Université Libre de Bruxelles
Section M-IRIFS
Academic Year 2022-2023
PROJ-H-402

Computing Project

Swarm Skill Wizard

Author

Nicolas **Seznec**

Supervisor

Darko **Bozhinoski**

August 26, 2023

Contents

1 Introduction

The aim of this project, as part of the PROJ-H-402 course, is to produce a software tool to allow a user to specify robot skills and design a simple robot swarm mission.

The context of this project is related to the AutoMoDe package, which is an approach to the automatic design of robot swarms [1]. Many concepts of the application come directly from AutoMoDe, and some features allow to create files that would ideally be compatible with the package (such as ARGoS files or loop functions files, as explained later in the report).

This report will first introduce some of the concepts used by the application, followed by the objective of the project and a description of all the implemented features in the final application. It will then go more in depth over the architecture of the code and the development of the project itself.

The project is made in python, with notably the *PyQt5* library for the creation and management of the user interface. A file `requirements.txt` allows for the installation of required dependencies, with more details provided in the file `README.md`.

2 Background and objective of the project

2.1 Background and concepts

Before delving into the details of the application, it is necessary to introduce some concepts useful to the comprehension of the task. This section briefly explains what is a robot skill, a behavior, and a mission, which will be featured prominently through the rest of the report.

The first important concept for this project is the concept of a robot skill. A skill serves as a means to instantiate the generic abilities of a robot. Each skill has its own set of parameters. For instance, a skill *perceive* means that the robot must be able to perceive something specific, where a parameter could be the distance at which to perceive it, with a certain resolution, accuracy, etc.

A behavior allows to specify skill realizations. It is more concrete than a skill, closer to how the robot interacts with other robots and its environment. Each behavior is mapped to a set of skills that can be used, and also has its own parameters. For instance, the behavior of *color elusion*, where the robot moves away from objects of a specific color, would be linked to the skill *perceive*, and one parameter could simply be the color to which the behavior reacts.

For this project, only a handful of skills and behaviors are considered. The three chosen skills are *perceive, navigate and communicate*, while the chosen behaviors are *exploration, stop, phototaxis, anti-phototaxis, attraction, repulsion, color following, color elusion, message attraction and message repulsion.* They are mapped as shown in the following figure :

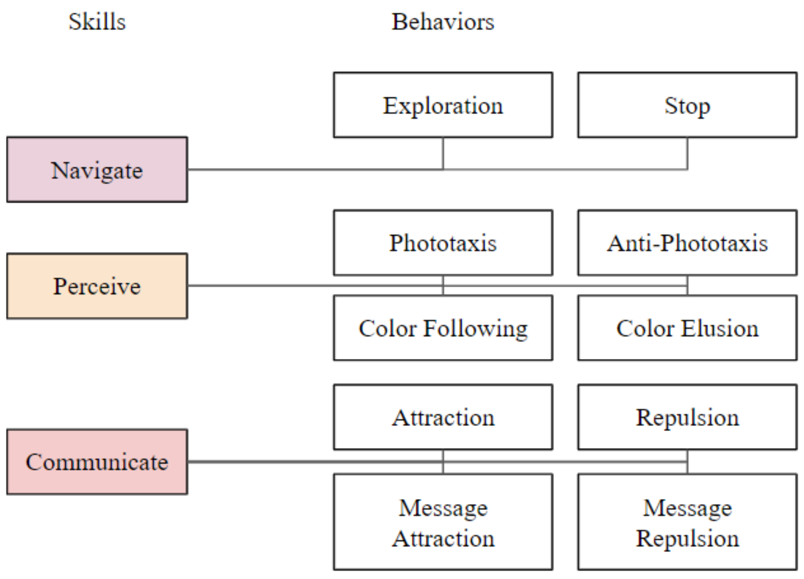


Figure 1: Link between chosen skills and behaviors

For a particular mission, the user can choose the set of skills required to accomplish it. A mission is simply the goal of the swarm, evaluated using an objective function which gives the performance of the swarm. For instance, example of mission are *stop* (the robots must stop moving as soon as a color signal appears) or *aggregation* (the robots must aggregate in a region where a specific color is displayed). In this project, the mission includes the whole configuration, such as the skills, the behaviors and their parameters, as well as the arena and the objective function. The arena is the area in which the mission is conducted, and is characterized by its shape and some specific features like the floor color, obstacles and lights. The objective function is how an objective value is computed during an experiment.

2.2 Objective of the project

As mentioned, the objective of this project is to develop a tool that would allow a user to create a simple robot swarm mission, as well as selecting the different skills required to accomplish that mission. The application should also allow to modify the parameters of those skills, and the parameters of the behaviors linked to those skills. Additionally, an arena editor would allow to design the mission terrain and modify mission parameters, such as the number of robots for instance, and an objective editor would allow to customize the objective function. Furthermore, those editors could also use the mission data to automatically generate some files, such as an ARGoS file for the arena and the C++ source files for the objective function.

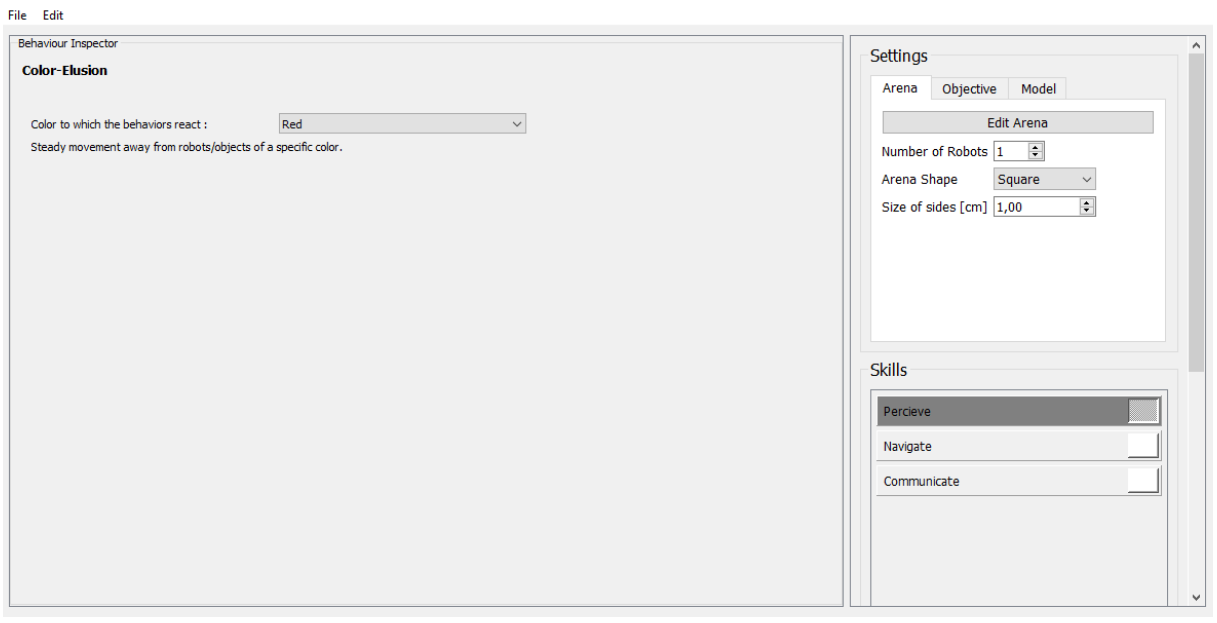
It is worth mentioning that most of the features implemented act more as a proof of concept rather than actual finished features. The goal here is to experiment and have a wide array of features but not necessarily to develop them in depth or with a lot of content. Some possible features, although useful, only require time to implement and are not necessarily as interesting as others.

3 Application and implemented features

This section goes over how the application works and the main implemented features.

The application treats each mission as a separate file (.json), that can be saved and loaded for subsequent modifications. On opening, the user is prompted to either open an existing mission or create a new one.

The interface for the mission editor is divided in two parts. The left part is the main inspector, empty by default, which contains the parameters and information about the current item being modified. On the right is a side panel that contains all items of interest, such as the different settings, skills and behaviors.


Color to which the behaviors react : Red

Steady movement away from robots/objects of a specific color.

Number of Robots 1

Arena Shape Square

Size of sides [cm] 1,00

Figure 2: Typical interface of the application

The user can select a skill or an active behavior, and it will be displayed in the inspector on the left, where the user can then tweak its parameters.

The side panel contains multiple sections, with the settings first, then the available skills, and finally all behaviors. Each skill can be enabled or disabled, depending on its relevancy for the considered mission. As behaviors are linked to skills, only those linked to enabled skills can be opened and modified. When selecting a specific skill, all behaviors linked to that skill are highlighted in blue in the behaviors list, and those that are disabled are grayed out. Similarly, the current active skills are marked with darker gray.

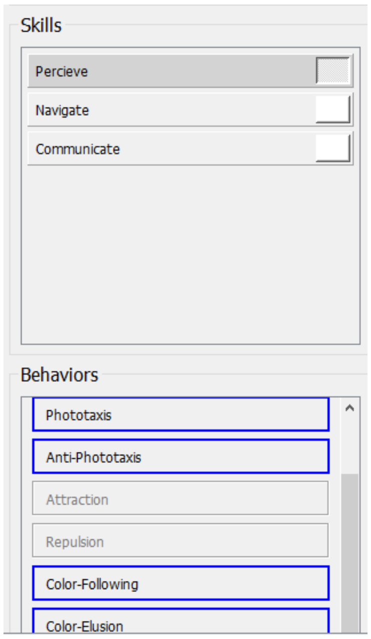
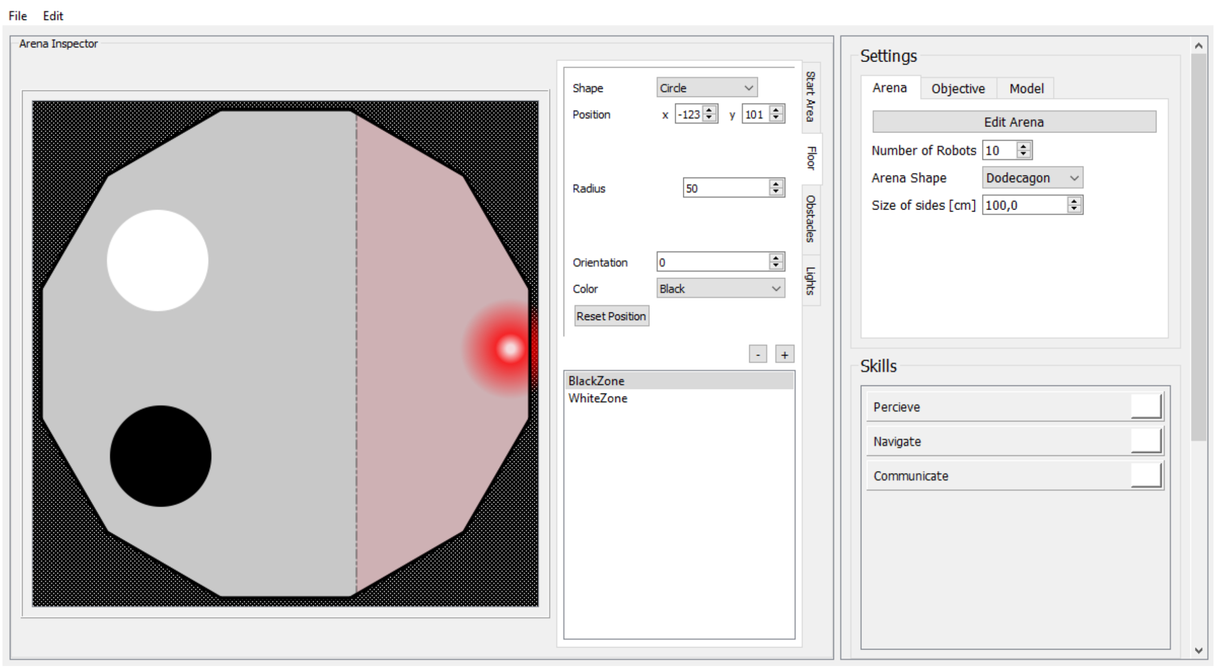


Figure 3: View of the list of skills and behaviors. Here the skill perceive is selected and active, highlighting the linked behaviors below.

The settings panel contains different tabs for different aspects of the mission, such as the arena, the objective function, and the reference model of the robots.

The arena settings tab allows to change some arena properties, as well as to actually edit the arena itself. The user can change its shape (e.g. dodecagon, octagon, ...), and in the main inspector, has access to various arena objects. Those include the start area of the robots, the floor, lights, and obstacles. Each type of arena element can be customized with various shapes, sizes, position and their specific properties.

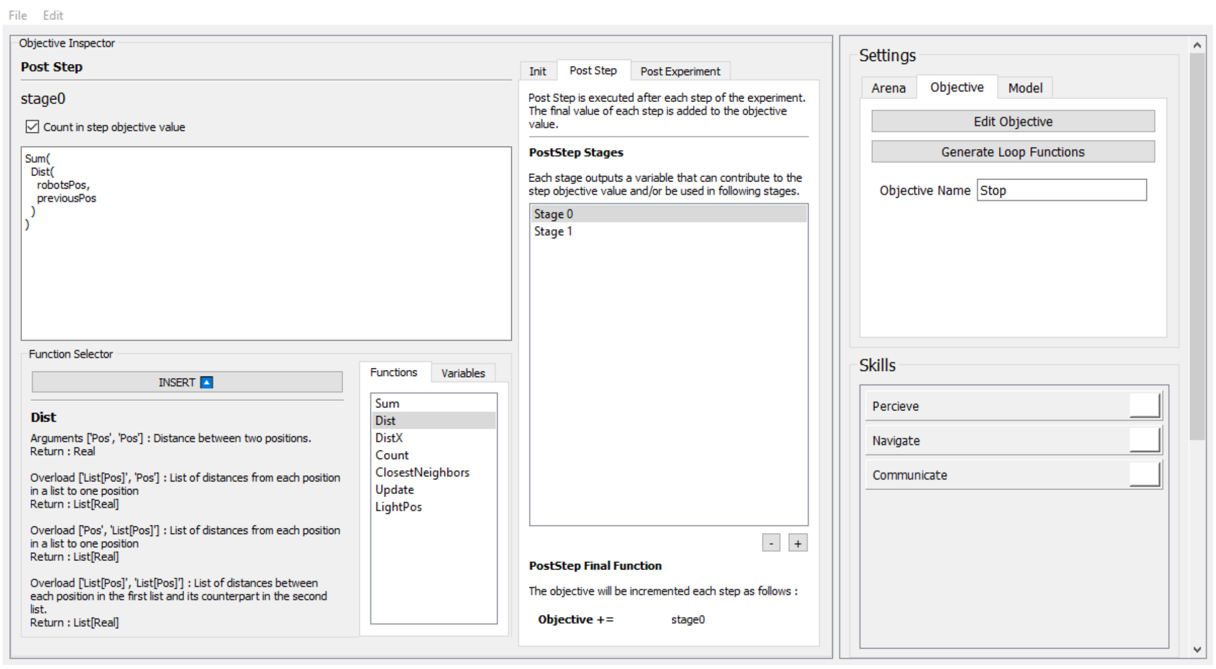


Shape: Circle
Position x -123 y 101
Radius: 50
Orientation: 0
Color: Black

Number of Robots: 10
Arena Shape: Dodecagon
Size of sides [cm]: 100,0

Figure 4: Interface of the arena editor

The objective settings tab allows to edit the objective function of the mission. The objective inspector is divided in three tabs : *Init*, *Post Step* and *Post Experiment*. Each tab allows to customize a specific part of the objective function. The Post Step part corresponds to the function called after each step of the experiment, which can increment the objective value. The Post Experiment part, as its name suggests, is called once the experiment ended, in order to process the objective value one last time. The Init part is optional and is called at the beginning of the experiment. It allows the user to define custom variables, that can be reused and updated later during the experiment.



☑ Count in step objective value

```
Sum(
 Dist(
  robotsPos,
  previousPos
 )
)
```

Dist

Arguments ['Pos', 'Pos'] : Distance between two positions.
Return : Real

Overload ['List[Pos]', 'Pos'] : List of distances from each position in a list to one position
Return : List[Real]

Overload ['Pos', 'List[Pos]'] : List of distances from each position in a list to one position
Return : List[Real]

Overload ['List[Pos]', 'List[Pos]'] : List of distances between each position in the first list and its counterpart in the second list.
Return : List[Real]

Post Step is executed after each step of the experiment. The final value of each step is added to the objective value.

PostStep Stages

Each stage outputs a variable that can contribute to the step objective value and/or be used in following stages.

PostStep Final Function

The objective will be incremented each step as follows :

Objective += stage0

Objective Name: Stop

Figure 5: Interface of the objective editor

The Post Step and Post Experiment parts are also divided into stages. Each stage is one expression that outputs a variable (i.e. the expression in Stage 0 is assigned to the variable `stage0`). This variable is evaluated each time the function is called (either Post Step or Post Experiment) and can increment the objective value and/or be used in following stages. The list of stage for each function is diplayed on the right (where the user can add, remove or select a stage) and the stage itself is detailed on the left, where its expression can be edited.

The core idea of stage editing is about inserting simple built-in functions and variables. At the bottom of the inspector, the list of available functions and variables is presented along with their descriptions. The user can insert the selected function or variable to modify the expression of the current stage.

It is then possible to generate the actual source files, at which point a simple parser ensures all stage expressions are valid and generates the corresponding C++ code.

Finally, a last feature allows the user to generate an ARGoS file from the current mission. Although the generated file is only partially complete, it allows to automatically create the arena and specify some characteristics of the robot's reference model.

It is worth mentioning that although there is already a variety of features implemented, there is still a lot of room for new features, as well as for refinement of the existing features, especially for those that are only partially complete.

An important possible addition could be to improve the cohesion between different parts of the application. For instance, elements in the arena could be directly referenced as variables in the objective editor, or skills and behaviors could impact the available functions of the objective editor. This would help to create a coherent and cohesive package, that has multiple layers of depth for each feature.

Regarding the ARGoS file generation, there are several parts that are not being generated since they would require more context than just the mission itself (e.g. the path to some libraries, or some settings related to AutoMoDe). A possible improvement would be to try and generate those parts as well, by directly asking the user to provide the missing information.

Additionally, implementing a more in-depth objective customization, with many more functions and variables, along with multiple quality of life improvements (such as a smarter code insertion for the function selector) would be pretty high on the list of possible additions.

4 Software architecture and development

It is now possible to focus more on how the project itself was carried out. This sections discusses both the software architecture itself and how the application was developed.

4.1 Architecture

The source code is located in the `src/` directory, while the `resources/` directory contains various data files for the application and its interface, which will be covered in the following. As the application needs a graphical interface, the designed architecture was made to generally follow the principles of the Model-View-Controller design pattern. The source code is therefore divided in those three parts, where each module contains either a model, view or controller, depending on its directory.

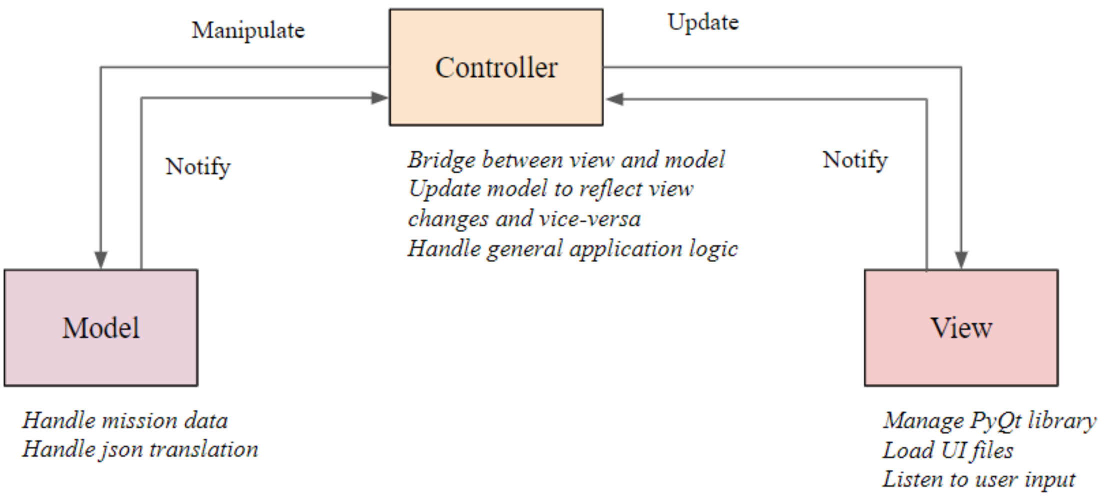


Figure 6: General diagram of the MVC architecture

In this architecture, the model contains all the data about the current mission. A mission model object therefore holds all selected skills and their associated behaviors as well as the arena model and all its properties. Every model object can be converted to *json* in order to be saved to a mission file, and conversely, can also be loaded back in order to load a mission file. The json format was chosen arbitrarily but is well suited for this kind of application since it is easy to read and edit manually.

The view is another crucial part of the architecture since it consists of everything that the user can see. A large part of the code for the views is hidden in *ui* files located in the resources directory. Those files are created using *Qt Designer* and thus are directly compatible with the *PyQt* library. This allows to avoid manually creating all interfaces details and can also simplify some of the simple logic (e.g. some buttons can automatically switch the current view). This also means modifying the views can be done somewhat independently of the code, and faster, which is desirable in a more iterative development where modifications are needed frequently.

However, not everything related to views can be handled via ui files, and more complex elements still require dedicated classes. This is notably the case with the arena editor, where the different arena objects are all extensions of simple graphic items, to allow for a very customized arena display.

Last but not least, the controller part is what actually drives the application and handles all of the logic. In this project, they are typically tasked with handling the link between model and view, when changes in the view need to update the model and vice-versa.

The interactions between the different components in the architecture are mainly handled through the use of events and signals. When a view gets modified by the user, it sends a signal to any listening controller, which can then update the model accordingly. This way, the model and view components have minimal interactions with controllers, and are better separated.

Most of the feature implementations are straightforward, as most of the features consist in simple views that only need to modify model data. An important aspect however is that most of the data (e.g. the different skills, behaviors, etc) are defined in special json data files, and are not hard-coded. This makes it easier to modify, notably to add more content, without the need to change the code, following the same principles as for the views. It could even be envisioned for it to be modifiable by the user himself, given the proper interface.

The more complex features come from the file generation, for the ARGoS and objective functions files.

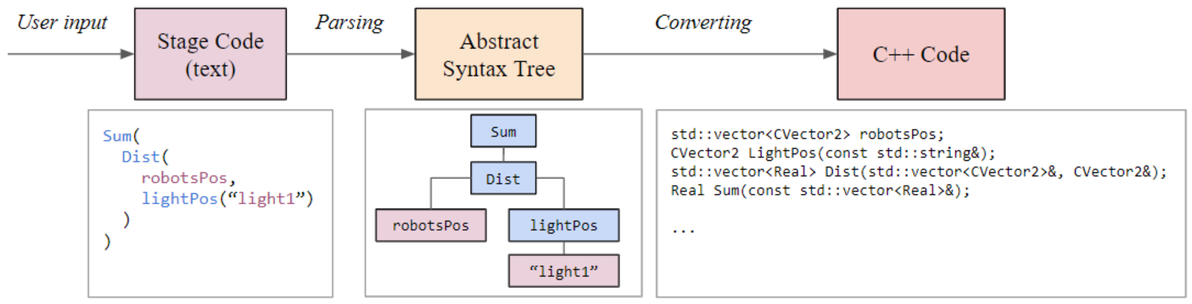
As with any file generation process, the problem is to output a valid file, both in its format and its content. A significant part of the file can be easily handled with the help of templates, notably regarding the C++ file and its header, which contain a lot of code that hardly changes from one mission to another. However, not everything can be handled via templates.

ARGoS files follow the syntax of xml files, and only contain specific pre-determined elements. The generator therefore only needs to translate those different elements from a mission model to the ARGoS format (this mainly concerns the arena and its objects). As ARGoS files only contain independent elements, the generation is simpler than for C++ code, and is done using a built-in python xml library, taking already existing ARGoS files as reference.

The objective function file generation, however, presents an important difference compared to the ARGoS file generation. The generator first needs to parse the user input to get the used functions and variables. The available set of functions and variables is defined in its own file (again to allow easy extension), along with their corresponding

C++ code. The generator can thus fetch the code and insert it into the generated file.

Since the user input for each objective function is text based, the generator uses a parser created from a grammar to identify the hierarchy of used functions and variables. It also allows to perform a first validity check on the types used. The library used to create the parser from the grammar is *Lark*. As the parser goes through the input text, it creates a tree where each node corresponds to a symbol, which in this case is a function or a variable. Each node has its own corresponding C++ code. To generate the whole code, all of the code from each node is merged, which outputs the code for one stage of a function. The process is then repeated for each stages, for Init, Post Step, and Post Experiment.



```
Sum(
  Dist(
    robotsPos,
    lightPos(“light1”)
  )
)
```

```
std::vector<CVector2> robotsPos;
CVector2 LightPos(const std::string&);
std::vector<Real> Dist(std::vector<CVector2>&, CVector2&);
Real Sum(const std::vector<Real>&);

...
```

Figure 7: Diagram of the code generation process

It can be noted that the user code could instead be based on nodes for example, following a visual programming approach, and not go through a text format at all. This is however outside the scope of this project and would not change the core of the file generation, only the graphical interface.

4.2 Development

The development of this project followed an iterative approach, with each iteration seeing the introduction of a new set of features for direct feedback. This is in line with a more agile development, which is to be expected with this kind of project, where many feature ideas come later in the development and more flexibility is required. A waterfall approach would obviously not be suited since the project is more about how many features can be developed in a fixed amount of time rather than how much time is required to develop a fixed set of features.

Each iteration started by designing how a specific feature should work. This is actually the hard part of the iteration, as the project presented more of a modeling problem than a computing problem. As mentioned, implementing a feature, provided it is already designed, is just a matter of creating the interface and ”wiring” it to a controller and a model. Actually designing the feature, its interface, and representing all its required data in the model, however, is not as trivial. Here the challenge is more finding what to do for feature rather than how to implement it.

Nonetheless, some features did have implementation problems to solve. The objective function generator, in particular, was divided in two distinct problems. One of modeling, on how to represent an objective function in a modular way, and one of computing, on how to parse the user input and generate a valid C++ file.

Another common challenge faced in this project was maintaining the code quality. This is achieved by planning refactoring sessions where the code is rewritten to be more stable and consistent. Although multiple refactoring sessions were carried out during the project's development, they were probably not as frequent as they should have been, resulting in additional work for code maintenance.

Additionally, another useful practice would have been to implement some unit testing, which would have both facilitated the refactoring process and helped with early detection of bugs. The project was arguably small enough to do without it, but for larger projects, unit testing remains an indispensable practice.

Perhaps also the iterations where not frequent enough. Shorter iterations would allow to more frequently assess progress, gather feedback, and make timely adjustments. This of course limits the amount of work that can be achieved in a single iteration, but it helps with having clearer, more tangible objectives.

5 Conclusion

In conclusion, the developed software allows the user to design a simple robot swarm mission and automatically generate some files. Most of the desired features have been implemented, although they still present room for improvements. The main challenges in the development were modeling and designing what the features should consist of, and how to build a visual interface to allow the user to adequately use them.

The iterative development allowed to refine the goals as the project was conducted and helped provide regular feedback. Shorter iterations and more formal requirements could perhaps have helped but were not ultimately necessary due to the relatively small scope of the project. Additionally, even though the application ergonomics were not the primary focus, they could probably be improved to ensure a better user experience.

Nonetheless, the final application is still serviceable and its development followed best practices, leading to an overall success in meeting the intended goals and delivering some value to its users.

References

[1] Ligot A. et al. "AutoMoDe, NEAT, and EvoStick: implementations for the E-puck robot in ARGoS3". In: *Technical report TR/IRIDIA/2017-002* (2017). URL: `https://iridia.ulb.ac.be/IridiaTrSeries/rev/IridiaTr2017-002r003.pdf`.