Laboratory: Mental Agent Programming with ASTRA

This first lab on ASTRA programming is focused on what I call mental agent programming, where we will develop programs that manipulate a mental representation of some problem domain. This is a useful exercise because it will help you to get to grips with the manipulation of the mental states used within agent programming languages. It is also necessary because we have not yet covered the concept of an environment or discussed how agents interact with environments in any great level of detail (this will happen in the coming weeks).

So, the next sections will introduce three scenarios that you will use as a basis for exploring mental agent programming:

- **Scenario 1: Light Switches** will take the light switch example covered in the class a little further.
- **Scenario 2: Household Activities** will focus on modelling the activity of a simulated human in an apartment.
- **Scenario 3: Building Automation** will build on the first scenario to cover more concepts around building automation.
-

While there are many valid ways of solving the problems we will cover in this laboratory, one of the key objectives in this worksheet is to try to link together the concepts of practical reasoning and agent-oriented programming. As a quick recap – when writing programs using practical reasoning, you should write 2 types of rule:

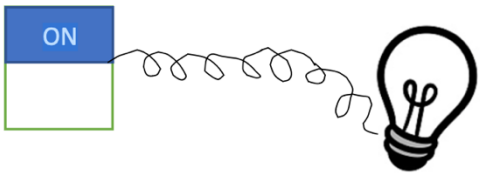
- **Deliberation Rules:** Identify undesirable states (encoded as events) and include a plan that identifies how to "recover" the environment to a desirable state (encoded as goals).
- **Means-End Rules:** Identify how to achieve any goals declared (both through deliberation and means-end reasoning).

Marking Scheme

THIS LABORATORY IS NOT ASSESSED

Scenario 1: Light Switches

Papers on Agent Programming languages often include toy examples to illustrate concepts. For ASTRA, the simplest toy example is the light switch. In this environment, an agent is created that is responsible for monitoring a light switch and turning a light on/off to reflect the state of the switch. Obviously, the simple solution to doing this is to connect the light and the switch together using a standard electrical circuit, but the point here is not to implement something practical, but to design something that is easy to understand.



Let's start with a simple system containing 1 switch and 1 light. The first thing we need to do is to model this domain. Let's start by modelling the objects in the universe of discourse. In our problem domain, we have 2 objects: a switch and a light. For the purposes of this lab, we are going to label the switch `s1` and the light `l1`. Notice that this is a little different to the examples I covered in the slides, where these two objects were represented implicitly via predicates that identified their states (e.g. light(on) represented the fact that the light is on). I have changed it here because the later steps of the laboratory will introduce additional lights.

That said, we still need to model the states of the switch and light. As with the previous examples, we can use two constants:

- `on`: the switch is in the on position or the light is turned on.
- `off`: the switch is in the off position or the light is turned off.

Because we have made the switch and light object explicit in this model, the predicates we use to model our problem domain need to be slightly different.

- `switch(Sw, St)` models the state (`St`) of switch (`Sw`) (e.g. if the switch is initially off, this would be modelled as: `switch(s1, off)`)
- `light(Lt, St)` models the state (`St`) of light (`Lt`) (e.g. if the light is initially off, this would be modelled as `light(l1, off)`)
- `connected(Sw, Lt)` models the connection between a switch (`St`) and a light (`Lt`) (e.g. in our example, this is modelled as `connected(s1, l1)`)

Part 1: The Basics

The initial task is to create an agent program that ensures that the light is in the correct state based on the state of the light switch. The idea here is to create an agent that ensures that the light state is the same as the switch state. Should this not be the case, the agent must update the state to be correct. Further, the agent should print out a message to the console whenever the Light state changes. The format of the message should take the form:

```
Light: <name> is in state: <state>
```

In the lecture on Practical Reasoning and Agent Programming (slides 6-21), I go over a version of this problem in AgentSpeak(L). I recommend looking at those slides and adapting the solution to both the new domain model and ASTRA. You will need to add code to print out the above message to the console. I would recommend doing this by writing a rule to handle a `+light(Lt, St)` event (this event will be generated whenever the `light(Lt, St)` belief is updated).

Initialise the program so that the light is off, but the switch is on. Run the program, you should see the following output:

```
Light: l1 is in state: off
```

This happens because events are generated for every initial belief of the agent.

Now add the following `+!main(…)` rule, which simulates the light being turned on:

```
rule +!main(list args) {
    +switch("s1", "on");
}
```

This rule is special because, when you run an ASTRA program, the `astra.main` property (in the `pom.xml` file) is used to identify an ASTRA class that should be instantiated when the program starts. In addition to any initial beliefs and goals, the instantiated agent receives a `+!main(list args)` event that can be used to trigger some initial behaviour. Critically, the list of `args` contains any parameters passed to the Java `main(String[] args)` method allowing you to pass initial information to your multi-agent program. This method is called when you use `mvn astra:deploy`.

When you run the program this time, the expected output is:

```
Light: l1 is in state: off
Light: l1 is in state: on
```

Part 2: When Two lights Go To War…

This second problem extends the first problem by introducing a second light (`l2`) that also connects to switch (`s1`). The behaviour is the same – if the switch is on, both lights should be on; if the switch is off, both lights should be off.

(a) Start by updating the beliefs (model) to reflect the addition of the new light. Run the code an see what happens…

(b) To make everything connected to a switch work, we need to modify the `+switch(…)` rule as follows:

```
rule +switch(string S, string state) {
    foreach(connected(S, string L)) {
        !light(L, state);
    }
}
```

Run this code an see what happens…

(c) So, for (b), you may have got something like this:

```
[main]Light: l1 is in state: on
[main]Light: l2 is in state: off
[main] Event was not matched to rule: +!light("l1","on")
[main] section1.part2.LightSwitch.!light("l1","on"):23
[main] +switch("s1","on")
[main]Light: l1 is in state: off
```

The "Event was not matched…" error arises because intentions are interleaved and events are processed as it. This means that the handling of the initial event +switch("s1", "off") has not completed before +switch("s1", "on") is executed and the program ends up in an inconsistent state. To fix this, we can use the synchronized keyword:

```
synchronized rule +switch(string S, string state) {
    foreach(connected(S, string L)) {
        !light(L, state);
    }
}
```

The code should now run correctly with expected output:

```
Light: l1 is in state: on
Light: l2 is in state: off
Light: l1 is in state: off
Light: l1 is in state: on
Light: l2 is in state: on
```

Part 3: The Flasher

The third problem is an adaptation of part 1. It involves adding an additional behaviour to the agent to make it flash the light on and off 10 times (i.e. on 5 times and off 5 times).

To achieve this, you should start with the light and switch in an off state. Modify the `+!main(list args)` rule to trigger the flashing behaviour.

The goal state of the behaviour is to have a light flash a number of times. This can be modelled as a goal `!flashed(string light, int times).` The goal is achieved when the light has been flashed the specified number of times. At that point, we should believe that the light was flashed `times` times (e. g. `!flashed("l1", 10)` -> `flashed("l1", 10)`).

Flashing involves turning the light `on` for a fixed amount of time and then `off`. We can achieve this by turning the switch on and off repeatedly. This behaviour can be achieved by the following code:

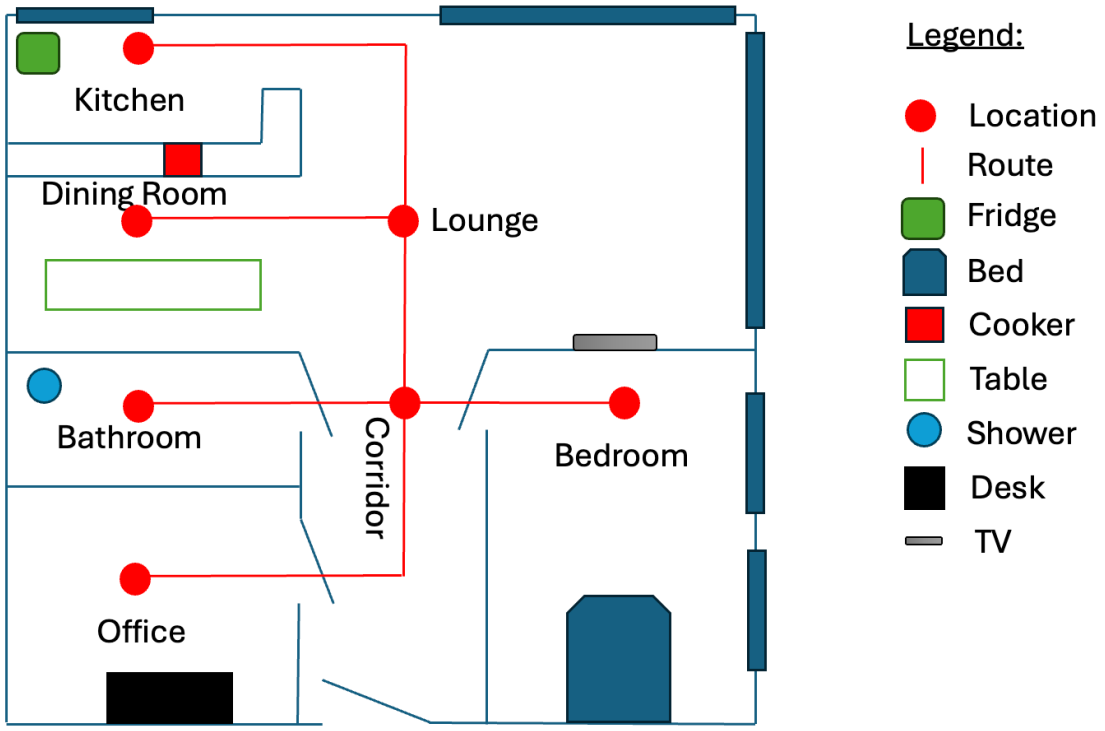
```
rule +!flashed(string light, 0) { }

rule +!flashed(string light, int times)
  : connected (string switch, light) & flashed(light, int T) {
    !switch(switch, "off");
    S.sleep(1000);
    !switch(switch, "on");
    S.sleep(1000);
    !switch(switch, "off");

    -flashed(light, T);
    +flashed(light, T+1);
    !flashed(times-1);
}

rule +!flashed(string light, int times) {
    +flashed(light, 0);
}
```

Here, S refers to the `System` module. Write some code to make the light flash 5 times (it should be in state `on` 5 times) and the output should look like this:

```
Light: l1 is in state: on
Light: l2 is in state: off
Light: l1 is in state: off
Light: l1 is in state: on
Light: l2 is in state: on
Light: l1 is in state: off
Light: l2 is in state: off
…
```

Section 2: Household Activity

This section of the worksheet has two aims: to further expand on the programming by practical reasoning approach, and to introduce you to the multiple inheritance reuse mechanism used in ASTRA. Because of this, I will be providing partial solutions and asking you to fill in the missing parts.



The scenario is based around simulating activities of a human in an apartment. A floorplan of the hypothetical apartment can be seen above. The apartment consists of 7 rooms/areas that are connected in line with the red graph (connected means the areas you can reach given the area you are currently in).

The parts in this section will refer to this floorplan, incrementally building an example system.

Part 1: Basic Movement

The first task is to implement support for movement around the apartment. To do this, we need to first model the connectivity graph (areas and connections) and the concept of a being in a location. This can be modelled by the following ontology:

- `location(string L)`: L is an area (location) in the apartment
- `connected(string L, string M)`: locations L and M are directly connected to each other (e.g. `connected("office", "corridor")`)
- `current(string L)`: L is the current location of the agent

(a) Create a file called `Mover.astra` (this will be where we implement the basic behaviour of defined in this part) and copy the following code into it:

```
agent Mover {
    module Console C;
}
```

(b) Start by creating an ASTRA types block to define the above ontology:

```
types mover {
    formula location(string);
    formula connected(string, string);
    formula current(string);
}
```

(c) Write a set of initial beliefs that model the apartment. This should include one `location(…)` belief per area; a `connected(…)` belief for each connection between areas (connections are bi-directional); and a belief about the current location (here the bedroom). I will get you started with the following initial beliefs:

```
initial location("bedroom"), location("corridor");
initial connection("bedroom","corridor"),
                  connection("corridor"," bedroom");
initial current("bedroom");
```

(d) *The movement behaviour:* Movement implies the update of the `current(…)` belief, hence moving can be achieved by declaring a `!current(…)` goal. Naturally, movement is a means-end reasoning activity. So, for this part, we do not define any deliberation rules. We start by defining a default rule that implies we cannot move to the location specified (this is so we can understand when our code is not working):

```
rule +!current(string target) : current(string location) {
    C.println("COULD NOT FIND ROUTE FROM: "+ location);
}
```

(e) *Testing the behaviour:* In this code, we are going to test code by creating a second agent program called `MoverExample.astra` that contains the code below:

```
agent MoverExample extends Mover {
    rule +!main(list args) {
        !current("bedroom");
    }
}
```

We will modify this program as we improve our Movement behaviour, but for now, run this program by adding the following to a standard ASTRA Maven Project file:

```
<properties>
    <astra.main>MoverExample</astra.main>
</properties>
```

Run the project and see what happens.

(f) So, the agent doesn't know how to move anywhere at the moment; it doesn't even know when it happens to already be at the location that it wants to go to. We can fix this with the following rule:

```
rule +!current(string target) : current(target) {
    C.println("Already at: " + target);
}
```

This rule should go **above** the default rule defined in part (d).

Re-run your project and see what happens (it should print out that the agent is already at the specified location).

(g) Now, let's start to think about how the agent can move. It can move to a location if it is at a connected location, which can be expressed by the following formula: `current(string C) & connected(string L, C)`. Here, L is directly reachable from C. Use this to write another rule to handle the `!current(…)` goal for the situation where the target location is directly connected to the current location (hint: use the formula above in the context with a small modification):

```
rule +!current(string target) : ??? {
    -+current(target);
    C.println("Moved from: " + C + " to: " + target);
}
```

This rule should appear **above** the rule defined in part (d) but **below** the rule defined in part (f).

Update the `+!main(…)` rule in `MoveExample.astra` to declare the following goal `!current("corridor")` and run the program to see what happens. You should see the agent move from the bedroom to the corridor…

(h) So, what about rooms that are not adjacent (connected)? In this case, you need to make the agent move via an intermediary location (for example, to get from the bedroom to the lounge, the agent must move via the corridor). We can model this as a rule of the form:

```
rule +!current(string target) : ??? {
    !current(middle);
    !current(target);
}
```

Again you need to work out how to define the "middle" location. Write a rule and test it by modifying and running your program.

(i) Repeat a variation of (h) so that all locations become reachable (and you are finished).

Part 2: Driving Movement through Daily Activities

This task is concerned with extending part 1 to include daily activities. This is basically about creating an internal clock that can be used to "motivate" the behaviour of the agent. In addition to the clock, we will need to be able to define beliefs about the activities to be performed at specified times. This can be realised through the following ontology:

- `clock(int D, int H)`: defines a belief about the current day (D) and the current hour (H) of that day.
- `daily_activity(int H, string A)`: defines a belief about a daily activity (A) that should be performed at hour H each day. A can be interpreted differently depending on the scenario (see part 3 for an illustration of this).

(a) Create a new file called `Daily.astra` that contains the code below (not it extends the Mover program) and add an ASTRA types block (with identifier "daily") for the ontology defined in this part:

```
agent Daily extends Mover { }
```

(b) Modify the `Daily` program to also extend an agent program called `Clock`. (e.g. modify the code after the extends keyword to be: "Mover, Clock"). This is an example of multiple inheritance and refers to the implementation of an internal clock for the agent. The code for this file is given below:

```
agent Clock {
    module Console C;
    module System S;

    types daily {
        formula clock(int, int);
        formula daily_schedule(int, string);
    }

    constant int SIM_DAYS = 3;
    constant int DELAY = 1000;

    initial !clock(0, 0);

    rule +!clock(SIM_DAYS, 24) {
        C.println("SIMULATION FINISHED!");
        S.exit();
    }

    rule +!clock(int D, 24) { !clock(D+1, 0); }

    rule +!clock(int D, int H) {
        -+clock(D, H);
        C.println("Time: " + H + " [DAY="+D+"]");
        S.sleep(DELAY);
        !!clock(D, H+1);
    }
}
```

Create the `Clock.astra` file and copy the code into it.

You can test the `Clock` program on its own by modifying the `astra.main` property in the project POM file to be `Clock`.

NOTE: You should do this because it demonstrates that not every ASTRA program needs a `+!main(…)` rule. This example is triggered by the initial goal `!clock(0,0).`

(c) Now, we need to link the clock to daily activities. Because this is driving behaviour, we will be writing a deliberation rule that uses the `!current(…)` goal to drive movement of the agent. For this example, we will assume the string part of the `daily_activity(…)` belief is a location:

```
rule +clock(int D, int H) : daily_schedule(H, string location) {
    !current(location);
}
```

This rule basically checks the daily schedule every time the clock "ticks" and if there is an item in the schedule then the agent moves to the specified location.

(d) We can test this code through a sample scenario by creating a new file `DailyExample.astra` that extends `Daily.astra`. This file should contain a set of `daily_activity(…)` beliefs that map to the schedule below:

7	go to the bathroom (for a shower)
8	go to the kitchen (to make breakfast)
9	go to the office (to start work)
13	go to the kitchen to eat lunch
14	go to the office (to continue working)
17	go to the kitchen (to make dinner)
18	go to the dining_room (to eat dinner)
19	go to the lounge (to watch tv)
22	go to the bedroom (to sleep)

Run the program and check that it goes to all the specified locations at the specified times.

Part 3: From Locations to Activities

This task will involve extending the domain model to include the concept of activities (like eating breakfast) and mapping those activities to items that are needed (to perform the activity) and their locations in the house. The revised behaviour of our agent will be to follow an agenda of daily activities (see 2(d)) rather than daily locations.

To extend our solution to include the concept of activities, we need to add some more concepts to our domain model:

- `item(I)`: I is an item that can be used in an activity (e.g. `item("bed")`)
- `in(I, L)`: Item I can be found in location L (e.g. `in("bed", "bedroom")`)
- `activity(A)`: A is the activity currently being performed by the simulated human (e.g. `activity("sleeping")`)
- `needs(A, I)`: Activity A needs item I to be performed (e.g. to model the fact that sleeping requires a bed, we can state: `needs("sleeping", "bed")`)

So, the expected behaviour is that agents will be motivated by their internal log of daily activities to perform specific activities as specific times. Each activity needs some specific item to be performed. So, in order to perform the activity, the simulated human must go to the room / area containing the required item to perform the necessary activity (e.g. if the activity is sleeping, then they need a bed which is in the bedroom, so they go to the bedroom and sleep).

Create a file called `Homer.astra` whose associated class extends the `Daily` agent class and do the following:

(a) Create a new ASTRA types block

(b) Define a domain model that captures the items and their locations from the floor plans defined at the start of the section. Define the mappings between activities (specified implicitly in the daily schedule defined in part 2(d)) and items. Set the default activity to "sleeping".

(c) Create a new deliberation rule that declares an `!activity(…)` goal when an their daily schedule explains that they should perform a given activity (it is a subclass, so simply writing a new rule will override the rule in the `Daily` agent).

(d) Write a set of means-ends rules to achieve the `!activity(…)` goal. (there should be 3: serendipity, you are in the right place to perform the activity, and you are not in the right place to perform the activity). The second rule (where the agent is in the right place to perform the activity, the rule should update the `activity(…)` belief to reflect the new activity and print out "Doing: X" where X is the activity.

(e) Create a `HomerExample.astra` file that extends the `Homer` agent class. Define a set of `daily_schedule(…)` beliefs that capture the daily schedule from part 2(d). Run the program and check that the output is as expected.

Section 3: Building Automation

The final section returns to our light switch environment (section 1) to explore how the example can be extended to include other aspects of Building Automation. Each part in this section explores one possible extension.

Part 1: Light, light baby

This task involves the introduction of a light sensor that is going to simulate daylight. Our new energy-aware agent should only allow the light to be turned on when the light level is below a given amount (let's say 750 lux, which is normal for a factory environment). This could be modelled by the belief `light_threshold(750)`.

The output of the light sensor should be modelled using a `light_level(int)` belief whose parameter is the current light level in Lux. We are going to try to simulate the change of light over a typical day. To model the changing light level, we will use a second belief of the form `hour_light(int, int)` where the first parameter is the hour of the day (based on a 24 hour clock) and the second parameter is the light level for that hour in lux (to read more about lux values, please see https://greenbusinesslight.com/resources/lighting-lux-lumens-watts/):

The table below contains sample values for a standard winter day.

0	1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	100	250	400	800	1000	1000
12	13	14	15	16	17	18	19	20	21	22	23
1000	1000	1000	800	700	600	600	400	100	100	1	1

You should model this as 24 initial `hour_light(…)` beliefs. An example for the first three hours is:

```
initial hour_light(0, 1), hour_light(1, 1), hour_light(2, 1);
```

To simulate the passing of time, we can use the internal clock implementation Clock.astra developed for Section 2, Part 2. With each tick, the agent should read the light level for the current hour and update the `light_level(…)` belief to reflect that.

The final program you write should exhibit the following behaviour:

- If the light level is above the light threshold, then the light should not turn on when the switch is turned on.
- If the light level falls below the light threshold and the switch is on, then the light should turn on.
- If the switch is turned off then the light should turn off.

Make sure the program runs as expected.