PDT-Jack Basic Agent System for the Gold Mining Game
User Manual

Sebastian Sardina
Version 4

July 13, 2012

This document briefly describes the Jack *core* agent framework for the gold mining game. This is a basic agent framework to serve as an initial system to build on more sophisticated systems.

On of the most important aspect of this framework is its support infrastructure implementing the "glue" between the player agents and the game server. Therefore, the framework already encapsulates all the connectivity code required to communicate with the game server. Besides this, the system also includes some very basic (random) agent behaviour too, as some useful code that could be used as examples for further extensions.

The design of the system was done using the PDT tool (as an ECLIPSe plugin), but substantial Jack coding was also done on the Jack sources to provide a basic behavior. When re-generating code, PDT will respect the added code.

Disclaimer

The current document should be treated as "informal" notes aimed to quickly help the user to understand the system and use it. Nevertheless, this document is far from complete and checked and has suffered many changes since its original version. Therefore, the reader should be aware that there may be many errors and inconsistencies.

1 File System Structure

The system file and directory structure is as follows:

`README.TXT` Short instructions on how to run the application.

`FILES.TXT` Description of file/directory contents in the application.

`design.pd` The PDT file containing the agent system design.

`run-team.sh` Shell script to *run* the agent system application.

`Makefile` Makefile for clean (`make clean`), compile (`make compile`), and pack the application into a JAR file (`make jar`). To do everything (i.e., clean-compile-pack) one can use `make all`.

`compile.sh` Shell script to *compile* the Jack application. This script will take the Jack source code and compile it using both Java compiler and Jack kernel. Binaries will be placed in `bin/` and a whole JAR `jackagt.jar` file will be placed in `lib/`.

`build-jack.xml` Ant script to *compile* the Jack application within ECLIPSe.

`lib/` Location where all the required libraries, as JAR files, should be placed, including `jack.jar` containing the Jack compiler and runtime environment and `gui.jar` containing the GUI interface. These libraries are used both at development time (when designing and compiling the application) as well as execution time (when running the application itself).

`src/` The location for all sources for the application, including the Jack sources. In particular, PDT will generate Jack code in subdirectory `src/rmit/ai/clima/jackagt/`.

`bin/` The location where the Jack *binary* application will be placed when the Jack source is compiled. It will also contain the corresponding `.java` files generating by Jack compiler.

2 Mini How-To

To set up your system you should first follow these initial steps:

1. Make sure JDK 1.6 is installed in the system. JRE is not sufficient as we will need to *compile* the JAVA code generated by Jack. JDK can be easily obtained from Sun web-page at `http://java.sun.com/`. Both Java and Javacc should be available in your system and in the path.
2. Make sure you have lastest ECLIPSe installed in your system.
3. Install the PDT plugin for ECLIPSe. This involves installing two plugins files, one for the PDT design within ECLIPSe and one for code generation. If you are using Linux, this just involves copying both files to your `~/.eclipse` plugin subdir. For Windows, it will depend whether you have administrative rights (e.g., your own laptop) or not (e.g., RMIT lab). Please see the FAQ section in the AOPD course web page for instructions.
4. Get the basic Jack agent system and install it as an ECLIPSe regular project (do not use a Java project, as this is Jack code).
5. Make sure JAR files `jack.jar`, `climacomms.jar`, and `gui.jar`, into the `lib/` sub-directory of your agent system.
6. Get the gold mining game server package and unpack it. Read its `readme.txt` file and configure the ports to be used.

Once you set-up both the server and the agent systems, you will generally follow these steps to update and run the agent system:

1. Open PDT file `design.pd` from inside ECLIPSe.
2. Using PDT, implement the desired *design* changes/extensions required for your system (e.g., add a new entity or link two entities).
3. Instruct PDT to generate all Jack source code files in source directory `src/rmit/ai/clima/jackagt`. PDT will generate the corresponding `.agent`, `.cap`, `.bel`, `.event`, `.plan` files in corresponding sub-directories. Remember to set-up both the source directory `src/` and the package `rmit.ai.clima.jackagt` correctly; you only need to do this the firs time you generate code, PDT will remember both options.
4. Using the editor of your choice (e.g., ECLIPSe), implement all the required code modifications in the Jack source files (e.g., programming code in the body of a plan, or programming the available queries in a new beliefset or the posting methods for an new event). Do all your programming *outside* the PDT protected block, as this will be overwritten by PDT the next time its code generation facility is used. PDT, however, will respect all the programming changes you do outside the protected area.
5. Next you must compile the Jack source files to produce the actual executable agent system. This can be done by just running `make all` in the project main directory, or running script `compile.sh`.
6. The (binary) Jack agent system should now be in directory `bin/` and the whole JAR file `jackagt.jar` should now be in `lib/` and ready to be run!
7. Finally, adjust the `run-team.sh` script to match the server location and the team *usernames* and *passwords*, and run the script to connect to the game server. (If you are running the game server yourself, remember to run the game server *before* the agent system or otherwise the agents will fail to connect.)

For more details on how to *run* the system and options to pass to the `run-team.sh` script, read file `readme.txt`.

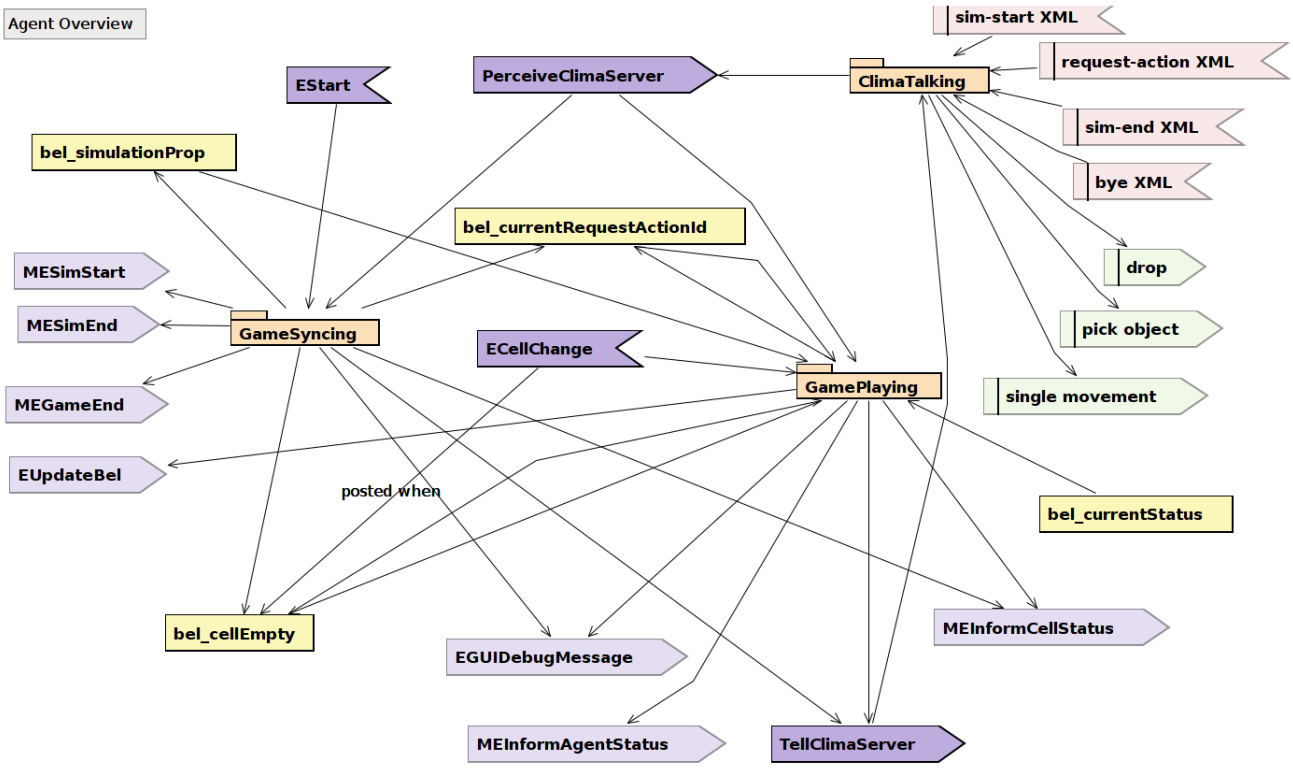

Figure 1: The overall design of a `Player` agent.

3 System Components

The system is composed of three agent types: `Player`, `Coordinator`, and `GUIAgent`. The last agent is responsible of showing the simulation in a graphical window online and to re-play saved simulation afterwards. Here, we shall focus only on the first two agents which are the ones playing the actual game.

A `Player` agent is an agent acting in the actual game simulator; whereas a `Coordinator` is an extra agent and does not interact with the game server. Typically, the application will run several player agents (6, in particular) and one coordinator agent.

Within each cycle of a simulation game, the players merely move randomly in the grid, update their current beliefs, and send some information to the coordinator. Meanwhile, the coordinator keeps track of the information received from the players (at this point, very little). Below, we first go over the components within a player agent and then review the ones for the coordinator agent.

3.1 Player components

Figure 1 depicts the overall design of a player agent.

Capabilities

For modularity, a `Player` agent is organised into four capabilities:

`ClimaTalking` All the functionality required for communication with the game server. This is an external capability (that is, it is not specified within the system); see Section 5.

`GameSyncing` All the functionality required for starting and ending game simulations and tournaments.

`GamePlaying` All the functionality required for actually playing a simulation game.

`InfoReporting` All the functionality for displaying information (e.g., in the console or the GUI window).

`ActionDecision` All the functionality to decide what action is to be done next on the game server.

Beliefs

- `CellEmpty`: keep track of which cells are known to have obstacles.
- `SimulationProp`: stores some properties of the current simulation being played.
- `LastActionSent`: stores the ID number of the last action sent to the game server (provided by the `ClimaTalking` capability).
- `CurrentRequestActionId`: stores the ID of the last `request-action` message received from the game server; the beliefset is also used to instructs an agent to do something (by posting an event `EAct`).

Events

- `EStart`: signal that the agent has just been created.
- `MESimStart` and `MESimEnd`: message events used to inform other agents (e.g., the coordinator) of the start and end of simulations, respectively.
- `MEGameEnd`: message event used to inform other agents (e.g., the coordinator) of the end of the whole tournament.
- `ECellChange`: signals the fac that the beliefs of a cell has changed (e.g., it is now known to be empty of obstacles).
- `EAct`: instructs the need to act in the game server (i.e., do an action).
- `EShowBeliefs`: instructs the agent to report on some or all of what the agent currently believes.
- `EExecuteCLIMAaction`: instructs the execution of a particular domain action (e.g., up or pick).
- `TellClimaServer` and `PerceiveClimaServer`: send and receive messages to and from the game server, respectively. Both are provided by the `ClimaTalking` capability, see Section 5.
- `EUpdateBel`: used to trigger the update the beliefs about cells around a particular location.
- `MEInformAgentStatus`: message used to communicate the state of the agent (e.g., its current position to the `GUIAgent`).
- `MEInformCellStatus`: message used to communicate the state of a cell (e.g., inform the `GUIAgent` that a cell contains an obstacle).

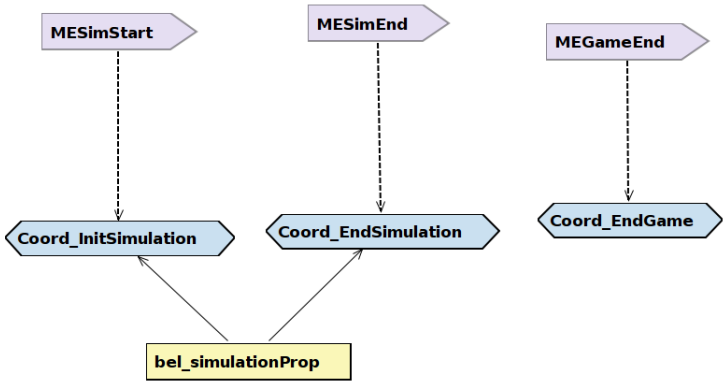

Figure 2: The overall design of a `Coordinator` agent.

Plans

- `AuthenticateToServer`: sends the login information to the GAME server by appealing to the `ClimaTalking` capability.
- `StartSimulation` and `FinishSimulation`: perform tasks associated to the start and end of simulations.
- `FinishGame`: perform tasks associated to the end of a complete tournament game.
- `HandlePercept`: absorbs all the information encoded in a `request-action` message from the game server.
- `MoveRandomly`: perform some random movement on the grid.
- `SendActionAndWait`: sends an action to the game server, by appealing to the `ClimaTalking` capability.
- `BeliefReporting` and `ConsoleBeliefReporting`: two plans to report on some beliefs of the agent. The first one is intended to report on the GUI; the second is intended to report information on the console via the agent interface `consoleIface`.
- `UpdateCellsAround`: updates the beliefs about a group of cells around a particular location.
- `ReportCellChangeToGUI`: reports to the `GUIAgent` information about a cell when this has changed.

3.2 Coordinator components

Figure 2 depicts the overall design of a player agent.

Beliefs

- `SimulationProp`: stores some properties of the current simulation being played.

Events

The following events are all used to implement the communication between the players and the coordinator. That is, the following events are sent by the players to the coordinator, who then handles them by means of its own plans (see Figure 3 for a graphical representation).

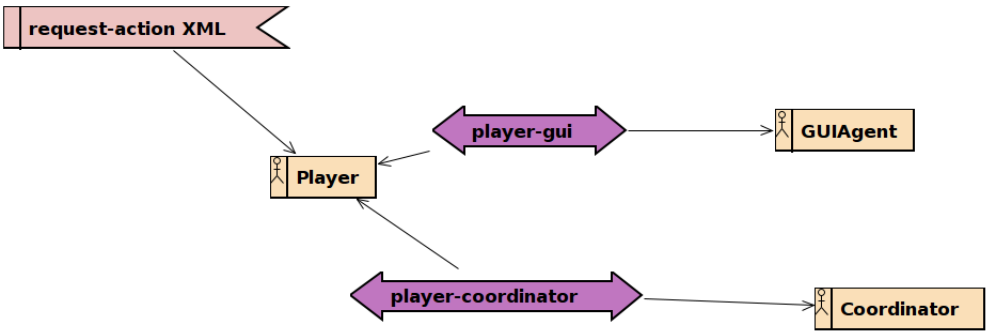

Figure 3: An overview of the agent communication.

- `MESimStart`, `MESimEnd`, and `MEGameEnd`: as described above to signal the start and end of simulations/games.

Plans

The following plans are all used to handle events that were sent by the players:

- `Coord_InitSimulation` and `Coord_EndSimulation`: perform tasks associated to the start and end of simulations (e.g., initializing or resetting beliefset `SimulationProp`.
- `Coord_EndGame`: performs tasks associated to the end of a complete tournament game.

4 System Operation Overview

System initialisation

Initially, the agent connects to the game simulator and authenticates itself using its login and password:

1. The agent's `ClimaTalking` capability opens a socket connection to the game server.
2. As soon as the player is created, a `EStart` event is automatically posted.
3. Event `EStart` is handled by plan `AuthenticateToServer`, which in turn posts a `TellClimaServer` event with an `AuthRequest` object as data containing the authentication information obtained from the command line (i.e., login and password).
4. Finally, the `ClimaTalking` capability handles the event `TellClimaServer` by sending corresponding XML authentication message to the game server with the player details.

At this point, the agent is connected and registered to the game server. If for any reason the connection drops, the `ClimaTalking` capability will automatically retry to connect and authenticate. Thus the `ClimaTalking` capability maintains a robust permanent connection with the game server.

Start of a new simulation

Once the agent has successfully authenticates to the game server, it just waits for a simulation to start. The start of a simulation happens as follows:

1. The game server sends an XML `sim-start` message to the agent.

2. The ClimaTalking capability reads such XML message and posts a PerceiveClimaServer event, whose data contains a SimStart object.
3. The pending PerceiveClimaServer event is then handled by the player's StartSimulation plan, which in turn sends an MESimStart event to the coordinator agent.
4. The coordinator handles the MESimStart event using its plan Coord_InitSimulation.

Playing a simulation game

Once a simulation has started, it follows the following pattern:

1. Agent receives a request-action as a PerceiveClimaServer event that is posted automatically by the agent ClimaTalking capability (see Section 5).
2. Plan HandlePercept then handles such event by doing the following:
   (a) Explicitly update part of the agent beliefs (e.g., number of gold being carried, current position, simulation step number, etc.)
   (b) Post, synchronously, an event EUpdateBel so that the agent can further update its beliefs about the surrounding cells.
   (c) Post, asynchronously, an event EShowBeliefs so that the agent reports on its current state of affairs.
   (d) Updates beliefset CurrentRequestActionId with the new id/step number of the simulation.
3. The agent handles event EUpdateBel using its plan UpdateCellArround, which shall update the beliefs about the cell around the agent current location.
4. As a result of the update on beliefset CurrentRequestActionId, an (automatic) event EAct is opsted to signal the need to select and perform some action on the game server, i.e., to act.
5. The agent will handle event EShowBeliefs using plans ConsoleBeliefPrinting and BeliefPrinting. For reporting anything on the console, one should use the first one.
6. The agent will handle event EAct using plan MoveRandomly, which will instruct to execute one of the four movement directions (up, down, left, right) by posting event EExecuteCLIMAaction.
7. Event EExecuteCLIMAaction is handled by plan SendActionAndWait which sends the corresponding action to the game server via the posting of interface event TellClimaServer offered by ClimaTalking capability—there are plans in such capability to do the corresponding XML exchange with the game server.

Figure 4 depicts the interaction between the different parts of the system that are involved in an agent playing a simulation game.

End of a simulation

The above process continues throughout the simulation until this is over, in which case the following occurs:

1. The game server sends an XML sim-end message to the agent.
2. The agent's ClimaTalking capability reads the XML message and posts a corresponding PerceiveClimaServer event, whose data contains an MESimEnd.
3. The pending PerceiveClimaServer event is then hanlded by plan FinishSimulation, which in turn sends an MESimEnd event to the coordinator agent
4. The coordinator would handle the MESimEnd event by means of plan Coord_EndSimulation.

Once the simulation is over, the players came back to their "waiting" state for a new simulation or the end of the whole tournament.

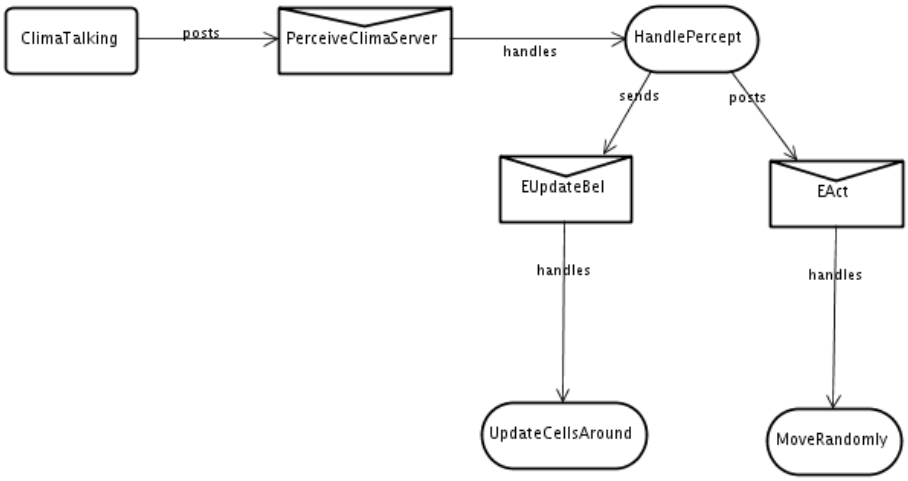

Figure 4: An overview of the player interaction within a simulation. Observe that event `EAct` is now posted automatically by beliefset `CurrentRequestActionId`.

End of tournament

Eventually, all the scheduled simulations were carried on by the server and the complete tournament is over. In that case, the following occurs:

1. The game server sends an XML `bye` message to the player agent.
2. The agent's `ClimaTalking` capability reads the XML message and posts a corresponding `PerceiveClimaServer` event, whose data contains a `Bye` object. Moreover, the capability disconnects completely from the game server by closing the corresponding socket.
3. The pending `PerceiveClimaServer` event is then hanlded by plan `FinishGame`. The plan basically may do some routine tasks and finally terminate the agent by calling the agent base method `finish()`. Before terminating, it informs the coordinator of the end of the tournament by sending it an `MEGameEnd` event.
4. The coordinator then handles the `MEGameEnd` event by means of plan `Coord_EndGame`. This plan will also terminate the coordinator agent, maybe after doing some routine tasks.

5 Connectivity Infrastructure: The `Climma-Talking` Capability

The support infrastructure containing the connectivity code is wrapped up into a single `ClimaTalking` *capability* (`rmit.ai.clima.iface.ClimaTalking`). An overview of this capability is depicted in Figure 5. This capability is already included in the agent as an *imported external* entity, and thus it *cannot* be modified within the agent. The binary of the capability is included in JAR file `lib/climacomms.jar`. To successfully compile the Jack agent system, it is necessary to have such JAR file within the CLASSPATH.

Before going into the details of the capability, it is worth noting that the agent programmer should be able to develop a full agent system for the game *without knowing almost any details about the capability*. Understanding the two events `TellClimaServer` and `PerceiveClimaServer`, together with the classes used to represent each XML message (see below), should be enough to successfully develop an agent. Figure 6 shows how the player interacts with the game server by either authenticating itself at connection time or sending domain actions during simulation games.

The `ClimaTalking` capability has the following four dynamic elements (two event types and two data):

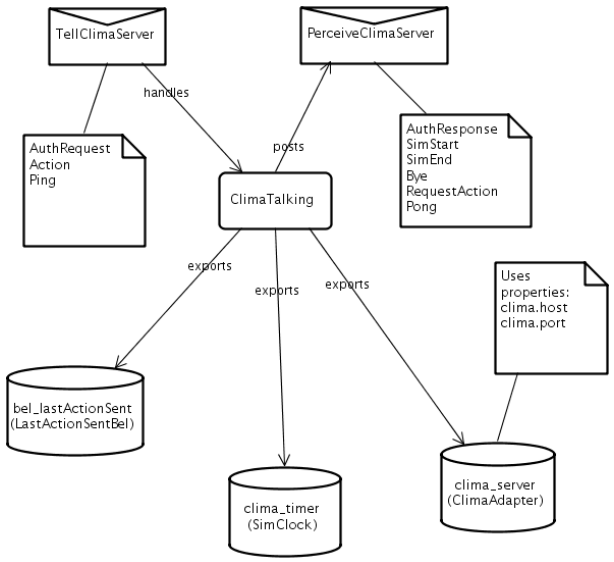

Figure 5: A usage view of the `ClimaTalking` capability.

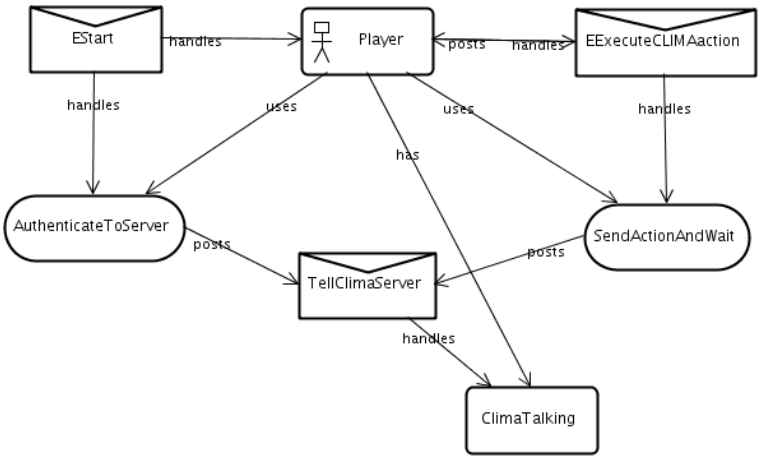

Figure 6: An overview of the player interaction with the game server.

1. The `TellClimaServer` event type (`rmit.ai.clima.iface.TellClimaServer`), to issue messages to the game server.

2. The `PerceiveClimaServer` event type (`rmit.ai.clima.iface.PerceiveClimaServer`), to carry received messages from the game server.

3. The `ClimaAdapter clima_server` data (`rmit.ai.clima.iface.ClimaAdapter`), containing the low-level settings and functions implementing the game server interactions.

4. The `SimClock clima_timer` data (`aos.jack.jak.util.timer.SimClock`), to keep the simulation time provided by the game server.

5. The `LastActionSentBel bel_lastActionSent_dat` belief data to keep the last "id" number of the action sent to the game server (`rmit.ai.clima.iface.LastActionSentBel`).

The game messages (e.g., `SIM-START`, `REQUEST-ACTION`, `ACTION`, etc.) are all modeled as JACOB[1] objects, and the above two events `TellClimaServer` and `PerceiveClimaServer` carry these as data. This means that each particular XML messages exchanged between the agent and the game server will be eventually seen by your agents as a mere JAVA object. The (JACOB) classes of those objects, which are already fully defined, correspond one-to-one with the XML messages as documented in file `c7c-protocol.txt`, except for trivial name renaming (e.g. `RequestAction` instead of `REQUEST-ACTION`).

Note that the ClimaTalking capability makes use of Java properties for connection information. These are:

- **clima.host** The host name of the Agent Contest Server. (e.g., `yallara.cs.rmit.edu.au`)
- **clima.port** The port name for client connections.

(We will advertise a range of port numbers you can use when running and testing).

The agent needs to provide the following for authorisation:

clima.agent.name.username The authorization user name for the agent named **name**.

clima.agent.name.password The authorization password for the agent named **name**.

For initial experimentation you can use `participant1, participant2, ..., participant6` and `1, 2, ... , 4, 5, 6` as agent names and passwords.[2] For the competition, we shall assign agent and password names.

The `ClimaTalking` capability is at the core of the connectivity infrastructure. It deals with the sub-functions of:

1. keeping a TCP connection with the game server, including the authorisation hand-shaking with the game server;

2. parsing of incoming messages, translating them into their corresponding JACOB objects, and then posting `PerceiveClimaServer` events carrying the data;

3. handling `TellClimaServer` events by encoding their JACOB messages into their corresponding XML messages, and sending them to the game server; and

4. maintaining the `clima_timer` time value, by peeping at the time stamps of receive messages.

5. maintaining the `bel_lastActionSent_dat` id value, by storing the id number of the last action sent to the game server (when the agent posted a `TellClimaServer` event).

The `ClimaTalking` capability is already included in the initial agent so all connection and all low-level communication with the game server is already included with it.

[1] Information on JACOB can be found in the Jack manual.

[2] If you want to run 2 teams, you can also use the dummy `botagent` team that comes with the game server.

5.1 XML Messages as JACOB Objects

The `ClimaTalking` capability of each agent is in charge of doing the low-level exchange of XML messages throughout the tournament. Outside this capability, these messages are not seen as XML messages anymore, but as JACOB objects. There is one class defined for each possible XML messages in the domain. The main events `TellClimaServer` and `PerceiveClimaServer` are designed to carry on objects of these classes as their data.

A `TellClimaServer` event can carry, as its data, an object of any of these classes: `AuthRequest`, `Action`, or `Ping`. Similarly, a `PerceiveClimaServer` event can carry as data an object of any of the following classes: `AuthResponse`, `SimStart`, `SimEnd`, `Bye`, `RequestAction`, and `Pong`.

Probably the most sophisticated and important class is `RequestAction`, corresponding to a `request-action` XML message sent by the game server at every step of the simulation. Apart from some basic information (e.g., the step number, the agent's current position, etc.) an object of that class includes an attribute that is an array of an auxiliary `Cell` class. Each object of class `Cell` would then contain the particular information of a cell.

All these classes are defined within the `ClimaTalking` capability in the package `rmit.ai.clima.comms`. So, for example, if an agent plan needs to access some `SimStart` object (e.g., to extract the position of the depot), it needs to import `rmit.ai.clima.comms.RequestAction`.

Finally, we point out that one can easily extract the form of each of these classes (i.e., their attributes and methods) by inspecting file `rmit/ai/clima/comms/msg.api`. As it is easy to observe, the attributes of each class correspond one-to-one to the data contained in the corresponding XML message, as dictated by the protocol.

6 The `gui.jar` Package

The `lib/gui.jar` file implements the graphical interface for the game. Besides providing such functionalities, used by agent `GUIAgent`, it the following useful libraries/packages:

- The actual GUI Window Java-based interface (class `GuiInterface`), within package `rmit.ai.clima.gui`.
- Java interfaces used in the application, in package `rmit.ai.clima.interfaces`.
- Class `GridPoint` in package `rmit.ai.clima.gui.grid` to manipulate points/cells `(x,y)` in the grid. It contains several utilities, some static, to manipulate cells in a convenient way (e.g., getting a cell as a String or calculating a cell adjacent to another one). Please refer to `GridPoint.java` to see what the class provides.
- Class `GameGraphic` in package `rmit.ai.clima.gui.graphic` providing different graphical shapes (e.g., circles, rectangles, etc.) to represent different objects (e.g., gold, agents, obstacles, etc.). The agent system defines static graphical objects that will be used to represent different entities in the game in class `GameGraphics` within package `rmit.ai.clima.gui.grid`. This class also provides a convenient mapping between such graphical objects and string representations (e.g., "gold").

Acknowledgments

We acknowledge the help of Ralph Ronnquist in developing the initial version of the `ClimaTalking` capability and providing support on many issues. Phil Donald developed a first prototype of the GUI interface. The current version of the GUI interface is due to Abhijeet Anand. We also thank Nitin Yadav for providing substantial support across the whole project.