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 Aphanasov

Correlatore:

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 use 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

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. Mottola and our mentor Mikhail Afanasov: without their help and support, this thesis would not have been possible.

We would like to thank Dr 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.

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.

x

Contents

List of Figures

List of Tables

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

We developed a programming framework for developing applications utilizing swarms of nano-drones.

We thought about three applications case study (see 6.1), from which we extracted a general behavioral model, which can be used for a wide range of applications; so we decided to create a general programming abstraction which describes a set of features that can be used for a wide range 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 manages the priority of a mission etc.

We wanted to facilitate as much as possible the work of the programmer, so we decided to develop a graphical editor (see 5.1) through which the programmer can create its application, 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 (see 4.2) for our programming abstraction, choosing a Java model to represent all the features we needed and enriching our description through Sequence diagrams, Visio diagrams and Pseudo-code representation of the model.

The editor and the model are strictly connected; the programmer can create its graphical schema through the graphical editor, and then the editor generates the Java code according to the model created.

We also developed the final user interface(see 5.5) to make the user able to fully exploit the applications created with Pluto programming framework, simply asking him what he wants the drones to do and where.

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.

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 in 3, the Pluto programming framework; first, the architecture of the system is analyzed, describing in details each component of framework. In this section we describe also the graphical editor and the user interface. Then the dataflow model of our framework is presented, using visual diagrams, in order to give a sound representation which is easy to understand. The last section of the chapter describes all the steps performed to arrive to the final system, showing all the previous 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 (see section 6.1). There is a description of many already existing applications and of three case study, and an analysis on whether they can be developed with Pluto. In section 6.2 we propose two exercises to real people, in order to evaluate the effective usability of Pluto. 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 of 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)[**?**], which provides a Publish/Subscribe coordination layer for decentralized computations, as shown in section 2.2.1; Karma[**?**], which lets programmers specify modes of operation for the swarm, such as "Monitor" or "Pollinate"(as shown in section 2.2.2); and Proto[**?**], which lets programmers specify actions in space and time(as shown in section 2.2.3).

2.2.1 Robot Operating System

ROS[**?**] 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.
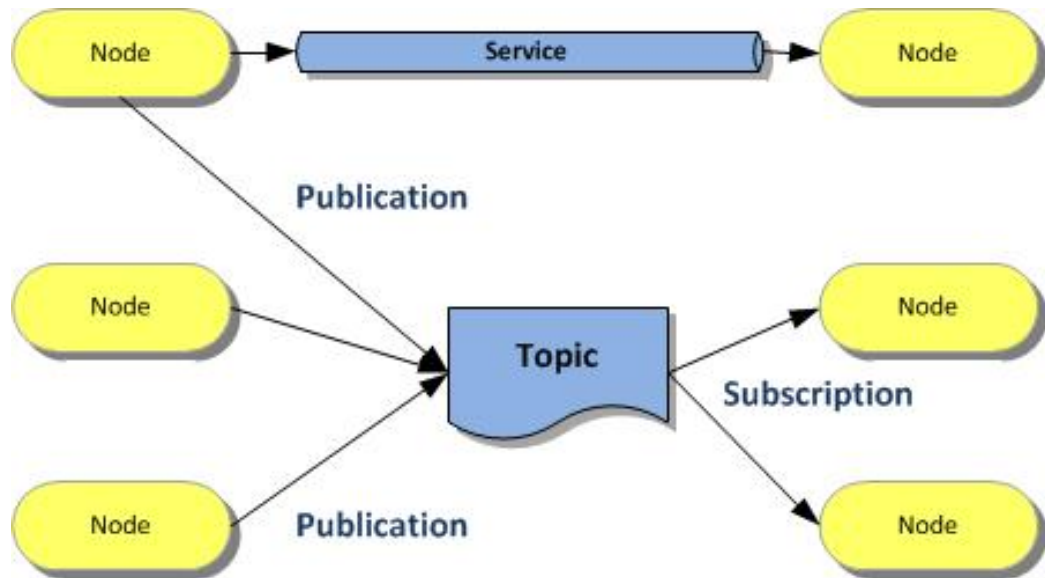
Figure 2.2: ROS communication layer functioning

2.2.2 Karma

Karma[**?**] 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[**?**] 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

9

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.
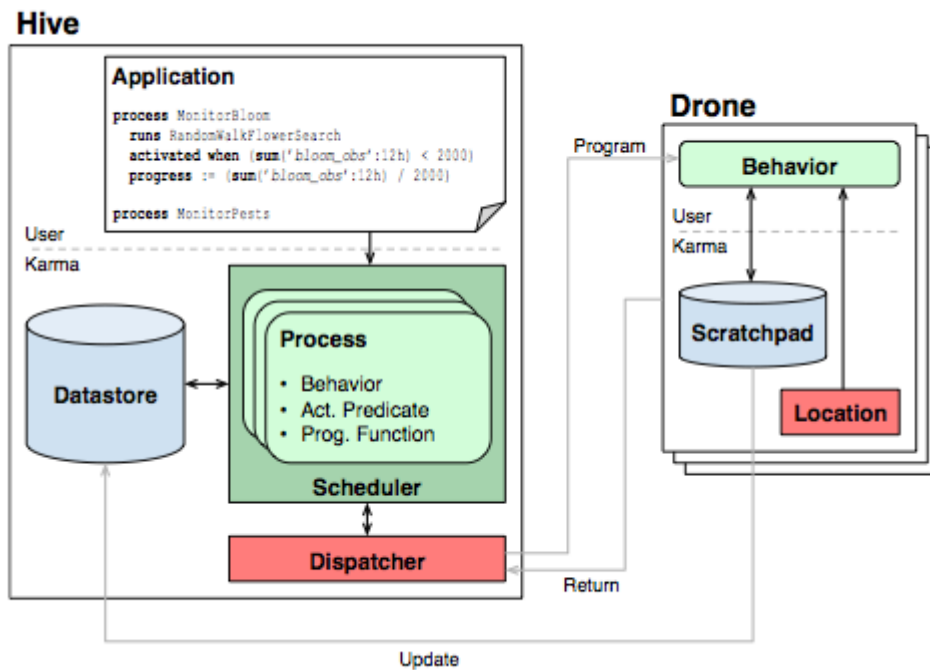
Figure 2.3: The basic schema of Karma

2.2.3   Proto

The amorphous medium abstraction[?] 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[?] 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.
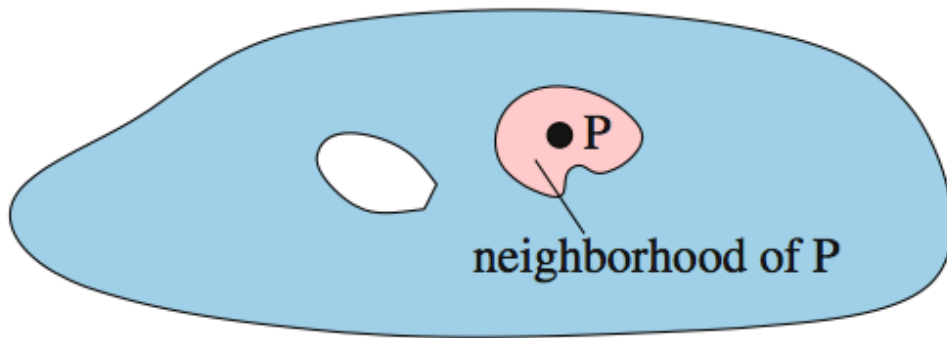
Figure 2.4: The amorphous medium abstraction

The Proto[?] language uses the amorphous medium abstraction[?] 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[?] language to mobile devices, such as drones in a swarm, the amorphous medium[?] must be extended with the concept of *density*; indeed for the vast majority of mobile applications, it is important to distribute drones depending
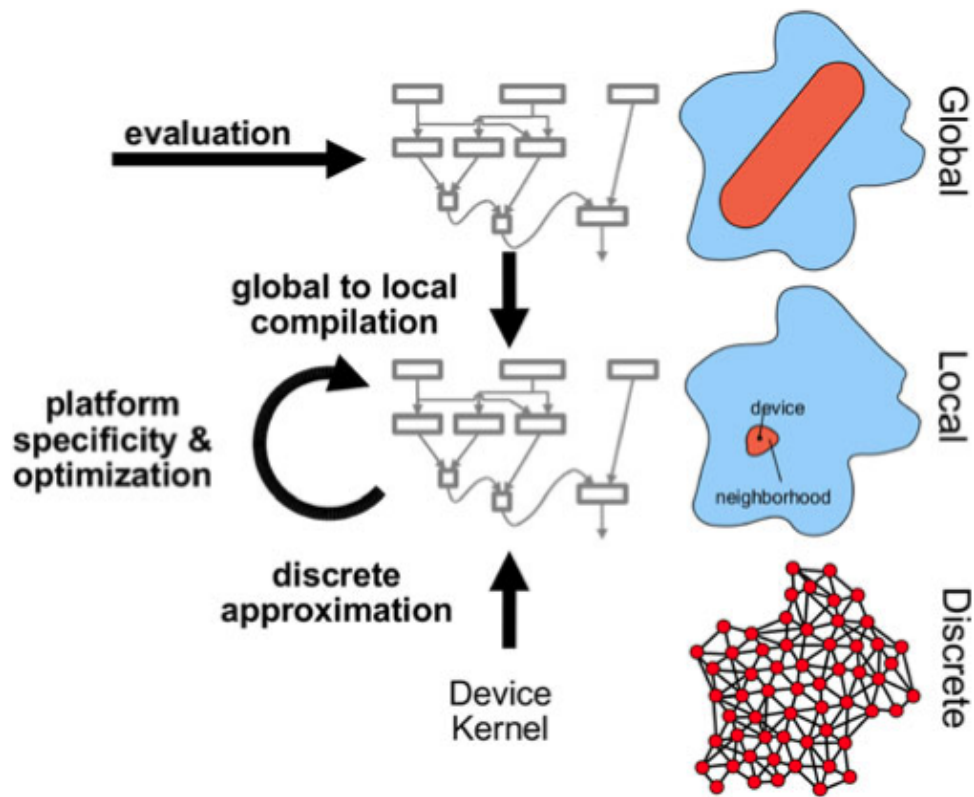
Figure 2.5: Proto: problem decomposition

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, 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[**?**], a system containing a set of programming constructs to explore the notion of team-level drone programming. Voltron's[**?**] 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[**?**] 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[**?**] 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, programmer constantly confronts 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,a roof or a ceiling. For that reason Indoor Positioning System(IPS) is widely applied for indoor localization. In this section we will discuss several implementations of IPS.

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. Although there is no "standard" for IPS methods, many commercial solutions have already been developed and are available on the market. 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. The localized nature of an IPS has resulted in design fragmentation, with systems making use of various optical,radio, or even acoustic technologies. Like in *trilateration*, an IPS needs three independent measurements to unambiguously find a location and there must be a sound method for reducing the unpredictable error budget significantly.

Today many IPS methods have been developed and they can be divided in two main categories:

- Non-Radio technologies

- Wireless technologies

Non-Radio 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.

Magnetic positioning

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 an accuracy of 1-2 meters and a confidence level of 90 percent, without using the wireless infrastructures.

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. Since IMUs cover only the differential of motions, the location gets determined by integration, and so there is need of integration constants to provide results.

Wireless technologies

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.

WiFi Positioning System (WPS)

WPS is based on measuring the intensity of the received signal(the well known 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

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.

RFID

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. Some mechanisms have to be applied to prevent from passing by out of range.

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. Existing advanced solutions add visual coverage with cameras to wireless coverage for the location, making possible indoor localization.

Angle of arrival

The angle of arrival is the inclination at which a signal arrives at the receiver, and is determined,having an array of directional sensors, by which sensor received the signal. Combining it with *triangulation* and a known base line makes it possible to determine location relative to two transmitters.

Time of arrival

Time of arrival is the amount of time a signal takes to propagate from transmitter to receiver. The travel time of a signal is used to determine this distance, since the signal propagates constantly. ToA, which is also used by GPS, require complicated synchronization mechanisms to leave sensors a reliable source of time. Since, in indoor contexts, objects like tables,walls or doors reflect and diffract the RF signal, temporal or spatial sparsity based techniques must be applied to reduce these effects.

RSSI

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.

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.
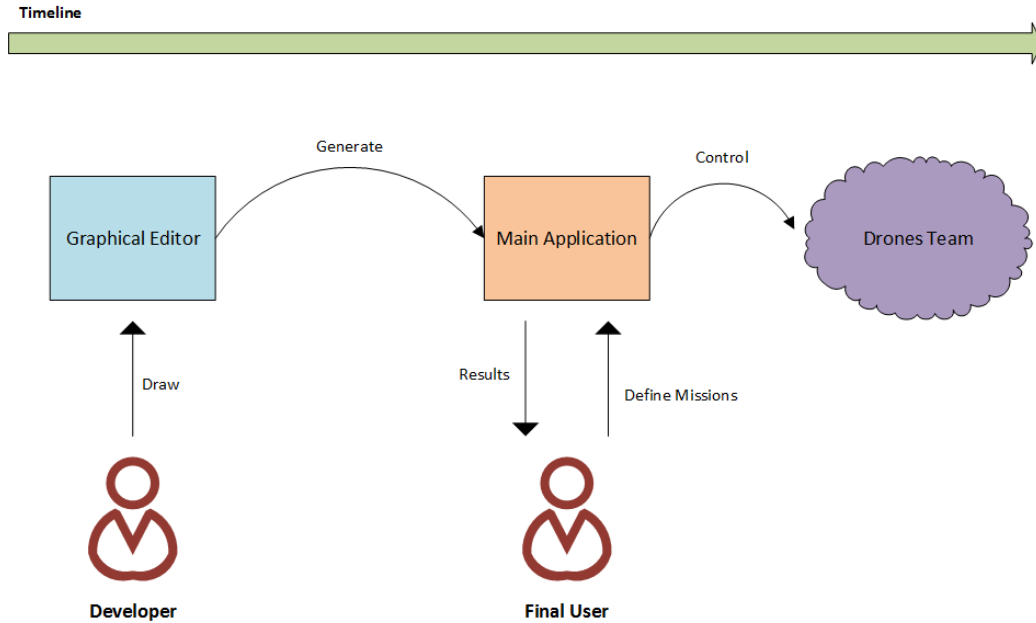

Figure 4.1: A complete life cycle of the application

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
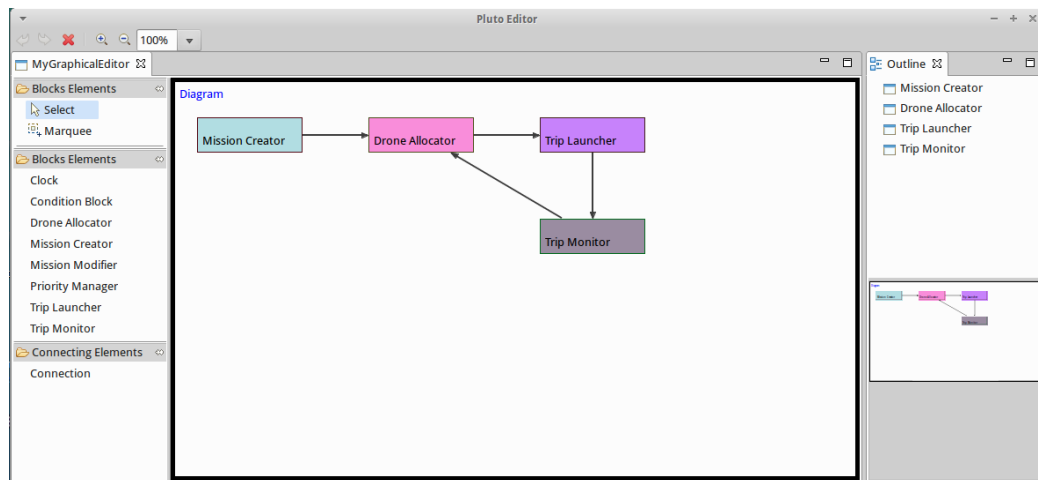
Figure 4.2: Pluto Graphical Editor interface

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

25

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.
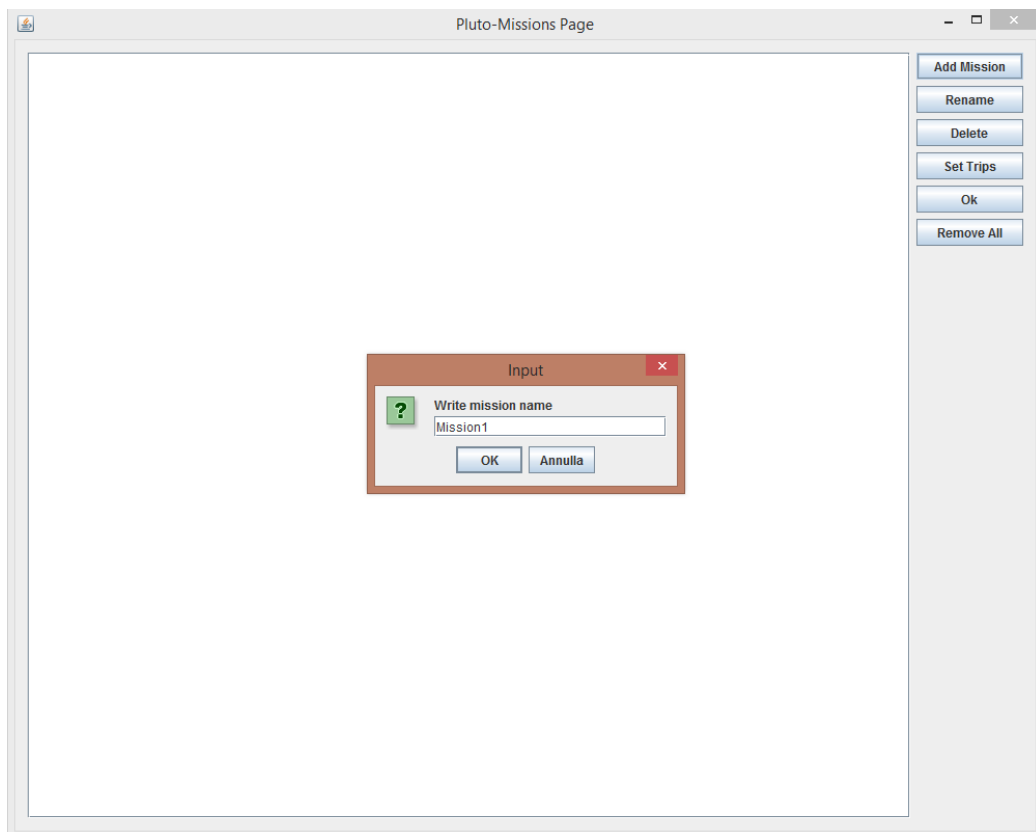
Figure 4.3: Mission Page interface

Then, the user is asked to decide if the mission must be repeated. If he click "Yes", that mission will continue to execute cyclically until the user decide to stop it, through the Stop button of the Monitor Page.

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 4.5, and the user can add Trips entity
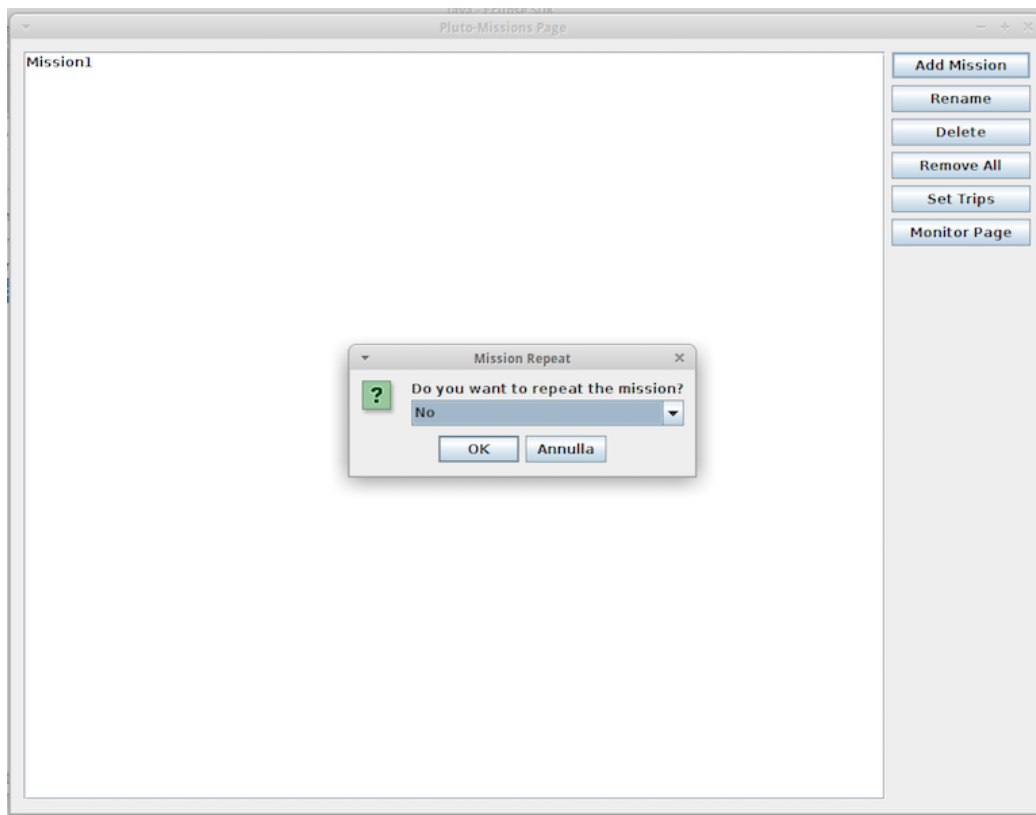
Figure 4.4: Mission Repeatable pop up

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.

So after all the Trips are added to a single Mission the user can create more
Mission or pass to the Monitor Page with the corresponding button. The Monitor
Page, shown in figure 4.6, 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

27
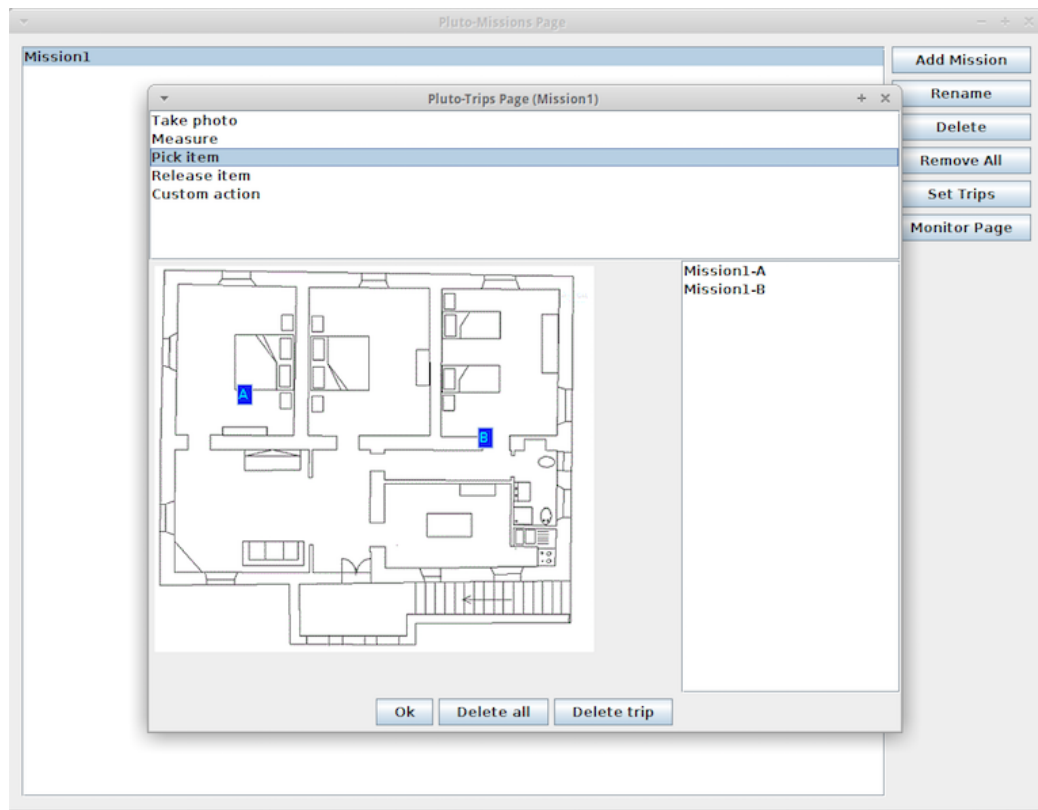
Figure 4.5: Trips Page interface

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.
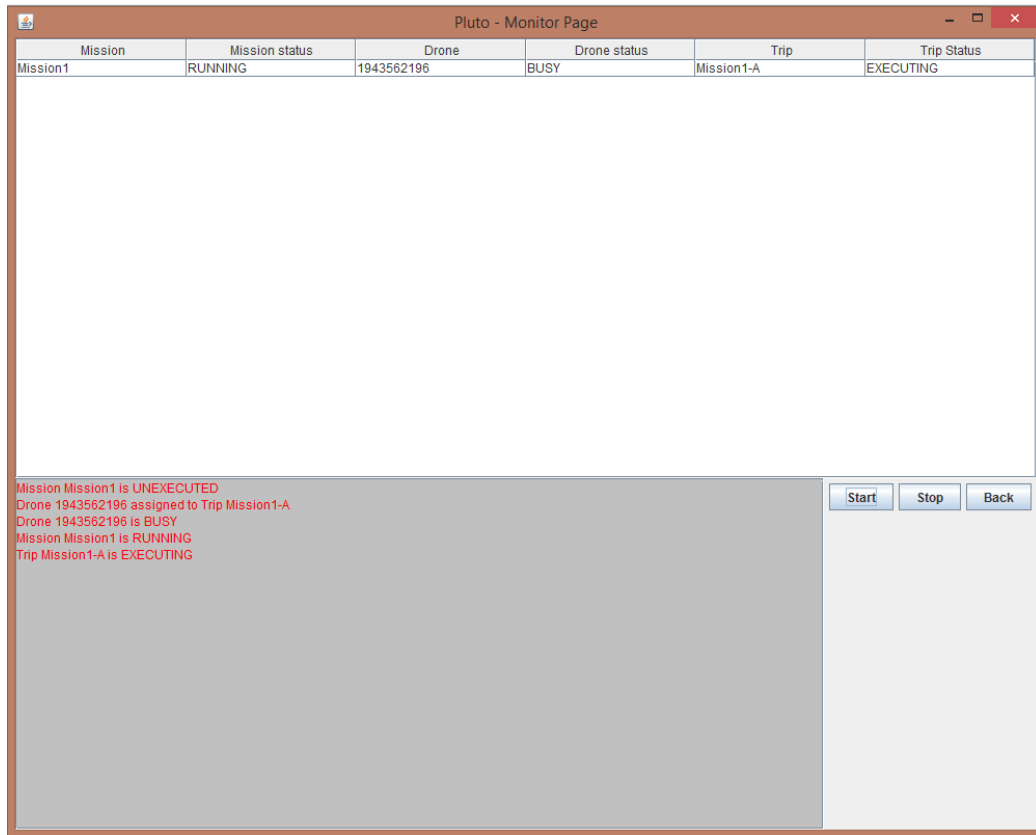
Figure 4.6: Monitor Page interface

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 4.7, 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 4.7: An example Pluto application

4.2.2 Model representation

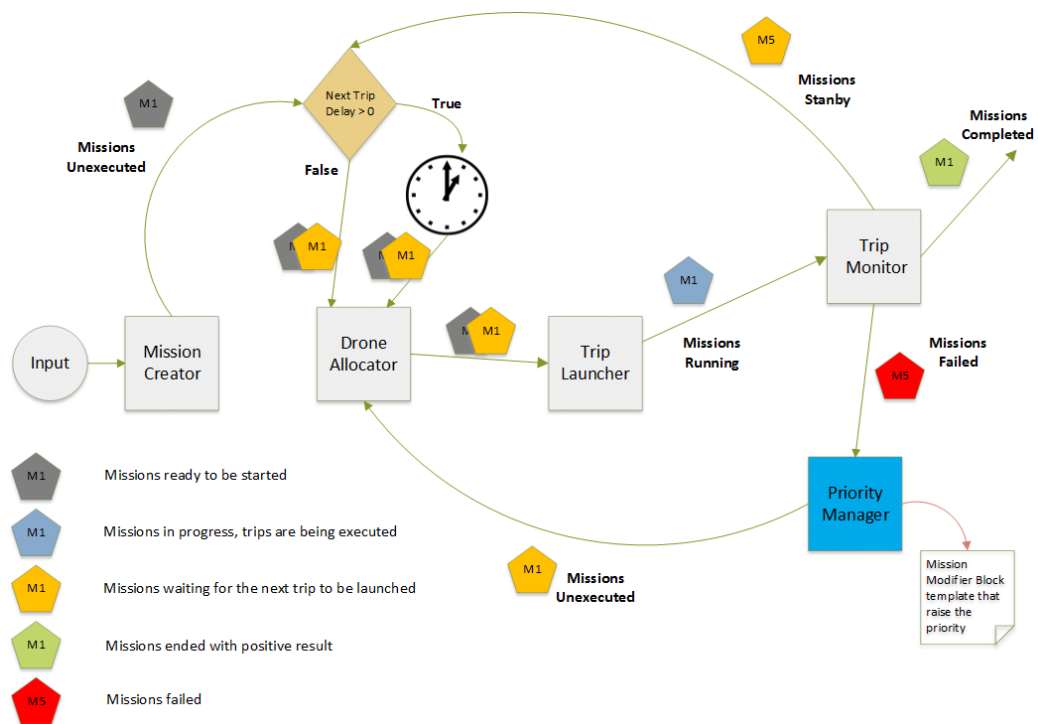In this section we show the fundamental entities of our model, in order to better understand the functioning of the blocks. Then, all the blocks of figure 4.7 are described in details, explaining what tasks they perform and showing a pseudo-code representation for each of them.

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.

So we identified the following entities:

- Mission

- Trip

- Drone

- Action

- Item

As shown in figure 4.8, the Mission entity, in addition to the set of Trips, has another important attribute that is the Status: it describes how the Mission is being executing.

The Trip entity has a Status attribute too and furthermore it contains the Drone and the Action entities. Then of course the coordinates of the source and the target locations.

The Drones are described by an 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[**?**], which will be described in section 6.1.1.

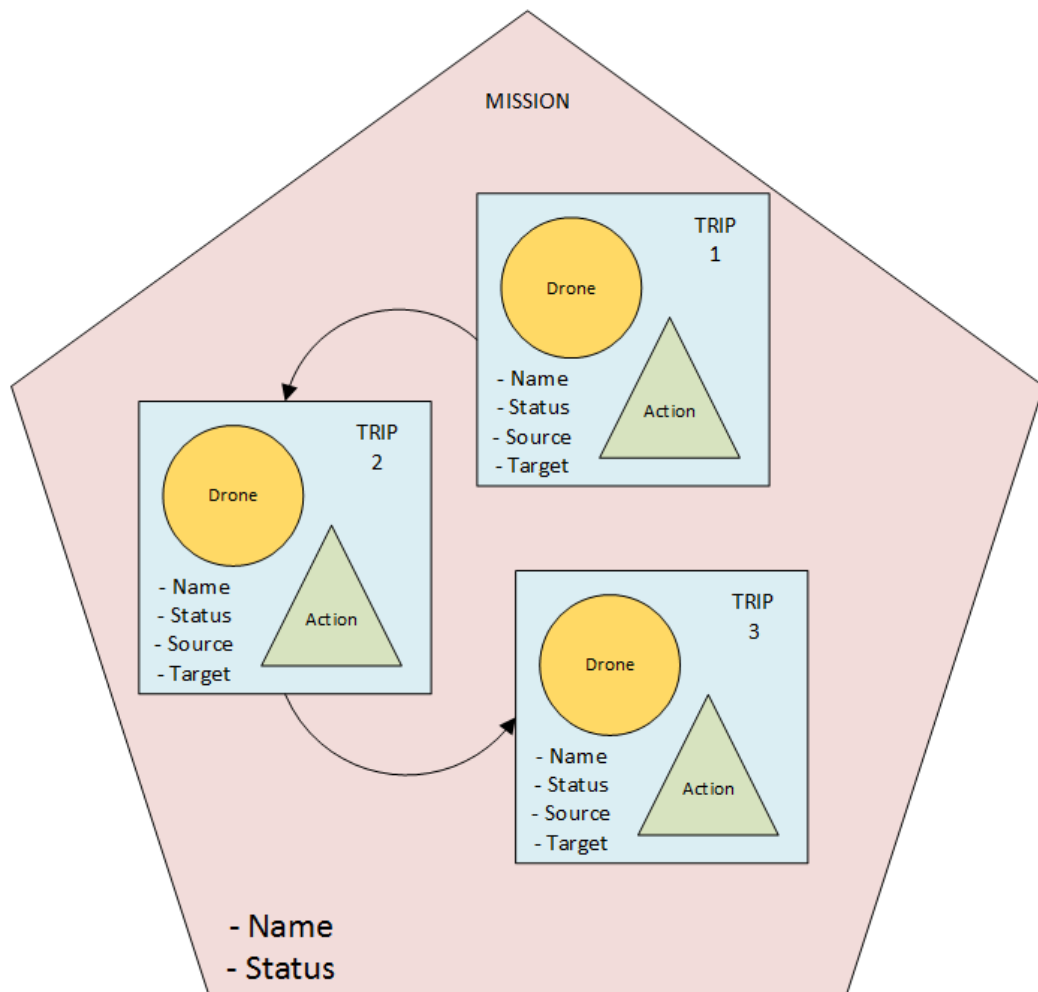The code implementation of these classes can be found in section 5.3.

Figure 4.8: 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 MissionCreator 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 MissionCreator and the DroneAllocator blocks.

DroneAllocator 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.

TripLauncher block

input: a Mission object
output: a Mission object
The TripLauncher block takes the next Trip to be performed from the Mission ob-

33

ject, 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.

TripMonitor 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.

MissionRepeater 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.

GateFIFO 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.

MissionEvaluator 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.

MissionModifier block

input: a Mission object

output: a Mission object

The MissionModifier block allows the programmer to create new custom blocks. As shown in figure 4.9, 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.
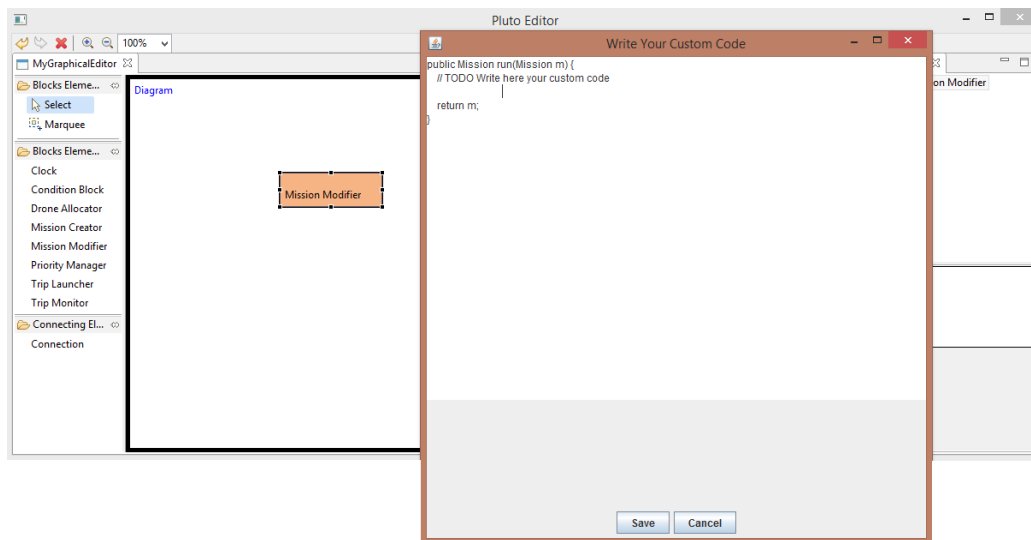
Figure 4.9: The MissionModifier block

PriorityManager block

input: a Mission object
output: a Mission object

The PriorityManager block raise the priority of the first Trip of the Mission passed in input. This block can be considered as a particular case of MissionModifier. This block can receive the input from the GateFifo or TripMonitor blocks, and sends the output to the DroneAllocator.

TimerMonitor 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.

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.
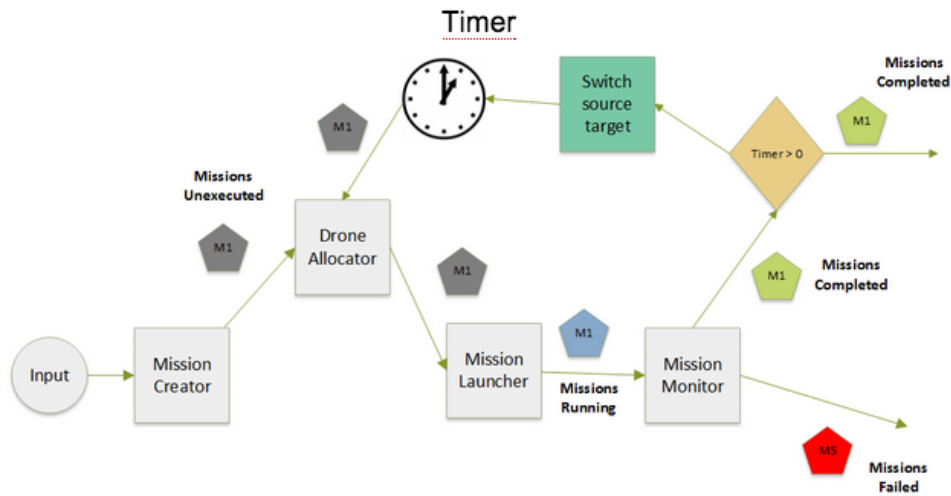

Figure 4.10: Solution without the Trip concept

After analysing 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. 4.12) and the missions with an associated timer(fig. 4.11), respectively. There wasn't the MissionModifier block, but only the "Priority Manager" one, so the programmer couldn't insert blocks(fig. 4.13) with custom code.
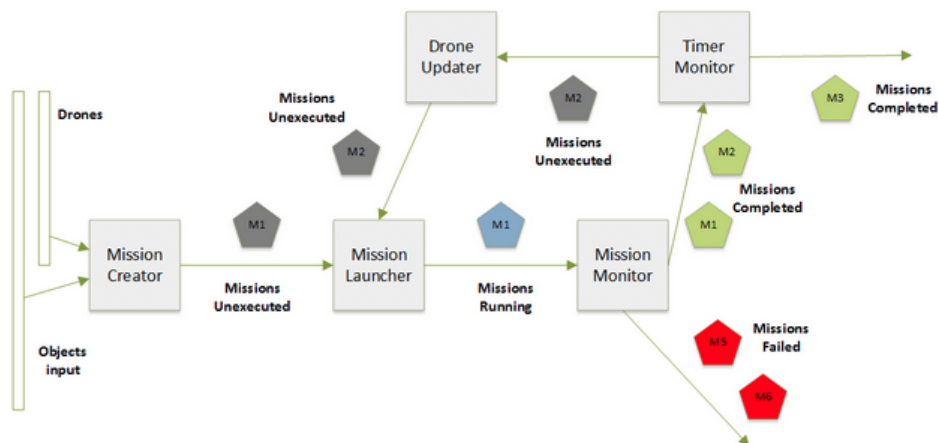
Figure 4.11: Solution with the TimerMonitor
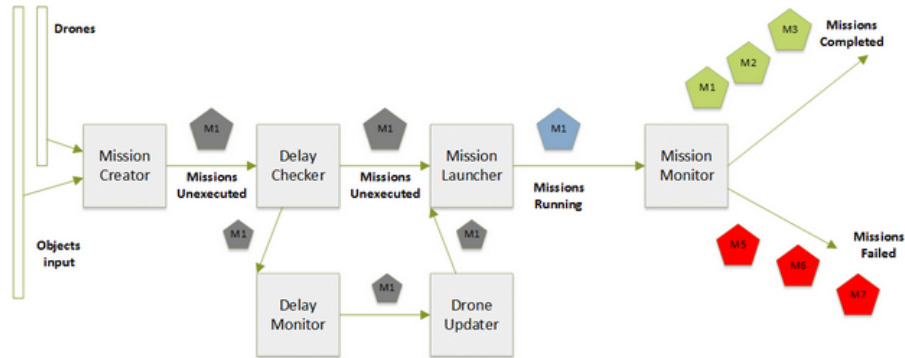
Figure 4.12: Solution with the DelayMonitor
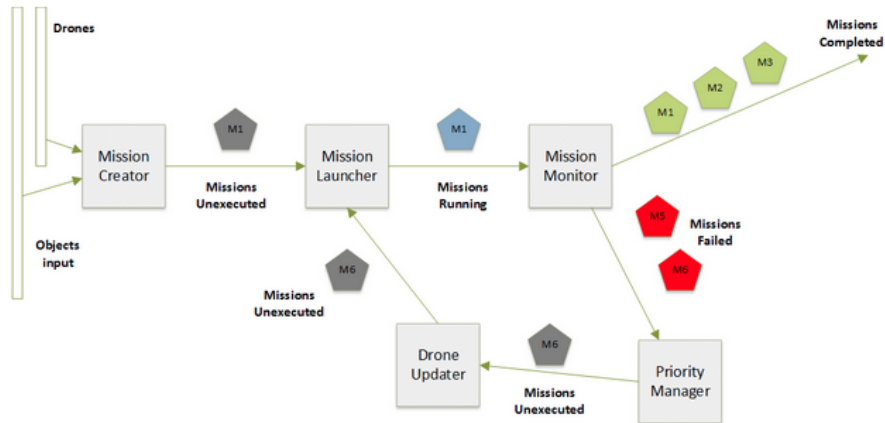
Figure 4.13: Solution without the MissionModifier block

Actually we decided to put the DroneAllocator block because we need 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 Priority Manager 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

5.1 Graphical editor

Description of the GEF Framework with code examples

Graphical Editing Framework (GEF) provides a powerful foundation for creating editors for visual editing of arbitrary models. Its effectiveness lies in a modular build, fitting use of design patterns, and decoupling of components that comprise a full, working editor. To a newcomer, the sheer number and variety of concepts and techniques present in GEF may feel intimidating. However, once learned and correctly used, they help to develop highly scalable and easy to maintain software. This section aims to provide a gentle yet comprehensive introduction to GEF. It describes our Pluto Graphical Editor.

5.2 Code generation

Description of the generation process of the code from diagram

5.3 Object-oriented approach

Why we decided to use JAVA, How we implement entities, UML diagrams of classes, MVC pattern

5.4 Runtime Management

Description of the management of threads, pub/sub pattern and log procedures

5.5 User interface

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:

- 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 widgets like trees and tables. Swing itself is written in Java. Swing is a 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.

We used Swing to develop the user interface of the Pluto programming framework. As already shown in section 4.1.2, the interface is composed by three main sections: the Missions Page, the Trips Page and the Monitor Page. In the proceeding of this section, we show the code of the three pages, commenting the main Swing features used to build them.

Missions Page

Here we describe the main Swing features of the Missions Page, whose graphical result is shown in section 4.1.2:

```
 1
 2    private JList<String> list;
 3    private JButton remallbtn;
 4    private JButton addbtn;
 5    private JButton renbtn;
 6    private JButton delbtn;
 7    private JButton tpsbtn;
 8    private JButton okbtn;
 9
10    public MissionsPage() {
11       initUI();
12    }
13
14    private void createList() {
15
16       DefaultListModel<String> model = new DefaultListModel<String>();
17       list = new JList<String>(model);
18       ...
19    }
20
21    private void createButtons() {
22
23       remallbtn = new JButton("Remove All");
24       addbtn = new JButton("Add Mission");
25       renbtn = new JButton("Rename");
26       delbtn = new JButton("Delete");
27       tpsbtn = new JButton("Set Trips");
28       okbtn = new JButton("Monitor Page");
29
30    }
```

This is the basic structure of the Missions Page: the *JList* component is the list of the Missions created by the user. At the beginning it's empty and it'll be filled with the string representing the name of the Missions the user choose to create. It is created thanks to the *createList()* method. There is a number of *JButton* components: these are all the buttons that the user can click on the Missions Page, and each of them perform a different action, as already explained in section 4.1.2. They are created thanks to the *createButtons()* method. The constructor of the Missions page only calls the *initUI()* method, whose implementation is shown in the following code snippet:

```
1
2    private void initUI() {
3
4       createList();
5       createButtons();
6       Container pane = getContentPane();
7       GroupLayout gl = new GroupLayout(pane);
8       pane.setLayout(gl);
9
10
11       gl.setHorizontalGroup(
12             ...
13          )
14
15       );
16
17       gl.setVerticalGroup(
18             ...
19          )
20
21       );
22
23       ...
24
25       setTitle("Pluto-Missions Page");
26       setSize(1000, 800);
27        ...
28       setDefaultCloseOperation(EXITONCLOSE);
29    }
```

First of all the *createList()* and *createButtons()* methods are called, in order to create the empty list of Missions and all the buttons of the Missions Page. We chose a *GroupLayout* for the visualization of the Components in the page, but there exists a lot of Swing layouts that can be used. Through the *setHorizontalGroup* and *setVerticalGroup* methods, we simply chose the graphical disposition of the components in the Missions Page. Finally, we set the title and the size of the Missions Page window, and, through the *setDefaultCloseOperation(EXITONCLOSE)* method, we make the Missions page close when one clicks on the close icon on the top of the window.

The rest of the code of the Missions page deals with the reaction of the model when an action on a component is performed. For example, when the *Add Mission* button is clicked, a little window asking the name for the Mission appears, followed by another window asking if the Mission must be repeated. Since in this section we want to show only the graphical part of the Pluto Main Application, and this

code involves the whole MVC pattern, we decide to not show it, because it would be too specific and we would have to show a lot of different features of the system, involving also the Model and Controller parts.

Trips Page

jhsd

5.6 The crazyflie nano-quadcopter

Description of crazyflie API

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 can 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 6.8 and 6.10), although it perfectly works also in the case of a single drone (see 6.6).

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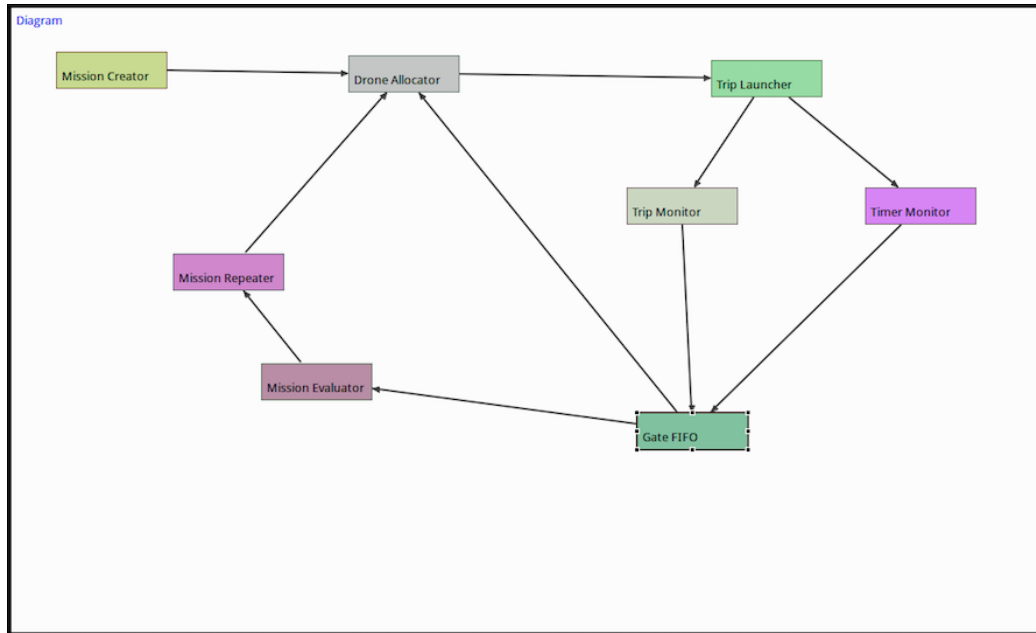
Figure 6.1: Pluto graph for the Alfalfa Crop Monitoring and Pollination application

We tried to implement this application drawing a diagram showed in figure 6.1. this diagram is an extension of the basic one showed in the chapter 4.2.1. In order to satisfy the application constraints, we needed to add a couple of new blocks. The first difference consist in the presence of the TimerMonitor block parallel to the TripMonitor. This block is added in parallel because we need it to monitor the running Trip at the same time of the TripMonitor. Then the GateFIFO will wait for the first instance of the Mission from one of these two blocks. After that, if the mission is completed, it will pass through the MissionEvaluator that will evaluate the actions done. Further the MissionRepeater will make the mission to start again from the first Trip. With these blocks we were able to respect the main constraints of the Alfalfa[?] application. After the generation of the code from the diagram, we needed to make some customizations in the Main Application sources, such as the add of a Pollination Action and the implementation of the Evaluator algorithm. Following there are these two pieces of code commented to better understand their behaviors.

```
1    String result = null;
2
3    // Retrieve all entries of the map
4    for (Map.Entry<Trip, Object> entry : dataMap.entrySet()) {
5
```

49

```
6          // consider only the trips related to the mission to evaluate
7          if (missionToEvaluate.getCompletedTrips().contains(entry.getKey())) {
8
9            // take the Photo of the current Trip iteration
10           Photo photo = (Photo) entry.getValue();
11
12           // checking pest and disease
13           if (photo.hasPest() || photo.hasDisease())
14
15             return "WARNING: Pest/disease at location: "
16                 + entry.getKey().getTargetLocation();
17
18           // checking for some bloom
19           if (photo.hasBloom()) {
20
21             // create a new Trip to pollinate the flowers
22             Trip trip = new Trip();
23             trip.setName("PollinateTrip");
24             trip.setTargetLocation(entry.getKey().getTargetLocation());
25             trip.setAction(Action.POLLINATE);
26             trip.setStatus(Trip.WAITING);
27
28             // add this new trip to the list of the mission
29             missionToEvaluate.getTrips().add(trip);
30             // set the mission status to STANDBY
31             missionToEvaluate.setStatus(Mission.STANDBY);
32           }
33         }
34       }
35
36       // set the result of the evaluation
37       result = "Success";
38       return result;
```

The "dataMap" attribute is an *hashmap* that binds each Trip with the picture taken by the action. For each photo taken by the drone, the code checks the presence of *pests* and *diseases*. In that case, the system will warn the farmer the location where the problem exists, through the log function. If there are *blooms*, the plants at that location must be pollinated, so a new Trip is created: its status is set to WAITING, its Action is set to Pollinate and the target location is set to the same location of the Trip that found the bloom. Finally this Trip is added to the execution queue of the mission. The Mission status is set to STANDBY because there is something new to do.

The following is 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        Camera camera = System.getCamera();
6        Photo photo = camera.takePhoto();
7        return photo;
8    }
```

In the end we simulated the final user, so we created two missions with eight trips each, as if we needed to monitor a real field.

Figure 6.2: Trip distribution screenshot

To further understand the behavior of the MainApplication we drawn a sequence diagram, showed in figure 6.3.

51

Figure 6.3: Sequence diagram of the MainApplication behavior

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 the "Domus dei Putti Danzanti" site application.
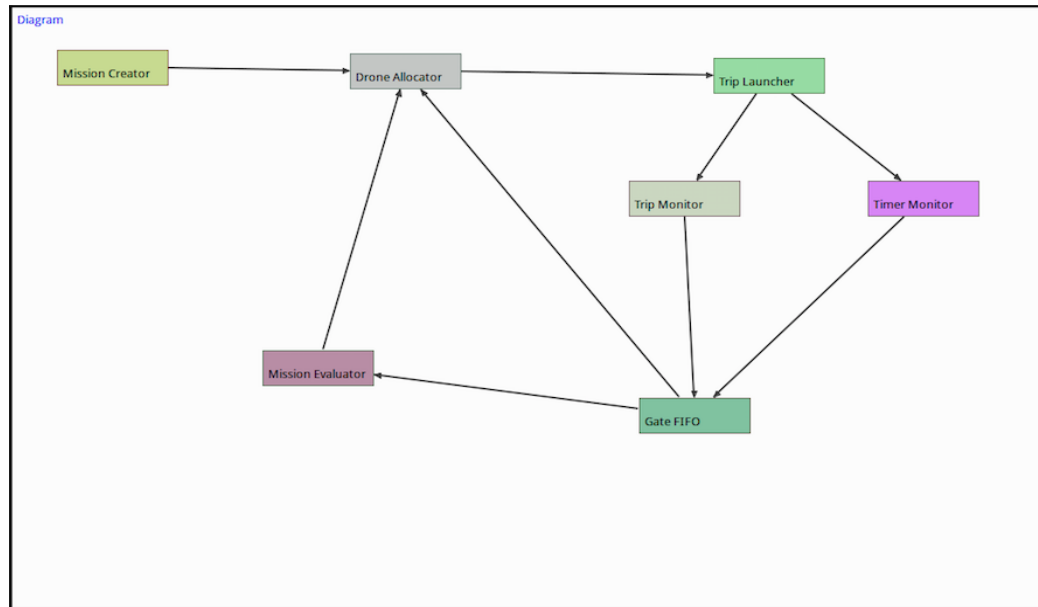

Figure 6.4: Pluto graph for the Aerial Mapping of the "Domus dei Putti Danzanti" site

The drones can already take pictures thanks to the *Take Photo* action. To send the drones to take pictures over the site location, the user has to simply create the Missions entities and add the Trips to the map he will be provided with. In the case the archaeologists want to send more drones on the locations where they can't obtain an adequate ortophoto, they just have to add more Mission entities. For example, if one archaeologist want to send 3 drones on a particular location, he has to create 3 Mission entities. Then for each Mission he simply has to drag and drop the *Take photo* action on that particular location. 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. Thanks to the ActionEvaluator 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 that specific location. 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.

Concerning the code of the Evaluator block needed for the development of the Aerial Mapping[**?**] 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[**?**] one, because the Evaluator algorithm acts in a different way: in Alfalfa[**?**] 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. This is why, as said in section 4.2.2, we decided to activate the ActionEvaluator block only when the Mission status is Completed. Following we show our implementation for the Evaluator algorithm:

```
1    String result = null;
2    // generate the final ortophoto
3    OrtoPhoto ortophoto = stitch(dataMap.values());
4
5    // if the aberration is greater than a fixed threshold
6    if(ortophoto.getAberration() > ABERRATION_THRESHOLD){
7
8      // for each photo that compose the ortophoto
9      for (Photo photo: ortophoto.getPhotoCollection()){
10
11       // if the overlap of the photo is not enough
12       if (ortophoto.getOverlapOfGivenPhoto(photo) < OVERLAP_THRESHOLD){
13
14         // Retrieve all entries of the map
15         for (Map.Entry<Trip, Object> entry : dataMap.entrySet()) {
16
17           // take the Photo of the current iteration
```

```
18              Photo p = (Photo) entry.getValue();
19
20              // if we found the photo in the map
21              if(photo.equals(p)){
22
23                // create a new Trip
24                Trip trip = new Trip();
25                trip.setName("NewTrip");
26                trip.setTargetLocation(entry.getKey().getTargetLocation());
27                trip.setAction(Action.TAKE_PHOTO);
28                trip.setStatus(Trip.WAITING);
29
30                // add this new trip to the list of the mission
31                missionToEvaluate.getTrips().add(trip);
32
33                // set the mission status to STANDBY
34                missionToEvaluate.setStatus(Mission.STANDBY);
35              }
36            }
37          }
38        }
39      }
40
41    // set the result of the evaluation
42    result = "Success";
43    return result;
```

In order to show the real runtime execution of the Aerial Mapping application with Pluto, we now show a real scenario: there are 7 drones and 1 big area 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, for each Mission, he has to create the list of Trips by using the map to drag and drop the action *Take photo* on the locations forming the big area.

6.1.3   PM10

The PM10[?] 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.

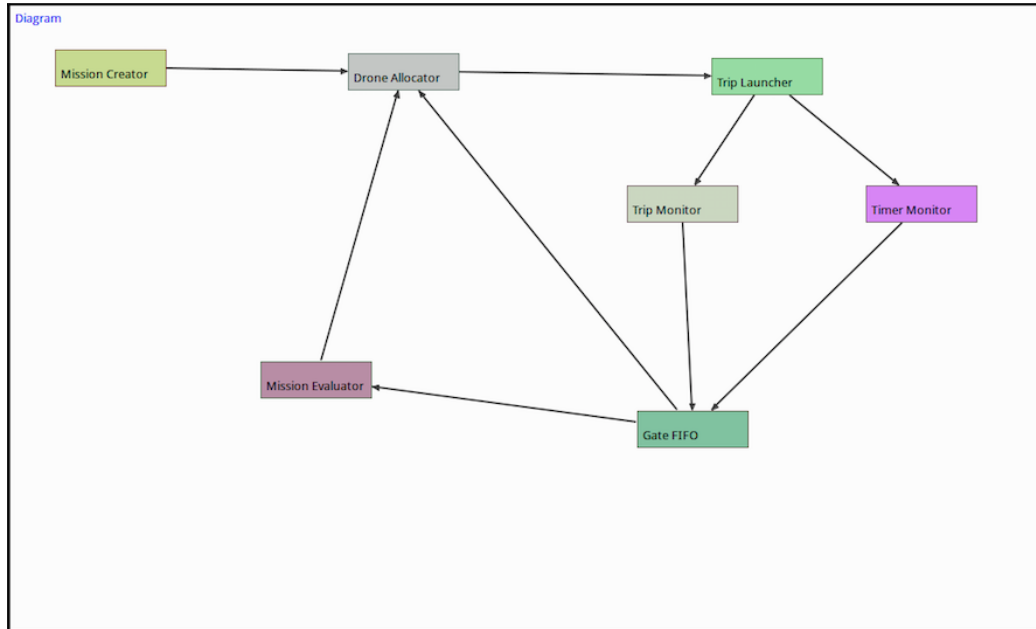The following figure is the Pluto editor graph needed to create the PM10 application:

Figure 6.5: 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 MissionEvaluator blocks, the *pollutionQuantity* variables are confronted and the gradients between areas of higher concentration are computed. Then the 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.

Below there is our implementation of a possible Mission 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 the mission
28     missionToEvaluate.getTrips().add(trip);
29     // set the mission status to STANDBY
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:

6.1.4 PURSUE

The PURSUE application[**?**] 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 immediately take a picture of the moving object. This problem can be lowered by inserting a lot of trips in the area to monitor, with strict time constraints on them. But still there is no way for the drones to actively follow the moving objects.

All the previous applications were already developed and tested with other systems, like Karma[?] and Voltron[?], 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); in the following subsections we will describe these applications in details, also using a visual representation of their behaviour, and sequence diagrams, to make understand better the whole functioning of each one of them.

6.1.5 Object-finder (OF)

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

- 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, deciding the path for each one of them

- the drones fly to the assigned location and, if found, bring the object back to the user, through a magnet or robotic arm
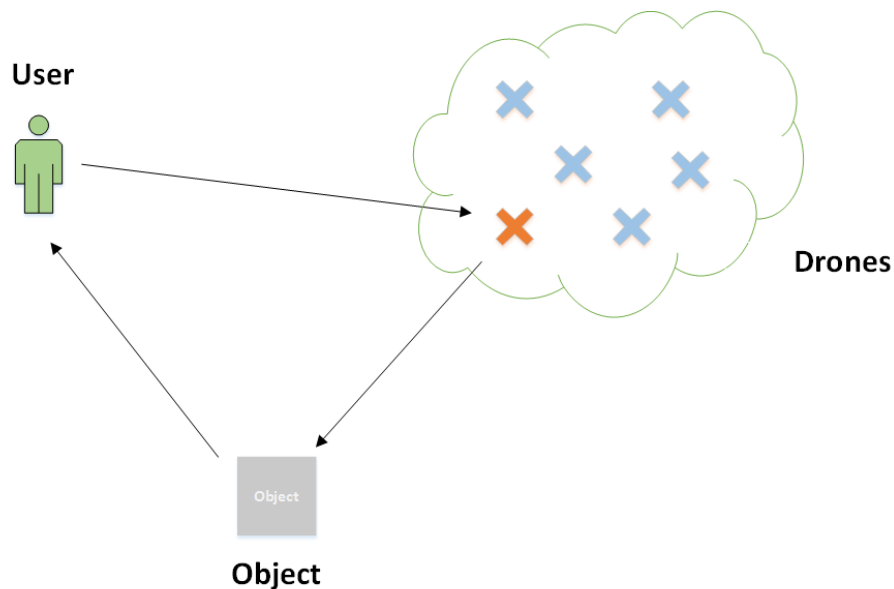
Figure 6.6: The basic functioning of the Object-finder application
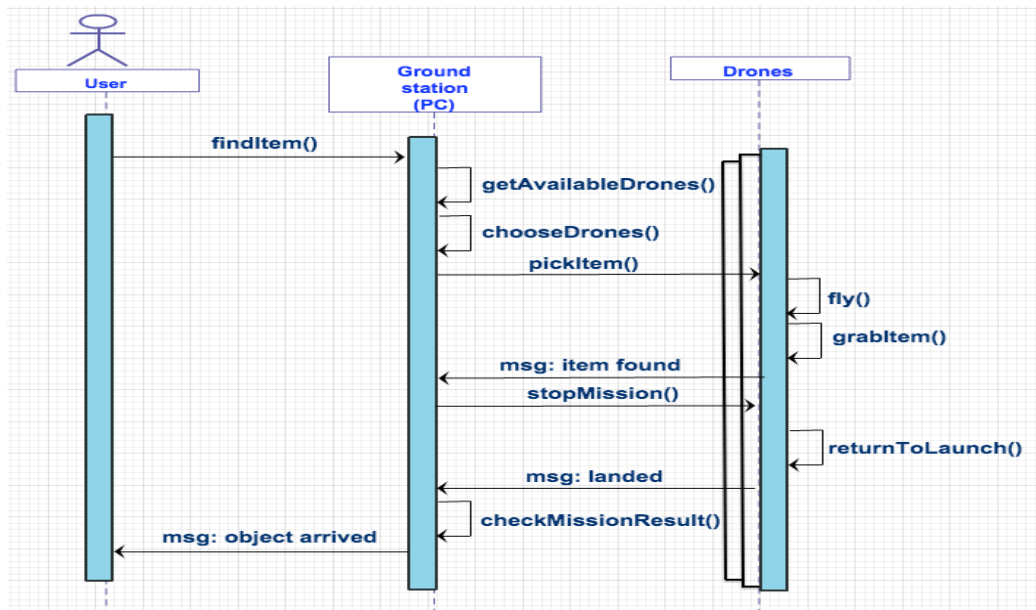
Figure 6.7: Sequence diagram of the Object-finder application

The user sets which item he needs in the ground station and sends the command; the ground station checks if there are drones unavailable, maybe broken or with low battery status. Then selects the drones that could carry out the mission and send them the command to start the mission. The ground station communicate with each drone, but in the sequence diagram we described the communication only with one of them. After the mission starts, all the drones start looking for the item in the house and when the lucky one finds it, it grabs it and send a notification message to the Ground Station. So, the Ground Station sends a "Stop Mission" message to each drone flying; in this way they could return to the home position. In the end the Ground Station checks if the drone with the item is returned and notifies it to the user.

6.1.6 Warehouse item-finder (WIF)

This application will be used to manage a warehouse, taking a list of objects to the user:

- the user makes a list of needed items and writes it on his laptop, tablet or smartphone

- the main system organizes the swarm of drones and decide which one will take each item in the list

- the drones fly to the assigned object and bring them back to the user, through a magnet or robotic arm
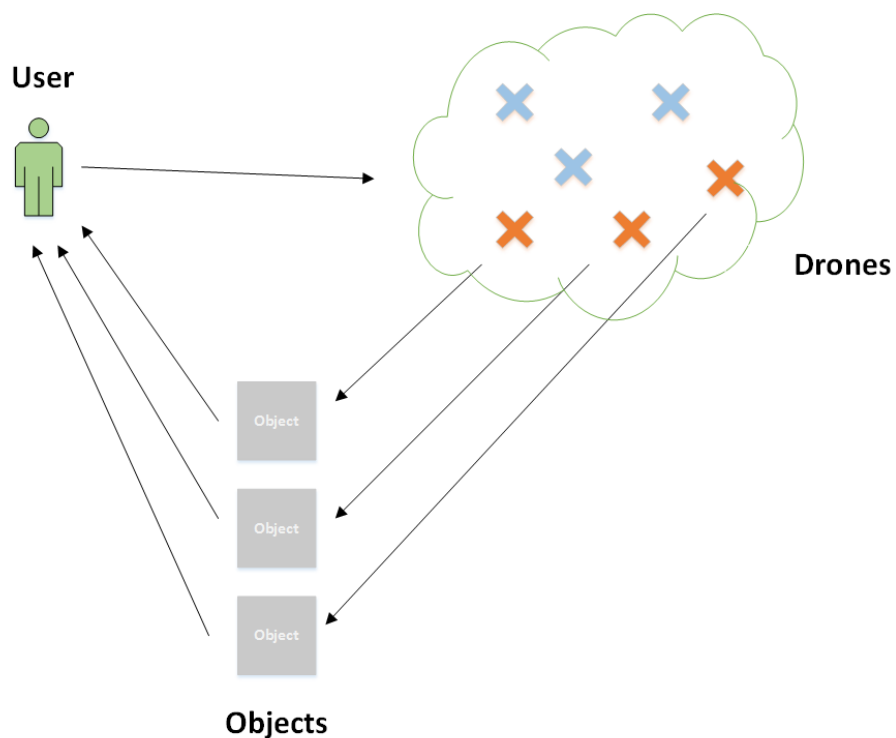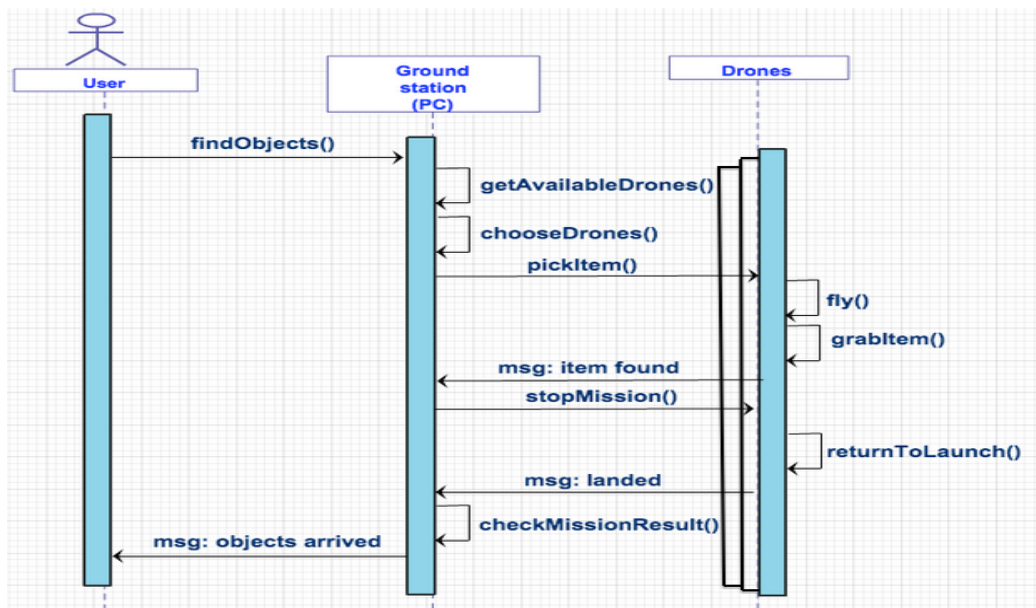

Figure 6.8: The basic functioning of the Warehouse item-finder application

Figure 6.9: Sequence diagram of the Warehouse item-finder application

6.1.7 Drugs distribution (DD)

This application is thought to assist elders to take their daily drugs, in an hospice context:

- the nurse will prepare a little box with each patient's daily medicine

- each drone, at the right time of the day, will bring the box to his assigned patient

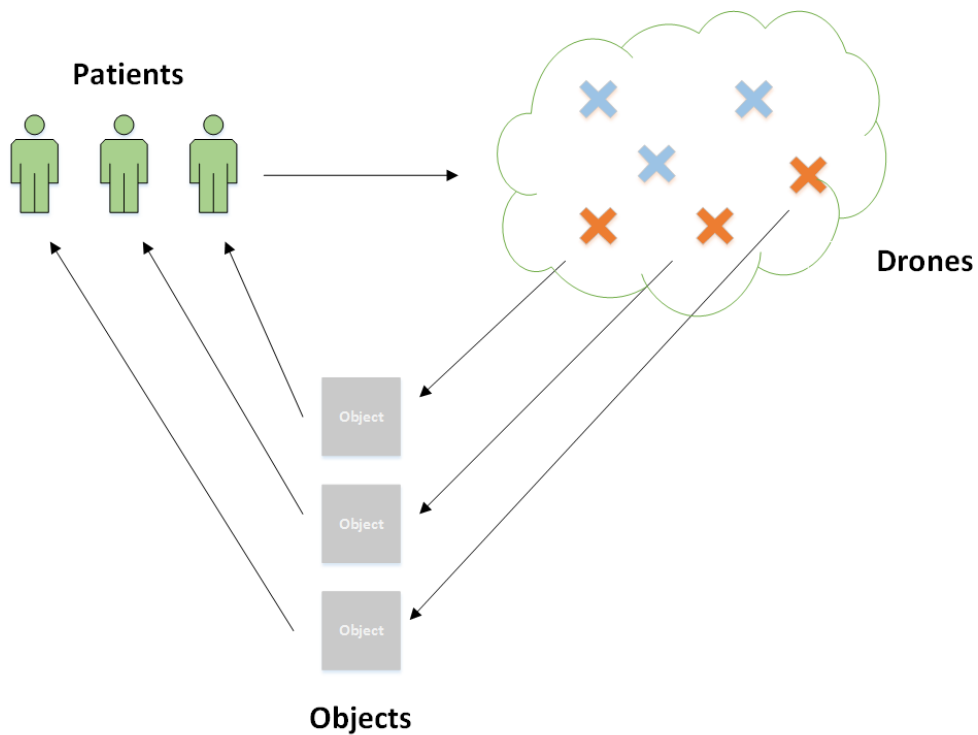- after carrying out their action, the drones return to the start location

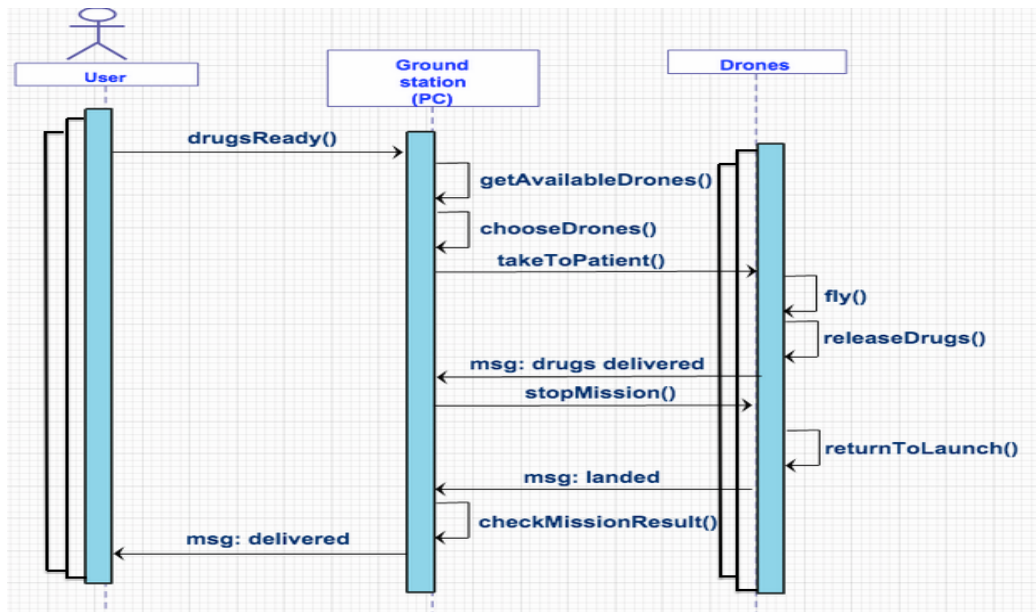Figure 6.10: The basic functioning of the Drugs distribution application

Figure 6.11: Sequence diagram of the Warehouse Drugs-distribution application

The OF,WIF and DD applications can be easily developed with the Pluto editor. Their basic versions are represented by the basic graph of each application, shown in figure (see 6.12),and they can be possibly enriched with the Priority Manager and Timer Monitor blocks, in order to improve the priority of the failed trips or to give a time constraint within the trips must be executed.

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". The tester were recruited 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 6.12, 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.
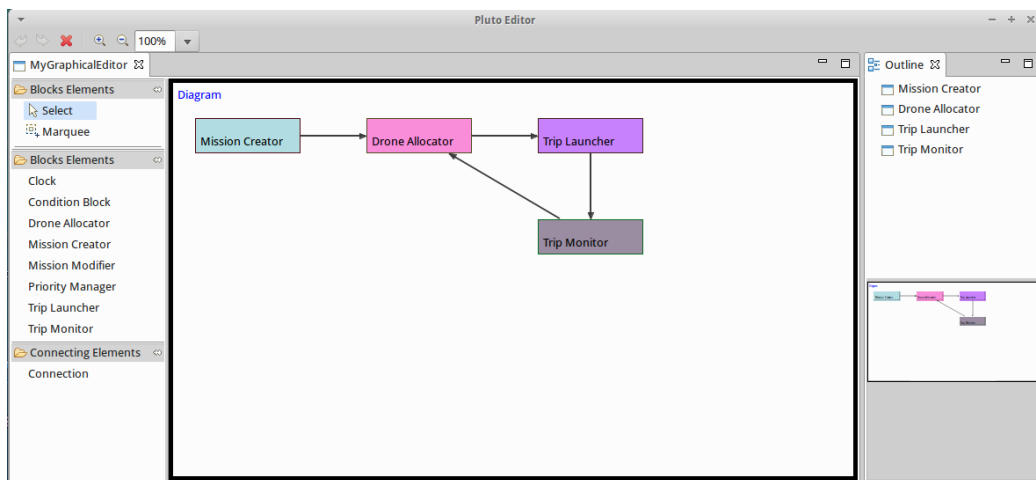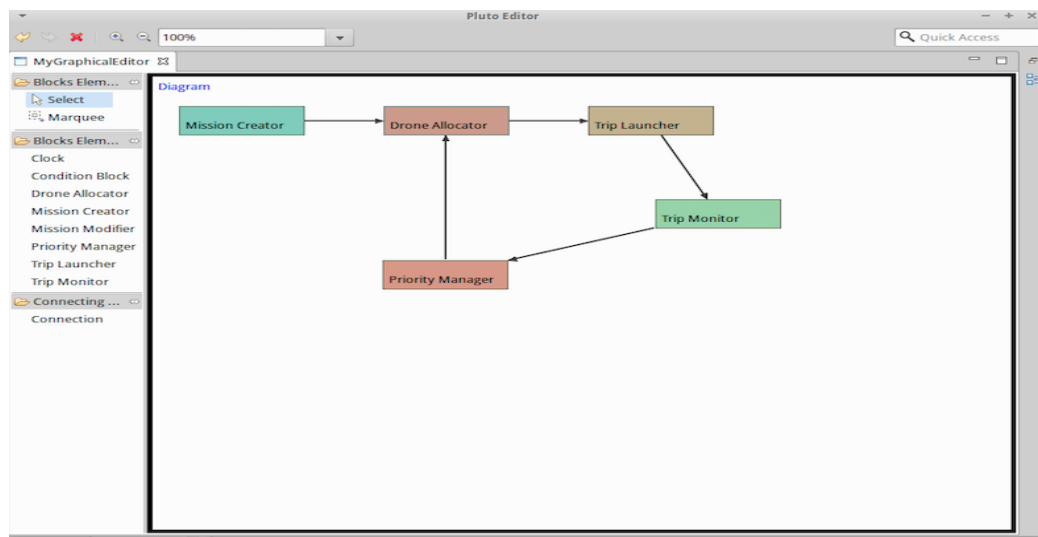

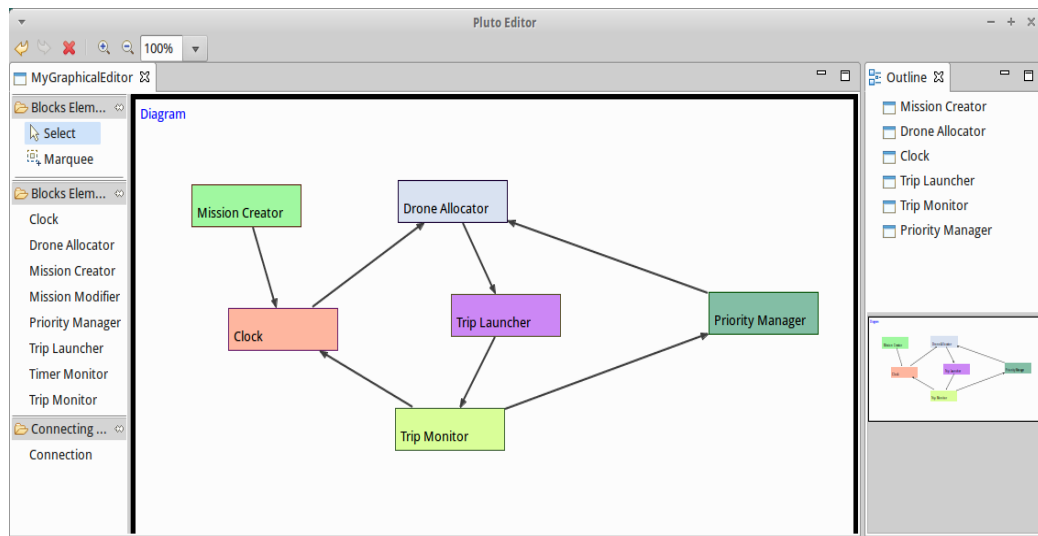Figure 6.12: Solution of the first step

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 6.13.

Figure 6.13: 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 6.14.

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 6.14: 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 Trip Monitor blocks, as shown in figure 6.15.

This is a very useful step to compute, because the user learns how to use Mission Modifier block, which is the most important feature of the Pluto Editor, because it allows the programmer to insert his custom code to characterize the application.
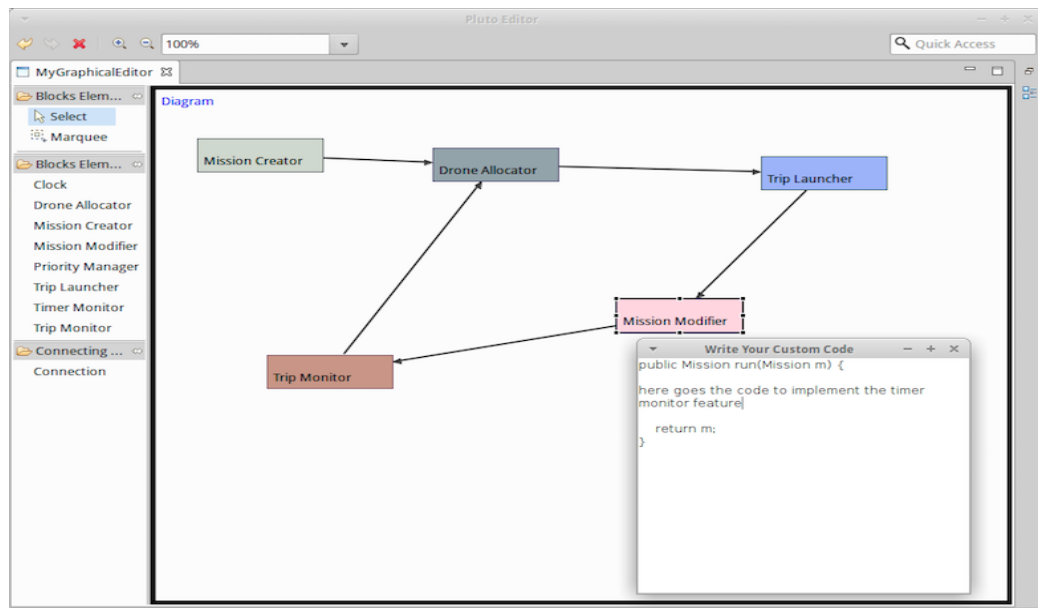
Figure 6.15: Solution of the third step

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

70

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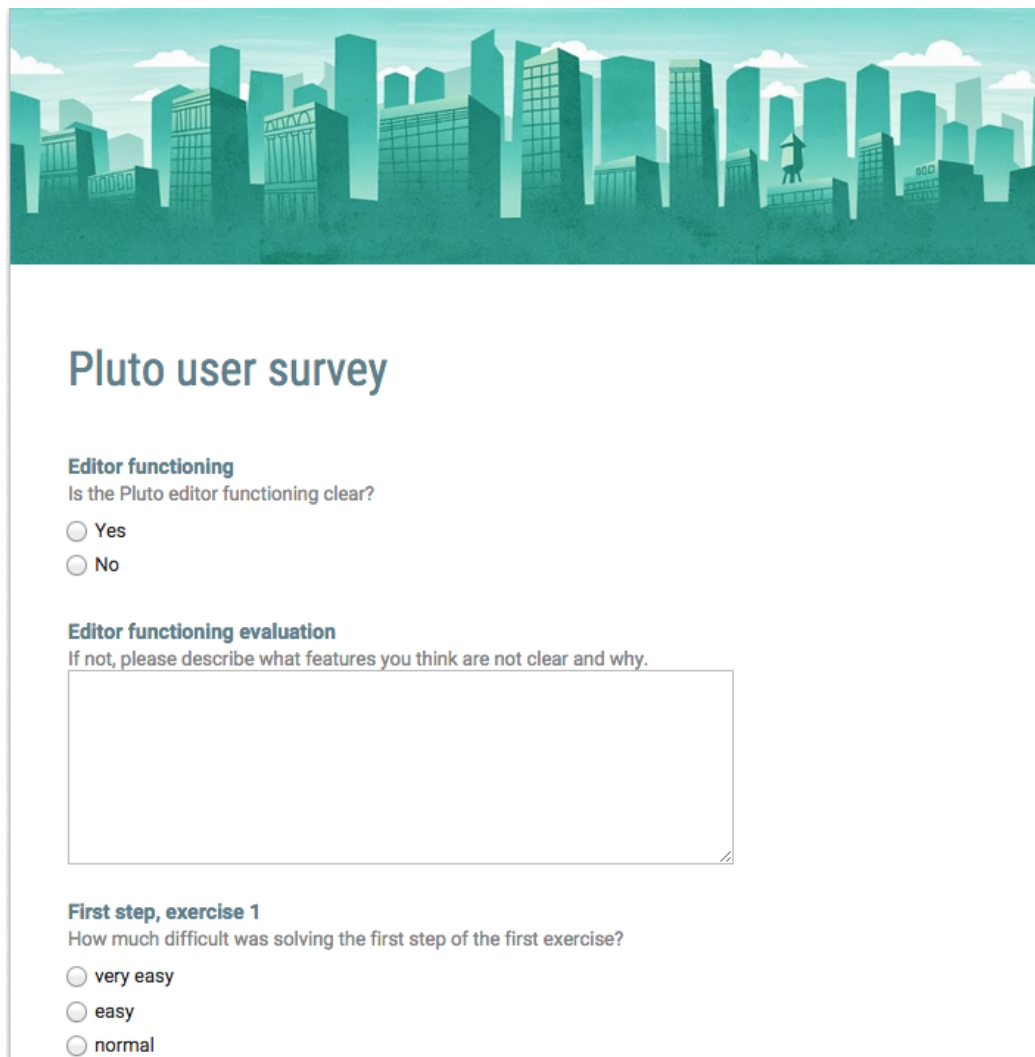
The Crazyflie API is written in Python, so we address to people who knows Python language features.

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, partly showed in figure 6.16, to the users, in order to understand how easy it is to use and which modifications should be applied to improve the user experience.

Figure 6.16: Pluto user survey

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   Final 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 can 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 a table, while the results of the user survey are presented with graphics.

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 Pluto Graphical Editor first and then on Pluto Main Application. To measure these parameters we used a very useful tool called VisualVM, shown in figure 6.17.
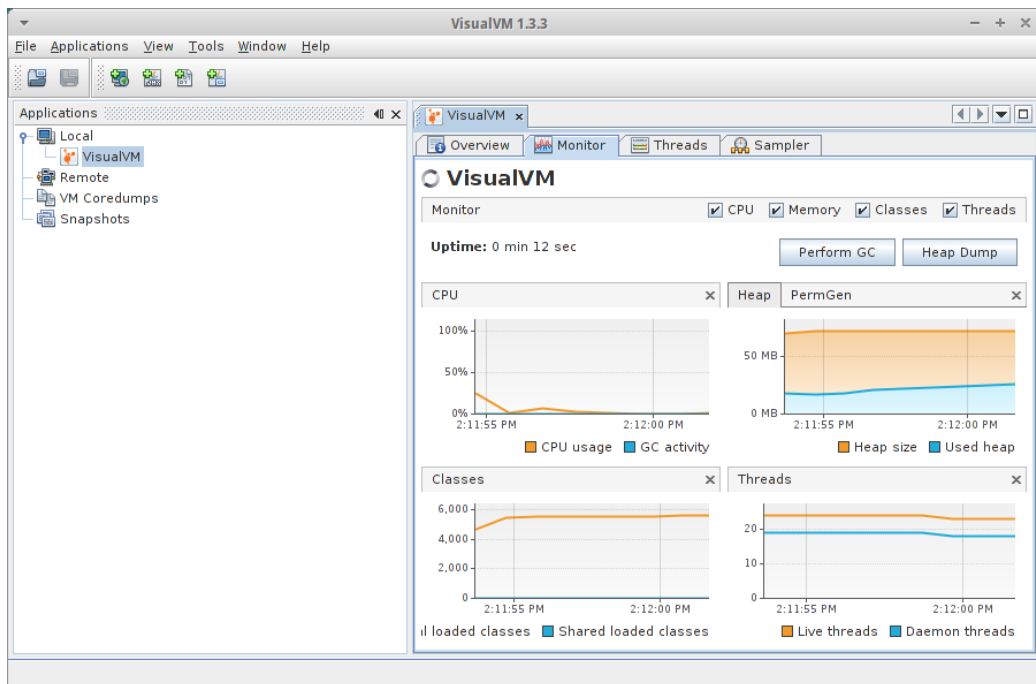

Figure 6.17: VisualVM interface

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. Then, in the Result section 6.3.3, we describe the outcome of the tests for Pluto Graphical Editor and

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 Pluto Framework:

- Lines Of Code (LOC)

- Number of classes attributes

- Number of methods

- Number of methods calls

- ...

6.3.2 Hardware Consumption

The hardware metrics are those parameters measured at run-time, during the execution of the two software, to check if it generates any performance issues, because it could require too much resources to run smooth. These metrics are:

- CPU Load

- Memory Consumption

- Threads Generated

- Threads Zombies (not terminated)

- ...

The profiling of the application were done on a machine with these specs:

- CPU: Intel i7 2640

- RAM: 4GB

- VGA: Nvidia GeForce 610M

- SSD: Kingston 120GB

- OS: Xubuntu 14.04

Concerning the Pluto Graphical Editor, we measured those parameters while creating a lot of blocks and a very complex web of connections, shown in figure 6.18 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. Firstly we fixed the former and we incremented the latter by a step of 5. Then we did the same operation fixing the number of connection and raising the number of blocks.
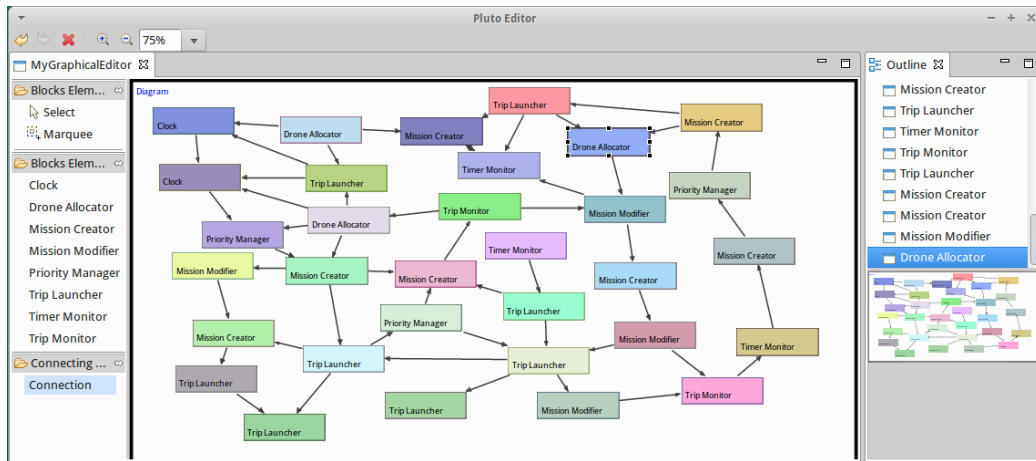

Figure 6.18: Very Complex Diagram Example

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 raised the third step by step. Then we did again this procedure fixing another couple of parameters and varying the left one. In this way, we could evaluate the performance of the Main Application in an accurate way and the result are shown in next section 6.3.3.

6.3.3 Result

For software metrics there will be a table with 2 column and each row will be a metric: LOC, classes, ... The 2 columns are Editor and Main app.
For the Performance part of the Editor there will be 2 graph, one with Connecton

fixed and number of block raising up, the second is inverted. on X there will be the variable parameter and on Y the metrics: cpu, memory,... For the performance part of the Main App, there will be 3 graphs: one with Mission and Trip fixed, then Drones will raise by a step of 5 starting from 1. On the X coord there will be Drones, on the Y coord there will be the metrics: cpu consume, memory usage, threads allocated, each line with a different color. Then the same thing with Mission free, and then with Trip free.

Chapter 7

Conclusions and future works

"Never let your head hang down.
Never give up and sit down and grieve. Find another way.
And don't pray when it rains if you don't pray when the sun shines."

Richard M. Nixon

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.