Introduction to the InfoSpeak Environment

Matei Craciun, s1374265@inf.ed.ac.uk

Jason interface

This section will give you a brief overview of the Jason interface.

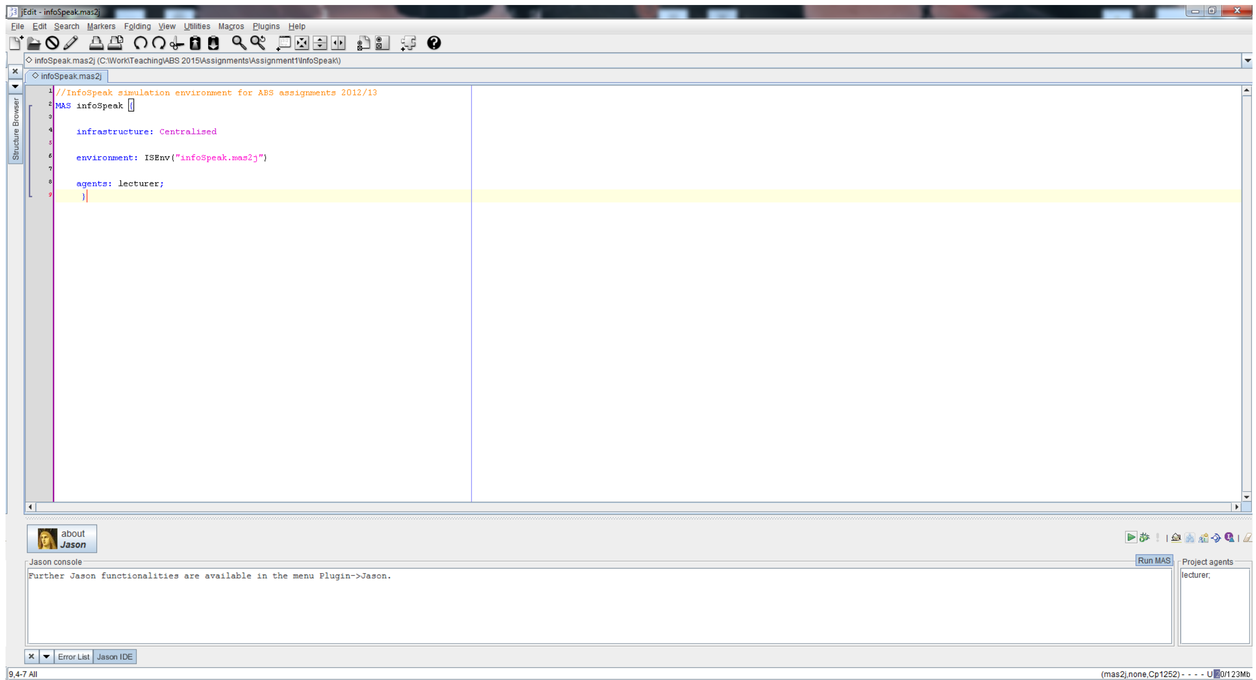

Figure 1: MAS2J file open in Jason

MAS2J files are project files for Jason, any project you want to load up and then run will have to be opened from this type of file. Figure 1 shows an opened infoSpeak.mas2j file: it defines the type of infrastructure and environment used (you do not have to concern yourself with either, centralized infrastructure basically has all agents run on the same machine and handles relevant communication channels and the environment is given and does not have to be modified by you; more on the environment in Section 4). The cursor hovers over the button to start the MAS.

Once started, you will see two windows: a visual representation of the InfoSpeak environment, as shown in Figure 2 and a console as depicted in Figure 4. The visual representation has a button (1) to start incrementing systems time, which you can track in the lower left corner of the window (2).

It is important to keep in mind that even when the time displayed is not incrementing/the system is paused, the simulation still runs. Jason runs reasoning cycles on all agents even when paused, regardless of the Hour/Day/Week system displayed.

It is very useful to know how to track your agent while the system is running and there are two ways to do that: either moving the mouse over the grid, which will give you the current coordinates of the cursor and any entities present at that position, displayed at (4). This does not update in real time so you should move the mouse to force an update, but it is still a useful way to remind yourself of coordinates for certain buildings and their names. A far better way

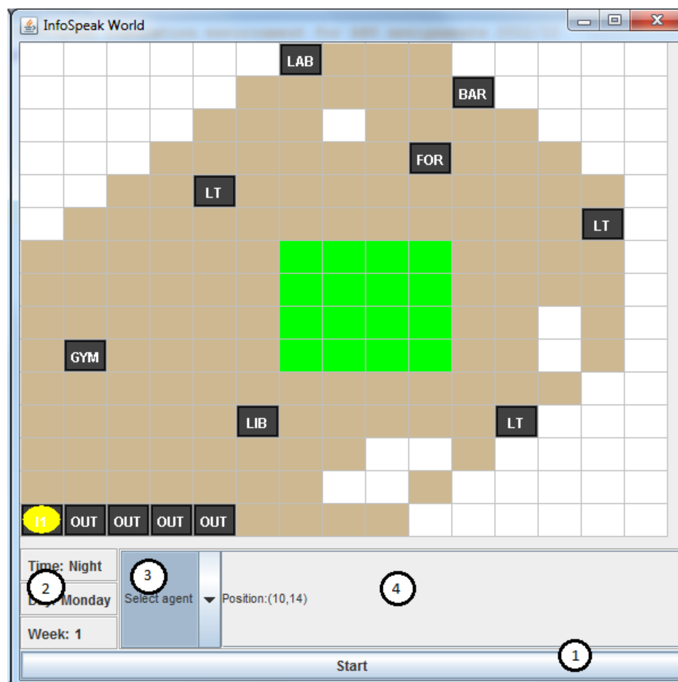

Figure 2: Visual interface of the gridworld

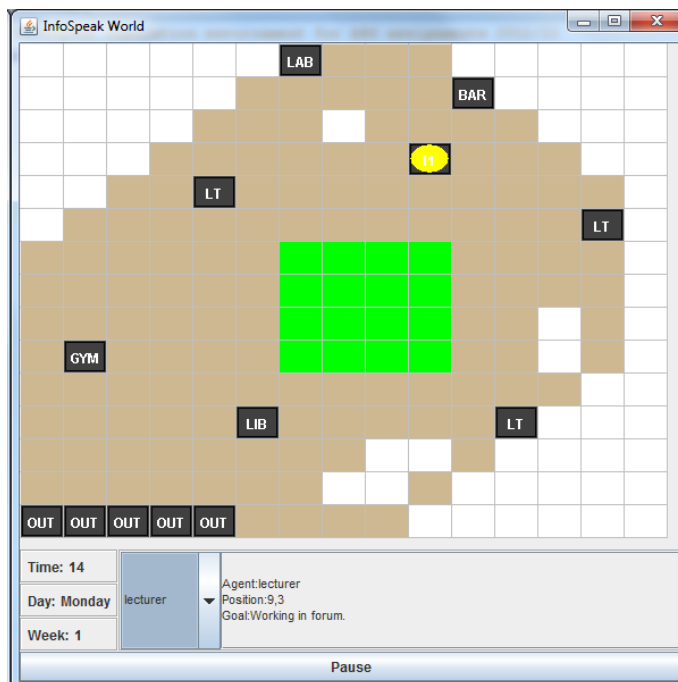

Figure 3: Tracking an agent using the drop-down menu

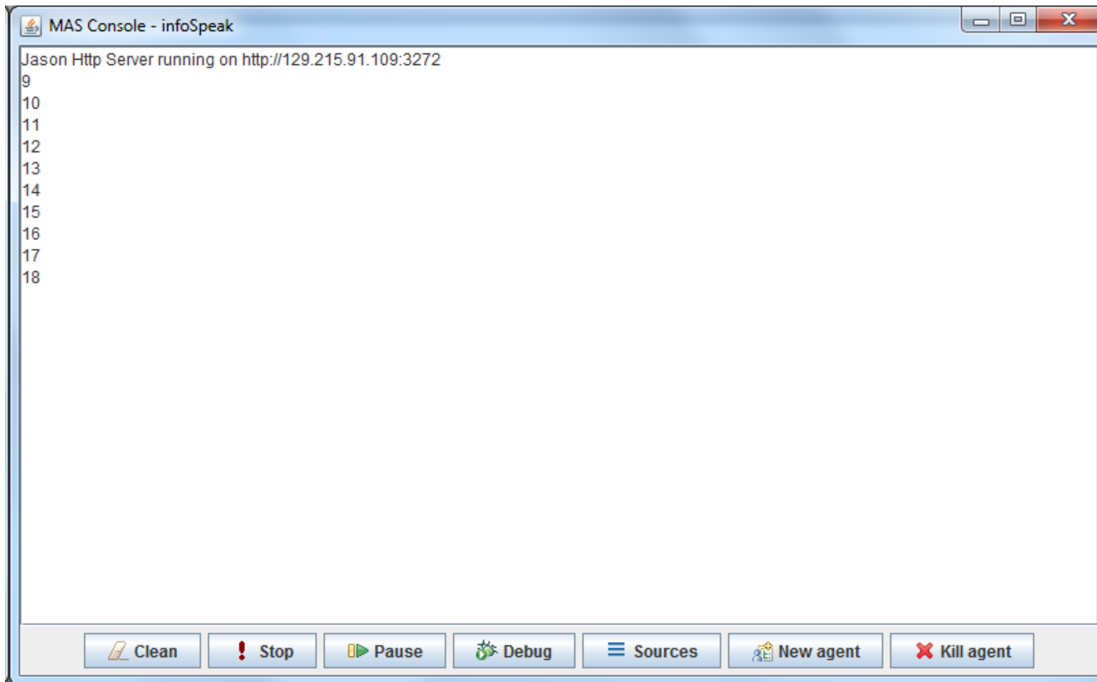

Figure 4: Console window

of tracking your agent is selecting it from the drop-down menu (3) which will keep its name, position and current goal displayed at (4), as seen in Figure 3.

The console window prints out the current hour and any other .print command you might insert in your agents program. Furthermore, the console prints out any error/exception should your application crash, making it very useful for debugging. By pressing "Debug" in this window you will open the mind inspector which will allow you to see the agent's belief base and its intentions, options and events, as shown in Figure 5.

This is where the fact that Jason keeps running even when paused becomes relevant: if you attempt to inspect the lecturer agent, the system will most likely crash. This is because the lecturer agent rapidly builds up a very large intention base while checking for any events to attend. It does this once every 20 ms. Depending on how you implement your agents, similar things might happen. In this case, either use .print functions to keep track of what/when beliefs are added to agents or make sure that your agent builds its intention stack more slowly.

Configuration file

The configuration file is used to define the environment. Every building, agent, event, etc. are defined in it. Please make sure that any entity you add is also represented in the configuration file.

The Lecturer Agent

This section provides information about the lecturer agent:

```
7 !start.
8
```

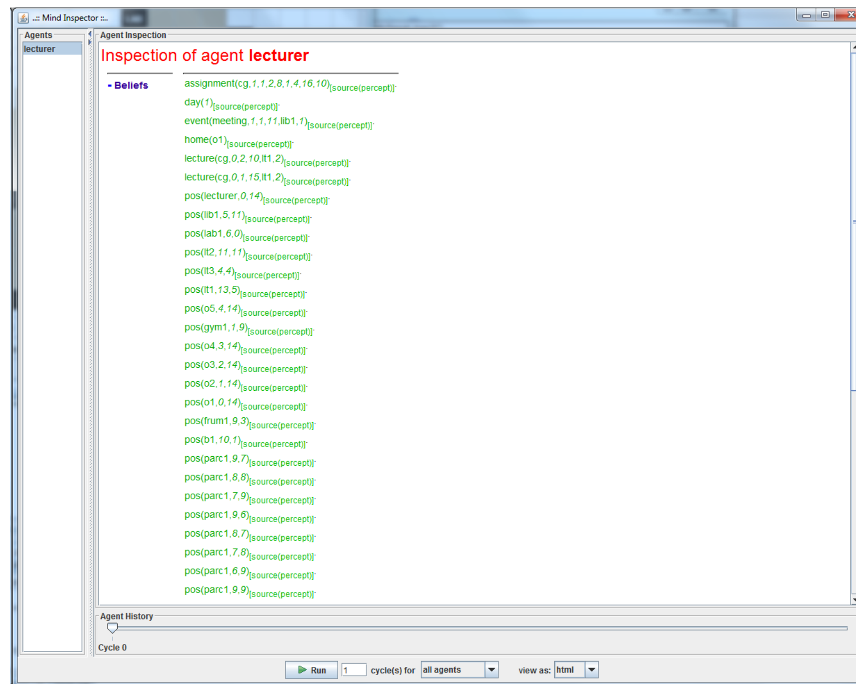

Figure 5: Debug window - beliefs

```
9  +!start <- .my_name(Me); !checkevents(Me).
```

Line 1 is the entry plan of the lecturer agent. After it is created it will attempt to execute the start plan. This intention will be matched by line 3, which is the next step in the agents execution. A function beginning with a "." signifies a pre-defined function. In this case `.my_name` will capture the agents name in the variable given. This variable (containing the name of the agent) is then passed as an argument to the `checkevents` plan. It is a good practice to always start your agents by capturing their names and then every other plan will have to take their name as a parameter. This is useful when you want an agent to find its own position and not the position of *any* agent.

```
12 +!checkevents(Me): week(WeekNow) & day(DayNow) & time(TimeNow)<- .findall(event(
      Name,Week,Day,Time,Place,Priority),event(Name,Week,Day,Time,Place,Priority)& (
      Week==0 | Week ==WeekNow) & Day == DayNow & Time == TimeNow,L); !pickaction(Me,
      L,WeekNow,DayNow,TimeNow).
```

The `checkevents` plan begins by matching the beliefs for week, day, and time. This means that `WeekNow`, `DayNow` and `TimeNow` will contain the values for the current week, day and time. Using these values we find all beliefs of the form `event(Name,Week,Day,Time,Place,Priority)` and place them in a list L. This is done using the `.findall` function. This function has three parameters: the first describes what form the beliefs which will be stored in L will have, the second represents what currently existing beliefs this information should be taken from and under what conditions (in this case we are just interested in events happening at the current hour, day and week). The last parameter will contain a list of all beliefs found in this way. We pass this list, along with the agent's identity and the current week, day and time to the `pickaction` plan.

```
14 +!pickaction(Me,L,WeekNow,Day,Time): .length(L) == 0 <- !givelectures(Me).
15 +!pickaction(Me,L,WeekNow,Day,Time): lecture(Name,Week,Day,Time,Place,Priority) &(
      Week == WeekNow | Week == 0) <- .concat(L,[lecture(Name,Week,Day,Time,Place,
```

```
    Priority)],M); ia.highest(M,X);!doaction(Me,X).
16 +!pickaction(Me,L,WeekNow,Day,Time) <- ia.highest(L,X);!doaction(Me,X).
```

The `pickaction` plan can be matched with three different situations: in the first one, the list we built is empty, there are no current events, in which case the lecturer will continue with the `givelectures` routine. In the second case, there is a lecture happening at the same time as an event (since the list in not empty and we successfully found a lecture with the same `Day`, `Time` and `Week` values). We add the lecture to the list using the `.concat` function and pass the new list to the predefined function `ia.highest`. This will return the highest priority event in the parameter X, which will then be passed to the `doaction` routine. In the third case, we simply use the `ia.highest` function to find the highest priority event and send that to the `doaction` routine (same as the second case, without adding the extra lecture).

```
18 +!doaction(Me,L): .nth(0,L,Type) & .nth(1,L,Name) & .nth(4,L,Time) & .nth(5,L,Place
    ) & pos(Place,X,Y) & not pos(Me,X,Y)<- .concat("Going to ",Name," ",Type,".",G)
    ; add_goal(G); !goto(Me,X,Y); .concat("Attending ",Name," ",Type,".",H);
    add_goal(H); one_hour(Time); !checkevents(Me).
19 +!doaction(Me,L): .nth(0,L,Type) & .nth(1,L,Name) & .nth(4,L,Time) & .nth(5,L,Place
    ) & pos(Place,X,Y) & pos(Me,X,Y)<- .concat("Attending ",Name," ",Type,".",H);
    add_goal(H); one_hour(Time); !checkevents(Me).
```

The `doaction` routine has two possible plans: in both cases we begin by extracting the value from the L argument, which is the event with the highest priority, previously selected. It has the form previously described, in the first argument of the `.findall` function. Where the two plans differ is that in the first case the event takes place at a different position than the one the lecturer agent currently is in, and in the second case the lecturer agent is already at the place in which the event is held, it simply has to attend it. It does this by first constructing a string consisting of the name and type of the event and then adding this string as the current goal. It will then execute the `one_hour` function (predefined inside the environment file ISEnv.java) which causes the agent to suspend its activity for an hour (attending the event), after which the `checkevents` routine resumes. To determine that the agent is at a different location, we first capture the coordinates of the event location with `pos(Place,X,Y)` and then attempt to see if the belief of the lecturer about its current position (given by `pos(Me,X,Y)`) has the same coordinates (since we use the same variables, X and Y).

```
21 +!givelectures(Me) : week(WeekNow) & day(Day) & time(Time) & Time > 17 <- !gohome(
    Me); !sleep(Me).
22 +!givelectures(Me) : week(WeekNow) & day(Day) & time(TimeNow) & pos(Place,X,Y) &
    lecture(Name, Week, Day, Time, Place,Priority) & Time == TimeNow & (Week ==0 |
    Week==WeekNow)<- add_goal("Going to the next lecture");!goto(Me,X,Y); .concat
    ("Attending ", Name, " lecture",G);add_goal(G); one_hour(Time);!checkevents(Me)
    .
23 +!givelectures(Me) : week(WeekNow) & day(Day) & time(Time) & Time == 12 & not (
    lecture(Name,Week,Day, Time, Place,Priority) & Week == 0 | Week == WeekNow)<- !
    gotobar(Me); add_goal("Having lunch"); !checkevents(Me).
24 +!givelectures(Me) : week(WeekNow) & day(Day) & time(Time) & not (lecture(Name,Week
    ,Day, Time, Place,Priority) & Week == 0 | Week == WeekNow) & Time > 9 & Time <=
    17<- !gotoforum(Me); add_goal("Working in forum.");!checkevents(Me).
25 +!givelectures(Me) : week(WeekNow) & day(Day) & time(Time) & pos(Place,X,Y) & not (
    lecture(Name,Week,Day, Time, Place,Priority) & Week == 0 | Week == WeekNow) <-
    !checkevents(Me).
26 +!givelectures(Me) <- !givelectures(Me).
```

The `givelectures` routine has many possible plans, each representing a different situation. They are as follows:

1. If it is after 5 p.m., the lecturer goes home and sleeps.

2. The lecturer has a lecture now so it will attend the lecture. This involves going to the required location with the `goto` plan, adding the goal of attending that lecture (a very useful debugging tool, since goals can be tracked), spending one hour and then returning to the `checkevents` routine.
3. If it is 12 p.m. (noon) and the lecturer does not have a lecture planned, it goes to the bar (using the `gotobar` plan) to eat. It adds the relevant goal and returns executing `checkevents` routine.
4. If it is between 9 a.m. and 5 p.m. and no lectures are scheduled, it will go to the forum, add the relevant goal and return to the `checkevents` routine.
5. If none of the above conditions are met, the agent will return to the `checkevents` routine.
6. The last situation is to ensure that the agent will not enter an error state. This can happen if while checking which plan to follow, the system will change the current clock. The last `givelectures` plan simply forces the lecturer agent, should it reach this condition, to recheck the others.

The other plans are relatively straightforward and easy to understand.

Details about ISEnv.java

You are not expected to go over the Java code describing the environment used. This section is meant to provide you with information which you would have trouble figuring out without going in detail through ISEnv.java.

Initial Beliefs

Every agent is initialized with a set of beliefs which help it navigate the environment. These beliefs have the following form:

- pos(E,X,Y) - where E is either a building, park location or agent and X and Y are its coordinates. These beliefs are managed by the system as long as the go_to function is used.
- week(W) - where W is the current week. A separate running thread is responsible for changing all time-related percepts. Synchronization issues might sometimes result in an agent crashing, which is noticeable by an exception being thrown and displayed in the console. Attempt to run the simulation a couple more times and if the same crash occurs email me.
- day(D) - where D is the current day. Similar to the week percept, it is managed by the thread responsible with incrementing time.
- time(T) - where T is the current time of the day. Similar to the week and day percepts, it is managed by the thread responsible with incrementing time.
- home(O) - where O is the name of the building the agent considers its home. This is assigned by the system, as one of the OUT buildings.

- event(Name,Week,Day,Time,Place,Priority) - an agent will know about any event relevant to it. This relevance is given by the settings of the configuration file. If you want your agents to have lectures or assignments assigned to them, remember to make the appropriate changes in the configuration file.
- lecture(Name,Week,Day,Time,Place,Priority) - similarly, an agent will know about any lecture relevant to it.
- assignment(Course, Number, StartWeek, StartDay, StartTime, EndWeek, EndDay, EndTime, WorkHours) - agents will have knowledge of relevant assignments. They must work on them from the time described by the "Start" coordinates to the time described by the "End" coordinates, in total putting in WorkHours hours of work. For your assignments, you do not need to keep track of the number of hours to work and the number of hours worked.

Movement

Agents should move using the `go_to` function. This makes use of predetermined paths that agents should take from building to building. Agents do not know how to go to the park. Should any error occur as a result of a `go_to` command please email me.

Closing buildings

Right clicking on any building in the graphical interface will close that building. This is unrelated to your assignments, so the logic to handle closed buildings has been left out.