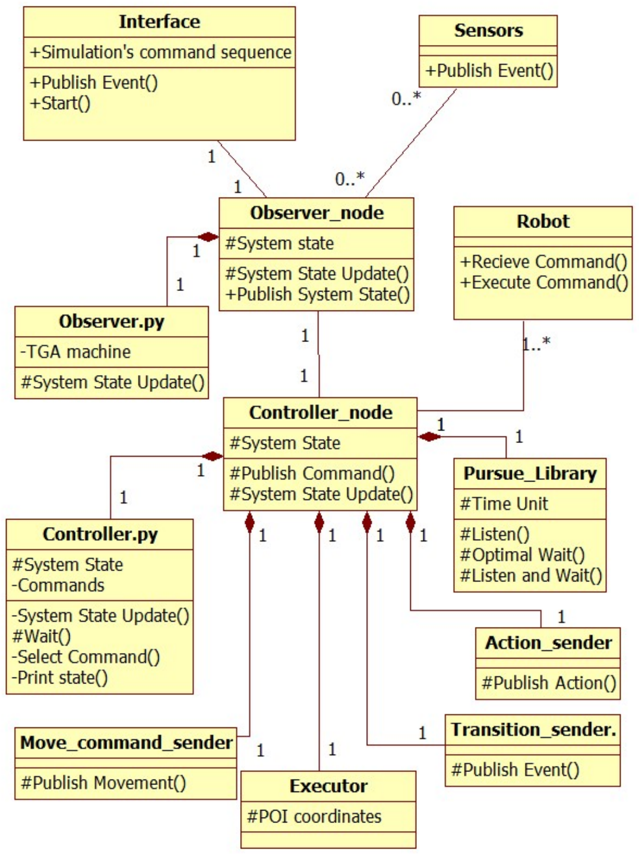


Figure 3.13: UML class diagram depicting the core elements of PuRSUE

information is important to compute a correct and feasible control strategy, once in the real scenario it can becomes harmful. That is because they lead the Observer_node to make the system evolve even if in the real situation the robots can present delays and problems to accomplish their tasks.
The other one is that it provides the structure for a single robot system, while it can be potentially used in a multi-robot system with distributed structure. The next chapter shows how these limitations can be overcome.

Chapter 4

Contribution of the thesis

In this section we describe the main idea to make PuRSUE framework suitable for distributed applications and to make robot's actions and movements not decided *a priori*. In this way, we can coordinate more robots to pursue a common goal and the Observer will compute a more realistic system state. Another goal of this work is to make the code more efficient, to simplify the execution and to speed up the command sending.

Hypothesis We made these hypotheses in order to be able to focus on more specific scenarios.
The first assumption is that we have only mobile robots that are endowed with computers, sensors, all the software for low-level controls (as localization, mapping, path trajectory, collision avoidance algorithms) and everything they need to complete their tasks. We also assumed that the robots can interface to the Ros middleware, so they can recognize the given instruction and to execute it.
The second assumption is that the environment where the robots act is limited and presents the necessary sensors that allow the Observer to be aware of all the system conditions.
We started to modify the Run-Time phase in order to overcome the limits highlighted with the previous work (*A priori* decision and single generated controller). To do this, we decoupled the design of the Run-Time components from the translation phase in order to allow a simpler analysis of the individual phases. To avoid a number of unnecessary and complex processes, we choose to start identify the characteristics that needed to be modified respect to the components already present. Otherwise, after completing the translation phase, the obtained result may be not valid and would force us to start over

again. With this choice, once identified the structure of the components, the translation is focused and the subsequent tests will be more reliable. It means that the development of parsing and deployment are neglected in this work and will be done manually.

4.1 Idea

We aim to divide the controllers to spread the computation of commands across all acting robots. So, we decide to move from a structure where there is a single controller to one where there are many controllers. The goal is to reach a configuration like the one shown in Figure 4.1, where, compared with interaction scheme described in Chapter 3 and exemplified in Figure 3.9, each robot has its own controller.

Therefore, we start analyzing the example of previous work to understand

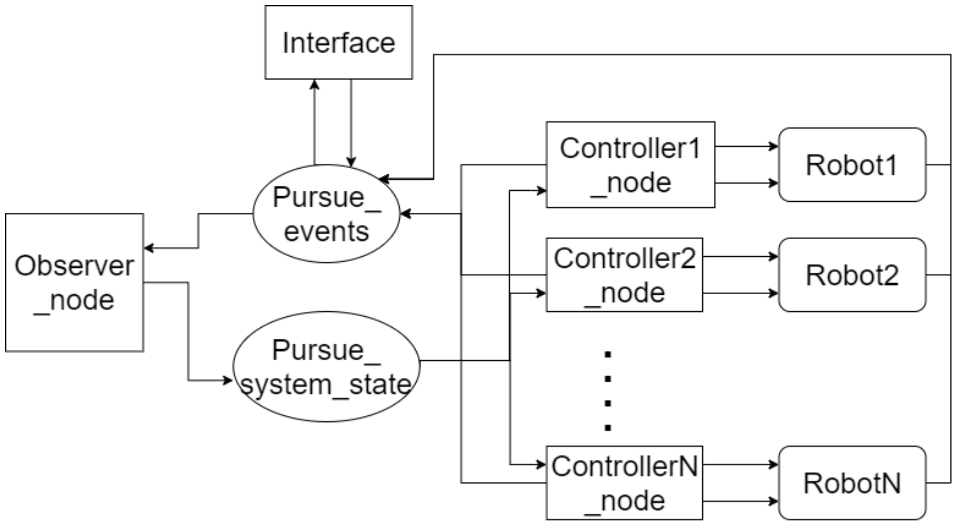


Figure 4.1: Distributed scheme

how the components behave. In order to see how the framework works with two robots, we have simplified the scenario and add the second agent (scenario 1, described in Chapter 5). This allowed us to have a simpler cases of *if structure*. These *if structures* are a set of instructions that associate the system state with the correct action that has to be taken. Generally, the controller, as it is created [1], has a significant number of *if structures*, but

[1]See Appendix B for more information.

the condition of having fewer commands has made it more readable. In this condition, the problem of the *a priori* decisions in the control strategy has become evident.
The *a priori* decisions introduce a problem because they notify the end of actions or movements regardless of what the real conditions are. For example, if it is the robot that reports that it has finished doing something, there is a very high probability that it has actually finished and so this information leads to having the correct representation of the real scenario. Instead, if this information is provided by a component, e.g. the controller, but *a priori* without checking the real situation but based only on the time passing, it can lead to having the saved system state that does not correspond to the real one. This mismatch can lead the controller (and therefore the robot) to take completely wrong or misleading decisions that could be counterproductive, if not harmful.
We, decided to eliminate these *a priori* decisions. To realistically provide the information previously provided by these *a priori* decisions, we modified the interface by inserting the appropriate commands while maintaining the original structure.
We split the description of the possible strategy of the two robots into two separate files by eliminating all the *a priori* conditions and maintaining the conditions related to only one robot. Changes have also been made to other components (described in the Implementation paragraph) which have led to better efficiency of the entire system. Validation tests on this scenario have been successful. We also verified that communication between the various agents was possible even in an isolated context simulated through the containers in Docker.
After some tests with increasing complexity, but which did not present any uncontrollable agents, we tried to simulate a simple Catch the Thief case. This scenario sees the presence of 2 bots and the thief (uncontrolled agent). All the tested scenarios are reported in Chapter 5. This scenario, that presents an uncontrollable agent, implies a higher degree of the overall complexity, due to the wider range of action possibility introduced with that kind of agent. This higher complexity is reflected also on the structure of the generated control strategy. This because, not only the number of possible policeBot decisions is greater, as are greater their ramifications, but especially because they are more influenced by external events. Actually, the bot decisions depend also on the timing of the events, e.g. if the thief changes direction then the policeBot could also change direction and follow a new tactic (which, however, is already planned in the control strategy). These new features have led us to separate the commands more carefully and forced us to find a new

condition able to separate the controllers in a general way.
Another feature that this scenario brought to light was the presence of an inner variable in the mechanism, this variable can not be deleted since it takes into account how close the system is to the goal. While in the first cases, those that presented only controllable agents, this variable involved little disturbance, in the last scenario its presence became more massive. This fact has forced us to no longer be able to ignore how to manage it. Solutions have been, therefore, identified in order to be able to take this variable into account without weighing on the overall system. The tests made on the Catch the Thief scenario, even if using only simple actions by the uncontrolled agent, has given encouraging results for the simulation of the bots behavior that are able to coordinate themselves and achieve their goal.

Now we will briefly show all the changes made taking into account the complexity introduced by the last scenario, in the next Section 4.2 these modifications will be examined more in detail.
Figure 4.2 reproduces the UML scheme of the new agents of the PuRSUE Run-Time environment, for a better comparison take into consideration the scheme of the previous features shown in Figure 3.13 in Chapter 3. Even though it does not seem to be different from the previous conception, novelties have been introduced. For example, the executor will implement also the functions of the Action_sender, Transition_sender and Move_command_sender. In this way, the controller_node presents fewer files and the execution will be more fluid. For more information on the previous formulation see Appendix B. In Figure 4.3 is reported an expansion of the UML scheme focused on the controller_node.

Another change is about the relation between the Observer_node and the Controller_node. In this case, the Observer_node can be associated with more than one Controller_node. Each Controller_node is associated with one controllable robot [2], so a single Observer_node can manage more than one robot.
In Figure 4.4 is reported an expansion of the UML scheme focused on the Run-Time agent that highlights this change.

The last important improvement is that the notification of the end of an action or a movement will be provided by the robot through its sensors and not by the controller_node.

[2]To be more precise the control strategy generated designs the behavior for every controllable agent. But each human that collaborates with the robot to pursue the goal is considered also as a controllable agent, for example the nurse in the Drug Delivery scenario. But it is not realistic to write down code to instruct a human being, for this reason, it was preferred to write "robot" instead of "controllable agent".

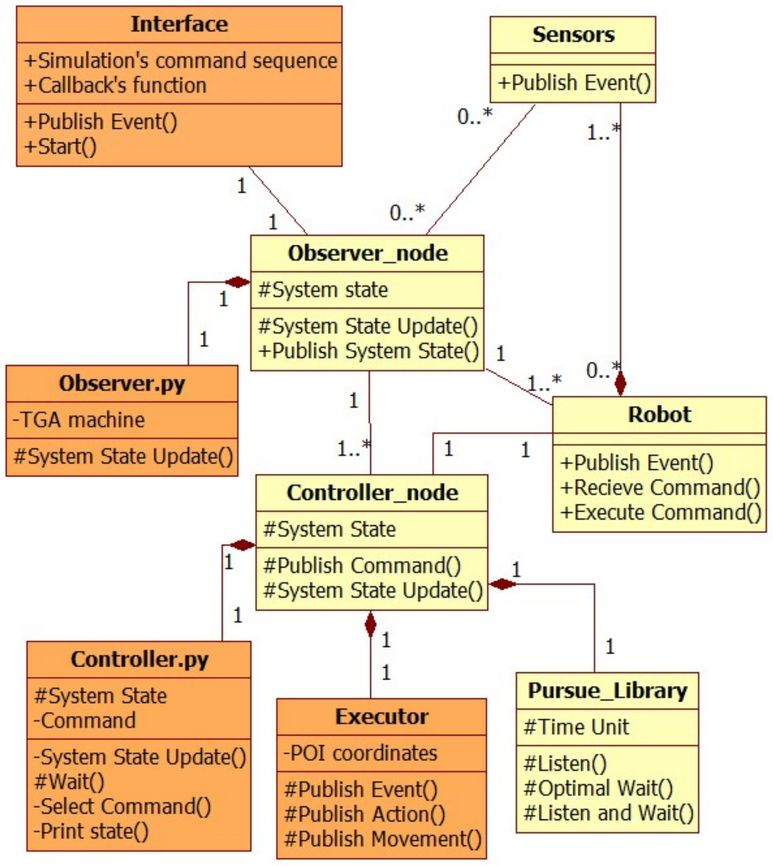


Figure 4.2: UML class diagram depicting the improved structure of the PuRSUE framework

In Figure 4.5 is reported an expansion of the UML scheme focused on the addition of the sensor's response.

4.2 Implementation

In this section we present the artifacts that we developed to improve the PuRSUE Run-Time environment; in particular, we present their goals and we highlight their differences with their previous version. The main components that were modified are indicated in Figure 4.2 with a darker color, they are: the `Controller.py`, the `Executor.py`, the `Observer.py` and the `Interface`.

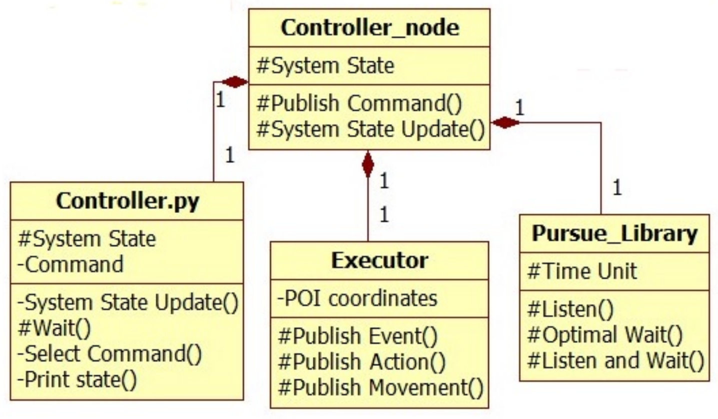


Figure 4.3: Executor UML scheme

Together with them, some changes on the PuRSUE-ML file and the Ros infrastructure are presented.

4.2.1 Input_language.pur

Even though is not necessary for the control strategy generation, this proposal can be useful in the Run-Time phases. As shown in Listing 4.1, the idea is to specify the coordinates of the point of interest directly into the input_language.pur file (written in PuRSUE-ML).
The coordinates will have the structure of the position (space coordinates) and orientation (unitary quaternion) as prescribed from Ros type of Pose variables. In this way, the information will be just forwarded in the Run-Time file without having further transformations.

```
//locations
poi "a" //[x, y, z, nu, ex, ey , ez],
poi "b" //[x, y, z, nu, ex, ey , ez],
poi "c" //[x, y, z, nu, ex, ey , ez],
```

Listing 4.1: Input_language modified

This additional information will be completely ignored for the generation of the `Observer.py` or of the `Controller.py`, but it will be added in another parser to generate the `Executor.py` specific for the considered situation. So the interpreters already existing will not need any changes, and the control strategy will be generated with only the information on the travel time taken

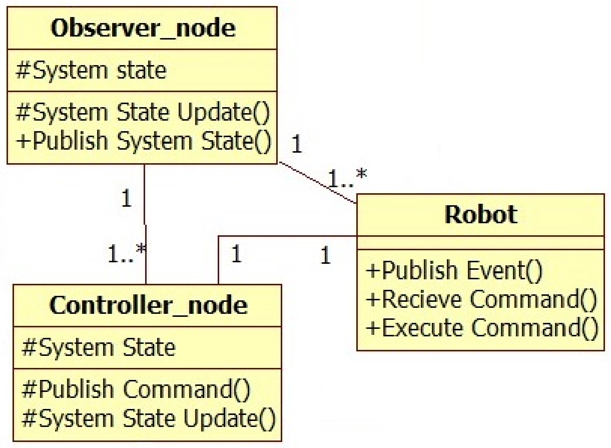


Figure 4.4: Run-Time components UML scheme

by each agent.
At the beginning was thought to add the POI dictionary to the `Controller.py` to avoid adding another parser, but it would have resulted in a reformulation of the mechanism of encrypting the commands. Also, in vision to realize a distributed system, the `Executor.py` will be maintained the same for every Controller_node, so just scenario dependant, while the `Controller.py` will be strongly scenario and robot dependant.
So, once this feature will be implemented will be the way to avoid, for the users, to open the `Executor.py` and enter manually the points' coordinates.

4.2.2 Controller.py

This is the most modified code. The main idea is to create one `Controller.py` for each controllable robot. In this way many advantages can be reached, for example, the computation for the correct action will be faster.
As first action, the Run-Time controller generated ad Design-Time was copied for every controllable agent. In this way, the agent will be completely independent from the others and can decide independently the action that has to begin based on the system state.
All the generated classes, the variables and the printed messages were renamed. This is vital, if not a completely not univocal and not understandable situation will be generated. For a more precise definition of how the elements

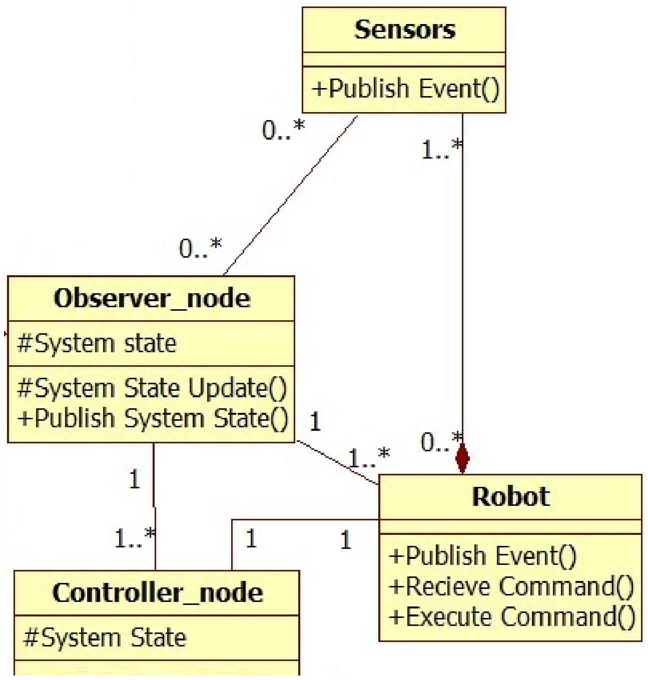


Figure 4.5: Addition of sensor's response UML scheme

have been renamed an example will be exposed in the next Subsection 4.2.2. Just with that procedure, the framework will be able to sustain more than one controller_node, but it was noticed that many improvements can be provided to make the code less redundant and, of course, to make the events of the end of action or movement not decided *a priori*.
It was noticed that only the *if structures* with the command for the considered robot were necessary. It means, that the robot does not need to know the complete control strategy to well behave. The information that it needs is just relative to the part of the actions it can perform and the system state in which it has to perform them. All the other information is useless and so can be omitted.

```
#state header
   if ( self.reachObj== "unlocked" and self.bot2== "a" and self.bot3==
        "a" and self.Prule1== "2" and self.Pbot3== "1" and self.Pbot2
       == "1" and self.rule1== "rule10" and self.Pbot1== "2" and self.
```

```
    bot1== "b" ):
  temps=set()
  temp0= optimal_wait([ 1 + self.TIMEUNIT-self.Cbot3, 1 + self.
      TIMEUNIT-self.Cbot2 ] , [ ] , [])
  if (temp0 >= 0):
   temps.add(temp0)
  if(temps):
   wait = min(temps)
   self.event_flag.clear()
   self.event_flag.wait(wait)
   self.exegg.ping_observer()
   if (self.reachObj== "unlocked" and self.bot2== "a" and self.bot3
       == "a" and self.Prule1== "2" and self.Pbot3== "1" and self.
       Pbot2== "1" and self.rule1== "rule10" and self.Pbot1== "2"
       and self.bot1== "b" ):
    if ( (1<self.Cbot3 and 1<self.Cbot2 ) ):
     #takes in agent ID, synchronizing action (or tau), and the states or
         ogirin and target of transition
     self.exegg.exeggute("bot2","bot2_ina!", "a", "
         doing_bot2_ina_in_a")
  else:
   self.event_flag.clear()
   self.event_flag.wait()
```

Listing 4.2: "If" deleted for the first reason

This thought is translated into the deletion of all the *if structures* that present commands for other robots, so not for the robot instructed by the controller. In Listing 4.2 is shown an example of an *if* that will be eliminated considering the `Controller1.py`, so related to a hypothetical bot1. That is because the command is referred to a hypothetical bot2. The command, in the *if structure*, is identified in the lines that begin with `self.exegg.exeggute()`. This function invokes the Executor.py and passes them four parameres: the agent that has to act, the encrypt name of the action that has to be done (the trigger), the origin and the target. For further information see Appendix B. The main result of this operation is that the `Controller1.py` will be more reactive because the system options that has to check are significantly reduced.

Another main change provided follows from the belief that the notification of the end of an action or a movement, the so-called *a priori* decisions, has to be provided from the robot, or external sensors, and not from the

`Controller.py` itself. For this reason, other *if structures* are deleted and the interface was modified, to be able to reproduce these events and allows the simulation of the entire system.

```
if ( (1<self.Cbot3 and 0<self.Creach and 9<self.Cbot1 and 1<self.Cbot2
    and self.Cbot1<=10 ) ):
        #takes in agent ID, synchronizing action (or tau), and the states or
            ogirin and target of transition
        self.exegg.exeggute("bot1", "tau", "going_b_to_a", "a")
    else:
        self.event_flag.clear()
        self.event_flag.wait()
```

Listing 4.3: "If" deleted for the second reason

In Listing 4.3 is shown an example of the *if structure* that will be eliminated for the aforementioned reason. The example refers again to a hypothetical bot1, and the *if structure* presents a command of an end of a movement. This information is contained in the value of the parameter `trigger` (the third) that begins with `going`. The other type of commands that has to be deleted is the one that presents the `trigger` parameter starting with `doing`. If in the PuRSUE-ML some rules or states-and-dependencies lines appear, it can happen that in the complete `Controller.py` are present some instruction about their change of states. These instructions can not be deleted because they keep the information on the priority of the events that have to be performed and can be identified because are referred to an inner variable called `reachObj`. Some solutions can be adopted to maintain them and split the controller anyway: they can appear in each sub-controller, or just in one or,otherwise, can be created a `ControllerObj.py` dedicated to the evolution of these rules, state and dependencies. To make all the variables update, it was also necessary to change the `Creach` variable into `CreachObj` in the `__init__` function.
In Figure 4.6 are summarized the main steps for the creation of the distributed controllers. The result is a set of sub-controllers, shorter and dedicated to a single robot. Again is important that they present a unique name otherwise they will be confused between each other.
To create a centralized `Controller.py`, that not presents the feature of the decision of the end of an action or a movement, can be just deleted the second presented type of *if structure*.

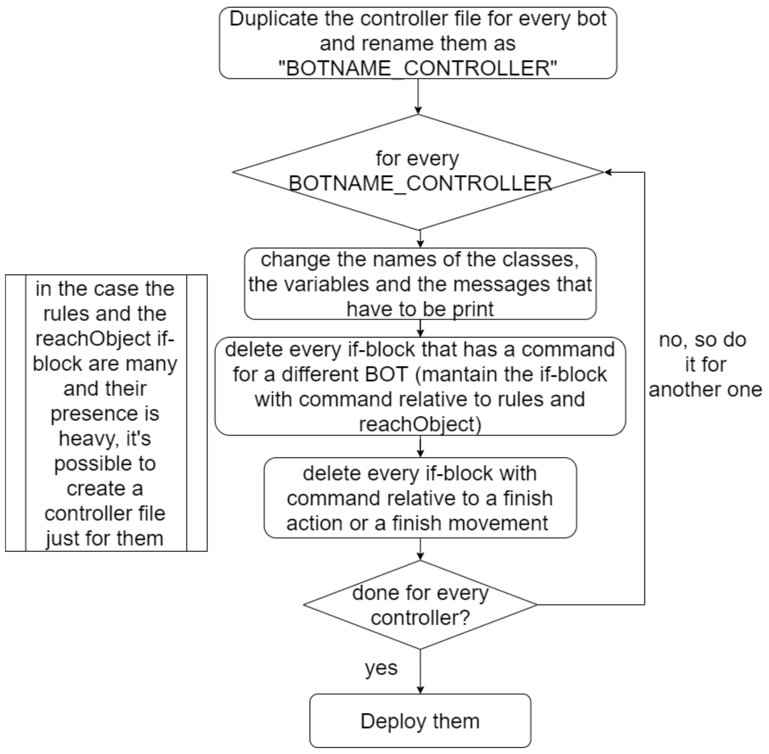

Duplicate the controller file for every bot and rename them as "BOTNAME_CONTROLLER"
for every BOTNAME_CONTROLLER
change the names of the classes, the variables and the messages that have to be print
delete every if-block that has a command for a different BOT (mantain the if-block with command relative to rules and reachObject)
delete every if-block with command relative to a finish action or a finish movement
done for every controller?
yes
Deploy them
no, so do it for another one
in the case the rules and the reachObject if-block are many and their presence is heavy, it's possible to create a controller file just for them

Figure 4.6: Distributed controllers algorithm

Example of a single-robot `Controller.py`

Here is presented a hypothetical `Controller1.py`, commented in the main changes performed, in terms of names, from the generated global controller. In Listing 4.4, are shown how the controller class has to be renamed based on which robot refers to it. In the same way, it has to change its name also in the main correlated function to correct invoke the run method.

```
class controller1:

 def if_start(self, event_string):
  if (event_string.data == "_start_"):
   self.startFlag = True
  return
...
def __init__(self):
  rospy.init_node('controller1_node')

  self.TIMEUNIT = 0.1
  self.startFlag= False
  self.event_flag = Event()
  self.event_flag.clear()
...
def print_state(self):
  print("controller1: the system state is:")
  print("controller1: reachObj is ",self.reachObj)
  print("controller1: bot3 is ",self.bot3)
  print("controller1: bot1 is ",self.bot1)
  print("controller1: Prule1 is ",self.Prule1)
  print("controller1: bot2 is ",self.bot2)
  print("controller1: rule1 is ",self.rule1)
  print("controller1: Pbot3 is ",self.Pbot3)
  print("controller1: Pbot2 is ",self.Pbot2)
  print("controller1: Pbot1 is ",self.Pbot1)
  print("controller1: Cbot2 is ",self.Cbot2)
  print("controller1: Cbot3 is ",self.Cbot3)
  print("controller1: Cbot1 is ",self.Cbot1)
  print("controller1: Creach is ",self.Creach)
...

def main():
 controllore = controller1()
 controllore.run()
if __name__ == "__main__":
 controllore = controller1()
 controllore.run()
```

Listing 4.4: Main changes

As the first thing, the name of the class is changed. It is important to notice that also the name of the corresponding node is changed while all the state's and clock's variables remain. The fact that the variables related to other agents are not removed is really important because the `Controller.py` computes the decisions based on the complete system state. It is vital that it has complete access to all the system variables, if not it can not be able to enter in any *if structure* and so it can not find any command that has to execute.
In the function `print_state`, it is useful to add which controllers are printing.

This function does not notify anything to any of the Ros agents, it just gives information on the prompt to make the users updated on what is happening. So, the adding of the information of who is writing and the fact that all the system's variables are printed does not affect the system's functioning.

A complex controller's case

In some situations, the controller can be complex and presents more commands for a single if structure. In Listing 4.5 is shown an example taken from the Catch the Thief original controller, in the Chaprter 5 is presented the complete scenario.

```
if ( (0<self.Creach and 1<self.CpolBot2 and self.Cthief<9 and self.
    CpolBot1<1 and self.CpolBot1<self.Cthief ) or (9<=self.Cthief and
    0<self.Creach and self.CpolBot1<=10 and self.CpolBot1<=self.
    CpolBot2 and self.CpolBot2- self.CpolBot1<=1 and self.Creach<=
    self.CpolBot1 and self.Cthief<self.CpolBot1 ) or (1<self.CpolBot2
    and 0<self.Creach and self.CpolBot1<=10 and self.CpolBot2<self.
    CpolBot1 and self.Cthief<self.CpolBot1 ) or (0<self.Creach and 9<=
    self.CpolBot1 and 1<self.CpolBot2 and self.CpolBot1<10 and self.
    CpolBot1==self.Cthief ) or (1<=self.CpolBot1 and 1<self.CpolBot2
    and 0<self.Creach and self.Cthief<9 and self.CpolBot1<=self.Cthief
    and self.CpolBot2<self.Cthief ) ):
        #takes in agent ID, synchronizing action (or tau), and the states or
            ogirin and target of transition
        self.exegg.exeggute("polBot2", "polBot2_a2b!", "a", "
            going_a_to_b")
      if ( (0<self.Creach and 1<self.CpolBot2 and 10<self.CpolBot1 and
            self.CpolBot1<=11 and self.Cthief<9 and self.CpolBot2<self.
          CpolBot1 ) or (0<self.Creach and 9<=self.Cthief and 10<self.
          CpolBot1 and 1<self.CpolBot2 and self.CpolBot1<=11 and
          self.Cthief<10 ) or (0<self.Creach and self.CpolBot1<=11 and
          self.Cthief<self.CpolBot2 and self.CpolBot2- self.CpolBot1
          <=-10 ) ):
        #takes in agent ID, synchronizing action (or tau), and the states or
            ogirin and target of transition
        self.exegg.exeggute("polBot1", "tau", "going_a_to_d", "d")
      if ( (0<self.Creach and self.CpolBot2<=1 and self.CpolBot1<self.
          Cthief and self.Cthief<=self.CpolBot2 and self.Creach<=self.
          Cthief ) or (0<self.Creach and self.CpolBot1<=11 and self.
          Cthief<9 and self.CpolBot1<=self.CpolBot2 and self.Cthief<=
                self.CpolBot1 ) or (0<self.Creach and self.CpolBot2<=1 and
                self.CpolBot2<self.CpolBot1 and self.CpolBot1- self.CpolBot2
                <10 and self.Cthief<=self.CpolBot2 ) or (0<self.Creach and
                self.CpolBot2<=1 and self.CpolBot1<=11 and self.CpolBot2
                <=self.Cthief and self.Cthief<=self.CpolBot1 and self.
                CpolBot2<self.CpolBot1 and self.Cthief- self.CpolBot2<9 ) or
                (0<self.Creach and 1<=self.CpolBot1 and self.Cthief<9 and
                self.Cthief<=self.CpolBot2 and self.CpolBot1<self.Cthief ) ):
            #takes in agent ID, synchronizing action (or tau), and the states or
                ogirin and target of transition
            self.exegg.exeggute("reachObj", "tau", "initial_location", "
                unlocked")
        else:
          self.event_flag.clear()
          self.event_flag.wait()
```

Listing 4.5: Controller's if with a complex command structure

In Listing 4.5, a complete *if structure* of the controller is reported, in the first place it controls the system state and calls a function of `pursue_library` to compute the optimal time to wait. After the controller had waited the optimal time and had checked that the system state is not changed, there are other if constructs. Here the conditions on the clocks are checked. Can happen that with the clock conditions can change the action to send, or that more than one action can be selected.
In this case, it was separated as the *if structure* was a normal one, maintaining the complete external structure and delete just the inner *if*s that are not relative to the controller in issue.
Relatively to the example, for the hypothetical controller2 the external if and the first inner one are maintained, while the inner if of the action relative to the controller1 is deleted because is not relative to the bot2. The command relative to `reachObj` is a completely different case, every action on it depends on the strategy it was chosen to manage that kind of command. In the following Paragraph 4.2.2 are described the main solutions, for now just ignore it. So in the end, in the controller2 only the first *if* will be present.
Tacking into consideration the hypothetical controller1, no option will remain. That is because the first one is relative to the bot2, the second one is relative to an end of a movement and the third one is deleted due to a design choice. In this case, where no inner *if* are maintained, there is no reason to maintain all the external *if structure*, so it is completely deleted.

The reachObj agent As seen, sometimes a command is relative to the reachObj agent. ReachObj is related to an Observer's class that helps to take into account the automata situation, if there is a risk to lose or if it is close to the win. So these commands are really important for the correct system's behavior and they can not be deleted. But they can be managed in more than one way:

- **Present in all controllers:** The commands are left everywhere they appear. In the phase of deletion, or as a parsing choice, all the *if structure* in which the reachObj commands appear they are considered good, even if a different controller is taken into consideration. This solution is advisable for simple scenarios.
- **Present in one controller:** One bot is chosen as the one that will maintain the reachObj commands. In the phase of deletion or parsing, the reachObj command will be treated as a command of the bot with the exception that all the command will be maintained even the ones that will regress the system state. So, in all the other controller these commands will disappear. This solution is advisable for simple/medium scenarios.
- **A dedicated controller:** A controller is created just to take these commands into account. It is called `controllerObj` and here are present all and only the command related to the reachObj class. In the phase of deletion or parsing, all the commands related to the other bot will be deleted but not the command related to a regress of the system. This solution is advisable for middle/complex scenarios. Due to the aim of the framework, it is the most suitable solution to be adopted and to be taken into consideration for the writing of the controllers parser.

4.2.3 Executor

Here the different executor code is proposed. The first main change is that it includes all the operative functions of the Controller_node. So the changes brought to eliminate the three files: Action_sender, Transition_sender and Move_command_sender. In Listing 4.6 is reported the beginning of the new executor. It does not import anymore the previous mentioned files, but the classes are directly declared inside it.

```
class Transition_sender:
  def __init__(self):
    self.pub = rospy.Publisher('pursue/events', String, queue_size = 10)
    sleep(1)
  def send_message(self, transition):
    self.pub.publish(transition)

class Move_command_sender:
  def __init__(self):
    self.pub = rospy.Publisher('move_base_simple/goal', PoseStamped,
        queue_size = 10)
    sleep(1)
  def send_message(self, coordinates):
    goal = PoseStamped()
    goal.header.frame_id = "map"
    goal.header.stamp = rospy.Time.now()
    goal.pose.position.x = coordinates[0]
    goal.pose.position.y = coordinates[1]
    goal.pose.position.z = coordinates[2]
    goal.pose.orientation.x = coordinates[3]
    goal.pose.orientation.y = coordinates[4]
    goal.pose.orientation.z = coordinates[5]
    goal.pose.orientation.w = coordinates[6]
    self.pub.publish(goal)

class Action_sender:
  def __init__(self):
    self.pub = rospy.Publisher('pursue/actions', String, queue_size = 10)
    sleep(1)
  def send_message(self, action):
    self.pub.publish(action)

class Exeggutor:
  def __init__(self, timeunit): # dovrebbe essere parsato dalla modifica
      proposta del linguaggio .pur

    self.location_dictionary = {
    "a" : [3.0, 1.54, 0.0, 0.0, 0.0, 0.67 ,0.73],
    "b" : [-0.57, 0.62, 0.0, 0.0, 0.0, 0.71 , 0.70] ,
    "c" : [0.47, 2.89, 0.0, 0.0, 0.0, 0.76, 0.65],
    "d" : [-3.19, 3.58, 0.0, 0.0, 0.0, -0.04, 1.0]
```

```
        }
        self.move = Move_command_sender()
        self.act = Action_sender()
        self.transition = Transition_sender()
        self.TIMEUNIT = timeunit
...

if (trg[0] == "unlocked"): #per gestire gli unlocked
  trigger_cleaned = "reachObj_going_initial_location2unlocked"
  self.transition.send_message(trigger_cleaned)
...
```

Listing 4.6: Executor

Another change brought is the removal of two of the cases in the method *executor.run()*. Since the commands of the end of a movement or an action are removed from the `Controller.py`, it has no sense to leave the cases that check for those types of commands. Instead were added new cases.
These new cases aim to manage the commands of the reachObj class. Once the controller sends the command, it is checked if the target is one of the possible reachObj states and, in this case, it forwards the message that makes the transition take place. For a comparison with the previous version or just major information, the original code is reported in Appendix B.

4.2.4 Observer

This is the agent that requires fewer changes than others. If the class `ClassreachObj` is present, into its function `trigger`, modified the line

```
if (trigger in self.machine.get_triggers(self.state):
```

into

```
if (trigger in self.machine.get_triggers(self.state) or
trigger=="reachObj_going_initial_location2unlocked"):
```

Even if this modification is minimal, makes possible for the Observer.py to update the class state and recognize the winning. There could be problems also for the Clock of the ReachObj because it could have a different name in `Observer.py` and `Controller.py`.
It was tried a solution to split also the Observer_node to seek for possible

modularity, but it results unfeasible. The conclusion of not feasibility is not final, but due to the complexity of the identification of irrelevant variables present where the controllers check the system state. All the system variables, states and clocks, are present in the *if* instruction and is not immediate to understand which one is not relevant and so erasable. This is the same reason for which in the controllers are saved the complete system state.

4.2.5 Interface

In Listing 4.7 is shown the part of the interface code modified. For a comparison with the previous version or just major information, the original code is reported in Appendix B.
While the same mechanism is maintained to notify the uncontrollable event and to test the path of an enemy strategy, he main differences are introduced to notify the events of the finish of an action or a movement. This it was necessary to be able to test automatically the overall functioning.

```
class Stater:
 def callback_ev(self, data): #stampa a video gli eventi -> dopo il tempo
     che serve vengono pubbliati gli eventi di fine spostamento o fine
     azione
  if (data.data == "_start_"):
   self.startTime = time()
  if (data.data != "_ping_"):
   print("\n\non topic events:" + data.data + " at time "+str(time()-
      self.startTime))
  if (data.data== "bot2_c2b"):
   sleep(3)
   print("da interfaccia: bot2 in b.")
   self.pub.publish("bot2_going_c_to_b2b")
  if (data.data== "doing_bot2_ina_in_a" or data.data=="bot2_ina"):
   sleep(6)
   print("da interfaccia: bot2 in a DONE!.")
   self.pub.publish("bot2_inaDONE")
  if(data.data== "bot2_inaDONE"):
   self.pub.publish("Finish!")
  if (data.data== "bot2_b2c"):
   sleep(3)
   print("da interfaccia: bot2 in c.")
```

```
    self.pub.publish("bot2_going_b_to_c2c")
```
Listing 4.7: The change of the Interface code

Inside the function `callback_ev` are added more if, besides the preexisting that checks if the data are `_start_` or `_ping_`. The added instructions check which event has taken place and waits for the respective time. After waiting, it will be published that the event is finished with the correct syntax[3].
Once in the real scenario, this task will be absolved by the system sensor or by the inner sensors of the robot. In this case, is necessary to have the events notified in this way to be able to simulate. In the way it was done before are not more possible because the commands were deleted on the controllers.

4.2.6 Ros

In Figures 4.7 4.8 and 4.9 are presented the actions to do to prepare the Ros environment. In Chapter 5, the new infrastructure is tested using also the Docker Tool. This tool needs to operate on different folders to be effective and that is another reason to separate the packages correctly. To correctly start everything, the files have to be placed into the correct folders and all the names have to correspond. Every node has to have a specific folder containing the respective files, to create everything without problems is necessary to proceed with order.
As first thing, it is copied a runtime folder[4] for each node and immediately renamed it with the node's names.
Beginning with the Interface, it is just delete everything that is not the code of interest (the Interface file).
Passing to the Observer_node, inside its folder are deleted the folders `Devel` and `Build` if present. This operation is necessary because these folders contain information related to the previously created environment. So, all the folders ad files except the `src` folder will disappear. Now, inside the remained folder it is again deleted everything except the Observer_node and the ms2_kth packages. Inside `src/ pursue_observer_node/src` is located the Observer.py generated and modified.
All previously described operations are repeated also for every generated `Controller.py`, but for them a further step is needed. Every folder, every file and some variables inside the files were renamed with the name of the current Controller. In this way, an unambiguous characteristic is given to

[3]The syntax is described in Appendix B, in the Executor's description.

[4]With the *runtime* folder, it is indicated the folder that contains the Run-Time environment. It can have a different name, but here it is used runtime for exemplification.

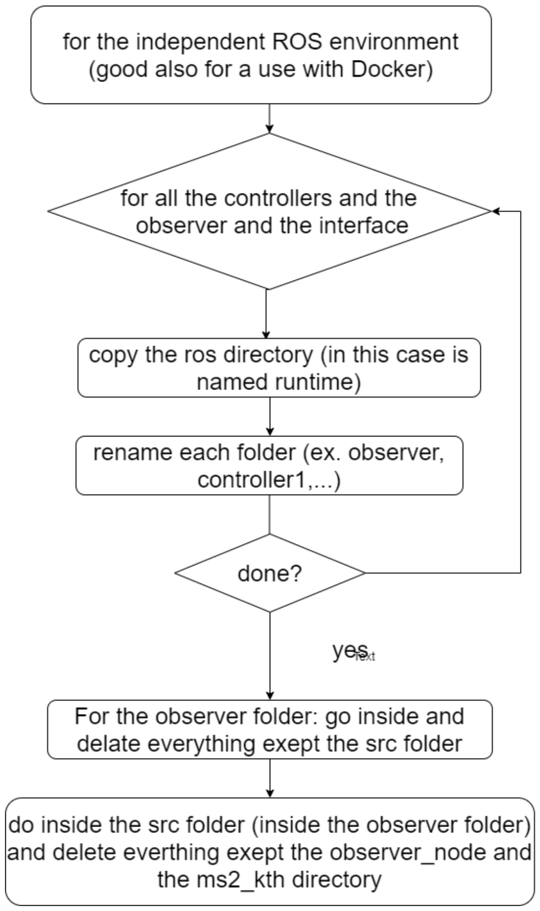

for the independent ROS environment (good also for a use with Docker)
for all the controllers and the observer and the interface
copy the ros directory (in this case is named runtime)
rename each folder (ex. observer, controller1,...)
done?
no
yes
For the observer folder: go inside and delate everything exept the src folder
do inside the src folder (inside the observer folder) and delete everthing exept the observer_node and the ms2_kth directory

Figure 4.7: Ros adaption algorithm: part 1

the system and is solved also the internal reference problem.
The Ros environment is ready to be launched.

Idea for ros: The channels related to the movement and the action that has to be forwarded to the robot has the same name for all the robots. To have a major level of security on what is happening in the real world, a different name can be given to each topic of each robot. In this way, every robot will have a dedicated topic into which publish the action and the location that it has to do or reach. In this way, the Ros interpreter, on robot side, will be sure to not execute a wrong command.
For the topic that interacts with the interface and the Observer, is better

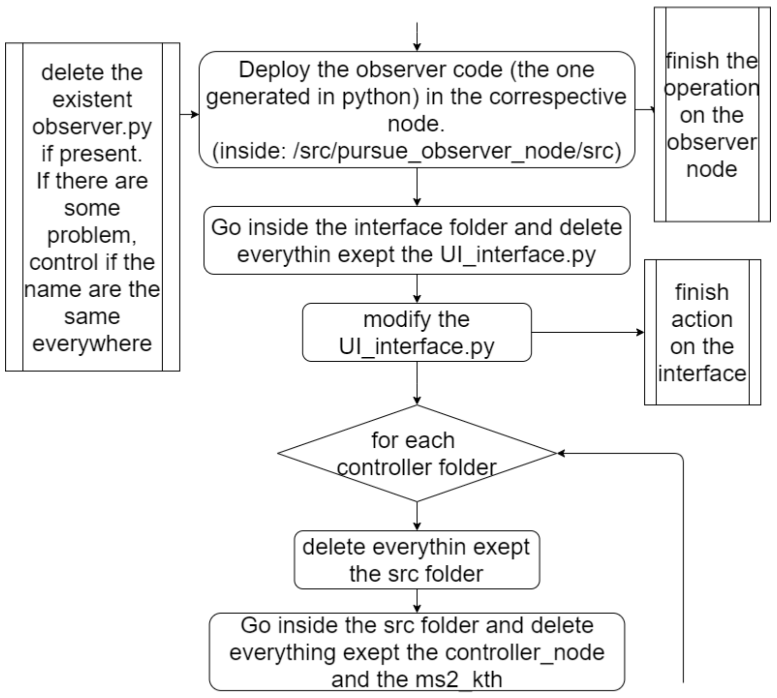

delete the existent observer.py if present. If there are some problem, control if the name are the same everywhere
Deploy the observer code (the one generated in python) in the correspective node. (inside: /src/pursue_observer_node/src)
finish the operation on the observer node
Go inside the interface folder and delete everythin exept the UI_interface.py
modify the UI_interface.py
finish action on the interface
for each controller folder
delete everythin exept the src folder
Go inside the src folder and delete everything exept the controller_node and the ms2_kth

Figure 4.8: Ros adaption algorithm: part 2

that they maintain their name, or it will become complex to manage all the possible interconnections.

4.2.7 Interactions between the Run-Time agents

In Figure 4.10 is shown the new Time Line sequence Diagram. The basic features are similar to the previous one but, instead of only one, there are n controller_nodes and one robots. The start signal came from the Interface. The Observer_node gives the information on the system state to the Controller_nodes, as shown in the red box. The controllers now start the computation of the commands and the commands are forwarded and executed by the robots. The new feature is that the finish of an action or a movement notification is given by the Interface or by the external world. Not through the controllers, but by the robot as shown in the blue boxes.
While the conception is closed to the single robot design, more complications can arise due to the overlap of events and due to the more frequent update of the system state computed by the Observer_node. This complexity, anyway, does not depends on the distributed proposal, but on the number of the

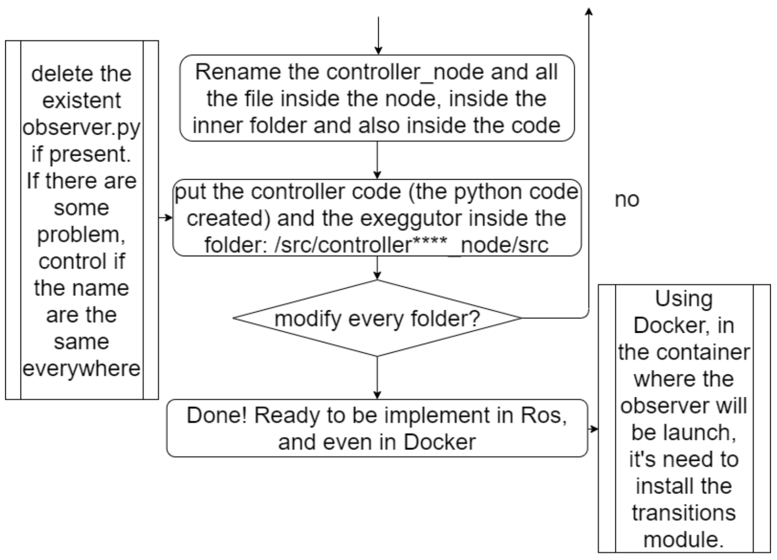

delete the existent observer.py if present. If there are some problem, control if the name are the same everywhere
Rename the controller_node and all the file inside the node, inside the inner folder and also inside the code
put the controller code (the python code created) and the exeggutor inside the folder: /src/controller****_node/src
no
modify every folder?
Using Docker, in the container where the observer will be launch, it's need to install the transitions module.
Done! Ready to be implement in Ros, and even in Docker

Figure 4.9: Ros adaption algorithm: part 3

agents.

4.3 Idea of automation

Up to this point, we have explained the changes in the code. The intention, however, is that the code should not be generated manually but automatically and these changes should be done, not so much in the code itself but, in the code generation mechanism. Unfortunately, in this work, there was no opportunity to intervene in the automatic generation of the code. In the following section we will explain, anyway, which changes would have been made in the main mechanisms for generating the code to obtain the modifications done. On a technological level, the PuRSUE language is written, translated and interpreted through parsers. These parsers have the descriptions of the grammar and, through Design-Time phase, are able to read the PuRSUE specifications and generate the corresponding executable code. To modify the automatic mechanism it is necessary to intervene in some points of the flow of translations. The proposed flow is shown in Figure 4.11 where the created files, in the slightly colored box, and the parsers, in white, are illustrated. The output files are grouped in the red rectangle. To have a comparison refer to Figure 3.5 in Chapter 3. The main changes are just two:

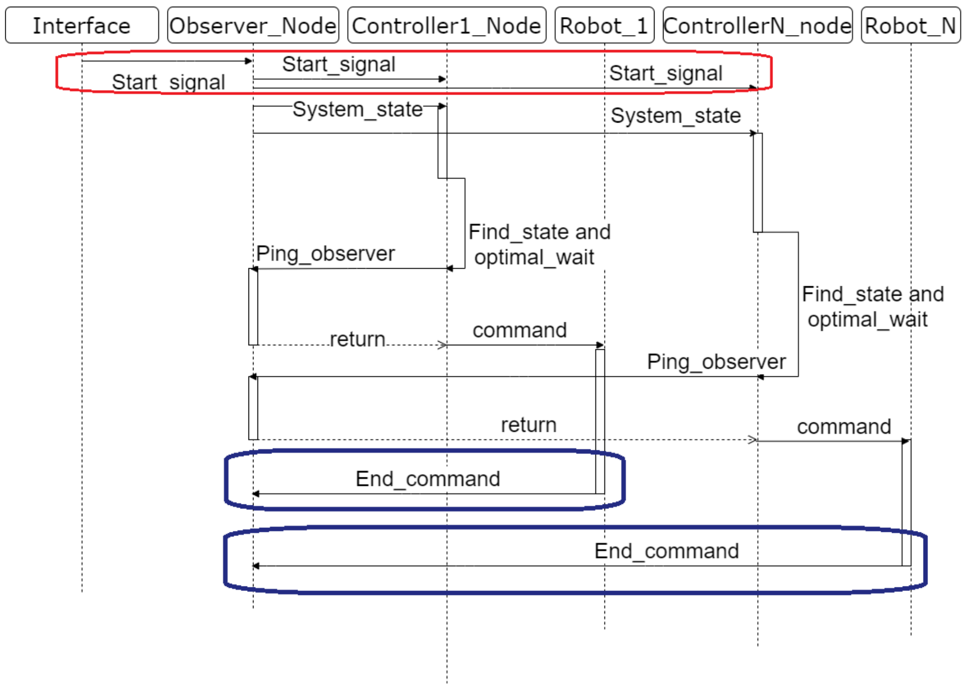


Figure 4.10: Time Line with more agents

- **From UP2CO to UP2SUBCO:** for the parser related to the controller that will now generate all the distributed one. It will take as input the plan generated by UPPAAL-TiGA, as in the previous design choice, but will give as output the divided controllers.
- **A new UP2EX:** a new parser is added, even at the end of the Design-Time mechanism. The new parser will take as input the PuRSUE-ML file and will synthesize the Executor.

4.3.1 UP2SUBCO

For the controllers are present slightly variant solutions for writing the parser. These solutions lead to the same result, here are presented both, a design decision has to be done in the next step.

1. The first thing to determine are the inputs, it has to read the `input.pur` or the file `UPPAAL_model.xml`, to know which are the controllable agents. It also has to read the `UPPAAL_plan.  txt` to know the overall control strategy.

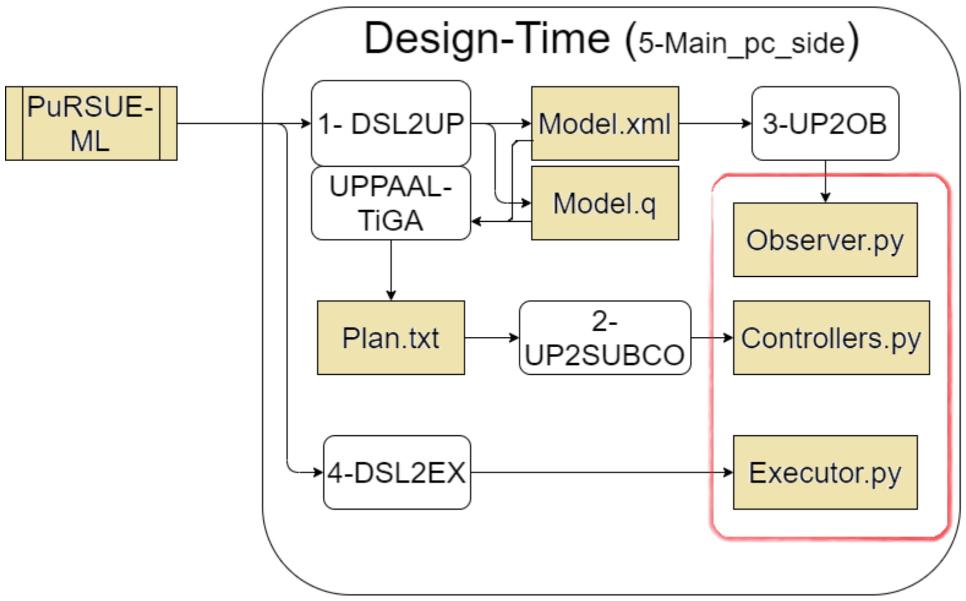


Figure 4.11: New Desing-Time tranlation flow

So in the Main.java related to the sub-controllers will begin with the command:

```
System.out.println("Parser for subcontrollers started");
CharStream in1 = fromFileName("./UPPAAL_plan.txt");
```

Then there will be the command:

```
CharStream in2 =fromFileName("./input\_language/input.pur");
```

or `./UPPAAL/UPPAAL_model.xml`. In this way, into in1, there will be the information of the control strategy while, in in2, there will be the information of the controllable agents. So, this information has to be extracted.

2. For the input `UPPAAL_plan.txt` is better to clear the beginning as done in the previous parser `UP2CO`.
3. The second step is to take the information about the controllable agents. At this version there are no possibilities to distinguish between a robot

and a human allay, so the sub-controllers will be generated for both. There will be just a problem with the deployment on the human being, in the future can be established a way to distinguish them and the parser will be slightly modified.

```
Create an array of string, where save the name of the
controllable agents.
```

The agents names can be composed of alphanumeric characters, so all combinations of them are acceptable, except for \ \ which is the sign of comments in the PuRSUE-ML. From the `input.pur`: the agents are declared after the rule section, so once read the file go after the locations, connections, events and rule. A commented line determines the beginning of the agents' declarations:

```
//agents (must specify if controllable and/or mobile,
initial location action and reactions)
```

Until the commented line that announces the objective declaration, there will be listed, one each line, an agent. In the second position there is its name and in the third is specified if it is controllable or just mobile. So, if the third information is "controllable" the agent's name has to be saved into the name's array.
From `UPPAAL_model.xml`: At the beginning of the file, there are declarations of events, movements, actions and agents. Ignore everything till the comment:

```
 //agents
//nota: rule and agents can not have the same name.
```

In the next line there is the declaration of the clock, ignore the word clocks and saved into the name's array the following name, deleting the "C" character that they present at their beginning. Go further until the \ \.

4. Now it has to create one controller for each name present into the name's array. Create a folder, named `sub-controllers` into which the files will be saved.

5. Begin a "for" cycle, for every name into the name's array the following action will be executed.
   From `UPPAAL_plan.txt`: can be used the classes already existing of the `UP2CO` parser. The input has to be cut, all the *import* has to be written in the same way, but the class name has to have the structure:

   ```
   controller_NAMEBOT
   ```

   The functions `if_start` and `update_state` has to be the same. Just before the end of `update_state`, on the `subcontroller.java` (the file where the command will be written), add the line:

   ```
   builder.append("\n\t\tcontroller_NAMEBOT end update");
   ```

   The function `__init__` is modified in the declaration of the node, from:

   ```
   builder.append("\n\t\trospy.init_node('pursue_controller_node')");
   ```

   becomes:

   ```
   builder.append("\n\t\trospy.init_node('controller_BOTNAME_node')");
   ```

   While in the function `print_state` the commands:

   ```
   this.allstates.forEach(st->builder.append("\n\t\tprint(\""
   +st+"is\",self."+st+")"));

   this.allstates.forEach(st->builder.append("\n\t\tprint(\""
   +cl+"is\",self."+cl+")"));
   ```

   become:

   ```
   this.allstates.forEach(st->builder.append("\n\t\tprint(\""
   +controller_BOTNAME+": "+st+"is\",self."+st+")"));

   this.allstates.forEach(st->builder.append("\n\t\tprint(\""
   +controller_BOTNAME+": "+cl+"is\",self."+cl+")"));
   ```

The run function begins in this way, but only the condition that satisfy the over mentioned condition will be reported. Seeing how the `UPPAAL_plan.txt` is structured, when begin the strategy to win, or not to lose, the command structure presents: State, "When you are in ()" (clock condition) and then:

```
take transition AGENT.WHERE_IT_IS-> AGENT.WHERE_IT_HAS_TO_GO
```

There could be more than one of the "When you are in ()" and "take transition", then there are the conditions of the waiting.
For every "take transition" checks if satisfies the introduced requests, if so use the already existing class to translate it and append them into the run function. Use the same criteria for the multiple possible transitions. If there are transitions of another bot or of the end of an action or an end of a movement, ignore them. These transitions can be identified in the AGENT.WHERE_IT_HAS_TO_GO and AGENT.WHERE_IT_IS part. If the state passes from a doing to a DONE, or from a going to a location, the relative instruction has to be deleted.
In the `main()` has to be changed the name of the controller's class, the command in the java class has to change from:

```
builder.append("def main():\n\tcontrollore=Runtime_
controler()\n\tcontrollore.run()\nif_name __name__ ==
\"__main\":\n\tcontrollore =Runtime_controller()\n\t
controllore.run()\n");
```

becomes:

```
builder.append("def main():\n\tcontrollore=controler_BOTNAME()
\n\tcontrollore.run()\nif_name __name__ ==
\"__main\":\n\tcontrollore=controller_BOTNAME()
\n\tcontrollore.run()\n");
```

6. The lexer, token and parser variable will be initialized and called as in the already existing parser, and the output will be translated into a string type.

7. Will be created a file where save the controller through the command:

   ```
   BufferWriter writer = new BufferWriter(new FileWriter
   (./subcontrollers/controller_BOTNAME));

   writer.write(output);
   writer.close();
   ```

8. Do this operation for each agent, so for each name in the name's array.

9. Print on the prompt the end of the parsing.

   ```
   System.out.println("Parser for subcontrollers finished");
   ```

4.3.2 UP2EX

Here is described how the new parser of the `Executor.py` can be written. Again it presents some slight variability that can be evaluated in the designing phases.

1. The only input that this parser need is the `input.pur` file, because it contains all the information needed. This information is the name of the POI and the corresponding positions.
   Idea for Ros: in order to implement even this feature, the `input.pur` is again enough but also the information related to the controllable agent is needed.

2. The Main.java related to the Executor parser will begin with the command:

   ```
   System.out.println("Parser for Executor started");
   CharStream in1 =fromFileName("./input_language/input.pur");
   ```

3. Create a variable into which saved the POI name and coordinates, so it has to be an array of objects with a data structure of eight fields. The first field is where to save the name of the P.O.I, the other seven are used to save the three spatial coordinates and the four of the orientation.
   Idea for Ros: for this idea is needed to create another array of string where to save the name of the controllable agent. Even here is maintained the problem for the distinction between a robot agent and a human ally.

4. Now the POI has to be identified. Here can be present a design choice, the advisable one is to maintain the coordinate written in the *input.pur* in the format:

   ```
   poi "NAME" //[x, y, z, nu, ex, ey, ez]
   ```

   In this way, the information is at the beginning of the document, after the comment "//location" and the "//" are useful to avoid changes in the `DLS2UP` parser. Until the comment "//connection" save the POI into the array of objects just created.So can be identified as, after the name, appear the characters "//[", the information is separated by a ",", the number can present some decimal, identified with the use of the dot (e.g. 0.20), and they end with a "]". Save into the array every location.
   Idea for Ros: for this function is also needed to take the name of the controllable agent, it can be used the strategy already described for the controllers.

5. Create a string variable into which compose the `Executor.py` as described in Section 4.2.3. So it has to have the same functions, just the dictionary has to be changed for different scenarios. Everything before the position's dictionary has to be maintained, and then the position dictionary has to be declared. The array of the coordinates has to be written into the dictionary with a similar structure. The file has to end in the already defined way.

6. Create a file, into the *output_files* or into the previously created `sub-controllers` or into a dedicated folder, a file named `exeggutor.py`[5]. Write inside it the string variable where all the Executor code is saved, as did for the controllers:

   ```
   BufferWriter writer = new BufferWriter(new FileWriter
   (./output_files/Exeggutor));

   writer.write(output);
   writer.close();
   ```

[5]it has to have this name, written wrong.

Idea for Ros: the previous two steps have to be iterated for every controllable agent, and added some changes into the Move_command_sender and Action_sender functions. Here the hypothetical command:

```
builder.append("self.pub = rospy.Publisher('move_base_simple/goal',
PoseStamped, queue_size = 10)");

builder.append(""self.pub = rospy.Publisher('pursue/action',
String, queue_size = 10)");
```

has to be replaced by:

```
builder.append("self.pub = rospy.Publisher('"+BOT_NAME+"
_move_base_simple/goal', PoseStamped, queue_size = 10)");

builder.append(""self.pub = rospy.Publisher('pursue/"+
BOT_NAME+"_action',String, queue_size = 10)");
```

A new file has to be created for each controllable agent.

7. End the Main.java with the notification of the end of the parsing.

```
System.out.println("Parser for Executor finish");
```

Other needed adaption: If the Res2Ros structure is maintained, also the Main.java of the folder 4, `main_pc_side` [6], has to be modified. All the new files have to be sent to MissionSender. It is advisable to create a *for* cycle to send all files inside the correct folders: `sub-controllers` and the `Executor.py` (or all of them).

[6] For more information see Chapter 3 or Appendix B

Chapter 5

Evaluation

This section reports on the experiments that have been carried out to validate the modified PuRSUE Run-Time mechanisms. First, to check whether with the proposed modifications the system still works correctly, the new proposal was tested on the Ros middleware. So, some simple scenarios have been written in PuRSUE-ML and have been translated according to the framework. The `Controllers.py` file is manually split into many separate files, according to the approach presented in Chapter 4. Also, the created `Observer.py` is modified and the `Executor.py` is created with the POI of the scenario being created and obviously with the structure that includes the functions to forward the commands.

Ros implements a Publish-Subscribe mechanism, in this way the nodes can exchange the messages that contains the information. Initially, all the code was contained in the same folder and, in these conditions, the Run-Time mechanism was launched in Ros. All the nodes (the Controller_nodes, the Observer_node and the Interface) were able to communicate providing, in this way, the first positive feedback.
Subsequently, we decided to separate the node's folders in order to verify that the communication was not conditioned by the fact that all the code was present in the same folder. So, one folder for each node was created and the respective workspace was built. Even in these conditions, communication has always taken place.
To be completely sure that the positive result was not conditioned by the sharing of the same host system, and that therefore in some way the presence of the other nodes was recognized, we decided to try to simulate the scenarios through Docker containers. In this way, the code was completely isolated

from all the contents of the host, in fact, through this technology, inside the container it was possible to interact only with the (pure) operating system and the generated image of Ros. In order to allow the communication, all the containers have been added to the same network using the Docker command that allows this simulation, of the connection to a common network. In these conditions, we try to simulate as close as possible to the real situation, where each node will run on a different device and does not share the Operative System or any other physical components. Even in these conditions, the nodes have been able to communicate and this was another important feedback. Even in a completely isolated environment, communication happens.
In the end, for each scenario are reported two tables that shown the reduction in terms of *if structure* and lines of code (loc) to show the advantages introduced by the distributed infrastructure.
Not only simple scenarios were taken into account, but at last, also a more complex one (Catch the Thief) was tested to show that the introduced artifacts are operative also in a scenario that presents a not-controllable agent.

PC specification The framework, with the new proposed features, was tested on a virtual machine with Ubuntu 18.04.4 LTS, disk of 60GB, 2GB of memory and Intel® core™ i5-7200U CPU @ 2.50GHZ.
The used Ros version: rosdistro melodic, rosversion 1.14.5.
The used Docker version: 19.03.6, build 369ce74a3c.

Ros: Using only the Ros middleware the folders containing the node can share the same workspace. Using only the Ros middleware the folders containing the node can share the same workspace. To separate correctly the files related to the different nodes, a procedure is presented in Chapter 4.
The processes followed to launch the Ros environment, in all the considered configuration, are reported in Appendix A.

Docker: For the use of Docker, a pair of attentions have to be adopted.
The image of Ros has to be created as has to be created the container for the ros master and the network on which all the robot has to connect on.
So, the first container, on which run the Ros master, is created through the command:

```
$ docker run -it --rm  --net pursue  --name master  ros
```

For all other nodes, Docker has to have the possibility to see the folder inside which the node's code is written. To make it possible run the command:

```
 $ docker run -it --rm \
--net pursue \
--name controller1 \
--env ROS_HOSTNAME=controller1 \
--env ROS_MASTER_URI=http://master:11311 \
-v "/home/runtime1/:/root/.ros/"
rosrun controller1_node controller1_node
```

All the node has to be connected at the same network to be able to communicate.
Inside the container of the Observer_node, the system needs the installation of the transitions package before running the node. So, the commands were run:

```
$ sudo apt update
$ sudo apt antrall python-pip
$ pip install transitions
```

For a more detailed report on how the environment was initialized in Docker see Appendix A.

5.1 Tested scenarios

The tested scenarios present a limited number of possible system states since the Run-Time agents are manually modified. This restriction does not affect the general validity of the functioning. The last tested scenario is the Catch the Thief problem, here the agents number is small but the corresponding number of system states possibilities increases due to the presence of an uncontrollable agent.
In all the tested scenarios the Run-Time components are able to communicate with each other, and the final goal was almost always reached.

5.1.1 Scenario 1

The first tested scenario presents the interaction between two robots. The first robot is named `bot1`, while the second one is named `bot2`. Both robots start in the same position, POI "c", the first one ha to go in place "b" and do an action that lasts 3 time units. In the meanwhile the bot2 has to go in place "a", once in place "a" it waits that the bot1 has finished its action. Only when the bot1 has finished, bot2 can do its action, that last 6 time units and it is also the final goal. In Figure 5.1 is reported one possible (but

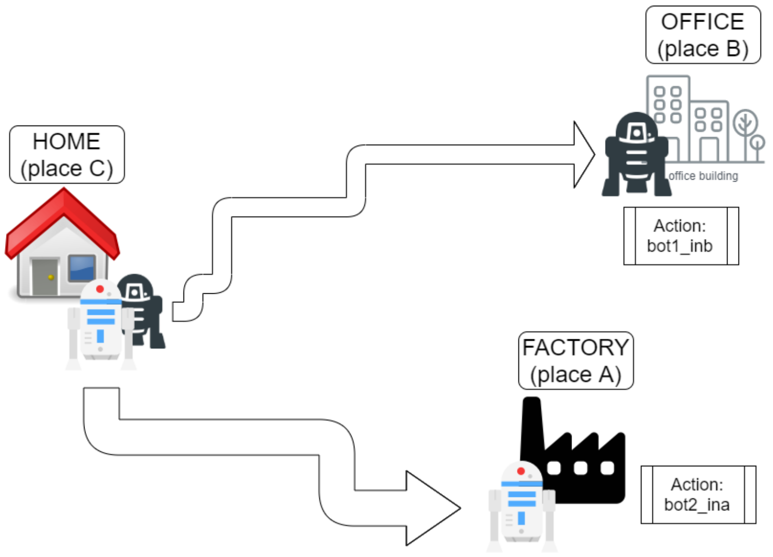


Figure 5.1: A possible interpretation of the case 1 scenario

unrealistic) interpretation of the first scenario. It was thought for an academic propose, to test easily the distributed proposal of the Run-Time phase, so the interpretation in Figure 5.1 is just to give a general representation. A more realistic interpretation of the scenario can suppose that the robots act on a floor with three rooms.
In Listing 5.1 is reported the corresponding PuRSUE-ML file:

```
//locations
poi "a" //[x, y, z, nu, ex, ey , ez],
poi "b" //[x, y, z, nu, ex, ey , ez],
poi "c" //[x, y, z, nu, ex, ey , ez],

//connections

connect a and c distance 3
connect c and b distance 3
```

```

//events
event "bot1_inb" location b duration 3
event "bot2_ina" location a duration 6

//Rule
rule "rule1": bot1_inb before bot2_ina

//agents (must specify if controllable and/or mobile, initial location
    actions
//and reactions)
agent "bot1" controllable mobile 3 location c can_do bot1_inb
agent "bot2" controllable mobile 3 location c can_do bot2_ina

//objectives (only action reaction within time limit for now
reach_objective: do bot2_ina after 0
```

Listing 5.1: PuRSUE-ML of the Scenario 1

All the files are modified following the instruction of the previous chapter, and all the instructions given to run it, are reported in Appendix A.
The infrastructure was started in all the Ros folder's configuration, with successful results. So we proceed to test the scenario also with Docker. Again successful results were reached.
Not only the bots are able to communicate, also the final goal is reached.

To show that with Ros everything works correctly, in Figures 5.2 and 5.3 the node's graph is reported. Show that even with the Docker containers the system works is more difficult, but in Figure 5.4 is reported a picture of the prompt with an extract of the controller2_node printing its system state and of the messages exchanged on the pursue/events topic that notified the reaching of the goal.
In order to have an idea of the advantages of the code, Table 5.1 reports the number of the *if structure* of each `Controller.py` and the relative percentage referred to the general controller. With General Controller we referred to the `Controller.py` generated from the previous PuRSUE framework feature. With Centralized Controller we referred to the General Controller without the *a priori* decisions, so the Controller of a centralized configuration but that leaves the notification of the end of an action or a movement to the external world. With the enumerate Controllers we referred to the single Controller related to a robot, so to the distributed version without the *a*

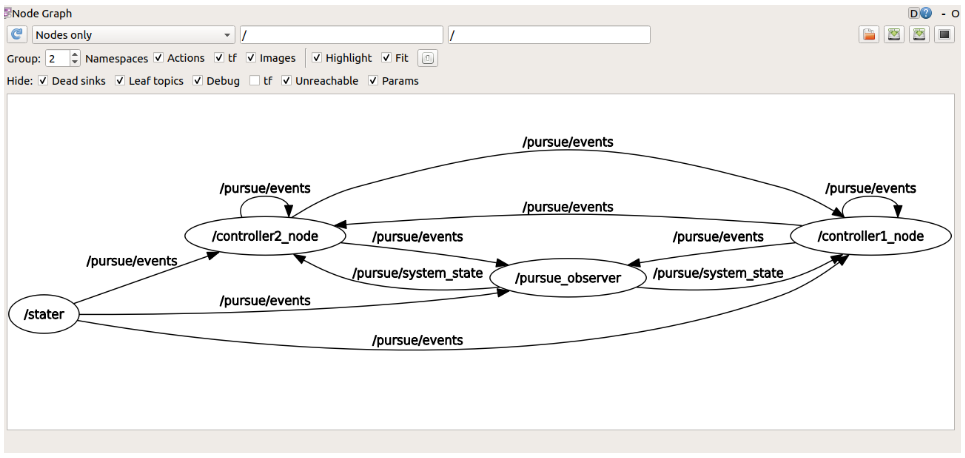


Figure 5.2: Scenario 1: Ros node's graph

priori decisions. While in Table 5.2 is reported the number of loc (lines of code) and their relative percentage referred to, again, the General Controller.
To correctly interpret the values reported in the Tables 5.1 and 5.2, it is

Controller	Number of "if"	Percentage
General Controller	17	100%
Centralized Controller	9	52,9%
Controller1	4	23,5%
Controller2	5	29,4%

Table 5.1: Scenario 1's comparative table: if structure

important to mention that the instruction related to the variable `reachObj` is reported only on the `Controller2.py` file. For this reason, it has one *if structure* more than the `Controller1.py`.
It is important to take into consideration also, for the Table 5.2, that the initialization and the beginning function are recopied more or less like the ones presented in the General controller. For this reason, the percentage results higher of those relatives to the *if structure*.

5.1.2 Scenario 2

The second scenario is a small variation form of previous one. To test that the artifacts will work even with more than two robots, another one is added. Being the controllers split by hand the scenario is again simple, but it presents

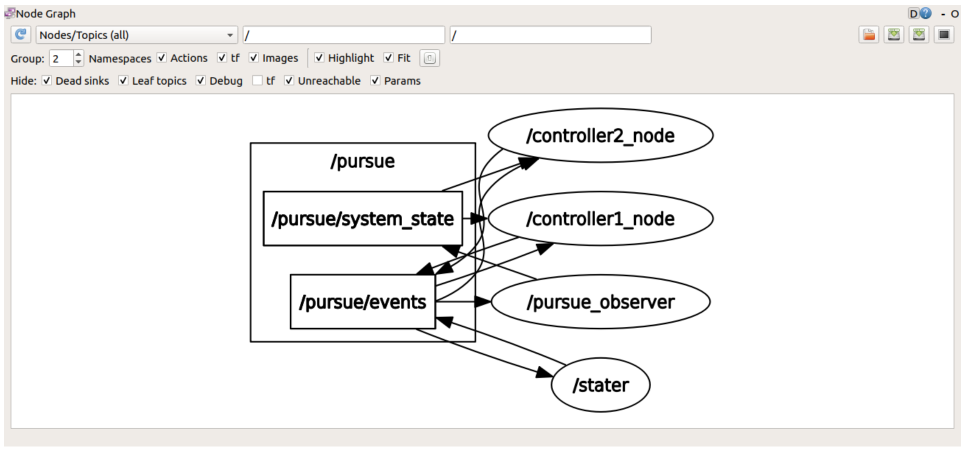


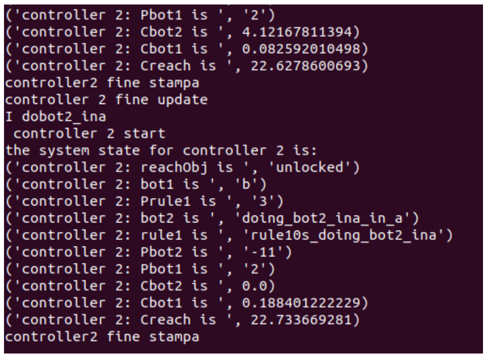
Figure 5.3: Scenario 1: Ros node's graph

Controller	**Number of loc**	**Percentage**
General Controller	425	100%
Centralized Controller	278	65,4%
Controller1	182	43%
Controller2	203	48%

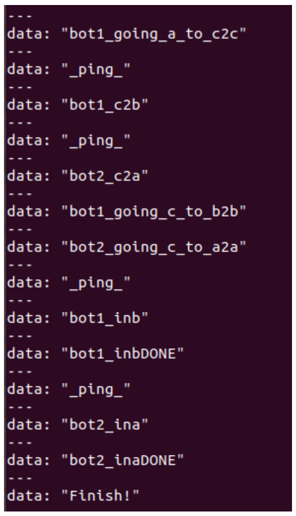
Table 5.2: Scenario 1's comparative table: loc

three controllable robots interacting with each other. The added robot just follows bot2 and helps it in doing its action, so the action they performed together was modeled as a collaborative event.
In Listing 5.2 is reported the corresponding PuRSUE-ML file:

```

//locations
poi "a" //[x, y, z, nu, ex, ey , ez],
poi "b" //[x, y, z, nu, ex, ey , ez],
poi "c" //[x, y, z, nu, ex, ey , ez],

//connections

connect a and b distance 3
connect a and c distance 3
connect c and b distance 3

```



```
('controller 2: Pbot1 is ', '2')
('controller 2: Cbot2 is ', 4.12167811394)
('controller 2: Cbot1 is ', 0.082592010498)
('controller 2: Creach is ', 22.6278600693)
controller2 fine stampa
controller 2 fine update
I dobot2_ina
 controller 2 start
the system state for controller 2 is:
('controller 2: reachObj is ', 'unlocked')
('controller 2: bot1 is ', 'b')
('controller 2: Prule1 is ', '3')
('controller 2: bot2 is ', 'doing_bot2_ina_in_a')
('controller 2: rule1 is ', 'rule10s_doing_bot2_ina')
('controller 2: Pbot2 is ', '-11')
('controller 2: Pbot1 is ', '2')
('controller 2: Cbot2 is ', 0.0)
('controller 2: Cbot1 is ', 0.188401222229)
('controller 2: Creach is ', 22.733669281)
controller2 fine stampa
```



```
---
data: "bot1_going_a_to_c2c"
---
data: "_ping_"
---
data: "bot1_c2b"
---
data: "_ping_"
---
data: "bot2_c2a"
---
data: "bot1_going_c_to_b2b"
---
data: "bot2_going_c_to_a2a"
---
data: "_ping_"
---
data: "bot1_inb"
---
data: "bot1_inbDONE"
---
data: "_ping_"
---
data: "bot2_ina"
---
data: "bot2_inaDONE"
---
data: "Finish!"
```

Figure 5.4: Scenario 1 prompt's screens

```
//events
event "bot1_inb" location b duration 3
event "bot2_ina" collaborative location a duration 6

//Rule
rule "rule1": bot1_inb before bot2_ina

//agents (must specify if controllable and/or mobile,
//initial location actions and reactions)
agent "bot1" controllable mobile 3 location c can_do bot1_inb
agent "bot2" controllable mobile 3 location c can_do bot2_ina
agent "bot3" controllable mobile 3 location c reacts_to bot2_ina

//objectives (only action reaction within time limit for now
reach_objective: do bot2_ina after 0
```

Listing 5.2: PuRSUE-ML of the Scenario 2

In Ros the system responds correctly, in all the separation levels.
In order to show the communication in the Ros environment, in Figure 5.5 is reported the corresponding Ros graph. It shows the nodes' interaction through the topics.
In this scenario the goal is reached, it was tested also with the Docker containers. Again the results were positive both for the communication and the

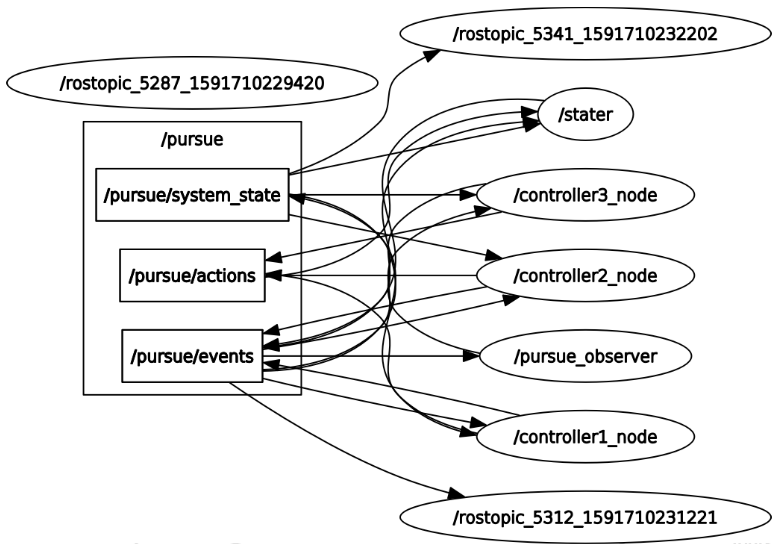


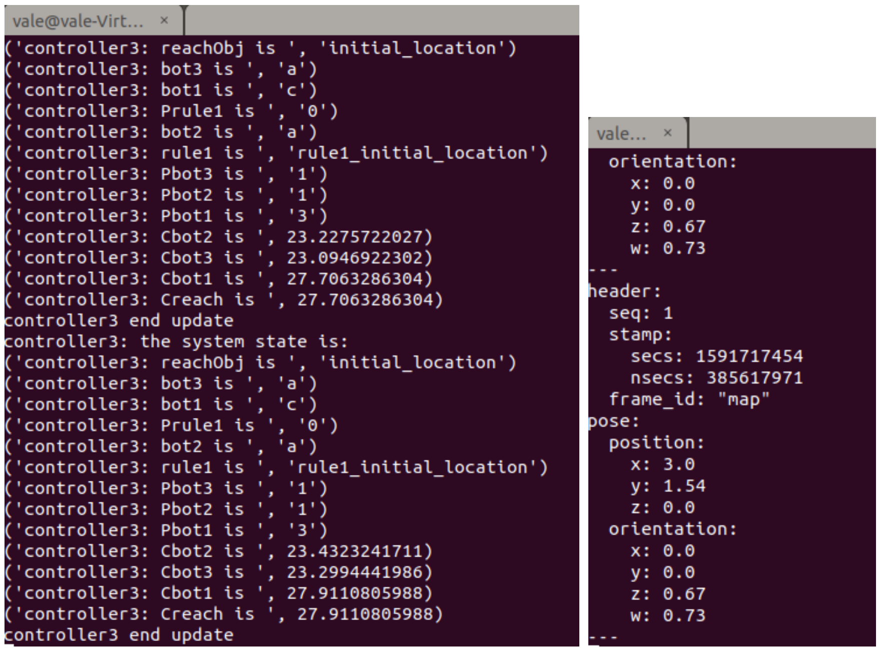
Figure 5.5: Scenario 2: Ros node's graph

reaching of the goal.
Screens of the prompt are reported in Figure 5.6, where is shown the Controller3_node printing its state and the messages exchanged on the topic `move_base_simple\goal`.
In order to have feedback also on the effectiveness of the split, in the Tables

Controller	**Number of "if"**	**Percentage**
General Controller	20	100%
Centralized Controller	10	50%
Controller1	6	30%
Controller2	3	15%
Controller3	2	10%

Table 5.3: Scenario 2's comparative table: if structure

5.3 and 5.4, are reported the percentage of the *if structure* and the loc of the distributed controllers referred to the General Controller. In this scenario, the commands relative to the `reachObj` are reported in all the controller's nodes.



```
vale@vale-Virt... ×
('controller3: reachObj is ', 'initial_location')
('controller3: bot3 is ', 'a')
('controller3: bot1 is ', 'c')
('controller3: Prule1 is ', '0')
('controller3: bot2 is ', 'a')
('controller3: rule1 is ', 'rule1_initial_location')
('controller3: Pbot3 is ', '1')
('controller3: Pbot2 is ', '1')
('controller3: Pbot1 is ', '3')
('controller3: Cbot2 is ', 23.2275722027)
('controller3: Cbot3 is ', 23.0946922302)
('controller3: Cbot1 is ', 27.7063286304)
('controller3: Creach is ', 27.7063286304)
controller3 end update
controller3: the system state is:
('controller3: reachObj is ', 'initial_location')
('controller3: bot3 is ', 'a')
('controller3: bot1 is ', 'c')
('controller3: Prule1 is ', '0')
('controller3: bot2 is ', 'a')
('controller3: rule1 is ', 'rule1_initial_location')
('controller3: Pbot3 is ', '1')
('controller3: Pbot2 is ', '1')
('controller3: Pbot1 is ', '3')
('controller3: Cbot2 is ', 23.4323241711)
('controller3: Cbot3 is ', 23.2994441986)
('controller3: Cbot1 is ', 27.9110805988)
('controller3: Creach is ', 27.9110805988)
controller3 end update
```

```
vale... ×
  orientation:
    x: 0.0
    y: 0.0
    z: 0.67
    w: 0.73
---
header:
  seq: 1
  stamp:
    secs: 1591717454
    nsecs: 385617971
  frame_id: "map"
pose:
  position:
    x: 3.0
    y: 1.54
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.67
    w: 0.73
---
```

Figure 5.6: Scenario 2 prompt's screens

5.1.3 Scenario 3

This is again a little complication of the first scenario. Another robot is added, but this time the third robot acts when the first two have finished. The action of the third robot is also the goal of the scenario. So the bot1 begins and does its action, then the bot2 performs its action and, after them, the bot3 begins its duty. In Listing 5.3 is reported the corresponding PuRSUE-ML file:

```
    //locations
poi "a" //[x, y, z, nu, ex, ey , ez],
poi "b" //[x, y, z, nu, ex, ey , ez],
poi "c" //[x, y, z, nu, ex, ey , ez],

//connections

connect a and b distance 3
connect a and c distance 3
connect c and b distance 3

```

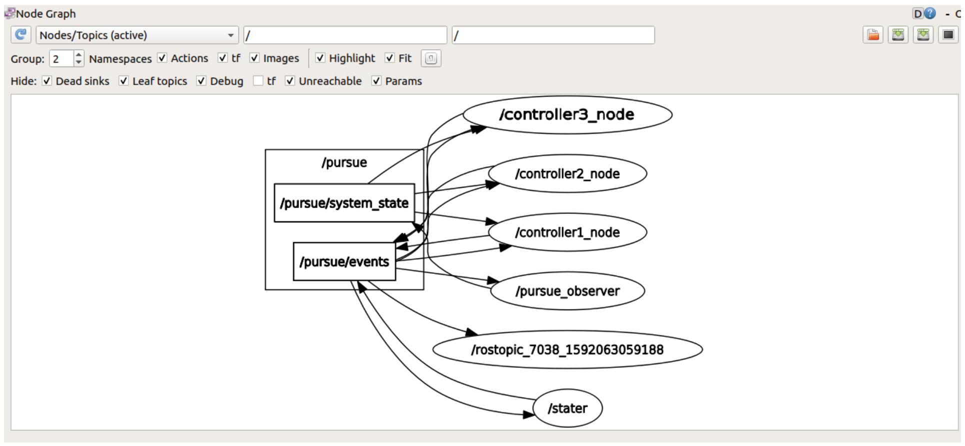
Controller	**Number of loc**	**Percentage**
General Controller	494	100%
Centralized Controller	307	62,1%
Controller1	233	47%
Controller2	176	35,6%
Controller3	152	30,8%

Table 5.4: Scenario 2's comparative table: loc

```
//events
event "bot1_inb" location b duration 3
event "bot2_ina" location a duration 3
event "bot3_Finish" duration 2

//Rule
rule "rule1": bot1_inb before bot2_ina
rule "rule2": bot2_ina before bot3_Finish

//agents (must specify if controllable and/or mobile,
initial location actions and reactions)
agent "bot1" controllable mobile 3 location c can_do bot1_inb
agent "bot2" controllable mobile 3 location c can_do bot2_ina
agent "bot3" controllable mobile 2 location c can_do bot3_Finish

//objectives (only action reaction within time limit for now
reach_objective: do bot3_Finish after 0
```

Listing 5.3: PuRSUE-ML of the Scenario 3

This scenario presents more difficulties to be executed from the previous two, but the system was able to reach the complete goals. The communication works in all levels of isolation. The ros graph is reported in order to show it, in Figure 5.7. In Figure 5.8 are reported the screens of the prompt, one reporting the `Controller1.py` printing its system state and the second one reporting the messages exchanged on the topic `/pursue_events`. Again it was tested with the Docker's containers with positive response. Also there the complete goal is reached. In the Tables 5.5 and 5.6 are reported the percentage of the *if structure* and the number of loc referred to the General Controller.
Also here the instruction relative to the variable `reachObj` is reported in all the controller_node code.


Node Graph

Nodes/Topics (active) / /

Group: 2 ✓ Namespaces ✓ Actions ✓ tf ✓ Images ✓ Highlight ✓ Fit

Hide: ✓ Dead sinks ✓ Leaf topics ✓ Debug ☐ tf ✓ Unreachable ✓ Params

Figure 5.7: Scenario 3: Ros node's graph

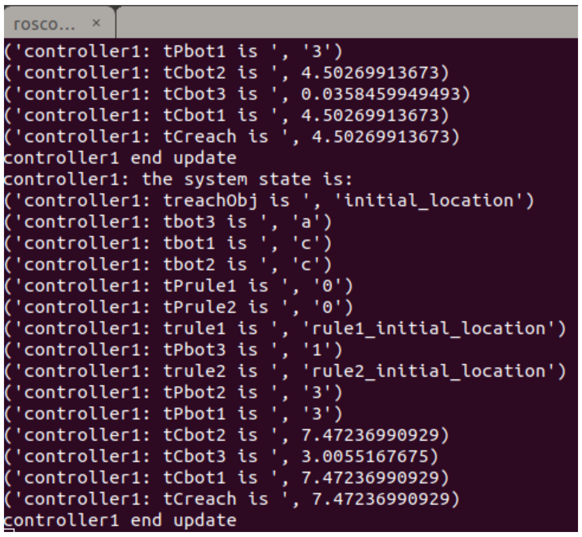
Controller	Number of "if"	Percentage
General Controller	41	100%
Centralized Controller	19	46,3%
Controller1	7	17%
Controller2	5	12,2%
Controller3	9	22%

Table 5.5: Scenario 3's comparative table: if structure

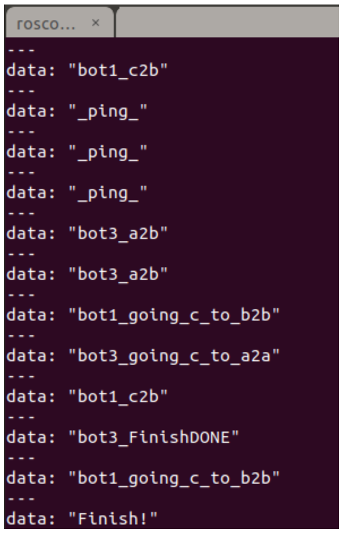
5.1.4 Scenario 4

This scenario is the combination of the second and the third ones. There are four agents, the first has to go from the point "c" to "b" and do an action that takes 3 time unit. After it, the bot2 and bot3 have to go from place "c" to place "a" and do together an action that takes 6 time unit. In the end, the bot4 has to do the last action, the main goal, and this action's location is not specified. In Listing 5.4 is reported the corresponding PuRSUE-ML file:

```

//locations
poi "a" //[x, y, z, nu, ex, ey , ez],
poi "b" //[x, y, z, nu, ex, ey , ez],
poi "c" //[x, y, z, nu, ex, ey , ez],
//connections

connect a and b distance 3
```



```
('controller1: tPbot1 is ', '3')
('controller1: tCbot2 is ', 4.50269913673)
('controller1: tCbot3 is ', 0.0358459949493)
('controller1: tCbot1 is ', 4.50269913673)
('controller1: tCreach is ', 4.50269913673)
controller1 end update
controller1: the system state is:
('controller1: treachObj is ', 'initial_location')
('controller1: tbot3 is ', 'a')
('controller1: tbot1 is ', 'c')
('controller1: tbot2 is ', 'c')
('controller1: tPrule1 is ', '0')
('controller1: tPrule2 is ', '0')
('controller1: trule1 is ', 'rule1_initial_location')
('controller1: tPbot3 is ', '1')
('controller1: trule2 is ', 'rule2_initial_location')
('controller1: tPbot2 is ', '3')
('controller1: tPbot1 is ', '3')
('controller1: tCbot2 is ', 7.47236990929)
('controller1: tCbot3 is ', 3.0055167675)
('controller1: tCbot1 is ', 7.47236990929)
('controller1: tCreach is ', 7.47236990929)
controller1 end update
```



```
---
data: "bot1_c2b"
---
data: "_ping_"
---
data: "_ping_"
---
data: "_ping_"
---
data: "bot3_a2b"
---
data: "bot3_a2b"
---
data: "bot1_going_c_to_b2b"
---
data: "bot3_going_c_to_a2a"
---
data: "bot1_c2b"
---
data: "bot3_FinishDONE"
---
data: "bot1_going_c_to_b2b"
---
data: "Finish!"
```

Figure 5.8: Scenario 3 prompt's screens

Controller	**Number of loc**	**Percentage**
General Controller	862	100%
Centralized Controller	447	51,9%
Controller1	248	28,8%
Controller2	205	23,8
Controller3	285	33,1%

Table 5.6: Scenario 3's comparative table: loc

```
connect a and c distance 3
connect c and b distance 3

//events
event "bot1_inb" location b duration 3
event "bot2_ina" collaborative location a duration 6
event "bot4_Finish" duration 2

//Rule
rule "rule1": bot1_inb before bot2_ina
rule "rule2": bot2_ina before bot4_Finish

//agents (must specify if controllable and/or mobile, initial
//location actions and reactions)
agent "bot1" controllable mobile 3 location c can_do bot1_inb
agent "bot2" controllable mobile 3 location c can_do bot2_ina
agent "bot3" controllable mobile 3 location c reacts_to bot2_ina
agent "bot4" controllable mobile 2 location c can_do bot4_Finish

//objectives (only action reaction within time limit for now
reach_objective: do bot4_Finish after 0
```

Listing 5.4: PuRSUE-ML of the Scenario 4

The Ros environment is started and, but the results were just partially positive. The robots are able to communicate but they can accomplish just the second action, the one done by the bot2 and bot3, but not the final one.
Due to the fact that the bot are able to communicate with each other in all

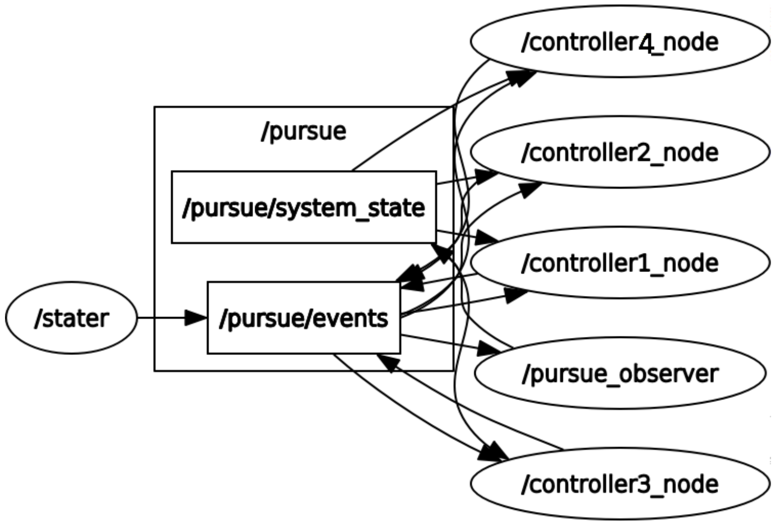
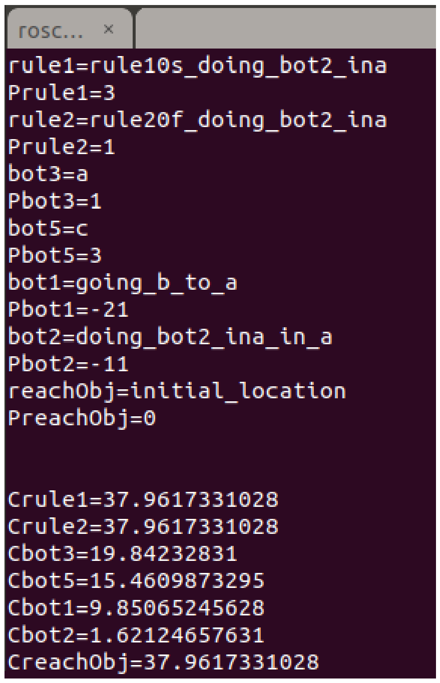
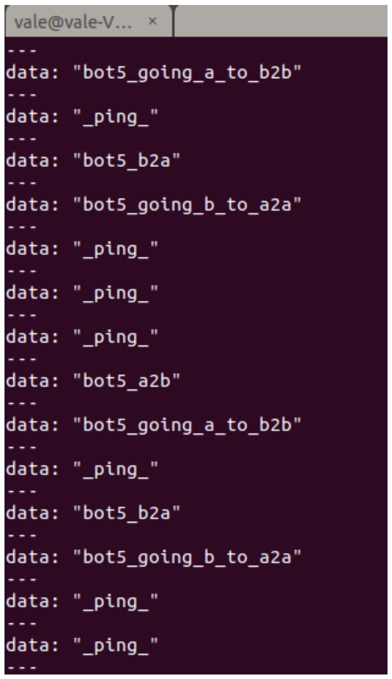


Figure 5.9: Scenario 4: Ros node's graph

the isolation levels, the theoretical feasibility is not disproved.
In Figure 5.9 is reported the ros graph in order to give a proof of the functioning. Also, screens of the prompt reporting the communication through the topics are reported in Figure 5.10.
This scenario was tested also in Docker as the previous ones. As expected it was not able to reach the final goal neither there, the same results given with Ros were reached. This confirms that the communication takes place even in an isolated environment. In the Tables 5.7 and 5.8 report the percentage



```
rule1=rule10s_doing_bot2_ina
Prule1=3
rule2=rule20f_doing_bot2_ina
Prule2=1
bot3=a
Pbot3=1
bot5=c
Pbot5=3
bot1=going_b_to_a
Pbot1=-21
bot2=doing_bot2_ina_in_a
Pbot2=-11
reachObj=initial_location
PreachObj=0


Crule1=37.9617331028
Crule2=37.9617331028
Cbot3=19.84232831
Cbot5=15.4609873295
Cbot1=9.85065245628
Cbot2=1.62124657631
CreachObj=37.9617331028
```



```
---
data: "bot5_going_a_to_b2b"
---
data: "_ping_"
---
data: "bot5_b2a"
---
data: "bot5_going_b_to_a2a"
---
data: "_ping_"
---
data: "_ping_"
---
data: "_ping_"
---
data: "bot5_a2b"
---
data: "bot5_going_a_to_b2b"
---
data: "_ping_"
---
data: "bot5_b2a"
---
data: "bot5_going_b_to_a2a"
---
data: "_ping_"
---
data: "_ping_"
---
```

Figure 5.10: Scenario 4 prompt's screens

of the *if structure* and the number of loc referred to the General Controller. The command relative to the variable `reachObj` is again reported in all the controller_node.

5.1.5 Simple Catch the Thief Scenario

This scenario is far more complex than the previous ones because provides the presence of a not controllable agent.

It aims to model the simplest catch the thief scenario[1] introduced in the Introduction 1, which sums up also the main aim of the PuRSUE framework proposal: to create an environment that can coordinate more robots to achieve a common goal in presence of uncontrollable agents.
Here two polBot are present acting on an simpliefied floor's map, reported in Figure 1.1, both bots start in place "a" and have to catch the thief (the uncontrollable agent) who starts in place "c".
In Listing 5.5 is reported the corresponding PuRSUE-ML file:

[1]This scenario is inspired by one proposed in the previous thesis [1, 2].

Controller	Number of "if"	Percentage
General Controller	25	100%
Centralized Controller	13	52%
Controller1	4	16 %
Controller2	3	12 %
Controller3	2	8%
Controller4	7	28%

Table 5.7: Scenario 4's comparative table: if structure

Controller	Number of loc	Percentage
General Controller	609	100%
Centralized Controller	402	66%
Controller1	212	34,8%
Controller2	193	31,7%
Controller3	171	28,1%
Controller4	266	43,7%

Table 5.8: Scenario 4's comparative table: loc

```
 //locations
poi "a" //[x, y, z, nu, ex, ey , ez],
poi "b" //[x, y, z, nu, ex, ey , ez],
poi "c" //[x, y, z, nu, ex, ey , ez],
poi "d" //[x, y, z, nu, ex, ey , ez],

//connections

connect a and b distance 3
connect b and c distance 6
connect c and d distance 3
connect d and a distance 6

//events
event "catch" collaborative

//agents (must specify if controllable and/or mobile,
initial location actions and reactions)

agent "polBot1" controllable mobile 1 location a can_do catch
```

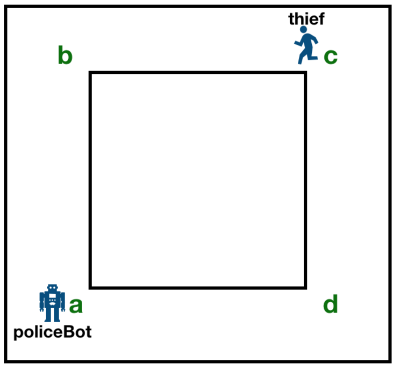


Figure 5.11: Catch the Thief simple map

```
agent "polBot2" controllable mobile 1 location a can_do catch
agent "thief" mobile 2 location c reacts_to catch

//objectives (only action reaction within time limit for now)
reach_objective: do catch after 0
```

Listing 5.5: PuRSUE-ML of the Scenario Catch the Thief

Here some of the system specifications can be noticed, as the assumption that the thief is slower than the robots [1].

This General Controller is really huge, in order to take into consideration all the necessary variation a dedicated controller of the inner variable reachObj was created. This controller is the bigger one, this is because the reachObj variable has many options to change its state, but they can not be merged into a more inclusive one.
The Ros environment, again, allows the communication and the goal is reached even with a simple enemy strategy given through the Interface.
In Figure 5.12 is reported the ros graph where is shown the connections between the acting nodes. Also, the screens of the polBots and the messages exchanged on the topic are provided in Figure 5.13. The scenario was again

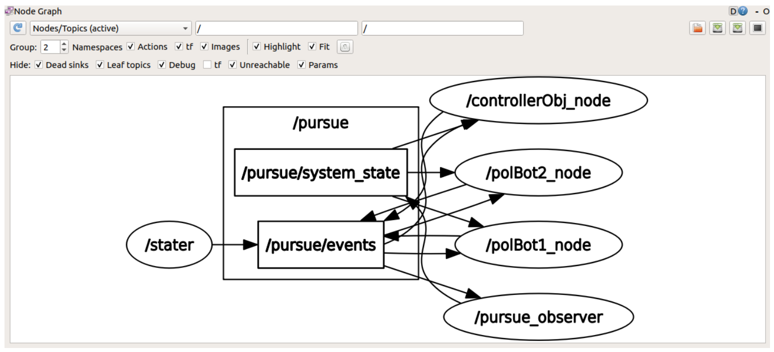


Figure 5.12: Scenario Catch the thief: Ros node's graph

Controller	**Number of "if"**	**Percentage**
General Controller	779	100%
Centralized Controller	539	69,2%
polBot1	170	21,8%
polBot2	172	22,1%
controllerReachObj	195	25%

Table 5.9: Catch the thief's comparison table: if structure

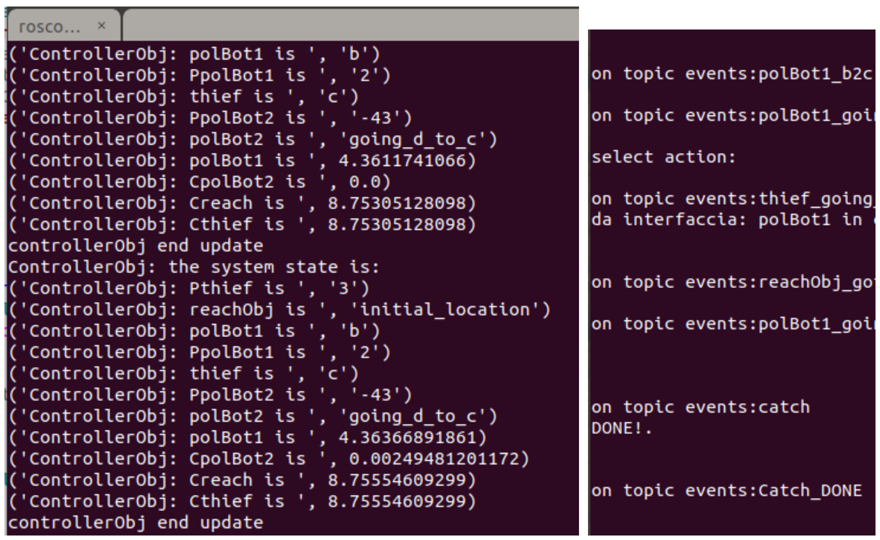
tested with Docker and positives results were achieved, even with the simple enemy strategy simulation. In Tables 5.9[2] and 5.10 are again reported the

Controller	**Number of loc**	**Percentage**
General Controller	14697	100%
Centralized Controller	12706	86,5%
polBot1	5652	38,5%
polBot2	5795	39,4%
controllerReachObj	7070	48,1%

Table 5.10: Catch the thief's comparison table: loc

number of the *if structure* and the loc referred to the General Controller. The first thing that is noticed is that, even if is a relatively simple case, the presence of an uncontrollable agent makes the control strategy explode.

[2]A note on Table 5.9: the *if structure* is computed as the number of commands presented, so the number of the *exeggute* invocation. This because some if *structure* presents a double command that depends on the clocks conditions. If the total number would have been computed manually, it will have presented errors due to the huge commands' number.



```
rosco... ×
('ControllerObj: polBot1 is ', 'b')
('ControllerObj: PpolBot1 is ', '2')
('ControllerObj: thief is ', 'c')
('ControllerObj: PpolBot2 is ', '-43')
('ControllerObj: polBot2 is ', 'going_d_to_c')
('ControllerObj: polBot1 is ', 4.3611741066)
('ControllerObj: CpolBot2 is ', 0.0)
('ControllerObj: Creach is ', 8.75305128098)
('ControllerObj: Cthief is ', 8.75305128098)
controllerObj end update
ControllerObj: the system state is:
('ControllerObj: Pthief is ', '3')
('ControllerObj: reachObj is ', 'initial_location')
('ControllerObj: polBot1 is ', 'b')
('ControllerObj: PpolBot1 is ', '2')
('ControllerObj: thief is ', 'c')
('ControllerObj: PpolBot2 is ', '-43')
('ControllerObj: polBot2 is ', 'going_d_to_c')
('ControllerObj: polBot1 is ', 4.36366891861)
('ControllerObj: CpolBot2 is ', 0.00249481201172)
('ControllerObj: Creach is ', 8.75554609299)
('ControllerObj: Cthief is ', 8.75554609299)
controllerObj end update
```

```
on topic events:polBot1_b2c

on topic events:polBot1_goi

select action:

on topic events:thief_going
da interfaccia: polBot1 in 


on topic events:reachObj_go

on topic events:polBot1_goi



on topic events:catch
DONE!.


on topic events:Catch_DONE
```

Figure 5.13: Scenario Catch the Thief prompt's screens

The positive results obtained from this scenario are really encouraging, the communication happen in all the isolation levels and the bots are able to adjust their strategy according to the thief's decisions

5.1.6 Other simulations

Other tests were made to check other properties of the Run-Time components with the distributed configuration. A scenario with a controllable agent without any task was designed and to see if the conditions relative to its variables are present on the control *if structure*. From this simulation was deduced that the `Observer.py` can not be split, for now, and that the in the command computation phase all the system variable are taken in consideration.

5.2 Considerations

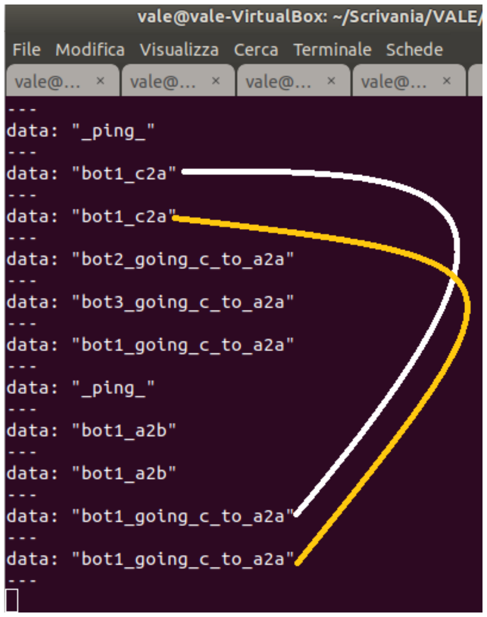
The experiments described above highlight a few crucial aspects, which are discussed in the following.
First of all, the code has been simplified and this has led to an improvement in the overall efficiency of the implementation. The computation of the commands by the controllers is faster and this has brought the possibility to truly succeed in some scenarios that previously were not manageable.

Scenario	**Centralized Time [s]**	**Distributed Time [s]**
Scenario 1	N/A	37,8622498512
Scenario 2	N/A	32,0065710515
Scenario 3	67,8209280968	48,3055899143
Scenario 4	N/A	N/A
Catch the Thief	7,7709290983	6,794880867

Table 5.11: Time comparison between the centralized and distributed controllers

These scenarios were not performant because they required a processing time that went beyond the inner conditions and the possibility of responding to the stimuli of the outside world. This long processing time, therefore, led to the stop of the system itself. In particular the removal of the *a priori* notification of the end of an action or a movement leads to a more reasonable approach for the Run-Time implementation. At the same time, the distributed configuration brings the system to be more reactive and robust. Reactive because each controller_node will be faster in the computation of the correct command and robust because, if the strategy allows it, the goal can be reached even if one robot does not respond rightly.
In order to give a more quantifiable idea of the benefits introduced with the new configuration, Table 5.11 reports the time that the scenarios take to reach their goals. The times are taken from both, the centralized and distributed, versions in order to have a comparison between them. The notification of the end of the actions and the movements are committed to the Interface, in order to be comparable.
Some considerations can be deduced from these values. The times of the centralized version of the first and the second scenario are not available because the systems present errors on the clock conditions. This means that, for simple examples, the framework is too rigid and the centralized proposal is too slow to satisfy its constraints. On the other hand, this highlights the improvement of the distributed proposal.
The times on the third scenario are the most significant because both the structures are able to reach the goal and the time related to the distributed one is meaningful less than the one relative to the centralized version.
The data are not available for the fourth scenario, since it can't reach its goal, while for the simple Catch the Thief one the results confirm the improvement. In this last scenario, the time difference between the two proposals is not as evident as for the third scenario but, to contain some errors introduced by

the simulation environment, the time was taken under the hypothesis that the thief does not move.
Only the scenario 5.1.4 was only partially successful, highlighting the limits in the simulation environment which will need to be further investigated. In the case of more than three robots, the scenario needs to be simulated differently, below we will give some indications for future developments, showing the most probable cause of the problems.
The main problem related to the simulation phase is the Interface limitation. As the Interface is designed, it presents a big issue in the management of multiple but equal commands. In Figure 5.14 is reported a screen of the topic


vale@vale-VirtualBox: ~/Scrivania/VALE/
File Modifica Visualizza Cerca Terminale Schede
vale@... ×
vale@... ×
vale@... ×
vale@... ×

```
---
data: "_ping_"
---
data: "bot1_c2a"
---
data: "bot1_c2a"
---
data: "bot2_going_c_to_a2a"
---
data: "bot3_going_c_to_a2a"
---
data: "bot1_going_c_to_a2a"
---
data: "_ping_"
---
data: "bot1_a2b"
---
data: "bot1_a2b"
---
data: "bot1_going_c_to_a2a"
---
data: "bot1_going_c_to_a2a"
---
```

Figure 5.14: The problem of the Interface

`pursue/event` of one of the tested scenarios (5.1.2). It can be taken as an example of the problem of the logic behind Interface.
For how it is designed, once a command is written, by the `Executor.py`, on the topic `/pursue/events` (for example *bot1_c2a*) the interface read it and, after the corresponding time, it publishes the event that notified the conclusion of that event (bot1_going_c_to_a2a). Problems arise once the controller is fast and notifies more than once the command, so the Interface publishes all the end of the commands. This is highlighted, in the Figure 5.14, by the two lines that linked the correlated events.

For simple scenarios, this eventuality is absorbed by its reactivity, but for a slightly more complex one that presents many agents, this causes great confusion and leads to missing the goal. For this reason, before going ahead to the development of the framework, is advisable to build a better testing and simulation system.
A limited number of robots can be taken into consideration also due to the complex computation on the translation phase. It was noticed that, if the number of controllable agents is high, the relative automata present many possible states and the computation on the `Complete package` becomes heavy and slow.
Two singular things were noticed also on the code correlated to the scenario 5.1.4. To give clearer reasoning, take into consideration the comparative Tables 5.7 and 5.8.
The first one is the strange distribution of the *if structure* inside of the single controllers. The third controller has only two of them, and one is related to the inner variable reachObj. It means that the bot3 has only one possible system configuration to go from place "c" to place "a". On the contrary, the bot4 has the major percentage of the *if structure*, it means is the bot with the most number of instructions.
It is strange anyway how the instructions are spread because the total number is seven, one is for the inner variable reachObj but only one is for the final goal. The main result is that, while is modeled that the bot4 could do its action wherever it prefers, but in the controller only when is in one location (POI "b"). In this way, a lot of possibilities to reach the final goal are lost.
After this consideration, a suspect rises: is the control strategy more focused on the bot that has to reach the final goal? Coming back to this example, the control strategy is focused on the action of bot4 and the other comes in second place, and this can be deducted by the disequilibrium on the distribution of the *if structure*.
Another suspected problem is that the control strategy is more rigid than the necessary and, due to the rigid TGA logic behind it, more constrain on clocks and triggers of transition are more rigid. To make an example, again, it can be found in this scenario. The real reason that prevents the reach of the goal is a condition on the clocks that does not allow the triggering of the action.
Another problem related to the control strategy can be linked to the fact that only the possible cases that can emerge on the evolution of the TGA system are taken into consideration. In the scenario where is present a not controllable agent, the control strategy becomes more and more complex and takes into consideration a lot of more possible system configuration. Instead,

in scenarios where only controllable agents act, the control strategy is simpler but does not present all the possible system configuration that will not be reached by the given command. In this way, lots of flexibility is lost. If the controllable system, due to an external factor, goes out from the expected system state, the mechanism just blocks itself.
As last, it was noticed that, even if the number of instruction in the Scenario 5.1.5 are bigger, the percentages remain comparable with one of the simpler scenarios. Even the ones related to the loc values. In the beginning, it was expected that the percentage related to the lines of code will be lower in a more complex case because the fixed lines of the initialization and the updating function will have had less weight. Despite the prevision, the data are similar because the more complex control strategy presents multiple-choice commands relative to the same system condition. For this reason, the same *if* condition is reported in more than one bot's controller (the one split) and these conditions become, also, more complex too.

Chapter 6

Conclusion and Future Work

The goal of this work was to improve the code generation mechanisms of the PuRSUE Framework. With respect to the stated objectives, we have identified the general principles to make the code more efficient and distributed. We have done the experiments that have allowed us to show the goodness of the results obtained, however, it was not possible to complete the automation part of the code generation. In particular, the first goal of this work was to define the already present mechanisms and features of the PuRSUE framework. Even if this goal was not directly concerned with producing tangible developments, it was necessary to be able to identify the weaknesses of the existing framework and lay down the basis for future improvements.
The second goal was to overcome the limitation of the real-system response and the applicability of a distributed structure. New features were proposed to reach these goals, and the experiments carried out showed their achievement. The tested scenarios were principally simple for the reason that they have to be split manually, but there is no reason to think that these limitations can affect the validity of the presented principles. The idea still has general validity.
In all the tested scenarios the robots are able to communicate. Moreover in the scenarios that present less than three robots, they are able to reach their goals, even in a strictly isolated environment (into the Docker's container). This means that the proposed changes on the `Controller.py` and the `Executor.py` actually work.
Since the split controllers are shorter and with less *if structure*, they are faster and this result is confirmed by, even if not many, by the time values taken from the experiments.
The main limitation of the testing phase was described. The Interface

introduces elements of chaos that bring difficulties to the coordination of more than three bots. So it brings to not contemplated system state or to a not feasibility on the constraints. A new system for simulating the external sensors has to be developed before going ahead with the development of the framework.
This situation has highlighted also a strange distribution on the commands through the controllers, the basic idea behind the control strategy has to be checked to determine the classification of the action's priority to be able to design correctly the scenario in the PuRSE-ML.
Another highlighted Run-Time feature that has to be taken into consideration is the fact that every *if structure* takes into account all the variables of the system. It means that even if a bot is not able to accomplish any action and does not influence the decisions, its state is present when the controllers check the system state and it can be translated into a major number of possible cases to take into consideration.
Other small problems were in the automatic translation of the controllers, the algorithm is not able to distinguish between an ally human agent and a controllable robot.
It was also noticed a rigidity due to the clock's constraints for the simple scenario that does not present any uncontrollable agent. Even if this problem may seem considerable, the Framework propose is to manage scenarios that present at least one uncontrollable agent, so the problem is destined to disappear.
Prepare the ros environment can be a long and accurate work, unluckily there is not a way to make it automatic.
The work that the PuRSUE framework still need is considerable but can lead to a very satisfactory result.
In the first place, safety function has to be added. Even just a small signal that notified the end of the functioning of the Run-Time system.
Some functionalities are not still implemented, the features that have to be included also at Design-Time are described in the previous thesis [1, 2].
Relative to the Run-Time phase some work ha to be done. The parser for the `Controller.py` and the `Executor.py` has to be implemented and tested.
A new interface had to be thought of as a new system to simulate the sensors and the event of the end of an action/movement.
A notification for the exit from the Ros-environment has to be added, in this way the possible users can have feedback from the mechanism functioning.
The analysis of how the complexity change with the increasing of the agents and the distance, and also with the possible action, has to be done. It can be interesting maintaining monitored the variation of the number of the *if*

structures and of the lines of code with the previously written variables.
Checking how is important the rigidity of the clocks present in the if to send the commands. It can be interesting also check, in a simple scenario, which is the control strategy and determinate its order of priority.
It means to establish if the `UPPAAL_plan.txt` gives more priority to the bot that has to reach the final goal, so taking in less consideration the events of the other robots, or to do more events as possible. This feature can be really important in the phase of the modeling of the scenario in PuRSUE-ML in order to give the correct priority to the more important event.
Figure out how to implement the connection between Design-Time and Run-Time. So, if maintain the Rest2Ros infrastructure and generalized it or completely developed a new one.
Another important task is deals with the correction of the difference, at the parsing level, that leads to a misunderstanding between the `Observer.py` and the `Controllers.py` for the updating of the `reachObj` variable.
The last idea is the possibility to make this framework scalable and hierarchic.

Appendix A

User-Guide

In the following section, a brief guide for future users and developers is written. it takes inspiration from [28, 29].
It is needed a Linux host, if not it is enough a virtual machine as used in this work.
Ros Middleware is widely used in Rest2Ros and Run-Time[1] phases, make sure to have a recent version installed.
Problems can arise if the rosnodes are continuously invoked and killed because some can not be killed correctly and get in conflict with the master. In this situation close all the Ros instances and relaunch everything, Ros master included.
Through the folders, there are anyway files that give instruction, also form [1].

Users

The first thing to do is to download the code. In the next section, the main instructions to use the PuRSUE Framework are written, both version: centralized and distributed.

Design Time

Here is treated only the part relative to the translations that take place in the Design folder.

1. Write in a text editor the system model in PuRSUE-ML, and save it as `input.pur`

[1]Is indicated with *runtime* the folder in which the Ros environment code is, but it is not necessarily its name

2. Move the file into the package: `CompletePackage/input_language`. Delete the old `input.pur` file if present.
   Alternately modify the existing `input.pur`.

3. Come back on the root folder, in order to be in `CompletePackage` and open it in the prompt.

4. Run the command:

   ```
   $java -jar main_pc_side.jar
   ```

5. Check if it worked. It should have modified the `UPPAAL_plan.txt` and the files inside the folder *output_files*.

If it does not work, there could be some reason. First of all, check if the problem is the executable file of *verifytga* placed inside the folder `UPPAAL`. In this case, just replace it with a new version, it can be downloaded from the Uppaal-Tiga website [24].
A second problem could be due to a little wrong detail in the Main code of the `DSL2UP` folder. See the section for Developers (few pages below), search and modify:

```
"./UPPAAL/verifytga -t0 UPPAAL/UPPAAL_model.xml"
```

into:

```
"./UPPAAL/verifytga -w0 UPPAAL/UPPAAL_model.xml"
```

Or could be necessary to install the package `transitions-0.6.4` [38].

Rest2Ros

See how to make the Rest2Ros operative. This phase is delicate since it takes into account a well-defined server, IP-address and port[2]. Unluckily was not possible had access to that server, so it is shown just how it theoretically works.

1. Rest2Ros is automatically invoked at the end of the Main code of `main_pc_side.jar`, so no command is needed to send the output files to the server.

[2]This server is a pc of the Goteborg University, linked to the Co4Robots project

2. Download the *communication manager* from github [30] and move the folder into the `src` folder of the runtime package.
3. Open a terminal and run the command:

   ```
   $ roscore
   ```

4. Optionally open another terminal and execute the command:

   ```
   $ rqt\_graph
   ```

5. Open another terminal and move into the runtime folder, it should be prepared with all the needed files to support the Ros structure.
6. Run the command:

   ```
   $ catkin\_make
   ```

7. Run the command:

   ```
   $ source devel/setup.bash
   ```

8. Run the command:

   ```
   $ source pursue_designtime.sh
   ```

It could be necessary to install the package:
`ms2_kth-master-1fdf7d12f65a3bea235939e313e9811f189a647e`, ask for it.
It has to be placed in the runtime folder.
Alternatively, it is possible to launch the Communication_manager_node and the reader_node separately.
If it does not work, the output files can be moved manually. Copy the `Observer.py` and paste it inside the folder:

```
runtime/src/pursue_observer_node/src
```

replacing a possible existing one. Follow the same procedure for the `Controller.py` into the folder:

```
run-time/src/pursue_controller_node/src
```

and check they have the same name as the previous file present.
Check anyway all the README file, the communication environment (Rest2Ros) has to run before invoke the Design-Time code.

Run-Time: ROS Centralized version

1. Open a terminal and run the command:

   ```
   $ roscore
   ```

2. Optionally open another terminal and execute the command:

   ```
   $ rqt\_graph
   ```

3. Open another terminal and move into the runtime folder, it should be prepared with all the needed files to support the Ros structure.

4. Run the command:

   ```
   $ catkin\_make
   ```

5. Run the command:

   ```
   $ source devel/setup.bash
   ```

6. Run the command:

   ```
   $ source pursue_runtime.sh
   ```

7. For display the messages exchanged between the nodes, open another terminal in the same folder and run the command:

```
$ rostopic echo name_of_the_topic

 for example:
$ rostopic echo pursue/actions
```

8. At least, it is important to launch the file that simulates the external environment. It can be in a different level of the runtime folder, and can have a slightly different name, as `pursue_UI.py` or `UI_prova.py`. Look for it, then open a terminal in the same folder of the file and then run the command:

   ```
   $ python pursue_UI.py     (or UI_prova.py)
   ```

9. More options will be displayed in the last terminal, choose the one that more fits the proposal. A start signal should be sent to every component, the structure is running.

If it is better to launch the single rosnode, instead of point 6 of the previous list follow this procedure:

1. In the already open terminal, to launch the observer_node run the command:

   ```
   $ rosrun pursue_observer_node pursue_observer_node
   ```

2. Open another terminal in the same folder, execute the point 4 and 5 of the previous list ("$ catkin_make","$ source devel/setup.bash").

3. To launch the controller_node run the command:

   ```
   $ rosrun pursue_controller_node pursue_controller_node
   ```

It should work. If there are problems in launching the nodes, it is possible to execute just the python file. Just go inside the node, into the *src/NODE_FOLDER/src* folder and write on the prompt:

```
$ python controller.py   (or the name of the file of interest)
```

Run-Time: ROS Distributed version

This section is dedicated to run the distributed system, before following the instructions make sure to have copied the runtime folder as many times as the number of the nodes and put inside them once node-folder each. For the `pursue_UI.py` (or `pursue_UI.py`) is not necessary to have the complete runtime folder.
If the nodes are in the same runtime folder, follow the previous section Run-Time: ROS Centralized version, launch the node separately or update the file *pursue_runtime.sh*.

1. Open a terminal and run the command:

   ```
   $ roscore
   ```

2. Optionally open another terminal and execute the command:

   ```
   $ rqt\_graph
   ```

3. Open another terminal and move into one the runtime folder and run the commands:

   ```
   $ catkin_make
   $ source devel/setup.bash
   $ rosrun controller_node controller_node
   (or anyone of the node of interest)
   ```

4. Repeat the previous step for each node being sure to open a new terminal each time and change the contextual folder. If problems in launching the node arise, see the end of *ROS Centralized version* for alternative ways.

5. For display the messages exchanged between the nodes, open another terminal in the same folder and run the command:

   ```
   $ rostopic echo name_of_the_topic

    i.g.:
   $ rostopic echo pursue/actions
   ```

6. To run the file that simulates the external environment, that can have a slightly different name, as `pursue_UI.py` or `UI_prova.py`, open a terminal in the same folder of the file and then run the command:

   ```
   $ python pursue_UI.py     (or UI_prova.py)
   ```

7. More options will be displayed in the last terminal, choose the one that more fits the proposal. A start signal should be sent to every component, the structure is running.

Run-Time: DOCKER Distributed version

In this phase is necessary to have installed Docker, moreover, the node has to be divided into different runtime folders, as described in the chapter Contribution of the Thesis and at the beginning of the previous section Run-Time: ROS Distributed version. This is a quick list of instructions to show how to use Docker in this specific case, for more information see the specific documentation and tutorials as [34] [32], [35], [33] on which this is based.

1. Create the Docker image of Ros through the command:

   ```
   $ docker pull ros
   ```

2. And check if the image is created through:

   ```
   $ docker images
   ```

3. In order to see the created container run the command:

   ```
   $ docker ps -a
   ```

4. In order to see the running container run the command without "-a":

   ```
   $ docker ps
   ```

5. Create the network in which the node will interact thought the command:

   ```
   $ docker network create pursue
   ```

6. Create the fist container on which run the Ros master through the command:

   ```
   $ docker run -it --rm  --net pursue  --name master  ros
   ```

   The addition of "-it" makes the container interactive; "- -rm" delete it once it stops, it is important because the containers occupy a lot of memory and if they are not removed at the end the memory will run out fast. The addition "- -net pursue" adds the container to the previously created network; "- -name master" gives an identification name while "ros" is the images it takes. Inside this container, run the Ros master.

7. Docker can ask for root user id, can be chosen between give it the password every time or do it once and remain registered.

8. In order to run a node inside a container, it has to have the possibility to see the folder inside which the node code is written. To make it possible open another terminal and run the command:

   ```
    $ docker run -it --rm \
   --net pursue \
   --name controller1 \
   --env ROS_HOSTNAME=controller1 \
   --env ROS_MASTER_URI=http://master:11311 \
   -v "/home/runtime1/:/root/.ros/"
   rosrun controller1_node controller1_node
   ```

   See what does the additional specifications made, "- -env ROS_HOSTNAME" gives a name at the host, so at the container. "- -env ROS_MASTER_URI" gives to the current container the URI on which the master runs; "-v " is the most important addition. It gives to the container the link to the folder on which is written the node code, the structure is the following: "/path/of/the/folder/in/the/pc"/:/"path/of/the/folder/on/the/container". Then there is the instruction to run.

9. Move inside the folder of interest and run the usual commands:

```
$ catkin_make
$ source devel/setup.bash
$ rosrun controller_node controller_node
(or anyone of the node of interest)
```

10. Create one container for each node with an analog procedure, with access only to the node relative folder. All the node has to be connected at the same network to be able to communicate between them.
    Inside the container of the Observer, it needs to install the transitions package before running the node. Run the commands:

```
$ sudo apt update
$ sudo apt install python-pip
$ pip install transitions
```

11. At last, create a container for pursue_UI.py (or UI_prova.py) and run inside it:

```
$ python pursue_UI.py     (or UI_prova.py)
```

    As always options will be displayed in the last terminal, choose the one that more fits the proposal. A start signal should be sent to every component, the structure is running.

12. In order to see what happened in Ros, as the running node or the messages exchanged on a topic some operation has to be done. In another terminal run the command:

```
$ docker exec -it master bash
$ source /ros_entrypoint.sh
```

    These instructions will make possible to run other commands inside a container, in this case, the master one. Now run the command for seeing what it is desired, for example:

```
$ rostopic echo pursue/actions
```

If the node presents some problem, could be enough to move in the *src/NODE_FOLDER/src* folder inside the node and run just the python file.

Developers

This section treats only the Design-Time phase because Res2Ros and Run-Time phases are closely structured, changing a part of them will mean change them substantially. For them, modifications are proposed and discussed in all this work.

Design Time

Once downloaded the complete code, open the Design-Time folder, if it is disorientating take the Appendix B of this thesis as a guideline. To modify the code is necessary to work on the original folder, for this reason, they are present next to the `CommpletePackage`.

1. Open the folder that needs to be modified, if more handy use an IDE. The Main code of each folder is a bit encapsulated.
2. Modify whatever it is desired to.
3. Go back in the root folder (the one with the number) and open it in the terminal.
4. Run the command[3]:

   ```
   $ mvn package
   ```

5. Go back inside the root folder (the one with the number) and open the folder named `target`.
6. Move the *name_of_the_folder-jar-with-dependencies.jar* into `CompletePackage` and rename it with the name of the older one (and delete the older one).
7. The previous point can be executed via graphical interface or via terminal command.

[3]Could be necessary have installed Maven and Antlr

Appendix B

PuRSUE Details

B.1 The PuRSUE framework

Here is reported the actual functioning of the framework, going into the code's details.In order to provide more grasp, a brief description of the used tools is reported, then the analysis will follow the order of the over mentioned phases: Design-Time, Rest2Ros and Run-Time.

Xtext

Xtext is an open-source software framework, it generates parser and class model for the abstract syntax tree using a powerful grammar language. The developer has to write a grammar in Xtext's grammar language and a code generator derives an ANTLR parser and the classes for the object model. A great pro of Xtext is that it has a customizable Eclipse-base IDE. [1, 26, 27] It is used in the parsing phase of the process. It is the tool that masters all the translation operation and flow of the control synthesis and its transformation into executable code.

Maven

Maven is a tool used for building and managing Java-based projects based on a Project Object Model (POM) [19, 20]. It is used to manage the parsing phases, in detail it is used to compress all the operations in file with .jar extension. It makes possible to have all the step inside the Complete Package simplifying the overall process.

Antlr

Antlr's name comes from "Another Tool for Language Recognition" [21] and it is a parser generator. It is used for the parsing in the design phase of the PuRSUE Framework, it generates the parser form the grammar and can build and tun the parse tree [22].

TurtleBot

The TurtleBot is a real mobile robot with open-source software [36], it is able to move in the environment and more software applications can be developed or downloaded form http://wiki.ros.org/Robots/TurtleBot. On this site, the complete code used and tutorials for the Ros framework are also available. So, TurtleBot is a complete independent robot with all the necessary code to make it move and with the possibility of developed more complex applications that can include also actions [37].

Co4Robots project

It is a European project for the development of decentralized control for coordination of interaction robots in whose contest the previous work was developed.
Few parameters, as the communication ports in the communication phase, are defined according to the Co4Robots project's specifications. For more information: http://www.co4robots.eu [31].

B.2 Design-Time: How it does it

In this section are analyzed the Main.java file of all the previously mentioned folders. For a deeper level of knowledge please refer directly to the code.

4-main_pc_side

This one, once opened, seems to be the most simple structured package, as shown in Figure B.1. There are not the folders for class files or tests, just the `Main.java`. Despite that, is the folder that manages the coordination of the translations. As first action is declared the package in which the application is developed, followed by more or less twelve lines of import. The Main class is declared and then the main function begins. The all process can be seen in the Listing B.1.

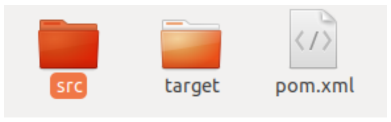


Figure B.1: Folders in "4-main_pc_side"

```

public class Main {
 public static void main(String[] args) throws ClientProtocolException,
     IOException, InterruptedException {
  System.out.println("This method performs the complete parsing from
      DSL to python controller, provided that the folder structure, file
      names and .jar files provided are conform to what's specified in
      readme.txt");
  String s = new String();
  // FIRST process
  System.out.println("///////running first parser and UPPAAL//////");
  Runtime runt = Runtime.getRuntime();
  ProcessBuilder p0 = new ProcessBuilder("java", "-jar", "DSL2UP.jar");

  p0.redirectErrorStream(true);
  Process processo0 = p0.start();
  BufferedReader stdInput0 = new BufferedReader(new
      InputStreamReader(processo0.getInputStream()));
  p0.redirectErrorStream(true);
  Process processo0 = p0.start();
  BufferedReader stdInput0 = new BufferedReader(new
      InputStreamReader(processo0.getInputStream()));
  while ((s = stdInput0.readLine()) != null) {
   System.out.println(s);
  }
  processo0.waitFor();
  System.out.println("///////it finished////////");

  //second process
  System.out.println("///////running UP2CO///////");
  ProcessBuilder p1 = new ProcessBuilder("java", "-jar", "UP2CO.jar");
  p1.redirectErrorStream(true);
  Process processo1 = p1.start();
```

```
    BufferedReader stdInput1 = new BufferedReader(new
        InputStreamReader(processo1.getInputStream()));
    while ((s = stdInput1.readLine()) != null) {
     System.out.println(s);
    }
    processo1.waitFor();
    System.out.println("///////it finished////////");

    //third process
    System.out.println("///////running UP2OB///////");
    ProcessBuilder p2 = new ProcessBuilder("java", "-jar", "UP2OB.jar");
    p2.redirectErrorStream(true);
    Process processo2 = p2.start();
    BufferedReader stdInput2 = new BufferedReader(new
        InputStreamReader(processo2.getInputStream()));
    while ((s = stdInput2.readLine()) != null) {
     System.out.println(s);
    }
    processo2.waitFor();
    System.out.println("///////it finished///////");

    File controller = new File("./output_files/runtime_controller.py");
            File observer = new File ("./output_files/runtime_observer.py");
      String controller_as_string = FileUtils.readFileToString(controller);
      String observer_as_string = FileUtils.readFileToString(observer);
      MissionSender sender = new MissionSender();
    String indirizzo = "192.168.1.140";
    String porta = "13000";
    sender.send(controller_as_string, indirizzo, porta);
    sender.send(observer_as_string, indirizzo, porta);

    System.out.println("finito!");
  }
}
```

Listing B.1: Main in "4-main_pc_side"

It is printed in the prompt window that the aim is the complete translation from the DSL to python, BUT all the files and all the packages are to be placed correctly.
It is created a string type variable and it is printed in the prompt that it will

start "the first parser and Uppaal".So from the PuRSUE-ML, it will generate the control strategy. It is defined "runt", and "p0"; with p0 three parameters are given. In this way, it is passed the command "java -jar DLS2.jar" that put in execution the code contained in the folder **1-parser DSL2UP and UPPAAL integration**. It makes it run invoking the function `start` and then saves its input. It is started a "while" cycle in which it prints the input on the prompt, line by line. It waits for a while and prints on the prompt it has finished the first translation. In this way, the user knows that everything is working.
As for the other translations, the input and the results are taken and saved in the correct folders, in such way once in Complete Package there will not be reference's problems.
The second parser is then begun and the procedure is close to the previous one. It is printed "running UP2CO" and created a second variable "p1" in which are passed similar parameters to execute the code contained into 2-parser2-UP2CO. It is invoked the start function, the result is saved into "processo1" and it is printed, line by line, the input. It waits a while again and then it prints on the prompt a message of the end of the translation.
In the next block of instructions are invoked the third and last parser, `UP2OB`, where the `Observer.py` is created. It prints on the prompt the parser is started. It is created "p2", similar to the previous variables, to which are passed similar parameters in order to execute the code contained in the folder **3-parser UP2OB**.
So, it is invoked the start function, saving the result into "processo2", the input is printed line by line. It waits a while and then prints on the prompt that it has finished.
It is starting a new phase in where the outputs are managed. Two File type variables are created into which are respectively saved the `Controller.py` and the `Observer.py`. They are read from an on-purpose file, created on the parsing phase. Then are converted into a string type and it is created a MissionSender class variable. Are initialized the variable of address and port, with the values specified in the "Co4Robots" project. The address, the ports, the `Controller.py` and the `Observer.py` are passes as parameters of the function send, invoked two times. It is printed on the prompt that everything is ended, because the parsing phase ends.

1-parser DSL2UP and UPPAAL integration

The code contented in this folder is responsible for the translation from the PuRSUE-ML file into a Timed Game Automata and the synthesis of the complete control strategy. For this reason, this is the folder that presents more substance, as shown in Figure B.2, because besides the parsing folder for the TGA translation here are present also the files related to the VerifyTGA. Despite the folder complexity, only the `Main.java`, contained into

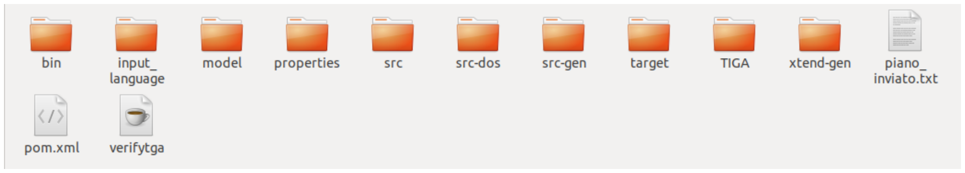


Figure B.2: Folders in "1-parser DSL2UP and UPPAAL integration"

an encapsulate folder structure inside `src`, will be described. The all process can be seen in the Listing B.2.

```
public class Main {

 public static void main(String[] args) throws IOException,
     MissingResourceException {
  System.out.println("This method takes as input a .pur file specifying the
      system and outputs a controller guaranteeing to always satisify the
      goal");
  //
  //define timer variables
  long startTime = 0;
  long endTime =0;
  long elapsedTime = 0;

  //massi inizio timer qui
  startTime = System.currentTimeMillis();
  // inizializzo parser
  Injector iniettore = new PursueStandaloneSetup().
      createInjectorAndDoEMFRegistration();

  // if file was specified use that one, otherwise use standard one
 String file_location;
  //if (args.length==0)
   file_location = "./input_language/default_input.pur";
  //else
  // file_location = "./input_language/" + args[0];
  // check che funga, va la
  System.out.println("considering source file:" + file_location);
  // tirar fuori la risosorsa
  // esco il resource set
  XtextResourceSet risorsaSet = iniettore.getInstance(XtextResourceSet.
      class);
  // nel file d'esempio qeuesto c'era, togliendolo funge comunque... lo
      ascio che
  // si sa mai
  risorsaSet.addLoadOption(XtextResource.OPTION_RESOLVE_ALL,
      Boolean.TRUE);
  // esco l'URI
  URI resource_URI = URI.createFileURI(file_location);
  // creo risorsa
  Resource resource = risorsaSet.getResource(resource_URI, true);

  // validation
   IResourceValidator validatore = iniettore.getInstance(
        IResourceValidator.class);
   List<Issue> report =validatore.validate(resource, CheckMode.ALL,
        null );
  if (report.isEmpty()) {
   System.out.println("validation succeded");
  // creo generatore(usando il Delegate che mi facilita la vita)
  GeneratorDelegate delegato = iniettore.getInstance(GeneratorDelegate.
      class);

  // creo FSA
  InMemoryFileSystemAccess fsa = new InMemoryFileSystemAccess();
  // generazione
  delegato.doGenerate(resource, fsa);
  /// manca la creazione vera del file xD
  String[] nomi = new String[2];
  nomi[0] = "UPPAAL_model.q";
  nomi[1] = "UPPAAL_model.xml";
  int i =0;
  if (args.length != 0 && args[0].equals("custom_property")){
   System.out.println("using custom property already positioned by user
        in folder");
   nomi[0] ="system_generated_property.q";
  }
  for (Entry<String, CharSequence> file : fsa.getTextFiles().entrySet()) {
   BufferedWriter writer = new BufferedWriter(new FileWriter("UPPAAL
        /"+nomi[i]));
   writer.write(file.getValue().toString());
   i++;
   writer.close();
  }
  endTime = System.currentTimeMillis();
  elapsedTime = endTime − startTime;
  System.out.println("UPPAAL model crated in "+elapsedTime+"
      milliseconds");

  // esecuzione di verifyTGA
  String s = null;
  String output = null;
  boolean flag = true;
  startTime = System.currentTimeMillis();
  Process p = Runtime.getRuntime().exec("./UPPAAL/verifytga −t0
      UPPAAL/UPPAAL_model.xml");
  BufferedReader stdInput = new BufferedReader(new InputStreamReader
      (p.getInputStream()));
  while (flag && (s = stdInput.readLine()) != null) {
   System.out.println(s);
   output = output + s + "\n";
   if (s.equals("Strategy to win:") || s.equals("Strategy to avoid losing:")) {

    flag = false;
    endTime = System.currentTimeMillis();
    elapsedTime = endTime − startTime;
    System.out.println( "plan created in " + elapsedTime +" milliseconds
         ");
   }
  }
  startTime = System.currentTimeMillis();
  while ((s = stdInput.readLine())!= null) {
   output = output + s + "\n";
  }
  BufferedWriter writer = new BufferedWriter(new FileWriter("
      UPPAAL_plan.txt"));
```

```
    writer.write(output);
    writer.close();
    endTime = System.currentTimeMillis();
    elapsedTime = endTime − startTime;
    System.out.println( "plan printed to file in " + elapsedTime+"
        milliseconds");
    //sender.send(messaggio, indirizzo, porta);
     }
    else {
     System.out.println("input file not conform to grammar rules. Issues
         detected:");
     for(Issue problema: report) {
       System.out.println(problema);
     }
    }
  }
}
```
Listing B.2: Main in "1-parser DSL2UP and UPPAAL integration"

At first place, is specified the package in which the application refers to, after it, there are around twenty lines of import command. The Main class is declared, so the main function begins. As first action, it prints on the prompt a string that informs on what it does. Then the time variables (startTime, endTime and elapseTime) are defined and initialized at zero value. The current time is saved in "startTime". Then is create a variable named "iniettore" of Injector class, calling a function of a class defined in *src-gen/se/cth/pursue*.
Is then defined a string variable called "file_location" inside which is saved the path to find the input file, once inside the CompletePackage. These instructions are surrounded by a commented if structure, this can allow the file to take as input a file with a different name, but the new name has to be specified in the "arg" variable.
In order to be sure that the path is saved correctly, the "file_location" variable is printed on the terminal. Is then defined "risorsaSet" variable, of "XtextResourceSet", in which is saved the result of the iniettore function invoked. The URI of "file_location" is extrapolated and saved, so the input file is saved into "resource", of Resource class.
To validate the process, a validator variable is created, into which is saved the class "IResourceValidator". A "report" is created, where the result of the validation function is saved. It has to check the properties of the input

resource and, if there are problems, saves them into "report". Indeed, a big if begin after it. If no errors are reported it create and solve the Automata, otherwise instructions are given far below. Assuming there are no problems, is printed on the prompt that the validation has been successful. It is created "delegato", whose save an instance of "iniettore" and it is created a variable "fsa" that will be passed as an argument to the delegato function "doGenerate". So, it is declared a variable in which will be saved the FSA. Is then created an array of two string, and allocates inside them the name of the files that will be afterward created. So respectively `UPPAAL_model.q` and `UPPAAL_model.xml`. It is checked a "args[]" variable and if at its first place it has "custum_property", it is notified on the prompt that it is using an already existing custom property, and the name of `UPPAAL_model.q` is changed.
A for cycle begins, for every fsa.getTextFiles().entrySet(), creates a files UPPAAL_model and saves inside them the values converted into a string format. Now in endTime is saved the current time, in this way it can compute the elapsedTime and prints on the prompt that the 'UPPAAL model is created in "elapsedTime" milliseconds'.
A block dedicated to verifyTga begins, some initializations are done and the current time is saved in startTime. VerifyTGA is invoked through the process "p". The input of p is saved in stdInput, then a while cycle begins. Until the flag is true and there are lines in stdInput, it prints on the prompt the line and saves them in "output". This is made to control if the control strategy generated is a winning strategy, or at least a strategy for not losing. That is because the output of verifyTGA could be a winning strategy, a not losing strategy or a winning strategy for the opponent. Every time it checks if in the selected line there is written "Strategy to win:" or "Strategy to avoid losing".
If it finds one of these two strings, it exits from the while cycle and saves the current time in endTime, to compute the elapsedTime. It prints in the prompt that the plan is created in 'elaspsedTime' milliseconds. Once out of the while cycle it saves the current time in startTime and keeps saving line by line stdInput in output, adding a new line each time (\ n).
It is created a variable named "writer", it will create a file named `UPPAAL_plan.txt` and will write inside it the variable output. It means that the plan created by verifyTGA, if it is not an opponent winning strategy, is saved into output and written into the new generated file. So it saves the current time in endTime, computes the elapsedTime and print on the prompt how much milliseconds takes to print the plan.
As last thing, begin the else branch of the big if. If there are errors with the

validation, it is printed on the prompt that the input does not respect the grammar rules followed by all the problems found.

2-parser2-UP2CO

This folder takes as input the plan produced by verifyTGA and generates the corresponding python code. It is easy to see that the Main file is written to work, read and write, once in the `CompletePackage` because all the paths and all the reference's names are written to work in that conditions.

The Main.java is encapsulated into the folder `UP2CO` with all the folder

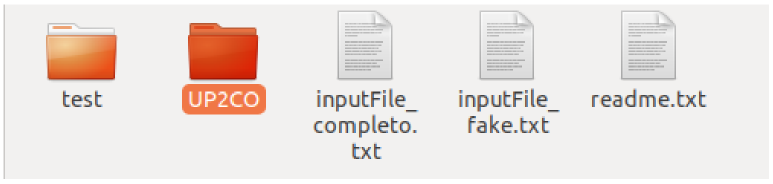


Figure B.3: Folders in "2-parser2-UP2CO"

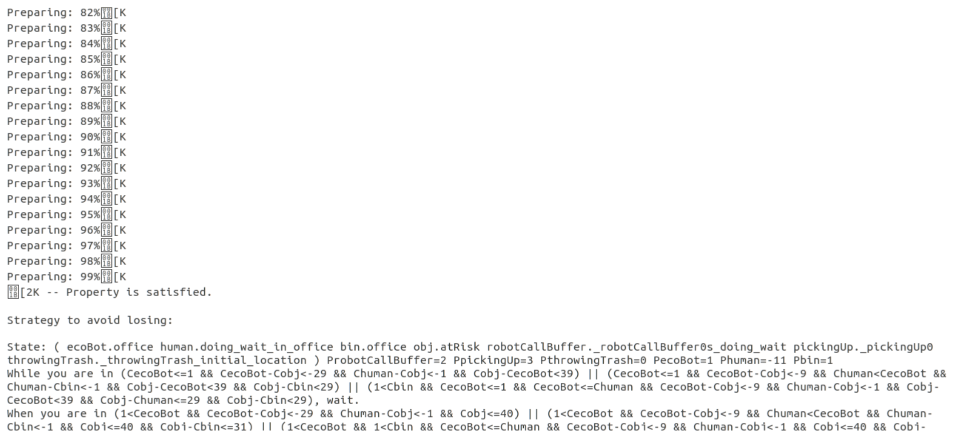
needed to define all the necessary structure for the parsing, as the file with extension `.g4` that contain the specific grammar. What is missed are the Lexer and Parser class that are used to begin the translations. In Figure B.3 are shown the files inside `2-parser2-UP2CO` folder and all processes can be seen in the Listing B.3.

```
public class Main {
    public static void main(String[] args) throws Exception {
     System.out.println("Parser UPPAAL to PYTHON started");
        CharStream in = fromFileName("./UPPAAL_plan.txt");
        int siz = in.size();
        int i = 0;
        boolean flag = true;
        String check = "";
        String checked = "";
        while ( i < siz && flag ){
            Interval intervallo = new Interval(i, i+2);
           // System.out.println(intervallo);
            check = in.getText(intervallo);
           // System.out.println(check);
            if (check.equals(":\n\n")){
                Interval intervallo2 = new Interval(i+3, siz);
```

```
                    checked = in.getText(intervallo2);
                    flag = false;
                    System.out.println("the total plan file is composed of "+
                        siz+" characters, we remove the first "+(i+2) );
                }
                else{
                    i++;
                }
            }
            CharStream cleansedIn = CharStreams.fromString(checked);

            UP2COLexer lexer = new UP2COLexer(cleansedIn);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            UP2COParser parser = new UP2COParser(tokens);
            parser.setBuildParseTree(false);
          Controller controller = parser.controller().contr;
          String output = controller.toString();

         //System.out.println(stampare);
         BufferedWriter writer = new BufferedWriter(new FileWriter("
             output_files/runtime_controller.py"));
   writer.write(output);
   writer.close();
            System.out.println("Parser UPPAAL to PYTHON finished");
      }
}
```

Listing B.3: Main in "2-parser2-UP2CO"

Now just analyze the Main.java: as first, thing it is declared the package in which the application is developed. It imports the libraries, declares the Main class and the main function. It prints on the prompt that the parsing begins, and saves in the variable "in" the control strategy contained in the file `UPPAAL_plan.txt`. There are lines of initialization, after them begins a while cycle that runs the code with a step of two positions, saves the two characters at the same time in "check" and verifies if they are two "\ n". Once found these two characters, it copies the remaining text in a new variable called "intervallo2" and exits from the cycle. It prints on the prompt it has cleaned the file, actually the cycle has cut out all the begging part of the file that is the loading percentage as shown in the Figures B.4 and B.5. It copies the obtained file in a new variable, then creates a UP2CoLexer class



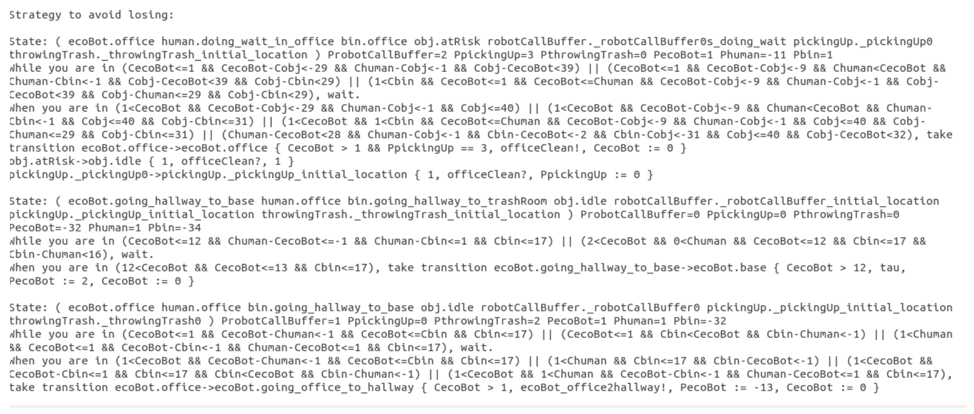
```
Preparing: 82%␛[K
Preparing: 83%␛[K
Preparing: 84%␛[K
Preparing: 85%␛[K
Preparing: 86%␛[K
Preparing: 87%␛[K
Preparing: 88%␛[K
Preparing: 89%␛[K
Preparing: 90%␛[K
Preparing: 91%␛[K
Preparing: 92%␛[K
Preparing: 93%␛[K
Preparing: 94%␛[K
Preparing: 95%␛[K
Preparing: 96%␛[K
Preparing: 97%␛[K
Preparing: 98%␛[K
Preparing: 99%␛[K
␛[2K -- Property is satisfied.

Strategy to avoid losing:

State: ( ecoBot.office human.doing_wait_in_office bin.office obj.atRisk robotCallBuffer._robotCallBuffer0s_doing_wait pickingUp._pickingUp0
throwingTrash._throwingTrash_initial_location ) ProbotCallBuffer=2 PpickingUp=3 PthrowingTrash=0 PecoBot=1 Phuman=-11 Pbin=1
While you are in (CecoBot<=1 && CecoBot-Cobj<-29 && Chuman-Cobj<-1 && Cobj-CecoBot<39) || (CecoBot<=1 && CecoBot-Cobj<-9 && Chuman<CecoBot &&
Chuman-Cbin<-1 && Cobj-CecoBot<39 && Cobj-Cbin<29) || (1<Cbin && CecoBot<=1 && CecoBot<=Chuman && CecoBot-Cobj<-9 && Chuman-Cobj<-1 && Cobj-
CecoBot<39 && Cobj-Chuman<=29 && Cobj-Cbin<29), wait.
When you are in (1<CecoBot && CecoBot-Cobj<-29 && Chuman-Cobj<-1 && Cobj<=40) || (1<CecoBot && CecoBot-Cobj<-9 && Chuman<CecoBot && Chuman-
Cbin<-1 && Cobj<=40 && Cobj-Cbin<=31) || (1<CecoBot && 1<Cbin && CecoBot<=Chuman && CecoBot-Cobj<-9 && Chuman-Cobj<-1 && Cobj<=40 && Cobj-
```

Figure B.4: UPPAAL_plan.txt before

variable. This type of class is not defined in this folder but is located in the "test" folder. That is because is a class that extends an already existing class called Lexer, and there are defined all the headwords that are going to be used for the identification of the text structure. Tokens are created with all the keyword. It is defined a "UP2COParser" class variable, whose code is located in the folder: *target→generated-sources→antlr4*, and here are defined all the structure starting from the token.
It is invoked the function of the Parser object that start the translation. The main subclass is the `controller.class` file. Here are contained the definitions of the structure related to the code for every state and every event. Are defined also the clock structure. For every line of UPPAAL_plan.txt it is written a string with its corresponding python instruction, after the required initialization and constructs.
This string is saved in the variable "output", it is created a text file in a given folder with a given name. The text in output is then saved into the just created file. This is the `Controller.py`.

3-parser UP2OB

This folder has a similar structure and functioning of the previous one. It takes as input the automata model, so the file with extension `.xml` that is the input also for verifyTGA, and translates it into the `Observer.py`. Even in this folder, there are two subfolders. One is for the test and the other one has the actual package for parsing, so the Main.java file encapsulated into the `src` folder, with the other classes for the automata's parsing. The inside



```
Strategy to avoid losing:

State: ( ecoBot.office human.doing_wait_in_office bin.office obj.atRisk robotCallBuffer._robotCallBuffer0s_doing_wait pickingUp._pickingUp0
throwingTrash._throwingTrash_initial_location ) ProbotCallBuffer=2 PpickingUp=3 PthrowingTrash=0 PecoBot=1 Phuman=-11 Pbin=1
While you are in (CecoBot<=1 && CecoBot-Cobj<-29 && Chuman-Cobj<-1 && Cobj-CecoBot<39) || (CecoBot<=1 && CecoBot-Cobj<-9 && Chuman<CecoBot &&
Chuman-Cbin<-1 && Cobj-CecoBot<39 && Cobj-Cbin<29) || (1<Cbin && CecoBot<=1 && CecoBot<=Chuman && CecoBot-Cobj<-9 && Chuman-Cobj<-1 && Cobj-
CecoBot<39 && Cobj-Chuman<=29 && Cobj-Cbin<29), wait.
When you are in (1<CecoBot && CecoBot-Cobj<-29 && Chuman-Cobj<-1 && Cobj<=40) || (1<CecoBot && CecoBot-Cobj<-9 && Chuman<CecoBot && Chuman-
Cbin<-1 && Cobj<=40 && Cobj-Cbin<=31) || (1<CecoBot && 1<Cbin && CecoBot<=Chuman && CecoBot-Cobj<-9 && Chuman-Cobj<-1 && Cobj<=40 && Cobj-
Chuman<=29 && Cobj-Cbin<=31) || (Chuman-CecoBot<28 && Chuman-Cobj<-1 && Cbin-CecoBot<-2 && Cbin-Cobj<-31 && Cobj<=40 && Cobj-CecoBot<32), take
transition ecoBot.office->ecoBot.office { CecoBot > 1 && PpickingUp == 3, officeClean!, CecoBot := 0 }
obj.atRisk->obj.idle { 1, officeClean?, 1 }
pickingUp._pickingUp0->pickingUp._pickingUp_initial_location { 1, officeClean?, PpickingUp := 0 }

State: ( ecoBot.going_hallway_to_base human.office bin.going_hallway_to_trashRoom obj.idle robotCallBuffer._robotCallBuffer_initial_location
pickingUp._pickingUp_initial_location throwingTrash._throwingTrash_initial_location ) ProbotCallBuffer=0 PpickingUp=0 PthrowingTrash=0
PecoBot=-32 Phuman=1 Pbin=-34
While you are in (CecoBot<=12 && Chuman-CecoBot<=-1 && Chuman-Cbin<=1 && Cbin<=17) || (2<CecoBot && 0<Chuman && CecoBot<=12 && Cbin<=17 &&
Cbin-Chuman<16), wait.
When you are in (12<CecoBot && CecoBot<=13 && Cbin<=17), take transition ecoBot.going_hallway_to_base->ecoBot.base { CecoBot > 12, tau,
PecoBot := 2, CecoBot := 0 }

State: ( ecoBot.office human.office bin.going_hallway_to_base obj.idle robotCallBuffer._robotCallBuffer0 pickingUp._pickingUp_initial_location
throwingTrash._throwingTrash0 ) ProbotCallBuffer=1 PpickingUp=0 PthrowingTrash=2 PecoBot=1 Phuman=1 Pbin=-32
While you are in (CecoBot<=1 && CecoBot-Chuman<-1 && CecoBot<=Cbin && Cbin<=17) || (CecoBot<=1 && Cbin<CecoBot && Cbin-Chuman<-1) || (1<Chuman
&& CecoBot<=1 && CecoBot-Cbin<-1 && Chuman-CecoBot<=1 && Cbin<=17), wait.
When you are in (1<CecoBot && CecoBot-Chuman<-1 && CecoBot<=Cbin && Cbin<=17) || (1<Chuman && Cbin<=17 && Cbin-CecoBot<-1) || (1<CecoBot &&
CecoBot-Cbin<=1 && Cbin<=17 && Cbin<CecoBot && Cbin-Chuman<-1) || (1<CecoBot && 1<Chuman && CecoBot-Cbin<-1 && Chuman-CecoBot<=1 && Cbin<=17),
take transition ecoBot.office->ecoBot.going_office_to_hallway { CecoBot > 1, ecoBot_office2hallway!, PecoBot := -13, CecoBot := 0 }
```

Figure B.5: UPPAAL_plan.txt before and after

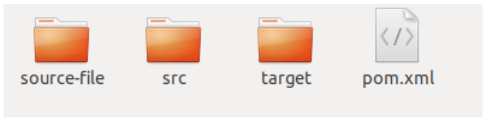


Figure B.6: Folders in "3-parser UP2OB"

structure of `UP2OB` is shown in Figure B.6 and the process is reported in the Listing B.4.

```
import static org.antlr.v4.runtime.CharStreams.fromFileName;
public class Main {
    public static void main(String[] args) throws Exception {
     System.out.println("Parser UPPAAL to OBSERVER started");
        CharStream in = fromFileName("./UPPAAL/UPPAAL_model.
            xml");
        UP2OBLexer lexer = new UP2OBLexer(in);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        UP2OBParser parser = new UP2OBParser(tokens);

        parser.setBuildParseTree(true);
      Model model = parser.model().mod;
      String output = model.toString();

     //System.out.println(stampare);
     BufferedWriter writer = new BufferedWriter(new FileWriter("
         output_files/runtime_observer.py"));
writer.write(output);
writer.close();
        System.out.println("Parser UPPAAL to OBSERVER finished");
    }
}
```
Listing B.4: Main in "3-parser UP2OB"

As first thing is declared the package into which the application is developed, followed by the import's lines. Then the class Main and the function main begin and it prints on the prompt that the parser starts. It copies the automata model into the variable "in", the path is written referring to the position the model has once in `CompletePackage`. It is created a variable of class "UP2OBLexer", an extension of class Lexer, giving it the model as a parameter. In a similar way of the folder **2-parser2-UP2CO**, all the tokens and lexers are defined to create a vocabulary. It is created the variable parser, of a class that extends the Parser class. Both the classes, UP2OBLexer and UP2OBParser, are located in the folder "antlr4" and both are based on the grammar file `UP2OB.g4` placed in the same folder of the Main. These operations are necessary in order to prepare the components for the parsing. It is launched the parser, the string version of the variable "model" is saved in "output". It is created a new file, ("runtime_observer"), where the variable "output" is written. In the end, it is notified the end of the translation via prompt[1].

COMPLETE PACKAGE

This is the last section of the translation phase, inside here there are all the previous operations condensed. To start all the mechanism is enough to open a prompt in the folder and write the command: "java -jar main_pc_side". For more information see Appendix A. Making references to the Figure B.7, it is shown the inside of the folder. Each element has a specific function and has to have that precise name. More specifically:

- **input_language:** is the folder in which is contained the file written in PuRSUE-ML, it is the input of all the translations.
- **output_files:** in this folder are contained the files `runtime_controller.py` and `runtime_observer.py`, the output of the complete translations.

[1]To see how the `Observer.py` is written see the next section (Run-Time: Components)

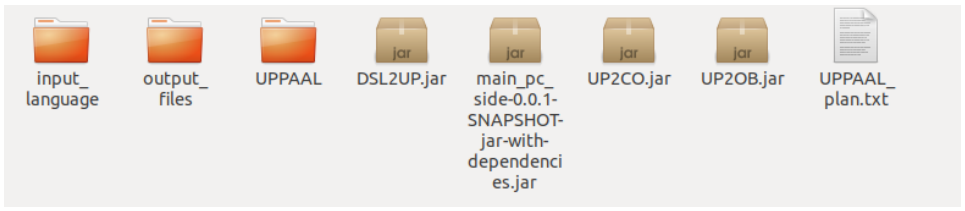


Figure B.7: Folders in "COMPLETE PACKAGE"

- **UPPAAL:** in this folder are contained the executable code of verifyTGA and is where will be saved the automata files (`UPPAAL_model.q` and `UPPAAL_model.xml`).
- **jar(s):** contain the code of the previous folders, they are invoked as described before starting from `main_pc_side`.
- **UPPAAL_plan.txt:** is the control strategy generated by verifyTGA, the input for `2-parser2-UP2CO`.

B.3 Rest2Ros: How it does it

In this case, the code is not all in the same folder, so here is reported also where the files are located.

Mission_sender

It is the first actor that takes place in the communication mechanism. It is located in [39]:

`https://github.com/claudiomenghi/NetworkCommunication/blob/master/src/main/java/se/gu/MissionSender.java`

```
public class MissionSender {

 public void send(String mission, String ip, String port) throws
     ClientProtocolException, IOException {

  CloseableHttpClient httpclient = HttpClients.createDefault();
  HttpPost httppost = new HttpPost("http://" + ip + ":" + port);

```

```
   List<NameValuePair> params = new ArrayList<NameValuePair>(1);
   String sentMission = mission;
   params.add(new BasicNameValuePair("mission", sentMission));

   try {
    httppost.setEntity(new UrlEncodedFormEntity(params, "UTF-8"));
   } catch (UnsupportedEncodingException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
   }
   HttpResponse response;
   response = httpclient.execute(httppost);
   HttpEntity entity = response.getEntity();

   if (entity != null) {
    InputStream instram = entity.getContent();
    instram.close();
   }
  }
 }
```

Listing B.5: Main in "MissionSender"

In Listing B.5 is reported the Mission_sender code. The first thing done is to declare the package where it is developed, then all the import commands are written. The MissionSender class begins with the "send" function that requires three parameters: mission, IP and port. The parameter mission contains the main information, that could be the `Controller.py` or the `Observer.py`, depends on the invocation. The IP address is the IP on whose the mission has to be sent, in this case is 192.168.1.140. The same is for the port, in this case is '13000', as specified on the "Co4Robot" project [31]. It then creates a variable of ClosableHttpClient class, this is an abstract class which implements java.io.Closeable on HttpClient. It saves the complete IP address + port in a HttPost variable to correctly invoke the method 'post', and create a variable named "params" of type <NameValuePair>. It is a special structure in which the data are organized in pairs, a key-value followed by the corresponding value.
The imported parameter mission is saved into a local variable, this variable became the filed value of a new NameValuePair, added with key "mission". Then it tries to send the new pair, keeping the dictionary structure with UTF-8. It is created a variable for saving the response of the execution of

httppost, if the operation has had a good effort or not. If the entity contained into response is not null, the instram is closed.
With this, the input "mission" is forwarded into the given "IPAddress.Port".

Communication_manager

This is the second and most complex agent in this phase. Its code can be found in [40]:

`https://gitc4r.pal-robotics.com:8060/claudiomenghi/communication_manager/blob/master/src/communication_manager.py`

but it has to run on the port where the mission is sent, so it could be necessary to download it. In this specific case, the Communication_manager runs on the port 13000 and writes on the port 13001 and the corresponding node is situated in the `src` folder of the ros workspace. To execute it, the node has to be launch on the prompt, through communication_manager.launch.[2]
In the folder, there are other files containing functions invoked by the main one. Those functions aim to make the mechanism work.
In Listing B.6 is reported the Communication_manager code.

```
class Rest:
 def __init__(self):
  rospy.init_node('communication_manager',anonymous=False,
      disable_signals=True)
  pubsubport = rospy.get_param('~pubsubport')
  self.publisher=Publisher(pubsubport)
  thread = threading.Thread(target=self.publisher.run, args=())
         thread.daemon = True # Daemonize thread
         thread.start()

 def publish(self,msg):
   actions = msg.data
   print ("Sending to the subscribers the set of actions %s" %str(actions)
      )
   self.publisher.send("actions %s" %str(actions))

 def run(self):
  for topic in rospy.get_param('~forwardedtopics').split(","):
```

[2]See Appendix A for more information.

```
        print ("Communication manager will forward messages regarding the
             topic %s" %str(topic))
        subtopic="~" + topic
        rospy.Subscriber(subtopic, String, self.publish)

    port = rospy.get_param('~port')
                        topicType = rospy.get_param('~topicName')
    while not rospy.is_shutdown():
        httpd = HTTPServer(('0.0.0.0', port),Request_Handler)
        print ("Waiting for a new mission on the port %s messages will be
             forwarded on the topic %s" %(port,topicType))
        httpd.serve_forever()

def main():
 print "Running the communication manager"
 rest=Rest()
            rest.run()
if __name__ == "__main__":
 rest=Rest()
 rest.run()
```

Listing B.6: Main in "Communication_manager"

As first thing, there are introductory comments that explain the goal. It is followed by the declaration of the package in which the application is developed and by some import commands. Between all the import commands there is one referred to *ms2_kth.msg*, they could make some problems arise. Lately is defined the Rest class, with all its method: `__init__`, `publish` and `run`. So, to see the main function, it has to go down in the code. After all the methods, the first thing the file does is to print that the Communication_manager is running. It is created an object of class Rest and invoked its method run. The main present a double invocation, that is due to the fact the file can be called also from the prompt.
As first thing, the variable is initialized running the method `__init__`; the ros node *communication_manager* is defined and invoked. It saves the in "pubsubport" the parameter of the number of the port, in this case is 13001, defined in the launch file with the same name, so taken from the outside.
It is defined a redirection of self.publisher, so once this command is invoked it is called a function Publisher, defined in another file, passing the parameter of the port number just obtained. This external file notifies, to its subscribers, that it is going to publish something and then it actually publishes the

message. Port 13001 is where the subscribers are listening.
It is defined a thread for the self.publisher function. That thread is made daemon, in this way the main program does not have to wait it to proceed, and the daemon thread is launched. Since the `__init__` is finished, the run method can start.
It begins a for cycle, for each "forwardedtopics" defined into the launch file, so tiago/mission_location and tiago/mission_action. Tiago is the name of a robot used in the "Co4robot" project, not fundamental for the understanding of the work. It is printed that the Communication_manager will forward the messages it is reading, and it adds a '~' before the previously mentioned topic names. It put itself on listening on the topic, waiting for a message of string type and with self.publish as callback. The callback it is a function that will be executed if something is listened, in this case it is printed that the read message will be forwarded and it is called **publisher.py**.
From the file with extension .launch is also read the port number, in this case 13000, and saved into a local variable and it is taken also the parameter "topicName", local_mission. There is now the command that takes everything active till Ros is on and it is defined a web server address. It actually creates a socket where it listens, and then it writes to **Request Handler** the requests that have to be managed. Then it is printed that it is waiting for a new mission and, in the end, there is a command that avoids exiting from the server.

Publisher.py :
At the beginning there are introductory comments that explain what the Publisher does, it accepts new connection on a specific port, keeps trace of the connected client and send them messages. Then there are few import and the definition of the Publisher class that has three functions: `__init__`, `run` and `send`. The main is declared after the methods, it prints that the Communication_manager is running. It is defined a Publisher object and, after it, it is invoked its method run. Even here there is the double invocation of the main, the second one allows us to call the function from the prompt.
To create the rest object has to be executed the __init__ method, it creates a variable that will take into account all the subscribers in an array of sockets. It is initialized a sock variable as null and then it is taken the port number from the launch file, in this case again 13001.
Once done the initialization, the method run can be executed. As first thing, it creates a socket, and then a symbolic local host address is saved in HOST, in this case as '0.0.0.0' and a bind function is invoked. The socket is putt in

listening, waiting for a connection. It can listen just one at time. It begins an infinite cycle where it prints on video that is waiting for a subscriber and once the connection is accepted the parameters will be saved. It is printed that a subscriber is added and then the server is added to socketset.
It begins the method send, it prints that the messages are been sending and then begin a for cycle. For each socket presents in socketset, it prints "sending the message to the first subscrber" and then the message is sent.

Request_Handler :
Here are described the function of Request_Handler. As always it begins with the introductory comment and the imports, then the Request_Handler class begins. There is not any main, are just defined the methods: _set_header, do_GET, do_HEAD and do_POST.
The first method is _set_header, where are defined self.sen_response(200) and self.send_header. These instructions are necessary for the following functions because they allow the consideration of the response validation, so to make the mechanism work. As the last thing, it invokes self.end_header. The second one is do_GET, it invokes _set_header and then writes a heading with an image in an XML format. The third is do_HEAD that simply invokes _self_header.
Finally do_POST begins, this is the most complex method. It takes the size of the data and then the real data is taken. It invokes a function that returns the data organized in a vocabulary format. Then it saves the topic name from the external, in this case local_mission. It prints that they are receiving, then it is initialized a publisher, through the Ros infrastructure, passing the topic name and the dimension of its queue. The mission is actually published on the topic in the next line. It is print that the mission is sent and is written the command self.send_responce(200). In the end, it is invoked self.end_header.

Pursue_Reader

As Communication_manager also pursue_reader is part of a Ros node that has more ways to be launch. It can be done via pursue_desgintime.sh as describe below or running directly the node or the file with extension launch. Everything that is written here is used to place the files into the correct folder.

```
class Reader:
  def __init__(self):
    #si potrebbe mettere anonymous true se volessimo avere piu nodi alla volta cheascoltano senza rischiare si picchino a vicenda
  rospy.init_node('reader',anonymous=False,disable_signals=True)
  self.counter = 0

  def callback(self, data):
  print("I received data")
  input= data.data
  #very unrobust check if it is controller or observer
  if (input[31]== 'c'):
   f=open("/home/co4robots/Desktop/received_components/runtime_controller.py", "w+")
   f.write(input)
   print("file created in"+f.name)
   f.close()
   f=open("/home/co4robots/turtlebot/src/pursue_controller_node/src/runtime_controller.py", "w+")
   f.write(input)
   print("file created in"+f.name)
   f.close()
  if (input[31]== 'o'):
   f=open("/home/co4robots/Desktop/received_components/observer.py", "w+")
   f.write(input)
   print("file created in"+f.name)
   f.close()
   f=open("/home/co4robots/turtlebot/src/pursue_observer_node/src/observer.py", "w+")
   f.write(input)
   print("file created in"+f.name)
   f.close()

  def run(self):
  print("listening")
  rospy.Subscriber("local_mission", String, self.callback)
  rospy.spin()

def main():
 print "Running the PURSUE reader"
 reader=Reader()
```

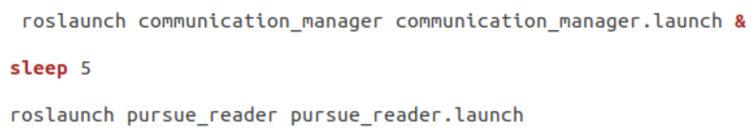
```
reader.run()
if __name__ == "__main__":
  reader=Reader()
  reader.run()
```

Listing B.7: Pursue_Reader

In Listing B.7 is reported the pursue_reader code. As first things, the package is declared then there are the introductory comments that explain what the file does. There are the import lines, keep attention to *ms2_kth.msg* and it starts the Reader class. This class includes three methods: `__init__(self)`, `callback(self,data)` and `run(self)`. The main is declared as the last thing, so really below, and it is written on the prompt "Running the PuRSUE reader". It is created an object of class Reader and then it is invoked its method run. As in many files, there is the double invocation to make the main called also from the prompt. Before the method run can be executed, the `__init__` has to be run. So, it is initialized a ros node called Reader and it is set to zero the inner counter. Once done the initialization, the method run can be executed.
As first thing, it is printed "listening" and puts itself listening on the topic "local_mission" with the callback written above.
After it, there is the command rospy.spin() which prevents that python exits before it is stopped.
Seeing what Callback does: first of all, it prints "I received data" and saved the data in the variable input. It is checked if the 31° character of the input is a 'c' (of controller). If so, it is opened a file called *runtime_controller* in the folder *received_components* and the input is written inside it. And it notified that it has created the file and repeats the writing operation into another folder, in that case the specific folder to make the system work, saving the `Controller.py` in pursue_controller_node. After that, there are similar instruction that checks if the 31 character of the input is an 'o' (of Observer). If so, the `Observer.py` is saved into the "received_components" and pursue_observer_node folders. In order to make this operation working for different applications, the folder's path has to be changed with the path of the case.

Pursue_designtime.sh

It is a really simple file used for launch two nodes together, in order to have fewer operations and fewer instructions. In this specific file, after the workspace is built, it launches the communication_manager. Then it waits



```
 roslaunch communication_manager communication_manager.launch &
sleep 5
roslaunch pursue_reader pursue_reader.launch
```

Figure B.8: Pursue_designtime.sh

five seconds and then launches pursue_reader. This is helpful for consecutive use of these nodes [3].

B.4 Run-Time: Components

Here are presented the Run-Time component individually. With the only exception of the Interface, all the components are contained into a Ros node. So more files make the node able to run, for simplicity here are reported only the main files.

Controller.py

This is for sure the bigger file of the Run-Time structure. The reason fof that has to be found on how the `Controller.py` is conceived. Because to select the correct command for the robot the controller has to check which is the current system state. For this reason, it presents a long list of if command each one corresponds to a possible system state. The Listing B.8 reports the first functions of a `Controller.py`.

```
from threading import Event
class runtime_controller:

 def if_start(self, event_string):
  if (event_string.data == "_start_"):
   self.startFlag = True
  print("controller 1 start")
  return

 def update_state(self, state_string):
```

[3]for more information see Appendix A.

```
  if hasattr(state_string, 'data'):
   all_updates=state_string.data.split("\n\n")
  else:
   all_updates=state_string.split("\n\n")
  state_updates =all_updates[0].split("\n")
  for st in state_updates:
   state_and_value = st.split("=")
   if (state_and_value[0]=="reachObj"):
    self.reachObj=state_and_value[1]
   if (state_and_value[0]=="bot1"):
    self.bot1=state_and_value[1]
   if (state_and_value[0]=="Prule1"):
    self.Prule1=state_and_value[1]
   if (state_and_value[0]=="bot2"):
    self.bot2=state_and_value[1]
   if (state_and_value[0]=="rule1"):
    self.rule1=state_and_value[1]
   if (state_and_value[0]=="Pbot2"):
    self.Pbot2=state_and_value[1]
   if (state_and_value[0]=="Pbot1"):
    self.Pbot1=state_and_value[1]
  clock_updates =all_updates[1].split("\n")
  for cl in clock_updates:
   clock_and_value = cl.split("=")
   if (clock_and_value[0]=="Cbot2"):
    self.Cbot2=float(clock_and_value[1])
   if (clock_and_value[0]=="Cbot1"):
    self.Cbot1=float(clock_and_value[1])
   if (clock_and_value[0]=="Creach"):
    self.Creach=float(clock_and_value[1])
  self.event_flag.set()
  self.print_state()

 def __init__(self):
  rospy.init_node('runtime_controller_node')

  self.TIMEUNIT = 0.1
  self.startFlag= False
  self.event_flag = Event()
  self.event_flag.clear()
  self.exegg = exeggutor.Exeggutor(self.TIMEUNIT)
  self.Cbot2= 0
  self.Cbot1= 0
  self.Creach= 0
  self.reachObj= ""
  self.bot1= ""
  self.Prule1= ""
  self.bot2= ""
  self.rule1= ""
  self.Pbot2= ""
  self.Pbot1= ""
  rospy.Subscriber("pursue/system_state", String, self.update_state)
  rospy.Subscriber("pursue/events", String, self.if_start)
  sleep(1)
  print("waiting for start signal")
  print("controller 1 inizializzato")
  while (not self.startFlag):
   sleep(0.2)

 def print_state(self):
  print("the system state for controller 1 is:")
  print("reachObj is ",self.reachObj)
  print("bot1 is ",self.bot1)
  print("Prule1 is ",self.Prule1)
  print("bot2 is ",self.bot2)
  print("rule1 is ",self.rule1)
  print("Pbot2 is ",self.Pbot2)
  print("Pbot1 is ",self.Pbot1)
  print("Cbot2 is ",self.Cbot2)
  print("Cbot1 is ",self.Cbot1)
  print("Creach is ",self.Creach)
```

Listing B.8: Controller.py

After the import command lines, the Controller class is declared. This class present five functions: _init_, if_start, update_state, print_state, and run. The order of the definition of the function is different from the order in which they are invoked. To maintain simplicity in the exposure, the function will be presented in the same order on which they are written.
The first function is `if_start` and is a callback of an instruction in _init_. If the read data is the string `_start_` the startFlag variable is set to true

and it is notified on the prompt that the controller has started. The second function is **update_state** and it is a bit longer than the previous one. This function is again a callback of instruction in _init_ and its goal is to update the system state once receives the message from the Observer_node.
The data can contain information on the state and the clocks. These two types of information are on the same messages but split by a \n \n character. So, the first thing that the function does is to check the format of the information and split it into the two types.
In first place, takes into consideration only the information on the state, these kinds of information are separated by a single \n character. The single information is added in an array. In order to update all the states, a big for cycle begins. For every information is checked which state is referred to and then its value is updated.
A similar approach is used for updates the clocks. The information about them are saved into an array, split on the \n character. A for cycle begins, for every information contained in clock_updates is checked the referred clock and update its value.
Two methods are then called, one for change the value of the flag and the other one to call the function print_state, which will be described below. In the end, a string that notified the happened updating is written on the prompt.
The third function is **_init_** and has to initialize the controller. The node is declared as all the internal variables. Two times is invoked a method that puts the controller as a subscriber but on two different topics. In this way, if on pursue/system_state is publish something the function update_state is executed. Similarly, if something is written on pursue/events the function if_start will be executed. On the prompt is written that the controller is initialized and that is waiting for the start signal. The fourth function is **print_state** that simply prints on the prompt the value of all the variables, relative to the state and the clocks.

```
def run(self):
  while(not rospy.is_shutdown()):
   sleep(self.TIMEUNIT)
  #state header
   if ( self.bot2== "b" and self.rule1== "rule1_initial_location" and
        self.bot1== "c" and self.Pbot1== "3" and self.reachObj== "
        unlocked" and self.Pbot2== "2" and self.Prule1== "0" ):
    temps=set()
    temp0= optimal_wait([ 1 + self.TIMEUNIT−self.Cbot1, 0 + self.
        TIMEUNIT−self.Cbot2 ] , [ ] , [])
    if (temp0 >= 0):
     temps.add(temp0)
    if(temps):
     wait = min(temps)
     self.event_flag.clear()
     self.event_flag.wait(wait)
     self.exegg.ping_observer()
     if (self.bot2== "b" and self.rule1== "rule1_initial_location" and
         self.bot1== "c" and self.Pbot1== "3" and self.reachObj== "
         unlocked" and self.Pbot2== "2" and self.Prule1== "0" ):
      if ( (1<self.Cbot1 and 0<self.Cbot2 ) ):
       #takes in agent ID, synchronizing action (or tau), and the states or
            ogirin and target of transition
       self.exegg.exeggute("bot1", "bot1_c2a!", "c", "going_c_to_a")
    else:
     self.event_flag.clear()
     self.event_flag.wait()

...

def main():
 controllore = runtime_controller()
 controllore.run()
if __name__ == "__main__":
 controllore = runtime_controller()
```

Listing B.9: Controller.py

The fifth function is the **run**, as reported in Listing B.9, this is the major one. Here is present a long list of if function, one for each possible system state, with associated the correct command. All the if structures are contained into a while cycle. So while the Ros is active the system state a constantly checked. For simplicity only one if is reported, but it does not lose generality. If the condition on the if is satisfied it means the system state is been recognized. In this case, a set variable named "temp" is declared. In another variable is saved the returned value of the function optimal_wait. The function `optimal_wait` is written in file `pursue_library`.
If the returned value is greater than zero, it is added to the temp set. In this way even if there is more than one condition on the time value, they are all taken into account. So, if the temps set is not empty, its minimal value is

researched and waited a respective time unit.
After have waited, it is asked to the Observer_node to republish the system state and, with another if, it is checked that the system is not changed. In this case, is sent to the executor the action that has to be done. The sent information is organized with its parameters: agent, trigger, origin and target. On the contrary, if the system state is changed the research of the corresponding if resumes.
The research resumes also when the command is sent to the executor.
The sixth and last function is **main**, but it is not part of the Controller class. Actually here is defined an object of class Controller and its method run is called. As in almost all other files, it is present the double invocation. This is made in order to be able to invoke the controller also via prompt.

Executor.py

This is a big player into the Controller_node, it is in charged to distinguish the action that has to be done and selects the correct topic into which forward the action. In the beginning, there are the introductory comments that explain the file functions. The import command lines follow and finally is declared the Exeggutor class. The executor is composed of four functions: _init_, ping_observer, start_observer and exeggute, as reported in Listing B.10

```
class Exeggutor:
  def __init__(self, timeunit):
    #dictionary here brutally defined for simplicity, in the future it should
        implemented so it is read form file
    self.location_dictionary = {
    "a" : [3.0, 1.54, 0.0, 0.0, 0.0, 0.67 ,0.73],
    "b" : [-0.57, 0.62, 0.0, 0.0, 0.0, 0.71 , 0.70] ,
    "c" : [0.47, 2.89, 0.0, 0.0, 0.0, 0.76, 0.65],
    "d" : [-3.19, 3.58, 0.0, 0.0, 0.0, -0.04, 1.0]
    }
    self.move = Move_command_sender()
    self.act = Action_sender()
    self.transition = Transition_sender()
    self.TIMEUNIT = timeunit

  #this function sends an "impossible" trigger, this will result in no
      transition taken but clocks updated and publishing of state
```

```
def ping_observer(self):
  self.transition.send_message("_ping_")
  print("I pinged the observer")
  sleep(self.TIMEUNIT)

def start_observer(self):
  self.transition.send_message("_start_")
  print("I started the observer")
  sleep(self.TIMEUNIT)

def exeggute (self, agent, trigger, origin, target):
  #first I check wether I'm triggering a movement
  trg = target.split("_")
  org = origin.split("_")

  if (trg[0] == "going"):
    #I send the coordinates to the bot andthe transition trigger to the
        enviroment
    self.move.send_message(self.location_dictionary[trg[-1]])
    if(trigger[-1]=='!' or trigger[-1]=='?'):
      trigger_cleaned = trigger[0:-1]
    else:
      trigger_cleaned = trigger
    self.transition.send_message(trigger_cleaned)

    #print to screen
    print("I send the trigger"+ trigger_cleaned+ "so I'll go to
        coordinates")
    temp = self.location_dictionary[trg[-1]]
    for p in temp : print p

  #then i check for action with duration
  elif (trg[0] == "doing"):
    if(trigger[-1]=='!' or trigger[-1]=='?'):
      trigger_cleaned = trigger[0:-1]
    else:
      trigger_cleaned = trigger
    self.act.send_message(trigger_cleaned)
    self.transition.send_message(trigger_cleaned)
    print("I do" + trigger_cleaned)
```

```

        #then I check if I finished a movement
        elif(org[0] == "going"):
          #I generate the trigger since as of right now the model doesn't have
              them for finishing movments
          if (trigger == "tau"):
            trigger_cleaned = agent+"_"+origin+"2"+target
          else:
            if(trigger[-1]=='!' or trigger[-1]=='?'):
              trigger_cleaned = trigger[0:-1]
            else:
              trigger_cleaned = trigger
          self.transition.send_message(trigger_cleaned)

        #then I check if I finished an action
        elif(org[0] == "doing"):
          if(trigger[-1]=='!' or trigger[-1]=='?'):
            trigger_cleaned = trigger[0:-1]
          else:
            trigger_cleaned = trigger
          self.transition.send_message(trigger_cleaned)
          print ("agent has finished action, trigger is "+ trigger_cleaned)

        #finally, last remaining option is that the action was istanteneus
        else:
          if(trigger[-1]=='!' or trigger[-1]=='?'):
            trigger_cleaned = trigger[0:-1]
          else:
            trigger_cleaned = trigger
          self.transition.send_message(trigger_cleaned)
          print("agent has done instantaneuos action" + trigger_cleaned)

        sleep(self.TIMEUNIT)
```

Listing B.10: Executor.py

The first function is _init_ and it is executed immediately. A dictionary containing all the P.O.I coordinate is defined. The coordinates are organized into an array of seven positions. The first three values are associated with the space coordinates, while the last four are associated with the unitary quaternion for the orientation.

Then are three command lines, the instructions to call the respective method, contained into the other Controller_node files. They are precisely Move_command_sender(), Action_sender() and Transition_sender(). At last, the time-unit value is saved into the respective local variable. The second function is ping_observer, short but important. Is the first one to be called by the `controller.py`, a "ping" message is delivered to the Observer due to this function. As consequences of that, the clocks will be updated and the Observer_node will republish the system_state. All is done invoking the method "self.transition.send_message()" with "_ping_" as argument. In order to notify the action, on the prompt is written: "I pinged the observer" and is waited a time unit.
Start_observer is another short but important function, invoking again "self.transition.send_message()" but with the argument "_start_" gives the start impulse to the observer. The action done is again notified on the prompt and it waits a time unit.
The last and major function is executor, which checks what type of command the `Controller.py` has sent. In trg is saved the first part of the input parameter target, so it could be "going" or "doing". In org is instead saved the first part of the input parameter origin, it could be "going" or one of the POI.
Based on the values of these two variables, are identified five different cases each one with different activities that have to be done.
Case one: the string saved into trg is "going". It means the command is "send the robot to this position", so the coordinates of the given position are recalled from the _init_ dictionary. Through Move_command_sender.send_message() are forwarded the command to the robot. Before notified the action also at the observer the trigger has to be cleaned. Trigger is a string that summarises the command selected, but can present a character "!" or "?" at its end. Trigger is used to notify to the observer what the robot will do, in order to have the system state always update and. To be recognized, the special character has to e removed and the valid trigger value is saved into trigger_cleaned. Through Transition_sender.send_message(), trigger_cleaned is published on pursue/events. A message with the trigger is printed also on the prompt, followed by all coordinates of the POI.
Case two: the string saved into trg is "doing", so the command is an action with duration. As first thing, the trigger is cleaned. The command has to be forwarded on pursue/action and pursue/event so are respective call the send_message() of act and transition. In this way, the trigger value is sent both to the robot and the observer. At last, the trigger is also printed on the prompt.
Case three: the string saved into org is "going", so a movement is finished.

In this case, trigger is not always given. If trigger exists is just cleaned as in the previous cases, if not it is generated. In order to know if the trigger has to be generated, is checked if its value is "tau". If so trigger_cleaned is composed of the formula: agent_origin2target.
Trigger_cleaned is notified to the observer in the same way: Transition_sender.send_message().
Case four: the string saved into org is "doing", so the action is finished. The trigger is cleaned, as always, and forwarded to the Observer_node. The end of the action is notified also on the prompt.
Case five: if it is not one of the previous chases, has to be an instantaneous action.
The trigger is cleaned, as always, and forwarded to the Observer_node. The end of the action is notified also on the prompt.
In the end, it waits a time unit and the execute function is finished.

Transition_sender.py

After the introductory comments, are present the import instructions. The Transition_sender class begins. It presents two methods: _init_ and send_message. In __init__ it is defined a function that publishes a message on the topic pursue_events with a queue of ten elements. In send_message is called the function defined above passing the message that will be forwarded.

Pursue_library.py

As the name suggests, this file is used as a library for three functions: listen, optimal_wait and listen_and_wait, as reported in Listing B.11. This function is important for the controller because returns how much time the command has to wait before being sent.

```
def listen(TIMEUNIT, heard, queue):
    while (True) :
      time.sleep(TIMEUNIT) #It seems necessary to put this in order to
          make the thing work
      #print("mi chiedo se il Main mi ha scritto <3")
      if not queue.empty(1):
        item = queue.get(True, 1)#at every cycle, it controls wether the
            main thread asked to quit
        if item == "quit":
          break
        with heard: #only enters this part if the method is unlocked
            if (False):#read something from topic, da implementare
                heard.notify()
    return 0

def optimal_wait(clock_higher, clock_lower, clock_const):
    wiggle = 0.5
    print("I enter optimal wait, wiggle is "+str(wiggle))

    for number in clock_const:
        if number < (0-wiggle):
 print("constant costraints dicono no causa di " + str(number))
            return -1.0

    highest_ch = 0.0 #smallest wait before a rising condition becoems true
          (if all true, thus all negative, you wait zero)
    for number in clock_higher:
        if (number > highest_ch):
            highest_ch = number

    if clock_lower:
        lowest_cf = clock_lower[0] #smallest wait before a falling condition
            becomes false
        for number in clock_lower:
            if number > (0+wiggle):
                print("clock lower costraints dicono no, a causa di " + str(
                    number))
                return -1.0 #if a clock_ lower condition is positive, the overall
                    condition shall never be true until some clock are reset
            else:
                if number > lowest_cf: #if it's closer to zero
                    lowest_cf = number

        if (highest_ch-wiggle) > (- lowest_cf):#remembering lowest_ cf is
            negative
            print("final check costraints dicono no, a causa di " + str(
                highest_ch- lowest_cf))
            return -1.0 #a "C <= int " condition will become false before all
                "int < C" condition can become true
    print("tutto ok, ritorno"+ str(highest_ch))
    return highest_ch #return the time needed before all conditions
        become true

def listen_and_wait(TIMEUNIT, wait):
    q = Queue.Queue() # declare a queue, to communicate between
        threads
    change = threading.Condition() #define a condition
    listener = threading.Thread(target = listen, args = (TIMEUNIT,
        change, q,) ) #define a thread, with target function listen
    change.acquire() #acquire the lock for the condition
    startTime = time.time()
    listener.start() #start thread
    print("I+ll wait", wait)
    change.wait(wait) #the method autohmatically releasos the lock
    q.put("quit") #wether I quit because of timeout or condition, I ask the
        thead to quit
    listener.join() #wait for thethread to quit
    endTime = time.time()
    #status.update (con molti threadbisognera stare attenti che non succeda
        nulla in contemporanea e abbia priorita, se dev dialgoare con altri
        thread problemi)

    elapsedTime = endTime - startTime
    return elapsedTime
```

Listing B.11: Pursue_library

The first function is `listen`, begins with an always true cycle. After has waited for one TIMEUNIT, checks if the queue is not empty. If there is at last one element it saves it into a local variable and removes it from the queue. If the get item is the string "quit" the cycle is broken. In the other case invokes the method heard.notify and returns. Optimal_wait is the longer function of the library; it takes as parameters clock_higher, clock_lower e clock_const from the controller. They are values associated with the inner clocks of the automata, and it is fundamental to respect the condition on them. It is defined a variable, wiggle, that has the duty to take into account the uncertainties on the time measure.
It is printed on the terminal that the function is starting and the current value of the wiggle. A for cycle begins, pass every value of clock_const and it is checked if is lower than the wiggle. If so it is notified the error, the time is to short and return -1.0. If no problem occurs, there is another "for" cycle

that research the highest clock value.
In the following instruction block, the clock_lowers, if there are, are checked. If one of them is lower than the wiggle value, an error message is printed and the function returns -1.0. Otherwise, the value of the lowest clock_lower is saved into lowest_cf.
At the end of the last control, if the highest clock_higher is greater than the lowest clock_lower, it prints an error message and returns -1.0 because there will never be a waiting value that will satisfy the constraints.
If the execution survives, it is print an OK message and the highest_ch is returned. This is the time the system has to wait before all the conditions become true.
The third and last function is `listen_and_wait`. A FIFO queue is declared thought of the queue.queue() command, this queue allows the threads to communicate with each other. A thread.Condition() type variable is declared; this type of return a condition variable that manages the thread execution. Defines a thread, named "listener", as a thread with listener as the target function.
Acquired the lock, saves the current time as StartTime and launch the thread. It is written on the prompt the waited time and automatically releases the lock after the wait time.
"Quit" is added to the thread queue, in this way the thread will terminate itself for inactivity. All the other threads are blocked, so the listener can expire its queue.
The current time is saved in endTime, the elapsedTime is computed and returned.

Move_command_sender.py

After the introductory comments are present the import instructions. The Move_command_sender class begins, it presents two methods:_init_ and send_message, as reported in the Listing B.12.

```
class Move_command_sender:
  def __init__(self):
    self.pub = rospy.Publisher('move_base_simple/goal', PoseStamped,
        queue_size = 10)
    sleep(1)

  def send_message(self, coordinates):
    goal = PoseStamped()
```

```
        goal.header.frame_id = "map"
        goal.header.stamp = rospy.Time.now()
        goal.pose.position.x = coordinates[0]
        goal.pose.position.y = coordinates[1]
        goal.pose.position.z = coordinates[2] #always
        goal.pose.orientation.x = coordinates[3]
        goal.pose.orientation.y = coordinates[4]
        goal.pose.orientation.z = coordinates[5]
        goal.pose.orientation.w = coordinates[6]
        self.pub.publish(goal)
```

Listing B.12: Move_command_sender

In __init__ it is defined a function that publishes a message on the topic: move_base_simple/goal, with a queue of ten elements.
In send_message it is defined a variable of type PoseStamped, this type of object is composed of nine fields. Two are of type header and are used by Ros to interpret the others. The others are used for saving the position that has to be reached. This is because the goal point is identified with the three spatial coordinates and the unit quaternion.
Once the variable is initialized, is invoked the method self.pub.publish. In this way, the position is published on the topic move_base_simple/goal.

Action_sender.py

After the introductory comments, are present the import instructions. The Action_sender class begins, it presents two methods: _init_ and send_message.
In __init__ it is defined a function that publishes a message on the topic pursue_action with a queue of ten elements.
In send_message is called the function defined above passing the message that will be forwarded.

Observer.py

This agent is in charged to keep the system state update. The Observer_node is composed of more files, but the main code is Observer.py, the one created in the Design phase. Here is reported only the Observer.py.
In the first line, the package is defined, and the import instructions follow. The file composition strongly depends on the scenario, because a class is defined for each automaton of the model. So, each class is composed of four functions: reset_clock, trigger, updateP and _init_, as reported in Listing

B.13. The last class is Observer(), always present, it has different functions: callback, _init_ and run, as reported on Listing B.14.

```
class Classstates():
 def reset_clock(self):
  self.need_reset = True

 def trigger(self, trigger):
  if (trigger in self.machine.get_triggers(self.state)):
   if (trigger=="isPlastic"):
    self.isPlastic()
    self.SpaperOrPlastic="0"
   if (trigger=="isPaper"):
    self.isPaper()
    self.SpaperOrPlastic="1"

 def updateP(self, newP):
  self.P = newP

 def printStates(self):
  stati = ""
  stati = stati +"\nSpaperOrPlastic="+str(self.SpaperOrPlastic)
  return stati

 def __init__(self):
  self.name="states"
  self.need_reset = False
  self.clock=0.0
  self.P =0
  self.SpaperOrPlastic=1
  states =['base']
  transitions = [
   {'trigger' : 'isPlastic' , 'source' : 'base' , 'dest' : 'base' },
   {'trigger' : 'isPaper' , 'source' : 'base' , 'dest' : 'base' }
  ]
  self.machine= Machine(model = self, states=states, transitions=
      transitions, initial='base',auto_transitions=False)
```

Listing B.13: Observer.py

Without losing generality, only one class related to the state automata is presented.

Its first function is **reset_clock** that simply set True the variable need_reset. The second function is **trigger**, which checks if the trigger is present in the trigger array of the local automata. If the trigger is recognized, the respective transition takes place and the automata evolve.
The next and third function is updateP, which simply saves the new value of P.
The last function is **_init_**, that initialized the inner variables. There are three specific variables, two arrays that save all the states and all the transitions of the local automata, and machine, that through a java class, actually creates the automata.
The part related to the "local" automata is ended, now is reported the function present in Observer class.

```
class Observer():
 def callback(self, data):
  trigger = data.data
  for automaton in self.automa:
   automaton.trigger(trigger)
  if trigger == "_start_":
   self.startTime=time.time()
  self.stopTime = time.time()
  elapsedTime =self.stopTime - self.startTime
  self.startTime = time.time()
  for automaton in self.automa:
   if automaton.need_reset:
    automaton.clock = 0
    automaton.need_reset = False
   else:
    automaton.clock += elapsedTime
  state = ""
  clocks = ""
  for automaton in self.automa:
   state +="\n"+ automaton.name + "=" + automaton.state + "\n"+"
        P"+automaton.name+"="+str(automaton.P)
   clocks += "\n"+'C'+automaton.name +"=" +str(automaton.clock)
  state = state + self.automa[0].printStates()
  print ("the following will be sent to state topic:\n"+state+"\n\n"+
      clocks)
  self.pub.publish(state+"\n\n"+clocks)

 def __init__(self):
  self.automa = []
  self.automa.append(Classstates())
  self.automa.append(ClasspickingUp())
  self.automa.append(ClassthrowingPaper())
  self.automa.append(ClassthrowingPlastic())
  self.automa.append(ClassmakingTrash())
  self.automa.append(ClassecoBot())
  self.automa.append(ClassplasticBin())
  self.automa.append(Classhuman())
  self.automa.append(ClasspaperBin())
  self.automa.append(Classobj())
  self.startTime = time.time()
  self.stopTime = self.startTime
  rospy.init_node('pursue_observer')
  self.pub = rospy.Publisher('pursue/system_state', String, queue_size=1)

 def run(self):
  rospy.Subscriber("pursue/events", String, self.callback)
  rospy.spin()

def main():
 observer = Observer()
 observer.run()
if __name__ == "__main__":
 observer = Observer()
 observer.run()
```

Listing B.14: Observer.py

This class is always present in every Observer.py and invokes all the other classes.
Its first function is **callback**, which is a callback of one of the instructions in the run function. The read data is saved and forwarded to each local automata, calling the function trigger.
If the data is the string `_start_`, is saved the current time in startTime. Out of the if command, the current time is saved into stopTime, and elapsedTime is computed. The current time is saved again into startTime and all clacks are updated. For every automaton, if the clock needs a reset is reset, otherwise it is incremented by the value of elapsedTime.
Two variables are declared, state and clock. Into these variables are saved

every state and every clock of every automaton separated by a $\backslash n$ character. On the prompt is write that these two variables will be published and they are actually published on pursue/system_state topic separated by a double $\backslash n$.
Now begin the function **_init_**. In this function, all the local automata classes are declared. The current time is saved into startTime and stopTime is initialized as startTime.
The pursue_node is declared and started. This node is put as a publisher on the topic pursue/system_state with a unitary queue.
The last function is **run** that simply put the node as a subscriber on pursue/events, and has callback as a callback. The method spin maintains the structure active while Ros works.
In the end, the main is defined. It is declared an object of class Observer and is invoked its function run. As in many cases before, is present the double declaration.

Interface.py

This agent is used to communicate with the outside. With it the uncontrollable event can be notified to the Observer, the start signal can be given and simulation can be done. The main parts are reported in the Listing B.15.

```
class Stater:
 def callback_ev(self, data):
  if (data.data == "_start_"):
   self.startTime = time()
  if (data.data != "_ping_"):
   print("\n\non topic events:" + data.data + " at time "+str(time()−
        self.startTime))

 def callback_act(self, data):
  print("\n\non topic actions:" + data.data+ " at time "+str(time()−
       self.startTime) )

 def callback_state(self, data):
  print("\n\non topic sysyem_state:" + data.data + " at time "+str(
       time()−self.startTime))

 def __init__(self):
  rospy.init_node('stater')
  self.pub = rospy.Publisher("pursue/events", String, queue_size = 1)
  self.mod= raw_input("select mode: ('f' = all communication, 'e'= only
       events ): ")
  self.sim= raw_input("would you like to start the automatic enemy
      strace?(traceID/n): ")
  self.startTime = time()
  print("initializing done")
 def run(self):
  rospy.Subscriber("pursue/events", String, self.callback_ev)
  if (self.mod == "f"):
   rospy.Subscriber("pursue/actions", String, self.callback_act)
  if (self.mod == "f"):
   rospy.Subscriber("pursue/system_state", String, self.callback_state)
  events = ["_ping_",
   "callBot",
   "wait",
    "waitDONE",
    "isPlastic",
    "isPaper"
   ]
  bin_mov =[ "bin_trashRoom2hallway",
    "bin_going_trashRoom_to_hallway2hallway",
    "bin_hallway2base",
    "bin_going_hallway_to_base2base"
   ]
  while(not rospy.is_shutdown()):
   #traccia EB1 -experimental
   raw_input("press any key to start")
   self.pub.publish("_start_")
   sleep(0.5)
   while (self.sim == "a" and not rospy.is_shutdown() ):
    sleep(1.1)
    self.pub.publish(events[1])
    sleep(1.1)
    self.pub.publish(events[2])
    sleep(61.1)
    self.pub.publish(events[3])
...
act_code = int(raw_input("select action: "))
   if (act_code < 0):
      i = 0
      while (i < len(events)):
       print("action #"+str(i)+" is "+events[i])
       i= i+1
     elif (act_code < len(events)):
     self.pub.publish(events[act_code])
     else:
      print("input incorrect")
    return

def main():
 stater = Stater()
 stater.run()
if __name__ == "__main__":
 stater = Stater()
 stater.run()
```

Listing B.15: PuRSUE_UI

After the import command, the Stater class is defined. This class present five function: callback_ev, callback_act, callback_state, _init_ and run. These functions will be presented in the same order on which they are written.
The first function met is **callback_ev**. As suggested from the name, it is a callback function called by one of the run instruction.
The read data value is checked, if it is the string "_start_" the current time is saved in startTime. Otherwise, if the data value is a string different from "_ping_", a message that says that the data is written on the topic event is written on the prompt.
The second function is **callback_act**. Even in this case is a callback function called by one of the run instruction. It simply writes on the prompt that the data were written on the topic action.
The third function is **callback_state**, completely analogous to callback_act, only for the system_state topic.
The next, and fourth, function is **_init_**. This is an internal function that has the duty to initialized all the inner variables. The first thing done is to start the node Stater, then this node becomes a publisher on the topic pursue/events. It reads, from the prompt, if a full communication is desired, or it is preferred to just considering only the events. It is also asked if it has to start an automatic event sequence (of the enemy), and in the case which sequence. The time is saved into startTime and is notified on the prompt that the stater is ready.

The last function is **run**, this is the major one. As first thing, the node is put as a subscriber on pursue/event, with callback_ev as callback. If the communication is full, the node subscribes also on pursue/action and pursue/system_state topics with callback_act and callback_state respectively, as callbacks.
Are then defined two arrays, one containing all the possible events and the other one for all the possible movements.
A while cycle begins, until the Ros mechanism is on, asks on the prompt to "press any key to start". After has received confirmation, the start signal is published on the topic `pursue/events`. Follows all the command for automatically simulate the enemy strategy, take for example the "a", case used on the ecoBot scenario [1][4] without losing generality. In this case, the enemy strategy is composed of three instructions. At the beginning is published the first event, "callBot" and wait 1.1 sec. Then is published the second event, "wait", and waits for 61.1 seconds. At last, it publishes the third event, "waitDONE" and resumes the while cycle.
If no one of the automatic sequences is selected, on the prompt is asked which action has to be notified on the Observer. The act_code is saved and checks if that action exists on the previously defined event array. Once is found, the event is published on the pursue/events topic. Otherwise is printed on the prompt "incorrect input".
In this way, the Stater class end. To complete the file there is the main. It declares an object of class Stater and invokes its run function. As in many other cases, the main has the double declaration.

Pursue_runtime.sh

Pursue_runtime.sh is a really simple file used for launch two nodes together, to have fewer operations and fewer instructions. This specific file, used once the workspace is built, launches the observer_node. Then it waits two seconds and then launches the controller_node. This is useful for a consecutive use of these node [5].

B.5 Other files

Corollary material is present in this framework version, these files are not essential for the correct operation described before. They are supportive

[4]For more information see the previous work
[5]for more information see Appendix A

files for development or testing, here are reported the main ones that can be useful.

Folder 5-pursue_receiver-main_robot_side

If it is present, it is located into the Design Folder, contains the Ros files used to run the reader_node.

Folder 6-controller generation components

If it is present, it is located in the Design Folder and contains the files contained in the `controller_node/src` folder. Can be useful develop the new version of these files inside this folder.

Eclipse code

Eclipse-code is a folder structure used in the Design-Time Development phase. Inside this folder, there are other eight levels of encapsulated folders that end with other three empty folders.

```
EclipseCode/cominicator/beacon/target/classes/org/xtext/thesis/
matt4->generator/scoping/validation.
```

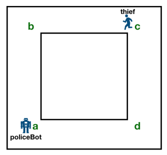
Matteo_maps

It follows the path "turtleBot environment/PuRSUE/Matteo_maps" and contains a folder with pictures and their description of an office's maps. These maps were used during the test on the real robot.

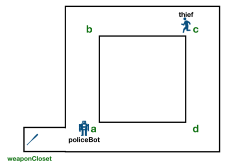
The problem examples

Here are reported the most significant scenarios taken as reference for the development of the PuRSUE Framework. These scenarios, also with small variation, are modeled with PuRSUE-ML in [1, 2]

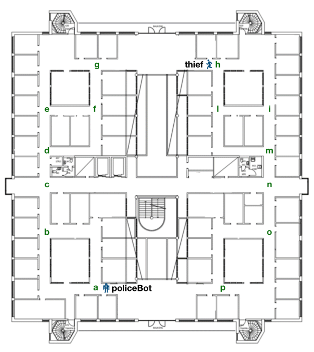
Catch the thief



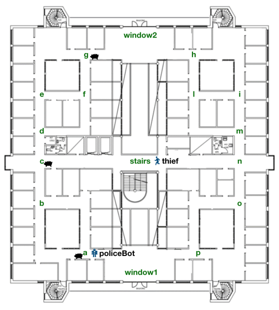
(a) Catch the Thief simple map



(b) Catch the Thief simple map with variation



(c) Catch the Thief complex map



(d) Catch the Thief complex map, reverse role

Figure B.9: Catch the Thief Maps

For an exemplification, it is taken into account Figure B.9a. In this situation, there is a policeBot that has to catch a thief, an uncontrollable agent free to move, and that has to escape. To accomplish the task the policeBot has to immobilized the thief with a tool it is provided with. This operation will be modeled as a collaborative event in PuRSUE-ML.
The policeBot has to know the environment in which it moves, it could be assumed that it has a map given and/or a perception and mapping algorithms. In this case the environment it is shown in Figure B.9a as a room, where are

identified four Point Of Interest (POI) labeled as a,b,c,d. In PuRSUE-ML the environment is described by the POI and the distances between them, through this information the framework is able to synthesize the control strategy which brings the policeBot to catch the thief. Other variables have to be taken into account, as the speed of the two agents or unfeasible robot movement and the PuRSUE-ML can model all of them. A lot of variation of this simple situation can be described and solved, e.g. consider Figure B.9b where the policeBot has to take the baton from another room or Figure B.9c where the environment is far more complex[6]. It is also possible to consider the Bot as the thief and humans as policemen as shown in Figure B.9d.

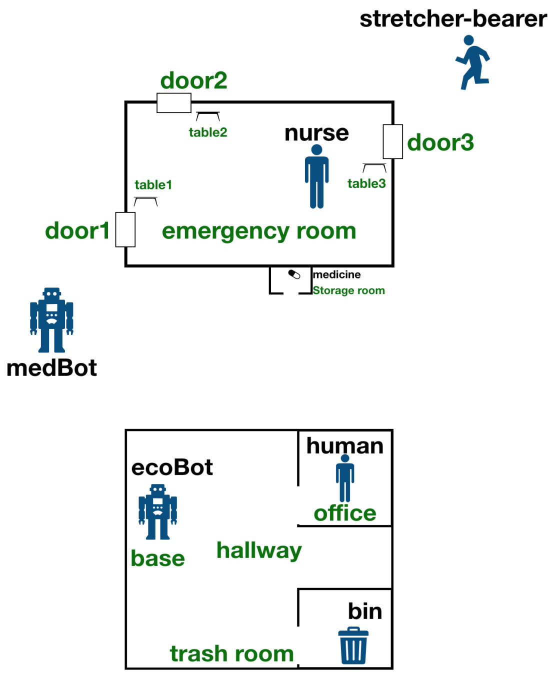


Figure B.10: Drug Delivery and EcoBot Examples

Drug Delivery

This scenario has more agent, the medBot, a controllable mobile robot that has the task to bring the medicine from the storage room to one of the three

[6]The map shown is the third floor of the Jupiter building of the Goteborg University

tables inside the emergency room. The tables are located near one of the three doors (correspondingly), the medBot has to cross the emergency room's door, but it can do that only if the door is open. If the door is close the medBot has to try if another door is open or has to ask the nurse to open the door and wait till the action is completed. The nurse can stay only inside the emergency room she is able to open and close the doors, she is model as a controllable agent because she works with the medBot and tries to help it. An uncontrollable agent is a stretcher-bearer, he is free to move all over the scenario, he can open and close the doors and the medBot has to avoid to collide him and also to be in his way.

Ecobot

This scenario was used in [1] for an experiment with a real mobile robot (TurtleBot) in the Jupiter building of the Goteborg University. In this scenario the protagonist is the ecoBot, a controllable mobile robot located in a room called Base. The uncontrollable agent is a human, who stay in the office, and can call the robot when he has some trash to throw away. Once the ecoBot receives the command pass through the hallway, it goes into the office and collects the trash. Once it has done it, it returns in the hallway to reach the trash room and put the trash into the bin. Also for this scenario some variants are present.

Bibliography

[1] Matteo Soldo, *"USER-FRIENDLY CONTROLLER SYNTHESIS FOR MULTI-AGENT ROBOTIC APPLICATIONS IN PRESENCE OF AGENTS WITH UNKNOWN BEHAVIOR"*, master thesis, Politecnico di Milano, A.A.2017-2018.

[2] Bersani, M.M., Soldo, M., Menghi, C. et al. PuRSUE -from specification of robotic environments to synthesis of controllers. Form Asp Comp (2020). https://doi.org/10.1007/s00165-020-00509-0

[3] Olaf Stursberg, Member, IEEE, and Christian Hillmann: "Decentralized Optimal Control of Distributed Interdependent Automata With Priority Structure", IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING, VOL. 14, NO. 2, APRIL 2017.

[4] H. Wong-Toi, G.Hoffmann; Stanford University, Stanford, CA 94305: "The Control of Dense Real-Time Discrete Event Systems"; Proceeding of the 30th Conference on Decision and Control, Brighton, England December 1991.

[5] Quottrup, M. M., Bak, T., & Izadi-Zamanabadi, R. (2004). Multi-Robot Motion Planning: A Timed Automata Approach. <Forlag uden navn>.

[6] Guangyuan Li, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Danny Bøgsted Poulsen: "Practical Controller Synthesis for MTL0,∞". Conference'17, Washington, DC, USA

[7] Giovanni Bacci, Patricia Bouyer, Uli Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, et al.. Optimal and Robust Controller Synthesis: Using Energy Timed Automata with Uncertainty. FM 2018 - International Symposium on Formal Methods, Jul 2018, Oxford, United Kingdom. pp.203-221, 10.1007/978-3-319-95582-7_12 . hal-01889222.

[8] Eugene Asarin, Oded Maler, Amir Pnueli, Joseph Sifakis: "CONTROLLER SYNTHESIS FOR TIMED AUTOMATA",Copyright © IFAC System Structure and Control, antes,France, 1998.

[9] M. Usman Iftikhar, Jonas Lundberg, Danny Weyns: "A Model Interpreter for Timed Automata".

[10] Daniel Opp, Mirko Caspar, Wolfram Hardt: "Code Generation for Timed Automata System Specifications ConsideringTarget Platform Resource-Restrictions", Research Paper, Information Technology Journal Vol. 7, No. 14, July - December 2011, pages 38-45.

[11] Howard Wong-Toi: "The Synthesis of Controllers for Linear Hybrid Automata", Proceedings of the 36th Conference on Decision & Control San Diego, California USA December 1997.

[12] Jonathan S. Ostroff, W. Murray Wonham : "A Framework for Real-Time Discrete Event Control", IEEE Transactions on automatic control, vol. 35, no. 4,April 1990, pages 386-397.

[13] Davide Marocco, Stefano Nolfi: *Emergenza della comunicazione in robot mobili.*

[14] Lynne E. Parker,University of Tennessee, Knoxville: *Distributed Intelligence: Overview of the Field and its Application in Multi-Robot Systems.*

[15] Félix Ingrand LAAS-CNRS, RoboSoft: Software Engineering for Robotics Royal Academy of Engineering, London, UK 13-14, November 2019: *Verification of Autonomous Robots A "humble" Roboticist Bottom Up Approach.*

[16] Tamio Arai, Enrico Pagello, Lynne E. Parker: *Editorial: Advances in Multi-Robot Systems*, IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 18, NO. 5, OCTOBER 2002: 655-661 1.

[17] Ceballos, A.; De Silva, Lavindra; Herrb, M.; Ingrand, F.; Mallet,; Medina, A; Prieto, M. :*GenoM as a Robotics Framework for Planetary Rover Surface Operations.*

[18] Anthony Mallet, Cedric Pasteur, Matthieu Herrb, Severin Lemaignan, Felix Ingrand : *GenoM3: Building middleware-independent robotic components.*

[19] https://maven.apache.org/

[20] https://maven.apache.org/what-is-maven.html

[21] https://www.antlr.org/index.html

[22] https://github.com/antlr/antlr4/blob/master/doc/index.md

[23] http://people.cs.aau.dk/ adavid/tiga/

[24] https://www.it.uu.se/research/group/darts/uppaal/download/ registration.php?id=5&subid=2&id=5&subid=2

[25] http://www.uppaal.org/

[26] https://www.eclipse.org/Xtext/#intro-quotes

[27] https://www.eclipse.org/Xtext/documentation/index.html

[28] https://www.ros.org/

[29] http://wiki.ros.org/

[30] https://github.com/claudiomenghi/NetworkCommunication/ tree/master/src/main/java/se/gu

[31] http://www.co4robots.eu

[32] https://www.docker.com/

[33] https://hub.docker.com/_/ros

[34] http://wiki.ros.org/docker/Tutorials/Docker

[35] https://www.redhat.com/it/topics/containers/what-is-docker

[36] https://www.turtlebot.com/

[37] http://wiki.ros.org/Robots/TurtleBot

[38] https://pypi.org/project/transitions-gui/

[39] https://github.com/claudiomenghi/NetworkCommunication/blob/master/src/main/java/se/gu/MissionSender.java

[40] https://gitc4r.pal-robotics.com:8060/claudiomenghi/communication_manager/blob/master/src/communication_manager.py