Laboratory: Programming Multi-Agent Systems with ASTRA

This lab on ASTRA programming is focused on agent communication using FIPA-ACL. That is, in this exercise we will develop a multi-agent system (system with more than one agent) in which the agents communicate with one another via an agent communication language.

The specific problem we will tackle in this lab is to develop agents that can play the game Tic-Tac-Toe. In this lab, we will focus mainly on the representation of the game (remember the domain modelling activity) and on implementing a custom protocol to models the in-game interaction between the agents.

As in the first lab, the agents will maintain a mental model of the game board rather than using an explicit shared resource. Also, while there are many valid ways of implementing the agent programs in this lab, I would ask you to again try and base your program on the practical reasoning model we covered in previous labs.

Marking Scheme

Complete 1 part	E
Complete 2 parts	D
Complete 3 parts	C
Complete 4 parts	B
Complete ALL parts	A

Submission Instructions

Create a new maven project for each part. The name of the project folder should be Part<num>. For example, the answer to Part 1 should be in a folder Part1. Each part folder should be in a folder called Lab3. At the end, please ZIP the Lab3 folder up and submit via Brightspace.

The Problem: Tic-Tac-Toe

Tic-Tac-Toe is a long established problem in Artificial Intelligence, the value of which is highlighted in Wargames (1983) where it is used to stop Global Thermonuclear War.

You can watch the relevant clip here: https://www.youtube.com/watch?v=NHWjlCaIrQo

While we don't have these lofty goals, Tic-Tac-Toe proves to be an elegant and simple problem with which to explore the basics of agent communication. The reason: Tic-Tac-Toe is a two-player game. To implement it, we need two agents…

Tic-Tac-Toe was used as an example in both the Domain Modelling. Below is a sample AgentSpeak(L) program that implements a single player of Tic-Tac-Toe based on that model.

```
// Inferences
free(L) :- location(L) & ~played(T, L)
winner(T) :- line(L1, L2, L3) & played(T, L1) &
               played(T, L2) & played(T, L3)
loser(T) :- player(T) & winner(T2) & T ~= T2
drawn() :- ~free(L) & ~winner(T2)
opponent(T, O) :- player(O) & O ~= T;

// Locations
location(1), location(2), location(3).
location(4), location(5), location(6).
location(7), location(8), location(9).

// Winning lines
line(1, 2, 3).
line(1, 5, 9).
line(1, 4, 7).
line(2, 5, 8).
line(3, 6, 9).
line(3, 5, 7).
line(4, 5, 6).
line(7, 8, 9).

// Players
player("X"), player("O").

// Making a move
+turn(T) : player(T) & opponent(T, T2) <-
     !move(); ?move(L); -move(L);
     !played(T, L);
     !turn(T2).

+!played(T, L) : free(L) <-
     +played(T, L).

+!turn(T2) : turn(T) & winner(W) <-
     -turn(T);
     .print("Winner: "  + W).

+!turn(T2) : turn(T) & drawn() <-
     -turn(T);
     .print("Drawn!").

+!turn(T2) : turn(T) <-
     -turn(T);
     +turn(T2).

+played(P, L) {
     .print("Player " + P + " played at location: " + L).
```

```
// Player move selection strategy…
+!move() : free(1) <- +move(1).
+!move() : free(2) <- +move(2).
+!move() : free(3) <- +move(3).
+!move() : free(4) <- +move(4).
+!move() : free(5) <- +move(5).
+!move() : free(6) <- +move(6).
+!move() : free(7) <- +move(7).
+!move() : free(8) <- +move(8).
+!move() : free(9) <- +move(9).
```

Part 1: Translation to ASTRA

The first task to complete is to translate the program from AgentSpeak(L) to ASTRA. The code should be placed in a file called `Player.astra`. Once you have done this, you should test the logic part of your code.

NOTE: There are a couple of changes to make when translating AgentSpeak(L) to ASTRA:

1. Because ASTRA allows unbound variables to be passed to subgoals, the first line of the `+turn(…)` rule can be simplified from:

   ```
   !move(); ?move(L); -move(L);
   ```

   to:

   ```
   !move(int L);
   ```

   This requires the modification of the `Part1` rules to look like this:

   ```
   rule +!move(int L) : free(1) { L=1; }
   ```

2. The dropping of the old `turn(…)` belief and the adoption of the new `turn(…)` belief can be combined into a single statement. So:

   ```
   -turn(P);
   +turn(O);
   ```

   becomes:

   ```
   -+turn(O);
   ```

To test your code, you should create a `Main.astra` agent program that extends the `Player.astra` agent program. The `Main` agent program should include a `+!main(..)` rule that tests specific scenarios. For example, to test if any tokens have been played, you can use:

```
rule +!main(list args) {
   if (~played(string T, int L))
      C.println("Game has not Started");
}
```

You should write an `if` statement for each of the following cases:

- Game won by player <X>
- Game lost by player <Y>
- Game is drawn
- The first free location is: <L>
- Game has not started

Now run the program. “Game has not started” should be displayed in the console.

Next, write out a game configuration in which player O is the winner. To do this, use the `initial` statement to specify a number of `played(…)` beliefs. When you run the program this time, you should see information about the winner, looser, and possibly the first free location being displayed.

Now, comment out the set of `initial` statements adding a comment immediately above them stating “Player O won configuration”. Make a new game configuration in which the game is drawn. Add a comment above the configuration stating “Game drawn configuration”.

Finally, test that the agents game logic works by adding the initial belief `turn(“X”)` to the `Main` agent program. What happens if you change the “X” to an “O”.

Part 2: Creating a Multi-Agent System

The second task is to create a multi-agent system consisting of 3 agents: the main agent (which we run to start the program), and 2 Player agents who will play Tic-Tac-Toe against one another.

Creating agents in ASTRA is done via the System API (module). To create an agent simply create an instance of the module:

```
module System system;
```

then you can create agents using the following statement:

```
system.createAgent("name", "ASTRAClass");
```

You can give the agent you have created a `!main(…)` goal with by writing:

```
system.setMainGoal("name", [list, of, arguments]);
```

This results in the specified agent having a goal of the form:

```
!main([list, of ,arguments])
```

To complete this task, modify the `Main.astra` program so that it no longer extends the `Player.astra` agent program. Update the `+!main()` rule to use the above statements to create two agents: `player1` and `player2`.

Add a `+!main()` rule to the `Player` agent program that allows you to initialize a player to have either an "X" or an "O" (represented by the belief `token("X")` or `token("O")` respectively). A second value should indicate whether or not the agent should believe it is their turn. Specifically set `player1` to "O"s and make it their turn and set `player2` to "X"s but that it **not** be their turn.

HINT: The arguments can be heterogeneous (e.g. `["O", true]` vs `["X", false]`) and you can match a specific pattern of arguments in the main (e.g. rule `+!main([string token, boolean turn]`).

What happens when you run this? Switch the initial `!main(…)` goals so that `player2` believes it is their turn as well

Part 3: The Game Protocol

The third task is to implement the protocol that will be used to implement turn taking in the game. In this lab, we will view the game as a collaborative activity in which agents maintain their own model of the game and share the moves that they make.

To achieve this, we will need to make some changes to the way that the `Player` works:

1. Identifying the opponent by name. This can be done by modifying the `opponent(…)` formula to have only one string parameter (the name of the opponent). The belief can be adopted via the `+!main(…)` rule by passing the name of the opponent as another parameter: `[<token>,<opponent>,<turn>]`. You should also remove the opponent inference rule.
2. Removal of the internal `!turn()` goal and its associated rules. Turn taking will be managed through interaction between the agents (see 4). The dropping of the turn belief still needs to be part of the behaviour – this can be done as the last step of the `+turn(…)` rule.
3. Movement of the check on winning / drawing to the `+turn(…)` rule (belief not goal). Basically, we do the same thing we did with the `!turn()` rules – overload the event so that it checks whether the game is won or drawn before the agent tries to make a move.

Now, we can start to implement the interaction between the players. This will take the form of a simple protocol that consists of a series of **FIPA Inform messages** that are passed between the players. Basically, when a player records its move via the achievement of the `!played(…)` goal, it also informs its opponent of the move it made. Receiving such a message causes the player to update its local state and initiates the turn taking. This will require modification of the context for the `+!played(…)` rule to identify the opponent's name.

To handle the receipt of the message, you need to write another rule, with event `@message(inform, string sender, played(string T, string L))`. This rule should update the game state, and adopt the turn belief so that the player can make their next move.

Finish by modifying the Main agent program so that it first creates a main goal for `player2` with parameters `["X", "player1", false]` and then a main goal for `player1` with parameters `["O", "player2", true]`. The inverted order is important her so that `player2` is set up to participate before `player1` starts the game.

Part 4: Strategizing

The next task is to introduce players with different strategies. To achieve this, you need to use the inheritance mechanism of ASTRA. Start by creating a new program called `LinearPlayer.astra`. This program should look as follows:

```
agent LinearPlayer extends Player {
}
```

Copy all the strategy rules (the `+!move(int L)` rules) into the new program and delete them from `Player.astra`. Modify the `Main` agent to use `LinearPlayer` instead of `Player`. The game outcome should be the same.

Once you have this working, create another program called `Opponent.astra` and implement an alternative strategy that can beat `LinearPlayer.astra`. For this first attempt, try to achieve this by simply re-arranging the order of the `+!move(…)` rules.

Modify your `+!main(…)` rule to make `player2` be the `Opponent` agent.

Part 5: Jazzing it up

The final task is to jazz things up a little more and develop your own strategy (that is more than simply rearranging the order of the `+!move(…)` rules. Create a file called `Better.astra`. Describe your strategy in a set of comments added to the file. Implement the strategy and check that it can beat both the `LinearPlayer` & `Opponent` agent programs.

To get an A+ grade, you need to do something impressive and unique here in terms of the strategy.