Functional Programming and Multi-Agent Systems

Thor Helms

Kongens Lyngby 2011
IMM-BSC-2011-13
Summary

This project revolves around a multi-agent contest [1] in which several competing teams, each consisting of several agents of different types, try to occupy territory on Mars, which is represented by a graph.

A simulation program for the above scenario has been implemented in a functional programming language (F# via Mono), exploring the advantages and disadvantages of using a functional programming language when making a GUI.

An artificial intelligence (AI) has been created for use in the simulator. This AI is documented and analyzed in this report.

En simulator for det ovenstående scenario er blevet implementeret i et funktionelt programmerings sprog (F# via Mono), og fordele og ulemper ved at bruge et funktionelt sprog til at lave en grafisk brugergrænseflade er blevet undersøgt.

En kunstig intelligens er blevet implementeret til simulatoren. Denne kunstige intelligens er dokumenteret og analyseret i denne rapport.
This is the bachelor project for Thor Helms, student at the Technical University of Denmark (DTU), with Jørgen Villadsen as counselor. The project period is February to June 2011.

Knowledge about functional programming has been retrieved from the course 02157 Functional Programming at DTU, Autumn 2010.

Some knowledge about multi-agent systems has been gathered during a previous attempt at a bachelor project on multi-agent systems in Autumn 2010.

Kgs. Lyngby, June 2011

Thor Helms
Contents

Summary i
Resumé iii
Preface v

1 Introduction 1
  1.1 About the scenario . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.2 About functional programming and F# . . . . . . . . . . . . . 2

2 Scenario details 5
  2.1 The world as a graph . . . . . . . . . . . . . . . . . . . . . . . . 5
  2.2 Teams, agents and roles . . . . . . . . . . . . . . . . . . . . . . 6
  2.3 Disabled agents . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  2.4 Agent actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.5 Occupying a node . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  2.6 Zones . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  2.7 Graph coloring . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.8 Zone scores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.9 Milestones and money . . . . . . . . . . . . . . . . . . . . . . . . 12
  2.10 Agent perceptions . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  2.11 Anonymous objects . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.12 Communication . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
  2.13 Winning condition . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3 User interface 15
  3.1 Requirement specification . . . . . . . . . . . . . . . . . . . . . . 15
  3.2 Graphical user interface . . . . . . . . . . . . . . . . . . . . . . . 17
4 Internals 29
4.1 Agent interface and artificial intelligence . . . . . . . . . . . . . . 29
4.2 Simulation specific types . . . . . . . . . . . . . . . . . . . . . . 31

5 Simulation calculation 39
5.1 Simulation step algorithm . . . . . . . . . . . . . . . . . . . . . . 39
5.2 Prepare perceptions . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.3 Send perceptions and receive actions . . . . . . . . . . . . . . . . 40
5.4 Execute attack and parry actions . . . . . . . . . . . . . . . . . . 41
5.5 Execute all other actions . . . . . . . . . . . . . . . . . . . . . . . 41
5.6 Color the graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.7 Update milestones and scores . . . . . . . . . . . . . . . . . . . . . 42

6 Executing the simulator 43
6.1 Compiling the simulator . . . . . . . . . . . . . . . . . . . . . . . 43
6.2 Running the simulator . . . . . . . . . . . . . . . . . . . . . . . . 44
6.3 Testing/verification . . . . . . . . . . . . . . . . . . . . . . . . . . 44

7 Artificial intelligence 49
7.1 Analysis and strategy . . . . . . . . . . . . . . . . . . . . . . . . 49
7.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
7.3 Testing/results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

8 Discussion 59
8.1 The competition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
8.2 Functional programming . . . . . . . . . . . . . . . . . . . . . . . . 60
8.3 AI performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
8.4 Perspectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

9 Conclusion 63

A Tests and results 67
A.1 Settings for simulations 1 and 2 . . . . . . . . . . . . . . . . . . . . 68
A.2 Simulation 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
A.3 Simulation 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

B Source code 95
B.1 Makefile . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
B.2 Agent.fs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
B.3 AgentHelpers.fs . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
B.4 Agents.fs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
B.5 AISAgent.fs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
B.6 DummyAgent.fs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
B.7 Edge.fs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
B.8 Generics.fs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.9  Graph.fs</td>
<td>122</td>
</tr>
<tr>
<td>B.10 GtkHelpers.fs</td>
<td>128</td>
</tr>
<tr>
<td>B.11 IAgent.fs</td>
<td>130</td>
</tr>
<tr>
<td>B.12 Initial.fs</td>
<td>131</td>
</tr>
<tr>
<td>B.13 Node.fs</td>
<td>132</td>
</tr>
<tr>
<td>B.14 ShapePrimitives.fs</td>
<td>134</td>
</tr>
<tr>
<td>B.15 SimSettingsGeneral.fs</td>
<td>135</td>
</tr>
<tr>
<td>B.16 SimSettingsMilestones.fs</td>
<td>138</td>
</tr>
<tr>
<td>B.17 SimSettingsRoles.fs</td>
<td>141</td>
</tr>
<tr>
<td>B.18 SimSettingsWindow.fs</td>
<td>147</td>
</tr>
<tr>
<td>B.19 SimTypes.fs</td>
<td>150</td>
</tr>
<tr>
<td>B.20 SimTypesDrawing.fs</td>
<td>154</td>
</tr>
<tr>
<td>B.21 Simulation.fs</td>
<td>159</td>
</tr>
<tr>
<td>B.22 SimulationSteps.fs</td>
<td>170</td>
</tr>
<tr>
<td>B.23 SimulationView.fs</td>
<td>175</td>
</tr>
<tr>
<td>B.24 SimulationWindow.fs</td>
<td>178</td>
</tr>
<tr>
<td>B.25 TS.fs</td>
<td>182</td>
</tr>
</tbody>
</table>
This report describes the development of a simulator for a multi-agent system (MAS), and an artificial intelligence for use in the simulator. The scenario is from an international competition in multi-agent systems [1]. The scenario is described in section 1.1 and in more detail in chapter 2.

Both the simulator and the artificial intelligence is developed in the language F# [7], which is created by Microsoft for their .NET framework, and supported by the cross-platform Mono [4]. Section 1.2 describes the F# language and the reasons for using it in this project.

1.1 About the scenario

The scenario simulates a number of robots, operating on Mars. The purpose of these robots are to find and occupy the best water wells on the planet. There are multiple teams, consisting of multiple types of robots (agents) each. All teams consist of the same number and types of robots. Some robots might be able to gather information about the world, and some robots might be able to sabotage opponent robots. There are a number of predefined roles, which are described in section 2.2 on page 6.
Introduction

The world is represented by a graph, where the nodes in the graph represent water wells, and the edges represent roads between water wells. Even though it is on Mars and the robots should be able to drive directly from any one point to any other point on the surface, the graph isn’t complete, but it is guaranteed to be connected.

1.2 About functional programming and F#

F# is Microsoft’s take on functional programming, and is in fact multi-paradigm, encompassing functional, objective-oriented and imperative programming [7]. It targets Microsoft’s .NET framework, and also runs on the open source, multi-platform Mono framework. When compiled, it will produce the exact same code as C#, which enables the two languages to be used together once compiled. This also means that F# is able to use all .NET and Mono frameworks, which are mostly object-oriented code.

Functional programming is often based on lambda calculus [10]. This is true for the functional parts of F# also. Functional functions have no side-effects, but in F# it is possible to define functions with side-effects, which is necessary for the object-oriented parts of the language.

Functions are first-class citizens in F#, and it is possible to define higher order functions, which means that a function takes another function as argument and/or returns a function.

In functional languages, types are usually inferred by the compiler, which means that the programmer doesn’t have to explicitly define types. In F#, this is true, but the programmer is able to define the types if needed. Types can also be generic, in which case the type will be inferred at compile-time.

Values are usually immutable in functional languages, but in F# it is possible to define a value to be mutable. There are some limitations when using a mutable value, but some can be overcome by using the ref type, which is a wrapper for a mutable value [8].

Values in F#, and most other functional languages, can be defined as tuples and as discriminated unions. An example [7] of a discriminated union can be seen below:

```fsharp
type A =
```
Discriminated unions allow the programmer to easily define compilers, or tree’s of types, which in an object oriented style would have to implemented as a hierarchy of classes, allowing functional code to be more succinct.

F# and most other functional languages, also support lists and records as first class citizens, which means that collections of values can be easily ordered and used in calculations.
In this chapter, the details of the scenario are explained, and any decisions to change the official scenario are argued for.

2.1 The world as a graph

The world in which the agents operate is represented as a graph, with a number of nodes and edges. Internally, the nodes are represented with a coordinate, making it possible to draw the graph. A restriction is placed on the edges, in that no two edges may cross when using the node-coordinates. This will allow the graph to be viewed as a 2D map, where the edges represent roads, and there are no intersections except at the nodes. The nodes represent points of interest, in this case water wells.

Each node in the graph will have a weight value, determining the quality of the water well. Higher is better.

Each edge in the graph will likewise have a weight value, determining the quality of the road and how much energy is required to traverse it.
2.2 Teams, agents and roles

In a simulation, several teams will operate. Each team consists of a number of agents. Agents can have different roles, and the different roles have different stats such as strength, visibility, energy and health. All teams consist of the same number of agents, with the same roles on each team for fairness.

Energy is used to perform most actions. The agent can recharge its energy, and the recharge rate will be expressed in percentage of the agent’s maximum energy level.

Health determines how robust the agent is, and thus how difficult it is to destroy. Health can only be restored when an agent is repaired by another agent from the same team. If an agent has no health left, it is referred to as being disabled. Otherwise, it is referred to as being active.

Visibility determines how far the agent can see. All nodes that a graph search would find when going to at most the same depth as the visibility range, will be visible for the agent. All edges connecting visible nodes, and all agents on visible nodes, will be visible as well.

Strength determines how hard an agent can attack, and thus destroy enemy agents. For instance, an agent with 4 strength would be able to reduce enemy agents’ health by 4 per attack.

The different roles are able to perform different actions. Actions are defined in section 2.4.

Table 2.2 lists the predefined agent roles.

2.3 Disabled agents

When an agent has no health left (because its been sabotaged by enemy agents), it is disabled and various limitations is placed on it. Its health can be restored fully when repaired by another agent, and the agent will thus cease being disabled.
2.4 Agent actions

As mentioned earlier, when an agent is not in the disabled state, it is in the active state.

As mentioned earlier, there are different actions the agents can perform. When performing an action, there is a certain percentage chance that the action will fail.

The actions are defined below.

2.4.1 Skip

Does nothing. This action will always succeed. Actions that have failed, for any reason, will be considered to be of this type.
2.4.2 Recharge

Attempt to recharge the agent’s energy. This action will fail if the agent has been successfully attacked by another agent. The amount of energy restored depends on whether the agent is disabled or not, and on the maximum amount of energy the agent is able to store.

2.4.3 Attack

Attempt to sabotage an enemy agent, identified by team and agent ID. A target agent on the same node as the attacking agent is required. An attack can be parried. Default energy cost is 2. A disabled agent can’t attack.

2.4.4 Parry

Parry any expected attacks. Default energy cost is 2, and will be charged regardless of number of incoming attacks. A disabled agent can’t, and wouldn’t gain anything from performing the parry action, as its health can’t be lower than 0.

2.4.5 Goto

Go to a neighbor node, determined in a parameter with the ID of the neighbor node. The weight of the edge connecting the current node and the neighbor node determines the energy cost. If the agent doesn’t have enough energy to traverse the edge, its energy will be reduced (if any energy is left). The default energy cost for a failed goto-action is 1.

2.4.6 Probe

Request the weight of the node the agent is at. Once a node has been probed, its weight is visible for the entire team at all times, otherwise its weight is regarded as unknown. Default energy cost of the probe action is 1. A probe action will fail if the probing agent is disabled or being attacked by another agent.
2.4 Agent actions

2.4.7 Survey

Request the weight for all edges connected to the node the agent is at. Once an edge has been surveyed, its weight is visible for the entire team at all times, otherwise its weight is regarded as unknown. Default energy cost of the survey action is 1. A survey action will fail if the surveying agent is disabled or being attacked by another agent.

The official scenario description defines the survey action as requesting the weight for “some” visible edges. As this is an unspecified amount, its been chosen to see the edges directly connected to the node the agent is at.

2.4.8 Inspect

Request the various stats for all enemy agents on the same node and on neighboring nodes, compared to the agent’s position. Once an enemy agent has been inspected, its stats and possible actions are visible for the entire team at all times, otherwise all stats and actions for the agent is regarded as unknown. Default energy cost of the inspect action is 2. An inspect action will fail if the inspecting agent is disabled or it has been successfully attacked by another agent.

2.4.9 Buy

Attempt to perform an upgrade. There are four types of upgrades, each of which upgrades either the agent’s maximum energy, maximum health, visibility range or strength. Only agents who can perform the attack action are able to upgrade its strength (As it is useless for all other agents). When performing an upgrade, the chosen stat is increased by 1. In case of upgrading maximum energy or health, the current energy and health is also increased by 1. Default energy cost of the buy action is 2. It is not possible to perform the buy action if the agent is disabled or is being attacked.

2.4.10 Repair

Repairs another agent on the same team. The agent that needs to be repaired must be identified with its agent ID, and both agents must be on the same
node. This action fully restores the target agent’s health, and brings it out of the disabled state if it was previously in it. It is not possible for an agent to repair itself. Default energy cost for the repair action is 2. The action will fail if the repairing agent is being attacked.

2.5 Occupying a node

A node is occupied by the team who has the most active agents on the node. In the case of a tie, the node is not occupied. A node occupied in this manner will be referred to as being directly occupied, or directly dominated.

Nodes can also be occupied if there are no active agents on them, and one of two conditions are met. The first condition is that the node has at least two neighbor nodes that are directly occupied by the same team, in which case it gets dominated by the same team. In case of a draw in the amount of occupied neighbor nodes by team, the node will not be dominated. If a node is occupied in this manner, it will be referred to as being indirectly occupied, or indirectly dominated. The second condition is that the node lies within an otherwise unoccupied zone, defined below. All nodes directly next to the nodes in the unoccupied zone, but not in it, will be referred to as a frontier. All nodes in the frontier must be occupied by the same team, for the zone to be occupied, in which case the team dominating the frontier will also dominate all nodes in the zone. No enemy agents must be able to move into the zone without crossing the frontier (Enemy agents standing on a node on the frontier doesn’t count). This means that no nodes in the zone are allowed to have any active agents.

If an enemy enters a zone dominated by one team, or breaks the frontier, the zone will no longer be dominated.

2.6 Zones

A zone is defined as a connected subgraph within the graph, all dominated by the same team, or all dominated by no team, where the subgraph can’t be extended with more nodes while still maintaining the requirements.
Graph coloring is an algorithm that determines which team, if any, occupies the various nodes. The color of a node represents the team dominating it. An example of a colored graph may be seen in figure 2.1.

Coloring of the graph, i.e. determining the domination of the various nodes, takes place in five steps. First the existing coloring is reset so no nodes or edges are dominated by any team. Second the directly occupied nodes are colored. Third the indirectly dominated nodes are colored. Fourth all unoccupied zones are determined, and colored if they comply with the restrictions given above. Fifth all edges connecting two nodes of the same color, are colored with the same color. Edge colors have no meaning, and they are only colored for making it visually easier to determine zones in the simulator.

Figure 2.1: An example of a colored graph

Zone scores

After the graph has been colored, the zone scores can be calculated using the colors. A zone score is the sum of node weight’s in an occupied zone. The scores are not actually calculated per zone, but per team. The zone scores are used when calculating the step score for each team.
2.9 Milestones and money

In a simulation, certain milestones may be defined. For instance, a milestone could be that a team has a total zone score of at least 50 in one simulation step. When a team reaches a milestone, the team will receive a reward in the form of money, which can be used to buy upgrades for the agents. The reward size may vary, depending on whether the team reaching a milestone is the first team reaching that particular milestone or not. Milestones should be defined before the simulation starts.

Several of the same type of milestone may be defined, for instance one for 5 successful attacks and one for 20 successful attacks, yielding (possibly) different amounts of money.

There are six types of milestones. These are:

- Zone values in a single step.
- Probed nodes in total.
- Surveyed edges in total.
- Inspected enemy vehicles in total.
- Successful attacks.
- Successful parries.

2.10 Agent perceptions

In the beginning of each simulation step, each agent is given a perception, i.e. their view of the world. The contents of the perceptions are:

- The current step number.
- The team score.
- The amount of money the team possesses.
- The stats for the agent itself, including the actions its role permits it to do.
2.11 Anonymous objects

- A list of visible nodes.
- A list of visible edges.
- A list of enemy agents that have been inspected by any agent on the same team.
- A list of messages, containing the ID of the sending agent and the content of the message.

The list of visible nodes and edges are shared with any friendly agent in the same zone as the agent itself. This encourages team work by creating and keeping zones. Nodes, edges and agents are anonymized if necessary (see below) before the perception is sent to the agent.

In the official scenario description, agents would also receive a list of all nodes and edges that has been surveyed or probed by its team. This has been disregarded, and it will be up to the agents to remember this information, and share it with the team.

In the official scenario description, agents would also be told the result of its last attempted action. This has been disregarded as well, and it is up to the agent itself to determine the success of the last action.

2.11 Anonymous objects

In perceptions, some nodes, edges and enemy agents may be anonymous to the agent receiving the perception. When nodes haven’t been probed, edges haven’t been surveyed or agents haven’t been inspecting by any agent on the same team as the receiving agent, they are anonymous. That is, unknown objects visible to the agent are anonymous. Anonymous nodes and edges will have their weight set to 1 when they are anonymous. All stats of an anonymous enemy agent will be set to 1, and their actions will not be visible.

When zone scores are determined, the weight of an anonymous node for a team will be regarded as being 1.
2.12 Communication

Communication agent-to-agent can and should preferably be done through the simulator. Communication consists of messages, where each message contains a receiver and some content. As mentioned earlier, the agent perceptions contains a list of incoming messages. When agents respond to the simulator with their requested action, they should also send a list of outgoing messages. Because of this structure, there will be a delay in the reception of the messages of one simulation step.

The receiver of a message can be either a single agent, or a broadcast to all agents – in either case, only agents on the same team as the sender can receive the message, simulating safe communication.

In the first scenario draft [2], communication was to happen through the simulation server, and there was a limitation on the amount of messages an agent could send in each simulation step. In the latest version of the scenario description [3], both of these requirements has been removed. In this implementation of the simulator, the first requirement is kept and the second has been removed.

2.13 Winning condition

The winning team is the team with the highest score after the last simulation step. The score is calculated as the sum of zone scores and money in the previous steps:

\[
Score = \sum_{s=1}^{steps} Zones_s + Money_s
\]
As the scenario is somewhat abstract, the simulator should somehow display the status of the simulation. For that, a graphical user interface (GUI) has been created. This section describes the design and implementation of the GUI.

3.1 Requirement specification

The following is a list of requirements for the simulator:

- Automatically run a simulation calculation.
- View already calculated simulation steps.
- Start and stop the playback of a calculated simulation.
- Add new artificial intelligences to the program without recompiling the simulator, and allow the program to automatically discover new artificial intelligences.
- The simulation must be displayed graphically.
• All agents on the same team should have the same color – each team should have different colors.

• Change various simulation variables:
  – Number of teams.
  – Number of agents on each team.
  – Number of simulation steps to be calculated.
  – Chance that agent actions will fail, in percentage.
  – Maximum response time for agents when requesting actions, in milliseconds.
  – Recover rate for agents in percentage, both when active and when disabled.
  – Number of nodes on the map.
  – Grid size of the map/graph.
  – Min/max node weight.
  – Min/max edge weight.
  – Energy cost of the various actions.
  – Price of the various upgrades.
  – Roles or types of the agents – either selected from the preset roles, or the user should be able to precisely define the stats and actions.
  – Colors for the different teams.
  – Select from a list of artificial intelligences, which one controls which role on each team.
  – Define milestones in the simulation. Select from one of the six types, along with what amount is needed to trigger the milestone, and the reward for the first team(s) and subsequent teams to reach the milestones.

• Generate a random map/graph using the simulation variables.

• Display the progress of the simulation:
  – The score and amount of money for the different teams.
  – The map/graph, colored by which teams dominate which areas.
  – Stats for the various agents:
    * Strength.
    * Current energy.
    * Max energy.
3.2 Graphical user interface

Given that the simulation needs to be visually available, a GUI has been made. Included in the GUI is the ability to change all simulation variables, as a contrast to loading the simulation variables from a configuration file, which would have to be created beforehand. The GUI consists of two windows: The main window, which displays the progress of the running simulation; and the simulation settings window, which enables the user to change the simulation variables.

Throughout the simulator, some generic functions are used, which aren’t simulation specific but rather just nice to have. The source code for these can be seen in appendix B.8 on page 118.

3.2.1 Main simulator window

The main simulator window, figure 3.1, consists of a menubar in the top, view-selection in the left side, simulation graphics view in the right side, and some simulation-view controls in the bottom of the window. Furthermore, it contains a status bar in the very bottom of the window, displaying the progress of the simulation calculation. The main simulator window is implemented as a class named SimulatorWindow, located in a module of the same name. The source code can be seen in appendix B.24 on page 178. The simulator window, as well
as the simulation settings window, use some helper functions to more gracefully handle certain functions in the Gtk library, as seen in appendix B.10 on page 128. A more detailed description of the various parts follows.

![Main simulator window](image)

**Figure 3.1: Main simulator window**

### 3.2.1.1 Menubar

The menubar contains four items. The first, “New simulation”, opens the simulation settings window, which can start a new simulation calculation. The second, “Open simulation”, opens an already calculated and saved simulation. The third, “Save simulation”, saves the progress of the current simulation – it will save only the view of the simulation, and thus a simulation calculation can’t be resumed. Simulation views are saved as vector graphics, and will take up a lot of space for even rather small simulations. The fourth item, “Exit”, will stop the current simulation and quit the program.
3.2 Graphical user interface

3.2.1.2 View-selection

The view-selection pane in the left side of the window allows the user to change which world-view to see. In figure 3.1 the “Simulation” view is selected (The selection is hidden behind the menu bar), and the user can thus see the status of the entire simulation. In figure 3.2 the user has selected to view the perception for an agent. Note that both figures display the same simulation in the same step.

The user has to enable agent perception views in the simulation settings window in order to see these.

If an artificial intelligence wishes to display its world view, it can be added to and can be selected from the view-selection pane.

![Figure 3.2: Agent perception in main simulator window](image)

3.2.1.3 Simulation graphics view

In the right side of the window is the simulation view, visually displaying the simulation status as seen in figure 3.1. In the top left corner of the simula-
tion view is the scores, money, etc. for the various teams displayed when the “Simulation” view is selected.

If the simulation view is too big to be displayed completely in the window, a scrollbar will appear and the user will be able to see the entire simulation view anyway.

The simulation graphics view is implemented as a class named `SimulationView`, located in a module of the same name. The source code for this can be seen in appendix B.23 on page 175. The graphics view draws a number of shape primitives, consisting of rectangles, lines, circles and text. Along with a description of each shape, is a description of the color to be used. Text and lines have a single color, but circles and rectangles may have two colors, one for the stroke color and one for the fill color.

The shape types are defined in a module; the source code can be seen in appendix B.14 on page 134.

A description of the various entities in the simulation view follows later in section 3.2.2.

### 3.2.1.4 Simulation-view controls

The simulation-view controls in the bottom of the window consist of three buttons and a scale-selector.

The first button, “Play/pause”, toggles the playback of existing simulation views. When enabled, it will automatically go to the next simulation step until no more exists, displaying three simulation steps per second.

The other two buttons, “Previous” and “Next”, displays the previous and next simulation steps respectively, in respect to the currently viewed step.

The scale-selector allows the user to select which simulation step to display in larger steps than the “Previous” and “Next” buttons. The user can drag the slider to select a simulation step, or click on the scale to the left/right of the slider, in order to go 10 steps forward or backward.
3.2 Graphical user interface

3.2.1.5 Status bar

The status bar, in the very bottom of the window, displays the progress of the simulation calculation.

3.2.2 Graphics

A description of how the various entities in the simulation is displayed in the simulator follows here. There are three types of entities when viewing the simulation: Agents, nodes and edges.

Nodes are displayed in a grid, the size of which is defined by the user. The position of the nodes have no meaning for the agents in the simulation, and it is used only to display the simulation.

3.2.2.1 Agent graphics

The different agents distinguish in their stats, their color, which teams they have been inspected by and their ID. All but the ID is possible to see in the simulator view. Figure 3.3 gives an example of how an agent could be displayed. The color determines its team, so the displayed agent is on the blue team.

![Figure 3.3: An example of an agent as viewed in the simulator](image)

Agents are displayed as a circle, with its team color in the center. From the center of the circle, some amount of lines extend, forming a larger circle. These lines displays the various stats of the agent. There is one line for each one stat, for instance a visibility range of 1 will be represented with a single line.

The long white lines represent the agent’s health, and as such some lines may be missing, meaning that the agent is not at full health.

The short white lines represent the agent’s energy, and as such some lines may be missing, meaning that the agent is not at full energy.
The long colored lines represent the agent’s visibility range.

The short colored lines represent the agent’s strength.

Around the stats-circle, there may be some amount of colored lines, displaying which teams the agent has been inspected by. The agent in figure 3.3 has been inspected by the red team, and only the red team.

### 3.2.2.2 Edge graphics

Edges in the graph distinguish by their weight, the team dominating the edge as described earlier, the nodes they connect and which teams they have been surveyed by. All of this information is visible in the simulation view, an example of which is displayed in figure 3.4 (Nodes are not visible).

![Figure 3.4: An example of an edge as viewed in the simulator](image)

A line is drawn between the two nodes the edge connects, thus representing the edge itself. The line has the color of the dominating team, i.e. the same color as the two nodes if they have the same color. If an edge isn’t dominated by a team, its color is white.

At the center of the line, the circumference of a circle is displayed in the same color as the line, with the weight of the edge in the center of the circle.

Around the circle, some colored lines may be displayed, displaying which teams the edge has been surveyed by. The edge in figure 3.4 has been surveyed by the red, and only the red team.

### 3.2.2.3 Node graphics

The various nodes distinguish by their weight, their ID, their neighbors, the team dominating the node and which teams the node has been probed by. All of this information, except for the ID, is visible in the simulation view, as seen in figure 3.5.
3.2 Graphical user interface

The node itself is displayed as a colored circle, in the same color as the dominating team. If a node isn’t dominated by any team, its color is white. In the center of the circle, the node’s weight is displayed.

Around the node circle, some colored lines may be seen. These lines display which teams the node has been probed by. The node seen in figure 3.5 has been probed by the green, and only the green team.

3.2.3 Simulation settings

In this section, the window for changing the simulation settings is described. It consists of three panes, where only one at a time is visible, and five buttons in the bottom of the window, as can be seen in figure 3.6. The three panes describe the general settings, the role definitions and the milestone definitions respectively. Which pane is visible can be changed by clicking on the appropriate pane name in the top of the window.

When the simulation settings window is opened, it will have some default values entered for all variables. The default values are loaded from the text file “DefaultSettings.txt”, which is expected to be located in the same folder as the simulator. Each time the simulation settings window is opened, it will have its values set to the contents of the above file.

The simulation settings-window is implemented as a class named SimSettingsWindow, located in a module of the same name. The source code can be seen in appendix B.18 on page 147. Each of the three panes is implemented as a class, located in their own modules.

3.2.3.1 General settings

In the general settings-pane, most of the simulation variables can be selected, as seen in figure 3.6. The pane consists of three columns.
The left-most column contains the number of teams and agents, the simulation length in ticks, the chance in percentage that agent actions will fail, the maximum response time for the artificial intelligences when queried about their wanted action, and the recharge rate in percentage for both active and disabled agents.

The middle column contains the map variables, which are the number of nodes, grid width and height, and node and edge minimum and maximum weights. It also contains a checkbox, where the user can toggle viewing of the agents’ perceptions. Per default, viewing the agents’ perceptions is disabled.

The right-most column contains the cost of performing the various actions, and the price of buying the various upgrades.

The class for the general settings pane is named *SimSettingsGeneral*, and it is located in a module of the same name. The source code for the general settings-pane can be seen in appendix B.15 on page 135.

Figure 3.6: General simulation settings
3.2 Graphical user interface

3.2.3.2 Defining roles

In the agent roles-pane, the user is able to precisely define all of the roles used in the simulation, as seen in figure 3.7, by either choosing from a predefined role, or defining a custom role with custom stats and available actions. The three actions “Skip”, “Goto” and “Recharge” are always available for the agents, and thus can’t be toggled for the agents. The reason for this is, that the scenario describes robots moving around on Mars, and thus requires them to be able to move, including getting more energy or choosing not to move. Other than these three actions, all remaining actions can be enabled or disabled for all agents.

The user is also able to set the colors for the various teams, and choose which artificial intelligence controls which agent on the different teams.

Note that the visible amount of roles and teams in the agent roles-pane, is defined in the general settings-pane. If the user changes the number of teams or roles, the agent roles-pane will reflect this change.

The roles-pane is defined as a class named SimSettingsRoles, and located in a module of the same name. The source code for the agent roles-pane can be seen in appendix B.17 on page 141.

![Figure 3.7: Defining the milestones to be used in a simulation](image-url)
3.2.3.3 Defining milestones

In the milestones-pane, the user can define the various milestones to be used in the simulation, as seen in figure 3.8.

The user can add a milestone by clicking the “Add milestone” button in the top of the window, or remove the last milestone by clicking the “Remove last milestone” button. The milestones are displayed in a table layout, with each row defining a single milestone.

For each milestone, there are four settings: The type of milestone, the number needed to trigger the milestone, and the reward for the first team(s) and the subsequent teams to achieve the milestone. There are six types of milestones, each selectable in a drop-down box in the left-most column. The three other settings for each milestone can be set in the other three columns.

Note that the same type of milestone can appear several times, and even with the same trigger-value if the user so chooses.

The milestones-pane is defined as a class named SimSettingsMilestones, and located in a module of the same name. The source code for the milestones-pane can be seen in appendix B.16 on page 138.

![Figure 3.8: Defining the roles to be used in a simulation](image)
3.2 Graphical user interface

3.2.3.4 Buttons

In the bottom of the simulation settings-window, there are five buttons.

The “OK” button will close the simulation settings-window and start a simulation using the entered values.

The “Cancel” button will close the simulation settings-window, and throw away the entered values.

The “Defaults” button will set all values to the default values, which are loaded from the file “DefaultSettings.txt”. The file is expected to be located in the same folder as the simulator.

The “Open settings” button will open a file-chooser window, where the user can select a file with simulation settings to open. When opening a such file, all values in the simulation settings-window will be replaced with the values from the chosen file, assuming the correct file format.

The “Save settings” button will open a file-chooser window, where the user can select where to save the entered values. The program will generate a settings-file that can be opened later. Note that it is possible to overwrite the default settings file, in order to get a new set of default values.
User interface
This section describes some internal parts of the simulator. In section 4.1 the interface for external AI’s is described. In section 4.2 the different custom types used in the simulator is described.

4.1 Agent interface and artificial intelligence

In order to be able to automatically load agents into the program without having to recompile the simulator, Microsoft’s MEF [14] framework is used. For this framework to work, an interface has been declared, that the artificial intelligences must implement in order to hook up to the simulator. This means that at least some part of the artificial intelligences has to be a class.

4.1.1 Agent interface

The agent interface defines three methods that any artificial intelligence wanting to use the simulator, must implement. The source code can be seen in appendix B.11 on page 130.
The first function in the interface is called MakeStep. This function is used when the agent needs to perform an action. The agent is given a perception, and must return an action and a list of messages it wishes to send. This function is called once per simulation tick, for each agent.

The second function, SetMaxResponseTime, will tell the agent how many milliseconds it is allowed to use when calculating a single action response.

The third function, SetNumNodesEdgesSteps, will tell the agent the number of nodes and edges in the simulation, and for how many steps the simulation will take place.

Its usage, in F#, is as follows:

```fsharp
[<ExportMetadata("Name", "Some name for the agent here")>]
[<Export(typeof<IAgent>)]

type SomeAgent() = ... // class definition
   ... // Variables and functions
   interface IAgent with
     ... // Interface functions
```

### 4.1.2 Automatically loading agents

Defining an interface for the agents is not enough, they must also be loaded into the program. For this, a module has been created, the source code for which can be seen in appendix B.4 on page 108.

The module will automatically load the agents, and enables the rest of the program to get an agent object based on its name, and get a list of all available artificial intelligences.

The simulator settings-window uses the list of artificial intelligences from this module to allow the user to choose which artificial intelligence to use for each agent on each team.

### 4.1.3 Dummy agent

In the program, a dummy agent is included. This is the default agent to be used in simulations. It implements the agent interface, but it is included directly in
the program. The source code for the dummy agent can be seen in appendix B.6 on page 115.

The dummy agent will randomly move around, probe, survey, inspect and recharge. It will only attempt to perform actions it is allowed to perform according to its role. Each of the actions mentioned above will be chosen with the same probability. When choosing to move, a random neighbor node is chosen.

4.2 Simulation specific types

In the simulation calculation, some custom types are used. The most notable of these are the Agent, Edge and Node types, which represent the objects seen in the GUI. Another important type is the Graph type, which represents a list of nodes and a matrix with edges, and thus defines a mathematical graph with an adjacency matrix for the edges.

The source code for the type definitions can be seen in appendix B.19 on page 150. Some functions have been developed to convert the Agent, Edge, Node and Graph types to a list of shapes to be used by the simulation view. The source code for these functions can be seen in appendix B.20 on page 154.

4.2.1 Action type

The Action type defines the different actions the simulation supports. This type is to be used by agents when answering with their requested action.

Some actions take one or two arguments, but most take none.

The Goto action needs an integer argument, representing the ID of the node the agent wants to move to.

The Attack action needs two integer arguments, representing the team ID and agent ID of the wanted target.

The Repair action takes an integer argument, representing the agent ID of the wanted target. Since repairs can only be performed to an agent on the same team as the repairing agent, only one argument is needed.

The Upgrade action takes an argument of a custom type. The custom type
defines the type of upgrade wanted, and can be either Battery (energy), Sensor (visibility range), Shield (health) or SabotageDevice (strength).

The other actions are Skip, Recharge, Parry, Probe, Survey and Inspect.

### 4.2.2 Agent type

The Agent type is defined as a record, containing the team and agent ID, the node ID of the node the agent is currently at, variables for the agent stats, a list of possible actions the agent can perform and a list describing which agents have inspected the given agent.

A module for manipulation of the Agent type has been created, which can be seen in appendix B.2 on page 97.

### 4.2.3 Edge type

The Edge type is defined as an EdgeInfo Option type. The reason for this is that not all possible edges will be present in the edge adjacency matrix of a graph, and thus there must be a way to distinguish existing from non-existing edges.

The EdgeInfo type is defined as a record, and contains the values describing an edge, which in this simulation is the two nodes it connects, plus its weight and a list of teams that the edge has been surveyed by. It also contains a variable determining the team that dominates the two nodes it connects, and thus the edge, if any.

A module for manipulation of the Edge and EdgeInfo types has been created, which can be seen in appendix B.7 on page 116.

### 4.2.4 Node type

The Node type is defined as a record, containing the information relevant for the simulation, which is the node ID, the weight of the node (i.e. the quality of the water well it represents), the agents that are currently located at the node, a list of the teams that has probed the node, the coordinate of the node when displayed and which team, if any, controls the node.
4.2 Simulation specific types

The *Node* type has an associated module for manipulation of it, the source code for which can be seen in appendix B.13 on page 132.

4.2.5 Graph type

The *Graph* type consists of a number of nodes and edges. The nodes are kept in an array, with their respective node ID as the index in the array. An array is used instead of a list, because many random access calls are expected, which has a runtime of $O(1)$ in an array and $O(n)$ in a list.

The outgoing edges for a single node is kept in an array, and all arrays of outgoing edges are kept in another array. Thus the data structure for the edges in a graph is a 2D-array, with all arrays having the same length (the number of nodes). This ensures that the 2D-array can be seen as an adjacency matrix. The edges could also be kept in a 2D-list instead of an array, but for the same reasons as the nodes, it is kept in a 2D-array. As all edges are considered bidirectional, the number of edges in a graph is half the amount of edges in the adjacency matrix.

The *Graph* type has an associated module for manipulation of it, the source code for which can be seen in appendix B.9 on page 122. Unlike the modules for *Edge*, *Node* and *Agent* manipulation, the functions in the *Graph* module manipulates the actual *Graph* objects, instead of returning a new and modified version. The reason for this is, that, for the given *Graph* object in the functions to remain immutable, a copy of the entire graph would have to be made. As the size of a graph is $O(n^2)$, and there are many manipulations in each simulation step, this would very quickly lead to much redundant copying and writing of data. This of course means that the state of a graph can’t be kept unless the entire graph is copied, unlike immutable datatypes where a reference to the object is all that needs to be saved. This has little influence though, as the previous simulation steps as seen in the graphical user interface is saved as lists of shape primitives.

Notable functions in the *Graph* module are the various functions used by the coloring algorithm, a function to find a zone for a starting node, a few functions for performing actions, and a function to create a new graph based on certain variables, as described below.
4.2.5.1 Find zone algorithm

The algorithm for finding a zone in a graph requires a starting node and two functions: One for determining when to accept a node into the zone; and one for determining whether or not a zone is invalidated by a found node. For instance, in the graph coloring algorithm, no active enemy agent should be able to move into the zone without passing a directly or indirectly colored node in the same color as the zone. This means that a found zone is invalidated if there is an active enemy agent in it. When determining a zone for sharing perceptions between friendly agents in the same zone, there is no way to invalidate a zone, and the zone is determined strictly by zones of the same color as the agents.

The algorithm maintains a list of accepted nodes, and a list of nodes to be investigated. Only if a node is accepted into the zone will its neighbors be added to the list of nodes to be investigated.

4.2.5.2 Coloring algorithms

The coloring algorithms consist of five different algorithms, used by the coloring algorithm in the simulation calculation.

The algorithm to reset coloring simply sets the dominating team for all nodes and edges to be non-existing.

The algorithm to color directly colored nodes calls a function from the Node module, which will determine the dominating team for the individual nodes, and set the dominating team in the nodes appropriately.

The algorithm for coloring indirectly dominated nodes will look at all uncolored nodes with no active agents, and determine whether or not they need to be colored, according to the color of its neighbor nodes.

The algorithm for coloring zones will find the zone for all uncolored nodes with no active agents that lie next to a colored node. The color of the neighbor node will determine which team the zone should belong to, if one such is found. The zone algorithm will accept nodes with no color and no active agents into the zone, and invalidate the zone if a non-colored node with an active agent is found, or if a colored node of an enemy team is found.

The algorithm for coloring edges will call a function from the Edge module on all edges, to determine whether or not they should be colored.
4.2 Simulation specific types

4.2.5.3 Action algorithms

The *Graph* module contains three functions that perform an agent action: A function for moving an agent from one node to another, a function for performing the survey-action from a given position and a function for performing the inspect-action from a given position.

The algorithm for moving an agent will check to see if the agent has enough energy to traverse the appropriate edge. If so, the agent will be moved. If not, the agents energy will be reduced by an amount defined in the simulation settings (The default value is 1).

The algorithm for surveying from a given position for a certain agent, will find all the outgoing edges from the given position, and then find the reverse edges as well. It will then set all of these edges to be surveyed by the agent if they aren’t already surveyed by an agent on the same team. The number of successfully surveyed edges will then be counted and added to the appropriate counter for that team.

The algorithm for inspecting from a given position for a certain agent, will find all the agents on the neighbor nodes as well as the agents on the same node. It will then set all of these agents to be inspected by the agent if they aren’t already inspected by an agent on the same team. The number of successfully inspected vehicles will then be counted and added to the appropriate counter for that team. Note that friendly agents will not be able to inspect each other.

4.2.5.4 Create graph algorithm

The algorithm for creating a new graph takes several arguments, such as maximum and minimum node and edge weights, the grid size and the number of nodes. It will create a list of all possible coordinates available within a grid of the given size, shuffle it and use it as the coordinates for the nodes it will create. This ensures that no two nodes have the same coordinate.

When the nodes have been given coordinates, it will perform a Delaunay triangulation [12] in the nodes, which will create the edges needed. For this, an external library is used [13]. It will then add these edges to the graph, which initially had no edges.

When an edge and a node is created, it will be assigned a random weight within their respective limits.
4.2.6 Milestone type

The Milestone type denominates one of the six predefined milestones available in the simulation, as well as three values: The trigger-value and the rewards for the first team(s) and all subsequent teams to reach the milestone.

In the simulation calculation, an extended type, MilestoneStatus, is used. MilestoneStatus consist of a Milestone and a set of integers. The set of integers denominates which teams has achieved the milestone already, if any.

4.2.7 Message type

The Message type is what the agents use to communicate with each other. The Message type consists of a recipient, which can either be a single friendly agent or all friendly agents, plus a MessageContent value.

The MessageContent type should enumerate all possible types of communication the agents could be imagined to send. As such, the content of the MessageContent is to be defined along with the development of the artificial intelligence. Unfortunately, a change of the MessageContent type would require the simulator to be recompiled.

4.2.8 Percept type

The Percept type is defined as a record type, and represents the perception given to the agents in each simulation step. It contains the current step number, the team score, the amount of money the team has, the stats and possible actions of the receiving agent using the Agent type, a list of visible nodes and edges (including the shared perceptions), a list of all inspected agents, a list of all milestones in the simulation using the MilestoneStatus type, and a list of messages consisting of an integer representing the agent ID of the sender, and the MessageContent from the sent message.

4.2.9 TeamStatus type

The TeamStatus type is defined as a record, containing information about the score, money and number of successful probes, inspections, surveys, attacks and parries a team has made.
The `TeamStatus` type has an associated module for manipulation of it, the source code for which can be seen in appendix B.25 on page 182.
Chapter 5

Simulation calculation

A simulation is calculated in subsequent steps, with the results of one step being used in the next, until the desired simulation length has been reached. This chapter describes the algorithm used for calculating a single step. The source code for all algorithms in this section can be seen in appendix B.21 on page 159.

5.1 Simulation step algorithm

The algorithm for calculating a single simulation step proceeds as follows:

1. If the desired simulation length hasn’t been reached:
2. Prepare the agent perceptions
3. Send the perceptions to the agents, and retrieve their requested actions and messages they wish to send
4. Perform all attack and parry actions
5. Perform all other actions
6. Color the graph
7. Update team scores and money
8. Draw the resulting graph
9. Recursively calculate the next simulation step

Step 8 in the algorithm will convert the graph to a list of shapes, and add it to the module that stores the different simulation steps, the source code for which can be seen in appendix B.22 on page 170.

Note that all actions other than attack and parry, are executed in the same step of the algorithm. The actions are of course executed one at a time, and not all at once, and thus a few of the actions may have influence on each other, such as the Goto and Inspect actions. If an agent performs the Inspect action, another agent can move one step away from the inspecting agent, and thus out of range, in the same simulation step. Which action will be executed first will thus have influence in the simulation calculation. However, this is how the official scenario description describes the algorithm. The same is true for the Repair and Goto actions.

5.2 Prepare perceptions

The algorithm for preparing the agents’ perceptions start by finding all the agents in the entire graph, and finding all the nodes and edges their visibility allows them to see. It does so by maintaining a list of nodes, and extending this list with all neighbors of all nodes in the list, until the correct visibility range (depth) has been reached. During this, it maintains the list of nodes as a set, so as to not add the same nodes several times. The algorithm will then find the zone for all agents, if any, and retrieve the perception for all friendly agents in the same zone.

The algorithm will also make sure to fill in all other fields of the Percept type correctly. It will return a Percept list with the perception for all agents.

5.3 Send perceptions and receive actions

The algorithm to send perceptions to agents takes a map of artificial intelligences to be used, and a list of Percept values, as created in the above algorithm. It will
5.4 Execute attack and parry actions

give each agent its perception, determining the appropriate artificial intelligence to query from the agent map, and collect their requested actions. These actions will be removed from the resulting list of actions with a certain percent chance (the action failed), determined in the simulation settings.

The algorithm will return a list of the requested actions for all agents that haven’t failed to perform their action, and a list of all messages sent by all agents, regardless of whether or not their action failed.

In the simulation settings, a max response time for agents can be defined. However, it is completely up to the artificial intelligences to respect this limit, as the algorithm for sending percepts and gathering actions won’t enforce it.

5.4 Execute attack and parry actions

The algorithm for performing attack and parry actions, will in reality only perform the attack actions. It will reduce the energy for all parrying agents though, regardless of whether or not they were attacked.

At first, the algorithm will create a list of all parrying agents and reduce their energy. It will then execute each attack, while determining whether or not the target is parrying. If so, the number of successful parries for the target’s team will be increased. Else, the number of successful attacks for the attackers team will be increased, and the target’s health will be reduced.

5.5 Execute all other actions

The algorithm for executing all other actions, other than attack, parry and of course skip actions, will determine which type of action is attempted to be performed by an agent. It will then determine whether or not the agent is able to perform the action, based on different parameters for all actions. Some actions can’t be performed if the agent has been attacked, and some actions can’t be performed when the agent is disabled. If the agent can perform the requested action, it will be executed accordingly.
5.6 Color the graph

The coloring algorithm consists of five steps, each calling a function from the Graph library:

1. Reset the coloring
2. Color all directly dominated nodes
3. Color all indirectly dominated nodes
4. Color all zones
5. Color all edges

5.7 Update milestones and scores

The algorithm for updating milestones and scores will first calculate the zone scores for all teams, not distinguishing between individual zones though. It will do so using the colors determined in the coloring algorithm. If a team hasn’t probed a node it dominates, its value will count as 1, otherwise it will achieve its weight as value.

Having calculated the zone scores, the milestones are updated one at a time. An entire milestone is updated at once, to ensure that two teams reaching the same milestone at the same time as the first teams, both will receive the reward for being the first team.

After updating the milestones, and subsequently the different team’s money, the scores will be calculated, as the sum of the score from the previous step, the zone score achieved in the current step and the total amount of money in the current step.
Chapter 6

Executing the simulator

The complete simulator consists of many different files, some of which are to be accessible by the artificial intelligence and thus need to be separated from the main executable created.

The executable needs a starting point, which in this case is a file created for this specific purpose. This file will simply create an instance of the main simulator window, show it and start the Gtk run-loop. The source code for this file can be seen in appendix B.12 on page 131.

6.1 Compiling the simulator

The simulator, when compiled properly, will consist of a DLL file and an EXE file. The DLL file contains all functions needed by the artificial intelligence, including functions allowing them to draw their world view in the GUI. The EXE file contains the files relevant for displaying the simulation, editing the simulation settings and performing the simulation calculation.

Compiling the simulator is easy when using the attached Makefile, see appendix B.1 on page 95. This will compile both the DLL and the EXE files to a folder
specified in the Makefile. It requires the following files to be in the specified folder:

- Triangulator.dll (Delaunay triangulator)
- System.ComponentModel.Composition.dll (MEF)

Furthermore, the simulator expects the file “DefaultSettings.txt” to be in the folder, containing the default simulator settings in the format used by the simulator.

It is also required that the compiling/running system has the Gtk library installed, including the Pango and Cairo modules.

### 6.2 Running the simulator

In a Windows system, the simulator can be run by executing the executable file. Note that the simulator may attempt to print to a console.

On other systems, Mono is required in order to run the simulator. In this case, the application can be run by entering a terminal, going to the appropriate folder, and executing the file by writing mono followed by the name of the executable.

### 6.3 Testing/verification

To ensure that the simulator works correctly, some tests has been performed. However, as the simulation is very complicated and F# is a very high level and robust language, only graphical tests has been performed. The tests performed are described in this section.

#### 6.3.1 Graph coloring

It is essential that the graph coloring algorithm works correctly, due to the fact that it determines the scores and thus the winning team.
Only active agents should be able to dominate nodes. In figure 6.1 an example of this can be seen. As seen, only the active agents can dominate nodes. Note: There is one disabled and two active agents on each team.

![Figure 6.1: Only active agents can dominate nodes](image)

Ties in the graph coloring algorithm should result in undominated nodes. An example of this can be seen in figure 6.2 in which an indirect tie is occurring on the top, number two from the right, node. Note that some of the agents are disabled.

In figure 6.3 an example of the zone coloring can be seen. The top and bottom nodes in the right side are only next to one directly occupied node each, however the two nodes are correctly colored as they are part of a zone. It can also be seen that there are several nodes that are not in a colored zone, as both green and red agents can reach them without crossing a node dominated by the other team first.

### 6.3.2 Other tests

The user can enter invalid values in the simulation settings. If so, the simulation shouldn’t start. In figure 6.4 the result of invalid settings can be seen. The program checks to see if there are more nodes than possible in the given grid size, and whether the minimum value is smaller than the maximum value for
node and edge weights.

If a single team reached a milestone first, it should get the proper reward for being the first team, and all subsequent teams should receive the reward for not being the first team. In figure 6.5, an example of this can be seen. All three teams has performed 4 inspections, and thus triggered a milestone, with a reward of 20 for the first team and 10 for the other teams.

If several teams reach the same milestone at the same time, they should all receive the reward for being the first team. An example of this can be seen in figure 6.6, where all three teams has reached an arbitrary milestone of 0 inspection, and thus all three teams has received the reward of 20 money.

In figure 6.7 an example of a shared perception can be seen. The three green agents are in the same zone, and thus all three agents receive the same perception. Note that the two agents to the left has a visibility range of 1, and thus shouldn’t be able to see the nodes in the right side of the graph.
6.3 Testing/verification

Figure 6.3: Coloring of zones

Figure 6.4: Result from invalid simulation settings

Figure 6.5: Only one team got the high reward for a milestone

Figure 6.6: Several team reached the same milestone at the same time
Figure 6.7: An example of a shared perception for the green team
Chapter 7

Artificial intelligence

Having created a simulator, it naturally needs something to simulate. It contains an extremely simple agent, known as the *DummyAgent*, but simulating using only this agent isn’t very interesting. This chapter focuses on the development of an artificial intelligence that will be able to perform better than the dummy agent, but still remain very simple, i.e. it won’t perform any complex calculations, regarding such things as the intentions of others. The created AI has been named *Aggressive Information Seeker*, or AIS for short.

7.1 Analysis and strategy

A simple agent using a constant strategy has been created. This section analyses and describes the strategy. One of the dangers when using a constant strategy, is that the opponents may be able to figure out the strategy and use this to their advantage. This danger is known and accepted for the developed agent.
7.1.1 Money and the Buy action

The objective for the agents is to maximize the team score over the course of the entire simulation. As mentioned earlier in this document, the score in each step consists of the zone scores for the various teams, plus their amount of money. As such, the more money spent, the lower the potential score will be. Even more so if the money is spent early in the simulation. This means that in order to spend money, the agents need to ensure that they will gain more than they lose. This is a potentially very complex calculation. Ignoring this calculation and thus ignoring the *Buy* action, will simplify the calculations for the agents, while the consequences will remain unknown and possible positive. In this implementation of a rather simple agent, the *Buy* action will thus be completely ignored.

7.1.2 Information sharing

In order for the agents to perform the best decisions, they must have a complete view of the world. There are two ways to receive information about the world: See it for your self, or hear-say. Some extent of first hand knowledge is given to all agents in each step. Second hand knowledge will have to be distributed via messages. Using the message system implemented in the simulation, there will however be a delay of one simulation step when sending messages.

Certain aspects of the simulation remain static, and certain aspects are dynamic. The static aspects of interest are the node and edge weights, and the stats/actions for enemy agents. The only dynamic aspect of interest is the position for all other agents. As the position of all inspected agents is known, via the perception given in each step, the information that need to be shared is that of nodes and edges. There are two cases in which information about a node or edge should be shared: Either the node/edge hasn’t been seen before, in which case its completely new to the agents; or its been recently probed/surveyed, in which case information about it is new. With this in mind, it is possible to restrict the amount of messages the agents need to send (bearing in mind that in the first scenario description, there was a limit on the amount of messages the agents could send in each simulation step).
7.1 Analysis and strategy

7.1.3 Information is power

For the agents, the following is true: The better information about the world they possess, the better decisions can be expected from them. In this simulation, they can know the state of the world, but not the intentions of other agents, which they can only guess.

As mentioned above, certain aspects of the simulation are dynamic and certain aspects are static. The static aspects need to be investigated, via the Probe, Survey and Inspect actions. As this information is static, the agents should remember this at all times.

The dynamic aspect, being the position of the other agents, will be automatically given to the agents of a team, for all agents that team has inspected.

To ensure the best decisions throughout the simulation, all information should be gathered as early as possible. Another advantage of gathering information early is that the full value of a node isn’t given until its been probed.

7.1.4 Aggression

Some agents have the ability to sabotage (attack) enemy agents, and thus potentially disable them, which in turn will make sure there are certain actions they can’t perform, and that they can’t help their team by dominating nodes. To determine whether or not an attack actually pays off can be a very complex calculation. One might instead assume that on average, attacks pay off, and thus agents should attack as much as possible. The most important question might be which enemy to attack, as this is likely to have a bigger effect than the choice of attack or not. However, this is likely to be a complex calculation as well. One might instead assume a static priority, or simply attack at random. The AI developed here will attack a random target if possible.

7.1.5 Movement in the graph

Movement in the graph, and pathfinding in it, can’t accurately be done using the weights of the nodes. These can be used as an approximation of the shortest path from one node to another, but a more accurate path can be found if assuming that the agent should have the same energy level when ending the path as when starting the path.
The amount of energy each agent can receive in each step is assumed to be known (and static, as per the rules of the simulation), denoted here as $E_{\text{perstep}}$. When traveling from one node to a neighbor node, the time taken, assuming the same energy level in the end as in the beginning, is determined by the formula:

$$Traveltime(a, b) = \frac{Weight(a, b)}{E_{\text{perstep}}} + 1 \quad (7.1)$$

Using this formula, the length of the paths in the graph can be calculated more precisely.

### 7.2 Implementation

The above reasoning has been implemented into a rather simple agent, with a priority of actions to perform.

#### 7.2.1 MakeStep algorithm

The $MakeStep$ function is the main function in the agent interface. It receives a perception, and answers with the requested action.

The implemented algorithm augments the agent’s memory with the information received in the perception, and then calls another function to determine the requested action based on the (believed) state of the graph.

The $MakeStep$ algorithm proceeds as follows:

1. Augment the believed state of the world with the information of nodes and edges sent from friendly agents.
2. Reset the coloring of the graph.
3. Augment the believed state of the world with the nodes and edges the agent is able to see, store the new information.
4. Augment the believed state of the world with the state of all inspected agents.
5. Calculate all possible and meaningful actions in the current state.
6. Determine the requested action using the calculated actions from step 5, using the PrioritizedAction algorithm as described below.

7. Draw the belief-state to the GUI.

8. Return the requested action from step 6, and the new information from step 3 in the form of messages.

### 7.2.2 PrioritizedAction algorithm

To determine which action the agent should perform, a very simple priority is used. It requires an Agent object and a list of possible actions, which it will use to determine whether or not an action is possible and meaningful to perform.

The algorithm (and priority) proceeds as follows:

1. Recharge if the agent has less energy than required to perform any of the actions in steps 2-7.
2. Probe.
4. Inspect.
5. Attack.
6. Parry.
7. Repair.
8. Determine the best position to be at in the graph, using the NextNode algorithm, and move towards it.

### 7.2.3 NextNode algorithm

The NextNode algorithm will determine the estimated best position to be at in the graph, while also considering the amount of time it takes to reach that position. The travel-time $T$ to a node will result in a delay factor of $0.99^T$.

The value for the nodes is calculated using an ad-hoc method, considering several different aspects such as the proximity of friendly and enemy agents, whether
the node is unprobed and the agent can probe, and more. Note that when the agent is disabled, only nodes with a friendly agent that can repair will have a positive value.

The NextNode algorithm requires an Agent object and a list of possible actions, and will return an action, which is always either a Goto action or the Recharge action. It proceeds as follows:

1. Calculate all shortest paths from the current node, to be used when calculating the delay factor.
2. Calculate node values and apply the delay factor.
3. Select the target node, i.e. the node with the highest value when also considering the delay factor.
4. Find the first node in the shortest path to the target node.
5. If the target node is the node the agent is currently at, or if the agent doesn’t have enough energy to move towards it, then return the Recharge action. Else, return the Goto action, with the node from step 4 as parameter.

7.3 Testing/results

To show how the AIS agents perform, this section describes two simulations in which the AIS agents has to fight against either three teams of dummy agents or three other teams of AIS agents. Primarily their ability to gather information and sabotage enemy agents is analyzed.

7.3.1 Simulation 1

In this simulation, one team of AIS agents are up against three teams of dummy agents. Each team has 10 agents, 2 of each of the predefined roles. There are 50 nodes in a 12x6 grid. The generated graph has 125 edges, and the length of the simulation is 1000 steps. Every 100th step, starting from step 0, can be seen in appendix A.2 on page 70. The milestones used can be seen in appendix A.1 on page 68 as well as in table 7.3.6.
7.3 Testing/results

<table>
<thead>
<tr>
<th></th>
<th>Simulation 1</th>
<th>Simulation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>All nodes probed:</td>
<td>151/-/-/- steps</td>
<td>312/252/168/285 steps</td>
</tr>
<tr>
<td>All edges surveyed:</td>
<td>44/-/-/- steps</td>
<td>79/71/76/88 steps</td>
</tr>
<tr>
<td>All enemy agents inspected:</td>
<td>83/-/-/- steps</td>
<td>69/63/68/75 steps</td>
</tr>
</tbody>
</table>

Table 7.1: Gathering information in the two simulations

7.3.2 Simulation 2

This simulation has the same basic setup as simulation 1, except that there are 4 teams with AIS agents and no teams with dummy agents. The generated graph has 125 edges.

Every 100th step, starting from step 0, can be seen in appendix A.3 on page 82.

Note: A lot of agents from the red, green and yellow teams gather at a single node rather fast (before the 100th step), and for the remainder of the simulation, this node is very populated.

7.3.3 Gathering information

One of the main targets with the AIS agent is that it should prioritize gathering information over all other actions.

The most important piece of information is the probing of nodes, as this will increase the potential score for the team. The figures in table 7.3.3 gives an idea of how fast this information can be gathered, against both passive (simulation 1) and aggressive (simulation 2) enemies. The best case is from simulation 1, in which it took 151 steps to probe 50 nodes, with 2 agents able to probe. This gives approximately one probe per 6 steps per agent, which means an average travel/recharge time of 5 steps between each probe for each agent. The worst case is in simulation 2, in which it took the red team 312 steps to probe all nodes. This gives approximately one probe per 12 steps per agent, which means an average travel/recharge time of 11 steps between each probe for each agent. This is however against aggressive opponents, which means that the two red agents that are able to probe, might have been disabled some of the time. In both simulations however, the probing agents may also have spent time surveying edges, as they are able to perform that action as well, and it is prioritized higher than movement.
Surveying of edges shows the same tendencies as probing nodes: Aggressive opponents increase the amount of time taken by at most a factor 2. In the best case (simulation 1) it takes 44 steps to survey all 125 edges, with 10 agents able to survey. This means that there is approximately 3.5 steps between each survey for each agent in the best case, and 7 steps in the worst case. This is faster than probing because the agents can survey multiple edges at a time, but only probe one node at a time.

Inspecting enemy agents take approximately the same amount of time with or without aggressive opponents. With two agents able to inspect, and 30 enemies to inspect, there is approximately 4.2 steps between each inspection in the best case, and 5.5 in the worst case. As with surveying, this is faster than probing because multiple opponents can be inspected at the same time.

Information about the layout of the graph is needed as well; however, this information is automatically gathered when the agents attempt to probe and survey everything. Once all nodes has been probed, or all edges surveyed, all nodes has with 100% certainty been visited, and all information about the graph has been available at one time or another to at least one agent on the team. Assuming perfect sharing of information, all agents on the team is expected to know the layout of the entire graph once all nodes has been probed or all edges surveyed.

### 7.3.4 Attacking/repairing

The AIS agents are supposed to be aggressive and thus attack enemies very often.

In simulation 1, all enemies of the single AIS team are dummy agents, and thus not able to attack or parry. The result of this, and the aggression from the AIS agents, is that all but three enemy agents are disabled in the end of the simulation. As can be seen in table 7.3.4 37 attacks successful attacks has been performed. As two agents was able to attack, this gives one attack per 54 simulation steps per agent. This figure suggests that the agents aren’t as aggressive as they could be.

In simulation 2, a lot of agents from team 1, 2 and 4 are gathered at a single node through most of the simulation. The cause of this is the priority of actions, such
### 7.3 Testing/results

<table>
<thead>
<tr>
<th></th>
<th>Simulation 1</th>
<th>Simulation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of attacks:</td>
<td>37/0/0/0 steps</td>
<td>357/394/80/328 steps</td>
</tr>
<tr>
<td>Total number of parries:</td>
<td>0/0/0/0 steps</td>
<td>11/126/1/234 steps</td>
</tr>
</tbody>
</table>

Table 7.2: Total number of attacks and parries for the various teams in the two simulations

<table>
<thead>
<tr>
<th></th>
<th>Simulation 1</th>
<th>Simulation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total score</td>
<td>140.677/11.263/5.337/2.931</td>
<td>61.678/58.289/111.381/48.474</td>
</tr>
<tr>
<td>Money in last step</td>
<td>24/4/0/0</td>
<td>22/28/24/26</td>
</tr>
<tr>
<td>AvgMin</td>
<td>11.67/0.73/0.53/0.29</td>
<td>3.97/3.03/8.74/2.25</td>
</tr>
</tbody>
</table>

Table 7.3: Scores over the course of an entire simulation, i.e. 1000 steps

that attacking and repairing is prioritized above moving away from the lump of agents. This means that the agents that can repair, will be caught on the node as long as there are agents to repair, and agents that can attack will likewise be caught, as long as there are enemies to attack. This yields an infinite loop given the right circumstances. The effect of this is seen in the number of successful attacks and parries for those teams, which are rather large compared to team 3, with the exception of the number of parries for team 1. In this case, aggression might be too high, due to the simple priority of actions.

#### 7.3.5 Forming groups/zones

When moving around, the agents should attempt to form groups, and thus increase the team score, while trying to not stand at the same node as other active agents.

Table 7.3.5 shows the ending score and amount of money for all teams in both simulations. The last row of the table displays the minimum possible average zone-score per agent per step, which is calculated using the following formula:

$$Avg_{Min,i} = \frac{Score_{i,end} - Money_{i,end} \cdot Steps}{Steps \cdot Agents_i} \quad (7.2)$$

In simulation 1, the AIS agents each scored at least 11.67 points in zone-score per step, which is higher than the uncooperative maximum of 9 (the highest possible
Table 7.4: Time taken for completion of milestones

<table>
<thead>
<tr>
<th></th>
<th>Simulation 1</th>
<th>Simulation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone score: 50</td>
<td>40/-/-/- steps</td>
<td>112/-/58/514 steps</td>
</tr>
<tr>
<td>Probed vertices: 25</td>
<td>59/-/-/- steps</td>
<td>102/87/88/105 steps</td>
</tr>
<tr>
<td>Surveyed edges: 100</td>
<td>12/215/-/- steps</td>
<td>18/14/27/26 steps</td>
</tr>
<tr>
<td>Inspected vehicles: 20</td>
<td>44/-/-/- steps</td>
<td>21/17/16/37 steps</td>
</tr>
<tr>
<td>Successful attacks: 40</td>
<td>-/-/-/- steps</td>
<td>136/182/194/151 steps</td>
</tr>
<tr>
<td>Successful parries: 30</td>
<td>-/-/-/- steps</td>
<td>-/216/-/125 steps</td>
</tr>
</tbody>
</table>

In simulation 2, the three teams that went into partial deadlock scored well below the team that didn’t. The agents that wasn’t in deadlock did however cooperate to some extent, which is evident from both the images from simulation 2 as well as the data in table 7.3.5.

### 7.3.6 Achieving milestones

An unintended, but positive, side-effect of gathering information and attacking enemy agents, is that the milestones (if any such are defined) can be achieved. This will result in money, which in turn will return in a higher score for the team.

Table 7.3.6 shows the amount of time take for the various teams to achieve the various milestones.

In simulation 1, the AIS team was very fast to achieve the first four milestones, while only a single dummy team achieved a single milestone.

In simulation 2, all teams took almost the same time to achieve the 2nd, 3rd, 4th and 5th milestones, while the other two (zone score and parries) were up to chance. This suggests that information-seeking and aggression will yield two thirds of the milestones consistently.
The previous chapters has had a rather objective view on the development process and product. In this chapter, a more subjective view on the process and product is given, along with a few comments on the future potential of multi-agent systems.

8.1 The competition

In the beginning of this project, there was no official simulator available. According to the time-schedule for the competition, the simulator should have been released prior to the start of this project though, and as such it wasn’t possible to tell when and in what state an official simulator would be released. This enforced the creation of a simulator in this project. Looking back, the creation of this took far too much time and removed time from the development of an artificial intelligence.

Along with the release of the official simulator, a changed scenario description was given. The new scenario was changed on some key points to simplify the development of AI’s, but as the development of the simulator in this report had already begun, some of the old requirements was kept.
One of the changes in the new scenario was the removal of enforced communication through the simulation server, and the removal of mixed teams. This would enable agents to communicate with friendly agents without a 1-tick latency. In this project, the agents aren’t strictly required to communicate through the simulation server, but it is possible and suggested as the teams can consist of several different types of AI’s. This reduces flexibility in the development of the AI’s, but highlights another interesting problem: multi-agent systems with delayed communication, which will be further discussed in section 8.4.

Time in the simulation is discrete, which causes both the complexity of solutions and realism of the simulation to be reduced. However, if the simulation ran in realtime it would increase the complexity of not only the AI development, but also the simulation process as the agents might be located on remote machines.

8.2 Functional programming

Using functional programming made the development process much faster than for instance using C#. One reason for this is that the code written is simply more succinct, and type definitions are rarely used, yielding shorter lines. Another reason is that many otherwise recurring errors, such as null-pointers, just doesn’t happen in a functional language. The task of memory management is also removed.

Comparing F# to C#, F# is faster to write, able to do nearly everything C# can, plus more. F# is however not able to exploit the .NET 4.0+ features as of writing, as it only exists to .NET 3.5-. It should run exactly as fast, as both languages are compiled to the same byte-code, which also means that they can work together once compiled. Both languages are able to run on Mono, which means they can run on multiple platforms. Up to Mono 2.8, F# needs to be installed separately, but in Mono 2.10+, F# is bundled. However, the F# plugin for MonoDevelop 2.4 doesn’t work with Mono 2.10 as of writing.

As it is possible to use all existing .NET and Mono frameworks, creating a GUI was easy, but it also required a lot of the code to be object oriented. A clear distinction between functional code and object oriented code has been sought, making most functions outside of classes purely functional with no side-effects (With the functions in the Graph module being a large exception).
8.3 AI performance

The AIS AI uses only a single strategy throughout the simulation, which opponents can exploit to their advantage. Furthermore, the strategy used doesn’t involve any form of planning or communication with friendly agents about plans, which means that the team will only get the optimum score by chance. The AIS AI also doesn’t try to determine the strategy of opponents. This means that it isn’t a truly multi-agent AI, but merely reacts to the current state of the world. It does however perform better than the dummy agent by a large margin, and would thus serve as a great replacement for it, for other AI’s to compete against.

8.4 Perspectives

Multi-agent systems has many uses in the real world. Robotic cars for instance can be considered agents in a multi-agent system, where all other agents have varying characteristics.

The development of a true multi-agent theory, considering both friendly and enemy agents, will likely have huge impacts on areas such as financial markets, robotic sports and cars, and to some extent the analysis of human behavior.

Multi-agent systems with delayed communication can for instance be used to improve the performance of high-speed trading, where several systems can be made to cooperate, and thus exploit the strategies of other high-speed trading systems. Due to the speed at which trading occurs, communication might not be possible in real-time though, which is why the notion of delayed communication can be used to improve this area. Furthermore, any multi-agent system with insecure communication might have to perform delayed communication: imagining that there is some state with (relatively) secure communication, the strategy until the next state with secure communication should be agreed upon before entering a state with insecure communication. For example, in (American) football, the players of one team agrees upon a strategy before each play, but changing strategy mid-play will reveal the strategy to the opponents.
Conclusion

During this project, a simulator and an artificial intelligence has been created. The scenario description that this project built upon changed during the project period, which has caused the scenario used to be a mix of the two different versions of the scenario description. In detail, mainly the newest version of the scenario was used, with an added requirement from the old version, which increased the complexity of the AI’s to be used. This in turn opened up another interesting problem, that of delayed communication in multi-agent systems. This however is in slight conflict with an assumption in the simulator, namely that of implied secure communication, whereas delayed communication is mostly relevant for systems with insecure communication.

The simulator is flexible in that it allows the different AI’s to display their world model in the GUI. Ensuring that AI’s can display whatever they want has however meant that saving a simulation to the disk will have to save the graphics, instead of simply the simulation state. This means that a saved simulation will take up much more disk space than would be necessary if only the actual simulation state was saved. One might assume that this will also have an influence on how much memory a running simulation will consume.

The language F# has been used to create the simulator and the AI. This has in both cases meant a mix of purely functional programming and object oriented programming. To truly be able to take advantage of the potentials of functional
programming, a clear distinction between functions with and without side effects has been kept, while trying to keep the amount of functions with side effects to a minimum.

The GUI was created using existing object oriented frameworks, but this proved to be no problem as F# is able to handle that with grace, and possible even easier than its object oriented counterpart, C#. As F# is able to do nearly everything C# can and more, one might ask: Why use C# at all? Multiparadigm languages gives the programmer more flexibility, but at what cost?

The created AI, named Aggressive Information Seeker, uses a single strategy, which makes it vulnerable to opponents that can predict behavior – but one is always vulnerable to smarter opponents. The strategy used is more than adequate to beat the built-in dummy agent, which is the only other AI is has been tested against. Against itself it does have a flaw though, in that several agents from several teams might lump together at the same one node, and become deadlocked in perpetual attacks and repairs.

The AIS agents cooperate to some extent, but the amount of cooperation is limited in that they don’t communicate about plans at all.
Bibliography


Appendix A

Tests and results

This section contains the screenshots from two simulations. Only every 100th step is kept here.

The first section displays the settings used. The next two sections display the two simulations, the first of which consists of one team of AIS agents and three teams of dummy agents, the second of which consists of four teams of AIS agents.
A.1 Settings for simulations 1 and 2

Figure A.1: General settings used in simulations 1 and 2
A.1 Settings for simulations 1 and 2

Figure A.2: Agent settings used in simulations 1 and 2

Figure A.3: Milestone settings used in simulations 1 and 2
A.2 Simulation 1
Figure A.4: Simulation 1, step 0
Figure A.5: Simulation 1, step 100
A.2 Simulation 1

Figure A.6: Simulation 1, step 200
Figure A.7: Simulation 1, step 300
Figure A.8: Simulation 1, step 400
Figure A.9: Simulation 1, step 500
Figure A.10: Simulation 1, step 600
Figure A.11: Simulation 1, step 700
Figure A.12: Simulation 1, step 800
Figure A.13: Simulation 1, step 900
Figure A.14: Simulation 1, step 1000
A.3 Simulation 2
Figure A.15: Simulation 2, step 0
Figure A.16: Simulation 2, step 100
Figure A.17: Simulation 2, step 200
Figure A.18: Simulation 2, step 300
Figure A.19: Simulation 2, step 400
Figure A.20: Simulation 2, step 500
Figure A.21: Simulation 2, step 600
Figure A.22: Simulation 2, step 700
Figure A.23: Simulation 2, step 800
Figure A.24: Simulation 2, step 900
Figure A.25: Simulation 2, step 1000
Appendix B

Source code

B.1 Makefile

```makefile
# F# compiler:
FSC = fsc

# Targets:
PATH = mybin
EXE = Simulator
DLL = MAS2011
AIS = AISAgent

# Dependencies:
GTK = ../gtk-sharp-2.0/gtk-sharp.dll
ATK = ../gtk-sharp-2.0/atk-sharp.dll
GDK = ../gtk-sharp-2.0/gdk-sharp.dll
PANGO = ../gtk-sharp-2.0/pango-sharp.dll
CAIRO = Mono.Cairo.dll
MEF = $(PATH)/System.ComponentModel.Composition.dll

# Files:
```
SIMFILES = 
   GtkHelpers.fs DummyAgent.fs Agents.fs SimSettingsRoles.fs SimSettingsGeneral.fs 
   SimSettingsMilestones.fs Simulation.fs SimSettingsWindow.fs SimulationView.fs SimulationWindow.fs Initial.fs

LIBFILES = 
   Generics.fs ShapePrimitives.fs SimTypes.fs SimulationSteps.fs Agent.fs Edge.fs Node.fs Graph.fs IAgent.fs 
   TS.fs SimulationSteps.fs SimTypesDrawing.fs

all: sim agents

sim: library
   $(FSC) -r:$(MEF) -r:$(PATH)/$(DLL).dll -r:$(GTK) -r:$(ATK) 
   -r:$(GDK) -r:$(GLIB) -r:$(PANGO) -r:$(CAIRO) 
   $(SIMFILES) -o:$(PATH)/$(EXE).exe

library:
   $(FSC) -r:$(PATH)/Triangulator.dll -r:$(GDK) $(LIBFILES) 
   --target:library -o:$(PATH)/$(DLL).dll

agents: ais

ais:
   $(FSC) AgentHelpers.fs $(AIS).fs -r:$(PATH)/$(DLL).dll 
   -r:$(MEF) --target:library -o:$(PATH)/$(AIS).dll
module Agent

(*
Module containing functions for manipulation of the Agent type.
*)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics

// Determine whether a given agent has been inspected by
// an agent on the given team.
// int * 'a -> Agent -> bool
let InspectedBy(team, _) agent =
  let otherteam, _ = agent.ID
  team = otherteam
  || List.exists(fun (t,_) -> team = t) agent.AgentsInspected

// Add, that an agent with id has inspected the given agent
// Agent -> int * int -> Agent
let AddAgentInspected id agent =
  if InspectedBy id agent
  then agent
  else
    { agent with AgentsInspected = id :: agent.AgentsInspected }

// Reduce the energy of an agent by n
// Agent -> Agent
let ReduceEnergy n agent =
  let newen = Max 0 (agent.Energy - n)
  { agent with Energy = newen }

// Completely repair an agent
// Agent -> Agent
let RepairA (a : Agent) =
  { a with Health = a.MaxHealth }

// Determine whether a given agent is disabled or not
// Agent -> bool
let IsDisabled (a : Agent) =
  a.Health < 1

// Determine whether two given agents are opponents
let IsOpponent a1 a2 =
  let team1, _ = a1.ID
  let team2, _ = a2.ID
  team1 <> team2

// Recharge an agent using the values in the given
// SimulationSettings
// SimulationSettings -> Agent -> Agent
let RechargeA (settings : SimulationSettings) (a : Agent) =
  let recoveragent recover =

let newen =
  (float a.MaxEnergy) * recover / 100.0
| > int
| > Max 1
| > (+) a.Energy
| > Min a.MaxEnergy
{ a with Energy = newen }
if (IsDisabled a)
then
  recoveragent (float settings.RecoverDisabled)
else
  recoveragent (float settings.RecoverNormal)
// Determine whether an agent can perform a given action.
// This function looks only at whether the action is in the
// given agents list of possible actions, and not its energy
// level.
// Agent  Action  ->  bool
let rec CanPerform agent action =
let isgoto ac =
  match ac with
  | Goto(%)  ->  true
  | _      ->  false
let isattack ac =
  match ac with
  | Attack(%,%)  ->  true
  | _      ->  false
let isrepair ac =
  match ac with
  | Repair(%)  ->  true
  | _      ->  false
let isbuy ac =
  match ac with
  | Buy(%)  ->  true
  | _      ->  false
let has f = List.exists f agent.Actions
match action with
  | Goto(%)  ->  has isgoto
  | Attack(%,%)  ->  has isattack
  | Repair(%)  ->  has isrepair
  | Buy(up)  ->
    match up with
    | SabotageDevice  ->
      has isbuy && CanPerform agent (Attack(0,0))
    | _      ->  has isbuy
    | _      ->  !mem action agent.Actions
// Anonymize a given agent, depending on a given id. If the
// given agent has been inspected by the given id, then
// nothing is changed (Except the list of inspected agents).
// If the agent has not been inspected, all of its values
// are hidden.
// int  *  'a  ->  Agent  ->  Agent
let Anonymize (team, a) agent =
  if InspectedBy (team, a) agent
then { agent with AgentsInspected = [team,a] } 
else

{ agent with
  Health = Min 1 agent . Health
  MaxHealth = 1
  Energy = 1
  MaxEnergy = 1
  Strength = 1
  Visibility = 1
  Actions = []
  AgentsInspected = [] }

// Upgrade an agent, using a given upgrade.
// Upgrade → Agent → Agent
let Upgrade up a =
match up with
| Battery ->
  { a with
    Energy = a . Energy + 1
    MaxEnergy = a . MaxEnergy + 1 }
| Sensor ->
  { a with Visibility = a . Visibility + 1 }
| Shield ->
  { a with
    Health = a . Health + 1
    MaxHealth = a . MaxHealth + 1 }
| SabotageDevice ->
  { a with Strength = a . Strength + 1 }

// Make an Agent using a list of stats and actions, and with a
given ID. Will only make an Agent when the list of stats
and actions have the correct lengths.
// int * int -> int lis -> bool list -> Agent Option
let Create ( id , statlist , aclist ) =
match statlist , List . length aclist with
| str : : en : : he : : vis : : [] , 7 ->
let actions =
  [ Attack ( 0 , 0 ) ;
    Parry ;
    Probe ;
    Survey ;
    Inspect ;
    Buy ( Battery ) ;
    Repair ( 0 ) ]
|> List . zip aclist
|> List . filter ( fun ( a , _ ) -> a )
|> List . map ( fun ( _, a ) -> a )
|> List . append [ Skip ; Recharge ; Goto ( 0 ) ]

let agent =
{ ID = id
  Node = 0
  Health = he
  MaxHealth = he
  Energy = en
  MaxEnergy = en}
Strength = str
Visibility = vis
Actions = actions
AgentsInspected = []

Some agent

| . . . | None
module MAS2011.Agents.AgentHelpers

(* Module containing some helper functions the AI’s can use. *)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics
open MAS2011.Shared.IAgent
open MAS2011.Shared.SimTypesDrawing
open MAS2011.Shared.SimulationSteps
open MAS2011.Shared.ShapePrimitives
open System.ComponentModel.Composition
open System

// Retrieve the amount of uninspected enemy agents from a given node. It will count using the node and its neighbours.
// Graph → Node → Agent → float
let NumUninspected ((nodes,_) as graph) node agent =
  if Agent.CanPerform agent Inspect
  & not (Agent.IsDisabled agent)
  then
    Graph.NeighbourNodes graph node.NodeID
    |> List.append [node.NodeID]
    |> List.map (fun n -> nodes.[n].Agents)
    |> List.concat
    |> List.filter (fun a -> not (Agent.InspectedBy agent.ID a))
    |> List.length
  else 0.0

// Get the amount of unsurveyed edges connected to a given node.
// Graph → Node → Agent → float
let NumUnsurveyed ((_,edges) : Graph) node agent =
  if Agent.CanPerform agent Survey
  & not (Agent.IsDisabled agent)
  then
    edges.[node.NodeID]
    |> Array.toList
    |> List.choose (fun a -> a)
    |> List.filter (fun e -> not (Edge.SurveyedBy agent.ID e))
    |> List.length
  else 0.0

// Determine whether a given node has been probed by a given agent or not. Will only ever return true if the agent can actually perform the probe–action and isn’t disabled.
// Agent → Node → bool
let IsUnprobed agent node =
  not (Node.ProbedBy agent.ID node)
&& Agent.CanPerform agent Probe
&& not (Agent.IsDisabled agent)

// Helper function to determine whether one agent can attack
// another. Will only return true if neither of the two agents
// are disabled, the attacking agent can actually attack and
// the two agents are opponents.
let internal CanAttack agent a =
  not (Agent.IsDisabled a)
  && not (Agent.IsDisabled agent)
  && Agent.CanPerform agent (Attack(0,0))
  && Agent.IsOpponent agent a

// Helper function to determine whether one agent can be
// attacked by another agent, using the above function.
let CanBeAttacked a b = CanAttack b a

// Determine whether a given node has a target that can be
// attacked by a given agent.
let HasTarget agent node =
  List.exists (CanAttack agent) node.Aagents

// Determine whether a given node has an enemy agent that can
// attack a given agent, while the given agent can perform the
// parry action.
let HasAttackingCanParry agent node =
  Agent.CanPerform agent Parry
  && List.exists (CanBeAttacked agent) node.Aagents

// Determine whether a given node has an enemy agent that can
// attack a given agent, while the given agent can't perform the
// parry action.
let HasAttackingCan'tParry agent node =
  not (Agent.CanPerform agent Parry)
  && List.exists (CanBeAttacked agent) node.Aagents

// Determine whether a given node has an enemy agent, compared
// to a given agent.
let HasEnemy agent node =
  List.exists
    (fun a ->
      Agent.IsOpponent agent a
      && not (Agent.IsDisabled a))
    node.Aagents

// Determine whether a given node has a friendly agent, compared
// to a given agent.
let HasFriendly agent node =
B.3 AgentHelpers.fs

List.exists
  (fun a ->
   not (Agent.IsOpponent agent a)
   && a.ID <> agent.ID
   && not (Agent.IsDisabled a))
node.Agents

// Determine whether a given node has an agent that can repair
// the given agent. Will only return true if the given agent is
// disabled, and the repairing agent is friendly but not the
// same agent, and can perform the repair action.
// Agent -> Node -> bool
let HasRepairingAgent agent node =
  let CanRepair a =
    Agent.CanPerform a (Repair(0))
    && not (Agent.IsOpponent a agent)
    && a.ID <> agent.ID
    && Agent.IsDisabled agent
    && List.exists CanRepair node.Agents

// Calculate all possible moves in a graph, using a given agent
// and a single action. For instance, if given the Goto action,
// the algorithm will determine all the possible Goto-actions
// for the given agent.
// Skip and Buy actions are ignored. The Skip action can always
// be replaced by the Recharge action, and the (so far) created
// agents all ignore the Buy action, as the amount of money
// in each step counts towards the score.
// SimulationSettings -> Graph -> Agent -> Action -> Action list
let AllPossibleMoves settings ((nodes, edges) as g : Graph)
  agent action =
    let en = agent.Energy
    match action with
    | Recharge -> [Recharge]
    | Goto(_) ->
      edges.[agent.Node]
      |> Array.toList
      |> List.choose (fun a -> a)
      |> List.choose
      (fun edge ->
       if edge.Weight > agent.Energy
       then None
       else Some(Goto(nodes.[edge.To].NodeID)))
    | Attack(_,_) when
      en >= settings.AttackCost
    && not (Agent.IsDisabled agent) ->
      nodes.[agent.Node].Agents
      |> List.choose
      (fun a ->
       if Agent.IsOpponent a agent && not (Agent.IsDisabled a)
       then
         let tid, aid = a.ID
         Some(Attack(tid, aid))
       else None)
    | Parry when
en >= settings.ParryCost
&\& not (Agent.IsDisabled agent) ->
let HasAttackingEnemy node =
  node.Agents
  |> List.exists
  (fun a ->
    Agent.IsOpponent a agent
    && Agent.CanPerform a (Attack(0,0))
  if HasAttackingEnemy nodes.[agent.Node]
        then [Parry]
    else []
| Probe when
en >= settings.ProbeCost
&\& not (Agent.IsDisabled agent) ->
if Node.ProbedBy agent.ID nodes.[agent.Node]
  then []
else [Probe]
| Survey when
en >= settings.SurveyCost
&\& not (Agent.IsDisabled agent) ->
let AnyUnsurveyed =
  edges.[agent.Node]
  |> Array.toList
  |> List.choose (fun a -> a)
  |> List.exists
  (fun edge -> not (Edge.SurveyedBy agent.ID edge))
  if AnyUnsurveyed
    then [Survey]
  else []
| Inspect when
en >= settings.InspectCost
&\& not (Agent.IsDisabled agent) ->
let agents =
  Graph.NeighbourNodes g agent.Node
  |> List.append [agent.Node]
  |> List.map (fun n -> nodes.[n].Agents)
  |> List.concat
  if List.exists
    (fun a -> not (Agent.InspectedBy agent.ID a)) agents
    then [Inspect]
  else []
| Repair(\_) when en >= settings.RepairCost ->
let agents =
  nodes.[agent.Node].Agents
  |> List.filter
  (fun a ->
    not (Agent.IsOpponent a agent)
    && a.ID <> agent.ID
    && a.Health < a.MaxHealth)
  let getaid a =
    let \.aid = a.ID
da
  agents
  |> List.map (fun a -> Repair(getaid a))
| _ -> [] // Ignore Skip and Buy actions

// Create an unknown node with a given ID. Unknown nodes are
// identified by their (-1,-1) coordinate.
// int -> Node
let UnknownNode i =

{ NodeID = i
Weight = 0
Agents = []
AgentsProbed = []
Coordinate = (-1,-1) // To identify an unknown node
DominatingTeam = None }

// Update a graph with the information from a given node. Will
// return a Node option, depending on whether the given node
// holds any relevant new information or not. If so, this
// information should be shared with all friendly agents, but to
// keep the information flow on a minimum, only certain new
// information will be shared.
// Has side effects.
// Graph -> Node -> Node option
let UpdateGraphWithNode ((nodes,_) : Graph) node =

let oldnode = nodes.[node.NodeID]
let ignoreupdate =
if not ignoreupdate then
oldnode.AgentsProbed <> [] \&\& node.AgentsProbed = []
if oldnode.Coordinate = (-1,-1) || (oldnode.AgentsProbed = [] \&\& node.AgentsProbed <> [])
then Some node
else None

// Update a graph with the information from a given edge. Will
// return an EdgeInfo option, depending on whether the given
// edge holds any relevant new information or not, for the same
// reasons as above.
// Has side effects.
// Graph -> EdgeInfo option
let UpdateGraphWithEdge ((_,edges) as g : Graph) edge =

let oldedge = edges.[edge.From].[edge.To]
let ignoreupdate =
match oldedge with
| Some e ->
e.AgentsSurveyed <> [] \&\& edge.AgentsSurveyed = []
| _ -> false
if not ignoreupdate then

let otheredge =

{ edge with
  From = edge.To
  To = edge.From}
Graph.ReplaceEdge g edge
Graph.ReplaceEdge g otheredge
match oldedge with
| None -> Some edge
let UpdateGraphWithAgent ((nodes,_) as g) agent =
  nodes |> Array.iteri (fun i n -> nodes.[i] <- Node.RemoveAgent n agent)
  Graph.AddAgent g agent.Node agent

let StandardSettings =
  { FailChance = 1,
    Length = 10000,
    MaxAgentResponse = 100,
    RecoverNormal = 1,
    RecoverDisabled = 1,
    AttackCost = 1,
    ParryCost = 1,
    ProbeCost = 1,
    SurveyCost = 1,
    InspectCost = 1,
    BuyCost = 1,
    RepairCost = 1,
    FailedGotoCost = 1,
    UpgradeBatteryPrice = 1,
    UpgradeSensorPrice = 1,
    UpgradeShieldPrice = 1,
    UpgradeSabotageDevicePrice = 1 }

let ActionToString action =
  match action with
  | Skip -> "Skip"
  | Recharge -> "Recharge"
  | Goto(n) -> sprintf "Goto%d" n
  | Attack(t,a) -> sprintf "Attack%d,%d" t a
  | Parry -> "Parry"
  | Probe -> "Probe"
  | Survey -> "Survey"
  | Inspect -> "Inspect"
  | Repair(n) -> sprintf "Repair%d" n
  | Buy(_) -> "Upgrade something"

let Some e when e.AgentsSurveyed = []
  && edge.AgentsSurveyed <> [] ->
  Some edge
  _ -> None

// Update a graph with an agent. Will remove the agent from all
// other nodes, to eliminate obsolete information.
// Has sideeffects.
// Graph -> Agent -> unit
let UpdateGraphWithAgent ((nodes,_) as g) agent =
  nodes |> Array.iteri (fun i n -> nodes.[i] <- Node.RemoveAgent n agent)
  Graph.AddAgent g agent.Node agent

// Default simulation settings for agents.
// SimulationSettings

let StandardSettings =
  { FailChance = 1,
    Length = 10000,
    MaxAgentResponse = 100,
    RecoverNormal = 1,
    RecoverDisabled = 1,
    AttackCost = 1,
    ParryCost = 1,
    ProbeCost = 1,
    SurveyCost = 1,
    InspectCost = 1,
    BuyCost = 1,
    RepairCost = 1,
    FailedGotoCost = 1,
    UpgradeBatteryPrice = 1,
    UpgradeSensorPrice = 1,
    UpgradeShieldPrice = 1,
    UpgradeSabotageDevicePrice = 1 }

// Convert an action to string format.
// Action -> string

let ActionToString action =
  match action with
  | Skip -> "Skip"
  | Recharge -> "Recharge"
  | Goto(n) -> sprintf "Goto%d" n
  | Attack(t,a) -> sprintf "Attack%d,%d" t a
  | Parry -> "Parry"
  | Probe -> "Probe"
  | Survey -> "Survey"
  | Inspect -> "Inspect"
  | Repair(n) -> sprintf "Repair%d" n
  | Buy(_) -> "Upgrade something"

// Determine whether an agent lacks energy. Will return true
// if the agent would be unable to perform the probe, inspect,
// attack, survey, repair or parry actions, or if the agent
// doesn’t have enough energy for a failed goto–action.
// SimulationSettings → Agent → bool
let LacksEnergy settings agent =
  let en = agent.Energy
  en < settings.AttackCost
  || en < settings.ProbeCost
  || en < settings.InspectCost
  || en < settings.SurveyCost
  || en < settings.RepairCost
  || en < settings.ParryCost
  || en < settings.FailedGotoCost
B.4  Agents.fs

```fsharp
module MAS2011.Agents

(*
Module for loading and creating agent objects.
Loads agents via MEF, the agents must implement the IAgent
interface and define a name by using the IAgentMetadata
*)

open MAS2011.Shared.IAgent
open MAS2011.DummyAgent
open System.ComponentModel.Composition
open System.ComponentModel.Composition.Hosting

// Class that MEF can use to load agents
type internal AgentHolder () =
[<ImportMany>]
let agents : ExportFactory<IAgent, IAgentMetadata>[] = [| |]

member __.GetAgents () =
    let find (ec : ExportFactory<IAgent, IAgentMetadata>) =
        ec.Metadata.Name = s
    match Array.tryFind find agents with
    | Some ec -> ec.CreateExport().Value
    | None -> new DummyAgent() :> IAgent

member __.GetAllNames () =
    let agentnames =
        agents |
        Array.map (fun ec -> ec.Metadata.Name) |
        Array.toList
    "Dummy agent" :: agentnames

// Set up MEF, load agents etc.
let internal catalog = new AggregateCatalog ()
let internal directoryCatalog =
    new DirectoryCatalog(@"./", "*.dll")
let internal container = new CompositionContainer(catalog)
catalog.Catalogs.Add(directoryCatalog)
let internal agentholder = new AgentHolder ()
container.ComposeParts(agentholder)

// Get the correct agent for a given string
// Will return a DummyAgent when the string is unknown
let GetAgent s =
    agentholder.GetAgent s

// Gets the names of all agents loaded
let GetAllNames () =
    agentholder.GetAllNames ()
```
module MAS2011.Agents.AISAgent

(*
AIS agent: Aggressive Information-Seeker

An agent with prioritized actions. At the beginning of a
simulation, this agent will seek out information and possible
attack enemy agents. When all information has been gathered, it
attempts to stay near friendly agents, but not on the same node
as friendly or enemy agents. When an agent has been disabled, it
should seek out a repairing agent.

The priority for actions are as follows:
1. Recharge if energy is too low to perform one of the other
   actions
2. Probe
3. Survey
4. Inspect
5. Attack
6. Parry
7. Repair
8. Find the optimal node position — either move towards it, or
   recharge
*)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics
open MAS2011.Shared.IAgent
open MAS2011.Shared.SimTypesDrawing
open MAS2011.Shared.SimulationSteps
open MAS2011.Shared.ShapePrimitives
open MAS2011.Agents.AgentHelpers
open System.ComponentModel.Composition
open System

// Weights used for finding the optimal node
let UninspectedWeight = 10000.0
let UnsurveyedWeight = 12000.0
let UnprobedWeight = 50000.0
let AttackTargetWeight = 8000.0
let AttackingParryWeight = -1000.0
let AttackingNoParryWeight = -10000.0
let EnemyWeight = -5000.0
let FriendlyWeight = 2000.0
let RepaireeWeight = 10000.0
let NodeWeight = 500.0

// Precalculated list of delay factors.
// The delay factor will enable the agents to prioritize nearby
// nodes.
// Float array
let DelayFactor =
  Array.init 1000 (fun n -> pow 0.99 n)

// Retrieve the estimated node value for a given node in a given graph.
// SimulationSettings -> Graph -> Agent -> Node -> float
let NodeValueSettings ((nodes,_) as graph : Graph) agent node =
  if Agent.IsDisabled agent then
    if HasRepairingAgent agent node then RepairerWeight
    else 0.0
  else
    let tempval = ref 0.0
    let add n = tempval := !tempval + n
    add ((NumUninspected graph node agent) * UninspectedWeight)
    add ((NumUnsurveyed graph node agent) * UnsurveyedWeight)
    if IsUnprobed agent node then add UnprobedWeight
    if HasTarget agent node then add AttackTargetWeight
    if HasAttackingCanParry agent node then add AttackingParryWeight
    if HasAttackingCantParry agent node then add AttackingNoParryWeight
    if HasEnemy agent node then add EnemyWeight
    if HasFriendly agent node then add (-5.0 * FriendlyWeight)
    add (node.Weight |> float |> (*) NodeWeight)
    let numfriendly l =
      l > List.map (fun n -> nodes.[n])
      |> List.filter (HasFriendly agent)
      |> List.length
      |> float
    let neighbours = Graph.NeighbourNodes graph agent.Node
    neighbours
      |> numfriendly
      |> (++) FriendlyWeight
    |> add
    neighbours
      |> List.map (Graph.NeighbourNodes graph)
      |> List.concat
      |> List.filter
      (fun n -> n <> agent.Node && not (lmem n neighbours))
      |> numfriendly
      |> (++) (5.0 * FriendlyWeight)
    |> add
    !tempval

    // Agent class
[<ExportMetadata("Name", "Aggressive Information-Seeker")>]
[<Export(typeof<IAgent>)>]
type AISAgent () =
  let mutable graph : Graph = ([[]],[[]])
let mutable settings = StandardSettings
let mutable lastnodevalues = [[ ]] // Algorithm from:
// Will find all shortest paths in the graph. Runtime of
// O(V^3)
let agent -> int array array * int array array
member __.FloydWarshall agent =
    let nodes, edges = graph
    let agent2 =
        Agent.RechargeA settings { agent with Energy = 0 }
    let enperstep = agent2 . Energy
    let initialweight i j =
        if i = j then 0
        else
            match edges . [i] . [j] with
            | Some edge when edge . Weight <= agent . MaxEnergy ->
                edge . Weight / enperstep + 1
            | _ -> 1000000
    let numnodes = Array . length nodes
    let path =
        Array.init numnodes
        ( fun i ->
            Array.init numnodes
            ( fun j -> initialweight i j ) )
    let next =
        Array.init numnodes
        ( fun _ -> Array.init numnodes ( fun _ -> -1 ) )
    let maxn = numnodes - 1
    for k in 0 .. maxn do
        for i in 0 .. maxn do
            for j in 0 .. maxn do
                let newpath = path . [i] . [k] + path . [k] . [j]
                if newpath < path . [i] . [j] then
                    path . [i] . [j] <- newpath
                    next . [i] . [j] <- k
    // Draw the graph as the agent sees/remembers it. Will also
    // draw the last estimated values for all nodes (which is only
    // updated when the agent has to move, in priority 8 as above)
    // Action -> Agent -> int -> unit
    member __.DrawGraph action agent stepnum =
        let team_id = agent . ID
        let colors =
            List . init 10 ( fun i -> if i = team_id then Green else Red )
        let ( nodes , edges ) = graph
        let ( cx , cy ) =
            nodes . [ agent . Node ] . Coordinate
            |> ScaleInts scaling
        let ac = action |> ActionToString
        let ( nodes , _ ) = graph
        let makenodevaluertext nodeid value =
let x, y = nodes.[nodeid].Coordinate |> ScaleInts scaling
let text = value |> sprintf "%d: %.1f" nodeid
Text((x-10.0,y-20.0),text,White)
let nodevalues =
|> Array.toList
|> List.map makenodevaluetext
let shapes =
GraphToShapes scaling colors graph
|> List.append [Circle((cx, cy), 15.0, Filled(Yellow))] |
|> List.append nodevalues |
|> List.map (DisplaceShape (scaling/2.0, scaling/2.0+100.0)) |
|> List.append [Text((30.0, 30.0), ac, White)]
let actions =
[Attack(0,0),"A"; Probe,"P"; Inspect,"I"; Survey,"S"; Repair(0),"R"]
|> List.fold
(fun olds (a, s) ->
if Agent.CanPerform agent a then olds + s else olds) ""
let name = sprintf "AIS agent %A %s" agent.ID actions
AddSimStep name (stepnum-1) ((1500,1500),shapes)

// Retrieve the prioritized action from a list of possible
// actions.
// Uses the priority mentioned in the top of this file.
// Agent -> Action list -> Action
member this.PrioritizedAction agent actions =
let IsAttack a =
match a with
| Attack(_,_) -> true
| _ -> false
let IsRepair a =
match a with
| Repair(_,) -> true
| _ -> false
if LacksEnergy settings agent
then Recharge
else if lmem Probe actions
then Probe
else if lmem Survey actions
then Survey
else if lmem Inspect actions
then Inspect
else if List.exists IsAttack actions
then List.find IsAttack actions
else if lmem Parry actions
then Parry
else if List.exists IsRepair actions
then List.find IsRepair actions
else this.NextNode agent actions

// Retrieve the wanted next node. It does so by calculating
// the estimated value for all nodes, applying a delay factor,
// and selecting the maximum value.
// Agent -> Action list -> Action
member this.NextNode agent actions =
    let (nodes, edges) = graph
    let nodevalues = nodes |> Array.map (fun _ -> -1000000.0)
    let (path, next) = this.FloydWarshall agent
    let UpdateNode node =
        let tt = path.[agent.Node].[node.NodeID]
        if tt < 1000
            then
                nodevalues.[node.NodeID] <-
                    (NodeValue settings graph agent node)*DelayFactor.[tt]
        Array.iter UpdateNode nodes
        lastnodevalues <- nodevalues
    let targetnode =
        nodevalues
        |> Array.toList
        |> List.map (fun i v -> (i, v))
        |> List.sortBy (fun (_, v) -> -v)
        |> List.head
        |> fun (i, _) -> i
    let rec getnext target =
        if next.[agent.Node].[target] = -1
        then target
        else getnext next.[agent.Node].[target]
    let nextnode = getnext targetnode
    if targetnode = agent.Node
        || not (lmem (Goto (nextnode)) actions)
        then Recharge
        else Goto (nextnode)
    (*
    interface IAgent with
    // Algorithm for updating the information the agent has and
    // calculating the action the agent should perform.
    // Perception -> Action
    member this.MakeStep perc =
        let nodemsgs =
            perc.Messages
            |> List.choose
                (fun (_, msg) ->
                    match msg with
                        | InfoNode n -> Some n
                        | _ -> None)
        let edgemsgs =
            perc.Messages
            |> List.choose
                (fun (_, msg) ->
                    match msg with
                        | InfoEdge e -> Some e
                        | _ -> None)
        List.map (UpdateGraphWithNode graph) nodemsgs |> ignore
        List.map (UpdateGraphWithEdge graph) edgemsgs |> ignore
        Graph.ResetColoring graph // old coloring irrelevent
    let newnodes =
        List.choose (UpdateGraphWithNode graph) perc.Nodes
        |> List.map (fun n -> Broadcast (InfoNode(n)))
```ocaml
let newedges = List.choose (UpdateGraphWithEdge graph) perc.Edges |> List.map (fun e -> Broadcast(InfoEdge(e))) perc.OtherAgents |> List.iter (UpdateGraphWithAgent graph)
let actions = perc.Agent.Actions |> List.map (AllPossibleMoves settings graph perc.Agent) |> List.concat
let action = this.PrioritizedAction perc.Agent actions
this.DrawGraph action perc.Agent perc.StepNum (action, newnodes @ newedges)

// SimulationSettings -> unit
member __.AddSettings s =
settings <- s

// This function is expected to be called before the first simulation step is executed. It will set up the agent's internal structures.
// int * int * int -> unit
member __.SetNumNodesEdgesSteps (n,e,s) =
let nodes = Array.init n (fun i -> UnknownNode i)
let edges = Array.init n (fun _ -> Array.init n (fun _ -> None))
graph <- (nodes, edges)
```
module MAS2011.DummyAgent

(*
Dummy agent used by the simulation. This is the default agent.
When asked for a move, the agent will randomly go to a
neighbour node with 50% probability, and recharge with 50%
probability.
*)

open MAS2011.Shared.IModule
open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics

type DummyAgent() =
  interface IAgent with
    member __.MakeStep perc =
      let mypos = perc.Agent.Node
      let neighbours1 =
        perc.Edges
        |> List.filter(fun x -> x.From = mypos)
        |> List.map(fun x -> x.To)
      let neighbours2 =
        perc.Edges
        |> List.filter(fun x -> x.To = mypos)
        |> List.map(fun x -> x.From)
      let neighbours =
        neighbours1 @ neighbours2
        |> RandomizeList
      match neighbours with
      | hd::_ ->
        let action =
          [Probe;Survey;Inspect;Recharge;Goto(hd)]
          |> List.filter(Agent.CanPerform perc.Agent)
          |> RandomizeList
          |> List.head
          (action ,[])
      | _ -> (Recharge ,[])

    member __.AddSettings  = ()
    member __.SetNumNodesEdgesSteps (_,_,:) = ()
module Edge

(*
Module containing functions for manipulation of the EdgeInfo
 type.*)

(*)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics

// Add that an agent with given id has surveyed the edge
// EdgeInfo -> int * int -> EdgeInfo
let AddAgentSurveyed edge id =
  if lmem id edge.AgentsSurveyed
  then edge
  else { edge with AgentsSurveyed = id :: edge.AgentsSurveyed }

// Remove the dominating team.
// EdgeInfo -> EdgeInfo
let RemoveDominatingTeam edge : Edge =
  match edge with
  | Some e -> Some { e with DominatingTeam = None }
  | None -> None

// Color the edge. The edge will be colored if the two nodes
// it is connecting, are both colored with the same color.
// The resulting color of the edge will be the same as the two
// nodes it connects.
// Graph -> Edge -> Edge
let Color ((nodes,_) : Graph) (edge : Edge) =
  match edge with
  | Some e ->
    let domfrom = nodes.[e.From].DominatingTeam
    let domto = nodes.[e.To].DominatingTeam
    if domfrom = domto
    then Some { e with DominatingTeam = domfrom }
    else edge
  | _ -> edge

// Determine whether a given edge has been surveyed by an agent
// on a given team.
// int * 'a -> EdgeInfo -> bool
let SurveyedBy (team,_) edge =
  List.exists (fun (t,_) -> team = t) edge.AgentsSurveyed

// Anonymize a given edge, depending on whether an agent on a
given team has surveyed the edge. If not, its weight will
be set to 1.
// int * int -> EdgeInfo -> EdgeInfo
let Anonymize (team,a) edge =
  if SurveyedBy (team,a) edge
  then { edge with AgentsSurveyed = [team,a] }
```plaintext
else
  { edge with
    Weight = 1
    AgentsSurveyed = [] }

// Make an EdgeInfo with a certain weight, between two given
// int -> int -> int -> EdgeInfo
let Create (w, f, t) =
  { Weight = w
    From = f
    To = t
    AgentsSurveyed = []
    DominatingTeam = None }
```
module MAS2011.Shared.Generics

(*
General functions used in the project.
*)

open System

// Randomizes the list 1
// 'a list -> 'a list
let RandomizeList l =
    let r = new Random(DateTime.Now.Ticks |> int)
    l |> List.map (fun a -> (a, r.Next()))
    |> List.sortBy (fun (_,a) -> a)
    |> List.map (fun (a,_) -> a)

// Converts an F# list to a generic list
// 'a list -> List<'a>
let ToGenericList (l : _ list) =
    Collections.Generic.List(l)

// Converts a generic list to an F# list
// Seq<'a> -> 'a list
let OfGenericList l =
    List.ofSeq(l)

// Scales two integers by a given float value
// float -> int * int -> float * float
let ScaleInts s (i1,i2) = (float i1) * s, (float i2) * s

// Creates n lines arranged in a circle, all going from center
// to the perimeter of a circle (with radius 1)
// int -> (float * float) list
let CreateCircularLines n =
    [ for a in 1..n do
        let nf = float n
        let af = float a
        let angle = af*2.0*Math.PI/nf
        yield (Math.Cos(angle),Math.Sin(angle)) ]

// Calculate the minimum//maximum of two given values
// Requires comparison
// 'a -> 'a -> 'a
let Min a b = if a < b then a else b
let Max a b = if a > b then a else b

// Determine whether a given element is a member of a given list
// 'a -> 'a list -> bool
let lmem a = List.exists (fun b -> a = b)

// Counts all occurences of all values in a list
// 'a list -> Map<'a,int>
let CountAll l =
let rec CountAllHelper l map =
  match l with
  | hd::tl ->
    match Map.tryFind hd map with
    | Some num -> CountAllHelper tl (map.Add(hd,num+1))
    | _ -> CountAllHelper tl (map.Add(hd,1))
  | _ -> map
CountAllHelper l Map.empty

// Splits a list in two lists, by using a given filter
// The same could be achieved by using List.filter twice, once
// for the filter, and once for the negated filter
// (a -> bool) -> 'a list -> 'a list * 'a list
let ListSplit f l =
let rec ListSplitHelper f l (ret1,ret2) =
  match l with
  | hd::tl ->
    if f hd
    then ListSplitHelper f tl (hd::ret1,ret2)
    else ListSplitHelper f tl (ret1,hd::ret2)
  | _ -> (ret1,ret2)
ListSplitHelper f l ([],[])

// Replace the nth occurrence in a list, with the same element
// after f has been applied to it
// int -> ('a -> 'a) -> 'a list -> 'a list
let ListReplace n f l =
  if 0 <= n && n < List.length l
  then
    let ar = l |> List.toArray
    ar.[n] <- f ar.[n]
    ar |> Array.toList
  else l

// Converts a list of characters to a string
// char list -> string
let StringFromChars cl =
  new System.String(cl |> Array.ofList) |> string

// Split a string by a given character, into a list of strings
// char -> string -> string list
let StringToListDel del (s : string) =
let rec StringToListHelper curlist curstr remchars =
  match remchars,curstr with
  | [],[] -> curlist
  | [],_ -> (StringFromChars (List.rev curstr))::curlist
  | hd::tl,[], when hd = del -> StringToListHelper curlist [] tl
  | hd::tl,_, when hd = del ->
    let newList = (StringFromChars (List.rev curstr))::curlist
    StringToListHelper newList [] tl
  | hd::tl,_,_ -> StringToListHelper curlist (hd::curstr) tl
  | chars = List.ofSeq s
  List.rev (StringToListHelper [] [] chars)
let StringGlue glue sl = match sl with
| hd :: tl ->
  let rest = List.fold (fun s t -> s + glue + t) "" tl
  hd + rest
| _    -> ""

let rec ListTryFindIndex f l = match l with
| [] -> None
| hd :: tl -> if f hd then Some 0 else match ListTryFindIndex f tl with
  | Some n -> Some (n + 1)
  | _   -> None

let rec ListDivide3 l = match l with
| a :: b :: c :: tl -> [a; b; c] :: ListDivide3 tl
| _    -> []

let rec MapAppend map n l = match Map.tryFind n map with
| Some el -> map.Add(n, List.append el l)
| _    -> map.Add(n, l)

let Dominator min (l : int list) = let counted = CountAll l
| Map.toList |
| List.sortBy (fun (_, a) -> -a)
| List.filter (fun (_, a) -> a >= min)
match counted with
| (team1, num1) :: (team2, num2) :: _ when num1 > num2 -> Some team1
| _       :: []    -> Some team1
| _        -> None
// Calculate a, lifted to the power of b (a^b)
// float -> int -> float
let pow a b =
  let rec powhelp c d =
    match d with
    | 0 -> 1.0
    | 1 -> c
    | _ -> powhelp (c*a) (d-1)
powhelp a b
module Graph

(*
Module with functions for manipulation of the Graph type.
*)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics
open DelaunayTriangulator
open System

// Add a given agent to the node with a given id.
// Has sideeffects.
// Graph -> int -> Agent -> unit
let AddAgent ((nodes,_) : Graph) n a =
  if Array.length nodes > n
  then nodes.[n] <- Node.AddAgent nodes.[n] a

// Get the neighbour nodes for a given id.
// Graph -> int -> int list
let NeighbourNodes ((_,edges) : Graph) n =
  edges.[n]|
  Array.choose (fun a -> a) |
  Array.toList |
  List.map (fun a -> a.To)

// Get all agents on a given graph
// Graph -> Agent list
let GetAllAgents ((nodes,_) : Graph) =
  nodes |
  Array.map (fun n -> n.Agents) |
  Array.toList |
  List.concat

// Replace an agent on a graph.
// Has sideeffects.
// Graph -> Agent -> unit
let ReplaceAgent ((nodes,_) : Graph) (agent : Agent) =
  let n = agent.Node
  if Node.HasAgent nodes.[n] agent
  then
    nodes.[n] <- Node.ReplaceAgent nodes.[n] agent

// Replace an edge on a graph.
// Has sideeffects.
// Graph -> EdgInfo -> unit
let ReplaceEdge ((_,edges) : Graph) edge =
  edges.[edge.From].[edge.To] <- Some edge

// Move an agent in a graph, the node with a given id.
// If the agent doesn't have enough energy, its energy should
// be reduced by an amount specified in the given settings.
// Else, its energy should be reduced by the weight of the edge connecting the two nodes and it should be moved.
// Has sideeffects.
// SimulationSettings → Graph → Agent → int → bool

let MoveAgent settings ((nodes, edges) as g : Graph) =
  (agent : Agent) n =
  let from = agent.Node
  match edges.[from].[n] with
  | Some e when e.Weight <= agent.Energy ->
    nodes.[from] <- Node.RemoveAgent nodes.[from] agent
    nodes.[n] <-
      Agent.ReduceEnergy e.Weight agent
  | > Node.AddAgent nodes.[n] true
  | Some e ->
    Agent.ReduceEnergy settings.FailedGotoCost agent
  | > ReplaceAgent g false
  | _ -> false

// Color the dominated nodes in a graph.
// Has sideeffects.
// Graph → unit

let ColorDominatedNodes ((nodes,_) : Graph) =
  nodes
  |> Array.iteri (fun n node -> nodes.[n] <- Node.Color node)

// Color nodes that have at least 2 neighbour nodes of the same color, using the color of the dominating neighbouring team.
// In case of a draw, it will not be colored.
// Has sideeffects.
// Graph → unit

let ColorNeighbours ((nodes,edges) as g : Graph) =
  let undominated =
    nodes
    |> Array.toList
    |> List.filter (fun n ->
      n.DominatingTeam = None
      && not (Node.HasActiveAgent n))
    |> List.map (fun n -> n.NodeID)
  let domination =
    undominated
    |> List.map (fun n ->
      NeighbourNodes g n
      |> List.choose (fun i -> nodes.[i].DominatingTeam))
    |> List.map (Dominator 2)
  let setdomination n dom =
    if dom <> None
      then nodes.[n] <- { nodes.[n] with DominatingTeam = dom }
    else ()
  List.iter2 setdomination undominated domination

// Find a zone given a starting node number. Uses two given
To determine whether to accept a node into the zone, or whether the zone is invalid if a certain node is given. The accept function must return true for nodes that are to be accepted. The fail function must return true for nodes that invalidate the zone.

\[(\text{Node} \to \text{bool}) \to (\text{Node} \to \text{bool}) \to \text{Graph} \to \text{int} \rightarrow \text{int list Option}\]

let GetZone accept fail ((nodes,_) as g : Graph) n =
    let rec GetZoneHelper zonelist trylist =
        match trylist with
        | [] when zonelist <> [] -> Some zonelist
        | hd::tl when accept nodes.[hd] ->
            let notmem n = not (lmem n (zonelist @ trylist))
            let neighbours =
                NeighbourNodes g hd
                |> List.filter notmem
            GetZoneHelper (hd::zonelist) (tl @ neighbours)
        | hd::tl when not (fail nodes.[hd]) ->
            GetZoneHelper zonelist tl
        | _ -> None
    GetZoneHelper [] [n]

// Color the zones in a graph.
// Has side effects.
// \text{Graph} \to \text{unit}
let ColorZones ((nodes,edges) as g : Graph) =
    let rec colorzone (dom,zoneoption) =
        match zoneoption with
        | None | Some ([]) -> ()
        | Some (hd::tl) ->
            nodes.[hd] <-
            { nodes.[hd] with DominatingTeam = dom }
            colorzone (dom,(Some (tl)))
        let accept col (node : Node) =
            node.DominatingTeam = None && not (Node.HasActiveAgent node)
        let fail col (node : Node) =
            let dom = node.DominatingTeam
            (dom = None && Node.HasActiveAgent node)
            || (dom <> None && dom <> col)
        let getuncoloredneighbours (node : Node) =
            let dom = node.DominatingTeam
            NeighbourNodes g node.NodeID
            |> List.map (fun n -> nodes.[n])
            |> List.filter (fun n -> n.DominatingTeam = None)
            |> List.map (fun n -> (dom,n))
            nodes
            |> Array.toList
            |> List.filter (fun n -> n.DominatingTeam <> None)
            |> List.map getuncoloredneighbours
            |> List.concat
            |> List.map (fun (col,n) ->
                    (col, GetZone (accept col) (fail col) g n.NodeID))
            |> List.iter colorzone

// Color the edges in a graph.
// Has side effects.
// Graph → unit
let ColorEdges ((nodes, edges) as g : Graph) =
  edges
  |> Array.iteri
  (fun x ->
    Array.iteri
    (fun y edge ->
      edges.[x].[y] <- Edge.Color g edge))

// Reset the coloring of a graph (Remove all coloring).
// Has side effects.
// Graph → unit
let ResetColoring ((nodes, edges) : Graph) =
  nodes
  |> Array.iteri
  (fun n node ->
    nodes.[n] <- Node.RemoveDominatingTeam node)
  edges
  |> Array.iteri
  (fun x ->
    Array.iteri
    (fun y edge ->
      edges.[x].[y] <- Edge.RemoveDominatingTeam edge))

// Survey the edges of a graph, starting at a certain node.
// It will survey all edges connected to the given node.
// Returns the amount of edges surveyed.
// Has side effects.
// Graph → int → (int * int) → int
let SurveyEdges ((nodes, edges) as g : Graph) n id =
  let initialedges =
    edges.[n]
    |> Array.choose (fun a -> a)
    |> Array.toList
    |> List.filter (fun e -> not (Edge.SurveyedBy id e))
  initialedges
  |> List.choose (fun a -> edges.[a.To].[a.From])
  |> List.append initialedges
  |> List.map (fun e -> Edge.AddAgentSurveyed e id)
  |> List.iter (ReplaceEdge g)
  List.length initialedges

// Inspect the nearby agents on a graph, given a node number.
// It will inspect all agents on the given node and on neighbour nodes, that haven’t already been inspected by any agent on the same team as the given agent id.
// Returns the amount of agents inspected.
// Has side effects.
// Graph → int → int * int → int
let InspectAgents ((nodes, edges) as g : Graph) n id =
  let agents =
    n:(NeighbourNodes g n)
    |> List.map (fun x -> nodes.[x].Agents)
    |> List.concat
| > List.filter (fun x -> not (Agent.InspectedBy id x))
| > List.map (Agent.AddAgentInspected id)
List.iter (ReplaceAgent g) agents
List.length agents

// Create an edge between two nodes in a graph, with a given
// weight. Will create two edges, as the graph is to be
// bi-directional.
// Has side effects.
// Graph -> int * int * int -> unit
let AddEdge graph (n1, n2, w) =
[Edge.Create(w, n1, n2); Edge.Create(w, n2, n1)]
| > List.iter (ReplaceEdge graph)

// Create a graph using given values for number of nodes,
// width and height of the graph (when drawing it), and
// minimum and maximum values for edges and nodes.
// The algorithm will make a Delaunay triangulation on the
generated nodes, and use this for the edges.
// Delauney triangulation module from:
// http://www.cs-hull.org/ (Phil Atkin’s C# code)
// int * int * int * int * int * int * int * int * int * int * int
// -> Graph
let Create (v, w, h, minn, maxn, mine, maxe) : Graph =
let rand = new Random(DateTime.Now.Ticks)
let coords =
[ for x in 0..w do for y in 0..h do yield (x, y) ]
| > RandomizeList | > Array.ofList
let nodes = Array.init v (fun n ->
let weight = rand.Next(minn, maxn)
Node.Create(weight, coords.[n], n))
let edges =
Array.init v (fun _ -> Array.init v (fun _ -> None))
let points =
nodes
| > List.ofArray
| > List.map (fun a -> a.Coordinate)
| > List.map (fun (a,b) -> new Vertex(float32 a, float32 b))
let angulator = new Triangulator()
let triangles =
angulator.Triangulation(ToGenericList points)
| > OfGenericList
let addtriad (t : Triad) =
let e1, e2, e3 =
rand.Next(mine, maxe),
rand.Next(mine, maxe),
rand.Next(mine, maxe)
[t.a,t.b,e1; t.a,t.c,e2; t.b,t.c,e3]
| > List.iter (AddEdge (nodes, edges))
triangles |> List.iter addtriad
(nodes,edges)

// Get the amount of edges in the graph, assuming all edges
// are bidirectional. A bidirectional edge will count as one
// edge.
// Graph -> int
let NumEdges ((_, edges) : Graph) =
  edges
  |> Array.toList
  |> List.map (Array.toList)
  |> List.concat
  |> List.choose (fun a -> a)
  |> List.length
  |> fun n -> n / 2

// Given an agent and a graph, get the version of the agent
// the graph knows. Assuming same position of old and updated
let GetUpdatedAgent ((nodes,_) : Graph) agent =
  let agents = nodes.[agent.Node].Agents
  match List.tryFind (fun a -> a.ID = agent.ID) agents with
  | Some a -> a
  | _ -> agent
module MAS2011.GtkHelpers

(*
 A collection of helper functions for the GTK# library
 Most of the functions are 'convenience' functions
 *)

open Gtk

// Attach options with short names
let fill = AttachOptions.Fill
let expand = AttachOptions.Expand
let fillEx = fill ||| expand

// Creates a new SpinButton with the 3 given values
// int * int * int -> SpinButton
let sb (a, b, c) = new SpinButton(float a, float b, float c)

// Set the value of the given SpinButton to n.
// Has sideeffects.
// SpinButton -> int -> unit
let sbLoadVal (s : SpinButton) n =
    s.Value <- (float n)

// Set the selected value of a given ComboBox to n.
// Has sideeffects.
// ComboBox -> int -> unit
let cbLoadVal (c : ComboBox) n =
    c.Active <- n

// Set the 'active' value of a given CheckButton according to n.
// Has sideeffects.
// CheckButton -> bool -> unit
let chLoadVal (c : CheckButton) b =
    c.Active <- b

// Set the color value of a given ColorButton according to the values in cl. Cl must be of length 3.
// Has sideeffects.
// ColorButton -> byte list -> unit
let colLoadVal (c : ColorButton) cl =
    match cl with
    | r::g::b::[] =>$> let newcol = new Gdk.Color(r,g,b)
    c.Color <- newcol
    | _ =>$> ()

// Creates a label from a given string.
// string -> Label
let l (s : string) = new Label(s)

// Gets the value from a SpinButton.
// SpinButton -> int
let spinval (s : SpinButton) = s.ValueAsInt

// Gets the 'active' value from a CheckButton.
// CheckButton -> boo
let checkval (c : CheckButton) = c.Active

// Gets the selected value from a ComboBox.
// ComboBox -> string
let comboval (c : ComboBox) = c.ActiveText

// Gets the color value from a ColorButton
// ColorButton -> Gdk.Color
let colorval (c : ColorButton) = c.Color

// Gets the color value from a ColorButton, as RGB values
// ColorButton -> uint16 list
let colorvalrgb (c : ColorButton) =
B.11 IAgent.fs

```fsharp
module MAS2011.Shared.IAgent

(*
Interface for agents. Agents should also define a name.
*)

Usage:

open System.ComponentModel.Composition

[<ExportMetadata("Name", "Some name for the agent here")>]
[<Export(typeof<IAgent>)>]
type SomeAgent() = ... // class definition
... // Variables and functions
  interface IAgent with
    ... // Interface functions
  *)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.ShapePrimitives

type IAgent =
  abstract MakeStep : Percept -> (Action * Message list)
  abstract AddSettings : SimulationSettings -> unit
  abstract SetNumNodesEdgesSteps : int * int * int -> unit

type IAgentMetadata =
  abstract Name : string
```
module MAS2011.Initial

(*
Initial file for the simulator. Creates a SimulationWindow,
shows it and starts GTK’s run-loop.*
)

open Monitor.SimulationWindow
open Gtk
open MAS2011.Simulation

Application.Init()

let sw = new SimulationWindow()
sw.Destructed.Add(fun _ -> Application.Quit())
sw.Destructed.Add(fun _ -> KillSim())
sw.ShowAll()
sw.Maximize()

Application.Run()
module Node

(*
Module containing functions for manipulation of the Node type.
*)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics

// Determine whether a given node contains an agent with a given id.
// Node -> int * int -> bool
let HasAgent node a = List.exists (fun x -> a.ID = x.ID) node.Agents

// Add a given agent to a node.
// Node -> Agent -> Node
let AddAgent node a = if HasAgent node a then node else let agent = { a with Node = node.NodeID } { node with Agents = agent::node.Agents }

// Remove an agent from a node.
// Node -> Agent -> Node
let RemoveAgent n a = let agents = n.Agents |> List.filter (fun x -> x.ID <> a.ID) { n with Agents = agents }

// Replace an agent on a node. If the node doesn't contain the agent in the first place, nothing is changed.
// Node -> Agent -> Node
let ReplaceAgent n a = if HasAgent n a then RemoveAgent n a |> fun n2 -> AddAgent n2 a else n

// Add that an agent probed a given node.
// Node -> int * int -> Node
let AddAgentProbed node id = if lmem id node.AgentsProbed then node else { node with AgentsProbed = id::node.AgentsProbed }

// Remove the dominating team from a node.
// Node -> Node
let RemoveDominatingTeam n : Node = { n with DominatingTeam = None }
// Color a given node by the dominating team on it.
// Considers only active agents.
// Node -> Node
let Color (node : Node) =
  let teams =
    node.Agents
    |> List.choose
    (fun a ->
      let (teambid, _) = a.ID
      if Agent.IsDisabled a
        then None
        else Some teamid)
    { node with DominatingTeam = (Dominator 1 teams) }

// Determine whether a node has any active agents on it.
// Node -> bool
let HasActiveAgent node =
  node.Agents
  |> List.exists (fun a -> not (Agent.IsDisabled a))

// Determine whether a node has been probed by a given team.
// int * 'a -> Node -> bool
let ProbedBy (team, _) node =
  List.exists (fun (t, _) -> team = t) node.AgentsProbed

// Anonymize a node, according to whether it has been probed
// by a certain team. If not, its weight is set to 1.
// int * int -> Node -> Node
let Anonymize (team, a) node =
  if ProbedBy (team, a) node
  then
    { node with
      AgentsProbed = [team,a]
      Agents =
        node.Agents
        |> List.map (Agent.Anonymize (team,a)) }
  else
    { node with
      Weight = 1
      AgentsProbed = []
      Agents =
        node.Agents
        |> List.map (Agent.Anonymize (team,a)) }

// Make a Node with a certain weight, ID and coordinate.
// The coordinate only has meaning when drawing the graph.
// int -> int * int -> int -> Node
let Create (w,c,n) =
  { NodeID = n
    Weight = w
    Agents = []
    AgentsProbed = []
    Coordinate = c
    DominatingTeam = None }
module MAS2011.Shared.ShapePrimitives

(*
Shape primitives used for drawing. The simulation can draw
anything, as long as it consists of these custom types.*)

type Point = float * float

type Color =
  | Hex of byte * byte * byte
  | Red
  | Blue
  | Black
  | Yellow
  | Green
  | White

type ColorType =
  | Stroked of Color
  | Filled of Color
  | StrokedFilled of Color * Color

// Shapes:
// Circle: coordinate (center) * radius * ColorType
// Rectangle: coordinate (upper left) * size * ColorType
// Line: Start coordinate * end coordinate * Color
// Text: Coordinate * text * Color

type Shape =
  | Circle of Point * float * ColorType
  | Rectangle of Point * Point * ColorType
  | Line of Point * Point * Color
  | Text of Point * string * Color

// Convert a given Gdk.Color to the custom Color-type
// Gdk.Color --> Color
let GdkColor (gc : Gdk.Color) =
  Hex (byte (gc.Red/256us),
       byte (gc.Green/256us),
       byte (gc.Blue/256us))

// Displace a shape by (dx,dy)
// float * float --> Shape --> Shape
let DisplaceShape (dx,dy) s =
  match s with
  | Circle ((x,y),r,ct) -> Circle ((x+dx,y+dy),r,ct)
  | Rectangle ((x,y),dim,ct) -> Rectangle ((x+dx,y+dy),dim,ct)
  | Line ((x1,y1),(x2,y2),c) ->
    Line ((x1+dx,y1+dy),(x2+dx,y2+dy),c)
  | Text ((x,y),s,c) -> Text ((x+dx,y+dy),s,c)
module MAS2011.Monitor.SimSettingsGeneral

(*
Module containing the 'general' pane of the simulation settings
window, in a class called SimSettingsGeneral.*
*)

open Gtk
open MAS2011.GtkHelpers
open MAS2011.Shared.Generics

// Class SimSettingsGeneral - inherits from Gtk.ScrolledWindow
type SimSettingsGeneral () as this = class
  inherit ScrolledWindow()

let x = (0,100,1)
let spinOptions1 = [1,10,1; 1,100,1; 50,10000,50; x; 50,50000,50; x; x]
> List.map sb
let spinOptions2 = [5,1000,1; 3,100,1; 3,100,1; x; 1,100,1; x; 1,100,1]
> List.map sb
let spinOptions3 = [x; x; x; x; x; x; x]
> List.map sb
let viewpercept = new CheckButton()

let contenttable = new Table(7u,6u,false)
do
  this.AddViewport(contenttable)
  let ca a n o =
    let un = uint32 n
    contenttable.Attach(o,a,a+1u,un,un+1u,fill,fill,4u,4u)
    List.iteri (ca 1u) spinOptions1
    List.iteri (ca 3u) spinOptions2
    List.iteri (ca 5u) spinOptions3
  ca 3u 7 viewpercept

["Number_of_teams";
"Number_of_agents_on_each_team";
"Simulation_length_(ticks)";
"Failed_action_chance_(%)";
"Max_response_time_for_agents_(ms)";
"Recoverrate_active";
"Recoverrate_disabled"]
> List.map l |> List.iteri (ca 0u)
["Number_of_nodes"; "Grid_width"; "Grid_height";
"Min_node_weight"; "Max_node_weight";
"Min_edge_weight"; "Max_edge_weight";
"View_agent_perceptions"]
| List.map l | List.iteri (ca 2u) |
| "Attack" | "Parry" | "Probe" |
| "Survey" | "Inspect" | "Buy" |
| "Repair" | "Failed goto" |
| "Upgrade battery" | "Upgrade shield" |
| "Upgrade sensor" | "Upgrade sabotage device" |

// Add a function to be called when the number of teams are changed. The function must be of type:
// (int --> unit)
member __.AddTeamsChangedHandler f =
  let n () = spinOptions1.[0].ValueAsInt
  spinOptions1.[0].ValueChanged.Add(fun x -> f(n(x)))

// Add a function to be called when the number of agents are changed. The function must be of type:
// (int --> unit)
member __.AddAgentsChangedHandler f =
  let n () = spinOptions1.[1].ValueAsInt
  spinOptions1.[1].ValueChanged.Add(fun x -> f(n(x)))

// Returns the information from the various boxes.
// unit --> int list
member __.GetInformation () =
  spinOptions1 @ spinOptions2 @ spinOptions3
| List.map spinVal

// Convert this to a string, for saving the settings.
// unit --> string
override this.ToString () =
  this.GetInformation() |> List.map string |> StringGlue ","

// Load settings from a string. The values are to be separated by a comma, and there must be 26 values.
// string --> unit
member __.LoadFromString s =
  try
    let vals =
      s |> StringToList ',,' |> List.map int |> Array.ofList
    if Array.length vals = 26
    then
      List.iteri sbLoadVal spinOptions1
      (vals.[..6] |> Array.toList)
      List.iteri sbLoadVal spinOptions2
      (vals.[7..13] |> Array.toList)
      List.iteri sbLoadVal spinOptions3
      (vals.[14..] |> Array.toList)
    else
      printfn "List of wrong length: %d (should be 26)"
      (Array.length vals)
    with
    | :? System.FormatException ->
printfn "String list couldn't be cast to integers"

// Whether the viewpercept CheckButton is active or not
// unit -> bool
member -- ViewPercept () = viewpercept.Active
B.16 SimSettingsMilestones.fs

module MAS2011.Monitor.SimSettingsMilestones

(*
Module containing the 'milestones' pane of the simulation
settings window, in the form of a class.
*)

open Gtk
open MAS2011.GtkHelpers
open MAS2011.Shared.Generics

type Milestone =
  { Type : ComboBox
  Values : SpinButton list }

// Class SimSettingsMilestones - inherits from Gtk.ScrolledWindow
type SimSettingsMilestones () as this = class
  inherit ScrolledWindow()

  let maxms = 25
  let mutable curms = 0

  let types =
    [|"Zone values";"Probed vertices";
     "Surveyed edges";"Inspected vehicles";
     "Successful attacks";"Successful parries"|]

  let makems =
    let x = 0.1,000.1
    { Type = new ComboBox(types, Active=0)
      Values = [x; x; x] |> List.map sb }

  let Milestones = Array.init maxms makems

  let contenttable = new Table(4u,2u,true)

  let ca r n o =
    let un = uint32 n
    contenttable.Attach(o,un,un+1u,r,r+1u,fill,fill,5u,5u)

  do
    this . AddWithViewport(contenttable)

    let addbutton = new Button("Add milestone")
    let rembutton = new Button("Remove last milestone")

    [addbutton; rembutton] |> List.iteri (ca 0u)

    addbutton . Clicked . Add(fun _ -> this . AddMilestone())
    rembutton . Clicked . Add(fun _ -> this . RemoveMilestone())

    [|"Milestone type";"Min. amount";"First team";"Other teams"|]
  > List.map l |> List.iteri (ca 1u)

  // Add a milestone.
  // unit -> unit
member __. AddMilestone () =
  if curms < maxms then
    contenttable.Resize(uint32(curms + 3), 4u)
    let ms = Milestones.[curms]
    let row = uint32(curms + 2)
    ms.Type |> ca row 0
    ms.Values |> List.iteri (fun n -> ca row (n + 1))
    curms <- curms + 1
    this.ShowAll()

// Remove the last milestone.
// unit -> unit
member __. RemoveMilestone () =
  if curms > 0 then
    let rem w = contenttable.Remove(w)
    let ms = Milestones.[curms - 1]
    ms.Type |> rem
    ms.Values |> List.iter rem
    curms <- curms - 1
    contenttable.Resize(uint32(curms + 2), 4u)

// Get the information the user has entered.
// unit -> (string * int list) list
member __. GetInformation () =
  let getval ms =
    (ms.Type |> comboval, ms.Values |> List.map spinval)
    Milestones |> Seq.take curms |> Seq.map getval
    |> Seq.toList

// Convert the information to a string.
// unit -> string
override __. ToString () =
  let mtostr ms =
    let tx = ms.Type.ActiveText
    let ss = ms.Values |> List.map spinval |> List.map string
    tx :: ss |> StringGlue ","
    if curms = 0 then ";" else
      Milestones.[..(curms - 1)]
      |> Array.toList
      |> List.map mtostr
      |> StringGlue ";;"

// Load values from a string. The different milestones
// must be seperated by a semicolon, and the values of a
// milestone must be seperated by a comma.
// string -> unit
member this.LoadFromString (s : string) =
  let ts = types |> Array.toList
  let getnumcb s =
    match List.TryFindIndex (fun x -> x = s) ts with
    | Some x -> x
let addsingle n sl =
  let vals = StringToList ', ' sl |> Array.ofList
  let ms = Milestones.[n]
  if Array.length vals = 4 then
    try
      let vallist =
        vals.[1..] |> Array.toList |> List.map int
        ms.Type.Active <- getnumcb vals.[0]
        List.iter2 sbLoadVal ms.Values vallist
      this.AddMilestone()
    with
      | :? System.FormatException ->
        printfn "Invalid integer for milestone" 
      else printfn "Wrong length list for milestones"
    while curms > 0 do this.RemoveMilestone()
  s |> StringToList '; ' |> List.iteri addsingle
module MAS2011.Monitor.SimSettingsRoles

(*
Module containing the 'roles' pane of the simulation settings
window, in the form of a class.
*)

open Gtk
open MAS2011.GtkHelpers
open MAS2011.Shared.Generic

type AgentRole =
  { Name : ComboBox
    Stats : SpinButton list
    Actions : CheckButton list
    Teams : ComboBox list }

// Set the values of a given AgentRole (see above), to the
// values given in two lists.
// Has sideeffects.
let SetValues a (stats,actions) =
  if List.length a.Stats = List.length stats
  && List.length a.Actions = List.length actions
  then
    List.iter2 sbLoadVal a.Stats stats
    let cbLoadVal (c : CheckButton) b =
      c.Active <- b
    List.iter2 cbLoadVal a.Actions actions

// Set the proper roles of a given AgentRole. The role depends
// on the selected name in the ComboBox.
// The roles are defined in the scenario description,
// describing the scenario to be simulated.
// Has sideeffects.
// AgentRole -> unit
let SetRole a =
  let t,f = true,false
  match a.Name.ActiveText with
    | "Explorer" ->
      SetValues a ([0;12;4;2], [f;f;t;t;t;t])
    | "Repairer" ->
      SetValues a ([0;8;6;1], [f;t;t;t;t;t])
    | "Saboteur" ->
      SetValues a ([4;7;3;1], [t;t;t;t;t;t])
    | "Sentinel" ->
      SetValues a ([0;10;1;3], [f;t;t;t;t;t])
    | "Inspector" ->
      SetValues a ([0;8;6;1], [f;f;t;t;t;t])
    | _ -> ()

// Get the ComboBox associated with a certain team in a given
// AgentRole. The ComboBox represents the AI to be used.
// int -> AgentRole -> ComboBox
let getteam n a = a.Teams.[n]

// SimSettingsRoles class - inherits Gtk.ScrolledWindow
// Instantiated with a list of available AI's, in the form
// of a string list
type SimSettingsRoles (agentlist) as this = class
  inherit ScrolledWindow()

  let agents = List.toArray (agentlist)
  let mutable numAgents = 0
  let mutable numTeams = 4
  let mutable RoleChangeDisabled = false

  let ainames = agentlist

  let names =
    ["Custom"; "Explorer"; "Repairer";
     "Saboteur"; "Sentinel"; "Inspector"]

  let CreateRow n =
    { Name = new ComboBox(names, Active=0);
      Stats = [ for i in 1..4 do yield sb(0,20,1) ];
      Actions = [ for i in 1..7 do yield new CheckButton() ];
      Teams = [ for i in 1..10 do yield new ComboBox(agents, Active=0) ] }

  let AgentRoles = Array.init 10 CreateRow

  let contenttable = new Table(1u,13u,false)
  let carco =
    contenttable.Attach(o,c,c+1u,r,r+1u,fill,fill,4u,4u)

  let teamcolors = List.init 10 (fun n -> new ColorButton())

  do
    this.AddWithViewport(contenttable)
    let ca2 n = ca 0u (uint32 (n+1))
    ["Strength"; "Max energy"; "Health"; "Visibility";
     "'Attack'"; "'Parry'"; "'Probe'"; "'Survey'";
     "'Inspect'"; "'Buy'"; "'Repair'"]
    |> List.map l |> List.iteri ca2
    teamcolors |> Seq.take numTeams
    |> Seq.iteri (fun n -> ca2 (n+11))

  let SetHandlers a =
    let sr _ =
      RoleChangeDisabled <- true
      SetRole a
      RoleChangeDisabled <- false
    a.Name.Changed.Add sr

  let SetCustom _ =

if not RoleChangeDisabled
    then a.Name.Active <- 0
    a.Stats |> List.iter
        (fun sb -> sbValueChanged.Add (SetCustom))
    a.Actions |> List.iter
        (fun cb -> cb.Toggled.Add (SetCustom))
AgentRoles |> Array.iter SetHandlers

// Display row of roles number n
// int -> unit
member internal __.AddRoleRow n =
    let un = uint32 (n+1)
    let ca2 d i = ca un (uint32 (i+d))
    let role = AgentRoles.[n]
    role.Name |> List.iteri (ca2 1)
    role.Actions |> List.iteri (ca2 5)
    role.Teams |> Seq.take numTeams |> Seq.iteri (ca2 12)

// Hide row of roles number n
// int -> unit
member internal __.RemoveRoleRow n =
    let role = AgentRoles.[n]
    let rem w = contenttable.Remove(w)
    role.Name |> rem
    role.Stats |> List.iter rem
    role.Actions |> List.iter rem
    role.Teams |> Seq.take numTeams |> Seq.iter rem

// Display column of teams number n
// int -> unit
member internal __.AddTeamColumn n =
    let un = uint32 (n+12)
    let ca2 d i = ca (uint32 (i+d)) un
    let ca2 0 0 teamcolors.[n]
    AgentRoles |> Seq.map (getteam n) |> Seq.take numAgents
    |> Seq.iteri (ca2 1)

// Hide column of teams number n
// int -> unit
member internal __.RemoveTeamColumn n =
    contenttable.Remove(teamcolors.[n])
    let rem w = contenttable.Remove(w)
    Array.map (getteam n) AgentRoles.[..numAgents-1]
    |> Array.iter rem

// Number of agent roles updated to n
// int -> unit
member this.UpdateNumAgents n =
    contenttable.Resize(uint32 (n+1),uint32 (numTeams+12))
    let nOld = numAgents
    if nOld < n // Increased number of agents
        then [nOld...(n-1)] |> List.iter this.AddRoleRow
    if nOld > n // Decreased number of agents
        then [n..(nOld-1)] |> List.iter this.RemoveRoleRow
numAgents <- n
this.ShowAll()

// Number of teams updated to t
// int -> unit
member this.UpdateNumTeams t =
  contenttable.Resize(uint32 (numAgents+1), uint32 (t+12))
  let tOld = numTeams
  if tOld < t // Increased number of teams
     then [tOld..(t-1)] |> List.iter this.AddTeamColumn
  if tOld > t // Decreased number of teams
     then [t..(tOld-1)] |> List.iter this.RemoveTeamColumn
  numTeams <- t
  this.ShowAll()

// Get the information the user has entered.
// unit -> Gdk.Color list * (int list * bool list * string list) list
member __.GetInformation () =
  let getval a =
    (a.Stats |> List.map spinval,
     a.Actions |> List.map checkval,
     a.Teams |> Seq.take numTeams
       |> Seq.map comboval |> Seq.toList)
  let roles =
    AgentRoles |> Seq.take numAgents |> Seq.map getval
       |> Seq.toList
  let teams =
    teamcolors |> Seq.take numTeams |> Seq.map colorval
       |> Seq.toList
  (teams, roles)

// Convert the information the user has entered, to a string.
// Information on all roles and all teams will be generated,
// not just the displayed ones.
// unit -> string
override __.ToString () =
  let btostr b =
    if b then "true" else "false"
  let artostr ar =
    let ct = ar.Name.ActiveText
    let st =
      ar.Stats
      |> List.map spinval
      |> List.map string
    let at =
      ar.Actions
      |> List.map checkval
      |> List.map btostr
    let ait =
      ar.Teams
      |> List.map comboval
    ct::(st@at@ait) |> StringGlue ","
| List.map colorvalrgb
| List.concat
| List.map string
| StringGlue ";"

AgentRoles
| Array.toList
| List.map artostr
| List.append [colors]
| StringGlue ","

// Load information from a string. The different roles must
// be separated by a semicolon, and the different values
// must be separated by comma.
// Furthermore, the first element in the semicolon-separated
// list must be a list of colors. The colors must be
// separated by comma, with the RGB values of each color
// all in the list of colors.Chs
// string -> unit

member __.LoadFromString s =
let strobbool s =
  match s with
  | "true" -> true
  | _ -> false

let ns = names |> Array.toList
let getnumname s =
  match ListTryFindIndex (fun x -> x = s) ns with
  | Some x -> x
  | _ -> 0

let getnumai s =
  match ListTryFindIndex (fun x -> x = s) ainames with
  | Some x -> x
  | _ -> 0

let addsingle n sl =
  let vals = StringToList ',’ sl |> Array.ofList
  let ag = AgentRoles.[n]
  if Array.length vals = 22
  then
  try
    let role = getnumname vals.[0]
    ag.Name.Active <- role
    let ais =
      vals.[12..] |> Array.toList |> List.map getnumai
    List.iter2 cbLoadVal ag.Teams ais
    if role = 0
    then
      let statlist =
        vals.[1..4] |> Array.toList |> List.map int
    let aclist =
      vals.[5..11] |> Array.toList |> List.map strobbool
    List.iter2 sbLoadVal ag.Stats statlist
    List.iter2 chLoadVal ag.Actions aclist
    with
    | :? System.FormatException ->
printfn "Invalid integer for agent−role"
else printfn "Invalid length of agent−role list"
let vals = StringToList ';
match vals with
| color::roles when List.length roles = 100 ->
List.iteri addsingle roles
try
  let colors =
  color
  |> StringToList ','
  |> List.map (fun x -> (int x)/256 |> byte)
  |> ListDivide3
  List.iter2 colLoadVal teamcolors colors
  with
  | :? System.FormatException ->
    printfn "Invalid number for color"
  | _ -> printfn "Wrong length of roles−list"
module MAS2011.Monitor.SimSettingsWindow

(* Module containing the simulation settings window, in the form
  of a class. *)

open Gtk
open MAS2011.GtkHelpers
open MAS2011.Monitor.SimSettingsGeneral
open MAS2011.Monitor.SimSettingsRoles
open MAS2011.Monitor.SimSettingsMilestones
open MAS2011.Simulation
open MAS2011.Shared.Generics

// Class SimSettingsWindow - inherits Gtk.Window
// Instantiated with a list of available AI's, in the form
// of a string list
type SimSettingsWindow (agentlist) as this = class
  inherit Window("Enter simulation settings")

let gen = new SimSettingsGeneral()
let roles = new SimSettingsRoles(agentlist)
let ms = new SimSettingsMilestones()
let contenttable = new Table(2u,6u,false)
let nb = new Notebook()

do
  this.DefaultSize <- new Gdk.Size(950,550)
  this.Add(contenttable)
  nb.AppendPage(gen,1 "General") |> ignore
  nb.AppendPage(roles,1 "Agent roles") |> ignore
  nb.AppendPage(ms,1 "Milestones") |> ignore

  let okb = new Button("OK")
  let canb = new Button("Cancel")
  let defb = new Button("Defaults")
  let openb = new Button("Open settings...")
  let saveb = new Button("Save settings...")
  okb.Clicked.Add(fun _ -> this.Destroy())
  canb.Clicked.Add(fun _ -> this.Destroy())
  defb.Clicked.Add(fun _ -> this.SetDefaultValues())
  contenttable.Attach(nb,0u,6u,0u,1u,fillx,fillex,5u,5u)
  contenttable.Attach(okb,0u,1u,1u,2u,fill,fill,5u,5u)
  contenttable.Attach(canb,1u,2u,1u,2u,fill,fill,5u,5u)
  contenttable.Attach(defb,2u,3u,1u,2u,fill,fill,5u,5u)
  contenttable.Attach(openb,3u,4u,1u,2u,fill,fill,5u,5u)
  contenttable.Attach(saveb,4u,5u,1u,2u,fill,fill,5u,5u)
  contenttable.Attach(new Statusbar(),5u,6u,1u,2u,fill,fill,5u,5u)
gen.AddTeamsChangedHandler roles.UpdateNumTeams
gen.AddAgentsChangedHandler roles.UpdateNumAgents
this.SetDefaultValues()

let startsim _ =
    let s = gen.GetInformation()
    let r = roles.GetInformation()
    let m = ms.GetInformation()
    if List.length s = 26 && s.[7] <= s.[8] * s.[9]
    then
        StartSim s r m (gen.ViewPercept())
    else
        printfn "Error, starting sim, following must be true:"
        printfn "%d <= %d (%d * %d) for %d <= %d"
            s.[7] (s.[8] * s.[9])
        printfn "%d < %d"
            s.[10] s.[11]
        printfn "%d < %d"
            s.[12] s.[13]
okb.Clicked.Add(startsim)

let openfile _ =
    let fc =
        new FileChooserDialog
            ("Choose the map to open",
             this, FileChooserAction.Open,
             "Cancel", ResponseType.Cancel,
             "Open", ResponseType.Accept)
    if fc.Run() = int(ResponseType.Accept)
    then
        this.OpenFile fc.Filename
        fc.Destroy() |> ignore
openb.Clicked.Add(openfile)

let savefile _ =
    let fc =
        new FileChooserDialog
            ("Choose the map to open",
             this, FileChooserAction.Save,
             "Cancel", ResponseType.Cancel,
             "Save", ResponseType.Accept)
    if fc.Run() = int(ResponseType.Accept)
    then
        let contents =
            [ gen.ToString(); roles.ToString(); ms.ToString() ]
        |> StringGlue "@"
        let file = fc.Filename
        System.IO.File.WriteAllText(file, contents)
        fc.Destroy() |> ignore
saveb.Clicked.Add(savefile)
this.ShowAll()
// the functions SimSettingsGeneral.LoadFromString, SimSettingsMilestones.LoadFromString and SimSettingsRoles.LoadFromString.

// string -> unit

member __.OpenFile path =
  if System.IO.File.Exists(path) then
    let contents = System.IO.File.ReadAllText(path)
    |> StringToList '@'
    match contents with
    | a::b::c::[] ->
      gen.LoadFromString a
      roles.LoadFromString b
      ms.LoadFromString c
    | _ -> printfn "Wrong length list, unable to open"
  else printfn "Unable to find file: %s" path

// Loads the values in the file DefaultSettings.txt if it exists.

// unit -> unit

member this.SetDefaultValues () =
  this.OpenFile "DefaultSettings.txt"
B.19 SimTypes.fs

```fsharp
module MAS2011.Shared.SimTypes

(*)
Module containing the different types used in the simulation calculations, along with some functions to manipulate/create the different types.
*)

open System
open MAS2011.Shared.Generics

// Type of upgrade an agent wants to perform
type Upgrade =
  | Battery
  | Sensor
  | Shield
  | SabotageDevice

type TeamID = int
type AgentID = int
type NodeID = int

// The action an agent wants to perform
type Action =
  | Skip
  | Recharge
  | Goto of NodeID
  | Attack of TeamID * AgentID
  | Parry
  | Probe
  | Survey
  | Inspect
  | Repair of AgentID
  | Buy of Upgrade

// Stats for an agent
type Agent =
  { ID : TeamID * AgentID
    Node : NodeID
    Health : int
    MaxHealth : int
    Energy : int
    MaxEnergy : int
    Strength : int
    Visibility : int
    Actions : Action list
    AgentsInspected : (TeamID * AgentID) list }

// Node in a graph, along with some simulation-specific information

type Node =
  { NodeID : int
```
// Information about an edge – in a graph, edges are
// EdgeInfo Option’s
type EdgeInfo =
{ Weight : int
  From : int
  To : int
  AgentsSurveyed : (TeamID * AgentID) list
  DominatingTeam : int Option }

type Edge = EdgeInfo option

// Graph, with edges in an adjacency matrix
type Graph = Node array * Edge array array

// Information about a milestone. Format is:
// Amount needed for the milestone to trigger *
// Award for the first team(s) to reach the milestone *
// Award for all other teams reaching the milestone
type Milestone =
| ZoneValues of int * int * int
| ProbedVertices of int * int * int
| SurveyedEdges of int * int * int
| InspectedVehicles of int * int * int
| SuccessfulAttacks of int * int * int
| SuccessfulParries of int * int * int

// A milestone and a set of teams that has completed it
type MilestoneStatus = Milestone * Set<int>

// Status for a team, in order to keep track of score/money
// and status for milestones.
type TeamStatus =
{ Score : int
  Money : int
  Probes : int
  Surveys : int
  Inspections : int
  Attacks : int
  Parries : int }

// Messages agents can send
type MessageContent =
| InfoEdge of EdgeInfo
| InfoNode of Node
| Feromone of NodeID * int

// Messages need a recipient
type Message =
| Broadcast of MessageContent
| Single of int * MessageContent
// Perception given to an agent when a move is requested

type Percept =
{
StepNum : int
Score : int
Money : int
Agent : Agent
Nodes : NodeList
Edges : EdgeInfo list
OtherAgents : Agent list
Milestones : MilestoneStatus list
Messages : (int * MessageContent) list
}

// Various simulation specific settings, defined in the user

// interface
type SimulationSettings =
{
FailChance : int
Length : int
MaxAgentResponse : int
RecoverNormal : int
RecoverDisabled : int
AttackCost : int
ParryCost : int
ProbeCost : int
SurveyCost : int
InspectCost : int
BuyCost : int
RepairCost : int
FailedGotoCost : int
UpgradeBatteryPrice : int
UpgradeSensorPrice : int
UpgradeShieldPrice : int
UpgradeSabotageDevicePrice : int
}

// Parse a Milestone from a string and a list of integers.
// Will only parse a Milestone when the string is one of six
// known values, and the integer list is of length 3.

// string * int list -> MilestoneStatus
let ParseMilestone (s, il) =
let es : Set<int> = Set.empty
match il with
| a::b::c::[] ->
  match s with
| "Zone values" ->
  Some (ZoneValues(a,b,c),es)
| "Probed vertices" ->
  Some (ProbedVertices(a,b,c),es)
| "Surveyed edges" ->
  Some (SurveyedEdges(a,b,c),es)
| "Inspected vehicles" ->
  Some (InspectedVehicles(a,b,c),es)
| "Successful attacks" ->
  Some (SuccessfulAttacks(a,b,c),es)
| "Successful parries" ->
  Some (SuccessfulParries(a,b,c),es)
let GetPrice settings upgrade =
  match upgrade with
  | Battery -> settings.UpgradeBatteryPrice
  | Sensor -> settings.UpgradeSensorPrice
  | Shield -> settings.UpgradeShieldPrice
  | SabotageDevice -> settings.UpgradeSabotageDevicePrice
This module contains functions to convert edges, nodes, agents and thus entire graphs to a list of Shape's. It also contains a few functions for adding these to SimulationSteps, which keeps track of the different steps in the simulation.

---

```fsharp
module MAS2011.Shared.SimTypesDrawing

(*
This module contains functions to convert edges, nodes, agents and thus entire graphs to a list of Shape's. It also contains a few functions for adding these to SimulationSteps, which keeps track of the different steps in the simulation. *)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.ShapePrimitives
open MAS2011.Shared.Generics
open MAS2011.Shared.SimulationSteps

// Text alignment in order to be able to write to the center of a circle.
let internal tdx, internal tdy = -3.0, 3.0

// Scaling value, i.e. distance between nodes when drawing the graph.
let scaling = 100.0

// Create a colored line, depending on a given function. This method is used to display which teams have probed/surveyed/inspected various nodes/edges/agents.
// float * float -> float -> int -> (int -> bool) -> int
// -> Color -> Shape Option
let internal InfoLine (x,y) scale total fn color =
    if fn then
        let lines = CreateCircularLines total
        let dx,dy = lines.[n]
        Some (Line((x,y),(x+scale*dx,y+scale*dy),color))
    else None

// Convert an edge to shapes. Needs the starting and ending coordinates. The edge will be drawn white if it is not dominated by any team.
// float -> (int * int) * (int * int) -> Color list -> EdgeInfo
// -> Shape list
let EdgeToShape scale (f,t) colors (e : EdgeInfo) =
    let x1,y1 = f |> ScaleInts scale
    let x2,y2 = t |> ScaleInts scale
    let x3,y3 = (x1+x2)/2.0, (y1+y2)/2.0
    let mutable color = White
    match e.DominatingTeam with
    | Some z when List.length colors > z ->
        color <- colors.[z]
    | _ -> ()
    let line = Line((x1,y1),(x2,y2),color)
    let circle = Circle((x3,y3),8.0,StrokedFilled(color,Black))
    let text = Text((x3+tdx,y3+tdy),(string e.Weight),color)
```
let fn = Edge.SurveyedBy (n,0) e
let infolines =
    List.mapi (InfoLine (x3,y3) 12.0 (List.length colors) f)
    colors
    |> List.choose (fun a -> a)
line::infolines @ [circle;text]

// Convert an agent to a Shape list. Needs the coordinate of
// the agent. Some lines will be drawn from the center of the
// agent and to the perimeter of its drawing area, displaying
// the agents stats.
// Strength is short, colored lines.
// Energy is short, white lines.
// Visibility is long, colored lines.
// Health is long, white lines.
// float * float -> Color list -> Agent -> Shape list
let AgentToShape (x,y) (colors : Color list) (a : Agent) =
    let maxe,en = a.MaxEnergy,a.Energy
    let maxh,he = a.MaxHealth,a.Health
    let str,vis = a.Strength,a.Visibility
    let team, = a.ID

    let makeline n (dx,dy) =
        let coord = (x+10.0*dx,y+10.0*dy)
        let coord2 = (x+8.5*dx,y+8.5*dy)

        // Draw strength lines
        if n < str
            then Some (Line((x,y),coord2,colors.[team]))
        // Draw energy lines
        else if n < str + en
            then Some (Line((x,y),coord2,White))
        // Don’t draw (max energy – energy) lines
        else if n < str + maxe
            then None
        // Draw visibility lines
        else if n < str + maxe + vis
            then Some (Line((x,y),coord,colors.[team]))
        // Draw health lines
        else if n < str + maxe + vis + he
            then Some (Line((x,y),coord,White))
        // Don’t draw (max health – health) lines
        else None
    let lines =
        CreateCircularLines (maxe+maxh+str+vis)
        |> List.mapi makeline
        |> List.choose (fun a -> a)
    let circle = Circle((x,y),4.0,Filled(colors.[team]))
    let bgcircle = Circle((x,y),10.0,Filled(Black))
    let fn =
        if n = team then false
        else Agent.InspectedBy (n,0) a
    let infolines =
        List.mapi (InfoLine (x,y) 14.0 (List.length colors) f)
        colors
        |> List.choose (fun a -> a)
    let disabledcross =
if Agent.IsDisabled a
then
  [Line((x−4.0,y),(x+4.0,y),Black); Line((x,y−4.0),(x,y+4.0),Black)]
else []

info @ bgcircle @ lines @ circle @ disabledcross
// Array of agent positions, relative to a node. A maximum
// amount of agents on each node is expected, and if this
// maximum is reached the application could crash. The maximum
// is about 90 agents per node.
// (float * float) array
let internal AgentRelPos =
  let makecircle n =
    CreateCircularLines (n*6)
    |> List.map (fun (x,y) ->
      let nf = (float n)*30.0
      (x*nf,y*nf))
    |> Array.ofList
    [| for a in 1..5 do yield makecircle a |]
  |> Array.concat

  // Convert a node to a Shape list. Will draw agents on the node
  // in a circular fashion around itself.
  // float -> Color list -> Node -> Shape list
  let NodeToShape scale colors (node : Node) =
    let x,y = node.Coordinate |> ScaleInts scale
    let mutable color = White
    match node.DominatingTeam with
      | Some z when List.length colors > z ->
        color <- colors.[z]
      | _ -> ()
    let circle = Circle((x,y),10.0,Filled(color))
    let text = Text((x+tdx,y+tdy),(string node.Weight),Black)
    let agentpos i =
      let xc,yc = AgentRelPos.[i]
      (xc+x,yc+y)
    let agents =
      node.Agents
      |> List.map
        (fun i -> AgentToShape (agentpos i) colors)
    |> List.concat
    let fn = Node.ProbedBy (n,0) node
    let infolines =
      List.map (InfoLine (x,y) 14.0 (List.length colors) f) colors
    |> List.choose (fun a -> a)
    infolines @ [circle;text] @ agents

  // Convert a graph to a Shape list, by converting all nodes
  // and edges to shapes.
  // float -> Color list -> Graph -> Shape list
  let GraphToShapes scale colors (g : Graph) =
    let nodes,edges = g
    let getcoords (e : EdgeInfo) =
let edgeshapes =
edges .[1..]
| > Array.mapi (fun i a -> a.[0..i])
| > Array.concat
| > Array.choose (fun a -> a)
| > Array.map
(fun e -> EdgeToShape scale (getcoords e) colors e)
| > List.ofArray
| > List.concat
let nodeshapes =
nodes
| > Array.map (NodeToShape scale colors)
| > List.ofArray
| > List.concat
edgeshapes @ nodeshapes

// Convert an agent's perception to a Shape list by converting
// all visible nodes and edges to shape's.
// float -> Color list -> Percept -> Shape list
let PerceptToShapes scale colors (p : Percept) =
let getcoord n =
match List.tryFind (fun x -> x.NodeID = n) p.Nodes with
| Some x -> x.Coordinate
| _ -> (0,0)
let getcoords (e : EdgeInfo) =
(getcoord e.From, getcoord e.To)
let nodes =
p.Nodes
| > List.map (NodeToShape scale colors)
| > List.concat
let edges =
p.Edges
| > List.map
(fun e -> EdgeToShape scale (getcoords e) colors e)
| > List.concat
edges @ nodes

// Draw a graph by converting it to a Shape list and adding
// it to the SimulationSteps module. Will also draw the
// TeamStatus for all teams.
// Has sideeffects.
// Color list -> int * int -> Graph -> TeamStatus list -> int
// -> unit
let DrawGraph (colors : Color list) (w, h) graph status stepnum =
let strtoshapes dy color x n s =
let y = dy + n*12 |> float
Text((float x, y), s, color)
let strtoshapes a b c = List.mapi (strtoshapes a b c)
let text =
["Score: ";"Money:","Probes: ";"Surveys:","Inspections:","Attacks:","Parries:"]
|> strtoshapes 15 White 20
let scorestext =
status
158 Source code

| > List.map TS.ToList
| > List.map (List.map string)
| > List.map (fun n -> str|to|shapes 15 colors.[n] (n*40+100))
| > List.concat
let shapes =
  GraphToShapes scaling colors graph
| > List.map (DisplaceShape (scaling/2.0,scale|ing/2.0+100.0))
let finalshapes = [text;shapes;scorestext] |> List.concat
let sc = scaling |> int
AddSimStep "Simulation" stepnum ((w*sc,h*sc+100),finalshapes)

// Draw an agents perception by converting it to a Shape list
// and adding it to the SimulationSteps module. Its name depends
// on the agent's ID.
// Has sideeffects.
// Color list -> int * int -> int -> Perce|pt -> unit
let DrawPercept (colors : Color list) (w,h) n p =
let shapes =
  Perce|ptToShapes scaling colors p
| > List.map (DisplaceShape (scaling/2.0,scale|ing/2.0+100.0))
let team,agent = p.Agent.ID
let name = sprintf "Team%d,agent%d" team agent
let sc = scaling |> int
AddSimStep name (n-1) ((w*sc,h*sc+100),shapes)
module MAS2011.Simulation

(*
This module contains the simulation algorithm, along with a lot of helper functions for it. It also contains a function to start a simulation calculation, which will kill a running simulation if one exists, and start a new thread with a new simulation.
*)

open System
open System.Threading
open MAS2011.Shared.SimTypes
open MAS2011.Shared.SimTypesDrawing
open MAS2011.Shared.IAgent
open MAS2011.Shared.Generics
open MAS2011.Shared.ShapePrimitives
open MAS2011.Shared.SimulationSteps

// Simulation thread.
// System.Threading.Thread
let mutable internal thread =
    new Thread(new ThreadStart(fun _ -> ()))

// Abort the running simulation. Used when loading saved simulations.
// Has sideeffects.
// unit -> unit
let KillSim () =
    thread.Abort()

// Expand the view from a list of nodes, IE. add all the neighbour nodes to the list. Do this recursively until n is 1. When n is 1, retrieve the nodes and the edges connecting the found nodes.
// Graph -> int -> int list -> Node list * EdgeInfo list
let rec internal ExpandView ((nodes,edges) as g : Graph) n
    nodelist =
    if n < 1 then
        let getedges i =
            edges.[i]
            |> Array.toList
            |> List.choose (fun a -> a)
            |> List.filter (fun e -> lmem e.To nodelist)
        let newnodes =
            nodelist
            |> List.map (fun i -> nodes.[i])
        let newedges =
            nodelist
            |> List.map (fun i -> getedges i)
            |> List.concat
    else
        internal ExpandView (g:n,m,n-1)
let newnodelist =
| (newnodes, newedges)

else

let newnodelist =
| nodelist
| > List.map (Graph.NeighbourNodes g)
| > List.concat
| > List.append nodelist
| > Set.ofList
| > Set.toList
ExpandView g (n − 1) newnodelist

// Share the perception between agents, based on the zone they
// are in.
// int list -> int -> Node list * EdgeInfo list -> Agent
// Node list * EdgeInfo list -> Node list * EdgeInfo list
let internal SharePercept zonelist team (nodeset, edgeset) agent
| (nodes : Node list, edges : EdgeInfo list) =
| let t, _ = agent.ID
| if t = team && List.exists (fun x -> x = agent.Node) zonelist
| then
| let newnodeset =
| > nodes
| | > Set.ofList
| | > (+) (nodeset |> Set.ofList)
| | > Set.toList
| let newedgeset =
| > edges
| | > Set.ofList
| | > (+) (edgeset |> Set.ofList)
| | > Set.toList
| (newnodeset, newedgeset)
| else (nodeset, edgeset)

// Prepare/create the perceptions for all agents.
// int -> Graph -> (Agent * Message list) -> TeamStatus list
// -> Percept list
let internal PreparePercepts stepnum (g : Graph) ml
| (status : TeamStatus list) milestones =
| let getscore (tid, _) = status.[tid].Score
| let getmoney (tid, _) = status.[tid].Money
| let allagents =
| Graph.GetAllAgents g
| > List.sortBy (fun a -> a.ID)
| // Sorted to get the correct order of agents in the GUI
| let getmsgmap map (agent, msgs) =
| let team, _ = agent.ID
| msgs
| > List.map (fun a -> agent, a)
| > MapAppend map team
| let messages =
| ml |> List.fold getmsgmap Map.empty
| let getmsg (team, agent) =
| match Map.tryFind team messages with
| | Some msgs ->
| | msgs
B.21 Simulation.fs

108 | > List.choose
109  (fun (a,msg) ->
110  let _, sender = a.ID
111  match msg with
112     | Broadcast m -> Some(sender,m)
113     | Single (r,m) when r = agent -> Some(sender,m)
114     | _ -> None)
115  let getagents (team,agent) =
116     allagents |> List.filter (Agent.InspectedBy (team,agent))
117  let vis (a : Agent) = Max 1 a.Visibility
118  let visibilities =
119     allagents
120     |> List.map (fun a -> ExpandView g (vis a) [a.Node])
121  let team a =
122     let (t,.) = a.ID t
123  let accept agent (node : Node) =
124     node.DominatingTeam = Some (team agent)
125  let fail _ = false
126  let zones =
127     allagents
128     |> List.map
129     (fun a -> Graph.GetZone (accept a) fail g a.Node)
130     |> List.map
131     (fun zone ->
132        match zone with
133            | Some z -> z
134            | _ -> []
135     )
136  let sharedvis =
137     List.zip3 allagents visibilities zones
138     |> List.map
139     (fun (a,v,z) ->
140        List.fold2 (SharePercept z (team a)) v
141        allagents visibilities)
142  let makepercept (a : Agent) (nodes,edges) =
143    { StepNum = stepnum
144    Score = getscore a.ID
145    Money = getmoney a.ID
146    Agent = a
147    Nodes = nodes |> List.map (Node.Anonymize a.ID)
148    Edges = edges |> List.map (Edge.Anonymize a.ID)
149    OtherAgents = getagents a.ID
150    Milestones = milestones
151    Messages = getmsg a.ID }
152  List.map2 makepercept allagents sharedvis
153
154 // Fail actions by a certain percentage chance.
155 // SimulationSettings -> 'a list -> 'a list
156 let internal FailActions settings actions =
157    let r = new Random(DateTime.Now.Ticks |> int)
158    let addrandom v =
159        (r.Next(100),v)
160    actions
161    |> List.map addrandom
| > List.filter (fun (a,.) -> a >= settings.FailChance)  
| > List.map (fun (_,a) -> a)  

// Send perceptions to agents, and retrieve their wanted actions  
// and messages they wish to send.  
// SimulationSettings -> Map<int * int,IAgent> -> Percept list  
// (Agent * Action) list * (Agent * Message list) list  
let internal SendPercepts settings (agents : Map<int,IAgent>)
    percepts =
    let answers =
      percepts  
      |> List.map (fun p -> (p.Agent,agents.[p.Agent.ID].MakeStep(p)))
    let msgs =
      answers  
      |> List.map (fun (agent,(.,msgs)) -> (agent,.msgs))
    let actions =
      answers  
      |> List.map (fun (a,(b,.)) -> (a,b))
      |> FailActions settings
      (actions,.msgs)
    // Filter for determining whether a given action is either  
    // attack or parry, or neither.  
    // 'a * Action -> bool  
    let internal AttackParryFilter (_,action) =
      match action with  
        Parry | Attack(_,_) -> true  
        _ -> false
    // Determine whether a given agent and the id of another agent,  
    // are opponents.  
    // Agent -> int * 'a -> bool  
    let internal IsOpponent a (tid,_) =
      let (tid2,_) = a.ID
      tid <> tid2
    // Get a list of the agents who has the energy to perform the  
    // parry action, can do it and isn’t disabled. The agents  
    // energy will be reduced before returning them.  
    // SimulationSettings -> (Agent * Action) list -> Agent list  
    let internal Parries settings actions =
      actions  
      |> List.filter
        (fun (a,ac) ->
          a.Energy >= settings.ParryCost  
          && ac = Parry  
          && not (Agent.IsDisabled a)  
          && Agent.CanPerform a ac)
      |> List.map (fun (a,.) -> a)
      |> List.map (Agent.ReduceEnergy settings.ParryCost)
    // Execute all attack and parry actions. In reality, only attack  
    // actions will be executed, depending on whether or not the  
    // target is parrying. A successful attacks of course also
// depends on the energy, health and position of the attacking agent, and whether or not it can perform the attack in the first place. Has side effects.

SimulationSettings -> TeamStatus list

let internal ExecuteAttackParry settings status ((nodes, _ ) as g : Graph) actions =
    let parries = Parries settings actions
    let ExecuteAttackHelper (status, attacked) (agent, action) =
        (tid, aid) =
        let is parrying = // Is target parrying?
            List.exists (fun a -> a.ID = (tid, aid)) parries
        let target = // Find target agent in graph
            nodes.[agent.Node].Agents |
            List.tryFind (fun a -> a.ID = (tid, aid))
        let newattacker = // Reduce energy for attacker
            Agent.ReduceEnergy settings.AttackCost agent
        match target, is parrying with
            | Some targetagent, false when
                not (Agent.IsDisabled targetagent) ->
                // Successful attack
                let newhealth =
                    Max 0 (targetagent.Health - agent.Strength)
                let newag = { targetagent with Health = newhealth }
                Graph.ReplaceAgent g newag
                Graph.ReplaceAgent g newattacker
                let teamid, _ = agent.ID
                let newstat = List.Replace teamid TS.IncAttacks status
                (newstat, (tid, aid)::attacked)
            | Some targetagent, true -> // Successful parry
                Graph.ReplaceAgent g newattacker
                (List.Replace tid TS.InCParries status, attacked)
            | _ -> // Unable to find target agent in graph
                (status, attacked)
        let ExecuteAttack (status, attacked) (agent, action) =
            match action with
                | Attack(tid, aid) when
                    not (Agent.IsDisabled agent)
                    && IsOpponent agent (tid, aid)
                    && agent.Energy >= settings.AttackCost
                    && Agent.CanPerform agent (Attack(0,0)) ->
                        ExecuteAttackHelper (status, attacked) (agent, action)
                        (tid, aid)
                | _ -> (status, attacked) // Unsuccessful attack
        // Reduce energy for all parrying agents
        List.iter (Graph.ReplaceAgent g) parries
        List.fold ExecuteAttack (status, []) actions

    // Execute single action other than attack and parry.
    // Has side effects.
    SimulationSettings -> Graph -> (int * int) list
    // -> TeamStatus list -> Agent * Action -> TeamStatus list
    let internal ExecuteOtherAction settings
        ((nodes, edges) as g : Graph) attacked status (a, action) =
        let agent = Graph.GetUpdatedAgent g a
if Agent.CanPerform agent action

then

let (teamid, agentid) = agent.ID
let isattacked = List.exists (fun id -> id = agent.ID) attacked
let disabled = Agent.IsDisabled agent
let reduce n =
let newa = Graph.GetUpdatedAgent g a
Agent.ReduceEnergy n newa |> Graph.ReplaceAgent g

match action, isattacked, disabled with

// Recharge - agent hasn't been attacked
| Recharge, false, _ ->
Agent.Recharge settings agent
> Graph.ReplaceAgent g

| Recharge, false, _ -> ignore

// Move agent
| Goto(n); ... when
agent.Energy >= settings.FailedGotoCost
& 0 <= n & n < Array.length nodes ->
Graph.MoveAgent settings g agent n |> ignore

// Probe node - agents hasn’t been attacked and isn’t disabled
| Probe, false, false when
agent.Energy >= settings.ProbeCost
& & not (Node.ProbedBy agent.ID nodes.[agent.Node]) ->
let node = nodes.[agent.Node]
reduce settings.ProbeCost
ListReplace teamid TS.IncProbes status

// Survey edges - agent hasn’t been attacked and isn’t disabled
| Survey, false, false when
agent.Energy >= settings.SurveyCost ->
reduce settings.SurveyCost
let num = Graph.SurveyEdges g agent.Node agent.ID
ListReplace teamid (TS.IncSurveys num) status

// Inspect an opponent agent - agent hasn’t been attacked and isn’t disabled
| Inspect, false, false when
agent.Energy >= settings.InspectCost ->
reduce settings.InspectCost
let num = Graph.InspectAgents g agent.Node agent.ID
ListReplace teamid (TS.IncInspections num) status

// Repair a friendly agent - agent hasn’t been attacked
| Repair(aid), false, _ when
aid <> agentid && agent.Energy >= settings.RepairCost ->
let target =
List.tryFind
(fun a -> a.ID = (teamid, aid))
nodes.[agent.Node].Agents

match target with
| Some a ->
Agent.RepairA a |> Graph.ReplaceAgent g
reduce settings.RepairCost
| - -> ()
| status
| // Upgrade - agent hasn't been attacked and isn't disabled
| Buy(up), false, false when
| agent.Energy >= settings.BuyCost ->
| let money = status.[teamid].Money
| let price = GetPrice settings up
| if money >= price then
| agent
| > Agent.ReduceEnergy settings.BuyCost
| > Agent.Upgrade up
| > Graph.ReplaceAgent g
| ListReplace teamid (TS.ReduceMoney price) status
| else status
| else status
| // Execute all actions other than parry and attack
| // Has sideeffects
| // SimulationSettings -> TeamStatus list -> Graph
| // -> (int * int) list -> (Agent * Action) list
| // -> TeamStatus list
| let internal ExecuteOtherActions settings status graph attacked
| actions =
| List.fold (ExecuteOtherAction settings graph attacked) status
| actions
| // Color a graph using the graph coloring algorithm.
| // Has sideeffects.
| // Graph -> unit
| let internal ColorGraph graph =
| Graph.ResetColoring graph
| Graph.ColorDominatedNodes graph
| Graph.ColorNeighbours graph
| Graph.ColorZones graph
| Graph.ColorEdges graph
| // Update a single milestone and team status'
| // int list -> TeamStatus list * MilestoneStatus
| // -> TeamStatus list * MilestoneStatus
| let internal UpdateMilestone zonesscores
| (status,(milestone,compset)) =
| let teallist =
| List.zip zonesscores status
| |> List.mapi (fun n (a,b) -> (n,a,b))
| let f (team, score, ts) =
| match milestone with
| ZoneValues(_,_,_) -> score
| ProbedVertices(_,_,_) -> ts.Probes
| SurveyedEdges(_,_,_) -> ts.Surveys
| InspectedVehicles(_,_,_) -> ts.Inspections
| SuccessfulAttacks(_,_,_) -> tsAttacks
| SuccessfulParries(_,_,_) -> ts.Parries
match milestone with
| ZoneValues(num, fst, scn) | ProbedVertices(num, fst, scn) | SurveyedEdges(num, fst, scn) | InspectedVehicles(num, fst, scn) | SuccessfulAttacks(num, fst, scn) | SuccessfulParries(num, fst, scn) ->
let completed =
  teamlist
  |> List.choose
  (fun ((team, _, _)) as a) ->
  if f a >= num then Some team else None)
|> Set.ofList
let newcomp = completed - compset
let amount = if compset.IsEmpty then fst else scn
let inc (ts : TeamStatus list) n =
  ListReplace n (TS.IncreaseMoney amount) ts
let newstatus =
  newcomp
  |> Set.toList
  |> List.fold inc status
  (newstatus, (milestone, compset + completed))

// Update all milestones and team status'
// Graph -> MilestoneStatus list -> TeamStatus list
// Graph -> MilestoneStatus list * TeamStatus list
let internal UpdateMilestonesAndStatus ((nodes, _): Graph)
    milestones status =
    let getweight n node =
      if Node.ProbedBy (n,0) node
        then node.Weight
        else 1
    let stepscore n =
      nodes
      |> Array.toList
      |> List.filter (fun node -> node.DominatingTeam = Some n)
      |> List.map (getweight n)
      |> List.sum
    let zonescores =
      [for a in 0...(List.length status - 1) do yield stepscore a]
    let tempstatus = ref status
    let newmilestones =
      milestones
      |> List.map
      (fun ms ->
        let newts, newms =
          UpdateMilestone zonescores (!tempstatus, ms)
        tempstatus := newts
        newms)
    let stepscores =
      zonescores
      |> List.map2
      (fun (st : TeamStatus) score -> score + st.Money) !tempstatus
    let newstatus =
// Main algorithm.
// Calculate a single simulation step. Do this recursively
// until all steps have been calculated.
// Has side effects.
SimulationSettings
  let rec internal SimulationStep settings drawgraph drawpercept
viewp agents stepnum graph messages milestones status =
if stepnum > settings.Length
  then SendMsg "Done" // simulation endeth
  else SendMsg
      ( sprintf "Calculating step\nd of\nd" stepnum
      settings.Length)
      let percepts =
      PreparePercepts stepnum graph messages status milestones
      let (actions, msgs) =
      SendPercepts settings agents percepts
      let (apactions, restactions) =
      ListSplit AttackParryFilter actions
      let (status2, attacked) =
      ExecuteAttackParry settings status graph apactions
      let status3 =
      ExecuteOtherActions settings status2 graph attacked
      restactions
      ColorGraph graph
      let (newms, newstatus) =
      UpdateMilestonesAndStatus graph milestones status3
drawgraph graph newstatus stepnum
if viewp
  then List.iter (drawpercept stepnum) percepts
SimulationStep settings drawgraph drawpercept viewp agents
      (stepnum+1) graph msgs newms newstatus

// Add agents to a graph, defined by their roles. The agents
// will be positioned randomly on the graph. The agents are
// created according to the values in a given int list and bool
// list.
// Has side effects.
// Graph -> (int list * bool list * 'a list) list -> unit
let internal NodeAddAgents ((nodes,_) as g) roles =

// SimulationSettings
-> TeamStatus list -> int -> unit
-> (int -> Percept -> unit)
-> bool
-> Map<int*int,IAgent>
-> int
-> Graph
-> (Agent * Message list) list
-> MilestoneStatus list
-> TeamStatus list
-> unit
let rec internal SimulationStep settings drawgraph drawpercept
viewp agents stepnum graph messages milestones status =
let n = Array.length nodes
let r = new Random(DateTime.Now.Ticks |> int)
let add a =
    let randid = r.Next(n)
    Graph.AddAgent g randid a
let makerole agentid (statlist, aclist, namelist) =
    let addagents teamid _ =
        Agent.Create((teamid, agentid), statlist, aclist)
    List.mapi addagents namelist
|> List.choose (fun a -> a)
    |> List.iter add
    |> List.iteri makerole roles

// Create the AI objects according to the defined roles.
// Tells the created object the maximum allowed response time.
// int -> ('a list * 'b list * string list) list
// Map<int*int,IAgent>
let internal CreateAI settings nums roles =
    let makeagent agentid teamid name =
        ((teamid, agentid), MAS2011.Agents.GetAgent name)
    let makeagents agentid (a, slist) =
        List.mapi (makeagent agentid) slist
    let agents = roles |> List.mapi makeagents |> List.concat
    List.iteri (fun (a : IAgent) -> a.AddSettings settings) agents
    List.iteri (fun (a : IAgent) -> a.SetNumNodesEdgesSteps nums) agents
    agents |> Map.ofList

// Start a simulation with a given list of values, colors, roles and milestones, and a boolean value for whether or not
// the agents’ perceptions are to be drawn.
// Creates the simulation in a thread. Aborts the existing
// simulation, if any.
// Has side effects.
// int list
// -> Gdk.Color list * (int list * bool list * string list) list
// -> (string * int list) list -> bool -> unit
let StartSim s (gdkcolors, roles) m viewp =
    match s with
    | nt :: na :: sl :: fp :: rt :: reen :: recd :: nn :: w :: h :: minn :: maxn ::
        mine :: maxe :: acost :: pacost :: prcost :: scost :: icost :: bcost ::
        rcost :: fgcost :: upbatt :: upshie :: upsens :: upsabo ::[] ->
        thread.Abort()
    let settings =
        { FailChance = fp
        Length = sl
        MaxAgentResponse = rt
        RecoverNormal = reen
        RecoverDisabled = recd
        AttackCost = acost
ParryCost = pacost
ProbeCost = prcost
SurveyCost = scost
InspectCost = icost
BuyCost = bcost
RepairCost = rcost
FailedGotoCost = fgcost
UpgradeBatteryPrice = upbatt
UpgradeSensorPrice = upsens
UpgradeShieldPrice = upshie
UpgradeSabotageDevicePrice = upsabo

let graph = Graph.Create (nn, w−1,h−1,minn,maxn+1,mine,maxe+1)
let nums = (nn, Graph.NumEdges graph , sl)
NodeAddAgents graph roles
let colors = List.map GdkColor gdkcolors
let agents = CreateAI settings nums roles
let drawg = DrawGraph colors (w,h)
let drawp = DrawPercept colors (w,h)
let status = List.init (fun −> TS.StartStatus)
milestones = List.choose ParseMilestone m
ResetSteps()
drawg graph status 0
thread <-
   new Thread
   (new ThreadStart
      (fun −> SimulationStep settings drawg drawp viewp agents 1
        graph [] milestones status))
   thread . Start()
| −−> printfn "wrong length of s: %d" (List.length s)
B.22 SimulationSteps.fs

```fsharp
module MAS2011.Shared.SimulationSteps

(* Module that keeps tracks of the saved simulation steps. *)

open MAS2011.Shared.ShapePrimitives
open MAS2011.Shared.Generics
open System

// TickView: (Width * Height) * Shapes
type TickView = (int * int) * Shape list

let mutable internal SimSteps : Map<string,Map<int,TickView>> = Map.empty

let mutable internal mylock = new Object()

let mutable internal MsgRecievers = []

let mutable internal NameAddedRecievers = []

let mutable internal ResetNamesRecievers = []

let mutable internal curpath = "Simulations/Default.txt"

let mutable internal min = 1

let mutable internal max = 1

// Whether to automatically save the simulation steps, or not.
// Steps will be saved to curpath
// Warning: File size is very large, and will slow application!
let AutoSaveSteps = false

// Updates the path for autosave files.
// Has sideeffects.
// unit -> unit
let internal UpdatePath () =
  curpath <- "Simulations/" + DateTime.Now.ToString() + ".txt"

// Deletes the currently saved steps. Called when a simulation
// starts, in order to display only the steps from the new
// simulation.
// Has sideeffects.
// unit -> unit
let ResetSteps () =
  lock mylock
  (fun _ ->
    List.iter (fun f -> f()) ResetNamesRecievers
    UpdatePath()
    min <- 1
    max <- 1
    SimSteps <- Map.empty)

// Add simulation step n to the given name.
```
53 // Has side effects.
54 // string -> int -> TickView -> unit
55 let AddSimStepNoWrite name n s =
56   lock mylock
57   (fun _ ->
58     if SimSteps.ContainsKey name then
59       let newmap = SimSteps.[name].Add(n, s)
60       SimSteps <- SimSteps.Add(name, newmap)
61     else
62       List.iter (fun f -> f name) NameAddedRecievers
63       SimSteps <- SimSteps.Add(name, Map.empty.Add(n, s))
64     if n < min then min <- n
65     if n > max then max <- n
66     List.iter (fun f -> f (name, n, min, max)) UpdateRecievers)
67   )
68
69 // Value to display when a requested simulation step doesn’t exist.
70 let internal noret =
71   ((200,200),[Text((20.0,20.0),"Nothing...here",White)])
72
73 // Get a certain simulation step.
74 // string -> int -> TickView
75 let GetSimStep name n =
76   lock mylock
77   (fun _ ->
78     match Map.tryFind name SimSteps with
79       | Some a ->
80         match Map.tryFind n a with
81           | Some x -> x
82           | _ -> noret
83       | _ -> noret)
84   )
85
86 // Convert a simulation step to a string representation, in a list. Returns [name;number;width;height;shapes]
87 // string -> int * TickView -> string list
88 let StepToString name (n, tv : TickView) =
89   let colortostr c =
90     match c with
91       | Hex(r,g,b) ->
92         sprintf "H,%d,%d,%d" r g b
93       | Red -> "R"
94       | Blue -> "B"
95       | Black -> "Bl"
96       | Yellow -> "Y"
97       | Green -> "G"
98       | White -> "W"
99   let colortypetostr ct =
100      match ct with
101        | Stroked(color) ->
102          "S," + colortostr color
103        | Filled(color) ->
104          "F," + colortostr color
105        | StrokedFilled(color1, color2) ->
106          sprintf "SF,%s,%s"
let shapetostr s =
match s with
| Circle((x,y),rad,color) ->
  sprintf "C,%.1f,%.1f,%.1f,%s" x y rad (color to str color1) (color to str color2)
| Rectangle((x,y),(w,h),color) ->
  sprintf "R,%.1f,%.1f,%.1f,%.1f,%s" x y w h (color to typetostr color)
| Line((x1,y1),(x2,y2),color) ->
  sprintf "L,%.1f,%.1f,%.1f,%.1f,%s" x1 y1 x2 y2 (color to typetostr color)
| Text((x,y),s,color) ->
  sprintf "T,%.1f,%.1f,%s,%s,%s" x y s (color to str color)

let shapesstr shapes = list.map shapetostr
| StringGlue ";

let (w,h), shapes = tv [name; string n; string w; string h; shapesstr shapes]

// Convert all simulation steps to a string.
// unit -> string
let StepsToString () =
let s =
  SimSteps
|> Map.toList
|> List.map (fun (name,map) ->
    Map.toList map
    |> List.map (StepToString name))
|> List.concat
|> List.concat
StringGlue "@" s

// Load simulation steps from a string representation.
// Has side effects.
// string -> unit
let LoadStepsFromString s =
let rec split5 l =
match l with
| a::b::c::d::e::tl ->
  (a,b,c,d,e)::split5 tl
| _ -> []

let tickviews =
s |> StringToList '@'
|> split5

let strstocolor sl =
machine sl with
| "H"::rs::gs::bs::tl ->
  let r,g,b = byte rs,byte gs,byte bs
  (Hex(r,g,b),tl)
| "R"::tl -> (Red,tl)
| "B"::tl -> (Blue,tl)
| "Bl"::tl -> (Black,tl)
```fsharp
let strstocolortype sl =
match sl with
| "S" : t l -> let (c, _) = strstocolortype tl
               Stroked(c)
| "F" : t l -> let (c, _) = strstocolortype tl
               Filled(c)
| "SF" : t l -> let (c1, t12) = strstocolortype tl
               let (c2, _) = strstocolortype t12
               StrokedFilled(c1, c2)
| _ -> Filled(White)

let strtoshapes s =
let vals = s |> StringToList ', '
match vals with
| "C" : x : y : rad : colortype ->
  let ct = strstocolortype colortype
  Some (Circle((float x, float y), float rad, ct))
| "R" : x : y : w : h : colortype ->
  let ct = strstocolortype colortype
  Some (Rectangle((float x, float y), (float w, float h), ct))
| "L" : x1 : y1 : x2 : y2 : color ->
  let (c, _) = strstocolortype color
  Some (Line((float x1, float y1), (float x2, float y2), c))
| "T" : x : y : str : color ->
  let (c, _) = strstocolortype color
  Some (Text((float x, float y), str, c))
| _ -> None

let addtickview (name, ns, ws, hs, shapes) = try
  let (n,w,h) = (int ns, int ws, int hs)
  let ss = shapes
  | > StringToList ', '
  | > List.choose strtoshape
  AddSimStepNoWrite name n (w, h, ss)
  with
  | :? System.FormatException ->
    printfn "Unable to format number when adding tickview"
    tickviews | > List.iter addtickview

// Add a simulation step. Saves the simulation to a file if
// the boolean value AutoSaveSteps is true.
// Has sideeffects.
// string -> int -> TickView -> unit
let AddSimStep name n s =
  let isempty = SimSteps.IsEmpty
  AddSimStepNoWrite name n s
  if AutoSaveSteps
```
then
  if isempty
  then System.IO.File.WriteAllText(curpath, StepsToString())
  else
    let str = StepToString name (n, s) |> StringGlue "@
    System.IO.File.AppendAllText(curpath, "@"+str)

  // Add a function to be called when a message is sent.
  // The function must be of type string -> unit
  // Has sideeffects.
  // (string -> unit) -> unit
  let AddMsgReciever f =

    lock mylock
    (fun _ -> MsgRecievers <- f::MsgRecievers)

  // Send a message to all functions in MsgRecievers.
  // Has potential sideeffects.
  // string -> unit
  let SendMsg (s : string) =

    lock mylock
    (fun _ -> List.iter (fun f -> f s) MsgRecievers)

  // Add a function to be called when a simulation step is added.
  // The function must be of type
  // string * int * int * int -> unit
  // The format is: name * step number * min * max in steps
  // Has sideeffects.
  // (string * int * int * int -> unit) -> unit
  let AddUpdateReciever f =

    lock mylock
    (fun _ -> UpdateRecievers <- f::UpdateRecievers)

  // Add a function to be called when a new name is added to the
  // simulation steps. The function must be of type
  // string -> unit
  // Has sideeffects.
  // (string -> unit) -> unit
  let AddNameAddedReciever f =

    lock mylock
    (fun _ -> NameAddedRecievers <- f::NameAddedRecievers)

  // Add a function to be called when the names (simulation
  // steps) are reset. The function must be of type:
  // unit -> unit
  // Has sideeffects.
  // (unit -> unit) -> unit
  let AddResetNamesReciever f =

    lock mylock
    (fun _ -> ResetNamesRecievers <- f::ResetNamesRecievers)
module MAS2011.Monitor.SimulationView

(*
This module contains a class to display the simulation steps, 
along with some internal helper functions. 
*)

open Gtk
open MAS2011.Shared.SimulationSteps
open MAS2011.Shared.ShapePrimitives
open Mono
open System

// Get the RGB values for a Color type, range 0.0 to 1.0
// Color -> float * float * float
let internal GetColor c =
match c with
| Hex(rc, gc, bc) ->
  let r1 = (float rc) / 255.0
  let g1 = (float gc) / 255.0
  let b1 = (float bc) / 255.0
  (r1, g1, b1)
| Red -> (1.0, 0.0, 0.0)
| Blue -> (0.0, 0.0, 1.0)
| Black -> (0.0, 0.0, 0.0)
| Yellow -> (1.0, 1.0, 0.0)
| Green -> (0.0, 1.0, 0.0)
| White -> (1.0, 1.0, 1.0)

// Set the active color for a Cairo.Context.
// Has sideeffects.
// Cairo.Context -> Color -> unit
let internal SetColor (g : Cairo.Context) c =
let (rc, gc, bc) = GetColor c
g.Color <- new Cairo.Color(rc, gc, bc)

// Colors the active path in a Cairo.Context, according to
// a given ColorType.
// Has sideeffects.
// Cairo.Context -> ColorType -> unit
let internal ColorShape (g : Cairo.Context) ct =
match ct with
| Stroked(c) ->
  SetColor g c
  g.Stroke()
| Filled(c) ->
  SetColor g c
  g.Fill()
| StrokedFilled(c1,c2) ->
  SetColor g c2
  g.FillPreserve()
SetColor g c1
let internal kappa = 0.5522847498

let internal MakeShape (g : Cairo.Context) s =

match s with
| Circle ((x, y), r, ct) ->
  let k = kappa*r
  g.MoveTo(x, y-r)
  g.CurveTo(x+k, y-r, x+r, y-k, x+r, y)
  g.CurveTo(x-r, y+k, x-k, y+r, x-r, y)
  g.ClosePath()
  ColorShape g ct
| Rectangle ((x, y), (w, h), ct) ->
  g.Rectangle(x, y, w, h)
  ColorShape g ct
| Line ((x1, y1), (x2, y2), c) ->
  g.MoveTo(x1, y1)
  g.LineTo(x2, y2)
  SetColor g c
  g.Stroke()
| Text ((x, y), t, c) ->
  g.MoveTo(x-1.0, y-12.0) // Pango position correction
  SetColor g c
  let layout = Pango.CairoHelper.CreateLayout(g)
  layout.FontDescription <-
  Pango.FontDescription.FromString("Verdana,12")
  layout.SetText(t)
  Pango.CairoHelper.ShowLayout(g, layout)

// SimulationView class — inherits from Gtk.DrawingArea
// The class that displays the simulation steps.
type SimulationView () as this = class
  inherit DrawingArea()

  let mutable activetick = 1
  let mutable cursize = (20,20)
  let mutable curname = "Simulation"

  do
    this.RedrawOnAllocate <- true
    this.ExposeEvent.Add(this.OnExposed)
    let (x, y) = cursize
    this.SetSizeRequest(x, y)
    member this.SetActiveTick n =
      activetick <- n
member this.QueueDraw()

member this.SetName name =
curname ← name
this.QueueDraw()

member _.Name = curname

member this.OnExposed _ =
let ((w, h), shapes) = GetSimStep curname active tick
if (w, h) <> cursize
  cursize ← (w, h)
  this.SetSizeRequest(w, h)
else
  let g = Gdk.CairoHelper.Create(this.GdkWindow)
  let background =
    Rectangle((0.0, 0.0), (float w, float h), Filled(Black))
  List.iter (MakeShape g) (background::shapes)
  let g2 = g |> IDisposable
  g2.Dispose()
B.24 SimulationWindow.fs

```fs
module MAS2011.Monitor.SimulationWindow

(* Module containing the class SimulationWindow. This class is
  the main window in a simulation. *)

open Gtk
open MAS2011.Monitor.SimulationView
open MAS2011.Monitor.SimSettingsWindow
open MAS2011.Shared.SimulationSteps

// Class SimulationWindow
// Inherits from Gtk.Window
type SimulationWindow() as this = class
  inherit Window("MAS2011 competition simulator by Thor Helms, s061377, DTU")

  let contenttable = new Table(4u,2u,false)
  let timelabel = new Label("Step 1/2", WidthRequest=100)
  let mutable displayedtick = 1
  let mutable mintick = 1
  let mutable maxtick = 2
  let scale = new HScale(1.0, float maxtick, 1.0, WidthRequest=200, DrawValue=false)
  let simview = new SimulationView()
  let statusbar = new Statusbar()
  let msgid = 1u
  let mutable playing = false
  let namebox = new VBox()
  let mutable names : RadioButton list = []

  do
    this.DefaultSize <- new Gdk.Size(600,500)

    let menubar = new MenuBar()
    let filemenu = new Menu()
    let newsimitem = new MenuItem("New simulation...")
    let openitem = new MenuItem("Open simulation...")
    let saveitem = new MenuItem("Save simulation...")
    let exititem = new MenuItem("Exit")
    let agentnames = MAS2011.Agents.GetAllNames()

    newsimitem.Activated.Add(fun => new SimSettingsWindow(agentnames) |> ignore)
    exititem.Activated.Add(fun => Application.Quit())

    filemenu.Append(newsimitem)
    filemenu.Append(openitem)
    filemenu.Append(saveitem)
    filemenu.Append(exititem)
```
let fileitem = new MenuItem("File")
fileitem.Submenu <- filemenu
menubar.Append(fileitem)

let openfile =
    let fc =
        new FileChooserDialog
            ("Choose the simulation to open",
                this, FileChooserAction.Open,
                "Cancel", ResponseType.Cancel,
                "Open", ResponseType.Accept)
        if fc.Run() = int(ResponseType.Accept) then
            let contents = System.IO.File.ReadAllText(fc.Filename)
            MAS2011.Simulation.KillSim()
            ResetSteps()
            LoadStepsFromString contents
            fc.Destroy() |> ignore
            openitem.Activated.Add(openfile)

let savefile =
    let fc =
        new FileDialog
            ("Select a name/location for the simulation",
                this, FileChooserAction.Save,
                "Cancel", ResponseType.Cancel,
                "Save", ResponseType.Accept)
        if fc.Run() = int(ResponseType.Accept) then
            let contents = StepsToString()
            let file = fc.Filename
            System.IO.File.WriteAllText(file, contents)
            fc.Destroy() |> ignore
            saveitem.Activated.Add(savefile)

let timebox = new HBox()
let playpausebutton = new Button("Play/pause")
let prevbutton = new Button("Previous")
prevbutton.Clicked.Add(this.PrevTick)
let nextbutton = new Button("Next")
nextbutton.Clicked.Add(this.NextTick)
scale.ValueChanged.Add(this.ScaleTick)
timebox-PackStart(playpausebutton, false, false, 4u)
timebox-PackStart(prevbutton, false, false, 4u)
timebox-PackStart(nextbutton, false, false, 4u)
timebox-PackStart(timebox, scale, true, true, 4u)

statusbar.Push(msgid, "Starting") |> ignore

let fill = AttachOptions.Fill
let expand = AttachOptions.Expand
let fillExp = fill || expand

let scrolled = new ScrolledWindow()
```javascript
scrolled.AddWithViewport(simview)
let scrollnames = new ScrolledWindow(WidthRequest = 200)
scrollnames.AddWithViewport(namebox)
this.Add(contenttable)
contenttable.Attach(menubar, 0u, 2u, 0u, 1u, fillx, fill, 4u, 4u)
contenttable.Attach(scrollnames, 0u, 1u, 1u, 2u, fill, fill, 4u, 4u)
contenttable.Attach(scroll, 1u, 2u, 1u, 2u, fillex, fillex, 4u, 4u)
contenttable.Attach(timebox, 0u, 2u, 2u, 3u, fillex, fill, 4u, 4u)
contenttable.Attach(statusbar, 0u, 2u, 3u, 4u, fillex, fill, 4u, 4u)

let playpause = playing =< not playing
  if playing
    then GLib.Timeout.Add(333u, fun -> this.Play()) |> ignore
playpausebutton.Clicked.Add(playpause)

AddUpdateReciever this.StepUpdated
AddMsgReciever this.SendMsg
AddNameAddedReciever this.AddName
AddResetNamesReciever this.ResetNames

member this.Play() =
  if displayedtick = maxtick
    then playing =< false
  else this.DisplayTick(displayedtick + 1)
  playing

member this.NextTick =
  this.DisplayTick(displayedtick + 1)

member this.PrevTick =
  this.DisplayTick(displayedtick - 1)

member this.ScaleTick =
  this.DisplayTick(int scale.Value)

member this.DisplayTick n =
  if mintick <= n && n <= maxtick
    then
      if n <> displayedtick
        then
          scale.Value <- (float n)
          timelabel.Text <- (sprintf "Step%d/%d" n maxtick)
          displayedtick <- n
          simview.SetActiveTick n
        else ()
    else ()

// When a simulation step is added, this method is called.
// It will then update the window and possible display the newly added simulation step.
member this.StepUpdated (name, n, min, max) =
  let f() =
    maxtick <- max
```
mintick ← min
if min < max
    then scale.SetRange(double min, double max)
    else scale.SetRange(double min, min+1 |> double)
if name = simview.Name &&
    if displayedtick = n || displayedtick +1 = max)
    then this.DisplayTick n
else
    timelabel.Text <-
    (printf "Step%d/%d" displayedtick maxtick)
Gtk.Application.Invoke(fun _ _ -> f())

// Will display the string s in the statusbar.
member this.SendMsg s =
let f () =
    statusbar.Pop(msgid)
    statusbar.Push(msgid, s) |> ignore
    Gtk.Application.Invoke(fun _ _ -> f())

// When a new type of name is added to the simulation
// steps, this method is called with the newly added name.
member this.AddName s =
let makeradiob (s : string) =
    if names = []
    then new RadioButton(s)
    else
        let group = names |> List.head
        new RadioButton(group, s)
    let () =
    let r = makeradiob s
    let toggled _ =
        if r. Active
        then simview.SetName s
        r.Toggled.Add(toggled)
        namebox.PackStart(r)
        namebox.ShowAll()
        names ← r :: names
    Gtk.Application.Invoke(fun _ _ -> f())

// This method removes all simulation step names from
// the window.
member this.ResetNames () =
let f () =
    List.iter (fun w ->
        namebox.Remove(w)
        w.Destroy()) names
    names ← []
    Gtk.Application.Invoke(fun _ _ -> f())
end
module TS

(* Module for manipulation of TeamStatus *)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.Generics

// Increase the amount of probes done by 1
let IncProbes ts =
{ ts with Probes = ts.Probes + 1 }

// Increase the amount of surveys done by n
let IncSurveys n ts =
{ ts with Surveys = ts.Surveys + n }

// Increase the amount of inspections done by n
let IncInspections n ts =
{ ts with Inspections = ts.Inspections + n }

// Increase the amount of successful attacks by 1
let IncAttacks ts =
{ ts with Attacks = ts.Attacks + 1 }

// Increase the amount of successful parries by 1
let IncParries ts =
{ ts with Parries = ts.Parries + 1 }

// Convert a TeamStatus to an int list
let ToList (ts : TeamStatus) =

// Reduce the money of a TeamStatus by a given amount
let ReduceMoney amount (ts : TeamStatus) =
{ ts with Money = ts.Money - amount }

// Increase the score of a TeamStatus by a given amount
let IncreaseScore amount (ts : TeamStatus) =
{ ts with Score = ts.Score + amount }

// Increase the money of a TeamStatus by a given amount
let IncreaseMoney amount (ts : TeamStatus) =
{ ts with Money = ts.Money + amount }

// Increase the amount of successful parries by 1
let IncParries ts =
{ ts with Parries = ts.Parries + 1 }

// Convert a TeamStatus to an int list
let ToList (ts : TeamStatus) =
let IncreaseMoney amount (ts : TeamStatus) =
  { ts with Money = ts.Money + amount }

// TeamStatus starts with all values as 0.
// TeamStatus
let StartStatus =
  { Score = 0
    Money = 0
    Probes = 0
    Surveys = 0
    Inspections = 0
    Attacks = 0
    Parries = 0 }