POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica

Programming Abstractions for Nano-drones Teams

Dipartimento di Elettronica Informazione e Bioingegneria

Relatore: Prof. Luca Mottola
Correlatore: Mikhail Afanasov

Tesi di Laurea di:
Manuel Belgioioso, matricola 804149
Alberto Cardellini, matricola 818246

Anno Accademico 2014-2015

Alle nostre famiglie.

Abstract

Autonomous drones are performing a revolution in the field of mobile sensing: various type of drones are used to perform a great number of applications, since they can carry rich sensor payloads, such as cameras and microphones. Often there is a simple abstraction which allows drones navigation: they can be controlled through smartphones and tablets interfaces or by setting waypoints. Drones can greatly extend the capabilities of traditional sensing systems while simultaneously reducing cost. They can monitor a farmer's crops, manage parking spaces, or monitor underwater telecommunication systems more practically and/or more cheaply than stationary sensors. The great innovation brought by drones is that they offer direct control on where to sample the environment; this was impossible with previous mobile sensing systems that could only passively sample the environment, laying on the mobility of smartphones or vehicles. So, the drones can greatly extend the field of mobile sensing, allowing the programmers to create a great number of applications, unthinkable and impossible to develop with the already existing technologies.

Sommario

I droni autonomi stanno compiendo una vera e propria rivoluzione nel campo del rilevamento mobile: vari tipi di droni sono utilizzati per eseguire un gran numero di applicazioni, in quanto possono trasportare una gran quantità di dati raccolti grazie a sensori quali fotocamere e microfoni. Spesso vi è una semplice astrazione che permette ai droni di navigare nell'ambiente: possono essere controllati tramite interfacce grafiche per smartphone e tablet o impostando dei waypoint. I droni possono notevolmente estendere le capacità dei sistemi di rilevamento tradizionali riducendo al contempo i costi. Si possono monitorare le colture di un agricoltore, gestire i parcheggi, o monitorare i sistemi di telecomunicazione sottomarini in maniera più pratica e/o più a buon mercato rispetto ai sensori fissi. La grande innovazione portata dai droni è che offrono un controllo diretto sui punti da monitorare nell'ambiente; questo era impossibile con i precedenti sistemi di rilevamento mobili che potevano solo passivamente rilevare l'ambiente, basandosi sulla mobilità di smartphone o veicoli. Così, i droni possono estendere notevolmente il campo del rilevamento mobile, permettendo ai programmatori di creare un gran numero di applicazioni, impensabili precedentemente e impossibili da sviluppare con le tecnologie già esistenti.

Acknowledgements

It is a pleasure to thank those who made this thesis possible with advices, critics and observations.
We would like to thank our supervisor Prof. Luca Mottola and our mentor Mikhail Afanasov: without their help and support, this thesis would not have been possible.

We would like to thank Prof. Thiemo Voigt, who kindly let us developing part of this work at SICS Swedish ICT and all the colleagues who have greeted and helped us during the three months in Sweden, in particular Simon Duquennoy, Liam McNamara, Joel Höglund and Niklas Wirström.

We owe our deepest gratitude to our families and friends for the continuous support during these years at university.

Finally, we would like to thank one with the other for having lived together this experience.

Contents

Abstract **V**

Sommario **VII**

Acknowledgements **IX**

1 Introduction **1**
1.1 General context 1
1.2 Brief description of the work 2
1.3 Outline 2

2 State of the art **5**
2.1 Drone-level approach 7
2.2 Swarm-level approach 8
2.2.1 Robot Operating System 8
2.2.2 Karma 9
2.2.3 Proto 11
2.3 Team-level approach 13
2.4 Conclusion 15

3 Programming indoor applications **17**
3.1 Indoor context and requirements 17
3.1.1 Indoor localization 17
3.1.2 Drones and Objects size limitation 20
3.2 Team level approach for the nano-drones coordination 21

4 Programming with Pluto **23**
4.1 System's architecture . . . 23
4.1.1 Pluto Graphical Editor . . . 24
4.1.2 Pluto Main Application . . . 26
4.2 Dataflow model . . . 29
4.2.1 Description of the model . . . 29
4.2.2 Model representation . . . 30
4.3 Aiming to the final model . . . 37
4.3.1 Solution without Trip entity . . . 37
4.3.2 Solution without the DroneAllocator . . . 38

5 Implementation **41**
5.1 Graphical editor . . . 42
5.2 Code generation . . . 42
5.2.1 From graph to code . . . 43
5.3 Object-oriented approach . . . 44
5.4 Runtime Management . . . 45
5.5 User interface . . . 46
5.6 The crazyflie nano-quadcopter . . . 47

6 Evaluation **51**
6.1 Applicability of the Pluto framework . . . 51
6.1.1 Alfalfa Crop Monitoring and Pollination . . . 52
6.1.2 Aerial mapping of archaeological sites . . . 61
6.1.3 PM10 . . . 66
6.1.4 PURSUE . . . 70
6.1.5 Object-finder (OF) . . . 72
6.1.6 Warehouse item-finder (WIF) . . . 73
6.1.7 Drugs distribution (DD) . . . 74
6.2 Usability of the model . . . 75
6.2.1 Proposed exercises . . . 75
6.2.2 Evaluation metrics . . . 80
6.2.3 Baseline . . . 81
6.2.4 User Survey . . . 81
6.2.5 Results . . . 82
6.3 Performance evaluation . . . 84

6.3.1 Software Metrics 85
6.3.2 Hardware Consumption 85
6.3.3 Results 86

7 Conclusions and future works **91**
7.1 Conclusions 91
7.2 Pluto limits and future works 93
7.2.1 Implementation limits 93
7.2.2 Technological limits 93

Bibliography **95**

List of Figures

2.1 The iRobot Create 7
2.2 ROS communication layer functioning 9
2.3 The basic schema of Karma 10
2.4 The amorphous medium abstraction 11
2.5 Proto: problem decomposition 12
2.6 Voltron APIs 14

4.1 A complete life cycle of the application 24
4.2 Pluto Graphical Editor interface 25
4.3 Mission Page interface 26
4.4 Trips Page interface 27
4.5 Monitor Page interface 28
4.6 An example Pluto application 30
4.7 Relationship among model entities 32
4.8 The MissionModifier block 36
4.9 Solution without the Trip concept 38
4.10 Solution with the TimerMonitor 39
4.11 Solution with the DelayMonitor 39
4.12 Solution without the MissionModifier block 40

5.1 Pluto architecture representation 41
5.2 Observer design pattern 44
5.3 The MVC pattern 45
5.4 Example of thread concurrency 46
5.5 The Crazyflie Nano-Quadcopter 48

6.1 Pluto graph for the Alfalfa Crop Monitoring and Pollination application 53

6.2 The circular area to monitor 57
6.3 Sequence diagram of a starting mission 58
6.4 Sequence diagram of the mission execution flow 59
6.5 Sequence diagram of the trip execution flow 60
6.6 Sequence diagram of the mission ending flow 60
6.7 Pluto graph for the Aerial Mapping of archaeological sites 62
6.8 The archaeological site on the map 65
6.9 Sequence diagram of the mission ending flow, without repetition . 66
6.10 Pluto graph for the PM10 application 67
6.11 The archaeological site on the map 69
6.12 The Pluto graph of the OF,WIF and DD applications 71
6.13 The basic functioning of the Object-finder application 72
6.14 The basic functioning of the Warehouse item-finder application . . 73
6.15 The basic functioning of the Drugs distribution application 74
6.16 Solution of the first step 76
6.17 Solution of the second step with Priority Manager 77
6.18 Solution of the second step with Clock block 78
6.19 Solution of the third step 79
6.20 VisualVM interface 84
6.21 Very Complex Diagram Example 86
6.22 Hardware metrics of Graphical Editor 88
6.23 Evaluation results of Main Application with fixed missions and trips 88
6.24 Evaluation results of Main Application with fixed missions and drones 89
6.25 Evaluation results of Main Application with fixed trips and drones 90

Chapter 1

Introduction

1.1 General context

Autonomous drones are performing a revolution in the field of mobile sensing: various type of drones are used to perform a great number of applications, since they can carry rich sensor payloads, such as cameras and microphones. Often there is a simple abstraction which allows drones navigation: they can be controlled through smartphones and tablets interfaces or by setting waypoints. Drones can greatly extend the capabilities of traditional sensing systems while simultaneously reducing cost. They can monitor a farmer's crops, manage parking spaces, or monitor underwater telecommunication systems more practically and/or more cheaply than stationary sensors. The great innovation brought by drones is that they offer direct control on where to sample the environment; this was impossible with previous mobile sensing systems that could only passively sample the environment, laying on the mobility of smartphones or vehicles. Many drones applications have been developed in the recent years, performing a wide range of different functionality, but they are all suitable for outdoor contexts and so they use medium/big sized drones. We aim to create a general model for the interaction of nano-drones, in order to extend the support for programmers who want to create new applications in indoor contexts.

1.2 Brief description of the work

In the beginning we developed three applications case study, shown in section 6.1, suitable for indoor contexts whose sensing tasks are performed by teams of nano-drones. We noticed that these applications had many common features, so we decided to create a general programming model which could be used to build a great variety of applications. Each of these features is represented by a block, each one concerning with a specific task of the work: there is a block which assigns a drone to an object to deliver, one that send the drones to sense the environment etc.

We wanted to facilitate as much as possible the work of the programmer, so we decided to develop a graphical editor, fully described in section 4.1.1. Thanks to this editor the programmer can create an application, by simply graphically connecting the functional blocks. He is not forced to use all the blocks provided by the editor and can connect only the blocks he needs according to the functionality he wants for the specific application.

We created the whole model for our programming abstraction, choosing the Java programming language to represent all the features we needed, and organizing the system with an MVC pattern.

We also developed the final user interface, the Pluto Main Application, shown in section 4.1.2, to make the final user able to fully exploit the applications created with Pluto programming framework. With the Pluto Main Application, the user can assign the sensing tasks to the drones, sending them in the environment.

The Editor, the Main Application and the Java model are strictly connected: the programmer creates the graphical model of the application through the Graphical Editor, then the Editor generates the code according to the Java model and finally the user uses the Main Application to define the sensing tasks for the created application.

1.3 Outline

This document is structured in a way that you can easily follow the proceedings and the reasoning behind our contribution.

In this chapter, we have given the general context and the general goals of the work together with a brief description of our work.

In Chapter 2 there is a description of the actual state of the art in the context of our work. We describe the three main existing approaches for drone programming, the "Swarm-level","Drone-level" and "Team-level" approaches, also proposing existing examples for each one of them.

Chapter 3 is focused on the problems issued by the indoor context and on the requirements deriving from it. The similarities and differences with respect to the outdoor applications are analyzed, in order to derive a new implementation of the system, suitable for the indoor context.

Chapter 4 presents our solution for the research problems described inchapter 3, the Pluto programming framework. In section 4.1 the architecture of the system is analyzed, describing in details the two components of the Pluto framework: the Graphical Editor and the Main Application. In section 4.2 we present our dataflow model: we show the entities of the Java Model and we describe the functionality of each block of the Pluto Graphical Editor. The last section of the chapter, the 4.3, describes all the steps performed to arrive to the final system, showing all the previously implemented solutions which, once refined, brought us to the development of the Pluto programming framework.

Chapter 5 shows how the designed choices have been implemented technically, describing all the software and tools used for the development of Pluto programming framework.

Once the solution has been deeply described at design and implementation level, we will show the evaluation and the results of the work.

Chapter 6 starts with an analysis on the applicability of the Pluto framework: there are descriptions of many already existing applications and of the three case study, and an analysis on whether they can be developed with Pluto. In section 6.2 we propose two exercises to human testers, in order to evaluate the effective usability of Pluto: the first one deals with the Graphical Editor, the second one with the Main Application. We also propose a survey to the users and then present the result in a graphical way. In section 6.3 we measure the software and hardware consumption metrics required by Pluto, presenting the results in a graphical way.

Finally, Chapter 7 draws the conclusion and recaps the results obtained, also showing the possibilities for future works to extend our programming model.

Chapter 2

State of the art

In the last years, the drone technology is expanding rapidly;especially the aerial drones, which are often used as toys controlled by joystick or to make aerial videos through mounted cameras. There exist also aquatic and terrestrial drones, which can be used for many applications. For example, through aquatic drones the submarine network infrastructure[**?**] could be managed more efficiently, and through terrestrial drones some emergency situations, like fires, can be managed without involving human life.

Basically, the mobile sensing through drones represents a technological revolution, opening the way for many applications which could not have been developed with the traditional technologies. Actually, there are a lot of fields where this new technology could be applied, improving performance and reducing costs: for example surveillance applications, instructing drones to fly over an area monitoring people, or an application in a domestic context, for example instructing drones to find a lost object or to perform some kind of actions, like bringing some objects to the final user.

In the following section we present three main approaches for programming drones , providing existing examples for each one of them:

- *Drone-level approach*, described in section 2.1
- *Swarm-level approach*, described in section 2.2
- *Team-level approach*, described in section 2.3

The former is focused on the programming of an every single drone, while the second implies basic rules to execute for the whole swarm of drones. The

third approach is the most modern work. It creates a middle-ground between drone and swarm approaches, by providing a flexibility in expressing sophisticated collaborative tasks without addressing to a single drone.

2.1 Drone-level approach

In the Drone-level approach, the programmer must manage the single drone, taking care of giving a list of instructions that the drone will perform sequentially. This approach may be suitable for applications where there is a single drone performing some actions, like searching for a lost object and bringing it back to the user.But scaling the application to a number of drones makes programmer to deal with concurrency and parallelism. Moreover, battery and crashes/failures should be managed manually for every drone. Finally, timing constraints and a dynamic load balance drastically increase the complexity of the programming. For these reasons drone-level approach is not suitable for a large number of drones.

A concrete example of the application of the Drone-level approach is the so called Robot Create(fig. 2.1), a hobbyist robot manufactured by iRobot[**?**] that was introduced in 2007 and based on their Roomba vacuum cleaning platform. The iRobot Create is explicitly designed for robotics development and improves the experience beyond simply hacking the Roomba. Since the built-in serial port supports the transmission of sensor data and can receive actuation commands, any embedded computer that supports serial communication can be used as the control system. A number of robot interface server / simulators support the iRobot Create. Most notably, the Player Project has long included a device interface for the Roomba, and developed a Create interface in Player 2.1. The Universal Real-time Behavior Interface (URBI) environment also contains a Create interface. This robot is designed for a single execution of a single task without being connected with other robots. Moreover, the design does not imply the collaboration with other robots.

Figure 2.1: The iRobot Create

2.2 Swarm-level approach

The Swarm-level approach[**?**] is more suitable for applications where a number of drones are supposed to perform the same actions. Indeed the programmer can give a set of basic rules that each drone in the swarm can follow. It is important to underline that, in swarm-level approach, there is no possibility to have a shared state between drones; each drone execute the actions specified by the programmer on his own local state. It enables the scaling of the approach, but it's not suitable for applications that require the drones to explicitly coordinate. For example, the swarm-level approach could be applied in an application where many drones take objects at different locations and bring them to the final user, without considering any time or space coordination between them; each drone will simply bring the object back to the user when found. There are several existing applications using the swarm-level approach, but we decided to describe three of them: the Robot Operating System (ROS)[1], which provides a Publish/Subscribe coordination layer for decentralized computations, as shown in section 2.2.1; Karma[2], which lets programmers specify modes of operation for the swarm, such as "Monitor" or "Pollinate"(as shown in section 2.2.2); and Proto[3], which lets programmers specify actions in space and time(as shown in section 2.2.3).

2.2.1 Robot Operating System

ROS[1] is not an operating system in the traditional sense, indeed it provides a layer for communication between many, possibly heterogeneous, operating systems connected in a cluster.

The whole functioning of ROS, shown in fig.2.2, is based on *Nodes*, which are software modules performing computations; the whole system is composed by many nodes exchanging messages, according to the *Publish-Subscribe* model: a node can send messages publishing them on a particular *Topic*, and nodes which are interested in a particular topic simply subscribe to it; publishers and subscribers don't know each others' existence. The publish-subscribe topic based communication model is very flexible, but is not suitable for synchronous exchanges, because of its broadcast functioning; for this reasons ROS provides also *Services*, which are composed by a name and two messages, one for the request and one for the response.

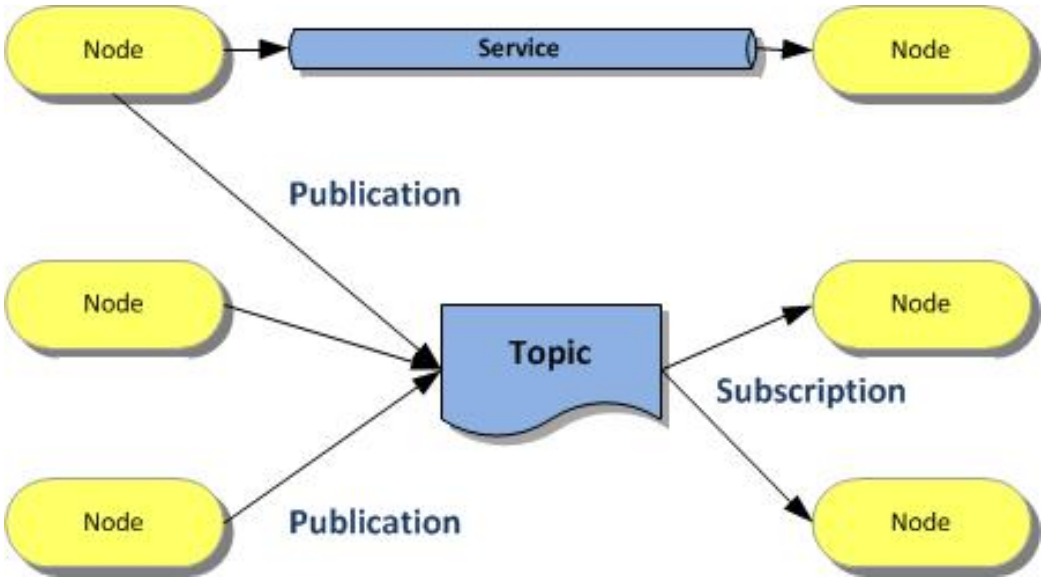

Figure 2.2: ROS communication layer functioning

2.2.2 Karma

Karma[2] is a resource management system for drones swarms based on the so called hive-drone model; the hive-drone model is a feature that moves the coordination complexity of the application to a centralized computer: the hive is the base station where drones can land, if they are not busy, and charge their batteries; the hive also takes care of dispatching the drones in order to perform the actions specified by the programmer to accomplish the swarm objectives; the programmer specifies the desired swarm behaviour through a programming model which allows him not to deal with a coordination between drones.

The Karma[2] runtime at the hive is composed by functional blocks, as shown in fig. 2.3:

- *Controller*: is the overall manager of the runtime and invokes the other modules when needed; when a user submits an application to the Karma system, the hive Controller determines the set of active processes, and invokes the Scheduler to allocate the available drones to them.
- *Scheduler*: is periodically invoked by the Controller to allocate drones to each active process.
- *Dispatcher*: is responsible for tracking the status of the drones; it programs

the drones with the allocated behavior prior to a sortie, tracks the size of the swarm, and notifies the Controller when a drone returns to the hive and is ready for redeployment.

- *Datastore*: when drones return to the hive, they transfer the data they collected to the Datastore.

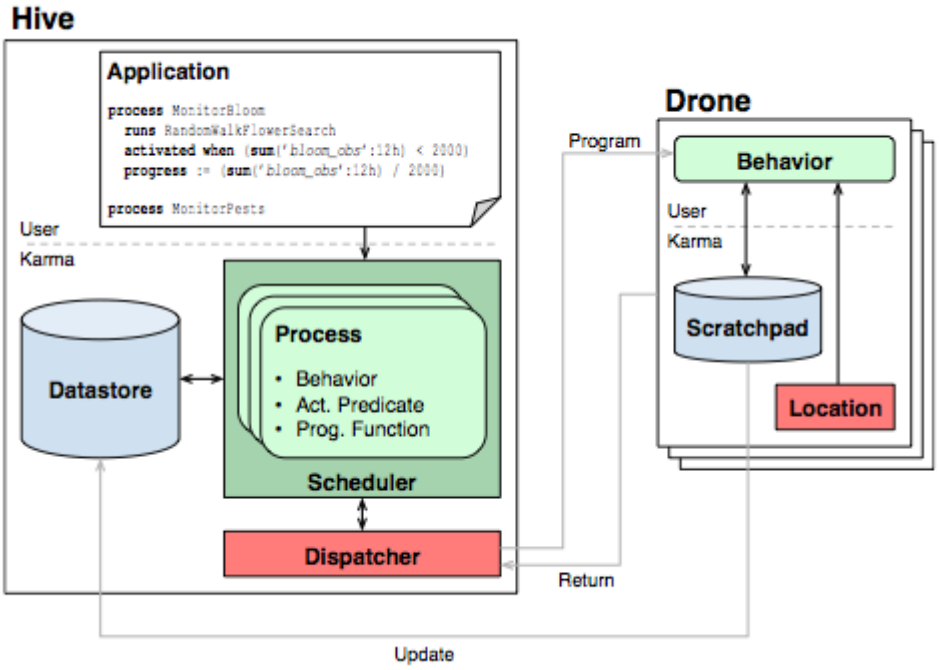

Figure 2.3: The basic schema of Karma

2.2.3 Proto

The amorphous medium abstraction[4] is derived from the observation that in many spatial computing applications, we are not interested in the particular devices that make up our network, but rather in the space through which they are distributed; indeed, for example, the only things that matter for a sensor network are the values that it senses, not the particular devices it's composed of. The amorphous medium[4] takes this concept to the extreme: indeed it is defined as a spatial area with a computational device at every point, as shown in fig.2.4: Information propagates through this medium at a maximum velocity. Each device is associated with a neighborhood of nearby devices, and knows the "state" of every device in its neighborhood, i.e. the most recent information that can have arrived from its neighbors.

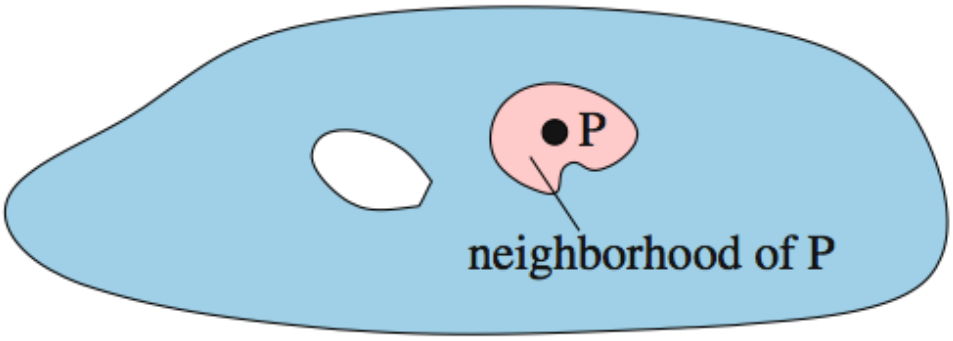

Figure 2.4: The amorphous medium abstraction

The Proto[3] language uses the amorphous medium abstraction[4] to divide the spatial programming problem in three sub-problems, as shown in fig. 2.5:

- global descriptions of programs as functional operations on fields of values
- compilation from global to local execution on an amorphous medium
- discrete approximation of an amorphous medium by a real network

To apply Proto[3] language to mobile devices, such as drones in a swarm, the amorphous medium[4] must be extended with the concept of *density*; indeed for the vast majority of mobile applications, it is important to distribute drones depending

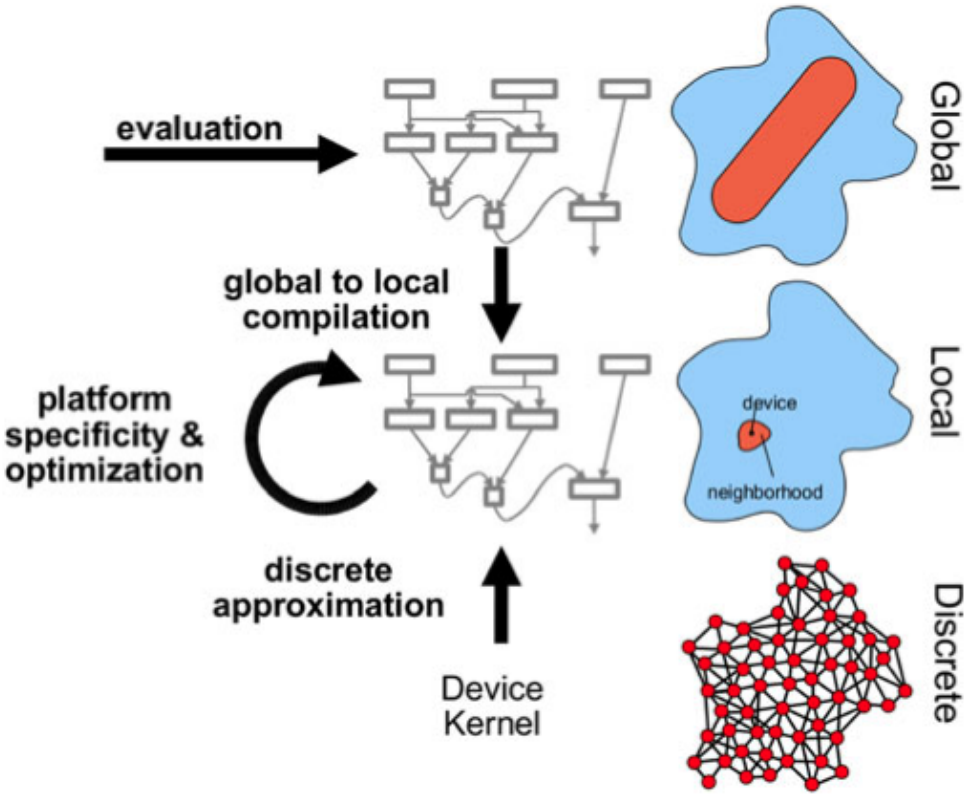

Figure 2.5: Proto: problem decomposition

on what is happening in the environment, for example one may want to send more drones in an area where something is happening; so it must be possible to distribute drones heterogeneously in the space. Adding the concept of *density*, Proto can express a lot of applications using the swarm-level approach. For example, a swarm of lightweight scout robots might search a disaster area and coordinate with a team of more capable rescue robots that can aid victims, or a swarm of aerial vehicles might team with firefighters to survey and manage wildfires and toxic spills, or a group of autonomous underwater vehicles might survey their environment and autonomously task portions of the swarm to concentrate data gathering on particular interesting phenomena.

2.3 Team-level approach

In this section we describe the team-level programming approach[5], which allows the user to express a list of sensing tasks to be performed by the system, without dealing with the management of the single drone and with complex programming tasks such as concurrent programming and parallel execution; the user can also require a layer of coordination, defining constraints in space and time for the tasks' execution, and the system will follow these constraints choosing the actions for each drone at run-time, in order to collaboratively accomplish all the tasks. This run-time drones management makes the whole system scalable, since one can add as many drones as he wants, and also fault tolerant, because it can easily manage crashes or exceptions. So, the main advantage of using the team-level approach is that the user can simply specify a list of tasks to be performed,together with constraints in space and time for the execution, not caring about the dispatching and coordination of the drones; this is also a limitation, because one cannot develop applications which require explicit communication between drones. So, the team-level approach is most suitable for applications involving tasks that could be also performed by a single drone, but require a large number of drones to be completed faster and/or to operate in a big area.

A concrete example of team-level approach application is Voltron[5], a system containing a set of programming constructs to explore the notion of team-level drone programming. Voltron's[5] basic functioning includes:

- the so-called *abstract drone*, which makes the application scalable, allowing to add drones without changing the code
- spatial semantics, which allow the drones to execute parallel tasks at different locations
- the possibility to dinamically re-schedule drones operations in case of crashes or failures
- the possibility to define time constraints for the tasks

Voltron[5] exposes an API, as shown in fig.2.6, that programmers can use to task the drones without individual addressing; since the abstract drone is the only entry point to the system's functionality, an application's code can remain unaltered no matter how many real devices are deployed.

Operation	Inputs	Description
`do`	`action` (singleton) `locations` (set $\neq \emptyset$) `parameters` (set) `handle` (singleton)	Perform an `action` at specific `locations`, customized by `parameters`, linked to `handle`.
`stop`	`handle` (singleton)	Stop the running action linked to `handle`.
`set`	`key` (singleton) `value` (singleton)	Set a $\langle key, value \rangle$ pair in the registry.
`get`	`key` (singleton)	Read the value of *key* from the registry.

Figure 2.6: Voltron APIs

The team-level approach represents a middle-ground between the drone-level and swarm-level approaches, and it also solves many problems. Unlike the drone-level approach, there is no need to address the single drone and, unlike the swarm-level approach, there can be a "global state" and also time and space constraints can be defined; as already said, the team-level approach's main limitation is that there is no possibility to perform tasks which require explicit communication between drones, such as passing an object between them.

2.4 Conclusion

In this chapter we have described the actual state of the art in the field of our work, showing the three main existing approaches for programming a team of drones to perform many actions.

Since the approaches we mentioned in this chapter are designed for outdoor applications, and our work is focused on indoor applications, we need to add a contribution to the actual state of the art. Neither the drone-level nor the swarm-level approaches are suitable, since we don't want to manage the single drone and the parallel execution. On the other hand, the swarm-level approach is feasible for a large number of drones to execute the same actions, but we need to manage a small number of drones (5 or 10) performing different actions, and we also need to address to time and space constraints, which cannot be expressed with this approach. The most suitable approach for our work is the team-level model, but we need to apply some modifications to it, in order to make it suitable for indoor contexts. Indeed even with many problems being solved by Voltron[5] architecture, we found some limitations: for example, the size of drones is essential indoors, but with size decrease, we bump into battery limitations; another problem is that of indoor localization, since GPS cannot be used inside buildings; we discuss this and other problems in details in the next chapter.

Chapter 3

Programming indoor applications

3.1 Indoor context and requirements

The goal of our work is to develop a programming framework to help programmers in creating applications for indoor usage of swarm of drones. For example, drones can find a lost item in a house, or bring objects in hospitals and warehouses. Besides of localization problem, indoor context also leads to the limits of the size of the drone. As a result, programmers constantly confront with a limited battery resource and a small weigh the drone can carry out. These problems, as well as their possible solutions, are described in the following section.

3.1.1 Indoor localization

The main issue that all developers are facing, working on an indoor application for drones, is that they are not able to use the Global Positioning System (GPS); it cannot be used because of walls,roofs or ceilings. For this reason Indoor Positioning System(IPS) is widely applied for indoor localization. In this section we will give an overview of existing IPS methods.

An indoor positioning system is a solution to locate objects or people inside a building using radio waves, magnetic fields, acoustic signals, or other information collected from the sensors of mobile devices. The IPS methods rely on alternative technologies, such as *magnetic positioning* and *dead reckoning*, to actively locate mobile devices and provide ambient location for devices to get sensed.

Today many IPS methods have been developed and they can be divided in two main categories: *Non-radio technologies* and *Wireless technologies*.

Non-radio technologies have been developed for localization without using the existing wireless infrastructures, and they can provide very high accuracy. Nevertheless, they also require expensive installations and costly equipment.

For example, *Magnetic positioning* is based on the iron inside buildings that create local variations in the Earth's magnetic field. Modern smartphones can use their magnetometers to sense these variations in order to map indoor locations.

With *Inertial measurements* pedestrians can carry an inertial measurement unit(IMU) by measuring steps indirectly or in a foot mounted approach, referring to maps or additional sensors to constrain the sensor drift encountered with inertial navigation.

Existing wireless infrastructures can be used for indoor localization; almost every wireless technology is suitable, although they won't be as precise as non-radio technologies. Localization accuracy can be improved at the expense of new wireless infrastructure equipment and installation. WiFi signal strength measurements are extremely noisy, so there is need to find a way to make more accurate systems by using statistics to filter out the inaccurate input data. WiFi Positioning Systems are sometimes used outdoors as a supplement to GPS on mobile devices, where only few reflection phenomena could happen.

WPS is based on measuring the intensity of the received signal(RSS) together with the technique of *fingerprinting*. In computer science, a fingerprinting algorithm is a procedure that maps an arbitrarily large data item to a much shorter bit string, its fingerprint, that uniquely identifies the original data for all practical purposes just as human fingerprints uniquely identify people for practical purposes. The accuracy of WPS improves with the increase of the number of positions entered in the database. WPS is subjected to fluctuations in the signal, that can increase errors and inaccuracies in the path of the user.

Bluetooth cannot provide a precise location, since it's based on the concept

of *proximity*, indeed it is considered an *indoor proximity solution*. However, by linking micro-mapping and indoor mapping to Bluetooth and through the usage of *iBeacons*, real existing solutions have been developed for providing large scale indoor mapping.

Passive radio-frequency identification(RFID) systems are based on the concepts of *location indexing* and *presence reporting* for tagged objects. These systems do not report the signal strengths and the distances between tagged objects, and do not renew the location coordinates of the sensors or the locations of the tags.

According to the *Grid concepts* low-range receivers can be used, and arranged in a grid pattern, for economy, in the space that must be observed. Since they are low-range effective, each tagged receiver can be identified only by some neighbors.

Received signal strength indication (RSSI) is a measurement of the power level received by sensors. Radio waves propagate following the inverse-square law, the distance can be computed considering the signal strengths at the transmitter and receiver.

As already explained, indoor contexts contains a lot of obstacles, like walls, doors, tables etc., and so the accuracy is lowered by absorption and reflection phenomena; some corrective mechanism must be adopted to lower these phenomena.

It is important to remember that, since we work on a higher level of abstraction, we don't have to choose a specific IPS method. Indeed our programming framework simply utilize an API to make the drone move in the indoor environment, but we are not interested in the implementation of the API and on the particular IPS method.

3.1.2 Drones and Objects size limitation

Indoor contexts imply small areas which are usually full of people and obstacles (think of an house context) hence, drones have to be small, in order to avoid crashes with both human and environmental obstacles.

Size limitations result in many problems; the first is battery duration, which can reach a maximum of 10 minutes, having a recharge time of about 20/30 minutes. This problem is partially solved by the fact that the vast majority of applications that can be developed with our final solution, the Pluto programming framework, involve a team of drones. So, if one drone's battery is about to get empty, the drone can return to the base station and recharge while another drone can substitute it. This obviously complicates the system's logic, because it must also take care to check the drones battery and to find a free drone that can eventually substitute the one whose battery is low. It limits the programmer in developing applications which don't require the drones to perform long trips to carry out their actions or to work in parallel.

Another problem arising from size limitations is that the smaller the drone is the less stable he is. Almost every kind of micro-drone has serious stability issues, and a lot of research efforts goes in this direction. This problem is lowered by the developing of programming libraries that could improve stability of the drones at real-time, adjusting a set of parameters while the drone is flying.

Micro-drones are obviously more fragile than the big ones, so a crash with humans or obstacles can definitely destroy the drone or make it seriously damaged. This is the price to be paid for having little drones that can operate in small indoor contexts.

Finally, the use of small drones means that only small objects can be took, so the applications developed with Pluto framework must take this into account. For example, a pair of keys can be brought to a person, not a book nor a pair of shoes. For big objects, a different kind of applications can be designed, such as one that find the object and then notify the user of the position of this object, through a camera or acoustic signaling.

3.2 Team level approach for the nano-drones coordination

Using a Team level approach, that we described in section 2.3, entails some problems. The user can neither address individual drones nor express actions that involve direct interactions between drones, such as those required to pass an object between them. This is the main limitation of the approach, but it does not affect the scenarios we focused in our work, which we show in section 6.1. Another problem of the Team-level approach is that, having a single brain which manages all the application logic and the dispatching of drones, the system has a single point of failure, so, if the central brain breaks then the whole system won't work. This problem can be fixed or at least weakened by applying dependable systems methods, improving reliability of the central brain, reducing is rate of failure etc.

Even though team-level approach has his own limitations, other approaches we discussed in section 2 are less suitable.

Indeed, the Drone-oriented approach's main problem is that the programmer has to manage the single drone's movements and interactions with other drones: he must give a list of instructions that the drone will perform sequentially. In the case of multiple drones,the programmer should deal with difficult programming tasks, like concurrency and parallelism, and it should also manage the drone batteries and their crashes/failures. Adding one or more drones to the system could complicate a lot the programming task. The programmer should also deal with timing constraints and he should dinamically balance the load between drones; so, the drone-level approach is most suitable for applications involving only one drone.

On the other and, the Swarm-level approach is more suitable for applications where there's need of a lot of drones performing the same actions, indeed the programmer can give a set of basic rules that each drone can follow. It is important to underline that, in swarm-level approach, there is no possibility to have a shared state between drones; each drone execute the actions specified by the programmer on his own local state. This means that this approach is very easy to scale up to many drones, but it's not suitable for applications that require the drones to explicitly coordinate.

Since we don't want to manage the single drone and we need a sort of "global state" for our framework, the team-level approach, as already said, is a good compromise.

Chapter 4

Programming with Pluto

In this chapter we present our solution to the issues introduced by the indoor context, its system architecture and, in the last section, how we achieved this result. We already said our goal is to complete user-defined missions, using nano-drones in an indoor context. We chose the Team Level approach, which we described in section 2.3 to manage the missions assignment because, as we have shown in Section 3.2, the Team-Level approach is the most suitable approach for the kind of applications that can be developed with our framework. The most important advantage of this approach is the reduced complexity given to the final user, while expressing the sensing tasks: there is no need to describe how the drones should execute them; these details are chosen by the Ground Control Station whose duty is to assign the right drone to the right task and check that each drone take its mission to the end with a successful status.

4.1 System's architecture

Pluto programming framework consists in two main components:

- Pluto Graphical Editor.
- Pluto Main Application.

The former is used by the first actor of the Pluto life-cycle: a developer. The latter is used by a final user whose duty is to insert the sensing task and start their execution. As you can see in figure 4.1, Pluto Graphical Editor lets the developer

to creates a scenario based on the Team Level approach as we show in Section 4.1.1. After that, the Pluto Main Application is generated based on the diagram created in the previous step, so that the final user needs only to insert the sensing task and wait for their accomplishment.

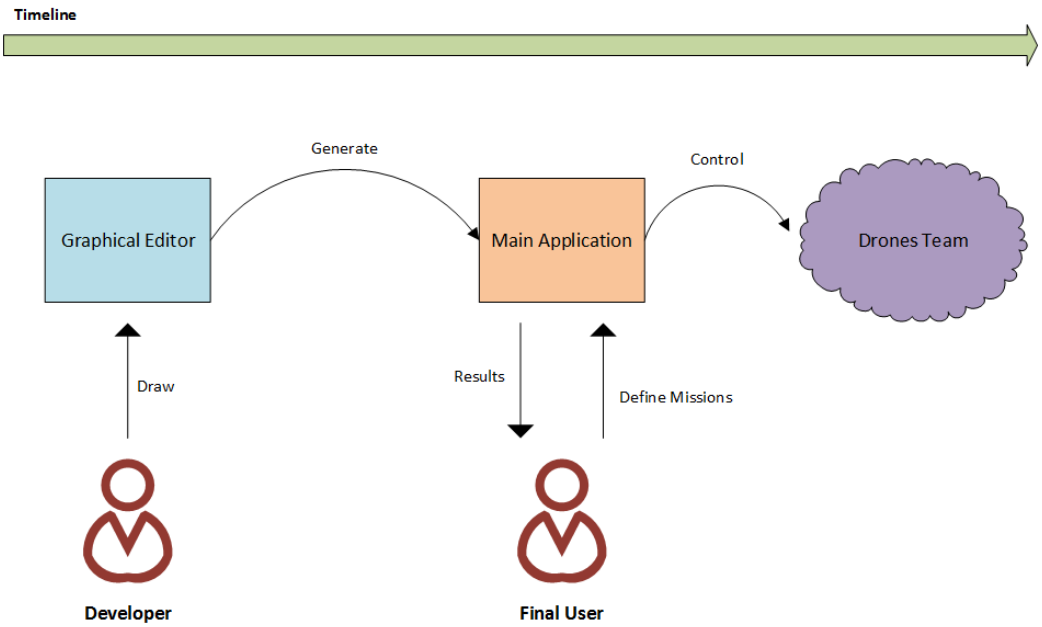

Figure 4.1: A complete life cycle of the application

4.1.1 Pluto Graphical Editor

We created a Graphical Editor in order to give freedom to the developer while designing the final app. The provided tools can be used to link together different kind of blocks, each one with a predefined and implemented logic. When the Editor starts, it shows three main sections: the Palette that contains all the tools available to create a fully functional diagram; the Editor space, where the user can move, link, manage all the created entities; last but not least is the outline with a tree-view of the blocks created by the developer in the editor space.

The developer can choose among several types of pre-created blocks, each one containing a certain logic, explained in section 4.2.2. Creating a block in the editor space can be done simply with a drag and drop gesture or clicking on the desired entity and then clicking on the chosen location in the editor. Then

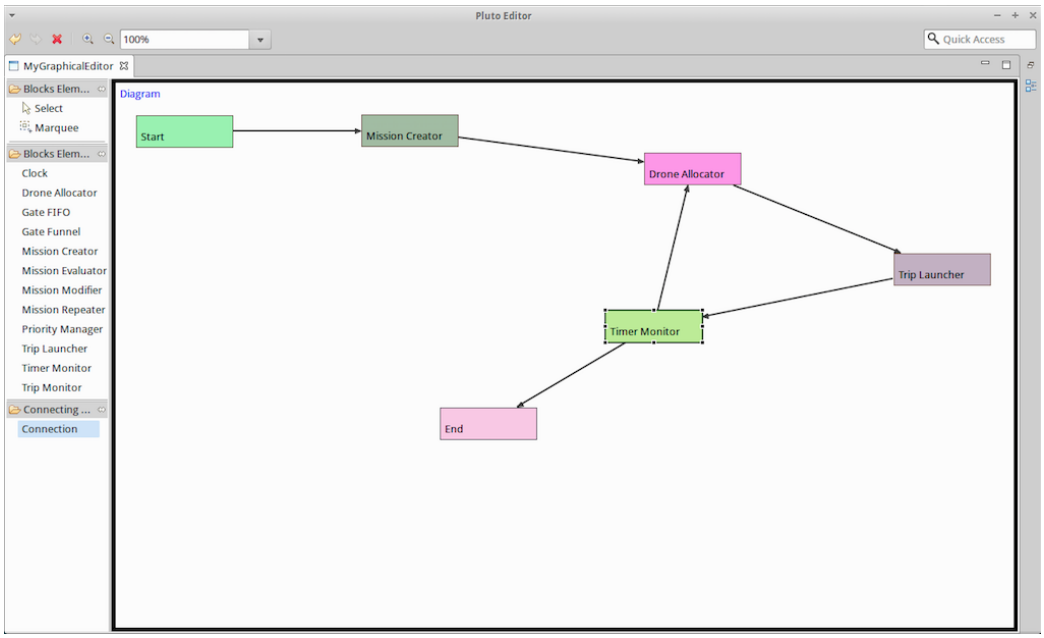

Figure 4.2: Pluto Graphical Editor interface

the user can connect blocks each other using the Connection tool in the Palette section. Apart from the standard functionality, such as Undo, Save, and Load, the Context Menu provides a command to generate a source-code of the Main Application based on the designed diagram. Toolbar provides Undo/Redo, Delete, and Magnify commands. To understand better Pluto Graphical Editor, it's worth to spend a word on the meaning of creating a diagram: each block in the Diagram is black box which is intended to manage a Mission entity. It takes a Mission as an input, works with it and sends it out as an output. The connections among blocks represents the path that the Mission entity will follow after going out from a block. Each block could have multiple outgoing and incoming connections. In the end, on the editor, the developer will have a set of blocks linked together with a set of connections. This drawing can be interpreted as the behavior of the Main Application in managing the missions. For example we designed a sketch of a possible diagram, as shown in figure 4.2. This design is very simple and could be a first skeleton for a more complex application, by simply adding new blocks. Besides of simplicity, our editor is very flexible as well, since it provides a Mission Modifier block whose implementation logic can be written directly in the Editor right-clicking on the block and choosing the option "Write Custom Code". This

will be explained better in section 4.2.2

4.1.2 Pluto Main Application

The Pluto Main Application is the final application that act as a Ground Control Station, managing all the drones and the missions. In this section we explain how it works. Everything starts in the Mission Page where the user can define the tasks that will be carried out by the drones. After clicking to the "Add Mission" button the user is asked to set a name for the mission.

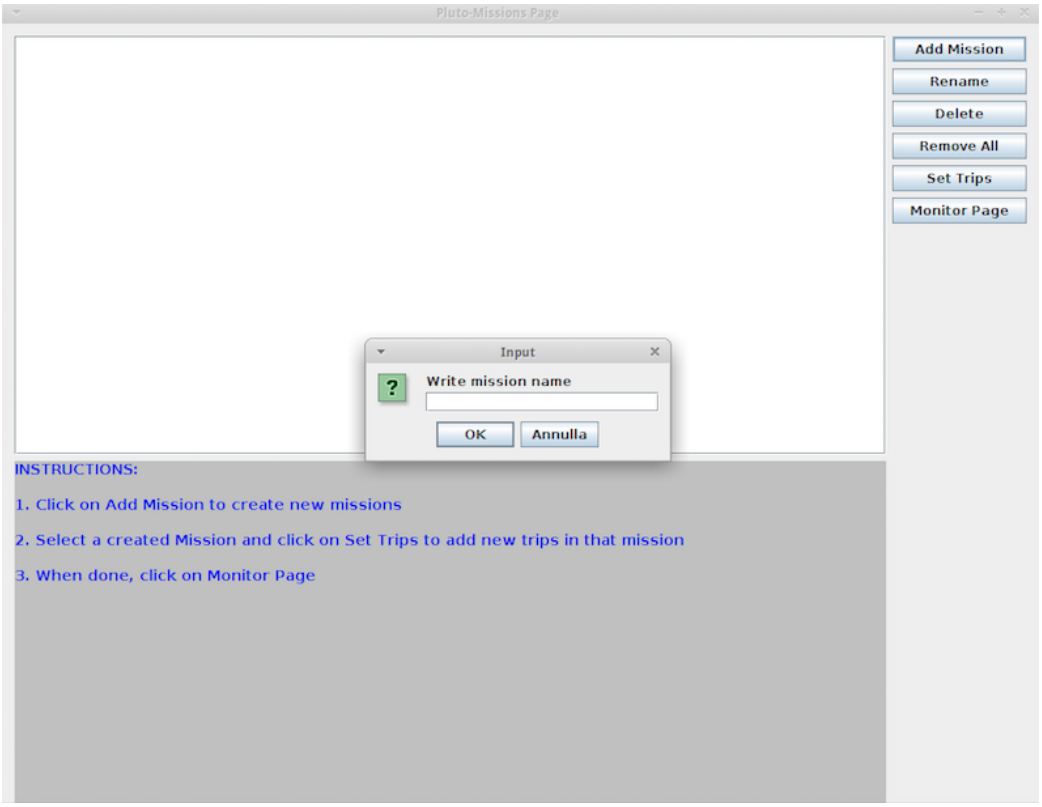

Figure 4.3: Mission Page interface

After that a Mission entity is created, but by now it doesn't contain any information about what has to be done. To add this information the user has to double-click on the mission in the main list or click on the "Set Trips" button. A new window will appear, as shown in figure 4.4, and the user can add Trips entity to the related Mission. A Trip is nothing but a movement from point A to point

B inside our indoor context. Trips are the basic entities that constitute a single Mission. The single Trip contains information about the Action to execute once point B is reached. Furthermore the Trip has a reference of the drone assigned to it, but we will talk about this in section 4.2.2.

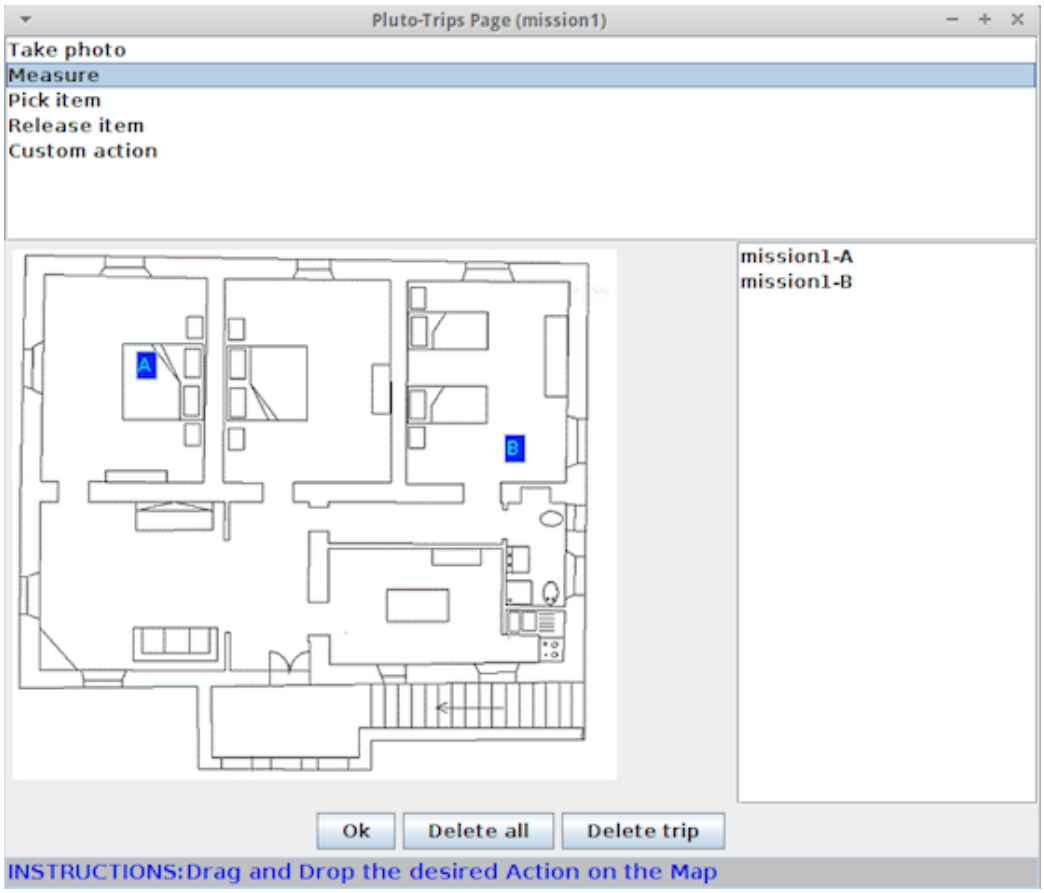

Figure 4.4: Trips Page interface

So after all the Trips are added to a single Mission the user can create more Mission entities or pass to the Monitor Page with the corresponding button. The Monitor Page, shown in figure 4.5, is the window where the user can obtain information about the running missions, at run-time. On the top, there is a table where each row is assigned to a Mission, each column will display the information about the current Trip that is executing and the Drone that has been assigned to that Trip. Below the table there is a console where log messages are printed during the execution of each missions. In this way the user can obtain run-time information

about the status of the entire system. Of course the Start button will start the execution of the created missions, while the Stop button will prompt the user to a behaviour choice: "RTL" or "Land". The first will make all drones to return to the home location instantly, while the second option will make all drones to land in their current locations. After the Stop command, the missions status will be preserved and could be continued in the future.

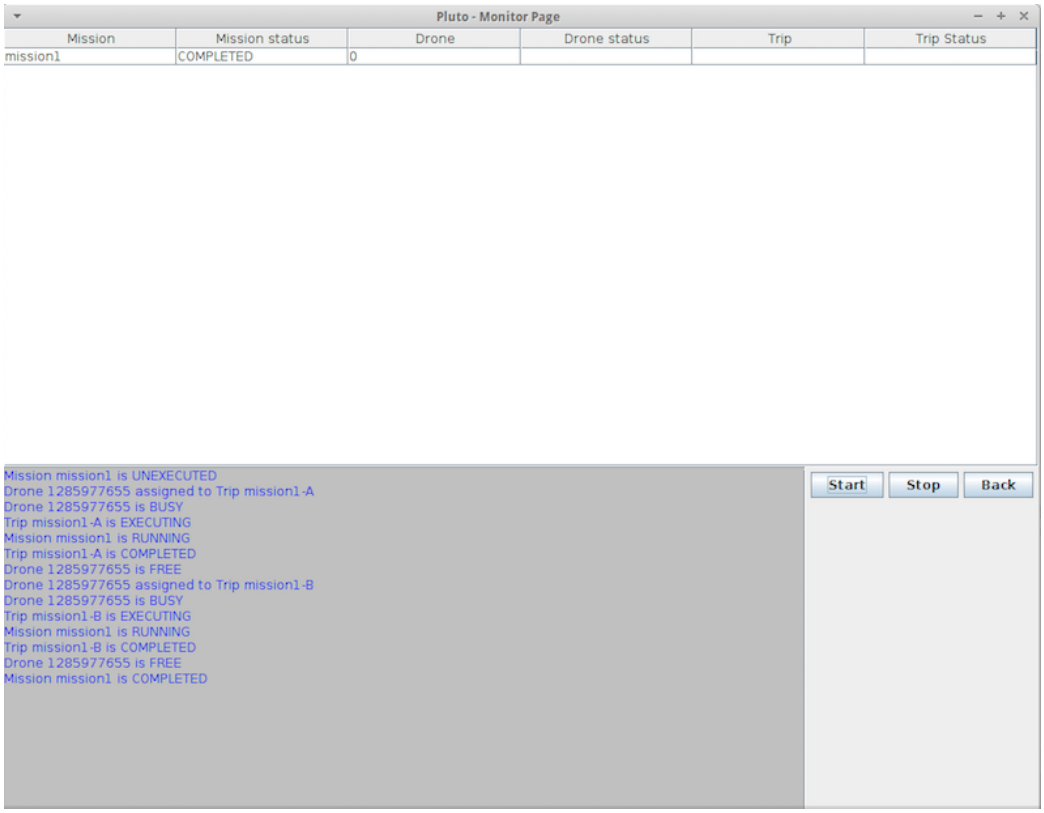

Figure 4.5: Monitor Page interface

4.2 Dataflow model

In this section we present the Dataflow model in a complete and sound way. First, we describe a general view of the model in Section 4.2.1. Then, in Section 4.2.2 we focus on the details of the each component of the model.

4.2.1 Description of the model

Each diagram created with the Pluto Graphical Editor is made of many functional blocks, each one is implemented with a particular logic; the user can select the blocks needed for his particular application and connect them through simple links, as explained in section 4.1.1. In the figure 4.6, an example application is showed, which contains some of the implemented blocks. Again, the user is not forced to insert all the blocks, he can choose only the blocks needed for his particular application. If you compare this figure with figure 4.2, you can see that this one is based on the the same drawing but with additional features provided by new blocks.

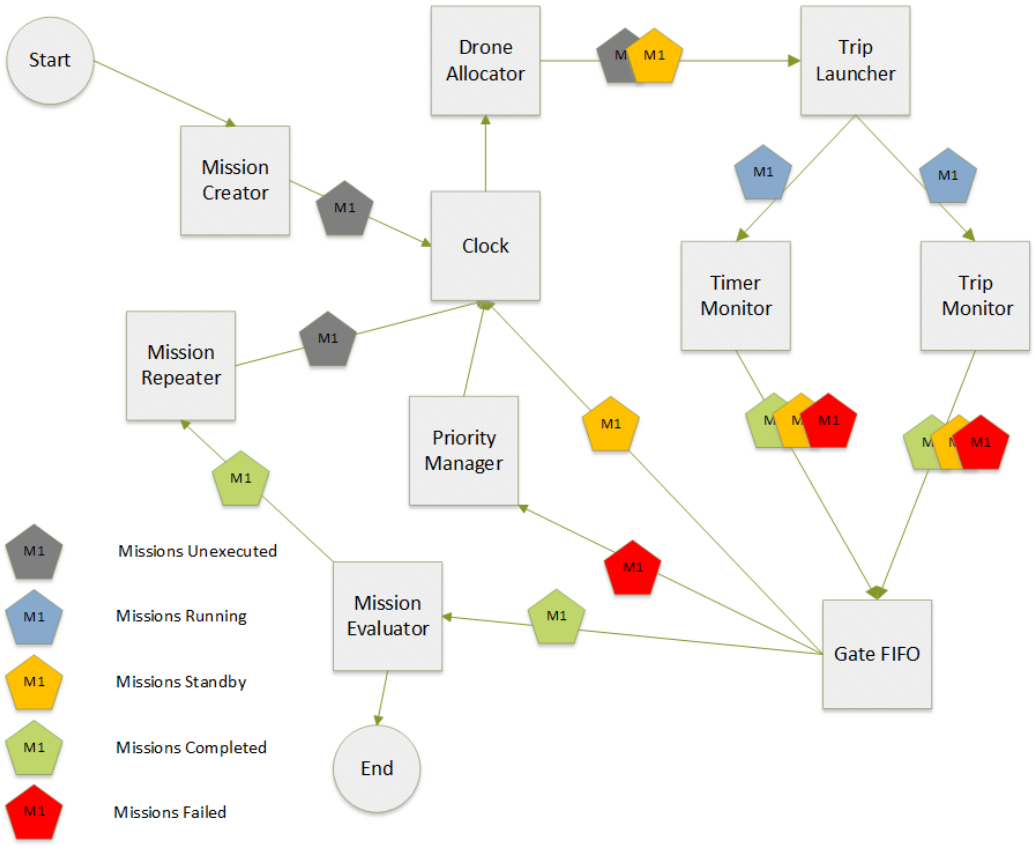

Figure 4.6: An example Pluto application

4.2.2 Model representation

In this section we show the fundamental entities of our model, in order to better understand the functioning of the blocks. Then, all the blocks of figure 4.6 are described in details, explaining which tasks they perform.

Entities

Basically, through the interface presented in section 4.1.2, the user specifies a list of Missions that the system must perform. Each Mission contains a list of Trips. A Trip is a path between a start location and a target location. The Trip is performed by a Drone which carries out an Action, for example it can bring an Item to a person, or take a photo or measure temperature at a particular location. It is important to underline that, inside each Mission entity, the trips are executed

sequentially, in general by different drones. The missions instead are executed in parallel.

So, as already said, we identified the following entities:

- Mission
- Trip
- Drone
- Action
- Item

As shown in figure 4.7, the Mission entity, in addition to the set of Trips, contains another important attribute that is the Status: it describes how the Mission is executing.
The Trip entity has a Status attribute too and furthermore it contains the Drone and the Action entities. Of course it also contains the coordinates of the source and target locations.
The Drones are described by a unique ID and by a Shape Category which tell the system what kind of items can be hold by the drone.
Regarding the Action entity, a very important feature is the *Custom Action*: it allows the programmer to implement a new action that the drones can perform. It obviously depend on the particular application, but for example a programmer can add the Pollinate action needed for the Alfalfa application[6], which will be described in section 6.1.1.

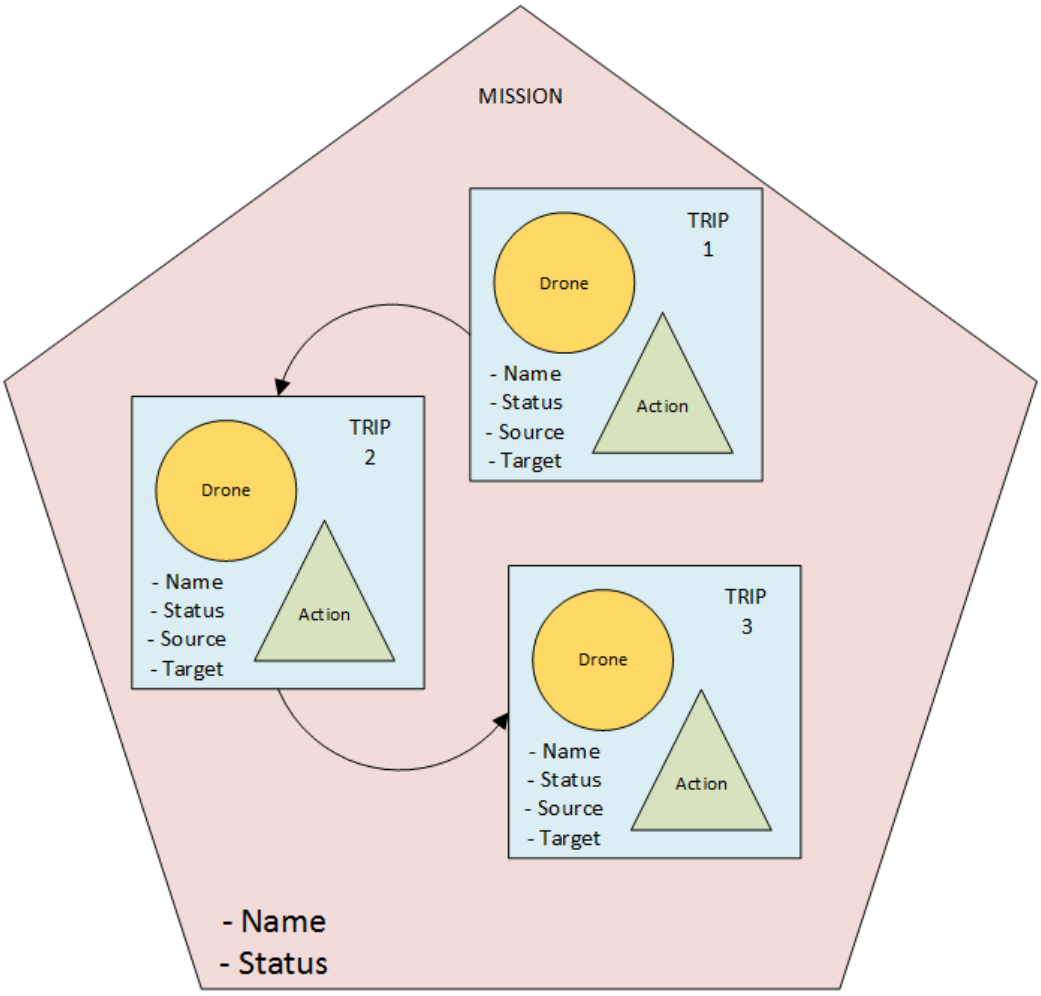

Figure 4.7: Relationship among model entities

Description of the blocks

Here we provide a detailed description of each functional block available in the Pluto Graphical Editor:

Mission Creator block

input: a list of Trips
output: a Mission object

The Mission Creator block receives as input the trips that the user wants to be performed by the drones, then it creates a Mission including all these trips and returns that Mission object. This block is the starting point of each Pluto-developed application.

Clock block

input: a Mission object
output: a Mission object

The Clock block checks the delay attribute of each Trip of a Mission: if it's not equal to zero, it makes the Trip wait for the amount of time set by the user in the Mission creation step, and finally returns the Mission Object. If the programmer puts the Clock block in the graph of the application, the Pluto Main application shown in section 4.1.2 will show an extra widow. On the Trips Page,after the final user drags and drops on the map the action to perform and the Priority panel is shown, the user will be asked to set a delay for the Trip. This block has to be put between the Mission Creator and the Drone Allocator blocks.

Drone Allocator block

input: a Mission object
output: a Mission object

The DroneAllocator block chooses the appropriate Drone to be assigned to each Trip, basing on the availability of the drones and their capability to perform the desired action. The output of the DroneAllocator is always sent to the TripLauncher, while its input can arrive from a variety of blocks, depending on the features of the considered application.

Trip Launcher block

input: a Mission object
output: a Mission object

The TripLauncher block takes the next Trip to be performed from the Mission object, and launches it. The assigned drone will fly to the target location and execute the defined Action. This block receives the input Mission from the DroneAllocator block, ans sends its output to the TripMonitor. If the application contains the TimerMonitor block, the TripLauncher's output will be also sent to it.

Trip Monitor block

input: a Mission object
output: a Mission object

The TripMonitor block checks the status of the Trip that is running in that moment. It changes the status of the Trip and of the Mission in the appropriate way, depending on whether the Trip is failed or complete. This block receives the input Mission from the TripLauncher, and its output can be sent to the GateFIFO, the MissionEvaluator or the DroneAllocator blocks, depending on the features of the particular application.

Mission Repeater block

input: a Mission object
output: a Mission object

The MissionRepeater block verifies if the *repeat* attribute of the input Mission is set to true. If so, it set the Mission status to STANDBY and the status of all the Trips of that mission to WAITING and inserts again them in the List of Trips to be executed. This block is put between the MissionEvaluator and the DroneAllocator.

Gate FIFO block

input: a Mission object
output: a Mission object

The GateFIFO block is used when two or more blocks works in parallel, and only one instance of the executing Mission must propagate. Among the multiple

input connections of the GateFIFO block, it propagates only the one that completed its task for first. That's why the FIFO acronym is used, since the first Mission instance that arrive is the only one that propagates in the graph. This block receives the input Mission from one of the blocks connected to it and can send its output to the MissionEvaluator, MissionRepeater or DroneAllocator blocks, depending on the particular application.

Gate Funnel block

input: a Mission object
output: a Mission object

This block is similar to the GateFIFO, but this gate wait for all the Mission instances it has in input to complete. So if before this gate, there are 4 blocks in parallel, the propagation of the Mission is activated only when all the 4 instances arrive.

Mission Evaluator block

input: a Mission object
output: a Mission object

The MissionEvaluator block can read and write data carried by the drones, and basing on them can perform various actions specific for the application developed. It receives the input Mission from the GateFIFO or TripMonitor blocks and can send the output to the DroneAllocator or MissionRepeater blocks, depending on the particular application.

Mission Modifier block

input: a Mission object
output: a Mission object

The MissionModifier block allows the programmer to create new custom blocks. As shown in figure 4.8, in the editor, he can inserts his custom code in this block using the appropriate option in the context menu. This block can be put in every

point of the Pluto Editor graph, depending on the particular feature it implements.

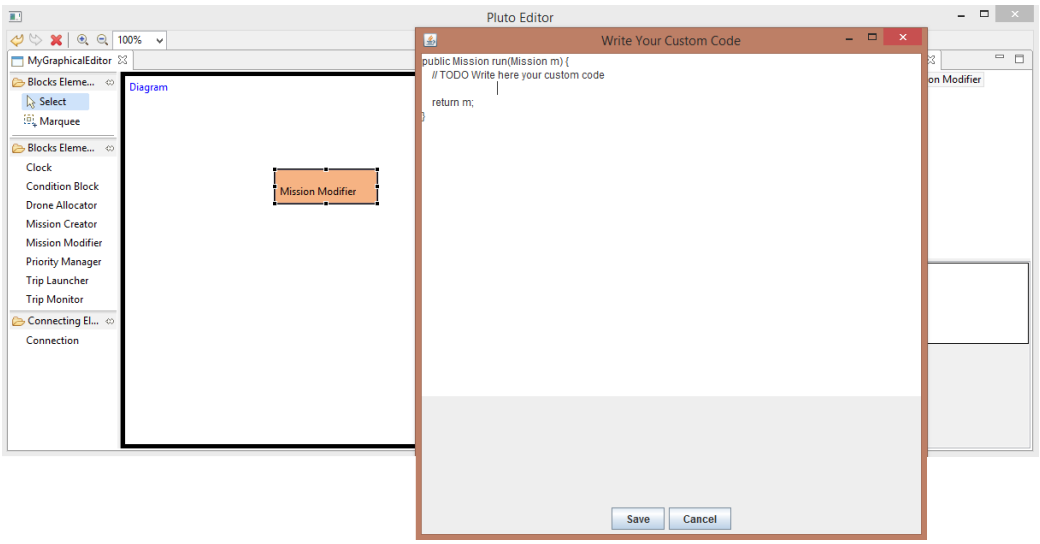

Figure 4.8: The MissionModifier block

PriorityManager block

input: a Mission object
output: a Mission object

The Priority Manager block raise the priority of the first Trip of the Mission passed in input. This block can be considered as a particular case of Mission Modifier. This block can receive the input from the GateFIFO or Trip Monitor blocks, and sends the output to the Drone Allocator.

Timer Monitor block

input: a Mission object
output: a Mission object

The TimerMonitor block takes the input mission and waits for the completion of the current running Trip. If the Trip doesn't end before the fail-safe time, it will set the status of the Trip and of the Mission both to FAILED, then it will

put the mission as output. This block can be considered as a particular case of MissionModifier. It's always put between the TripLauncher and the GateFIFO blocks.

Start block
State the beginning of the diagram

End block
State the end of the diagram, where the completed missions will go into.

4.3 Aiming to the final model

In this section we describe our previously developed solutions, which we refined many times in order to obtain the final working version of the Pluto programming framework; this is done by using a top-down approach, starting from the final implementation to the very first one.

4.3.1 Solution without Trip entity

In the version precedent to the final solution presented in section 4.2, we did not have a concept of Trip, and Mission was the main concept the whole model was based on. The following diagrams shows this in the particular case of the Timer feature, which contains also a "Switch source-target" block. Later, this block was included in the logic of the Trip.

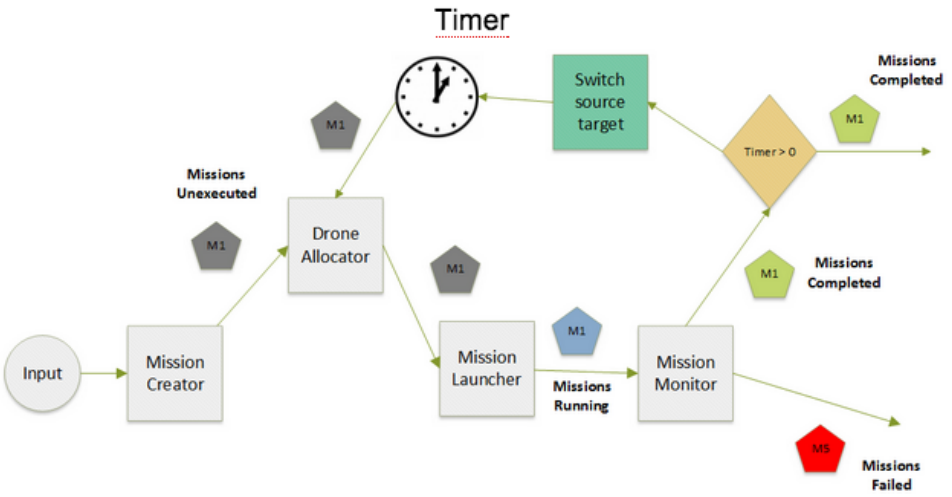

Figure 4.9: Solution without the Trip concept

After analyzing the model, we realized that we needed the concept of Trip, because the final user must have control on the single Trip of a drone, in order to decide which action the drone must perform, and to have an opportunity to control the Trip such as delay, stop, or delete, without deleting or stop the whole Mission. With this precedent solution it is not possible, because having only the entire Mission to manage, the user can no control the single Trip, and if he/she wants to delete only a part of the Mission he cannot do so, and he/she is forced to delete and build again the whole Mission.

4.3.2 Solution without the DroneAllocator

This solution instead of the DroneAllocator block, used the "Drone Updater" one. This block managed the assignment of the Drone to a Mission, only in special conditions. This means that, generally, there were no need of this block unless the developer put some special block such as the old TimerMonitor or the old DelayMonitor. In this solution, the MissionCreator managed the assignment of a Drone to the Mission. This is the reason why we didn't need the DroneUpdater in normal conditions. As said, there were also the "Delay Monitor" and "Timer Monitor" blocks, instead of the Clock block. These blocks managed the delayed missions(fig. 4.11) and the missions with an associated timer(fig. 4.10), respectively.

There wasn’t the MissionModifier block, but only the "Priority Manager" one, so the programmer couldn’t insert blocks(fig. 4.12) with custom code.

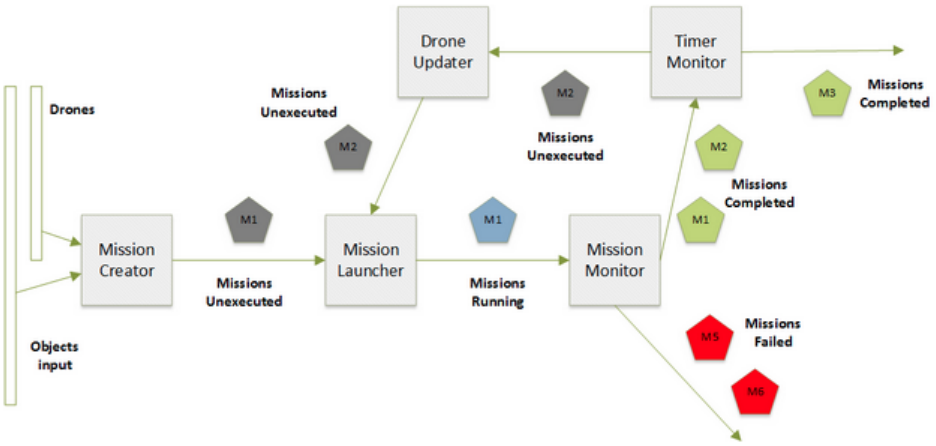

Figure 4.10: Solution with the TimerMonitor

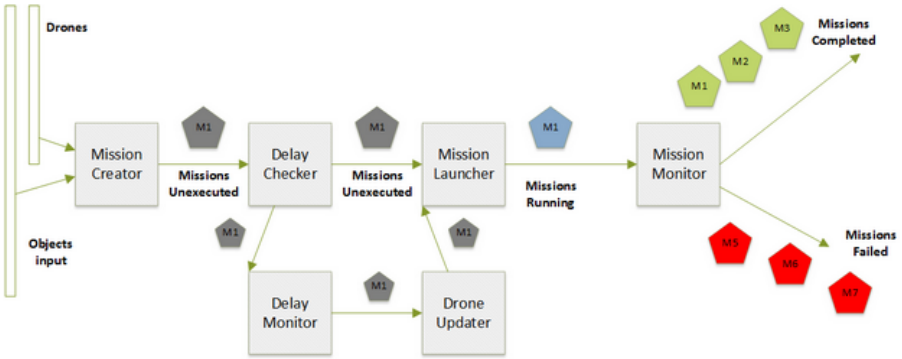

Figure 4.11: Solution with the DelayMonitor

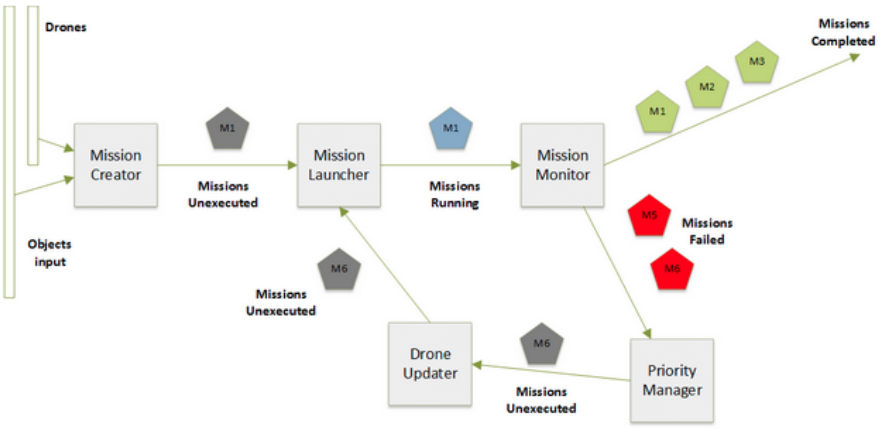

Figure 4.12: Solution without the MissionModifier block

Actually we decided to put the DroneAllocator block because weneed to separate the creation of a Mission Object from the assignment of a Drone to it; in this way we could also remove the DroneUpdater block, because now we have an apposite block which manage only the assignment of Drones, so there is no more need to distinguish between the normal assignment and the special assignment(in the case of delayed trip). Also the DelayMonitor and TimerMonitor blocks were useless, because there is no need to distinguish between a delay and a timer, so a single Clock block can manage these cases both. In this solution only the PriorityManager block existed, but we decided to create a MissionModifier block in which the user can put his own code he needs for a particular application, so the PriorityManger block can be seen as a particular case of the MissionModifier one.

Chapter 5

Implementation

In this chapter we show how we implemented the Pluto Framework, describing the main elements of the project separately, in order to better understand their behaviors.
In figure 5.1, we show the final architecture scheme that includes all the parts described in the following sections.

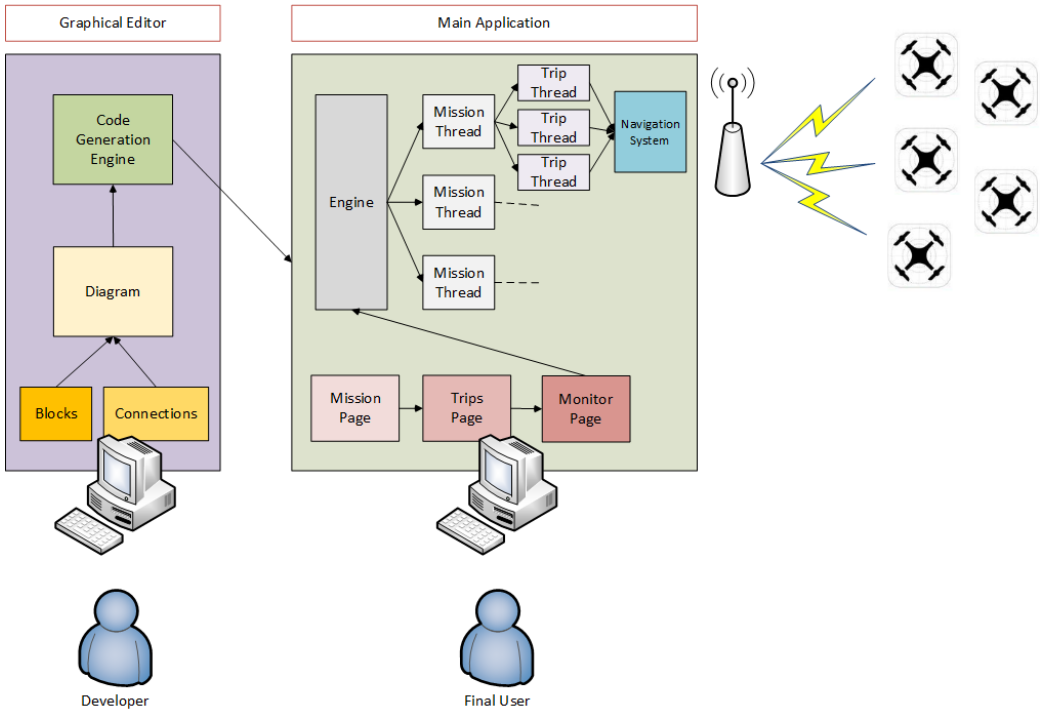

Figure 5.1: Pluto architecture representation

5.1 Graphical editor

In order to create the Graphical Editor, described in section 4.1.1, we decided to use the GEF (Graphical Editing Framework) project. This framework is a Java technology and it is part of the Eclipse framework developed by IBM. It gives developers a full solution for the graphical modeling of a Java object model, and it can be used in conjunction with other technologies such as EMF (Eclipse Modeling Framework) or GMF (Graphical Modeling Framework), to enable the creation of a complete graphical modeling suite. This means that the Pluto Graphical Editor has been developed as an Eclipse Plugin, so the developer has to install the Eclipse IDE in order to exploit the editor.

First of all, we created the Java classes of the blocks. Each class contains the code of the corresponding block. We defined each block as a rectangle figure, then we added the connection entity in order to enable links between them. All these entities are children of a main container class that represent the diagram itself.

When the user creates a new block in the editor area, a relative block entity is automatically created and added to the diagram container class. The same operation stands for the connections creation.

After the user draws the desired graph he can choose to generate the final code of the Main Application, through the apposite voice in the context menu. See section 5.2 for further details.

5.2 Code generation

Once the programmer has created the graph of the application through the Pluto Graphical Editor, he can generate the code in order to make the Pluto Main Application behavior coherent with the graph.

This can be done by right clicking on the graph and choosing the "Generate code" command.

The Editor includes a template of the Main Application, in which almost all the lines of code are ready to be executed. However, this template contains several tags that the generation process will replace with specific lines of code, depending

on the drawn graph.

The generation process consists in the search for these tags inside the template application. For example, there is a "<dec>" tag that is a placeholder for the declaration part of the created blocks. It means that this tag will be replaced with the declarations of the Java classes of the blocks.

There are also tags that are placeholder for attribute values. For example the "<tDelay>" tag will be replaced with a "true" or "false" string, depending on the presence of the Clock block in the diagram. This allow us to ask for a delay value for the trip, only if the developer has inserted a Clock block in the diagram. The same solution is adopted for the "Mission Repeater", "Timer Monitor" and "Priority Manager" blocks. For each of them, there is a tag that will make the application ask the user for the respective values, if these blocks have been inserted in the graph.

5.2.1 From graph to code

The main issue in the generation process was to understand how to generate the code from a general diagram. Potentially, a developer could draw a very complex graph with lot of blocks and connections between them. At first, we focused on graph exploration methods, but we immediately noticed that they were too complex. So, we decided to adopt a *Publish-Subscribe* design, making use of the *Observer* pattern, shown in figure 5.2.

After the generation of the declaration part, we added an "<exe>" tag, that is replaced by the subscription part of the cited pattern. The *Observer* pattern consists in the declaration of some elements as *observers* and of other entities as *observable*. When an observable object change its status, it sends a notification to all its observer entities. These observers will react according to the change of the observable object.

In this way, we made each declared block both Observer and Observable. This means that each block observes another block that comes before it, but at the same time, it is observed by other blocks coming after it. The change of status consists in the output of the Mission object. When a block ends to perform its operations, it notifies all its observers passing them the Mission entity. Then, all the observers take as an input that mission.

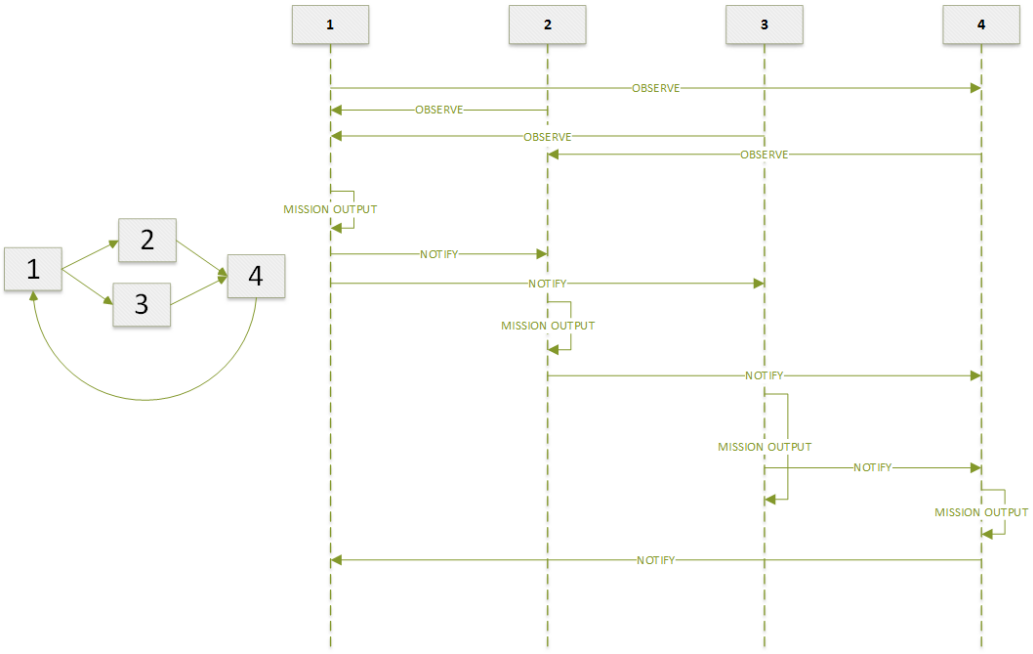

Figure 5.2: Observer design pattern

This mechanism let us describe the execution flow of a virtually very complex diagram besides those that are more simple.

5.3 Object-oriented approach

We used the Java programming language to implement both the Graphical Editor and the Main Application. We made this choice because we are very familiar with Java, since almost every academic project we implemented in these years made use of this Object-Oriented programming language. The Object-Oriented approach perfectly suits the Pluto model, since we have different independent entities such Drones,Missions,Trips that interact together in the execution of tasks.

We decided to adopt a Model View Controller(MVC) approach, whose functioning is shown in figure 5.3. The central component of MVC, the model, captures the behavior of the application in terms of its problem domain, independent of the user interface. The model directly manages the data, logic and rules of the

application. A view can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible. The third part, the controller, accepts input and converts it to commands for the model or view.

A controller can send commands to the model to update the model's state. It can also send commands to its associated view to change the view's presentation of the model. A model notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands. A view requests information from the model that it uses to generate an output representation to the user.

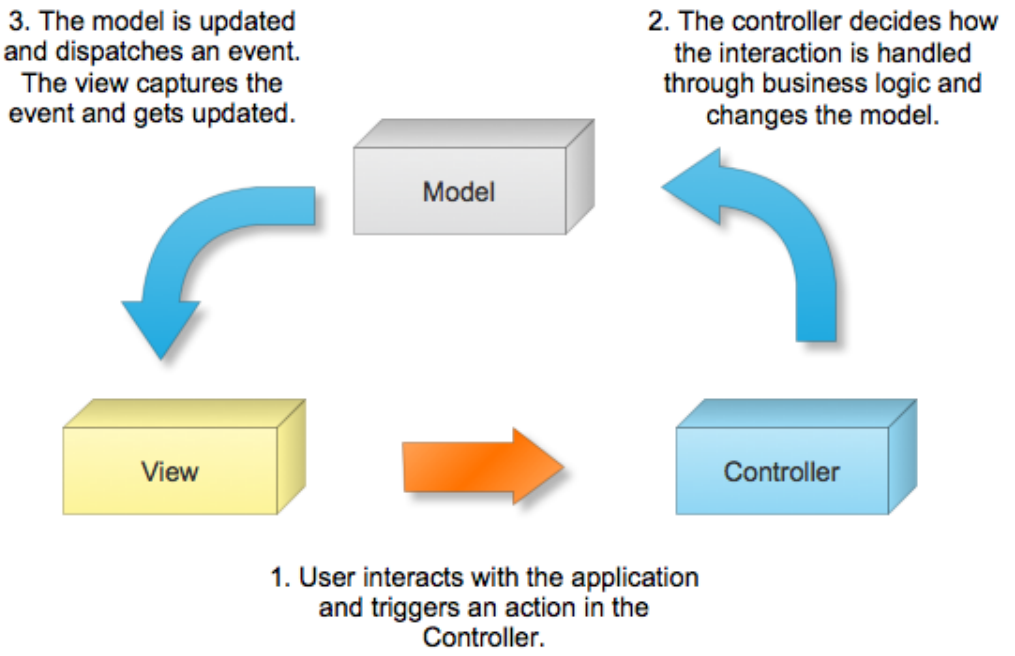

Figure 5.3: The MVC pattern

5.4 Runtime Management

Figure 5.4 shows as the Main Application manages the mission execution with a parallel programming architecture. Indeed when the user starts the execution,

the system launches each mission in a new thread, in order to guarantee a reliable parallel execution.

Then each mission starts its flow among the blocks, thanks to the Observer design pattern, described in section 5.2.1. When the mission enters in a new block, the application launches a new thread, in order to run the mission management code of the block. We need this new thread because there is the possibility that two or more parallel blocks have to manage the same mission entity at the same time.

Therefore, when the mission reaches the "Trip Launcher" block, the system starts the trip. Doing this, it creates a new thread, to manage the trip execution till its end.

As said, each mission and each trip created by the user, have a respective thread that deal with the execution of the entity from the beginning to the end. In this way, the blocks that need to monitor these entities can observe the status of the threads, in order to know if the trip/mission is still running or has already completed.

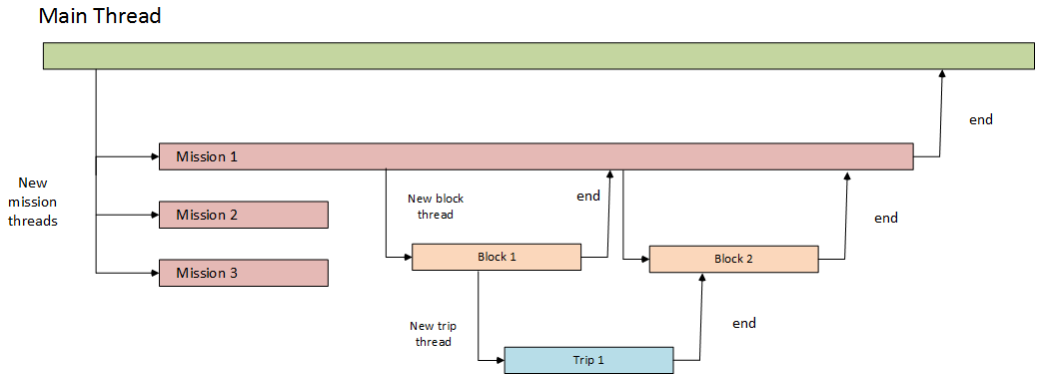

Figure 5.4: Example of thread concurrency

5.5 User interface

We chose the Swing framework to develop the Pluto User Interface, already described in section 4.1.2. We made this choice since Swing is well known to

us. Indeed we used it for the development of many academic projects, where we noticed that it allows to build graphical interface in a very fast and easy way and to add a great variety of components.

Swing library is an official Java GUI toolkit released by Sun Microsystems. It is used to create Graphical user interfaces with Java. The main characteristics of the Swing toolkit are:

- platform independent
- customizable
- extensible
- configurable
- lightweight

Swing is an advanced GUI toolkit. It has a rich set of widgets: from basic widgets like buttons, labels, scrollbars to advanced like trees and tables. Swing itself is written in Java and is part of JFC, Java Foundation Classes: it is a collection of packages for creating full featured desktop applications.

There are basically two types of widget toolkits: *Lightweight* and *Heavyweight*. A heavyweight toolkit uses OS's API to draw the widgets. For example Borland's VCL is a heavyweight toolkit since it depends on WIN32 API, the built in Windows application programming interface. As already said, Swing is a lightweight toolkit since it paints its own widgets.

5.6 The crazyflie nano-quadcopter

For the concrete actuation of the sensing tasks required by each application, we chose the Crazyflie Nano-Quadcopter, shown in figure 5.5.

Figure 5.5: The Crazyflie Nano-Quadcopter

The Crazyflie is a tiny quadcopter often referred to as a nano-quad, built using the PCB itself as the frame,developed solely by open source tools. The Crazyflie specs are the following:

- Small and lightweight, around 19g and about 90mm motor to motor
- Flight time up to 7 minutes with standard 170mAh Li-Po battery
- Standard micro-USB connector for charging which takes 20min for the stock 170mAh Li-Po battery
- On-board low-energy radio@1mW based on the nRF24L01+ chip. Up to 80m range (environment dependent) when using the Crazyradio USB dongle
- Radio bootloader which enables wireless update of the firmware
- Powerful 32 bit MCU: STM32F103CB @ 72 MHz (128kb flash, 20kb RAM)
- 3-axis high-performance MEMs gyros with 3-axis accelerometer: Invensense MPU-6050
- Available footprints to manually solder magnetometer HMC5883L/HMC5983 or/and barometer MS5611

- 4-layer low noise PCB design with separate voltage regulators for digital and analog supply

To concretely control the Crazyflie, there is a Python library which gives high level functions and hides the details. We used a particular API to send the control command:

```
send_setpoint(roll, pitch, yaw, thrust)
```

The arguments roll/pitch/yaw/trust is the new set-points that should be sent to the copter. For example, to send a new control set-point:

```
roll  = 0.0
pitch = 0.0
yawrate = 0
thrust = 0
crazyflie.commander.send_setpoint(roll, pitch, yawrate, thrust)
```

Changing the *roll* and *pitch* will make the quadcopter tilt to the sides and thus change the direction that it's moving in. Changing the *yaw* will make the quadcopter spin. The thrust is used to control the altitude of the quadcopter.

By dynamically adjusting these four parameters we can make the Crazyflies move to the locations specified by the user through the Pluto User Interface.

Chapter 6

Evaluation

6.1 Applicability of the Pluto framework

We developed our programming framework thinking about indoor applications utilizing nano-drones. Actually, since we work at a sufficiently high level of abstraction, because we can use an API which make the drones navigate in the environment, the model can be extended to almost every kind of drone, aerial terrestrial and aquatic. So, we can say that the Pluto programming framework is "drone independent", and this greatly extends its applicability, including also outdoor, aquatic and terrestrial environments; it is in charge of the programmer to manage the interaction between Pluto and the specific type of drone he wants to use for the particular application he's developing.

Pluto is fully exploited when there is a team of drones to manage (see 6.14 and 6.15), although it perfectly works also in the case of a single drone (see 6.13).

Since we decided to use a Team-level approach (see 2.3), our model can be used for developing applications where the user can give to the system a set of actions to be performed; the dispatching of these actions is managed by the "central brain", which takes care of assigning the drones to the action and to handle all the exceptions (battery low, crashes etc.). So, the drones are only actuators that perform an action, there is no communication between them, their behavior is monitored and decided by the central brain.

Since drones cannot communicate between them, Pluto cannot be used for applications where drones must perform some kind of action requiring explicit communication or data exchange between them; the logic is managed by the central

brain, so communication between drones is always mediated by this component; a drone can send data to the central brain, and this could send again that data to another drone.

Hereinafter we analyzed some already existing example applications, showing whether they can be managed/developed with the Pluto programming framework or not.

6.1.1 Alfalfa Crop Monitoring and Pollination

The Alfalfa Crop Monitoring and Pollination[6] is a typical example of swarm-level approach application. Alfalfa is an important food crop for cattle and requires an external pollinator (e.g. bees) to produce seeds. In recent years, colony collapse disorder has devastated honeybee populations and jeopardized the cultivation of important crops[**?**]. A swarm of drones can pollinate the alfalfa plants and also monitor them for pests and diseases, trough visual spot checks. So, the whole application provide three periodic actions: searching for pests, searching for diseases, and looking for flowers in bloom. Each one of these actions is achieved by taking pictures of the plants. The user may need to define time constraints within the pollination action must be completed.

The following Pluto Editor graph describes the behavior of the Alfalfa Crop Monitoring and Pollination[6] application:

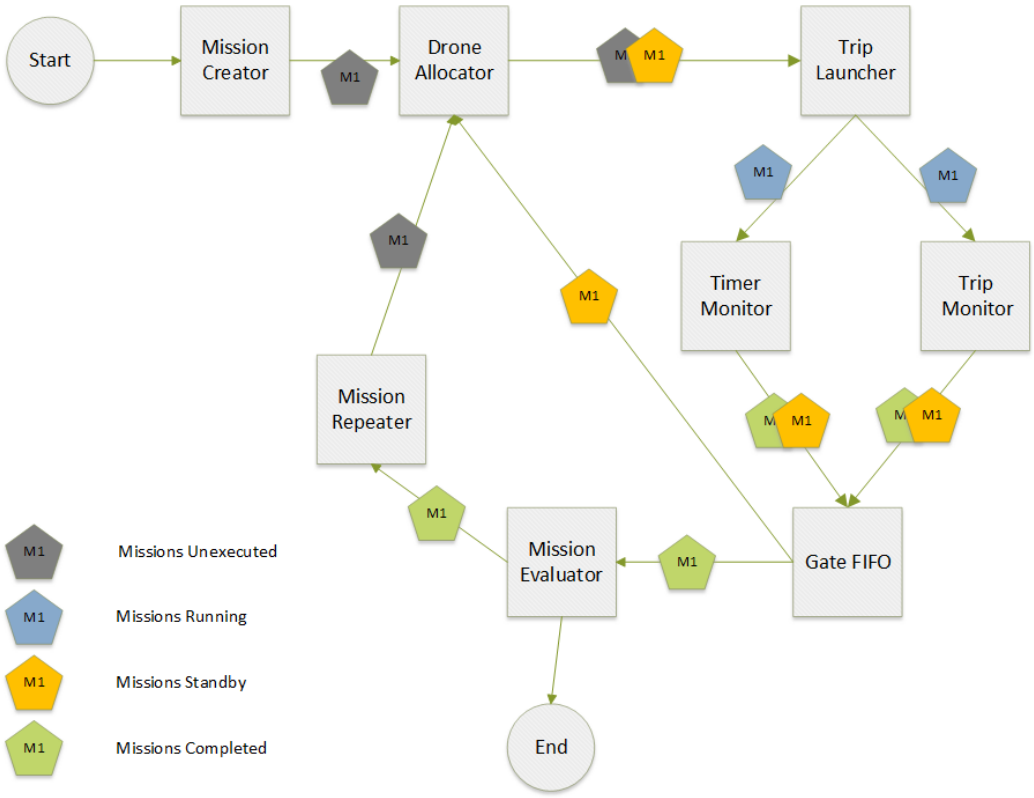

Figure 6.1: Pluto graph for the Alfalfa Crop Monitoring and Pollination application

Thanks to the *Take photo* action, already implemented in Pluto, the drones are able to perform the monitoring of leaves for pests, diseases and flowers in bloom.

As already explained, a Mission object contains a list of trips to be executed. Inside the Mission entity, these trips are performed sequentially, in general each one by a different Drone, as already explained in section 4.2.2. So, if the user wants to send more than one drone simultaneously on the same location, he has to create more than one Mission. Indeed missions are executed in parallel, so if the user wants to simultaneously send 3 drones on the same location, he simply has to create three missions. Then, since each Mission has its own Trips Page, he has to choose the same locations on the maps of the three Trip pages.

To concretely choose the specific locations to monitor, the user is provided with a map over which he can drag and drop the action *Take photo*.

The Mission Repeater block takes care of continuously sending the drones to monitor these locations. Regarding the "Pollination" task, it can be added thanks to the *custom action* feature, through which the programmer can add to the model a brand new Action, making use of a specific external API.

Concerning the pictures evaluation to detect pests, diseases and flowers in bloom, the developer has to add the custom code in the Evaluator class, using again an external API. Each photo will have three associated parameters: the boolean attributes *pest*, *disease* and *bloom*. These attributes are false by default and they are set to true when the leaves are damaged, their color turns greenish-white or the flowers are in bloom, respectively.

The MissionEvaluator block enables the evaluation of the photos taken by the drones. If the pest and/or disease attributes are true, the system notifies the farmer of the damaged location, adding a log line in the console of the Monitor Page of the Pluto User Interface. If the bloom attribute is true, a new Trip will be created and a new Drone, capable to perform the Pollinate action, will be sent to that location to pollinate the flowers.

The time constraints, which are set by the user thanks to the timer attribute of the Mission entity, will be respected thanks to the TimerMonitor block. This block takes care of setting the Mission status to FAILED if one of its Trips is not completed within the timer.

The GateFIFO block takes as input two Mission instances, one from the Trip Monitor and the other one from the Timer Monitor. It takes care of propagating only the first instance that arrives to it. For example, if the timer has expired then it will propagate the Timer Monitor instance, otherwise the Trip Monitor one.

After the GateFIFO block there is a bifurcation: if the Mission is completed, it is sent to the MissionEvaluator, otherwise there are some trips of the mission that must be executed again, and so the Mission is sent to the DroneAllocator, which will assign new drones to these trips. The full explanation of the functionality of each block can be found in section 4.2.2.

The following is the code of the Evaluator needed for the development of the Alfalfa[6] application: "dataMap" is an hashmap that binds each Trip with the picture taken. The Trip is the key, which represents the journey performed by the drone, the Photo is the value, which is the picture taken by the drone once the Trip has been completed. For each photo, if the *pest* or the *disease* attribute are true, the system will signal to the farmer the location where the problem exists, through the log function. If the *bloom* attribute is true, the plants at that location must be pollinated: a new Trip entity is created, its Action is set to "Pollinate" and the target location is set to the same location of the Trip that found the bloom. Finally the Trip is added to the list of trips of the current mission. The Mission status is set to STANDBY because there is at least a new inserted Trip to execute.

```
1     String result = null;
2
3     // Retrieve all entries of the map, it means we are iterating
4       // all the completed Trips that wrote their result in the Evaluator
5     for (Map.Entry<Trip, Object> entry : dataMap.entrySet()) {
6
7       // we need to consider only the Trips related to the current
8         // mission we are evaluating
9       if (missionToEvaluate.getCompletedTrips().contains(entry.getKey())) {
10
11        // retrieve the Photo related to the current Trip
12        Photo photo = (Photo) entry.getValue();
13
14        if (photo.hasPest() || photo.hasDisease())
15
16          return "WARNING: Pest/disease at location: "
17               + entry.getKey().getTargetLocation();
18
19        if (photo.hasBloom()) {
20
21          // create a new Trip to pollinate the flowers
22          Trip trip = new Trip();
23          trip.setName("PollinateTrip");
24
25              // Set the same target location
26              // of the Trip that has found the flowers
27          trip.setTargetLocation(entry.getKey().getTargetLocation());
28
29              // This action must be implemented by the developer
30          trip.setAction(Action.POLLINATE);
31
32              // the status WAITING means that this Trip
33              // is ready to be launched
```

```
34              trip.setStatus(Trip.WAITING);
35
36              // adding this Trip to the Trip list
37                  // that contains all the Trips to be launched
38              missionToEvaluate.getTrips().add(trip);
39
40              // The status STANDBY means that the mission
41                  // has some Trips to be executed
42              missionToEvaluate.setStatus(Mission.STANDBY);
43            }
44          }
45        }
46
47        // Result is a "success" because all the Photos of this
48          // mission have been evaluated
49        result = "Success";
50
51        return result;
```

The following is a possible source code of the Pollination Action:

```
1   // This is the method called by the drone
2   // after it reaches the target location
3   @Override
4   public Object doAction() {
5     Pollinator pollinator = System.getPollinator();
6     pollinator.pollinate();
7     return true;
8   }
```

To further clarify the development of the Alfalfa[6] application with Pluto, we now show a real execution of the Alfalfa application in a concrete scenario: imagine we want to simultaneously send three drones to monitor the plants distributed in a circular area. So, we create three Mission entities, and, for each of them, we drag and drop the action *Take photo* on the map of its Trip page, in order to create the trips composing the circular area, as shown in figure 6.2. The figure 6.2 represents the trips of one of the three Missions. Each one of the three missions has its own map where the user distribute the trips to perform. Then, when the missions start, the drones will take photos over these spots and, in case of bloom, new Trips will be created to Pollinate the area.

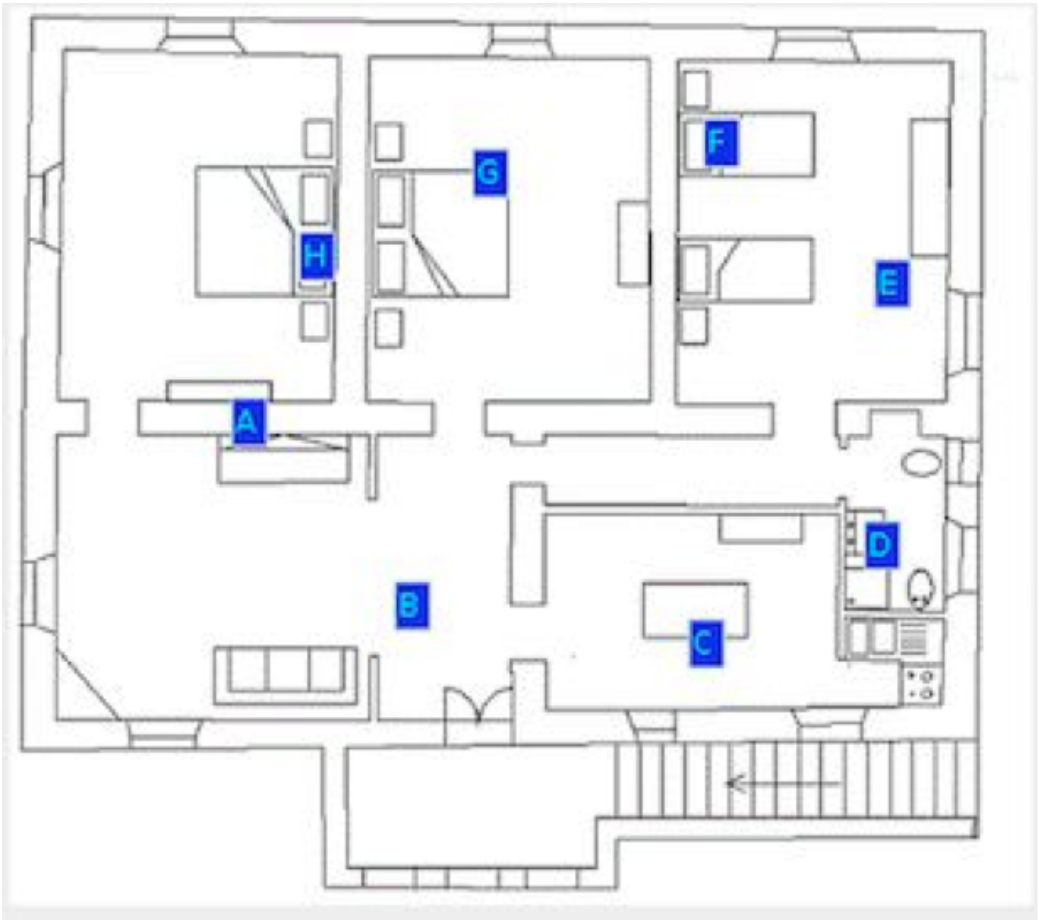

Figure 6.2: The circular area to monitor

To describe in a detailed way the execution flow, we now show the sequence diagrams of the Pluto Main Application behavior:

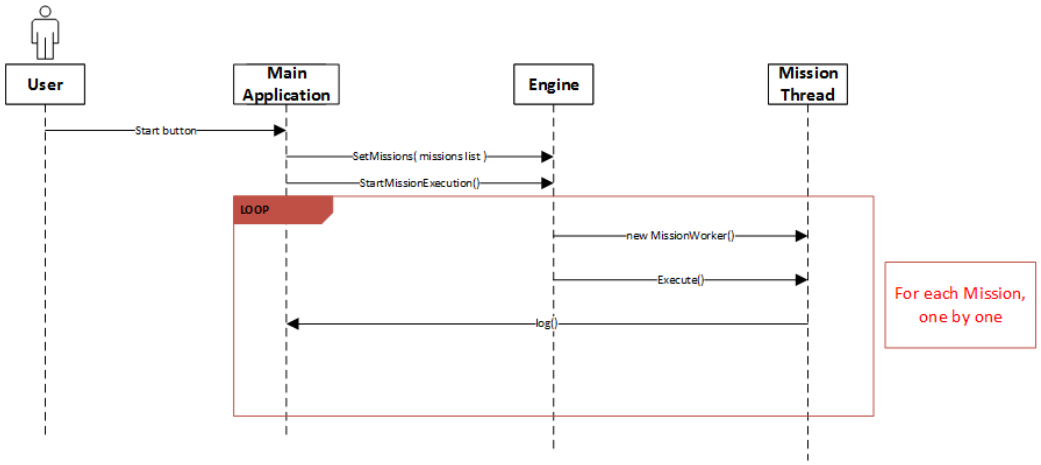

Figure 6.3: Sequence diagram of a starting mission

In figure 6.3 we show the first calls after the user clicks on the Start button in the Monitor Page. The Main Application receives the start command from the user, then activates the Engine entity. Now, a new Thread is created and started for each missions. The status of the Missions are sent to the Pluto Main Application and showed to the final user, through the *log()* function.

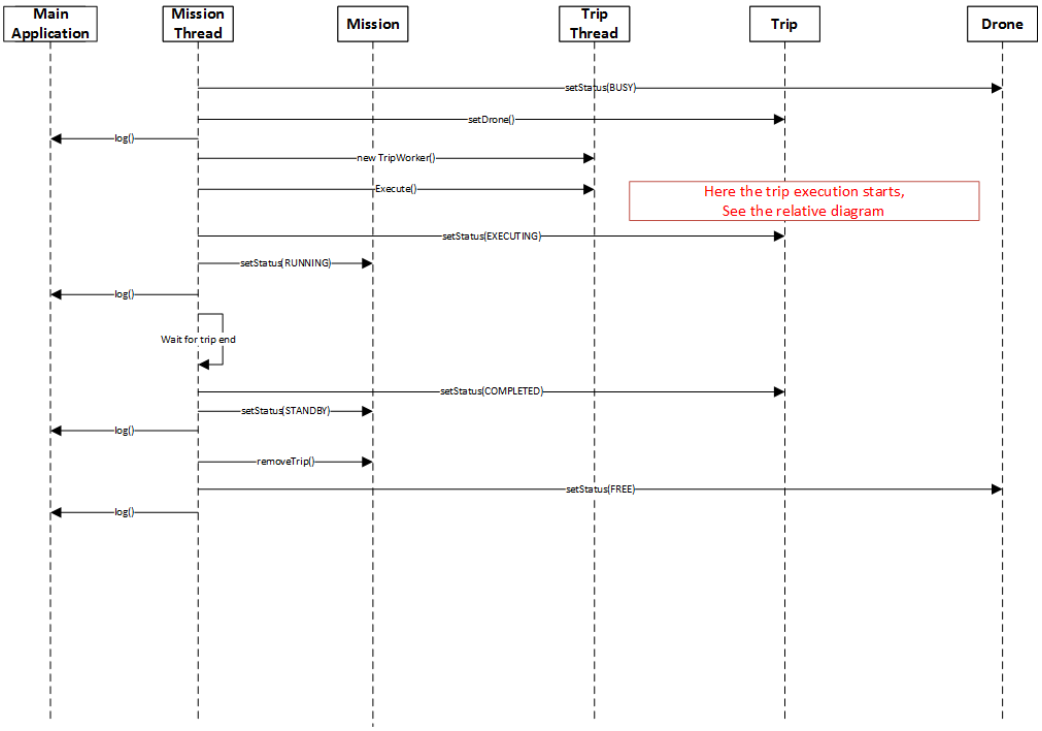

Figure 6.4: Sequence diagram of the mission execution flow

In figure 6.4 the Mission Execution is shown. The *Mission Thread* entity manages all the diagram blocks logic and the Mission flow between them. For example, the first two method calls belong to the Drone Allocator. For each Trip a new *Trip Thread* instance is created, that will manage the parallel execution of the trips. After that, the mission thread waits for the completion of the started Trip, and then the Mission Status is set to "STANDBY", since there are other trips to be executed in the list. Finally, the completed Trip is removed from the list of trips to be executed and the Drone status is set to "FREE". Otherwise, if the launched Trips was the last one, the Mission status would have been set to COMPLETED.

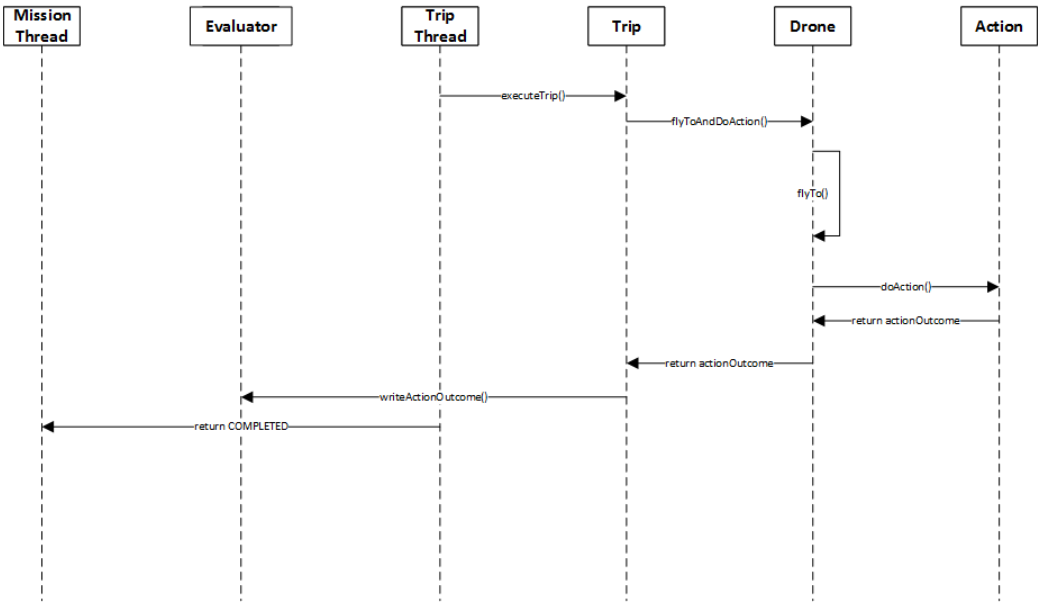

Figure 6.5: Sequence diagram of the trip execution flow

In figure 6.5 the Trip execution is shown. The Drone assigned to the Trip is sent to the established location to take the pictures. After that, the resulting photo is written into the Evaluator entity, where the specific algorithm of the application will perform the evaluation, once the mission will be completed.

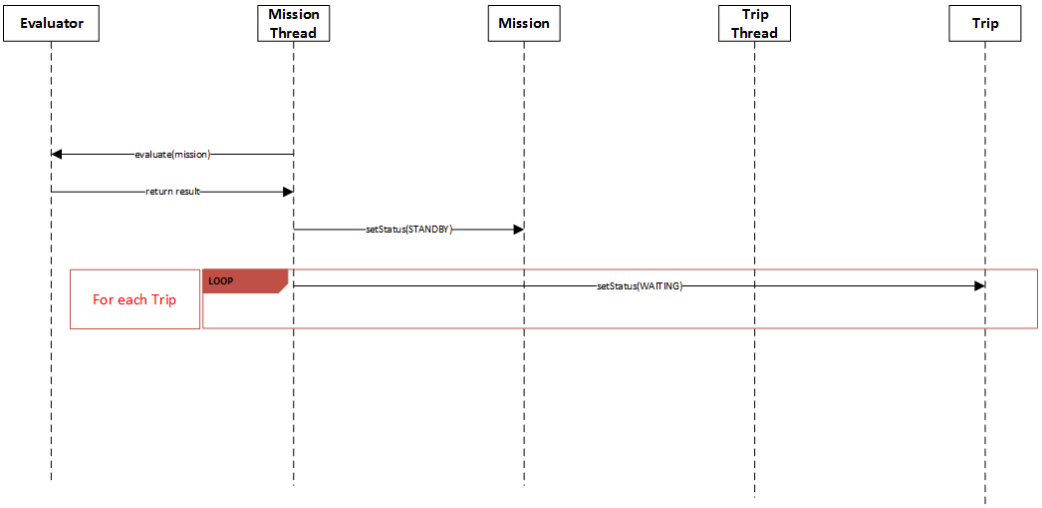

Figure 6.6: Sequence diagram of the mission ending flow

In figure 6.6 the final steps of the Mission execution are shown. The *Evaluator* will check the pictures taken by the Drones, looking for pest, diseases or blooms. Then the Mission status is set to STANBY because of the Mission Repeater logic. The Trips are re-inserted in the execution list, their status is set to WAITING,and the Mission is set to STANDBY because we need to start it again.

6.1.2 Aerial mapping of archaeological sites

This application allows archaeologists to survey ancient sites without involving their direct presence on it. Many Ortophotos of the site are taken, so that the archaeologists can see the geometric layout of the site, without physically walking near it, which could cause irreparable damages. An orthophoto is an aerial photo that is geometrically-corrected so that distances between pixels are proportional to true distances, such that the photo can be used as a map. Drones are sent to take a series of ortophotos which then will be stitched together to derive a single ortophoto; if the individual pictures do not have sufficient overlap, the resulting orthophoto will show excessive aberrations, and, in that case, the drone is sent out again to take more pictures. If the obtained ortophoto is not adequate, the archaeologists should be able to send more drones on that particular area. The drones must perform their actions in a limited amount of time, since if too much time pass between two ortophotos, the scene may change.

The following figure is the Pluto Editor graph needed to create the Aerial Mapping of archaeological sites application.

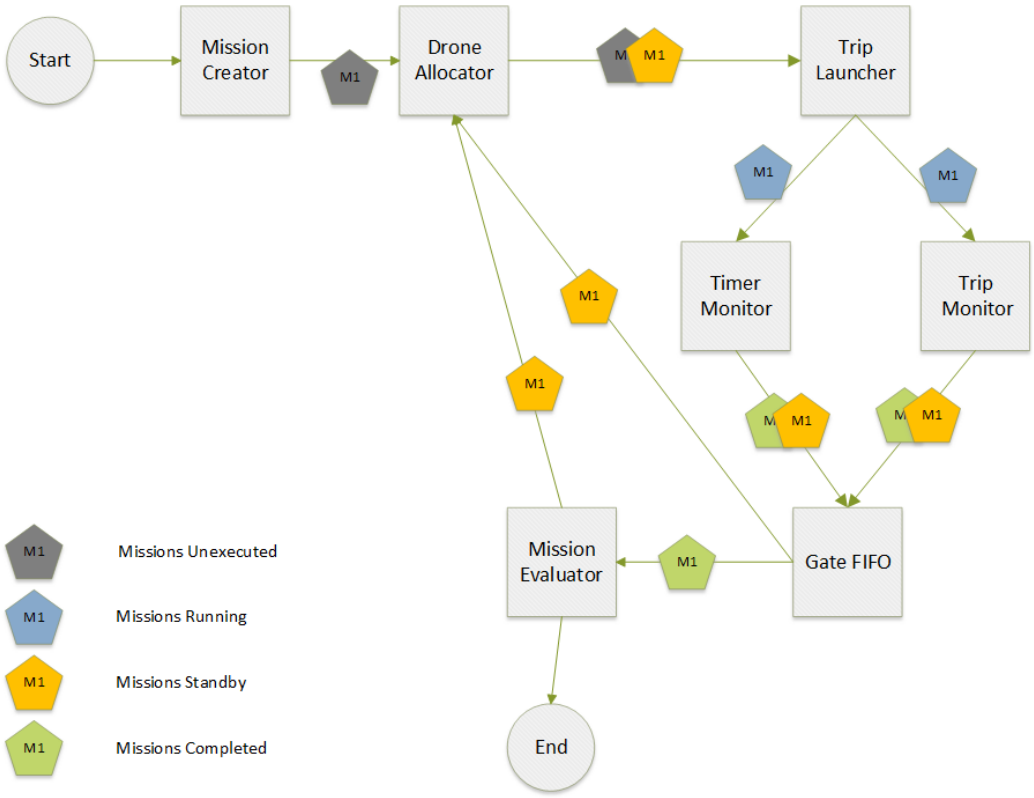

Figure 6.7: Pluto graph for the Aerial Mapping of archaeological sites

The drones can already take pictures thanks to the *Take Photo* action.

To send the drones to take pictures over the site locations, the user has to simply create the Mission entities and add the trips in the Trips Page of each Mission. In case the archaeologists want to send more drones on the locations where they can't obtain adequate ortophotos, they just have to add more Mission entities. For example, if one archaeologist want to send 3 drones simultaneously on a particular location, he has to create 3 Mission entities. Then, in the Trips pages of each Mission, he simply has to drag and drop the *Take photo* action on that particular location.

Thanks to the MissionEvaluator block, that analyzes the drones data at the end of the missions, the Main Application can decide if the photos are good enough or if more drones must be sent out to take new pictures in those locations.

It is important to underline that, using the Pluto framework, it is not possible to obtain the very same behavior of the original application. Indeed, two consecutive photos must be taken within a time constraint. This cannot be fulfilled with Pluto, since the Timer Monitor block deals with a time interval that starts when the drone leaves the base station, ensuring that it will take the picture before the time interval expires. So, there is no way to state a time constraint between two consecutive photos.

The GateFIFO block takes as input two Mission instances, one from the Trip Monitor block and the other one from the Timer Monitor. It takes care of propagating only the first instance that arrives to it. For example, if the timer has expired then it will propagate the Timer Monitor instance, otherwise the Trip Monitor one.

After the GateFIFO block there is a bifurcation: if the Mission is completed, it is sent to the MissionEvaluator, otherwise there are some trips of the mission that must be executed again, and so the Mission is sent to the DroneAllocator. The full explanation of the functionality of each block can be found in section 4.2.2.

Concerning the code of the Evaluator block needed for the development of the Aerial Mapping[7] application, as for the previous application, we can find each photo taken during the missions in the "dataMap" parameter, that is an hashmap that create a relation between a Trip and the photo taken through its Action. First of all, the ortophotos are stitched together to obtain the final ortophoto, trough the *stitch* function. If the final ortophoto shows excessive aberration, the ortophotos composing it are analyzed and if they don't have sufficient overlap, few new Trips are created with the same target locations of the photos to be taken again. The Mission status is set to STANDBY because there is at least a new inserted Trip to execute. It is important to underline that this application differs from the Alfalfa[6] one, because the Evaluator algorithm acts in a different way: in Alfalfa[6] application the algorithm takes care of analyzing the photos of a single mission without considering the others; now, instead, the evaluation needs to merge all the photos of every missions to calculate the aberration. The following code snippet shows our implementation of the Evaluator algorithm:

```
1     String result = null;
```

```
    // The "stitch" method takes a collection of photos as input
     // and return an Ortophoto object derived by a proper algorithm
     // based on the passed photos
    OrtoPhoto ortophoto = stitch(dataMap.values());

    if(ortophoto.getAberration() > ABERRATION_THRESHOLD){

      // Iteration on the photos that were used in the stitch method
        // to generate the Ortophoto
      for (Photo photo: ortophoto.getPhotoCollection()){

        // if the overlap of the single photo is not enough
        if (ortophoto.getOverlapOfGivenPhoto(photo) < OVERLAP_THRESHOLD){

          // Loop on all the Trips of every missions
          for (Map.Entry<Trip, Object> entry : dataMap.entrySet()) {

            // take the Photo of the current iteration
            Photo p = (Photo) entry.getValue();

            // When we found the photo that has the low overlap
            if(photo.equals(p)){

              // create a new Trip that will take a new photo
                   // from the same location
              Trip trip = new Trip();
              trip.setName("NewTrip");
              trip.setTargetLocation(entry.getKey().getTargetLocation());
              trip.setAction(Action.TAKE_PHOTO);
              trip.setStatus(Trip.WAITING);

              // add this new trip to the list of trips to be executed
              missionToEvaluate.getTrips().add(trip);

              // set the mission status to STANDBY, since a new trip has been
                   created
              missionToEvaluate.setStatus(Mission.STANDBY);
            }
          }
        }
      }
    }

     // All decisions were been chosen so we end the evaluation
    result = "Success";
    return result;
```

In order to show the real runtime execution of the Aerial Mapping application with Pluto, we now show a possible scenario: there are 7 drones and 1 big archaeological site to monitor, and we want to send all the drones in that area at the same

time to take the ortophotos. As usual, the user has to create 7 Mission entities. Then, in each Mission's Trips Page, he has to drag and drop the action *Take photo* on the locations forming the site, shown in figure 6.8. Once again, the figure shows the map on the Trips Page of one of the seven missions.

Figure 6.8: The archaeological site on the map

The sequence diagrams related to this application are the same shown in section 6.1.1. The only difference is that now the application doesn't require the repetition of the missions. This leads the Main Application to end the execution flow when the Mission reaches a successful evaluation. As shown in figure 6.9, after the evaluation the execution doesn't go further.

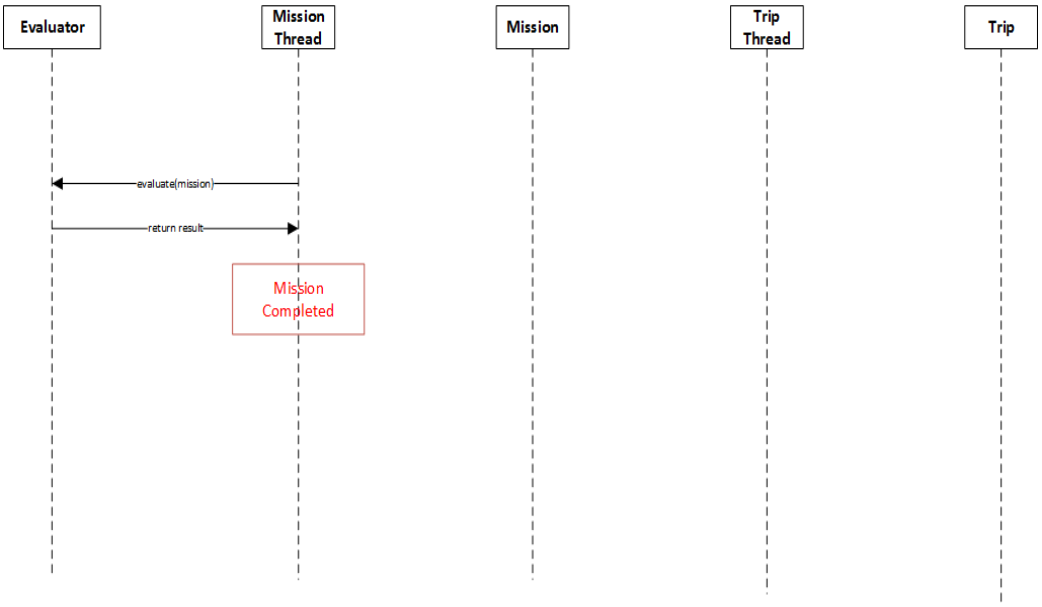

Figure 6.9: Sequence diagram of the mission ending flow, without repetition

6.1.3 PM10

The PM10[8] application is used to build 3D maps of pollution concentration in the atmosphere. Initially, there is a predefined 3D grid over which drones are sent to sample the quantity of pollution. So the drones build a spatial profile of pollution concentration and compute gradients among the areas of higher concentration. Finally the drones are sent along this gradients to sample the pollution concentration, in order to improve the spatial profile representation. Any two consecutive samples must be gathered within a given time bound, otherwise the system will take care of speeding up the execution.

The following figure is the Pluto editor graph needed to create the PM10 application:

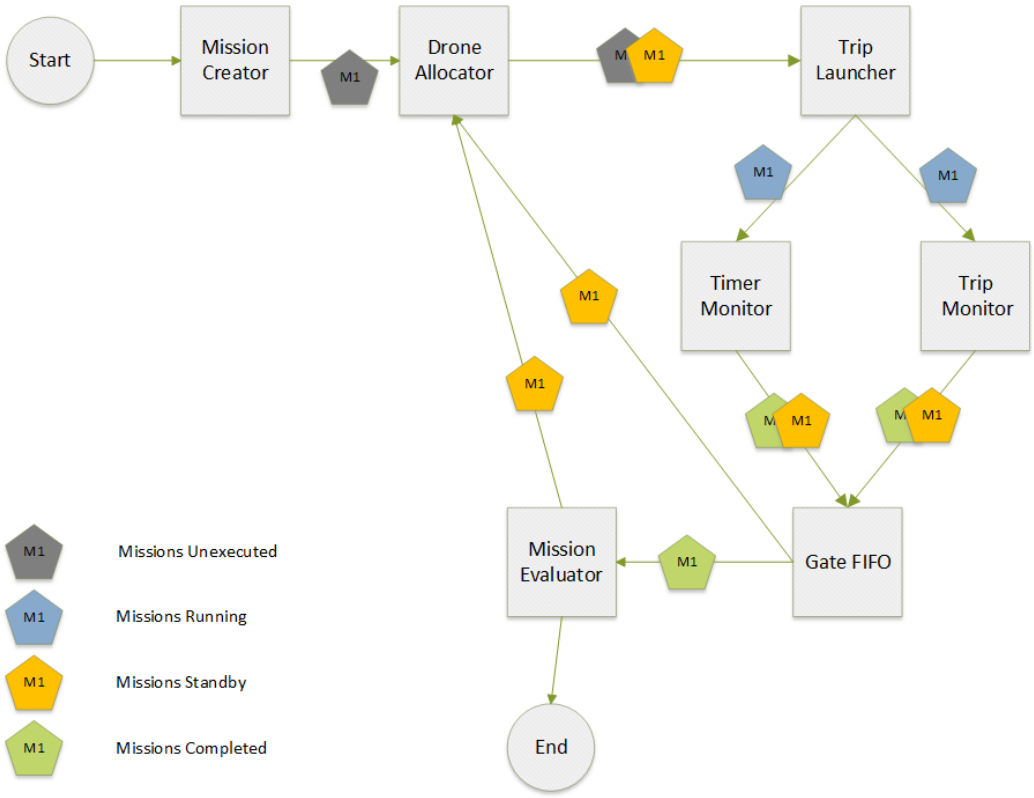

Figure 6.10: Pluto graph for the PM10 application

For the measurements of pollution quantity, the *Measure* action can be used.

The spatial grid must be manually built by the user, organizing the Trips of each Mission on the map he's provided with.

The data collected by the drones are not photos anymore, but a *pollutionQuantity* value which indicates the percentage of pollution in that area.

Then, thanks to the Mission Evaluator blocks, the *pollutionQuantity* variables are confronted and the gradients between areas of higher concentration are computed. So new drones will be sent along this gradients, improving the spatial profile.

As for the Aerial Mapping application, shown in section 6.1.2, Pluto cannot fulfill the time constraint between two consecutive pollution samples. As already explained, the timer of Pluto starts when the drone leaves the base station and

ensures that it will perform the action within that time interval, but there is no way to constrain the time between two consecutive samples.

The GateFIFO block takes as input two Mission instances, one from the Trip Monitor block and the other one from the Timer Monitor. It takes care of propagating only the first instance that arrives to it. For example, if the timer has expired then it will propagate the Timer Monitor instance, otherwise the Trip Monitor one.

After the GateFIFO block there is a bifurcation: if the Mission is completed, it is sent to the MissionEvaluator, otherwise there are some trips of the mission that must be executed again, and so the Mission is sent to the DroneAllocator. The full explanation of the functionality of each block can be found in section 4.2.2.

Below there is our implementation of a possible Evaluator algorithm:

```
1     String result = null;
2
3     // building a new map with only the trip-measure couples of the mission
4     // to evaluate
5     Map<Trip, Integer> missionMap = new Map<Trip, Integer>();
6     for (Map.Entry<Trip, Object> entry : dataMap.entrySet()) {
7       if (missionToEvaluate.getCompletedTrips().contains(entry.getKey())) {
8         missionMap.put(entry.getKey(), (Integer) entry.getValue());
9       }
10    }
11
12    // this method use the Trip location and the pollution measure to
13    // calculate
14    // the gradients and then return a list of String that indicates
15    // the positions of these gradients
16    List<String> gradientsPositions = calculateGradients(missionMap);
17
18    for (String position : gradientsPositions) {
19
20      // create a new Trip to calculate pollution at the gradient position
21      Trip trip = new Trip();
22      trip.setName("GradientTrip");
23      trip.setTargetLocation(position);
24      trip.setAction(Action.MEASURE);
25      trip.setStatus(Trip.WAITING);
26
27      // add this new trip to the list of trips to be executed
28      missionToEvaluate.getTrips().add(trip);
29      // set the mission status to STANDBY, since there are new trips to perform
```

```
30          missionToEvaluate.setStatus(Mission.STANDBY);
31
32      }
33
34      // set the result of the evaluation
35      result = "Success";
36      return result;
```

Now we show the execution of the PM10 application with Pluto in a particular scenario: we have 5 drones and we want to measure the pollution quantity in the area shown in figure 6.11 using all of them. As usual, the user has to create 5 Missions and has to choose the locations of the area to sample through the map on the Trips Page of each Mission. In this way, five drones will monitor simultaneously the area to sample.

Figure 6.11: The archaeological site on the map

As for the Aerial Mapping[7] application, the repetition of missions is not required. So the sequence diagrams don't change and the reader can see them in sections 6.1.1 and 6.1.2.

6.1.4 PURSUE

The PURSUE application[9] is representative of surveillance applications. A team of drones monitor an area and they have to follow moving objects which pass through, taking a picture of each one of them when they enter in the camera field. To do so, drones can operate in two distinct modes: when in "patrolling mode" they simply inspect an area, while when an object is found they switch to "pursuing mode" and start to follow the object. Since an object could move faster than the drones, no drone can follow it constantly, the system must take care of switching between the real drones in order to constantly follow the target. There are time constraints to respect between the detection of a moving object and when its picture is taken and, in case of violations,every tracked object with at least one acquired picture is released from tracking, to regain the drone resources and lower the acquisition latency for the next object.

The PURSUE application represents a limit for the Pluto programming framework. Indeed, in our model, the drones perform their action only at the end of the Trip, so it's not possible for them to actively take a picture in the very same moment the moving object enters in the camera field. This problem can be lowered by inserting a lot of trips in the area to monitor, with strict time constraints on them, in order to obtain a lot of pictures of the monitored area. But in this way, not only it's not sure to capture the moving object, but there will be a lot of useless empty pictures. And, above all, there is still no way for the drones to actively follow the moving objects.

We can conclude that the PURSUE application is too "dynamic" for the Pluto programming framework, and this can be an hint for a future expansion of our work.

The applications described above were already developed and tested with other systems, like Karma[2] and Voltron[5], but we also developed new applications and tested our framework on them: the Object-finder (OF), the Warehouse item-finder (WIF) and the Drugs distribution(DD).

The three applications are modeled by the same Pluto Editor graph, shown in figure 6.12:

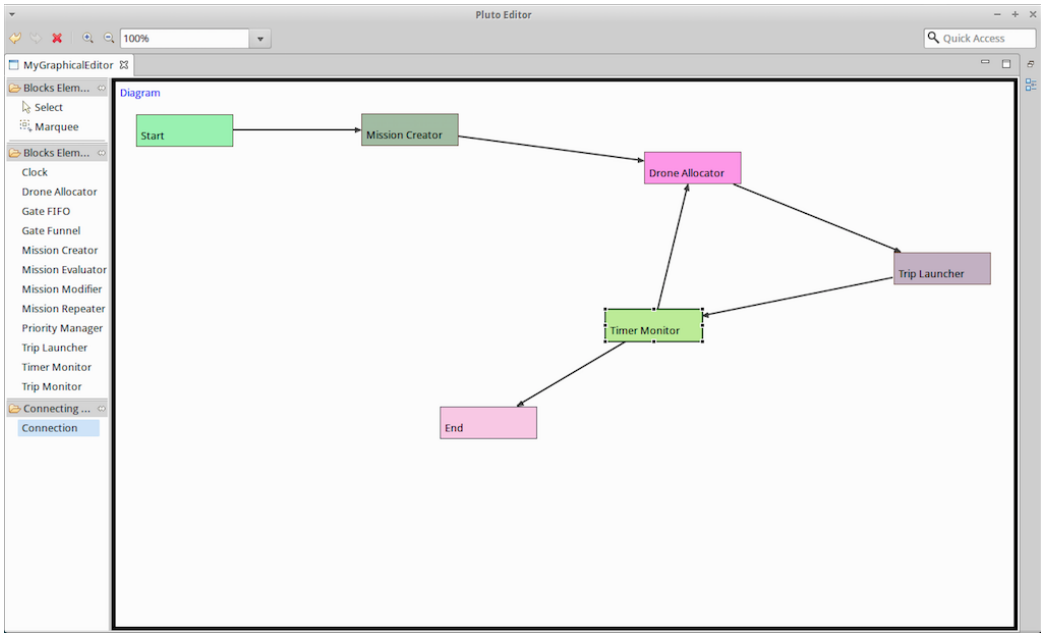

Figure 6.12: The Pluto graph of the OF,WIF and DD applications

In the following sections we describe these applications in details, also using a visual representation of their behavior, and a sequence diagram to make understand better the whole functioning of each one of them.

6.1.5 Object-finder (OF)

This application help users to find various objects, like shoes, keys, books, in a domestic fashion:

- the user decides which item wants the drones to look for and the area to be inspected
- the main system organizes the team of drones, sending them on the specified locations
- the drones fly to the assigned location and, if found, bring the objects back to the user

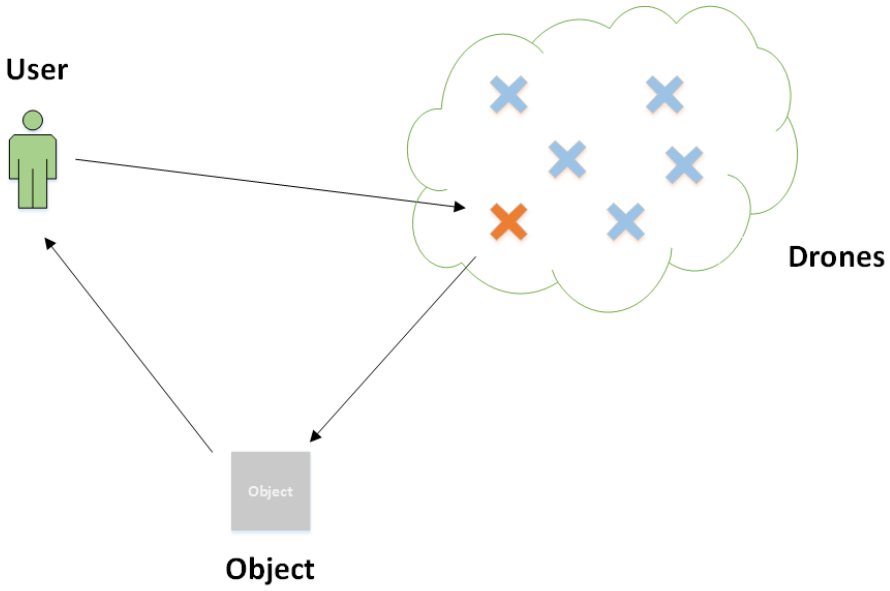

Figure 6.13: The basic functioning of the Object-finder application

6.1.6 Warehouse item-finder (WIF)

This application help users to manage a warehouse, bringing a list of objects to the them:

- the user makes a list of needed items and writes it on his laptop, tablet or smartphone
- the main system organizes the drones and decide which one will take each item in the list
- the drones fly to the assigned objects and bring them back to the user

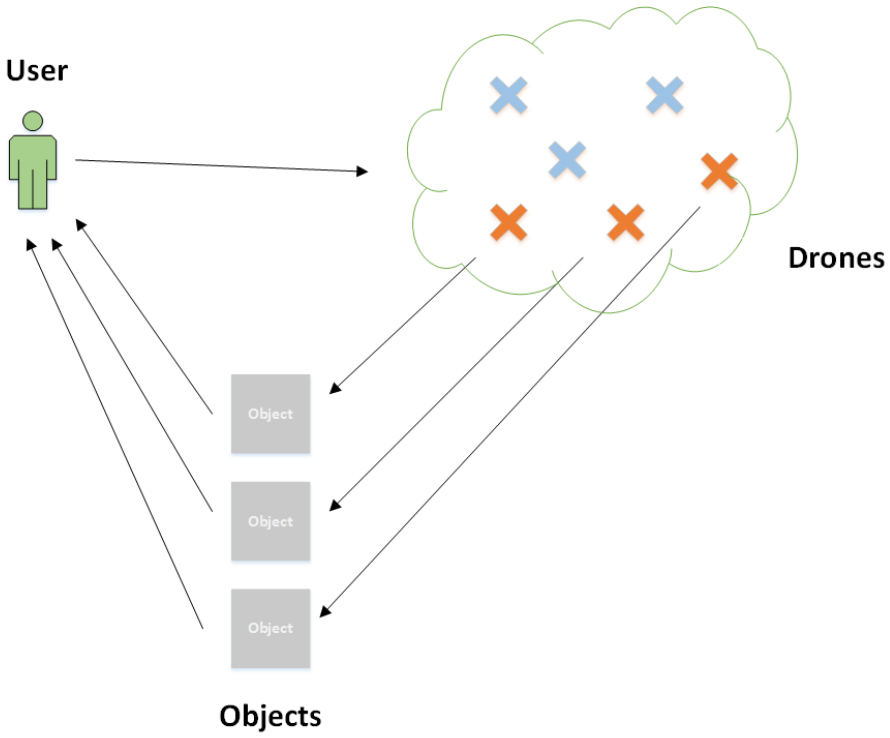

Figure 6.14: The basic functioning of the Warehouse item-finder application

6.1.7 Drugs distribution (DD)

This application help nurses in assisting elder people to take their daily medicines, in an hospice context:

- the nurses prepare some little boxes with each patient's daily medicine
- each drone, at the right time of the day, brings the box to its assigned patient
- after carrying out their action, the drones return to the start location

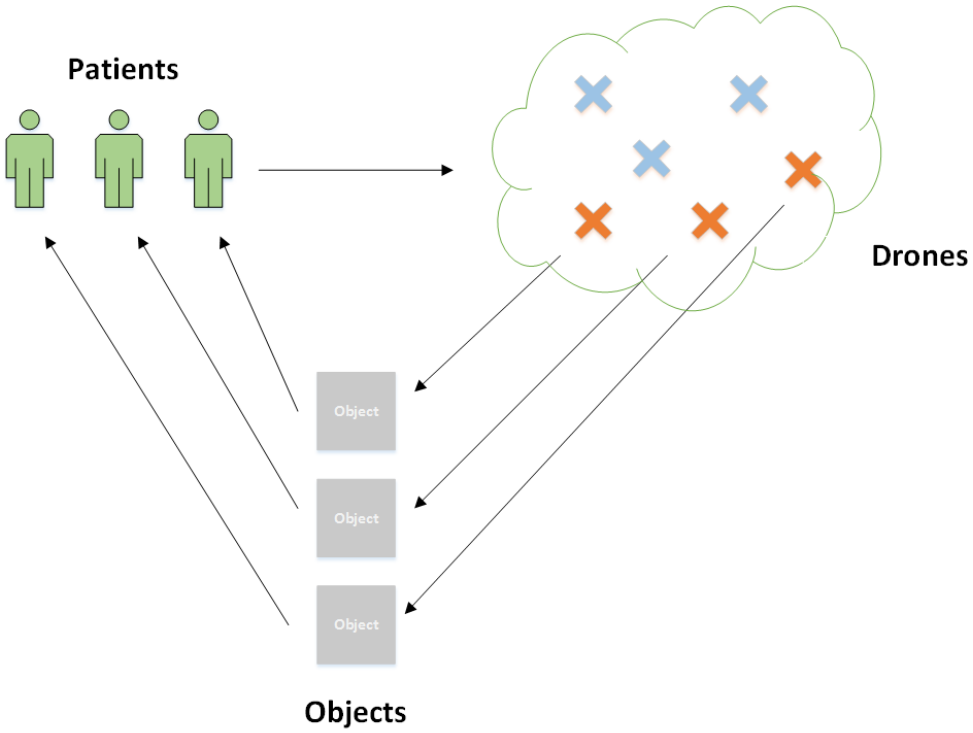

Figure 6.15: The basic functioning of the Drugs distribution application

Here there is a sequence diagram showing the behavior of the three applications:

6.2 Usability of the model

To evaluate the concrete usability of the Pluto programming framework, we decided to test it on real people, proposing them two "exercises". We involved five testers, recruited both in the "Politecnico di Milano" and "SICS Swedish ICT" environment, in order to guarantee a solid development background and to avoid possible lack of programming knowledge. We created one exercise for each component of Pluto framework. The first one consists in the development of an application using the Pluto Graphical Editor. The exercise is split in three levels, starting from a very basic version and going through more difficult versions. Each version asks the user to add a new functionality by using the available components in the Editor. The second exercise, instead, asks the user to use the generated code from the previous exercise to run the Pluto Main Application, then asks to create some missions and, in the end, to run them. The application we choose for the exercises is the Drugs Distribution, already described in section 6.1.7, because it's very suitable for the type of evaluation we want to perform. Indeed its basic version can be extended with many features, for example using the Mission Modifier block (section 4.2.2), and this is exactly our purpose. The results are shown in section 6.2.5. After the execution of the exercises, we asked the users to leave a feedback, proposing them a survey, built according to some metrics that we have defined and that we describe in section 6.2.4.

6.2.1 Proposed exercises

We give the user a complete and sound explanation of the Pluto programming framework, showing how the graphical editor works (shown in section 4.1.1) and giving him the list of the available entities (shown in section 4.2.2) and of the implemented blocks (shown in section 4.2.2), together with an explanation of the meaning and functionality of each one. The same explanations are given for all the components of the Pluto User Application (shown in section 4.1.2).

First Exercise

The exercise proposes the development of the Drugs Distribution application (shown in section 6.1.7) in three different versions, increasingly harder to implement:

- *basic version*: we ask the user to implement the basic version of the Drugs Distribution application
- *medium version*: we ask the user to raise the priority of the failed trips and to re-insert them in the queue of next trips to be launched, and to add a delay for the trips.
- *hard version*: we ask the user to add a time constraint within each trip must be completed, that is the same feature implemented by the Timer Monitor block, but using the Mission Modifier block.

To solve the first part of the exercise, the user has to create the graph shown in figure 6.16, that represents the very basic scheme of each application, since it uses only the basic blocks. Once created the graph, the user has to right click on the panel and choose "generate code" to accomplish the first step of the exercise. This is a very easy task to perform, but we think it's useful, because make the user confident with the basic features of Pluto, such as the basic blocks and the code generation mechanism.

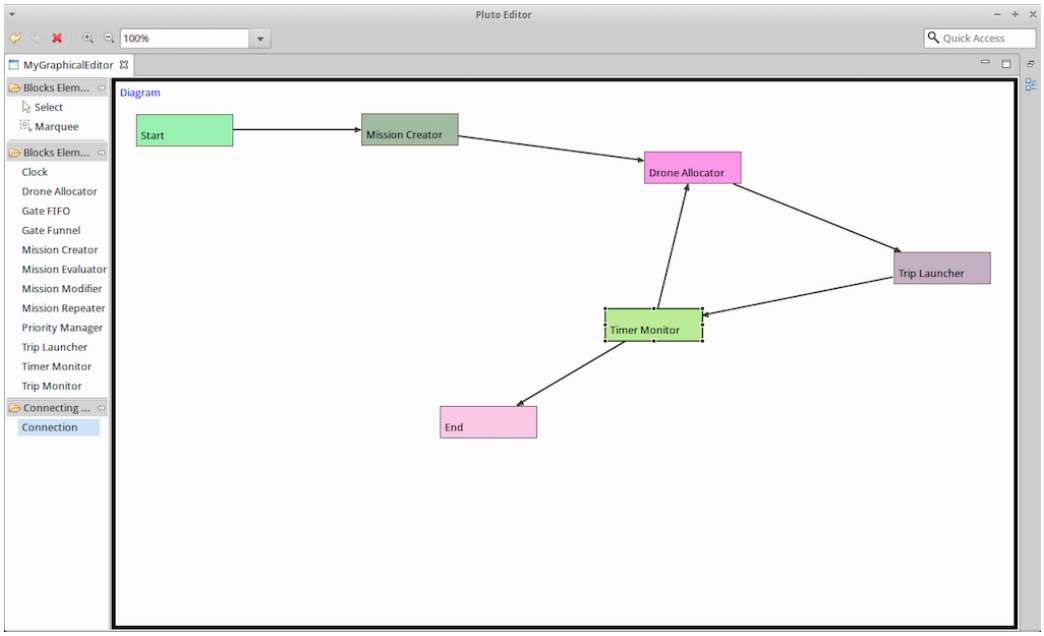

Figure 6.16: Solution of the first step

To solve the first part of the second step of the exercise, the user has only to understand that the functionality to add is already implemented by the Priority Manager block, so he has only to add this block to the graph in the right point. Since we ask him to re-insert the failed trips in the queue of unexecuted trips he has to put the block between the Trip Monitor and the Drone Allocator, as shown in figure 6.17.

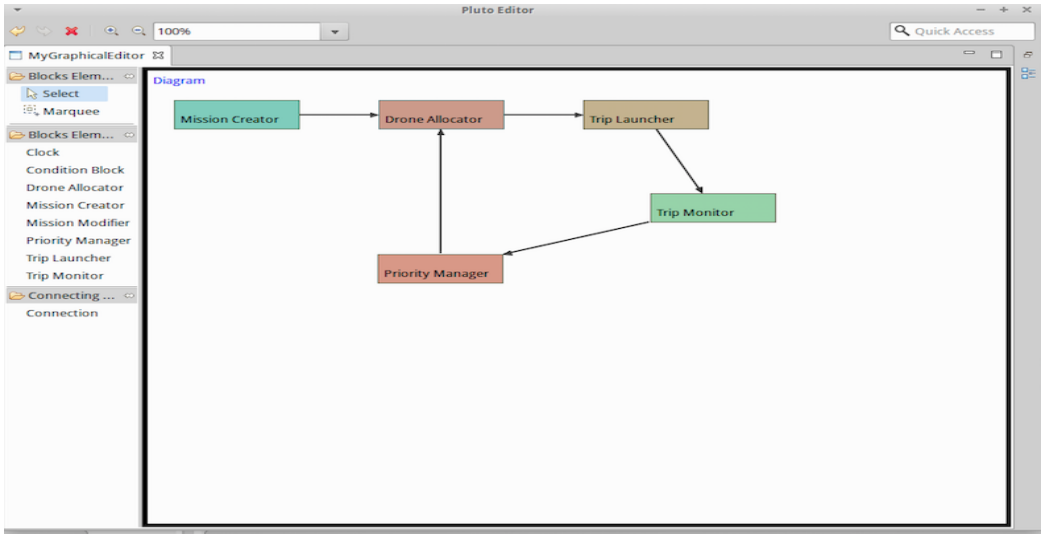

Figure 6.17: Solution of the second step with Priority Manager

Then to add the Delay feature in the diagram the user needs to do the same thing with the Clock block, that provides the feature to wait for an amount of time, set in the delay attribute of a Trip. This block is taking as input the mission provided by the Trip Monitor and the by the Mission Creator then, after the delay time has passed, gives the mission to the DroneAllocator block, as shown in figure 6.18.

This is a useful step to compute, because the user learns how to use the connection element, that is a very important feature in the Pluto framework, and also the Priority Manager and Clock blocks.

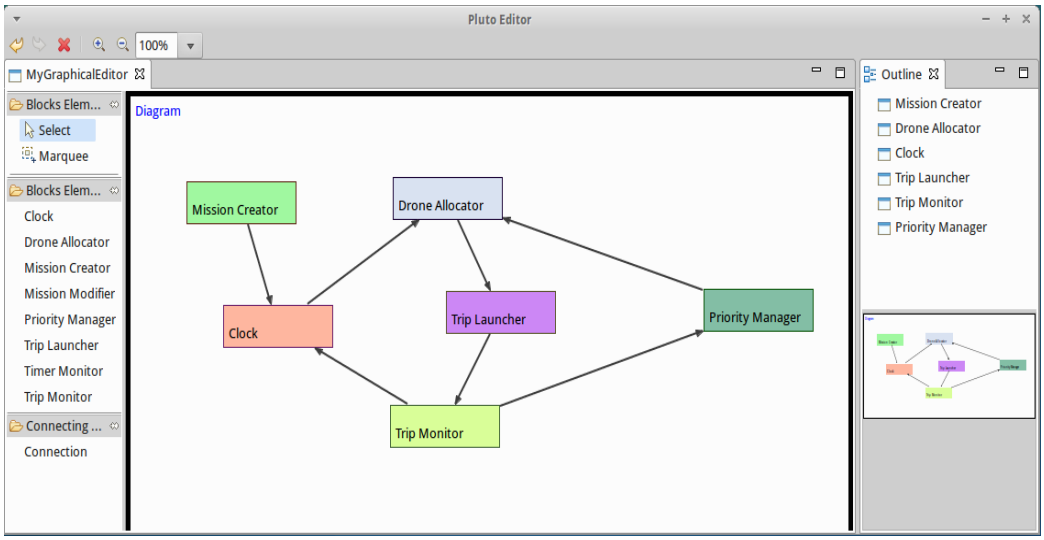

Figure 6.18: Solution of the second step with Clock block

For the third step, the user has to implement the feature of the Timer Monitor block without using it. So, he has to use the Mission Modifier block, through which he can insert his code in the application, and put it between the Trip Launcher and GateFIFO blocks, in parallel with the TripMonitor, as shown in figure 6.19. This is a very useful step to compute, because the user learns how to use Mission Modifier block, which is an important feature of the Pluto Editor, because it allows the programmer to insert his custom code to characterize the application.

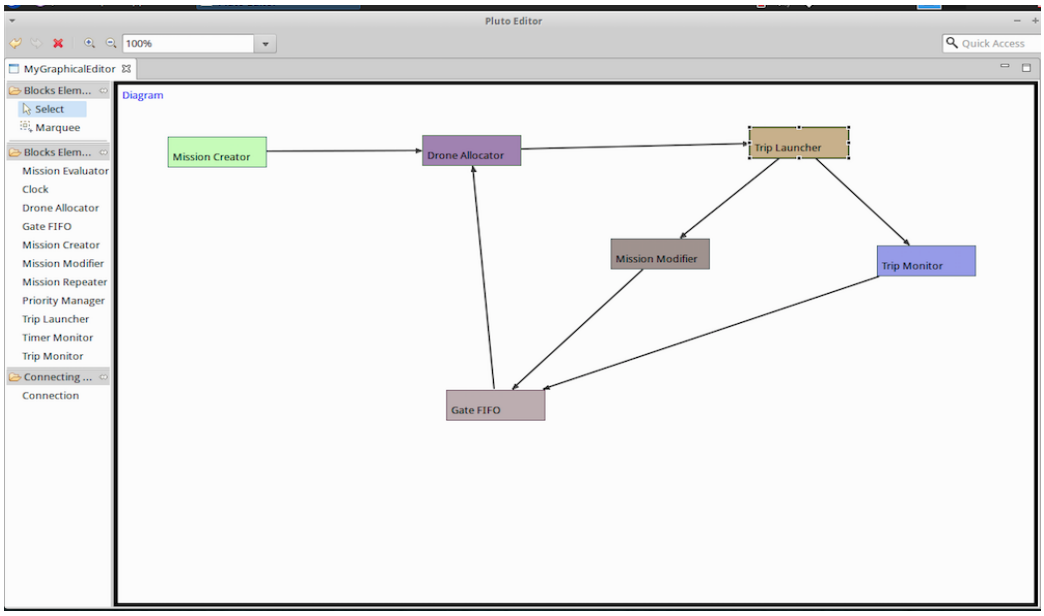

Figure 6.19: Solution of the third step

Second Exercise

The main purpose of this exercise is to underline possible issues in the code generated with the Pluto Graphical Editor. We must be sure that all the functionality described by the diagram are enabled in the generated code and that the missions execution will run smooth as the user expects. Any lack in the user experience may compromise the usability of the entire application, so it is important to evaluate the User Interface too. In this way, we check if the visual disposition of the graphics elements is appropriate. The first step asks every user to create and run the same kind of missions, as described in the following list:

- Mission 1

  Trip A -> Action: Take Photo

  Trip B -> Action: Take Photo

  Trip C -> Action: Take Photo

- Mission 2

  Trip A -> Action: Measure

  Trip B -> Action: Measure

- Mission 3

  Trip A -> Action: Pick Item

  Trip B -> Action: Release Item

  Trip C -> Action: Take Photo

  Trip D -> Action: Measure

The location in the map of the Trips was not important while testing, so the tester could decide any place. In the second step the users were asked to open the Monitor Page and to start the missions following their execution using the provided table and console. The third step of the exercise consists in calling back the Drone with an RTL command.

6.2.2 Evaluation metrics

To concretely evaluate the usability of the Pluto programming framework we defined the following metrics, which we applied for both exercises:

- Number of people who correctly solved the first part of the exercise
- Number of people who correctly solved the first and second parts of the exercise
- Number of people who correctly solved the whole exercise
- Mean time for the resolution of the first part of the exercise
- Mean time for the resolution of the second part of the exercise
- Mean time for the resolution of the third part of the exercise
- Mean time for the resolution of the whole exercise
- Number of people who solved the whole exercise, but in a wrong way
- Number of people who could not solve the exercise at all

Through metrics 1,2 and 3 we can understand which parts of the exercises are not clear for the user and/or too difficult to implement. Through metrics 4,5,6 and 7 we can understand, once the user has understood how to implement each feature, how much is difficult to solve each part of the exercises by measuring the time required to solve each step. Through metrics 8 and 9, finally, we can understand how easy is to confuse the specifications and how many people couldn't solve any step of the exercises.

6.2.3 Baseline

We want to demonstrate the effective usefulness of the Pluto programming framework, so we decide to compare its features with the API of the Crazyflie Nano-quadcopter, which was described in section 5.6.

Actually, the crazyflie is the drone we chose to use for our applications case study, and we want to demonstrate that, without Pluto and using only the Crazyflie API would be more difficult to build the same kind of applications.

So we decide to propose another exercise to our users, but this time they can use only the Crazyflie API and they have a limited amount of time.

The Crazyflie API is written in Python, so we address to people who knows Python language features.

The exercise consists in make the drone moving from a point A to a point B on a map, performing a single Trip.

It may seem easy, but it can can take a long time to fully understand and apply the API in the correct way.

6.2.4 User Survey

Since we want to evaluate the usability of Pluto, we propose a survey to the users, in order to understand how easy it is to use and which modifications should be applied to improve the user experience.

We ask users to tell us how easy was the development of the various steps of the exercises, and to provide us with a feedback on the usability of the editor and the main application underlining any problems found. We also ask for suggestions to improve the usability of Pluto.

The survey can be found following this link:

https://docs.google.com/forms/d/1b_52e7VLuns6AH1jiT3TeIRZ_KPRTLJYJe9ckrJanWY/viewform?usp=send_form

Actually, this survey gives us very useful information about the Pluto framework. We can understand how "usable" it is and which modifications should be performed to improve the user's experience, also thanks to the visualization of the answers in a graphical way, shown in the next section, the 6.2.5. Through the questions on the exercises development we can understand how difficult it is to create, modify,customize and execute a particular application, validating "on field" the use of the various blocks, especially the Mission Modifier, and the usability of the user interface.

6.2.5 Results

Thanks to the combination of the answers to the user survey of section 6.2.4 and the numeric data collected according to the metrics defined in section 6.2.2, in this section we show the results of the Pluto evaluation. We make use of graphical representation in order to make the results clearer and easily understandable by everyone. The metrics data are put into the table 6.1, while the results of the user survey are presented with graphics.

Table 6.1: The values of the metrics for the two exercises

Metric	1	2	3	4	5	6	7	8	9
Exercise 1	5	5	5	10 mins	10 mins	20 mins	40 mins	0	0
Exercise 2	5	5	5	5 mins	5 mins	5 mins	15 mins	0	0

A graphical representation of the answers to the user survey can be found at the following link:

https://docs.google.com/forms/d/1b_52e7VLuns6AH1jiT3TeIRZ_KPRTLJYJe9ckrJanWY/viewanalytics

Examining both the user survey and the results of the evaluation metrics, we can say that Pluto is quite easy to use. Indeed, all the five testers managed to solve the two exercises, and, through the survey, they stated that Pluto is easily usable and its functioning is quite fast to understand. They also give some suggestions, especially on the Pluto Main Application, which made us improve some features that appeared tricky or not easily understandable.

6.3 Performance evaluation

In order to strengthen the evaluation of Pluto, after the user study, described in Section 6.2, we evaluated some qualitative metrics. These metrics are divided in two main types: software metrics and hardware consumption metrics. The former let us know the complexity of our software. On the other hand, the latter are useful to underline possible issues at run-time, such as thread deadlock or a too high memory consumption. Since our framework is composed by two main components, we decided to split this evaluation in two parts: this means that each kind of evaluation was performed on the Pluto Graphical Editor first and then on the Pluto Main Application. To measure these parameters we used a very useful tool called VisualVM, shown in figure 6.20.

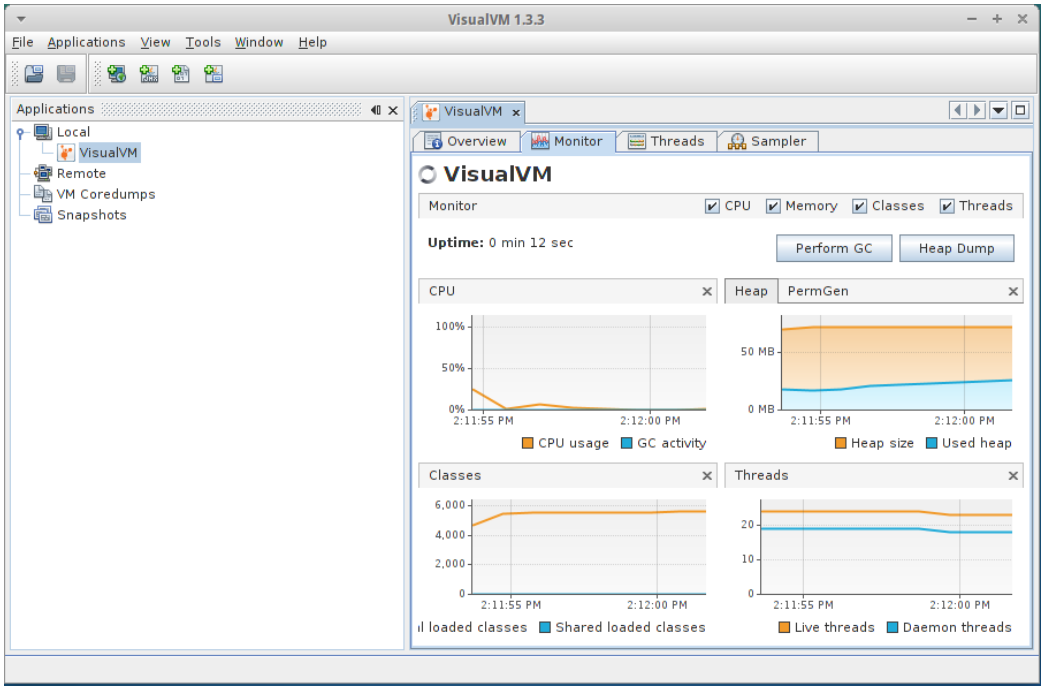

Figure 6.20: VisualVM interface

It let the user have a global monitoring of the running Java application in your local Java Virtual Machine, at run-time. Furthermore, it has a useful feature that records the profiling of an application in a dump file so that the user can compare different dump files concerning different application sessions. Finally, in the Result section 6.3.3, we describe the outcome of the tests for the Pluto Graphical Editor

and the Pluto Main Application separately.

6.3.1 Software Metrics

The software metrics let us understand the complexity of the software. We decided to record these parameters for each component of the Pluto Framework:

- Total Lines Of Code (LOC)
- Number of attributes
- Weighted methods per class
- Number of classes
- Number of methods

6.3.2 Hardware Consumption

The hardware metrics are those parameters measured at run-time, during the execution of the two software, that check if they generate any performance issues, because they could require too many resources. These metrics are:

- CPU Load
- Memory Consumption
- Live Threads

The profiling of the application was done on a machine with these specifications:

- CPU: Intel i7 2640
- RAM: 4GB
- VGA: Nvidia GeForce 610M
- SSD: Kingston 120GB
- OS: Xubuntu 14.04

- Java: JDK 1.7

Concerning the Pluto Graphical Editor, we measured these parameters while creating a lot of blocks and a very complex net of connections, shown in figure 6.21 and then we observed the stress level while generating the source code of the Main Application from that drawing. We concentrated the evaluation on two parameters: the number of blocks and the number of connections. At first we fixed the former and we incremented the latter by a step of 5. Then we did the same operation fixing the number of connections and raising the number of blocks.

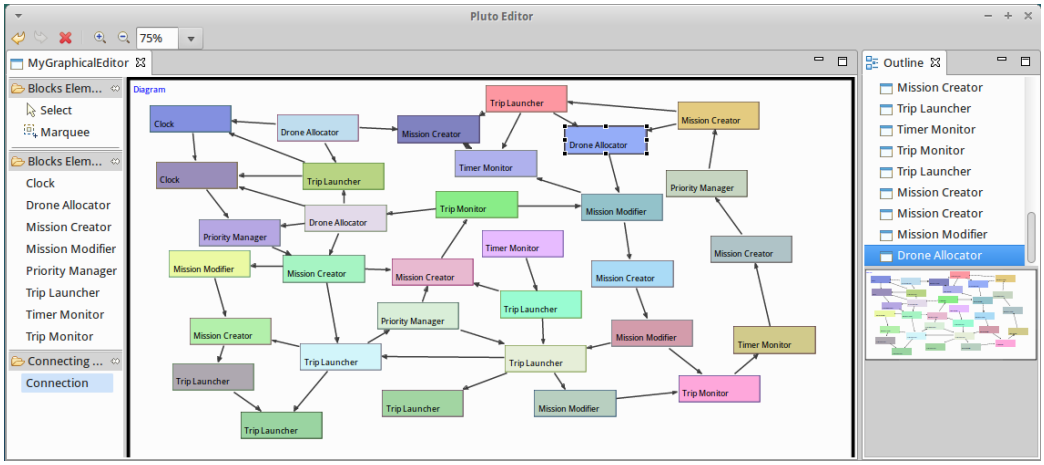

Figure 6.21: Very Complex Diagram Example

Furthermore, we evaluated the same metrics concerning the Main Application. We decided to focus this evaluation varying 3 important parameters: the number of Mission, the number of Trips related to a single Mission and the number of available Drones. We started fixing two of these parameters and raising the third step by step. Then we did again this procedure fixing another couple of parameters and varying the third one. In this way, we could evaluate the performance of the Main Application in an accurate way. The results are shown in next section 6.3.3.

6.3.3 Results

In this section we show the software and hardware evaluation results, achieved with the methodologies described in previous section 6.3. First of all, we evaluated the software complexity of the Pluto framework, according to some important software metrics. The table 6.2 proposes the results.

	Main Application	Graphical Editor
Total lines of code	2132	5072
Number of classes	48	129
Number of attributes	104	141
Number of methods	197	569
Weighted methods per class	325	848

Table 6.2: The metrics concerning the two Pluto components

Thanks to these measures, we can make some considerations about the complexity of the Pluto framework. Considering all the data, we reached to maintain a light code.
Concerning the Main Application, two thousand lines of code are an average amount for a not complex project. The same applies to the other values too.
Instead, the Graphical Editor has a higher complexity, with all the values doubled. This can be explained by the fact that, to develop the Editor, we based our code on Eclispe GEF Framework. This means that most of the classes, have inherited from other parent classes of this framework and we inherited its complexity too. Despite this, 5 thousand lines of code cannot been considered too much, but again they are an average amount for a normal size project.

In the end, we present the results concerning the harware consumption of the two Pluto framework components.
Starting from the Graphical Editor, the diagrams in picture 6.22 describe the hardware consumption when the connections number raise with fixed number of blocks and when increasing the blocks amount with fixed connections.

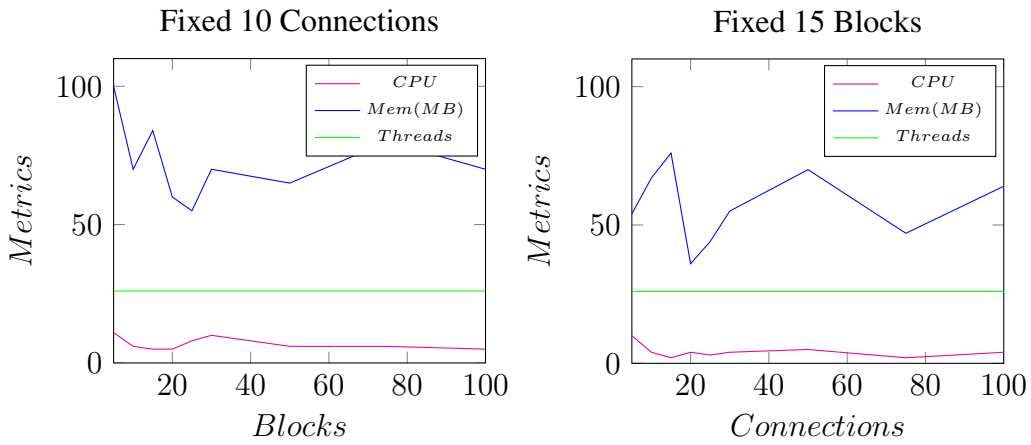

Figure 6.22: Hardware metrics of Graphical Editor

As you can see the Graphical Editor doesn't require a big amount of hardware resource, even with complex diagrams.

The Main Application evaluation gave us different results, shown in the following figures.

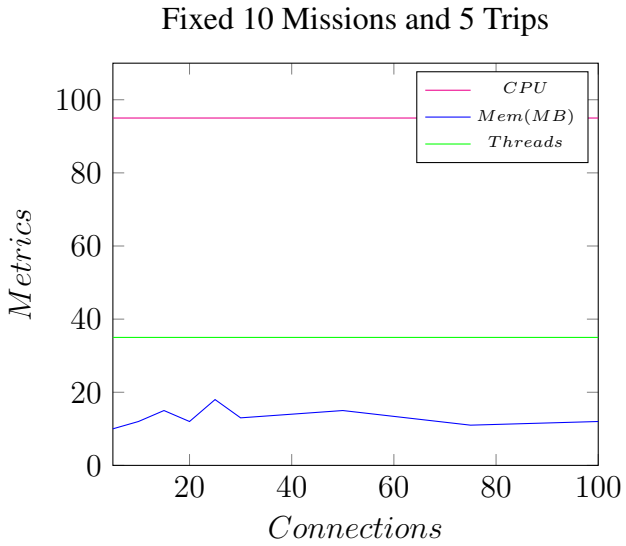

Figure 6.23: Evaluation results of Main Application with fixed missions and trips

The diagram 6.23 describes the hardware consumption of the Main Application

with a fixed number of missions and trips and a raising number of drones. We created 10 missions with 5 trips each.

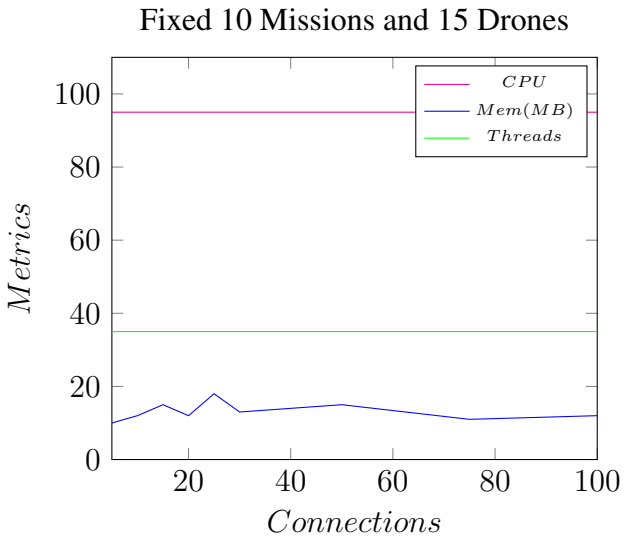

Figure 6.24: Evaluation results of Main Application with fixed missions and drones

The diagram 6.24 describes the hardware consumption of the Main Application with a fixed number of missions and drones and a raising number of trips for each mission. We created 10 missions and made available 15 drones.

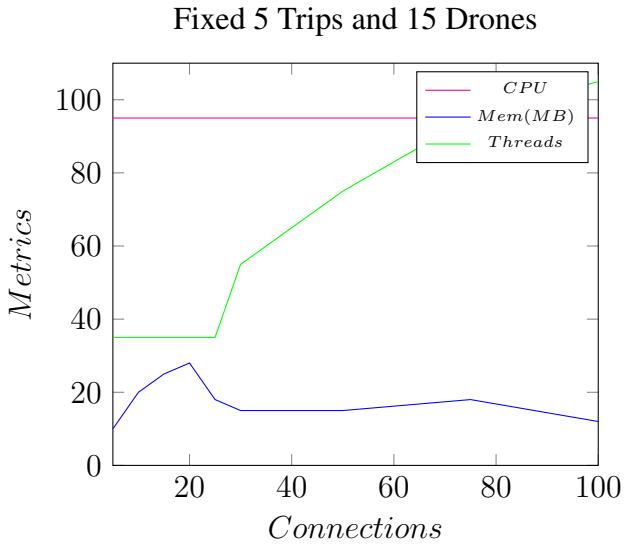

Figure 6.25: Evaluation results of Main Application with fixed trips and drones

The diagram 6.25 describes the hardware consumption of the Main Application with a fixed number of trips and drones and a raising number of missions. We created 5 trips for each mission and made available 15 drones.

As you can see, the general consumption is almost the same for each situation: there is a low request of memory, a limited number of live threads, and a quite high consumption of the CPU.

It's important to put in evidence an unexpected situation regarding the third experiment, the one in which we fixed the number of trips and drones. When we reached a certain number of missions, the system fell in a deadlock situation. For example, whit 15 drones and 5 trips, we had no problems until we reached 25 missions. Then, when we raised the amount of available drones, for example setting it to 20, the system didn't fall in the deadlock situation and all the missions were completed successfully.
This problem put in evidence a possible dependency between the number of missions created and the number of available drones. The causes of this issue could be hardware related, meaning that a more powerful machine could be able to complete all the missions even with a low number of available drones.

Chapter 7

Conclusions and future works

7.1 Conclusions

We have developed the Pluto Programming Framework, a system which allows to build indoor applications by simply graphically connecting blocks.

Each one of these blocks, described in section 4.2, contains the implementation of a specific functionality.

So the programmer can build an application just deciding which features he needs and consequently connecting the right blocks, using the editor shown in section 4.1.1.

The final user decides the sensing tasks that have to be performed by the drones thanks to the user interface, fully described in section 4.1.2.

The system takes care of assigning the drones to each sensing task, greatly simplifying the work of both the programmer and the final user, which has only to decide a list of sensing tasks without dealing with the drones dispatching.

The main innovations with respect to the actual state of the art, fully described in chapter 2, are represented by the indoor context and the graphical editor.

Indeed, there exist systems similar to Pluto, but they all manage outdoor applications, where the GPS can be used for localization and big drones for actuation of the sensing tasks.

Furthermore, the building of applications through connection of blocks really simplifies the development process, still allowing the programmer to insert custom code.

In chapter 6 we fully evaluated Pluto, confronting it with other similar systems,

trying to apply it to existing applications, proposing its use to real people and asking them for feedback through a survey, and finally measuring its performance, through bot hardware and software metrics.

We state that Pluto is a useful programming abstraction, which allows the development of a great variety of applications in a simple and fast way. As any other system, Pluto has its limits, both for its implementation and for technological issues, which we show in the next section.

7.2 Pluto limits and future works

In this section we show the limits of the Pluto programming framework.

There are two type of limits: the limits in the implementation can possibly be overcome by modifying the source code and/or adding new features, or changing the whole model of the system. The technological limits cannot be overcome in the present, and only research and studies can find a way to improve or find new technologies which would solve these problems.

7.2.1 Implementation limits

The PURSUE application, described in section 6.1.4, put in evidence the Pluto main limitation: the immediate execution of actions in response to instantaneous events.

As already explained in chapter 4, the Pluto system allows the drones to perform their actions only at the end of the trip, that is a movement from a point A to a point B in the environment. For example, if the Drone has to take a picture in a specific location, it will fly from the ground station to that location and then take the picture.

There is no way to actively performing some action, reacting on events: so, as already explained, this is the problem of the PURSUE application, which requires to actively follow a moving object when it enters in the camera range.

This is an hint for the future expansion of Pluto, in order to manage also this kind of applications.

7.2.2 Technological limits

As already explained in chapter 3, the actuation in indoor contexts is tricky because of the localization problem. We showed some IPS methods, but still they are not as efficient and standardized as GPS. They introduce latency in the localization mechanism and their precision is lowered by physical obstacles, roofs and ceilings.

In this direction, research and future studies will certainly find a better indoor localization method, and Pluto will take advantages from it. Indeed the actuation tasks performed by the drones completely relies and depends on a localization base: indeed, in order to send a drone in a specific location to take a picture we need a method to precisely indicate that location.

Research and future studies will also find a way to improve the capacity of the nano-drones batteries. Nowadays their duration is approximately of 7 minutes, with a recharge time of 20 minutes. This is a great limitation, because the programmer is forced to develop applications where the sensing tasks must be performed within this limited amount of time.

Finding a solution to the limitation of the instantaneous actuation, together with new technological discoveries that will improve the drones battery duration and find a good and stable indoor localization method, can greatly enrich the Pluto programming framework.

With these future improvements the Pluto programming framework will be able to manage almost every kind of drones application.

Bibliography

[1] Morgan Quigley and Brian Gerkey and Ken Conley and Josh Faust and Tully Foote and Jeremy Leibs and Eric Berger and Rob Wheeler and Andrew Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[2] Karthik Dantu and Bryan Kate and Jason Waterman and Peter Bailis and Matt Welsh, "Programming Micro-Aerial Vehicle Swarms With Karma," in *SenSys '11 Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2011, pp. 121–134.

[3] Jonathan Bachrach and Jacob Beal and James McLurkin, "Composable continuous-space programs for robotic swarms," in *Neural Computing and Applications*. ACM, IEEE, 2010, pp. 825–847.

[4] Jacob Beal, "Programming an amorphous computational medium," in *Unconventional Programming Paradigms*. Springer Berlin, 2005, pp. 97–97.

[5] Luca Mottola and Mattia Moretta and Kamin Whitehouse and Carlo Ghezzi, "Team-level Programming of Drone Sensor Network," in *SenSys '14 Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. ACM, 2014, pp. 177–190.

[6] Keith S. Delaplane and Daniel F. Mayer, *Crop Pollination by Bees*. CABI New York, 2000.

[7] F. Nex and F. Remondingl, "UAV for 3D mapping applications: A review," in *Applied Geomatics*. Springer, 2003.

[8] U.S. Environmental Protection Agencyl, "Air Pollutants," in *goo.gl/stvh8*.

[9] J. Villasenorl, “Observations from above: Unmanned Aircraft Systems,” in *Harvard Journal of Law and Public Policy*, 2012.