POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica

Programming Abstractions for Nano-drones Teams

Dipartimento di Elettronica Informazione e Bioingegneria

Relatore: Prof. Luca Mottola
Correlatore: Mikhail Afanasov

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

Anno Accademico 2014-2015

Alle nostre famiglie.

Abstract

Drone-teams programming is rapidly expanding since it allows to automatically perform a lot of useful tasks. Existing systems are able to manage a group of drones and dispatching them in the environment. All these systems deal with outdoor applications, where medium/big sized drones collaboratively perform tasks making use of the Global Positioning System (GPS) to navigate in the space. We want to deal with the indoor context, and no one of the existing system is fully suitable for it. Indeed, the indoor context implies applications with different requirements compared to the outdoor ones: there is need for a small number of drones(5/10), each one performing a different action independently from the others, while in the outdoor environment generally there is need for a large number of drones to perform the same action. We formalize this problem with the concept of *Trip*, that is nothing but a movement of a drone from a point A to a point B at the end of which an action (picture,measurement etc.) is performed. No one of the existing systems provides the concept of Trip. Furthermore the second important goal is to make the system autonomous in the choice of the drone to allocate for each Trip. This means that the user does not need to take this important decision. From a technological point of view, GPS cannot be used in indoor contexts, so we need to find an appropriate indoor localization method. Then indoor contexts imply small areas which are usually full of people and/or obstacles (think of an house context) hence, drones have to be small, in order to avoid crashes. Size limitations result in many problems such as the short battery autonomy and the maximum weight transportable. We propose the Pluto programming framework as a solution to these problems. It consists of two main components: the Graphical Editor and the Main Application. With the former a programmer can build an application by simply connecting blocks. Each block implements a precise functionality, for example there is one that chooses the drones to assign to each sensing task, one that manages the priority of the sensing tasks etc. Then, the developer can generate, from the

graph built with the Pluto Graphical Editor, the source code of the second main component, the Pluto Main Application. The final user uses this generated Pluto Main Application to specify and execute the sensing tasks. We fully evaluated the Pluto programming framework by proposing its use to real testers and asking them for a feedback. Moreover we measured its software and hardware performances and also tried to implement some existing applications with it. After the evaluation we noticed that, even if it has some limits, Pluto is useful to simplify the developing of drone-teams applications.

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

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

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

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

Contents

List of Figures

Chapter 1

Introduction

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 instruments. Often there is a simple abstraction which allows drones navigation: they can be controlled through mobile devices or by setting waypoints from a desktop application. The great deal with drones is that they can greatly extend the capabilities of traditional sensing systems while simultaneously reducing cost.

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. We aim to create a new programming model for the collaboration of nano-drones, in order to extend the support for the developers who want to create new applications in indoor contexts.

The indoor context implies applications with different requirements compared to the outdoor ones: there is need for a small number of drones(5/10), each one performing a different action independently from the others, while in the outdoor environment generally there is need for a large number of drones to perform the same action. We formalize this problem with the concept of *Trip*, that is nothing but a movement of a drone from a point A to a point B at the end of which an action (picture,measurement etc.) is performed. Existing frameworks do not allow the developer to deal with the concept of Trip. We also want to make our framework autonomous while choosing which drone has to physically carry out the Trip.

From a technological point of view, GPS cannot be used in indoor contexts. Furthermore, the indoor context implies moving in small areas which are usually full of people and obstacles, so the drones have to be small, in order to avoid crashes with both human and environmental obstacles. Size limitations result in other problems such as the low battery duration and the maximum transportable weight. So, we need to give a contribution to the actual state of the art in order to derive a new programming system crossing all the previous requirements. Our programming framework has an architecture in which the central brain is independent from the particular navigation API, which means that the system manages the dispatching of drones and their failures independently of the specific navigation algorithm.

1.1 Contribution

Imagine a medical context where the nurses' duty is to deliver, every day at the same hour, the patients' daily medicines. A drone could achieve this task flying room by room and leaving the pills to the relative patient's desk. It would be way better if a group of drones could manage these tasks simultaneously.

In order to create an application able to carry out this task, a developer could program each drone individually, making use of a specific API to command them, and taking care of the complex duty of managing the coordination between them. Our goal is to build a more clever programming framework, where a central brain manages the dispatching of the drones simultaneously and expresses how the drones should behave to accomplish these tasks, managing also the failures due to crashes of the drones and batteries getting empty. Chapter 2 illustrates in details the possible ways to achieve this goal: instead of using a *Drone Oriented* approach, the developer could use either a *Swam Oriented* or a *Team Level* approach. There are several available frameworks such as Karma[1] and Voltron[2], but no one of them is fully suitable for our programming model, shown in section 4.1.

As already said, we deal with the indoor context, that implies applications with different requirements compared to the outdoor ones: there is need for a small number of drones(5/10), each one performing a different action independently from the others, while in the outdoor environment generally there is need for a large

number of drones to perform the same action. We formalize this problem with the concept of *Trip*, that is nothing but a movement of a drone from a point A to a point B at the end of which an action (picture,measurement etc.) is performed. No other framework allows the user to define the idea of a Trip as a movement from point A to point B to accomplish an action. Another fundamental feature of our framework is that the central brain takes care of the important decision to choose which drone to assign to a given Trip, facilitating the work of the programmer in the development of his application. As said, the system manages the failed trips in the same way. For example, when a drone crashes, the system chooses a new drone to complete the interrupted Trip, without any intervention of the programmer and the final user.

We wanted to facilitate as much as possible the work of the programmer, so we decided to create the Graphical Editor, fully described in Section 4.3.1. Thanks to this editor, the programmer can develop its application by drawing functional blocks and then connecting them with arrows. Each block represents a feature: there is a block that assigns a Drone to a Trip, one that sends the drones to the target location, etc. The developer can connect the blocks needed according to the requirements of the specific application. From the drawn graph the programmer can generate the source code of the Main Application mentioned before. This source code is dependent on the graph and adds to the Main Application all the features expressed by the chosen blocks.

In order to define the sensing tasks, we developed the Pluto Main Application, that is the final user interface fully described in Section 4.3.2. It allows the final user to choose the actions to perform and where they must be performed, by simply dragging and dropping the actions on a map. An Action could be the taking of a photo, a measurement of some parameters or a custom task expressed by the developer.

Finally we evaluated the Pluto framework under different points of view, as described in Chapter 6. First of all, we chose some already existing applications and tried to develop them from scratch with Pluto. Then we evaluated the framework usability, proposing some exercises to real testers and asking them for a feedback through a survey. Finally we focused our attention on the software and hardware

metrics such as the code complexity and the CPU consumption. We defined some metrics to evaluate both the users' exercises and the software and hardware metrics. After a detailed analysis of both the metrics results and the answers to the user survey, the results convinced us about Pluto capability to simplify the duties of a developer in implementing a drone application. The whole evaluation process is shown in Chapter 6

1.2 Outline

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

In Chapter 2 there is a description of the actual state of the art in the context of our work. In Sections 2.1, 2.2 and 2.3 we describe the three main existing approaches for drone programming, the "Swarm-level","Drone-level" and "Team-level" respectively, also proposing existing examples for each one of them. We show that no one of these approaches is suitable for our requirements, since we need the concepts of missions and trips. A mission is a list of sensing tasks to be performed sequentially and a trip is a movement from a point A to a point B in the environment to perform an action. Then, in Section 2.4, we describe the dataflow programming method, that we adopted for the Pluto Graphical Editor, providing two existing examples of it. Also in this case, we show that we need a different approach, since we need to re-use blocks and the existing solutions don't allow us to do this.

Chapter 3 is focused on the problems stemming from the indoor context and on the requirements deriving from it. In Section 3.1, we show a motivating example application, in order to better explain the requirements and problems deriving from our work. In Section 3.2 we show the implementation problems deriving from using a Team-level approach for our system, also proposing the solutions to fix them. Finally, in Section 3.3 we show the technological limitations affecting our system, such as the indoor localization and nano-drone batteries problems.

Chapter 4 presents our solution for the research problems described in Chapter 3, the Pluto programming framework. In Section 4.1 we present our programming

model: we show the entities of our model and the relationships between them. We also describe the blocks architecture of the Pluto Programming Framework, which is shown in Section 4.2. In Section 4.2 we describe in details the functionality of the available blocks of the Pluto Graphical Editor, that are the basic elements that the programmer can connect to graphically build an application. In Section 4.3 we describe in details the two components of the Pluto framework: the Graphical Editor, that is used by the programmer to graphically build an application and the Main Application, that is used by the final user to specify the sensing tasks to be performed. The last Section of the Chapter, the 4.5, describes all the steps performed to arrive to the final system, showing all the previously implemented solutions which, once refined, brought us to the development of the Pluto programming framework.

Chapter 5 shows how the designed choices have been implemented technically, describing all the software and tools used for the development of Pluto programming framework. In Section 5.2 we describe the GEF framework, which we used to implement the Pluto Graphical Editor. In Section 5.3 we show the code generation process that creates a Java application from the graph built with the Pluto Graphical Editor. In Section 5.1 we motivate the choice of an Object-Oriented programming model for the Pluto framework. In Section 5.4 we describe the runtime features of Pluto: the parallel architecture and the management of all the needed threads. In Section 5.5 we describe the SWING tool, which we used to develop the Pluto Main Application. Finally, in Section 5.6 we describe the Crazyflie nano-quadcopter, which we used to perform the sensing tasks of our prototype applications.

Chapter 6 starts with an analysis on the applicability of the Pluto framework. In Section 6.1 we describe four already existing applications and three case study, and we discuss on whether they can be developed or not with Pluto. In Section 6.2 we propose two exercises to real testers, in order to test "on the field" the effective usability of Pluto: the first one deals with the Graphical Editor, the second one with the Main Application. Then we propose a third exercise, in which we ask the users to directly use the API of the Crazyflie nano-quadcopter, shown in section 5.6, to make it move from a point A to a point B. We also propose a survey to the users and then present the result in a graphical way, in order to have opinions on the framework and possibly to improve it with the suggestions of the testers. In Section

6.3 we measure the software and hardware consumption metrics required by Pluto, in order to evaluate the effective impact of Pluto on an ordinary computing machine.

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

Chapter 2

State of the art

In the last years, the drone technology is expanding rapidly;especially the aerial drones, which are often used as toys controlled by joystick or to make aerial videos through mounted cameras. There exist also aquatic and terrestrial drones, which can be used for many applications. For example, through aquatic drones the submarine network infrastructure[3] 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.

After presenting the three main approaches for programming drones, we show some examples of data-flow

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[4] 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.

To control the Create, the developer should send a sequence of commands through the serial interface. Each command starts with a one-byte opcode and some of them must be followed by data bytes. For example to control the wheel actuators the developer should send a command like this:

[137] [Velocity high byte] [Velocity low byte] [Radius high byte] [Radius low byte]

It takes four data bytes, interpreted as two 16-bit signed values using two's

complement. The first two bytes specify the average velocity of the drive wheels in millimeters per second (mm/s), with the high byte being sent first. The next two bytes specify the radius in millimeters at which Create will turn.

Figure 2.1: The iRobot Create

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.

2.2 Swarm-level approach

The Swarm-level approach[5] 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)[6], which provides a Publish/Subscribe coordination layer for decentralized computations, as shown in section 2.2.1; Karma[1], which lets programmers specify modes of operation for the swarm, such as "Monitor" or "Pollinate"(as shown in section 2.2.2); and Proto[7], which lets programmers specify actions in space and time(as shown in section 2.2.3).

2.2.1 Robot Operating System

ROS[6] 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.

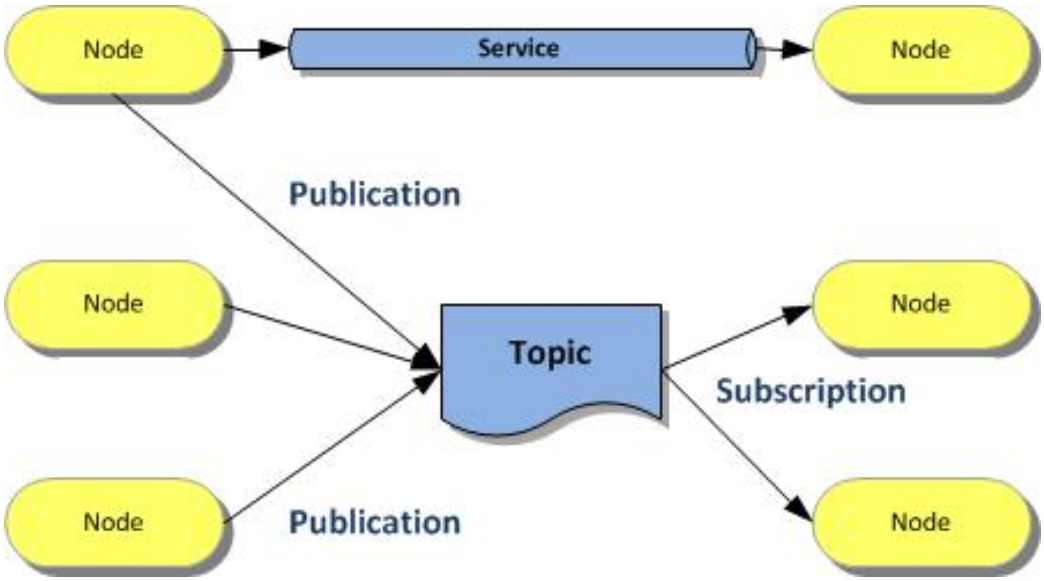

Figure 2.2: ROS communication layer functioning

2.2.2 Karma

Karma[1] 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[1] runtime at the hive is composed by functional blocks, as shown in fig. 2.3:

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

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

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

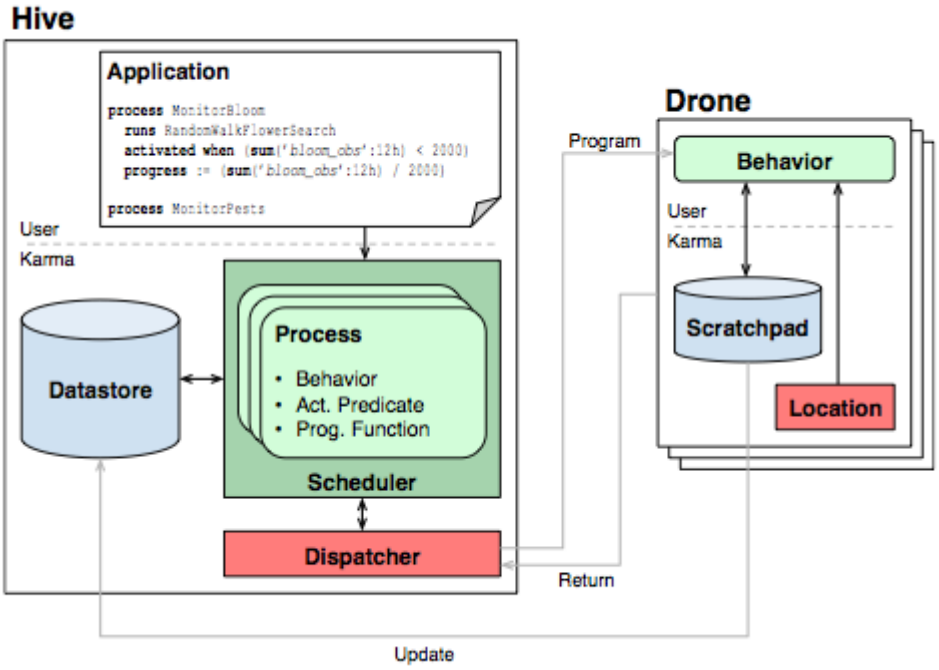

Figure 2.3: The basic schema of Karma

2.2.3 Proto

The amorphous medium abstraction[8] 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[8] 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.

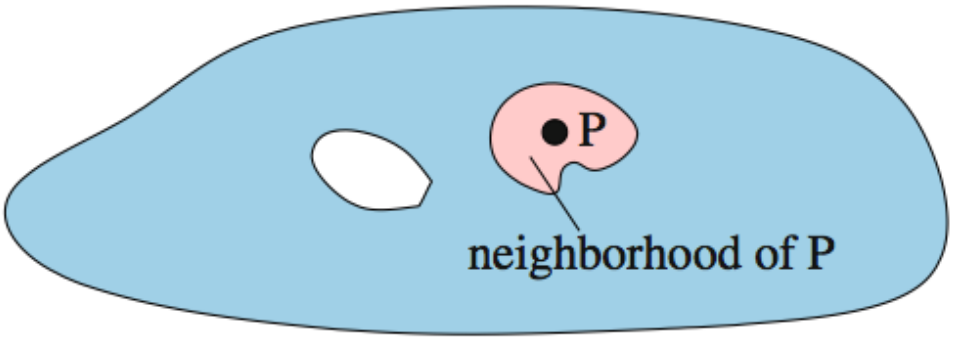

Figure 2.4: The amorphous medium abstraction

The Proto[7] language uses the amorphous medium abstraction[8] 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[7] language to mobile devices, such as drones in a swarm, the amorphous medium[8] must be extended with the concept of *density*; indeed for the vast majority of mobile applications, it is important to distribute drones depending

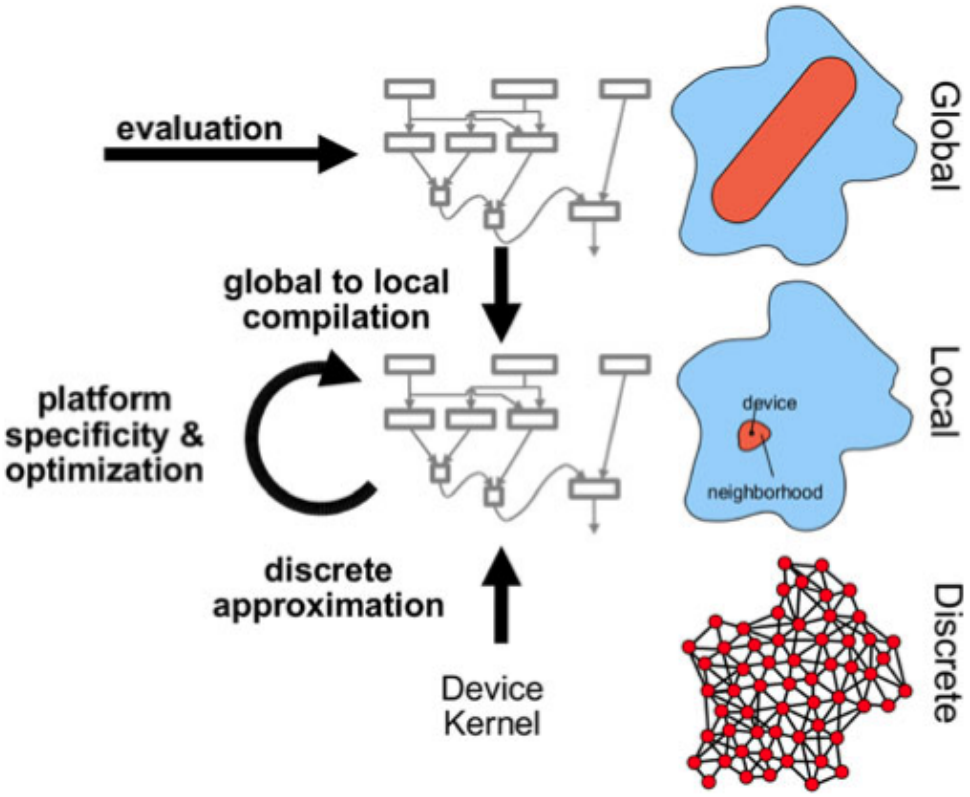

Figure 2.5: Proto: problem decomposition

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[2], 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[2], a system containing a set of programming constructs to explore the notion of team-level drone programming. Voltron's[2] 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[2] 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 Data-flow programming

Dataflow Programming is a useful programming paradigm that allows a developer to represent the execution model of his application through a directed graph. The flow of the data processed during the execution streams between all nodes. Each node is a functional block that accept data as input, manages it performing its tasks and then drive it forward to the next block.
The final dataflow application is nothing but a composition of these active blocks with at least one initial and one ending blocks, connected by directed arrows. A block is connected to another when it has a dependency on the result of the manipulation of the data from that block. Values are propagated after they are processed from all the dependent blocks triggering their execution.

This paradigm present some limits of expression. The main one is that each implementation of this paradigm is specific for the context where it is used. This means that there are no general frameworks that can be used in more than one context. This is why we created our own dataflow editor, shown in section 4.3.1.

2.4.1 Business Process Modeling Notation

Business Process Modeling Notation (BPMN) is an example of the power of the dataflow programming paradigm. Its primary goal is to help business users providing a readily understandable notation, filling the gap between the business process design and the final process implementation.

BPMN defines a diagram that contains graphical models of the business process operations. A Business Process Model, is nothing but a graph where nodes represent graphical models representing the operations, and edges are the flow controls that define their order of performance. An example is shown in figure 2.7.

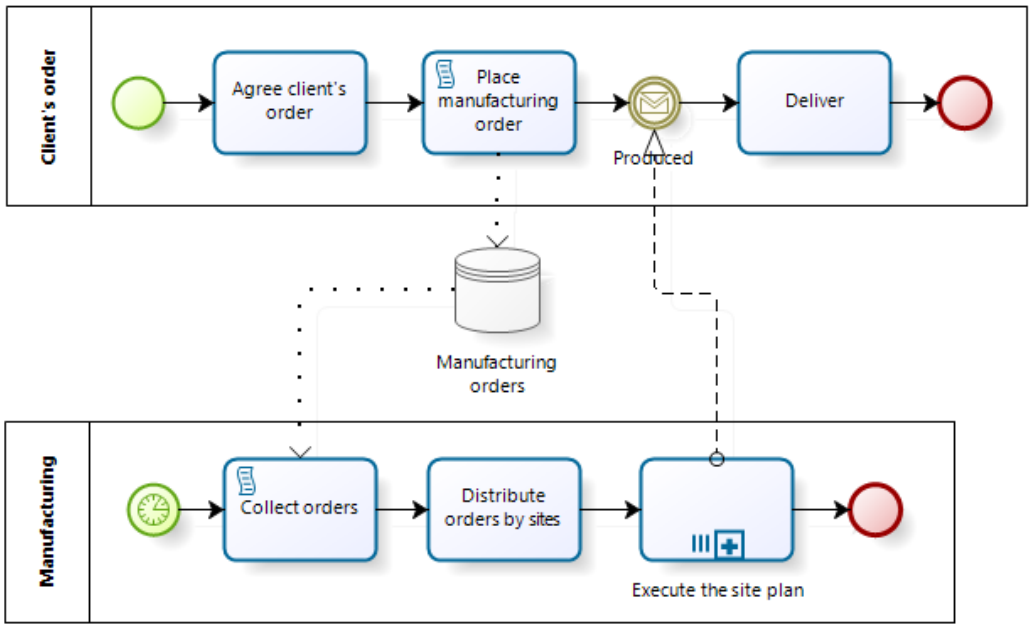

Figure 2.7: Example of a BPMN diagram

There are different kinds of elements in a diagram:

- *Flow Objects*: they can be Events (circles), Activities (rounded rectangles) or a Gateway (diamonds)
- *Connecting Objects*: they can represent a Sequence Flow (solid line), a Message (dashed line) or an Association (dotted lines)
- *Swimlanes*: they can be a Pool representing the Actor of the process, or a Lane that is a sub-partition within a Pool
- *Artifacts*: they are useful to extend the basic notation adding new ways to describes context based information

2.4.2 Node-RED

Node-RED is an example of the dataflow paradigm applied to the *Internet of Things* world. It is a browser-based editor that let the developer wire together different nodes with directed edges. Each node provides a different feature or a different management of the input data. All features are web-based functionality, such as receiving an HTTP request or a *Javascript* snippet. In this context, a diagram represents the back-end of the web application and When the graph is completed, the user can deploy it with a single-click in the runtime environment. The light-weight runtime is built on *Node.js*, taking full advantage of its event-driven, non-blocking model. This makes it ideal to run at the edge of the network on low-cost hardware as well as in the cloud.

In figure 2.8 we show an example web application:

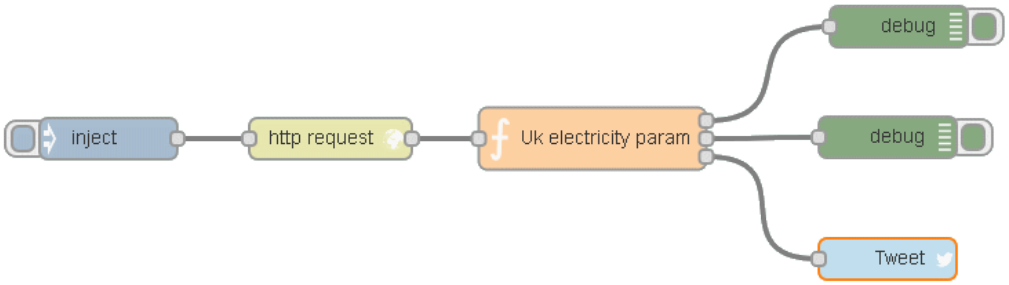

Figure 2.8: Example of a nodeRED application

The yellow node makes a GET request to the UK electric company website.

The response data follow the edges and pass through the orange node, which lets the developer write its custom Javascript code inside it. In this case, the custom code takes the payload data in input and put them on the third output, formatting them in readable way. Through the first two outputs, the block sends only debug strings that go to the green nodes. In the end the blue node will send a tweet containing the data received as input.

In this Chapter we have described the actual state of the art in the field of drone programming and dataflow programming. We needed to overcome the limits of the actual state of the art by performing some modifications to the existing solutions, which we show in Section 3.2.

Chapter 3

Indoor applications using autonomous drones

In this Chapter we show all the problems we had to face in the development of our programming framework. We first show a concrete application we wanted to develop, and then the problems deriving from its development. No one of the existing drone programming approaches, shown in Chapter 2, is fully suitable for our problem. We chose a Team-level programming approach, described in Section 2.3, but we had to perform some modifications on it, as we explain in Section 3.2. There are also some technological limitations, such as the lack of a stable indoor localization system and the short duration of nano-drones battery, which we explain in Section 3.3.

3.1 Motivating scenario

In order to concretely show all the limitations and problems encountered in the development of our system, we start this Chapter describing a concrete scenario. We want to develop an application to assist elders to take their medicines, for example in a hospital context. A team of nano-drones could help the nurses to deliver the daily medicines to the patients at the right time of the day. A representation of the behavior of the application, which we named Drugs Distribution(DD), is shown in figure 3.1:

- the nurses prepare the little boxes with each patient's daily medicine

- each drone, at the right time of the day, brings the box to its assigned patient
- after carrying out their action, the drones return to the start location

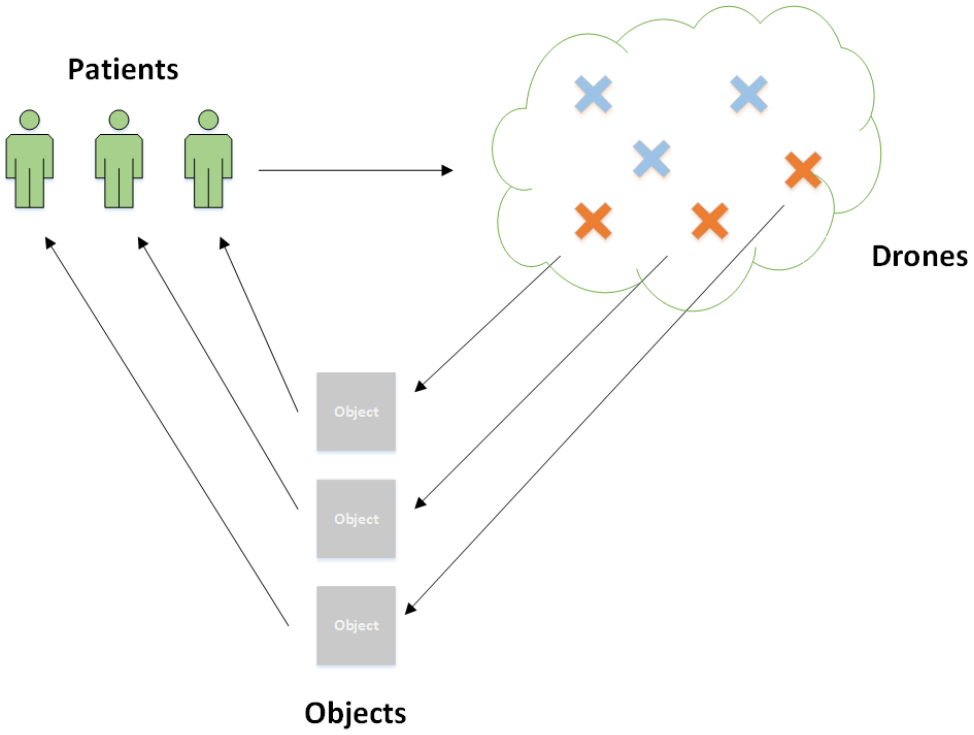

Figure 3.1: The basic functioning of the Drugs distribution application

The development of this application made us to face some problems, both in the implementation of the system and in the technological lacks, which we describe in the next two Sections.

3.2 Drone programming

Since the approaches for programming drones we described in chapter 2 are designed in a way that is impossible to describe an application through the concepts of Mission and Trip, we give our contribution to the state of the art creating a new framework based on these entities. A *Mission* is nothing but a list of sensing tasks to be performed sequentially in the environment. Each one of these sensing tasks is a *Trip*, that is a movement from a point A to a point B to perform an Action.

Neither the Drone-level nor the swarm-level approaches, described in Sections 2.1 and 2.2 respectively, are suitable for our goal. The former because we do not want the user to deal with the coding of each drone separately with an external API. The latter because we want to avoid the complexity to create a communication network protocol between drones and because it would be difficult to maintain the status of the missions and trips entities among the swarm. Moreover we also need to address time and space constraints, which cannot be expressed with this approach.

The most suitable approach for our framework is the Team-level model, described in Section 2.3, but we need to apply some modifications to it, in order to make it suitable for our work. Using a Team level approach 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 directly affect the development of our DD application, described in Section 3.1. So we have to modify the Team-level approach in order to make each drone deliver the box of medicine to its assigned patient, independently from the other drones. So we need the concept of *Trip*. A *Trip* is nothing but a movement from a point A to a point B in the environment to perform an action. In this way, we can tell each drone to go to the precise location of its assigned patient, making the Trip of each Drone independent from the others. The concept of *Trip* is a fundamental feature of our model, and it is fully described in Section 4.1.

Another very important feature of our system is the transparent dispatching of drones: the central brain takes care of assigning the drones to the sensing tasks to be performed, managing also the drones failures, without involving the programmer.

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 get a single point of failure, so, if the central brain breaks then the whole system crashes. This problem can be fixed or at least weakened by applying some dependable systems methods, improving reliability of the central brain, reducing its 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, described in section 2.1, has the problem that the programmer has to manage individually the drone's movements and the interactions with other drones: he must code a list of instructions and commands that the drone will perform sequentially. This can only be achieved with the exploiting of specific API of each drones. 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 discharge 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 balance the load between drones in a dynamic way. Is clear that the drone-level approach is most suitable for applications involving only a few drones.

On the other hand, the Swarm-level approach, described in section 2.2, 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 should follow. It is important to underline that, in swarm-level approach, there is no possibility to have a shared state between drones; each one execute the actions specified by the programmer on his own local state. This means that this approach is very easy to scale up adding new drones, but it's not suitable for applications that require the drones to explicitly coordinate.

Regarding the dataflow programming, we need a new framework that allows the user to design the behavior of the central brain taking care of the missions execution, from the beginning to the end. This modeling tool helps the developer to add the features needed by the application simply drawing the proper nodes in the dataflow graph. The BPMN and Node-RED dataflow models, described in Sections 2.4.1 and 2.4.2 respectively, are too general for our system, since they allow to model almost every kind of project. They offer a great number of components, but we only need basic components for our editor, like rectangles and arrows. So we decided to develop our own dataflow model, offering only the functionality and components needed for our programming framework. This part of the project is fully described in Section 4.3.1.

3.3 Implementation challenges

The DD application, shown in section 3.1, makes drone bring medicines to the elders in an hospital, so in an indoor context. One big problem is that there is not a stable localization method for the indoor environment. Besides of localization problem, indoor context also leads to the limits of the size of the drone. As a result, programmers constantly confront with a limited battery resource and a small weigh the drone can carry out. These problems, as well as their possible solutions, are described in the following Section.

3.3.1 Indoor localization

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

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

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

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

For example, *Magnetic positioning*[9] is based on the iron inside buildings that create local variations in the Earth's magnetic field. Modern smartphones can use their magnetometers to sense these variations in order to map indoor locations.

With *Inertial measurements*[10] pedestrians can carry an inertial measurement unit(IMU) by measuring steps indirectly or in a foot mounted approach, referring

to maps or additional sensors to constrain the sensor drift encountered with inertial navigation.

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

WPS[11] is based on measuring the intensity of the received signal(RSS) together with the technique of *fingerprinting*. In computer science, a fingerprinting algorithm is a procedure that maps an arbitrarily large data item to a much shorter bit string, its fingerprint, that uniquely identifies the original data for all practical purposes just as human fingerprints uniquely identify people for practical purposes. The accuracy of WPS improves with the increase of the number of positions entered in the database. WPS is subjected to fluctuations in the signal, that can increase errors and inaccuracies in the path of the user.

Bluetooth[12] 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.

It is important to underline that we made the choice to decouple the system working logic from the *Navigation System*. The navigation system is the module of our central brain that makes use of indoor localization API, providing to the central brain a way to control the drones with accurate coordinates. In this way all previously described technologies are suitable with our system, on condition that the developer provides the proper API.

3.3.2 Drones and Objects size limitation

Indoor contexts imply small areas which are usually full of people and obstacles 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 8 minutes, having a recharge time of about 20/30 minutes. It limits the programmer in developing applications which require the drones to perform their actions in this limited amount of time.

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 taken, 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.

Chapter 4

Programming with Pluto

We already said our goal is to perform user-defined sensing tasks using nano-drones, in an indoor context. We chose the Team Level approach, which we described in Section 2.3, applying some modifications to it, to manage the sensing tasks execution. 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 related task and check that each drone takes its mission to the end with a successful status. In this chapter we describe the whole programming model of our system as a solution for the problems shown in Chapter 3.

4.1 Programming model

Through our programming model, the user is able to specify a list of sensing tasks that drones can perform.
Each one of the sensing tasks is represented by the *Trip* entity that is the representation of a physical movement of the drone from a source location to a target one. The Trip always ends with an *Action*, that is the physical representation of the task. For example the drone can move an *Item*, take a photo or measure the temperature. Then, we decided to create a container entity called *Mission* that includes the list of tasks that drones should accomplish. This means that each Mission contains a

list of Trips.
It is important to underline that, inside each Mission entity, the trips are executed sequentially, one by one. The missions instead are executed in parallel.

So, to summarize, we identified the following entities:

- **Mission**: a list of trips(sensing tasks) to be performed sequentially.
- **Trip**: a Drone movement from a point A to a point B to perform an Action.
- **Drone**: the physical executor of the Trip and the Action.
- **Action**: the type of task to be performed by the Drone: "Take Photo", "Pick Item", "Release Item", "Measure", etc.
- **Item**: the entity that represents the object carried by the Drone, only used in case of "Pick Item" and "Release Item" Actions.

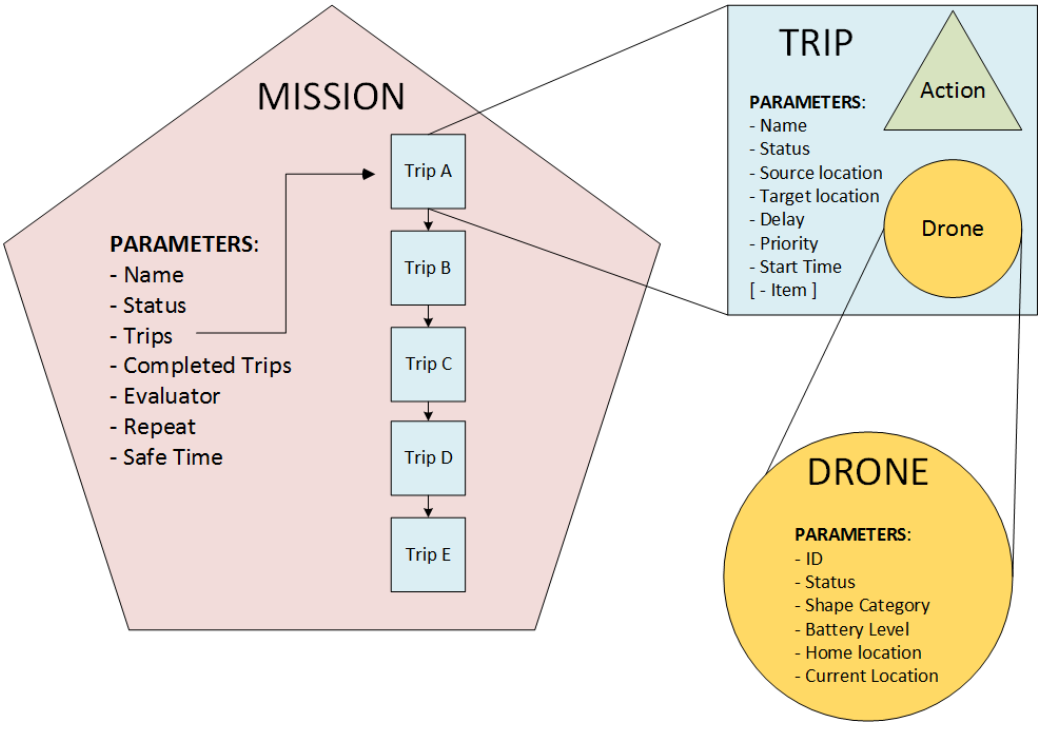

Figure 4.1: Relationship among model entities

Mission

As shown in figure 4.1, the Mission entity, in addition to the set of Trips, contains other important attributes that describe the mission itself:

- **Name**: the name given by the user while creating the mission
- **Status**: describes how the Mission is being executed. For example, it is set to "RUNNING" while the drone is carrying out a Trip, or "FAILED" when a trip fails after a crash of the related drone
- **Trips**: the list of trips to be executed sequentially
- **Completed Trips**: the trips completed successfully
- **Evaluator**: the reference to the Evaluator entity. It is optional, see the end of this Section for a brief description
- **Repeat**: states if the Mission must be repeated after its completion. It is optional, see the Mission Repeater description in Section 4.2
- **Safe Time**: contains the maximum amount of time within each trip inside the Mission must be completed. It is optional, see the Timer Monitor description in Section 4.2

Trip

Regarding the Trip entity, it contains several parameters too and furthermore it contains the Drone and the Action entities references.

- **Name**: the name that identifies the Trip
- **Status**: describes how the Trip is being executed. For example, it is set to "EXECUTING" while the drone is carrying out the Trip, or "FAILED" when a crash happens while a drone is executing this Trip
- **Source Location**: the starting point of the movement represented by the Trip
- **Target Location**: the target point of the movement represented by the trip

- **Delay**: the amount of time that the trip must wait before starting. It is optional, see the Clock description in Section 4.2
- **Priority**: states the priority level of the Trip. It is optional, see the Priority Manager description in Section 4.2
- **Start Time**: contains the timestamp of the moment in which the drone assigned to this Trip starts the flight.
- **Item**: the Item entity reference representing the item that the assigned drone bring. It is optional, based on the Action entity.
- **Action**: the Action entity reference that describe the task to accomplish after the target location is reached.
- **Drone**: the Drone entity reference that represent the physical drone assigned to the Trip.

Drone

Another important entity in the programming model is the Drone. It represents the physical drone that performs the assigned Trip. It has some important parameters:

- **ID**: it is a unique ID to distinguish each Drone
- **Status**: states if the drone is "FREE" and available for a trip or if it is "BUSY" because it is flying.
- **Shape Category**: this parameter lets the system know if a drone is able to accomplish certain Action or move certain kind of Items.
- **Battery Level**: contains the battery level of the Drone.
- **Home Location**: contains the coordinates of the home location.
- **Current Location**: contains the coordinates of the current location of the Drone. The system uses this parameter to localize the drone when needed.

Action

The Action entity describes the tasks that the drones must perform at the end of their respective trips. We decided to create four basic actions that explain themselves: "Measure", "Take photo", "Pick item" and "Release Item". Moreover we added a "Custom Action" which enables the developer to define a personal implementation of a new Action depending on the application requisites. The developer can add this implementation after the code generation phase, explained in Section 5.3, by writing his custom algorithm directly in the code.

Evaluator

In figure 4.1, inside the Mission object, there is a reference to the Evaluator entity. The Evaluator is the entity whose duty is to evaluate the outcomes of the actions performed inside a Mission.
This means that some trips can be completed successfully but the Action done at the end can return a bad result. Therefore these actions should be repeated and so the related trips. This feature is provided by a functional block described in next Section 4.2.
We decided to decouple this mechanism from our system, so that the developer is able to include its personal implementation of the evaluation algorithm in our framework. In Section 5.3 we describe this process in a more detailed way.

4.2 Functional blocks

As already said in Section 3.2, we decided to create a new dataflow programming framework that allows the user to design the behavior of the system while taking care of the missions execution.

This framework consists in a modeling tool that allows the developer to add the application requested features by drawing the proper nodes in an editor area.
Each diagram created with this tool is made of many functional blocks, each one including a particular logic. The user can select the blocks needed for his particular application and connect them through simple connection elements. The graphical

editor is fully described in Section 4.3.1.

Here we provide a detailed description of each functional block available in the editor:

Mission Creator block

Input: a list of Trips
Output: a Mission object

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

Clock block

Input: a Mission object
Output: a Mission object

The Clock block checks the delay attribute (Figure 4.1) of the next Trip to be executed in the Mission. If it is greater than zero, it makes the Trip waiting for that amount of time, and finally returns the Mission Object. Of course delaying the execution of the next Trip, stops the Mission execution too, because as said in Section 4.1, the trips are executed sequentially.
If the programmer puts the Clock block in the graph, the Main Application will ask the user, during the Mission definition phase, the amount of time of the delay. This block could be used in an application where the user wants to measure the temperatures in a location every 10 minutes. He has to set a delay of 10 minutes in every Trip, so that they will wait for that time before starting.
Usually, this block is put between the Mission Creator and the Drone Allocator blocks, in order to wait for the delay time before allocating a drone to the Trip.

Drone Allocator block

Input: a Mission object
Output: a Mission object

The Drone Allocator block allocates the proper Drone to the next Trip of the Mission taken as input. It bases its choice on the availability of the drones and their capability to perform the desired action.
This block can be implemented with different policies. Indeed, this choice is nothing but an optimization problem and the developer can choose a custom algorithm. We intentionally decoupled this problem from the system implementation, in a way that it is possible for the developer to change the policy as he wishes.
This block is usually put before the Trip Launcher, because an assigned drone is essential for the Trip execution. For example it could be put between the Clock and the Trip Launcher blocks in order to assign a Drone to the next Trip after the delay.

Trip Launcher block

Input: a Mission object
Output: a Mission object

The Trip Launcher block takes the next Trip to be performed from the Mission, checks if it has an allocated Drone and then starts its execution. The assigned drone fly to the target location and execute the defined Action. Usually this block is drawn right after the Drone Allocator and is fundamental in order to start the execution of the trips of the Mission.

Trip Monitor block

Input: a Mission object
Output: a Mission object

The Trip Monitor block continuously checks the status of the Trip that is running in that moment. As said in Section 4.1, the system executes the trips sequentially, so only one trip at a time is executing and is monitored by this block. We need to monitor the running trips because a flying drone could crash or its

battery could become empty before ending the Trip. In this way we can guarantee the correct completion of the Mission even if a failure happens.
Depending on whether the Trip is failed or completed, this block changes its status parameter in the appropriate way. Of course, the Mission that contains that Trip will change its status,accordingly. The Trip Monitor is put after the Trip Launcher because it needs to monitor a Trip that has already started its execution.

Mission Repeater block

Input: a Mission object
Output: a Mission object

This block takes as input only a completed Mission object.
So when a completed Mission arrive to the Mission Repeater, the block verifies if the *Repeat* attribute (Figure 4.1) of the Mission is true. if so, this block moves the list of the completed trips into the list of trips to be executed. Then resets the status of the Mission itself and the status of all the trips to execute again.
This block could be useful if a Mission should be repeated over and over. For example in a surveillance application it lets the drones monitor the neighborhood without stopping when the mission ends.
In order to emulate this behavior without the block, the user should create a new identical Mission every time the previous one ends.
This block is usually put after the Trip Monitor and before the Drone Allocator, because a Mission can be completed only after it pass through the Trip Monitor and the trips' status must be set to "WAITING" before the allocation of the drones.

Mission Evaluator block

Input: a Mission object
Output: a Mission object

The Mission Evaluator block, as the Mission Repeater, takes as input only a completed Mission. Its task is to evaluate all the actions done by the drones at the end of the trips. Since the Mission arrive to this block only at the end of its lifecycle, all the trips are already executed and all the actions are already performed.

The evaluation consists in invoking the *Evaluator* entity which is referenced inside the Mission object (Figure 4.1).
If the evaluation returns a failed result, it means that some actions should be repeated and then the related trips too. The block finds these trips and put them in the list of trips to execute. After that, it changes the status of the Mission because it is not Completed anymore since it has still some trips to complete.
This block could be used in an application that apply the photogrammetry technique in order to get an orthophoto of a geographical location. Therefore the actions would be taking photos in different coordinates and the evaluation would consist in trying to get the orthophoto from those simple photos. If some photos are not good enough the evaluation would fail and the related trips and actions will be repeated.
Usually this block is put after the Trip Monitor and before the Drone Allocator for the same reasons explained in the Mission Repeater block.

Mission Modifier block

Input: a Mission object

Output: a Mission object

The Mission Modifier block allows the programmer to express new features not implemented yet by other functional blocks. It allows the developer to create a brand new block with a specific feature.
This feature must be expressed with a code snippet that the developer can write directly in the editor.
For example imagine a programmer that needs a new feature that change some parameters of the Mission after the assignment of the Drone, but before the Trip starts. He could insert the Mission Modifier block between the Drone Allocator and the Trip Launcher blocks. Then he could open the window, shown in figure 4.2 by clicking on the proper menu entry called "Write Custom Code".
Inside this window, the programmer can write the code snippet that describes the features requested by the new functional block. So following the example, here he will change the parameters of Mission.
In the end, we strongly recommend to rename the Mission Modifier block with a meaningful name that describes the implemented new feature.
This block can be put in every point of the Pluto Editor graph, depending on the

particular feature it provides. The inserted code will be executed by the system when the Mission object reach this block, following the graph flow.

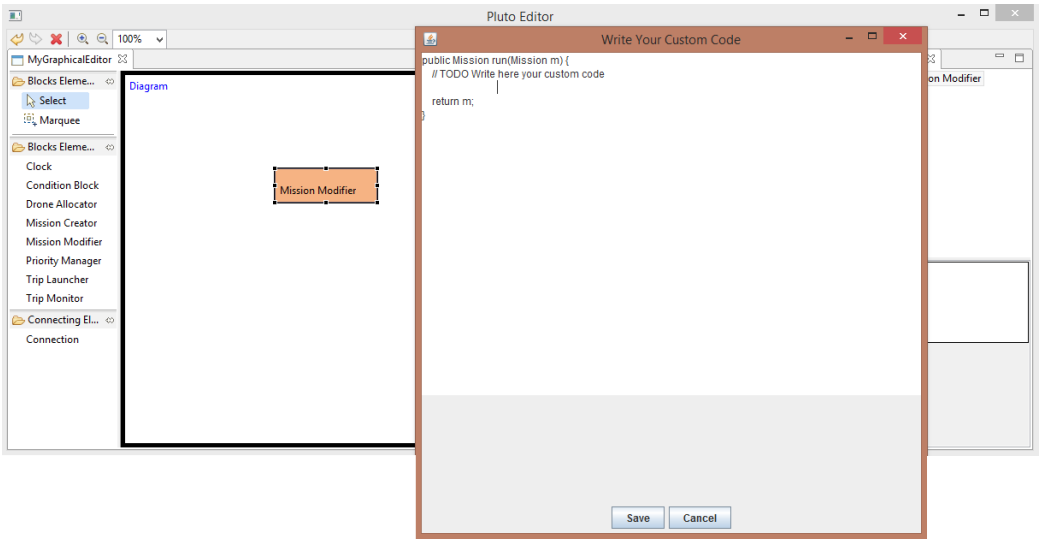

Figure 4.2: The MissionModifier block

PriorityManager block

Input: a Mission object

Output: a Mission object

This block accepts as input only Failed missions. Then the Priority Manager takes the first Trip in the list and increments its priority. After that it sends the same Mission object as an output to the next blocks.
This block is useful in order to not stop the execution flow of a Mission in case a Trip fails. Normally, without this feature, if the Trip Monitor finds out that the monitored Trip has failed, it sets the Mission to "FAILED" too and stops the execution flow.
Instead this block raises the priority of the failed Trip and changes the status of the Mission from "FAILED" to "STANDBY", as if the failure has never happened.
Then to restart the execution flow of the Mission, we need a connection from the Priority Manager to the Drone Allocator.

Timer Monitor block

Input: a Mission object

Output: a Mission object

The Timer Monitor block is an accessory block of the Trip Monitor. It adds a time constraint check to the Trip Monitor supervision. It IS useless to use a Timer Monitor without the Trip Monitor, they should be used in parallel.
This means that while the Trip Monitor monitors the Trip execution, the Timer Monitor supervises the same Trip execution, ensuring that the execution time will not exceed the amount of time set in the "Safe Time" Mission's parameter (Figure 4.1).
As for the Trip Monitor, the developer should put the Timer Monitor after the Trip Launcher. Therefore the Trip Launcher has at least two output connections, one that goes to the Trip Monitor and the other one that goes to the Timer Monitor.
Adding this block in parallel with the Trip Monitor we are cloning the Mission object which goes into the two blocks at the same time. This is why we need the Gate blocks described further.
There are several applications that could request the feature introduced by the

Timer Monitor. Indeed it can be used to consider a Trip as failed if its execution exceeds the Safe Time amount. For example we could consider a drone as crashed if the Trip takes more than 20 minutes to complete.

Gate FIFO block

Input: a Mission object
Output: a Mission object

The GateFIFO block is used when two or more blocks works in parallel, and only one instance of the cloned Mission object must propagates. This block must be put right after these parallel blocks in order to reunite the cloned objects.
For example when the developer inserts in the graph either the Trip Monitor and the Timer Monitor in parallel. In this case the Mission object will be cloned and the two block will receive the same Mission object. Then the GateFIFO connected after them to both the blocks will bring back together the Mission instances.
The GateFIFO usually has more than one incoming connections and it propagates only the first Mission object that comes from one of them. That's why the FIFO acronym is used, since the first Mission instance that arrive is the only one that propagates in the graph.

Gate Funnel block

Input: a Mission object
Output: a Mission object

This block is similar to the GateFIFO, but its implementation logic is different. It waits for a Mission instance coming from each incoming connections. Only after all of them have come, then it will merge them in one single instance and propagates it. For example if before this gate, there are 4 blocks in parallel, the propagation of the Mission is activated only when all the 4 instances arrive.

In the figure 4.3, we show an example application, which contains most of the above described blocks. The pentagons represent the mission object and the different colors stand for the status of the Mission in that particular flow position.

For example after the Mission Creator every Mission is always *Unexecuted*. The Start and End blocks states the beginning and the ending of the mission flow.

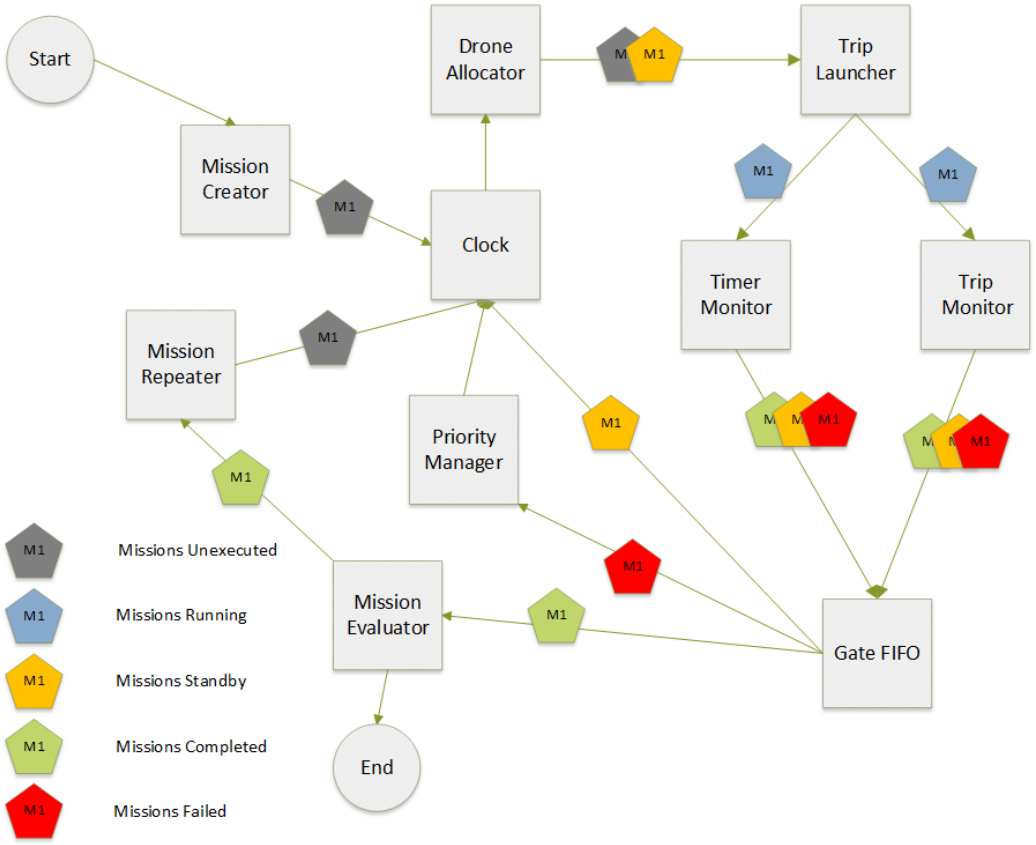

Figure 4.3: An example Pluto application

4.3 Toolchain

The 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 tasks and to start

their execution. As you can see in figure 4.4, the Pluto Graphical Editor lets the developer to creates a scenario based on the Team Level approach as we show in Section 4.3.1. After that, the Pluto Main Application is generated based on the diagram created in the previous step. Then the developer can personalize some feature adding his personal code so that the final user needs only to insert the sensing tasks and wait for their accomplishment.

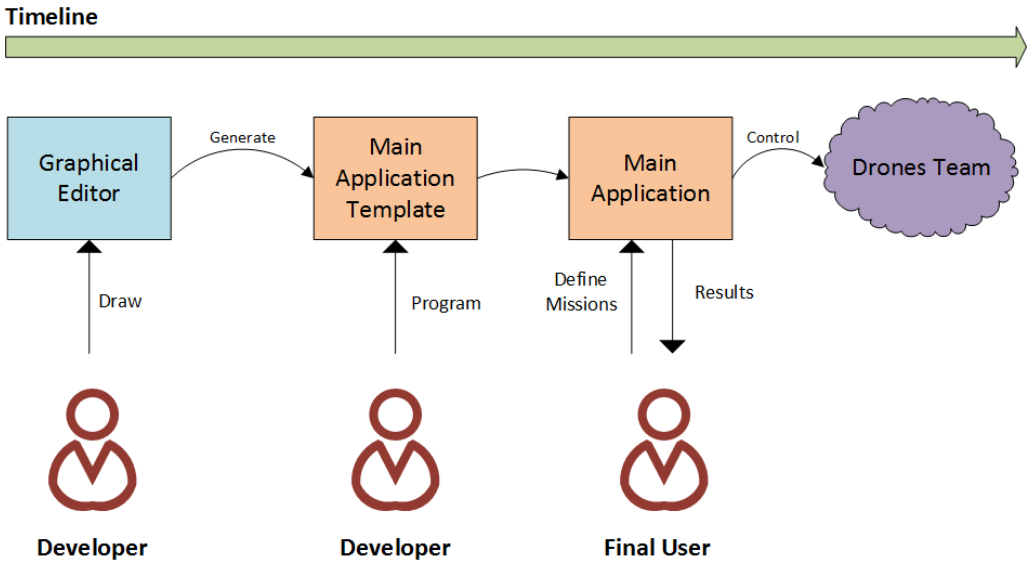

Figure 4.4: Working with the Pluto framework

4.3.1 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 functional blocks, each one with a predefined and implemented logic. When the Editor starts, it shows three main sections: the Palette (letter A in figure 4.5) that contains all the tools available to create a fully functional diagram; the Editor space (letter B in figure 4.5) where the user can move, link and manage all the created entities; last but not least is the Outline (letter C in figure 4.5) 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, as explained in section 4.2.

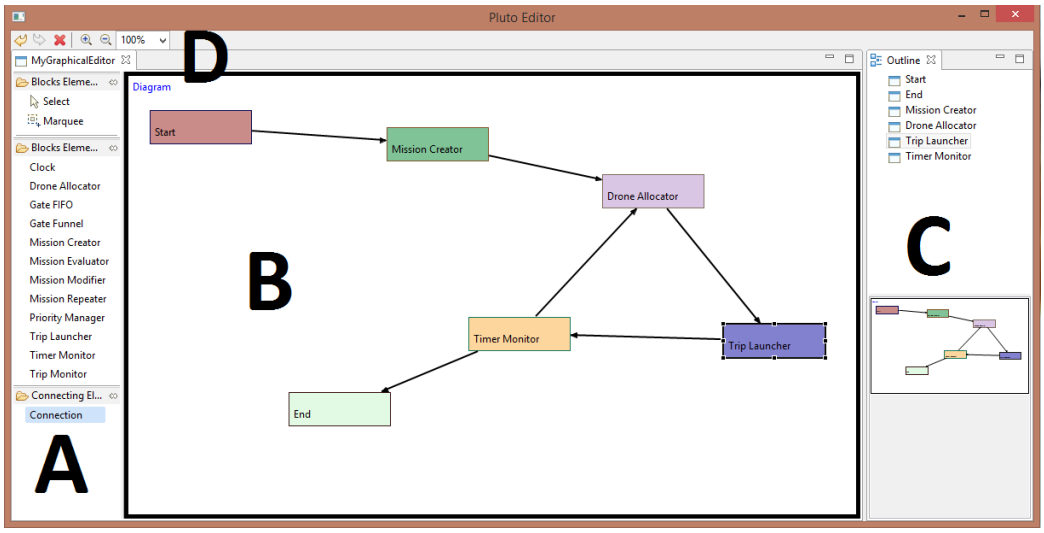

Figure 4.5: Pluto Graphical Editor interface

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 the user can connect blocks with each other using the Connection tool in the Palette section.
The Connection is a directed arch that defines the direction of the execution flow in the graph. This means that the Mission object traveling among the functional blocks can only move in the direction pointed out by the arrow.

Apart from the standard functionality, such as Undo, Save, and Load, the Context Menu provides a command to generate the source-code of the Main Application, based on the designed diagram. The Toolbar provides Undo/Redo, Delete, and Magnify commands (letter D in in figure 4.5).

To better understand the 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, in the editor area, 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 the figure 4.5 represents a possible diagram that can handle the example described in Section 3.1 of the Drug Distribution application.
In this case the graph includes only the basic features: the creation of the mission, the drone assignation, the trips starting and their monitoring. It would be possible to add other blocks in order to define more features for the application.

Besides simplicity, our editor is very flexible as well, since it provides the 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", as explained in Section 4.2.

4.3.2 Pluto Main Application

The Pluto Main Application is the final application that acts as a Ground Control Station, managing all the drones and the missions. In this Section we explain how it works and how can be used.
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.

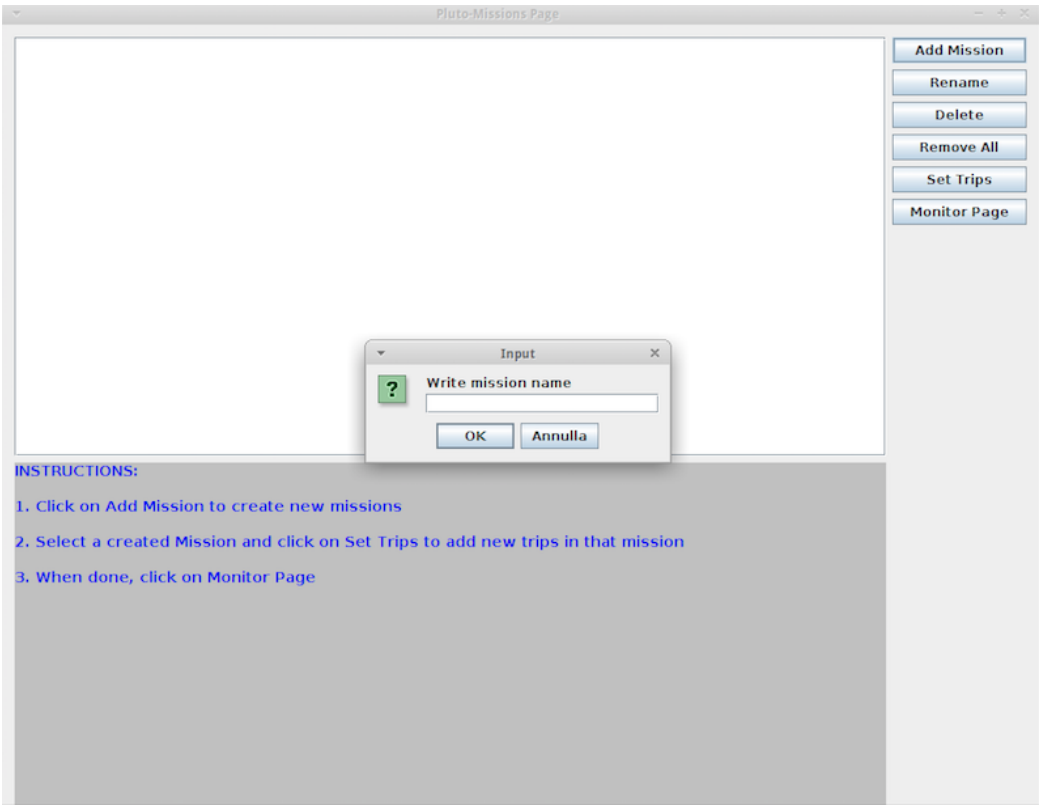

Figure 4.6: Mission Page interface

After that a Mission entity is created, but by now it doesn't contain any information about the task to do. 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 4.7, and the user can add new Trip entities to the related Mission. As explained in Section 4.1, 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.
In order to add a new Trip the user may drag and drop the desired Action from the upper list (letter A in figure 4.7) to the map displayed below (letter B in figure 4.7). The added trips are shown on the right in a proper list (letter C in figure 4.7).

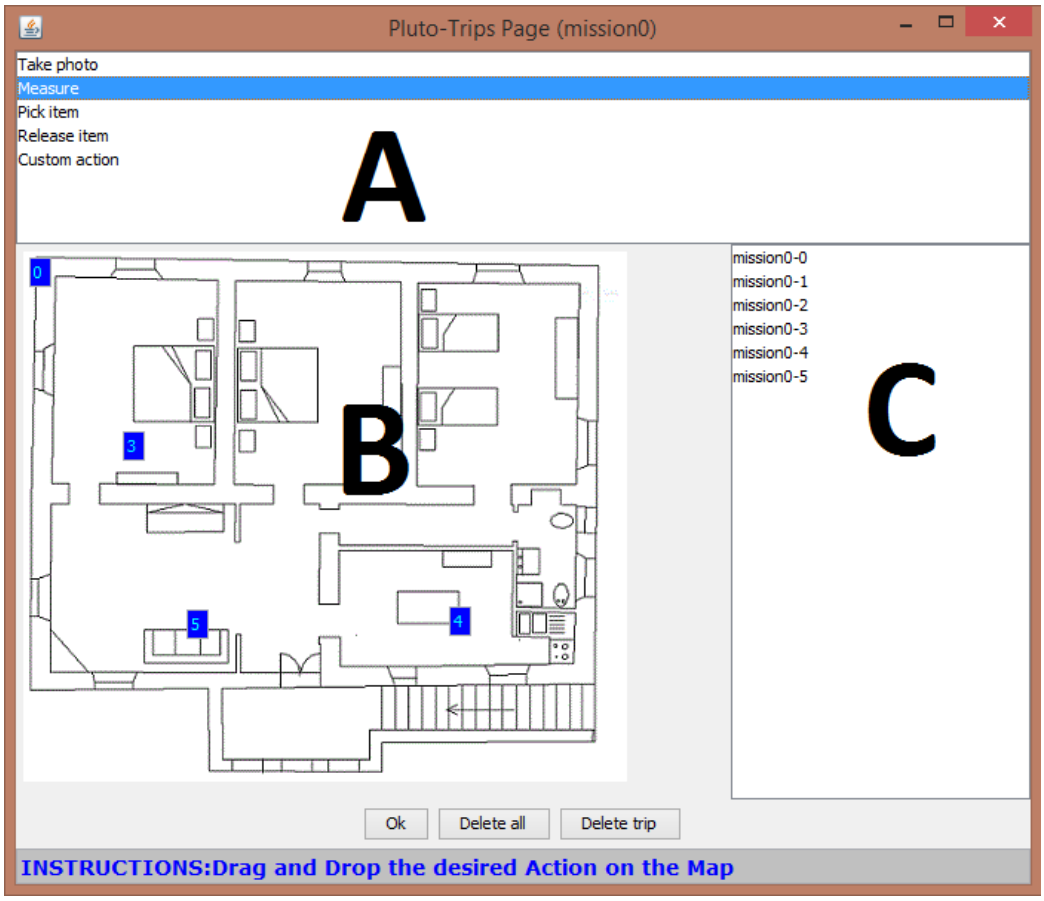

Figure 4.7: Trips Page interface

So after all the Trips are added to the Mission, the user can close the Trips page and then is able to create more Mission entities.
Finally he can pass to the Monitor Page with the corresponding button. The Monitor Page, shown in figure 4.8, is the window where the user can obtain information about the running missions, at run-time.
On the top, there is a table (letter A in figure 4.8) 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 (letter B in figure 4.7) where log messages are printed during the execution of each missions. In this way the user can obtain run-time information about the status of the entire system.

Of course the Start button will start the execution of the created missions, while

the Stop button will prompt the user to a choice: "RTL" or "Land". The first is the Return To Launch and it 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.

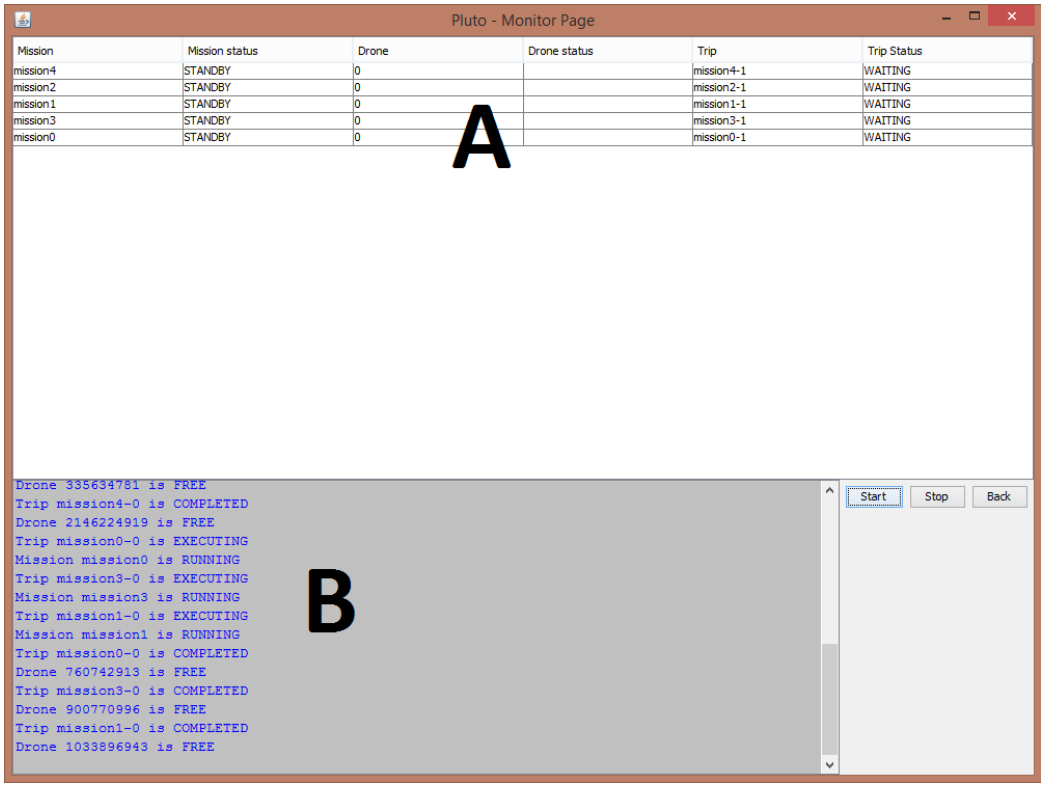

Figure 4.8: Monitor Page interface

4.4 Navigation System

An important component inside the Main Application is the Navigation System. More precisely, it is the conjunction point between the Main Application and the drones team.
This component makes use of the chosen localization API to obtain precise coordinates for each drone. Then it sends commands to them accordingly to their trips information.

These API depends on the technologies chosen to localize the drones in the indoor context, described in Chapter 3.

We decided to decouple the implementation of the Navigation System from the Main Applicaiton so that the developer is free to use his preferred technology. This can be achieved by adding the chosen API in the Navigation System component without modifying other parts of the Main Application.
This choice derived from the conclusions obtained in Chapter 3, where we discussed about some possible ways to enable the indoor localization.

The Navigation System, as said, is the internal component that directly communicates with the drones team. This means that besides the localization API, it makes use of specific drone libraries too. In Section 5.6 we will give a brief explanation about the Crazyflie quadcopters API.

4.5 Design Choices

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.

4.5.1 Solution without Trip entity

In the version precedent to the final solution presented in Section 4.1, we did not have a concept of Trip, and the Mission was the main concept the whole model was based on. Figure 4.9 shows this in the particular case of the Timer feature, which contains also a "Switch source-target" block. This block was in charge to make the Drone perform also the return journey, from the destination to the home location. This is done inside the same Trip entity, simply switching its *sourceLocation* and *targetLocation* attributes. Later this block was eliminated, since we decided to decouple the two journeys. It is sufficient to create a new Trip entity for the return journey.

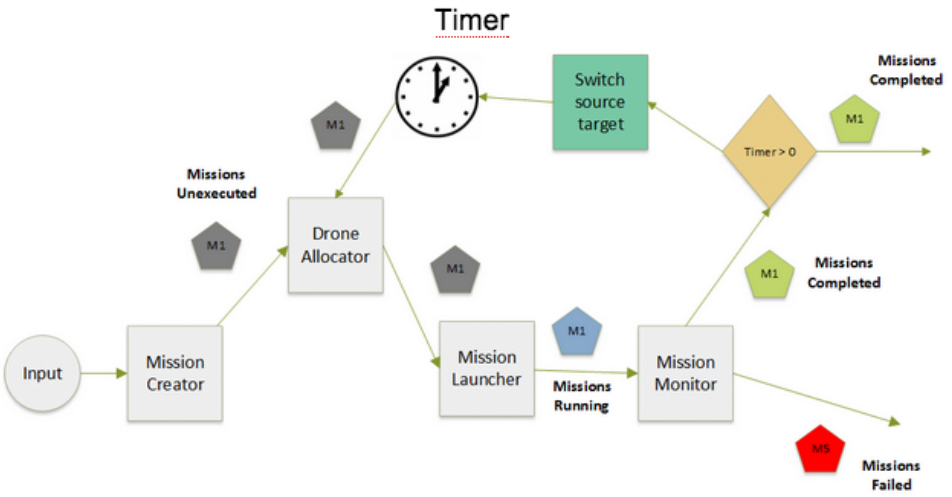

Figure 4.9: Solution without the Trip concept

After analyzing the model, we realized that we needed the concept of Trip, because the final user must have control on the single Trip of a drone, in order to decide which action the drone must perform, and to have an opportunity to control the Trip attributes such as delay, stop, or delete, without deleting or stop the whole Mission. With this solution it was not possible, because having only the entire Mission to manage, the user cannot 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.

4.5.2 Solution without the DroneAllocator and MissionModifier

This solution made use of the "Drone Updater" block instead of the DroneAllocator. This block managed the assignment of a Drone to a Mission(there was not the Trip concept yet), but only under certain conditions. Indeed the "Drone Updater" was used only if the system had to assign another drone to a Mission, for example because of a failure of the precedent assigned Drone. This happened because the MissionCreator managed the first assignment of a Drone to the Mission, so we did not need the DroneUpdater for the first assignment of a Drone to a Mission. Furthermore, there was not the MissionModifier block, but only the

PriorityManager case, shown in figure 4.10, so the programmer could not insert his custom code in the application.

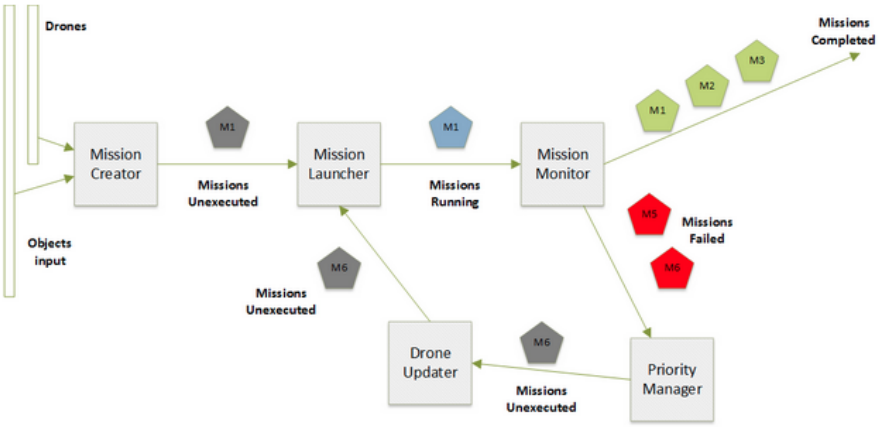

Figure 4.10: Solution without the MissionModifier block

We decided to create the DroneAllocator block because we needed to separate the creation of a Mission Object from the assignment of a Drone to it. So, the MissionCreator should only create a Mission entity and the DroneAllocator should take care of the drones dispatching. In this way we could also remove the "Drone Updater" block, because now we have a specific block which manage only the assignment of drones, so there is no more need to distinguish between the first assignment of a Drone to a Mission and the "special assignment" in case of a failure. We also decided to create a MissionModifier block in which the user can put his own code to customize the application.

Chapter 5

Implementation

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

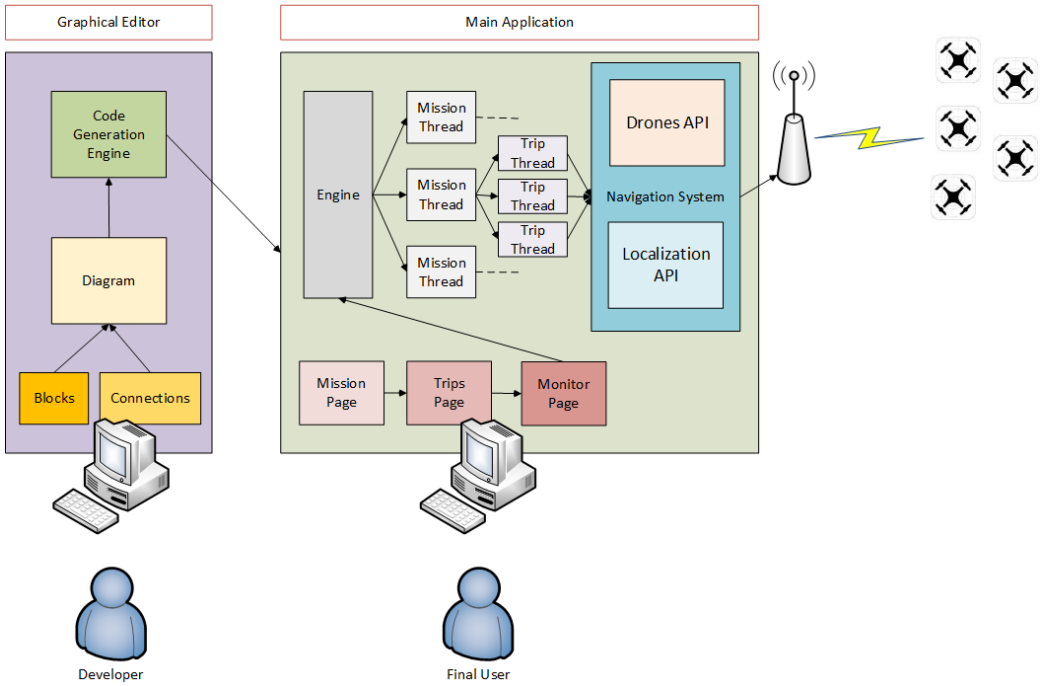

Figure 5.1: Pluto architecture representation

5.1 Object-oriented approach

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

We decided to adopt a Model View Controller(MVC) approach. The central component of MVC, the model, captures the behavior of the application in terms of its problem domain, independent of the user interface. The model directly manages the data, logic and rules of the application. A view can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible. The third part, the controller, accepts input and converts it to commands for the model or view.

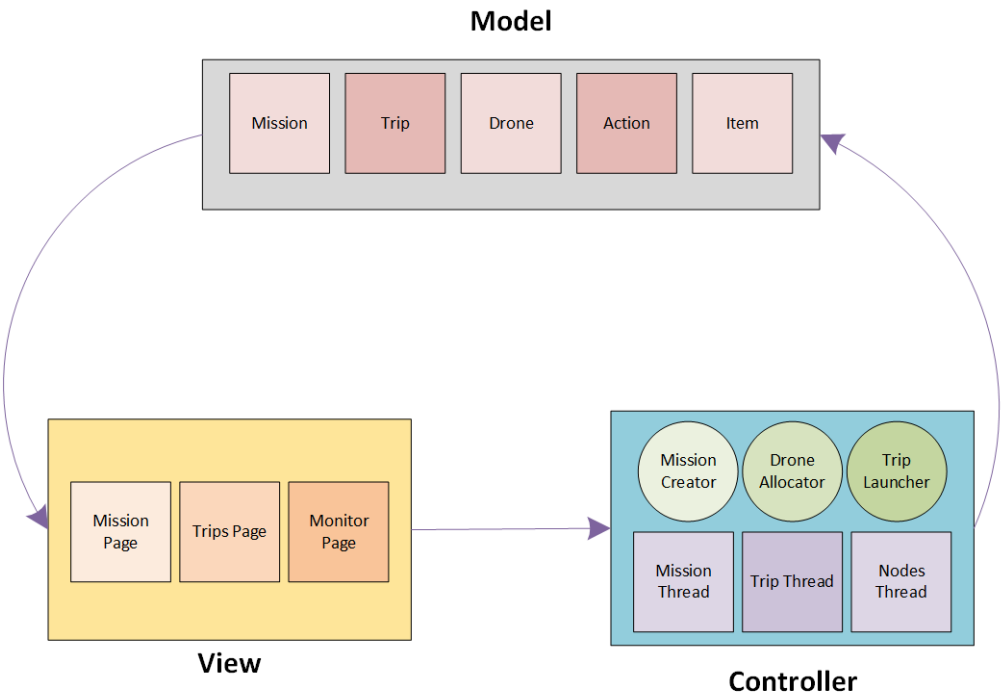

Figure 5.2: MVC design applied to the Main Application

As shown in figure 5.2, the model part contains all the Java classes of the entities shown in Section 4.1. There is a class for the Mission, one for the Trip etc. The controller part contains the Java classes of all the blocks of the Graphical editor shown in Section 4.2. The controller part also deals with the threads management needed for the execution of both the missions and trips. The thread structure is shown in section 5.4. The view part contains the Java classes of the three pages of the Main application shown in Section 4.3.2.

5.2 Graphical editor

In order to create the Graphical Editor, described in section 4.3.1, we decided to use the GEF (Graphical Editing Framework) project. This framework is a Java technology and it is part of the Eclipse framework developed by IBM.

It gives developers a full solution for the graphical modeling of a Java object model, and it can be used in conjunction with other technologies such as EMF (Eclipse Modeling Framework) or GMF (Graphical Modeling Framework), to enable the creation of a complete graphical modeling suite. This means that the Pluto Graphical Editor has been developed as an Eclipse Plugin, so the developer has to install the Eclipse IDE in order to exploit the editor.

First of all, we created the Java classes of all the blocks. Each class contains the code implementation of the corresponding block, since each block perform a specific functionality. We defined each block as a rectangle figure, then we added the connection entity in order to enable links between them. All these entities are children of a main container class that represents the diagram itself that is simply a container for the graph. Figure 5.3 shows these components: A is the diagram entity, the container of the graph; B is the connection entity, through which the user connects the blocks; C is the block entity.

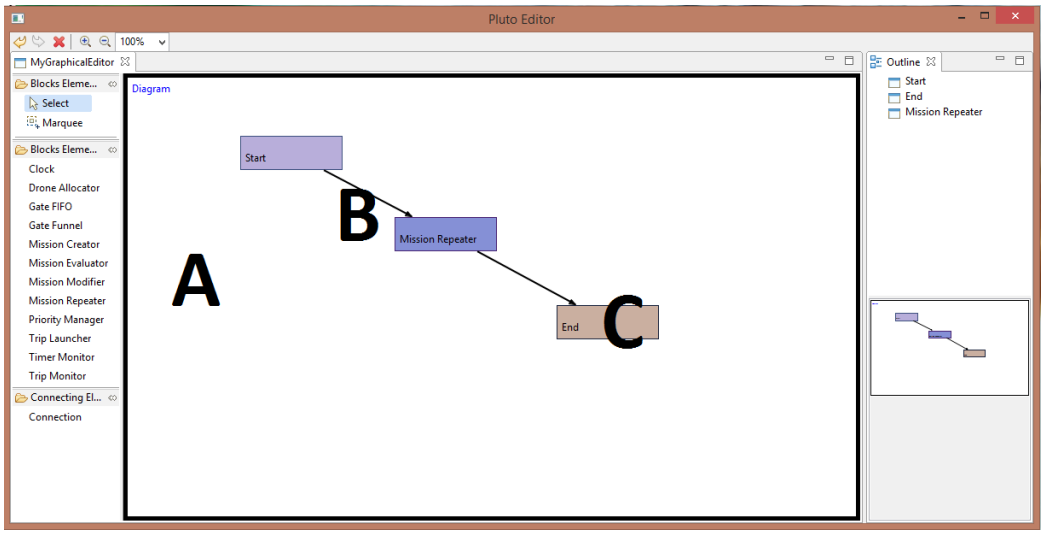

Figure 5.3: Graphical entities in the editor

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

After the user draws the desired graph he can choose to generate the final code of the Main Application, through the apposite voice in the context menu. See section 5.3 for further details.

5.3 Code generation

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

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

The main issue in the generation process was to understand how to generate the code from a general diagram. Potentially, a developer could draw a very complex graph with lot of blocks and connections between them. At first, we focused on graph exploration methods, but we immediately noticed that they were too complex.

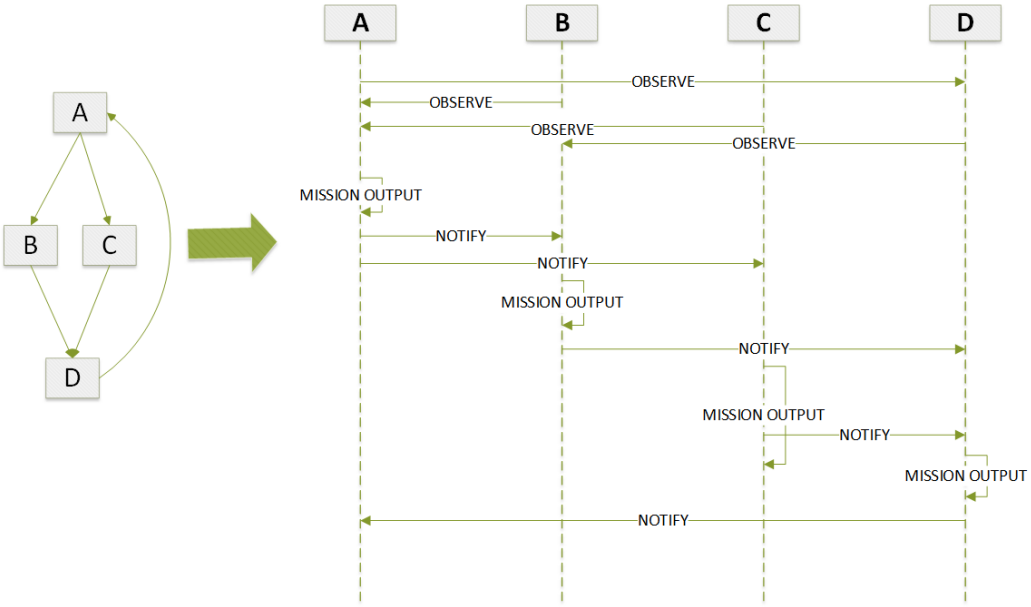

Figure 5.4: Observer design pattern example

So, we decided to adopt a *Publish-Subscribe* design, making use of the *Observer* pattern. This mechanism let us describe the execution flow of very complex diagrams, besides the more simple ones.

The *Observer* pattern consists in the declaration of some elements as *observers* and of other entities as *observable*. When an observable object change its status, it sends a notification to all its observer entities. These observers will react according to the change of the observable object.

In Figure 5.4 there is a sequence diagram that describes how the observer pattern work with a simple diagram with four blocks (A, B, C and D).
The "Observe" message in the sequence is the declaration of a block that wants to observe another block. The "Notify" message represents the notification that a block sends to its observers when it changes its status, or in this case when the blocks ends the management of the Mission object.

In our specific case, we made each declared block of the graphical editor both Observer and Observable. This means that each block observes another block that comes before it, but at the same time, it is observed by other blocks coming after it.

The change of status consists in the output of the Mission object. When a block ends to perform its operations, it notifies all its observers passing them the Mission entity.
For example, in figure 5.4 the A block observes the D one; the B and C blocks observe the A and the D observes the B and C both.
So, for example, when the A block ends to modify the Mission object it sends it to the B and C blocks at the same time, which are its observers.

The Editor includes in itself a template model of the Main Application, in which almost all the classes are ready to be executed.
However, this template contains several tags that the generation process will replace with specific lines of code, depending on the drawn graph.
The generation process consists in the search for these tags inside the template application. The tags are:

- **<dec>**: This tag is the placeholder for the code part where the generator engine puts the declaration and the initialization of the entities represented by each blocks.
- **<exe>**: This tag is the placeholder for the code part where the generation process declares the Observer pattern. Here the system defines the observers of each blocks depending on the connections in the diagram.
- **<tDelay>**: This tag is the placeholder for a boolean attribute. If the diagram includes the Clock block, this tag will be replaced with a "true" value in order to make the Main Application to ask the user for a delay during the Trip definition.
- **<mRep>**: This tag is the placeholder for a boolean attribute. If the diagram includes the Mission Repeater block, this tag will be replaced with a "true" value in order to make the Main Application to ask the user if he wants the Mission to repeat itself.
- **<tSafe>**: This tag is the placeholder for a boolean attribute. If the diagram includes the Timer Monitor block, this tag will be replaced with a "true" value in order to make the Main Application to ask the user for a maximum safe time during the Mission creation.

- **<tPrt>**: This tag is the placeholder for a boolean attribute. If the diagram includes the Priority Manager block, this tag will be replaced with a "true" value in order to make the Main Application to ask the user for a priority during the Trip definition.
- **<num>**: This tag is a placeholder for an integer attribute. If the diagram includes the GateFIFO or GateFunnel blocks, this tag will be replaced by the number of incoming connections of the related Gate block.
- **<act>**: This tag is not replaced by the generation process but we need them in order to warn the developer where to insert his Custom Action code.
- **<eval>**: This tag is not replaced by the generation process but we need them in order to warn the developer where to insert his custom Evaluator algorithm.

5.4 Runtime Management

The Main Application manages the mission execution with a parallel programming architecture, as shown in figure 5.5. Indeed when the user starts the execution, the system launches each Mission in a new thread, in order to guarantee a reliable parallel execution.

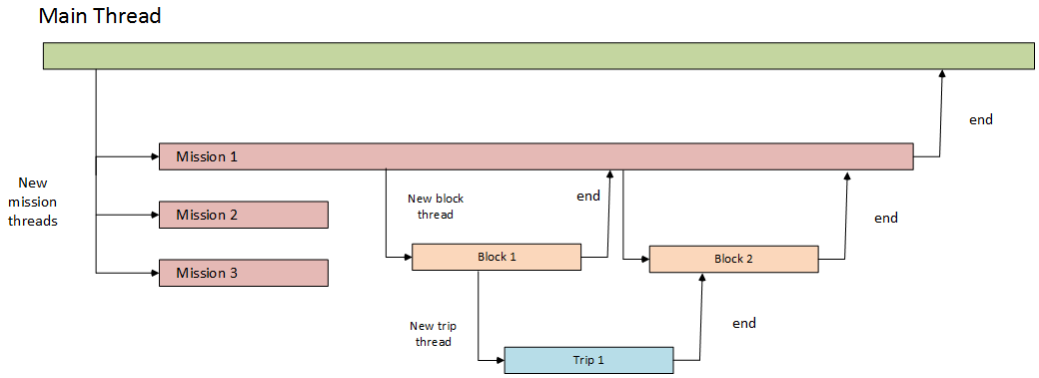

Figure 5.5: Example of thread concurrency

Then each mission starts its flow among the blocks, thanks to the Observer design pattern, described in Section 5.3. When the mission enters in a new block,

the application launches a new thread, in order to run the mission management code of the block. We need this new thread because there is the possibility that two or more parallel blocks have to manage the same mission entity at the same time.

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

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

It is important to underline that we don't have any synchronization problem among the various threads. Indeed, there are no dependencies between missions, since each one of them is executed independently from the others.
Inside each Mission entity the trips are executed sequentially: one trip can start its execution only if the precedent trip in the list has been completed.
Given this independence between missions, the system could dispatch them in a multiple machines cluster. In this way each Mission would run on a different environment maximizing the performance and reducing the load on the single machine.

5.5 User interface

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

We used the Swing framework to develop the view part of the MVC pattern shown in section 5.1, We needed to develop the three pages of the Pluto Main Application, already described in section 4.3.2, and we knew that Swing provides all the components that we wanted to put in them. Indeed, since we used it for

the development of many academic projects, we noticed that it allows to build graphical interface in a very fast and easy way.

As an illustrative example, figure 5.6 shows the structure of the Trips Page:

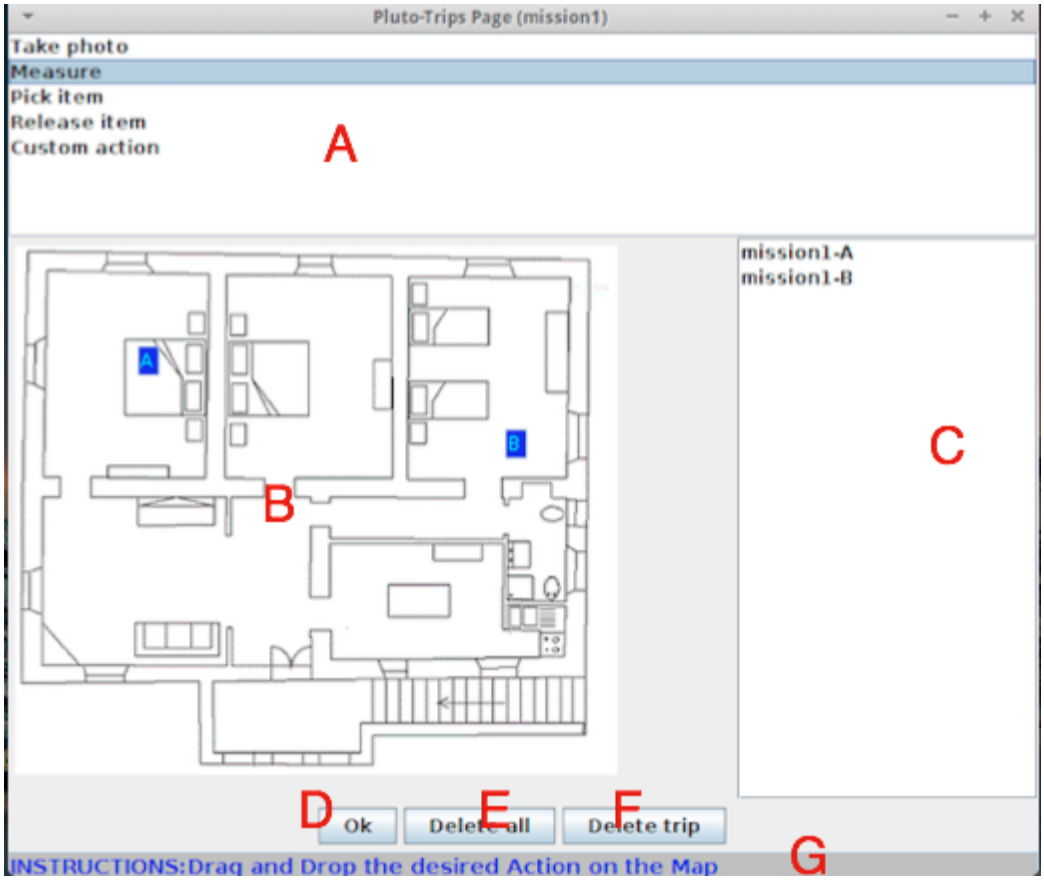

Figure 5.6: Trips Page structure

The Trips Page contains 7 components:

- Component A is a *JList*, a list of textual values, in this case the name of the Actions
- Component B is a *JImage*, simply an image representing the Map
- Component C is a *JList*, a list of textual values, in this case the name of the created trips
- Components D,E and F are *JButton*, rectangles that the user can click on

- Component G is a *JTextArea*, an area containing text, in this case the instructions on how to use this page

  All these components are easily usable in Swing, we only needed to import the already provided libraries. The structure of the other two pages of the Pluto Main Application is very similar to these one, differing only for the contained components.

5.6 Prototype drone

For the concrete actuation of the sensing tasks required by each application, we chose the Crazyflie Nano-Quadcopter, shown in figure 5.7.

Figure 5.7: The Crazyflie Nano-Quadcopter

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

- Small and lightweight, around 19g and about 90mm motor to motor
- Flight time up to 7 minutes with standard 170mAh Li-Po battery

- Standard micro-USB connector for charging which takes 20min for the stock 170mAh Li-Po battery
- On-board low-energy radio@1mW based on the nRF24L01+ chip. Up to 80m range (environment dependent) when using the Crazyradio USB dongle
- Radio bootloader which enables wireless update of the firmware
- Powerful 32 bit MCU: STM32F103CB @ 72 MHz (128kb flash, 20kb RAM)
- 3-axis high-performance MEMs gyros with 3-axis accelerometer: Invensense MPU-6050
- Available footprints to manually solder magnetometer HMC5883L/HMC5983 or/and barometer MS5611
- 4-layer low noise PCB design with separate voltage regulators for digital and analog supply

We use a particular API that makes the drone move from a *startingLocation* to a *destination* in the environment:

```
1  move(startingLocation, destination)
```

To concretely control the Crazyflie, there is a Python library which gives high level functions and hides the details. The precedent API uses the following to send the control commands to the Crazyflie:

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

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

```
1  roll  = 0.0
2   pitch = 0.0
3   yawrate = 0
4   thrust = 0
5   crazyflie.commander.send_setpoint(roll, pitch, yawrate, thrust)
```

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

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

Chapter 6

Evaluation

Intro to evaluation, description of the three parts and why they are important/-complementary (limitations!)

6.1 Generality

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

Pluto is fully exploited when there is a team of drones to manage (see 6.11 and 6.12), although it perfectly works also in the case of a single drone (see 6.10).

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 Object Finder, Warehouse Item-Finder, Drugs Distribution

6.1.2 Alfalfa Crop Monitoring and Pollination

The Alfalfa Crop Monitoring and Pollination[13] 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[14]. A swarm of drones can pollinate the alfalfa plants and also monitor them for pests and diseases, trough visual spot checks.
So, the whole application provide three periodic actions: searching for pests, searching for diseases, and looking for flowers in bloom. Each one of these actions is achieved by taking pictures of the plants. The user may need to define time constraints within the pollination action must be completed.

The following Pluto Editor graph describes the behavior of the Alfalfa Crop Monitoring and Pollination[13] application:

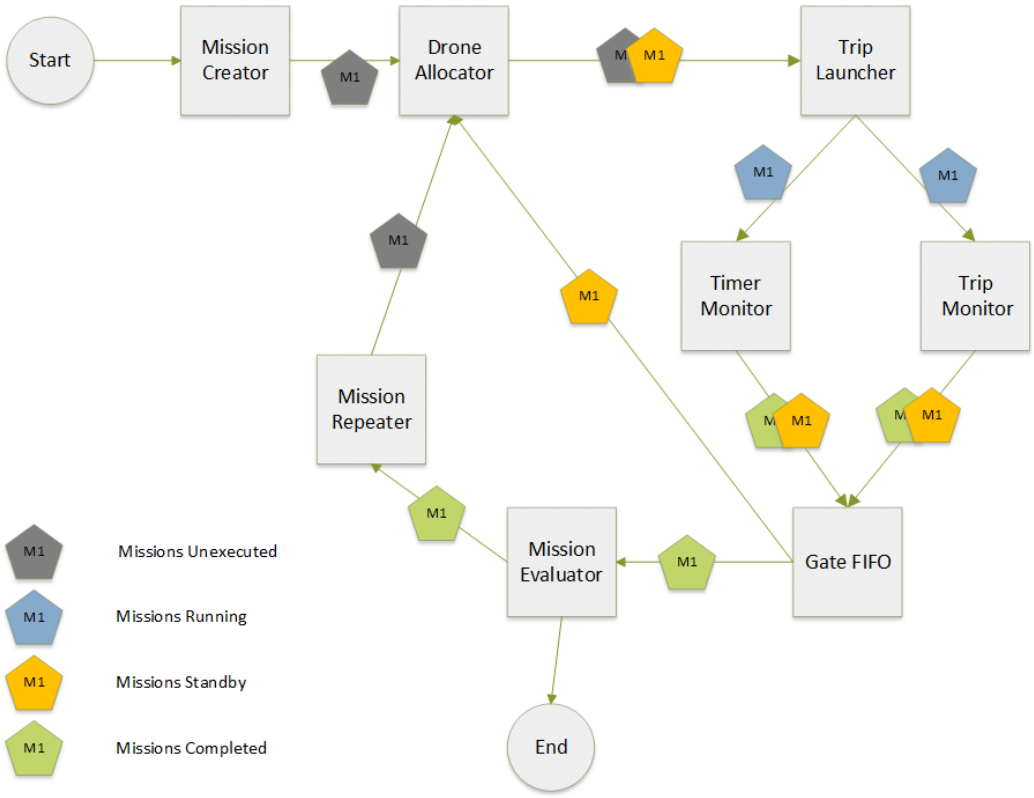

Figure 6.1: Pluto graph for the Alfalfa Crop Monitoring and Pollination application

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

The flow starts with the MissionCreator block, that creates a Mission instance to propagate in the graph. As already explained, a Mission is a list of sensing tasks to be executed sequentially. Then the DroneAllocator block takes care of assigning the right Drone to the next Trip in the list of trips to be executed of the Mission entity. A Trip is nothing but a movement of the Drone from a point A to a point B in the environment, at the end of which the Drone performs an Action. Then the Mission entity arrives to the TripLauncher that simply starts the execution of the first Trip and sets the its status to RUNNING.

After the TripLauncher the Mission entity is doubled. One instance goes to the TripMonitor block and the other one to the TimerMonitor. The TripMonitor changes the status of the executing Trip depending on the outcome of the Drone's

journey: if the Drone accomplishes to complete the journey and perform the Action, then the Trip status is set to COMPLETED, if the Drone crashes the Trip status is set to FAILED. The time constraints, which are set by the user thanks to the *timer* attribute of the Mission entity, are fulfilled thanks to the TimerMonitor block. This block takes care of setting the Mission status to FAILED if one of its Trips is not completed within the timer.

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

After the GateFIFO block there is a bifurcation: if the Mission is completed, it is sent to the MissionEvaluator, otherwise there are some trips of the mission that must be executed again, and so the Mission is sent to the DroneAllocator, which will assign new drones to these trips.

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

The Mission Repeater block takes care of continuously sending the drones to monitor these locations. Regarding the "Pollination" task, it can be added thanks to the *custom action* feature, through which the programmer can add to the model a brand new Action, making use of a specific external API. The full explanation of the functionality of each block can be found in Section 4.2.

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

flowers are in bloom, respectively.

The following is the code of the Evaluator needed for the development of the Alfalfa[13] application: "dataMap" is an hashmap that binds each Trip with the picture taken. The Trip is the key, which represents the journey performed by the drone, the Photo is the value, which is the picture taken by the drone once the Trip has been completed. For each photo, if the *pest* or the *disease* attributes are true, the system will signal to the farmer the location where the problem exists, through the log function. If the *bloom* attribute is true, the plants at that location must be pollinated: a new Trip entity is created, its Action is set to "Pollinate" and the target location is set to the same location of the Drone that found the bloom. Finally the Trip is added to the list of trips of the current mission. The Mission status is set to STANDBY because there is at least a new inserted Trip to execute.

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

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

To concretely choose the specific locations to monitor, the user is provided with a map over which he can drag and drop the action *Take photo*. As already explained, a Mission object contains a list of trips to be executed. Inside the Mission entity, these trips are performed sequentially, in general each one by a different Drone, as already explained in Section 4.1. So, if the user wants to send more than one drone simultaneously on the same location, he has to create more than one Mission. Indeed missions are executed in parallel, so if the user wants to simultaneously send 3 drones on the same location, he simply has to create three missions. Then, since each Mission has its own Trips Page, he has to choose the same locations on the maps of the three Trip pages.

To further clarify the development of the Alfalfa[13] application with Pluto, we now show a real execution of the Alfalfa application in a concrete scenario: imagine we want to simultaneously send three drones to monitor the plants distributed in a circular area. So, we create three Mission entities, and, for each of them, we drag and drop the action *Take photo* on the map of its Trip page, in order to create the trips composing the circular area, as shown in figure 6.2. The figure 6.2 represents the trips of one of the three Missions. Each one of the three missions has its own map where the user distribute the trips to perform. Then, when the missions start, the drones will take photos over these spots and, in case of bloom, new Trips will be created to Pollinate the area.

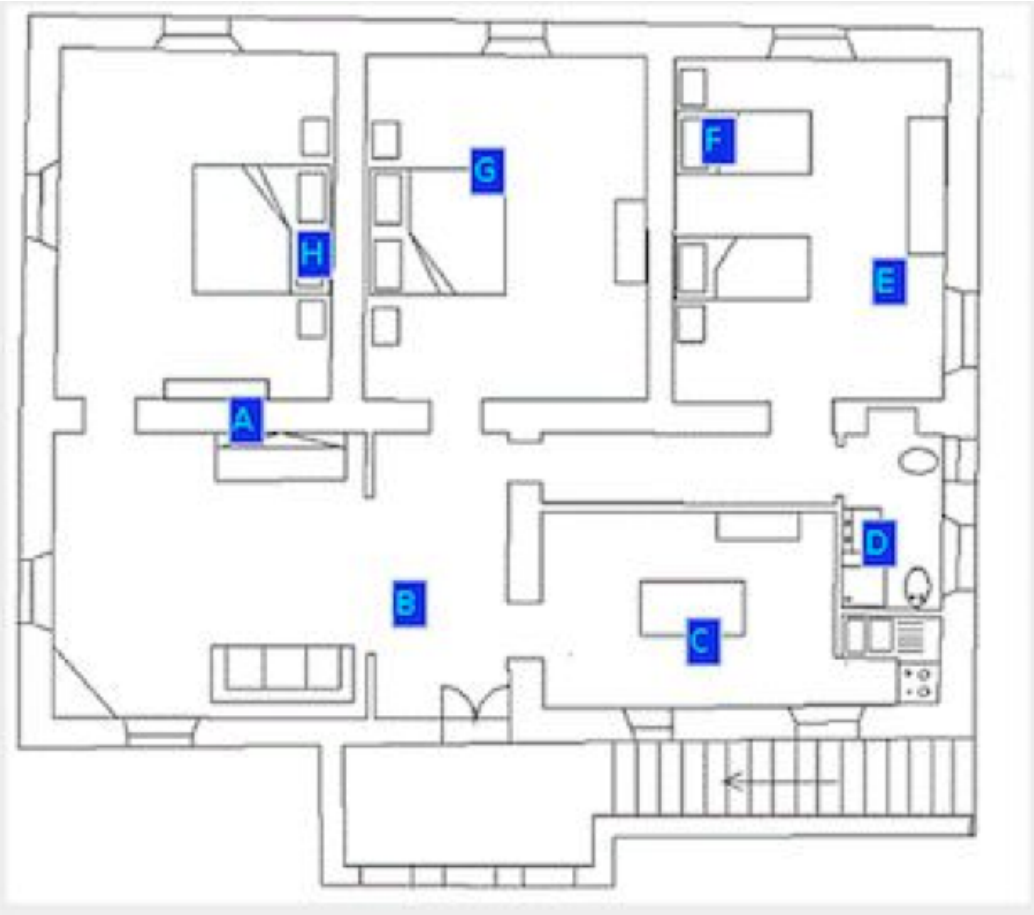

Figure 6.2: The circular area to monitor

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

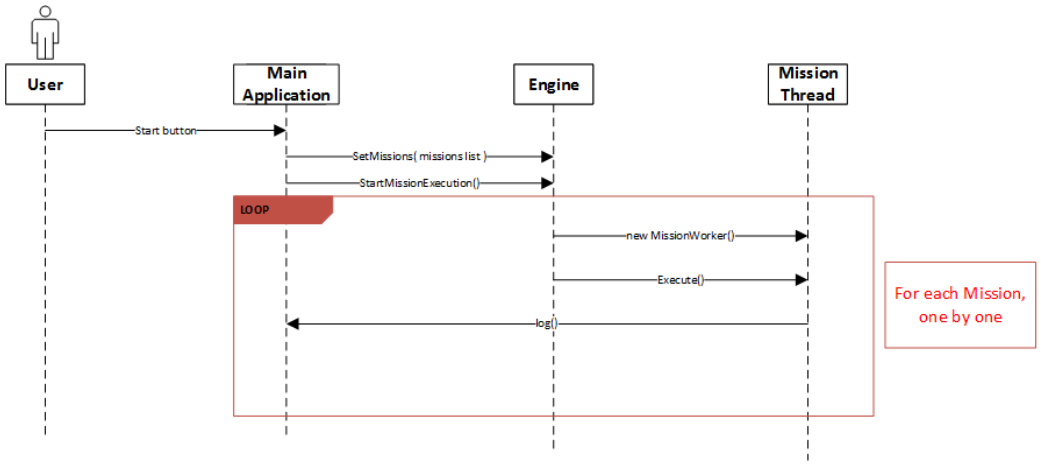

Figure 6.3: Sequence diagram of a starting mission

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

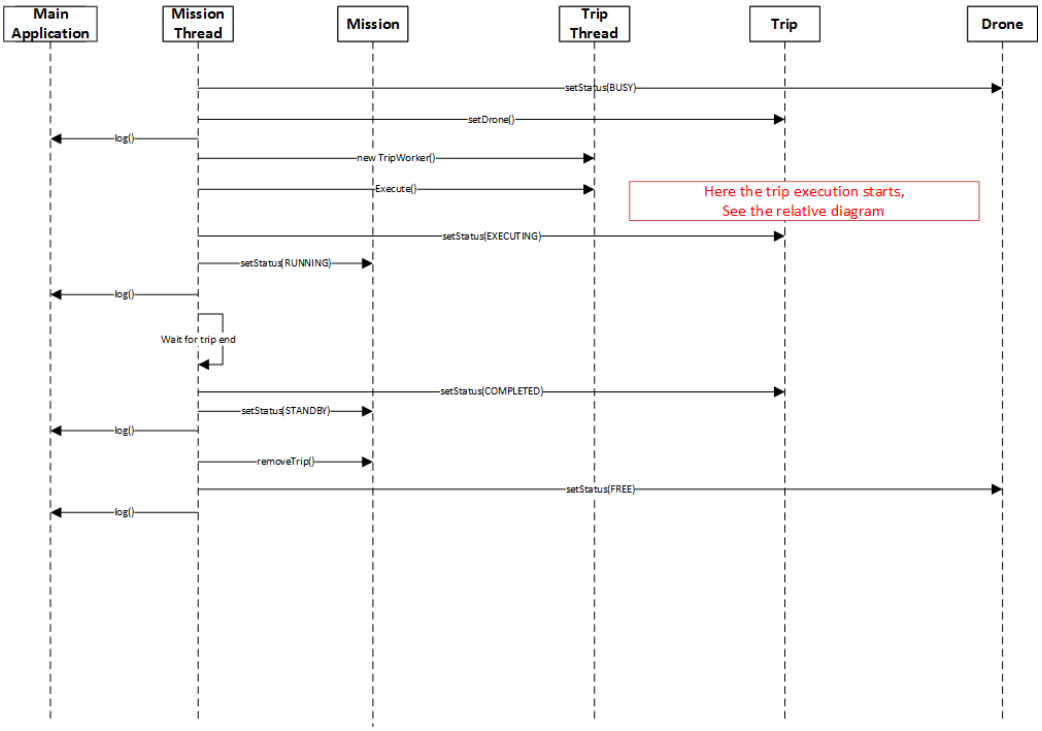

Figure 6.4: Sequence diagram of the mission execution flow

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

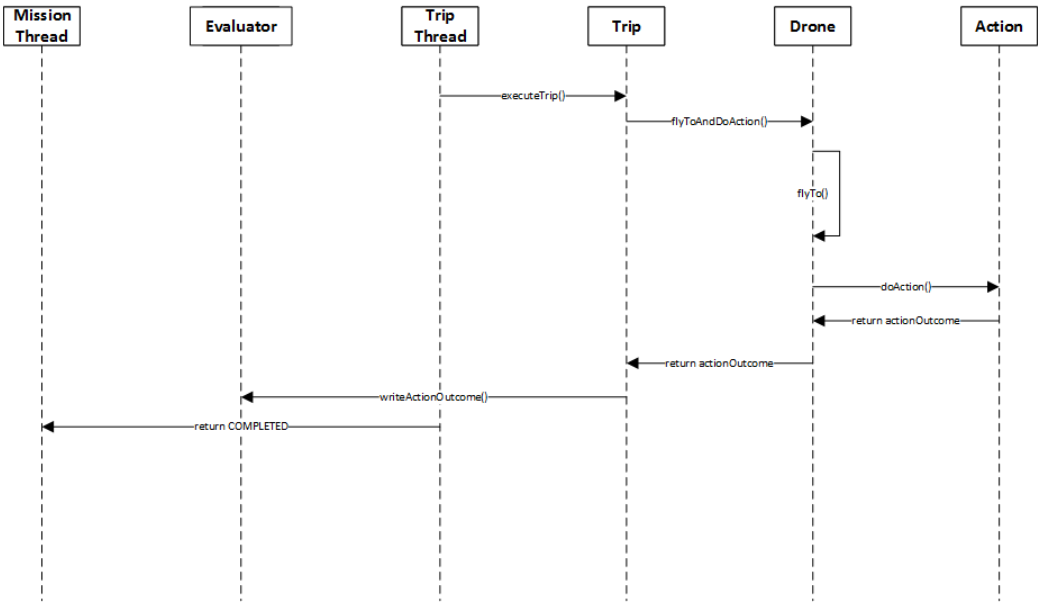

Figure 6.5: Sequence diagram of the trip execution flow

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

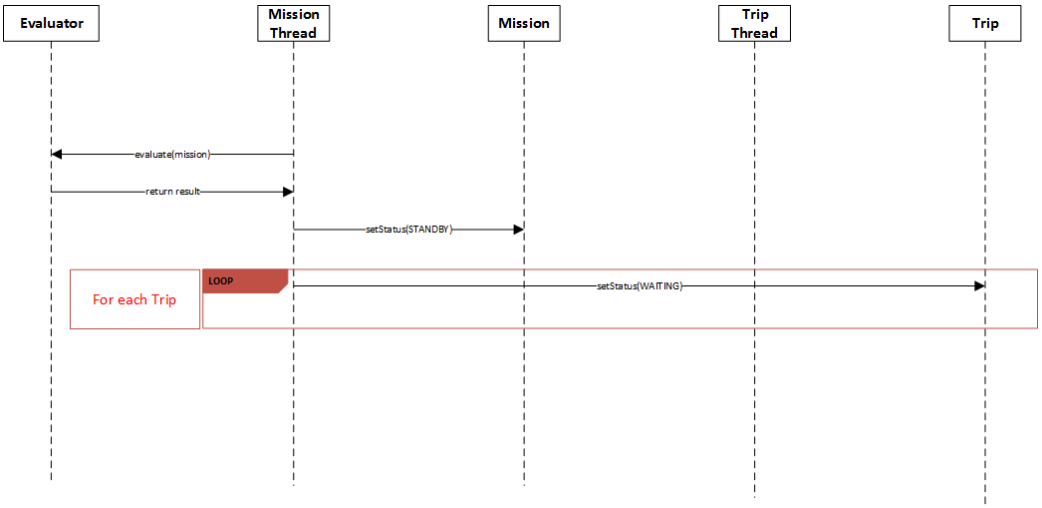

Figure 6.6: Sequence diagram of the mission ending flow

In figure 6.6 the final steps of the Mission execution are shown. The *Evaluator* checks the pictures taken by the Drones, looking for pest, diseases or blooms. The Trips are re-inserted in the execution list, their status is set to WAITING,and the Mission status is set to STANDBY because of the MissionRepeater block. The MissionRepeater takes care of executing again a Mission, and to do so the status of the Mission must be STANDBY.

6.1.3 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 that 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.

Regarding the Pluto graph needed for this application, it is very similar to the Alfalfa[13] application one, shown in figure 6.1. The only difference is the absence of the MissionRepeater block, since this application does not need the drones to repeat their tasks continuously. So, in the graph of the "Aerial mapping of archaeological sites" application the MissionEvaluator block is directly connected to the DroneAllocator.

The flow of the Mission entity on the graph is the same of the Alfalfa[13] application, and is fully described in section 6.1.2. In this case the MissionEvaluator block analyzes the drones data at the end of the missions, allowing the Main Application to decide if the photos are good enough or if more drones must be sent out to take new pictures in these locations.

It is important to underline that, using the Pluto framework, it is not possible to obtain the very same behavior of the original application. Indeed, two consecutive photos must be taken within a time constraint. Since each photo is taken by the Drone at the end of its assigned Trip, there should be a way to define a time constraint between the execution of two trips. This cannot be fulfilled with Pluto, since the TimerMonitor 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 with Pluto.

Concerning the code of the Evaluator block needed for the development of the Aerial Mapping[15] 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[13] one, because the Evaluator algorithm acts in a different way: in Alfalfa[13] 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 possible thanks to the centralized data store of Pluto: each Drone sends its data to the central brain, that collects them together and perform the computations on the data following the Evaluator algorithm.

The following code snippet shows our implementation of the Evaluator algorithm:

```
1     String result = null;
2     // The "stitch" method takes a collection of photos as input
3       // and return an Ortophoto object derived by a proper algorithm
4       // based on the passed photos
```

```
5       OrtoPhoto ortophoto = stitch(dataMap.values());
6
7       if(ortophoto.getAberration() > ABERRATION_THRESHOLD){
8
9         // Iteration on the photos that were used in the stitch method
10          // to generate the Ortophoto
11        for (Photo photo: ortophoto.getPhotoCollection()){
12
13          // if the overlap of the single photo is not enough
14          if (ortophoto.getOverlapOfGivenPhoto(photo) < OVERLAP_THRESHOLD){
15
16            // Loop on all the Trips of every missions
17            for (Map.Entry<Trip, Object> entry : dataMap.entrySet()) {
18
19              // take the Photo of the current iteration
20              Photo p = (Photo) entry.getValue();
21
22              // When we found the photo that has the low overlap
23              if(photo.equals(p)){
24
25                // create a new Trip that will take a new photo
26                     // from the same location
27                Trip trip = new Trip();
28                trip.setName("NewTrip");
29                trip.setTargetLocation(entry.getKey().getTargetLocation());
30                trip.setAction(Action.TAKE_PHOTO);
31                trip.setStatus(Trip.WAITING);
32
33                // add this new trip to the list of trips to be executed
34                missionToEvaluate.getTrips().add(trip);
35
36                // set the mission status to STANDBY, since a new trip has been
                       created
37                missionToEvaluate.setStatus(Mission.STANDBY);
38              }
39            }
40          }
41        }
42      }
43
44       // All decisions were been chosen so we end the evaluation
45      result = "Success";
46      return result;
```

To send the drones to take pictures over the site locations, the user has to simply create the Mission entities and add the trips in the Trips Page of each Mission. In case the archaeologists want to send more drones on the locations where they can't obtain adequate ortophotos, they just have to add more Mission entities. For example, if an archaeologist wants to send 3 drones simultaneously on a particular

location, he has to create 3 Mission entities. Then, in the Trips pages of each Mission, he simply has to drag and drop the *Take photo* action on that particular location.

In order to show the real runtime execution of the Aerial Mapping application with Pluto, we now show a possible scenario: there are 7 drones and 1 big archaeological site to monitor, and we want to send all the drones in that area at the same time to take the ortophotos. As usual, the user has to create 7 Mission entities. Then, in each Mission's Trips Page, he has to drag and drop the action *Take photo* on the locations forming the site, shown in figure 6.7. Once again, the figure shows the map on the Trips Page of one of the seven missions.

Figure 6.7: The archaeological site on the map

The sequence diagrams related to this application are the same shown in Section 6.1.2. The only difference is that now the application does not require the repetition of the missions. This leads the Main Application to end the execution flow when the Mission reaches a successful evaluation.

6.1.4 PM10

The PM10[16] 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.

Regarding the Pluto graph needed for this application, it is very similar to the Alfalfa[13] application one, shown in figure 6.1. The only difference is the absence of the MissionRepeater block, since this application does not need the drones to repeat their tasks continuously. So, in the graph of the "Aerial mapping of archaeological sites" application the MissionEvaluator block is directly connected to the DroneAllocator.

The flow of the Mission entity on the graph is the same of the Alfalfa[13] application, and is fully described in section 6.1.2. The data collected by the drones are not photos anymore, but a *pollutionQuantity* value which indicates the percentage of pollution in that area. In this case the Mission Evaluator block takes care of confronting *pollutionQuantity* variables are confronted and the gradients between areas of higher concentration are computed. So new drones are sent along this gradients, improving the spatial profile.

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 is provided with.

As for the Aerial Mapping application, shown in section 6.1.3, 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.

Below there is our implementation of a possible Evaluator algorithm:

```
String result = null;

// building a new map with only the trip-measure couples of the mission
// to evaluate
Map<Trip, Integer> missionMap = new Map<Trip, Integer>();
for (Map.Entry<Trip, Object> entry : dataMap.entrySet()) {
  if (missionToEvaluate.getCompletedTrips().contains(entry.getKey())) {
    missionMap.put(entry.getKey(), (Integer) entry.getValue());
  }
}

// this method use the Trip location and the pollution measure to
// calculate
// the gradients and then return a list of String that indicates
// the positions of these gradients
List<String> gradientsPositions = calculateGradients(missionMap);

for (String position : gradientsPositions) {

  // create a new Trip to calculate pollution at the gradient position
  Trip trip = new Trip();
  trip.setName("GradientTrip");
  trip.setTargetLocation(position);
  trip.setAction(Action.MEASURE);
  trip.setStatus(Trip.WAITING);

  // add this new trip to the list of trips to be executed
  missionToEvaluate.getTrips().add(trip);
  // set the mission status to STANDBY, since there are new trips to perform
  missionToEvaluate.setStatus(Mission.STANDBY);

}

// set the result of the evaluation
result = "Success";
return result;
```

Now we show the execution of the PM10 application with Pluto in a particular scenario: we have 5 drones and we want to measure the pollution quantity in the area shown in figure 6.8 using all of them. As usual, the user has to create 5 Missions and has to choose the locations of the area to sample through the map on the Trips Page of each Mission. In this way, five drones will monitor simultaneously the area to sample.

Figure 6.8: The area to sample on the map

As for the Aerial Mapping[15] application, the repetition of missions is not required. So the sequence diagrams do not change and the reader can see them in sections 6.1.2 and 6.1.3.

6.1.5 PURSUE

The PURSUE application[17] is representative of surveillance applications. A team of drones monitor an area and they have to follow moving objects which pass through, taking a picture of each one of them when they enter in the camera field. To do so, drones can operate in two distinct modes: when in "patrolling mode" they simply inspect an area, while when an object is found they switch to "pursuing mode" and start to follow the object. Since an object could move faster than the drones, no drone can follow it constantly, the system must take care of switching between the real drones in order to constantly follow the target. There are time constraints to respect between the detection of a moving object and when its picture is taken and, in case of violations,every tracked object with at least one acquired picture is released from tracking, to regain the drone resources and lower the acquisition latency for the next object.

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

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

The applications described above were already developed and tested with other systems, like Karma[1] and Voltron[2], but we also developed new applications and tested our framework on them: the Object-finder (OF), the Warehouse item-finder (WIF) and the Drugs distribution(DD).

The three applications are modeled by the same Pluto Editor graph, shown in figure 6.9:

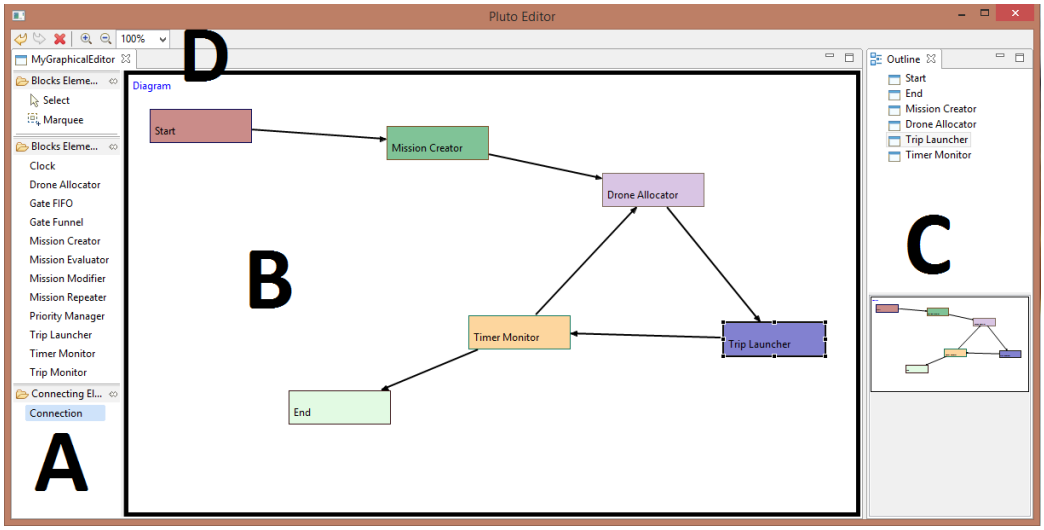

Figure 6.9: The Pluto graph of the OF,WIF and DD applications

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

6.1.6 Object-finder (OF)

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

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

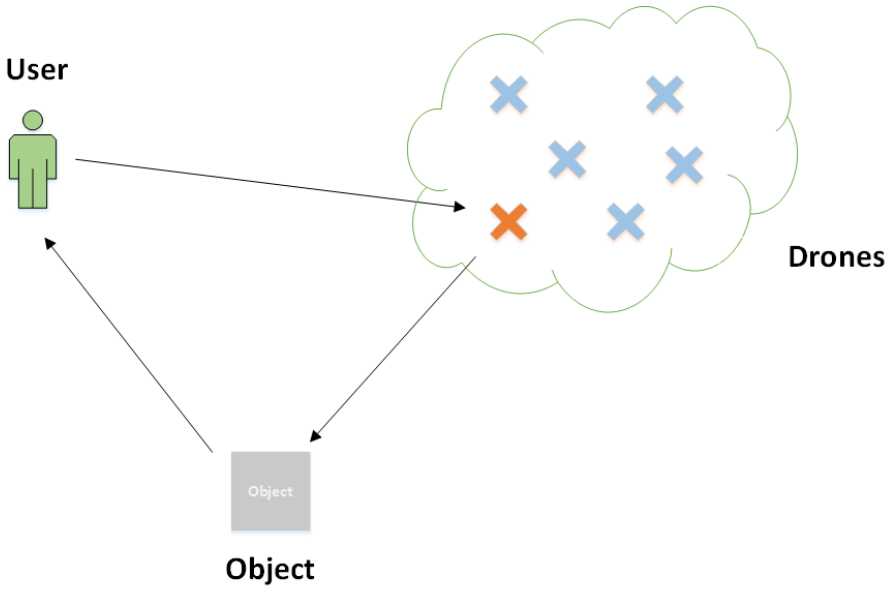

Figure 6.10: The basic functioning of the Object-finder application

6.1.7 Warehouse item-finder (WIF)

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

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

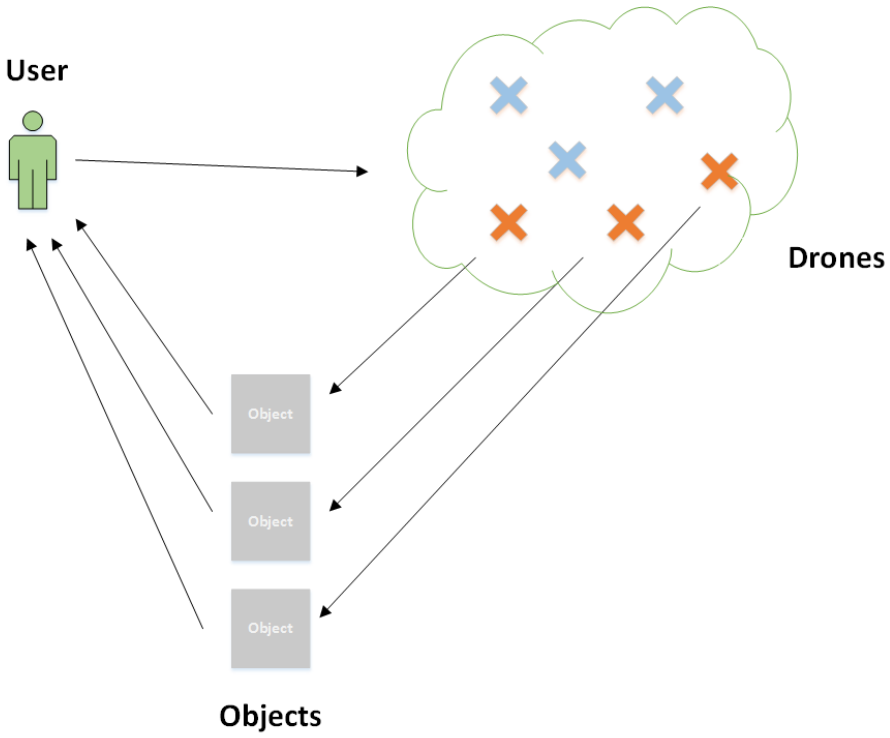

Figure 6.11: The basic functioning of the Warehouse item-finder application

6.1.8 Drugs distribution (DD)

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

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

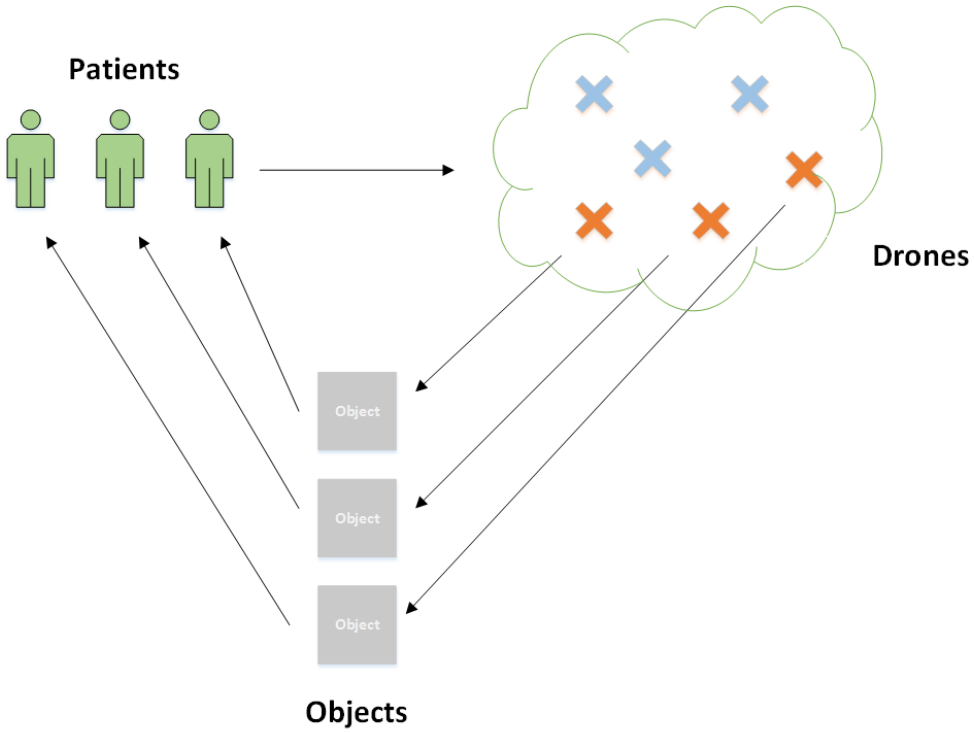

Figure 6.12: The basic functioning of the Drugs distribution application

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

6.2 Usability of the model

To evaluate the concrete usability of the Pluto programming framework, we decided to test it on real people, proposing them two "exercises". We involved five testers, recruited both in the "Politecnico di Milano" and "SICS Swedish ICT" environment, in order to guarantee a solid development background and to avoid possible lack of programming knowledge. We created one exercise for each component of Pluto framework. The first one consists in the development of an application using the Pluto Graphical Editor. The exercise is split in three levels, starting from a very basic version and going through more difficult versions. Each version asks the user to add a new functionality by using the available components in the Editor. The second exercise, instead, asks the user to use the generated code from the previous exercise to run the Pluto Main Application, then asks to create some missions and, in the end, to run them. The application we choose for the exercises is the Drugs Distribution, already described in section 6.1.8, 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), 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 4.3.1) and giving him the list of the available entities (shown in Section 4.1) and of the implemented blocks (shown in Section 4.2), 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 4.3.2).

First Exercise

The exercise proposes the development of the Drugs Distribution application (shown in section 6.1.8) 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 6.13, 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.

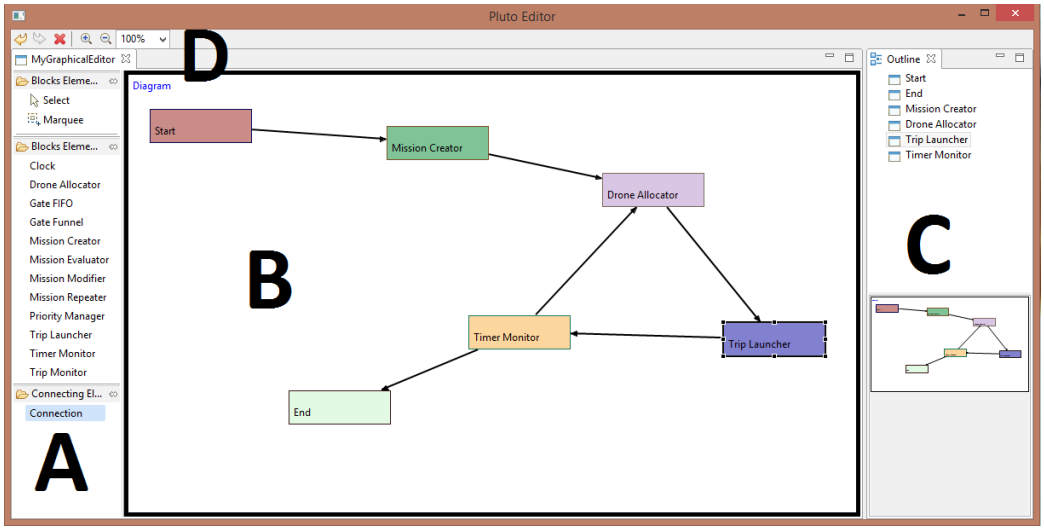

Figure 6.13: Solution of the first step

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 6.14.

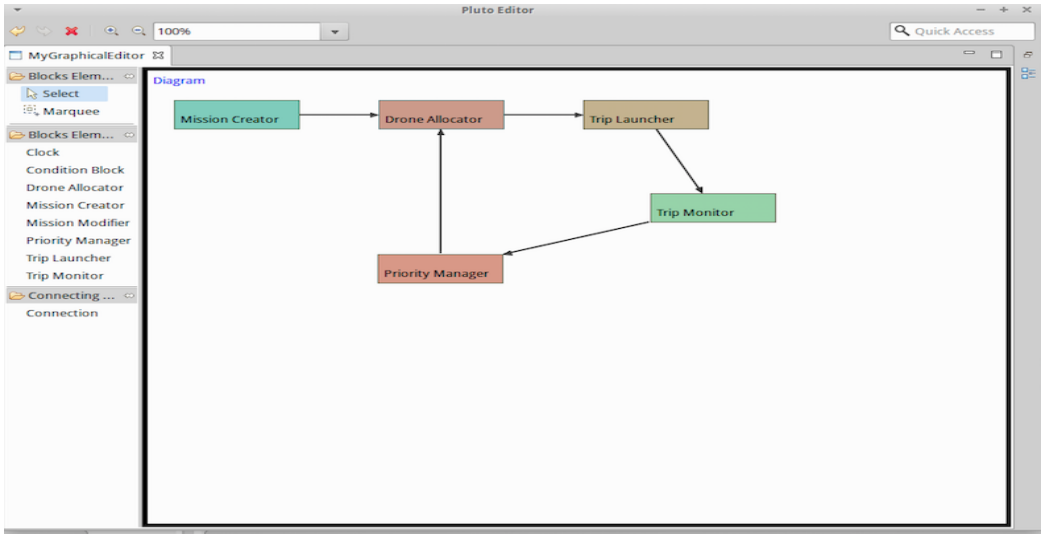

Figure 6.14: Solution of the second step with Priority Manager

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 6.15.

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.

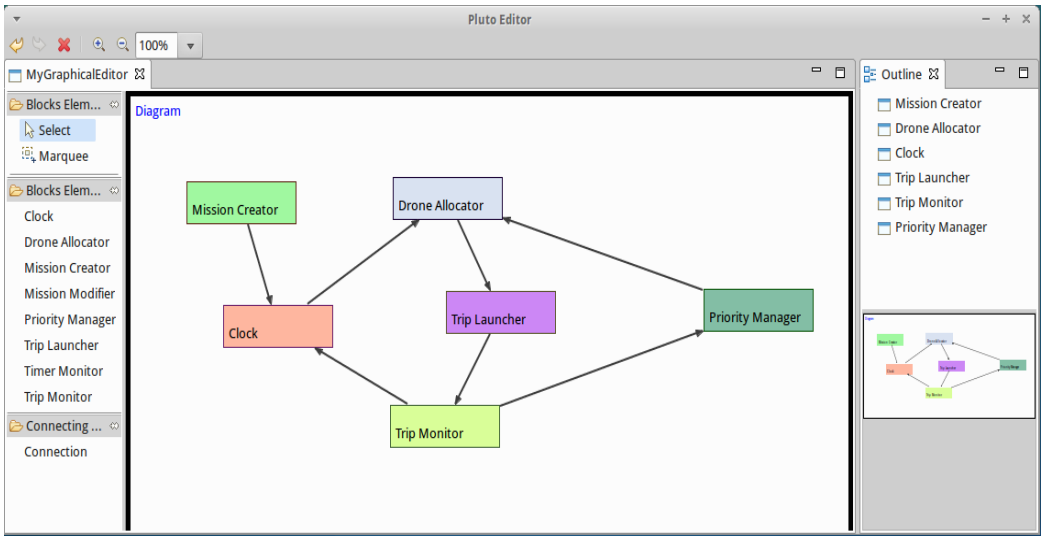

Figure 6.15: Solution of the second step with Clock block

For the third step, the user has to implement the feature of the Timer Monitor block without using it. So, he has to use the Mission Modifier block, through which he can insert his code in the application, and put it between the Trip Launcher and GateFIFO blocks, in parallel with the TripMonitor, as shown in figure 6.16. This is a very useful step to compute, because the user learns how to use Mission Modifier block, which is an important feature of the Pluto Editor, because it allows the programmer to insert his custom code to characterize the application.

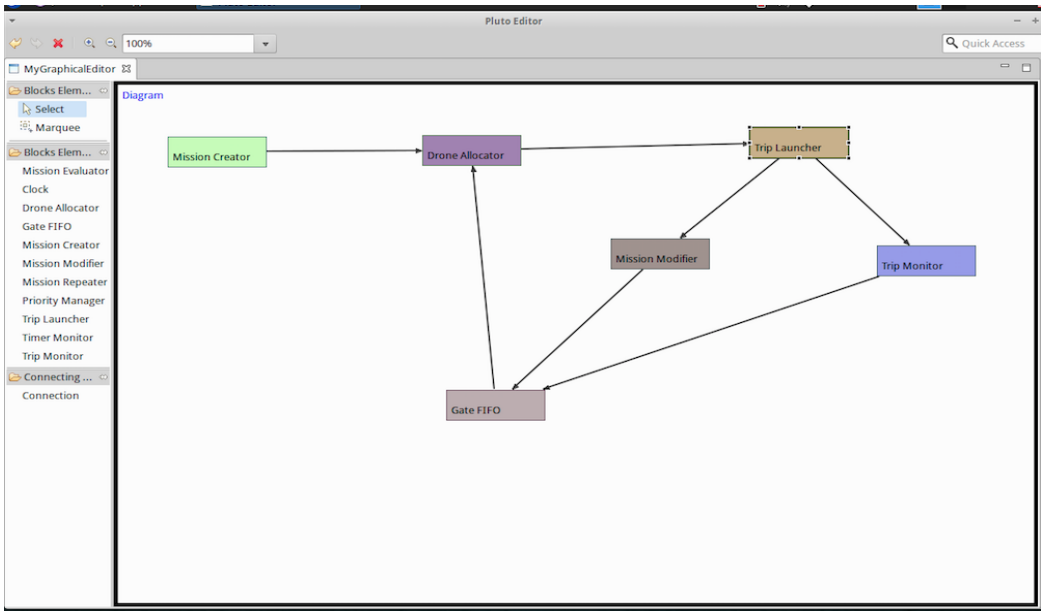

Figure 6.16: Solution of the third step

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:

1. Number of people who correctly solved the first part of the exercise
2. Number of people who correctly solved the first and second parts of the exercise
3. Number of people who correctly solved the whole exercise
4. Mean time for the resolution of the first part of the exercise
5. Mean time for the resolution of the second part of the exercise
6. Mean time for the resolution of the third part of the exercise
7. Mean time for the resolution of the whole exercise
8. Number of people who solved the whole exercise, but in a wrong way
9. Number of people who could not solve the exercise at all

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

6.2.3 Baseline

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

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

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

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

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

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

6.2.4 User Survey

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

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

The survey can be found following this link:

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

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

6.2.5 Results

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

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

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

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

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

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

6.3 Performance evaluation

In order to strengthen the evaluation of Pluto, after the user study, described in Section 6.2, we evaluated some quantitative metrics. These metrics are divided in two main types: software metrics and resources 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 (the Graphical Editor and the Main Application) we decided to split this evaluation in two parts: this means that each kind of evaluation was performed on the Pluto Graphical Editor first and then on the Pluto Main Application. To help us in this procedure we used a very useful tool called VisualVM, shown in figure 6.17.

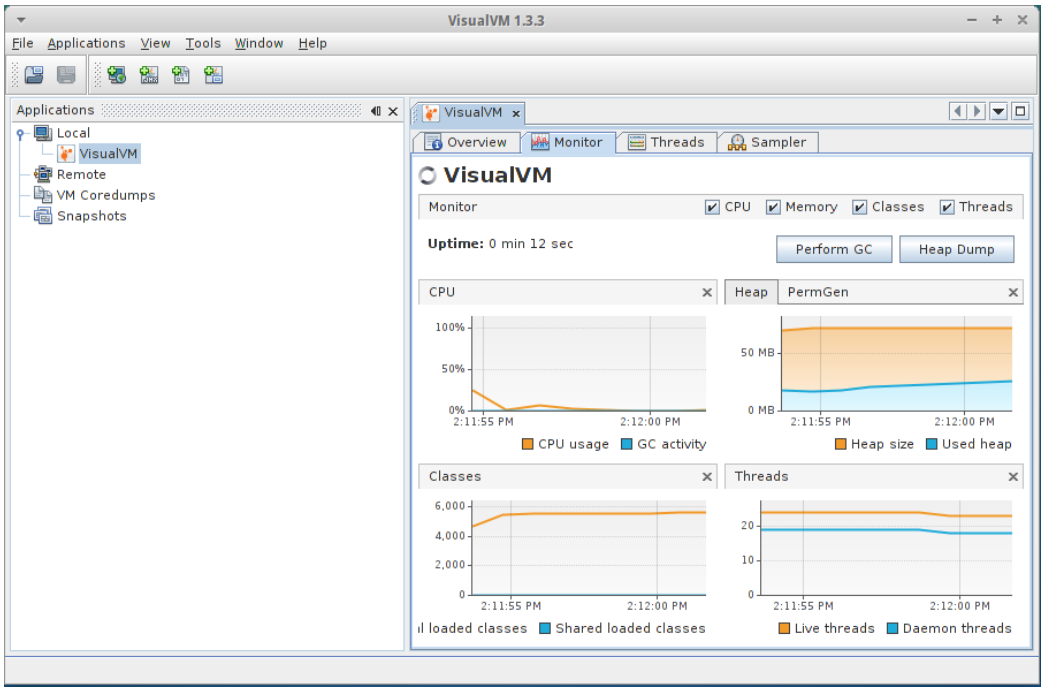

Figure 6.17: VisualVM interface

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. Finally, in the Result section 6.3.3, we describe the outcome of the tests for the Pluto Graphical Editor and the Pluto Main Application separately.

6.3.1 Software Metrics

The software metrics let us understand the complexity of the software. We decided to record this information for each component of the Pluto Framework, because it lets us to understand how much expandable it is and then how much effort a different developer should spend to add new features in the future.

These parameters are:

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

6.3.2 Resources Consumption

The resources consumption metrics are those parameters measured at run-time, during the execution of the software. With this evaluation we checked if the Pluto framework generates any performance issues, because of a too high requirement of resources.
Furthermore, due to the team level approach we have chosen, as said in Chapter 3, it is important to verify if critical issues happen during the normal execution because the central brain introduce a single point of failure. It is essential to avoid any bottleneck situations. These metrics are:

- CPU Load
- Memory Consumption
- Live Threads

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

- CPU: Intel i7 2640
- RAM: 4GB
- VGA: Nvidia GeForce 610M
- SSD: Kingston 120GB
- OS: Xubuntu 14.04
- Java: JDK 1.7

Concerning the Pluto Graphical Editor, we measured these parameters while generating the source code of the Main Application from an increasingly more complex drawing. We raised step by step two parameters: the number of blocks and the number of connections. At first we fixed the former and we incremented the latter by a step of 5, starting from 1 connection. Then we did the same operation fixing the number of connections and raising the number of blocks by 5, now starting from 2 blocks.

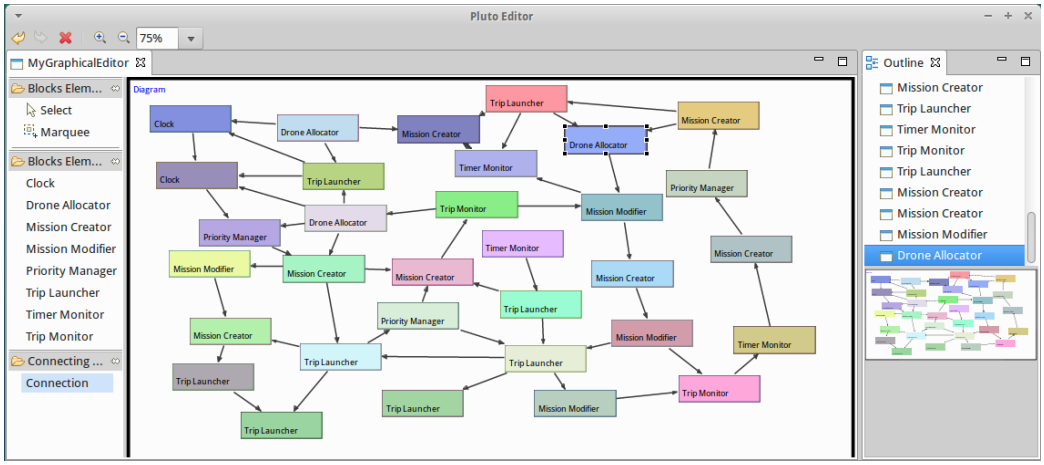

Figure 6.18: Very Complex Diagram Example

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 the number of missions and trips varying instead the number of drones. Then we fixed the missions and the drones varying the trips. In the end we fixed the trips and the drones varying the number of missions. In this way, we could evaluate the performance of the Main Application in an accurate way. The results are shown in next section 6.3.3.

6.3.3 Results

In this section we show the results of the software and resources consumption evaluation. First of all, we evaluated the software complexity of the Pluto framework, according to some important software metrics. The table 6.2 proposes the results.

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

Table 6.2: The metrics concerning the two Pluto components

Thanks to these measures, we can make some considerations about the size and the spent effort of the Pluto framework.
Concerning the Main Application, these values are dependent from the diagram generation process because the amount of the generated surplus code doesn't exceed the 2-3% of the total lines of code of the template Main Application. Indeed the generation adds no more than 50 lines.
Instead, the Graphical Editor has a higher volume than the Main Application, with all the measured values doubled. This can be explained by the fact that, to develop the Editor, we based our code on Eclispe GEF Framework. Then most of the classes, have inherited from other parent classes inside this framework and we inherited its complexity too.
The higher size explains also the time we spent to develop the Graphical Editor and the time needed to add new features during the software revision steps. Indeed

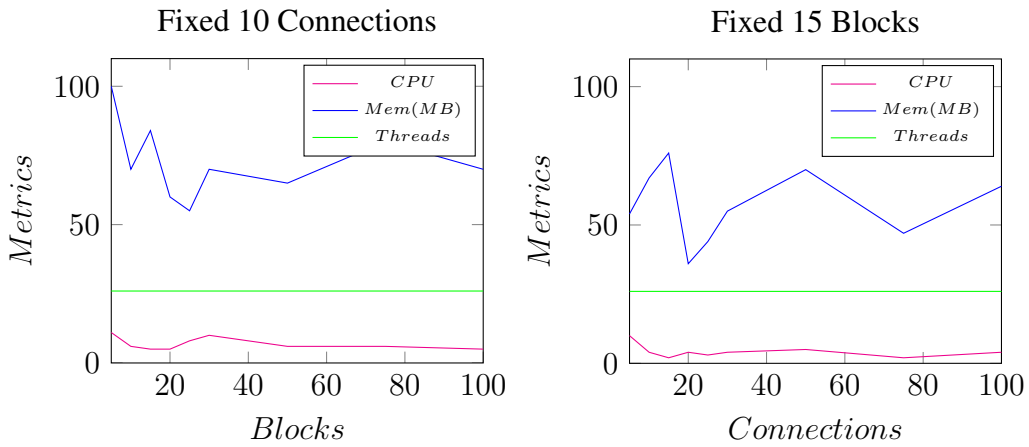

Figure 6.19: Resources consumption metrics of Graphical Editor

we found the GEF framework a bit hard to maintain.

In the end, we present the results concerning the resources consumption of the two Pluto framework components.
Starting from the Graphical Editor, the diagrams in picture 6.19 describe the resources consumption when the connections number raise with fixed number of blocks and when increasing the blocks amount with fixed connections.

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

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

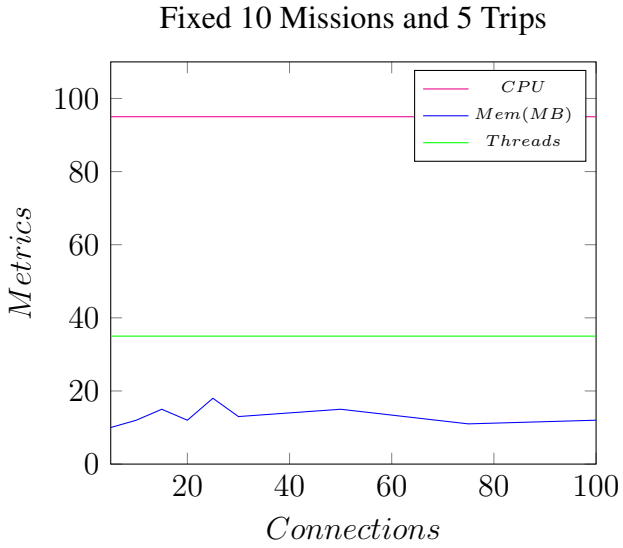

Figure 6.20: Evaluation results of Main Application with fixed missions and trips

The diagram 6.20 describes the resources consumption of the Main Application with a fixed number of missions and trips and a raising number of drones. We created 10 missions with 5 trips each.

The diagram 6.21 describes the resources consumption of the Main Application with a fixed number of missions and drones and a raising number of trips for each mission. We created 10 missions and made available 15 drones.

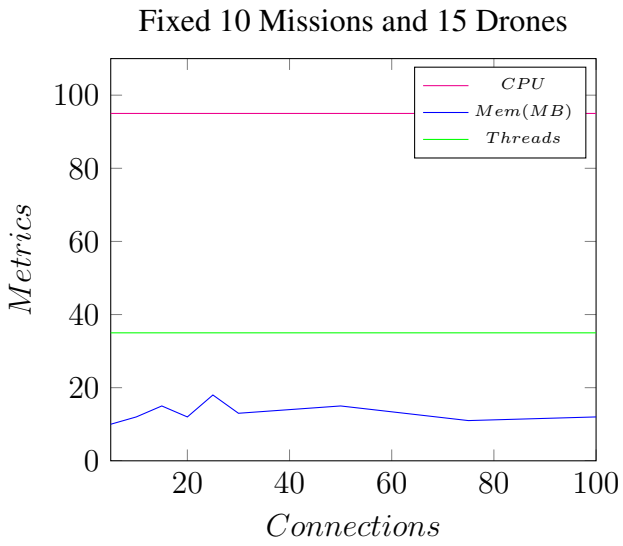

Figure 6.21: Evaluation results of Main Application with fixed missions and drones

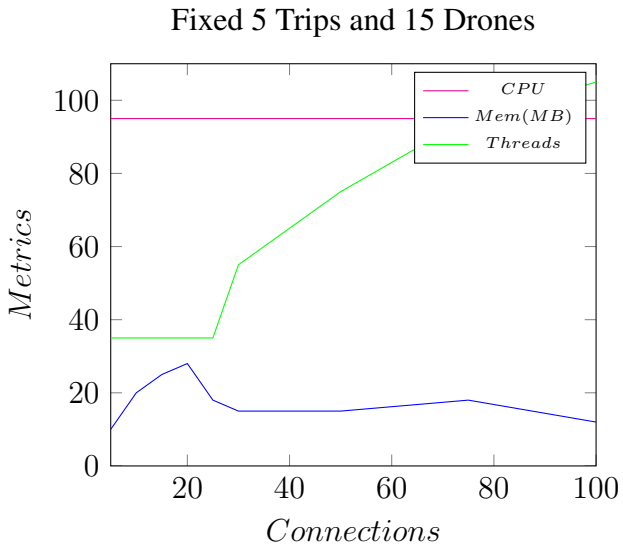

Figure 6.22: Evaluation results of Main Application with fixed trips and drones

The diagram 6.22 describes the resources consumption of the Main Application with a fixed number of trips and drones and a raising number of missions. We created 5 trips for each mission and made available 15 drones.

As you can see, the general consumption is almost the same for each situation:

there is a low request of memory, a limited number of live threads, and a quite high consumption of the CPU.

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

Chapter 7

Conclusions and future works

7.1 Conclusions

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

Each one of these blocks, described in section 4.1, contains the implementation of a specific functionality.

So the programmer can build an application just deciding which features he needs and consequently connecting the right blocks, using the editor shown in section 4.3.1.

The final user decides the sensing tasks that have to be performed by the drones thanks to the user interface, fully described in section 4.3.2.

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

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

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

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

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

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

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

7.2 Pluto limits and future works

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

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

7.2.1 Implementation limits

The PURSUE application, described in section 6.1.5, put in evidence the Pluto main limitation: the immediate execution of actions in response to instantaneous events.

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

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

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

7.2.2 Technological limits

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

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

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

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

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

Bibliography

[1] 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.

[2] 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.

[3] CNN.com, "Your personal 849 dollars underwater drone," in *goo.gl/m1JRuD*.

[4] iRobot, "Create Programmable Robot," in *goo.gl/bJhrMR*.

[5] M. Brambilla et al., "A review from the swarm engineering perspective," in *Swarm Intelligence*, 2013.

[6] 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.

[7] 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.

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

[9] Greg Sterling, "Magnetic positioning," in *Opus Research Report*, 2014.

[10] Fan Li and Chunshui Zhao and Guanzhong Ding and Jian Gong and Chenxing Liu and Feng Zhao, "A reliable and accurate indoor localization method using phone inertial sensors ," in *UbiComp '12 Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM New York, 2012, pp. 421–430.

[11] Frédéric Evennou and François Marx, "Advanced integration of WIFI and inertial navigation systems for indoor mobile positioning," in *EURASIP Journal on Applied Signal Processing*. Hindawi Publishing Corp, Ne York, 2006, pp. 164–164.

[12] Silke Feldmann and Kyandoghere Kyamakya and Ana Zapater and Zighuo Lue, " An indoor Bluetooth-based positioning system: concept, Implementation and experimental evaluation ," in *International Conference on Wireless Networks - ICWN*, 2003, pp. 109–113.

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

[14] E. Stokstad, "The Case of the Empty Hives," in *Science*, 2007.

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

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

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