Assignment 1

COSC1204/2048 - AOPD: Agent-Oriented Programming & Design

Semester 2, 2012
Due date: **Monday 13th August 09:00am**
Total points: 100

General guidelines

The purpose of this assignment is for you to learn about the basic programming constructs of BDI agent languages (in particular, Jack), as well as to get started with the Prometheus Design Tool (PDT) to design and develop agent systems. Normally, in building a system one would start with fleshing out the basic ideas of the system - system specification/requirements/etc. However, we have found in teaching Agent Oriented Programming and Design that students need to have a concrete grasp of the programming constructs, before doing design. Assignment one therefore consists of small programming tasks introducing you to the important programming constructs of agent system development. In order to provide a more cohesive and integrated project experience, we base these tasks on small additions to the core program we provide as a starting point for your major project. The Prometheus Design Tool produces skeleton code, and you will be using this feature in your large project. Consequently we will guide you from the beginning, in introduction of entities at the design level, followed by augmentation of the skeleton code produced, to give an executable program.

The assignment is structured, and builds on existing code, so each step should be relatively straightforward. The steps are cumulative and build on each other. It will also provide you with an initial framework for your final assignment. It is divided into three parts:

- **Part A** is *core* and the most challenging (worth 60 points).
- **Part B** is also *core*, but should be somewhat easier (worth 25 points).
- **Part C** is advanced and *optional* in order to get a high distinction mark (worth 15 points).

The assignment is due **Monday 13th August 09:00am**, is worth 25% of your total mark, and is a hurdle. **The assignment can be done in groups of up to two students.** Each group will need to hand in:

1. a report as detailed below;
2. the zipfiles `exercise-n.zip` for each of the exercises as described below. Among other things, you will be required to include the agent source code after completing each exercise. The agent source includes file `design.pd` and the whole sub-directory `src/`. To make a zip file with the whole agent source (e.g., `agent-2.zip` for exercise 2), you can use the script `pack-agent.sh`. Make sure your code ***compiles with no errors*** before building the source code zip package.

All items should be submitted electronically using Weblearn. **In addition, hard copies of the report should be handed in using the assignment box (room 14.08.03).**

Detailed Instructions

You should do each of the specified exercises as explained below. Note that you are asked to save your source and executable code at each stage. This is so that you can receive partial credit, as it is possible for mistakes in later exercises to affect the correct functioning of earlier steps. First read the full assignment description to get a complete understanding of the steps you will be doing. Do the exercises in increasing order. After each exercise, write the relevant section of your report and pack all the code in a file called `exercise-N.zip`, where `N` is the corresponding exercise number.

We strongly recommend keeping copies of each "stable" version you produce. PDT do have bugs and sometimes they are hard to trace back or even revert (e.g., the `design.pd` file may get corrupted). In these cases, the only solution is to go back to a previous "healthy" version. **The best strategy is to use a version control system, like svn or git.**

In most cases, you will follow three steps to implement a change or extension to your agent, namely:

1. Do the *design* changes within the PDT tool (e.g., adding a new entity or linking two entities). Once saved, this step will just modify the PDT application file `design.pd`.
2. Re-generate the Jack sources by using the code generation facility of PDT so that the design changes you did are reflected in the Jack sources. The new Jack files should be generated in directory `src/rmit/ai/clima/jackagt/`, by setting the source directory `src/` and the package to `rmit.ai.clima.jackagt`.
3. Finally, using the editor, implement the necessary code in the Jack source files (e.g., modify the body of a plan or add the corresponding queries for a beliefset).

The Report

Your report should be a complete, unambiguous, and precise description of your solution to each exercise. One should be able to reconstruct your solutions and the process towards those solutions fully: **one should not require to inspect any source code, the report should be self-contained.**

The report should contain a separate section for each exercise. For each step within the exercise sections, you should:

- Show the code you have added/changed, with a brief text explaining what you intend the change/addition to do. Do not just paste or print the whole source, just present the snippets that are necessary to understand your work: *precision will also count towards the mark of each answer*.

  For example, if you modify agent plan `HandlePercept`, then your report should explain such modification in a clear, complete but concise, and unambigous manner so that it is not necessary to check the sources.
- Indicate if you believe you have completed the step successfully, and note any problems/issues.
- Insert a small part of your trace to illustrate this if appropriate.

At the end of the exercise, summarise what you have learnt from this exercise and note any issues, concerns or problems. Answer any questions asked in the exercise. Your report may therefore have the following structure:

Section X
Step 1:
modified plan foo to print out the cell positions.
<snippet of file foo.plan inserted>
Step 1 completed successfully
<small part of trace showing printed output>

Step 2
...
Step 3

Exercise X summary
In this Exercise I learned to ...

Section X+1
...

PART A

Exercise 1: Warming up! **[5 points]**

First, let us make ourselves familiar with Jack, the PDT design tool, and the game server.

1. Get the latest Jack core agent system (not the one you used for the Warm Up task) and the game server. Read the user manual for the core system and the `README.TXT` file that comes with the system, and prepare everything as instructed, including possibly installing the PDT plugin in ECLIPSE. Make sure you familiarise with the directory structure of the agent system.
2. Open the Jack agent system with PDT, by opening file `design.pd`.
3. Get a diagram for the `Player` agent in a file named `player-design.png`. (Use the PDT export image utility.) **[1]**

4. Set the following description for player agent's `HandlePercept` plan: "*This plan assimilates each percept information received from the game server.*" (Select the `HandlePercept` plan in *Detailed Design* and use the Properties panel.) **[1]**

5. Generate the Jack code for the agent system using the code generation capability of PDT. Make sure you enter the correct directory for the sources and the right package. **[1]**

6. Check that the Jack sources are now present; do a listing of dir `src/rmit/ai/clima/jackagt/events/`.

7. Next, compile your Jack application by using the shell script by running `make all` or script `compile.sh`.

   ★ Run *one full simulation of* 20 *steps*[1] of the dummy Jack agent system against the server. You should redirect all the output of the Jack agent to a file named `trace1-7.txt` and collect the corresponding server SVG directory for the simulation.

Put together the whole agent source (that is, file `design.pd` and the whole directory `src/`) in a file called `agent-1.zip` (you can just run `pack-system.sh agent-1`). Finally, put together files `player-design.png`, `trace1-7.txt`, `agent-1.zip`, and the SVG directory you collected in a file named `exercise-1.zip`. **[2]**

Exercise 2: Knowing where we are **[25 points]**

In this exercise, you will learn how to deal with events and beliefs. (See Sections 5 and 9 in Jack manual.)

At every simulation cycle our initial dummy agents would just try to move randomly to some adjacent cell without much thought. To do that, the agent does not even need to know or remember its current location. However, to achieve more intelligent behaviour, the agents would eventually need to "know" their current location in the grid. In this exercise, you will extend our initial agents with a beliefset to store the agent's location at every point in time.

1. Using your editor of choice, modify plan `HandlePercept` (located in `src/rmit/ai/jackagt/plans/` directory) to print out *in the console* the location of the agent as encoded in the `request-action` event. That is, your agents should print the following whenever a new `request-action` arrives: **[5]**

   ```
   A request-action event states that <agent> is currently at loc.  <(x,y)>
   ```

   where `<agent>` is the agent's id within JACK (you can get the agent's name with `getAgent().name()`) and `(x,y)` are the current coordinates of the agent.

   ★ Run a full 20 steps simulation of your agent and store the output in file `trace2-1.txt`.

2. Create a beliefset named `CurrentPosition` to remember the agent's location. You do this by adding a new *data* object in the Detailed design of the `Player` agent. Set the beliefset name to `bel_currentPosition` and then, in the properties panel, set and create the new *Data Type* called `CurrentPosition`. Click on the new data type and create the corresponding fields for the new data type. Finally, set the JackCode modifier to `#private`. **[6]**

3. Generate the Jack code using PDT code generation capability and verify that your new beliefset has been created, that is, you should now see a new file `src/rmit/ai/jackagt/data/CurrentPosition.bel`.

4. By using your editor of choice, edit this new file to include the following two queries for the new beliefset:

   (a) Query `check(x,y)` is true if and only if the agent is at location `(x,y)`. **[2]**

   (b) Query `get(x,y)` returns the agent's current location. **[3]**

   Notice that you will *not* do this step within PDT, but with your editor (e.g., ECLIPSE) (*after* you have generated the code with PDT). It is important that all changes you do in this way are performed, *outside the protected* PDT *area*, which are explicitly marked in each file. When you (re)generate the code again, PDT will not touch anything that you modified outside the protected block, but it will override fully the protected area.

5. Modify (existing) plan `HandlePercept` to correctly update the new introduced beliefset `CurrentPosition`. To do that you need to:

   (a) first, *link* the plan and the new beliefset in PDT (in the Detailed Design). Because the plan will modify the beliefset the link should go from the plan to the data;

   † OBS.: Because the plan is inside a capability you will have to import the beliefset from the main agent into the corresponding capability by setting the modifier in the JackCode section to `#import`

[1] To do that, you should carefully edit file `conf/serverconfig.xml` and modify the tag `maxNumberOfSteps`.

(b) next, instruct PDT to *(re)generate* the Jack code. Observe that you should now see a corresponding `#modifies` statement in the protected are of the plan file `HandlePercept`;

(c) finally, by editing file `src/rmit/ai/jackagt/plans/HandlePercept.plan`, *implement* the necessary code in the body of the plan to correctly update the agent's beliefset. **[4]**

6. Modify plan `ConsoleBeliefPrinting` to print out the location that the agent believes it is in. That is, your agents should print the following: (observe you first need to link the corresponding beliefset with the plan) **[5]**

```
Agent <agent> believes it is currently at location <(x,y)>
```

where `<agent>` is the agent's id within JACK (you can get the agent's name with `getAgent().name()`) and `(x,y)` are the current coordinates of the agent.

★ Run a full 20 steps simulation of your agent and store the output in file `trace2-4.txt`.

Finally, write your report, make a zip file named `agent-2.zip` containing the agent source, and put it all together (that is, report, traces, and agent source) in a file called `exercise-2.zip`.

Exercise 3: Keeping track of gold pieces **[30 points]**

In this exercise, we will further investigate beliefsets (see Section 9.4 of Jack manual) by extending the agent's belief to keep track of locations that have gold pieces. The initial agent already keeps track of which cells are known to be blocked by an obstacle and which ones are not. Beliefset `CellEmpty` is updated when plan `UpdateCellsAround` is executed in order to address a pending event `UpdateBel`, which is in turn posted by plan `HandlePercept`.

1. Familiarise yourself with beliefset `CellEmpty` and the way it is updated. Explain why the updating of beliefset `CellEmpty` has been decoupled from plan `HandlePercept`, that is, why it is not the case that the beliefset `CellEmpty` is actually updated by plan `HandlePercept` itself? **[1]**

   ⁎ HINT: Consider the players potentially sharing information (see Exercise 6).

2. Create a beliefset named `GoldAt` to represent the agent's beliefs about gold pieces in the world. The beliefset should provide, at least, the following queries: **[4]**

   (a) Query `checkIfGold(x,y)` is true when it is believed that input location `(x,y)` contains a gold piece, and false when it is believed that location `(x,y)` does not contain gold. **[2]**

   (b) Query `getLocWithGold(x,y)` returns any location `(x,y)` that is believed to contain gold. **[2]**

   (c) Functional query `countGold()` returns the number of locations that are believed to contain gold. **[4]**

   ⁎ HINT: **Use an auxilarly indexed query and `.next()` on it.**

   (d) Complex query `checkIfGoldInRow(x)` is true when it is believed that somewhere in row `x` there is a gold piece, and false when it is believed that there is no gold in row `x`. **[2]**

3. Modify plans `StartSimulation`, `FinishSimulation`, and `UpdateCellsAround` to correctly initialise, reset, and update the new beliefset `GoldAt`. **[1,6]**

4. Using beliefsets `CellEmpty` and the newly introduced `GoldAt`, modify plan `ConsoleBeliefPrinting` to print the following message in the console: **[6]**

```
Agent <agent> believes there is gold/obstacle in the <direction>
```

when the corresponding agent believes that there is gold or an obstacle, respectively, in the cell that is adjacent to its current position towards `<direction>`. The direction can be any of the 8 possible directions north, east, north-east, south-west, etc. For example, `player1` agent should print-out (among possibly other things):

```
Agent <player1@portal> believes there is gold in the north
Agent <player1@portal> believes there is an obstacle in the south-east
```

if it is in location $(15, 8)$ and believes there is gold in cell $(14, 8)$ and an obstacle in cell $(16, 9)$.

★ Run a full 20 steps simulation of your agent and store the output in file `trace3-5.txt`.

⁎ HINT: You will have to use the beliefset `CurrentPosition` you created in Exercises 2 so as to obtain the current location of the agent and then iterate around the adjacent positions. You may want to use the class methods in class `rmit.ai.clima.gui.grid.GridPoint` to manipulate locations in the grid (e.g., to obtain the coordinates of a cell north to another cell we can use the `getNorth` class method).

5. As you can see, whereas obstacles are displayed in the GUI interface, gold pieces are not. This is because, until now, your agents were basically unaware of gold pieces in the grid. Now, your agents know about gold pieces they have seen! So, all the agents have to do is to inform the GUI Agent (in charge of managing the GUI interface) of changes in cells wrt gold. To that end, first link in the design event `ECellChange` to the new beliefset created to track gold pieces, re-generate the code, and *copy-paste* the following code to event `ECellChange.event`: **[2]**

```
#posted when (bel_goldAt_dat.getLocWithGold($x, $y)){       // Cell has a gold
  gridObject = new GridObject($x.as_int(),$y.as_int(),GameGraphics.getGoldString());
  message = "Just learned of gold at " +GridPoint.toString($x.as_int(),$y.as_int());
}

#posted when (!bel_goldAt_dat.getLocWithGold($x, $y)){      // Cell has no gold
  gridObject = new GridObject($x.as_int(), $y.as_int(), "nogold");
  message = "Just learned no gold at " + GridPoint.toString($x.as_int(),$y.as_int());
}
```

Compile and run your new system, you should now be able to see gold pieces in the GUI as the agents discover them.

Finally, write your report, make a zip file named `agent-3.zip` containing the agent source, and put it all together (report, agent, and traces) in a file called `exercise-3.zip`.

PART B

Exercise 4: Should I pick or should I go? **[15 points]**

In this exercise, you will learn to add plans and plan selection (see Section 7 in Jack manual).

So far, our agents do not pick up any gold at all, even if they are in a cell with a gold piece in it! In this exercise we will make our agents smarter by taking advantage of when they see gold at their current position.

1. Add a new agent beliefset `NumCarryingGold`, and the necessary code to update it, so as to keep track of the number of gold pieces the agent is currently holding. Extend plan `ConsoleBeliefPrinting` to print out: **[2]**

   ```
   Agent <agent> is currently carrying <n> pieces gold
   ```
   [1]

   ★ Run a full 20 steps simulation of your agent and store the output in file `trace4-1.txt`.

   † OBS.: See that since agents never try to pick up gold, they will always report 0 pieces of gold!

2. Next, add a plan called "`PickGold`" to handle event "`EAct`" such that the robot will execute a `pick` action, (that is the plan should send a `pick` action to the game server by using the given `ExecuteCLIMAaction` wrapper event). In addition, the robot should print out the following message when the plan is executed: **[4]**

   ```
   Agent <agent> tries to pick up gold
   ```

   ★ Run a full 20 steps simulation of your agent and store the output in file `trace4-2.txt`.

   ★ If you do not see the above message appearing in the trace, explain why?

3. Improve your agents such that they always pick gold if it is possible to do so, that is, if they are in a cell containing gold and they are not fully loaded. Explain how your code ensures that `MoveRandomly` is used only when `PickGold` cannot be used, and if you see other ways to achieve this, briefly explain them. **[6]** **[2]**

   ★ Run a full 20 steps simulation of your agent and store the output in file `trace4-3.txt`.

   † OBS.: Hopefully you will now see how your agents are smart and quick enough to pick up gold when possible. When they do, they should report, in the next cycle, that they are carrying some gold.

Finally, write your report and put it all together (report, traces, and agent source) in a file called `exercise-4.zip`.

Exercise 5: Realising that we moved **[10 points]**

In this exercise you will learn how to post events automatically by using the `#posted when` statements in events (see Section 5.4 and 5.7 in the Jack manual). As you may have realised, whenever the agent tries to move randomly to some adjacent cell, the movement may be successful or not. In fact, the movement may fail because the agent tried to move towards a place where there is an obstacle or just because the action failed (remember actions may fail with some (low) probability).

1. Define a new event that is triggered automatically whenever beliefset `currentPosition` changes. As usual, this will involve adding a new event within PDT, re-generating the Jack code, and editing the new file created for the event in question. Using PDT, document the condition under which the event shall be triggered. **[7]**

2. Make a plan which responds to this new even by printing out the following message: **[3]**

   ```
   Agent <agent> has just moved to location <(x,y)>
   ```

   where `(x,y)` stands for the new location of the agent. Notice that such message should be printed out *only* when the agent has moved from one cell to another.

   ★ Run a full 20 steps simulation of your agent and store the output in file `trace5-2.txt`.

Write your report and put it all together (report, traces, and agent source) in a file called `exercise-5.zip`.

PART C

Exercises 6: Sending a message to a coordinator agent **[15 points]**

You will learn now how to make agents communicate by sending events to each other. (See Section 5.8. in Jack manual).

In the initial framework, there is an extremely simple "coordinator." Such coordinator agent does almost nothing and stores almost no information. Nonetheless, it can be extended to potentially collect and fuse information from all six agents and help the team with more "global" decisions. This exercise will help on developing such coordinator agent.

So, at this point, the coordinator only has one beliefset in order to store the information on the current simulation (i.e., the grid size, the depot position, and the number of steps). The interaction between the players and the coordinator is minimal, and it amounts to players informing the coordinator when a new simulation has started, when the current simulation has ended, or when the whole tournament is over. In what follows, we will extend the coordinator to keep track of more information so that eventually it can help the team with relevant decisions.

1. Define a beliefset for the coordinator to keep track of the player positions. **[3]**

2. Modify existing plan `EndSimulation` of the coordinator to "reset" the new beliefset when a simulation ends. **[2]**

3. Do all the modification required so that each agent *informs* the coordinator agent of its new position when it has successfully moved from one cell to another cell. Whenever the coordinator receives this information from any of the players, the coordinator should print out the current location of *each of the six robots*. **[4]**

   ★ Run a full 20 steps simulation of your agent and store the output in file `trace6-3.txt`.

4. Equip the coordinator with the beliefsets already defined to keep track of empty cells and cells with gold, that is beliefsets `CellEmpty` and `GoldAt`. **[1]**

5. Do all the modification required so that coordinator agent can update both beliefsets `CellEmpty` and `GoldAt` with information provided by all the players. That is, each player should inform, at every cycle of the simulation, all the cell information received in the cycle and the coordinator should update its beliefs according to that. See that because *every player* will be informing the coordinator, the coordinator agent will eventually have a broader and more complete view of the whole grid. **[3]**

   ⁎ HINT: Use the existing event `EUpdateBel` and plan `UpdateCellsAround`.

6. Define an event `ECellStatusChanged` that will trigger automatically when the coordinator learns *(i)* of a new obstacle in the grid; and *(ii)* of a player agent that has changed its position; and a plan to handle such event that will print the location of the new obstacle found or the location where the agent in question has just moved. **[2]**

   ⁎ HINT: Store the message to be printed already in the event as a String and make the plan just print such message.

   ★ Run a full 20 steps simulation of your agent and store the output in file `trace6-5.txt`.

Once you have ran and checked that all works fine, write your report. Save this version of your agent in a file named `agent-6.zip` and put everyting together into a file file named `exercise6.zip`.