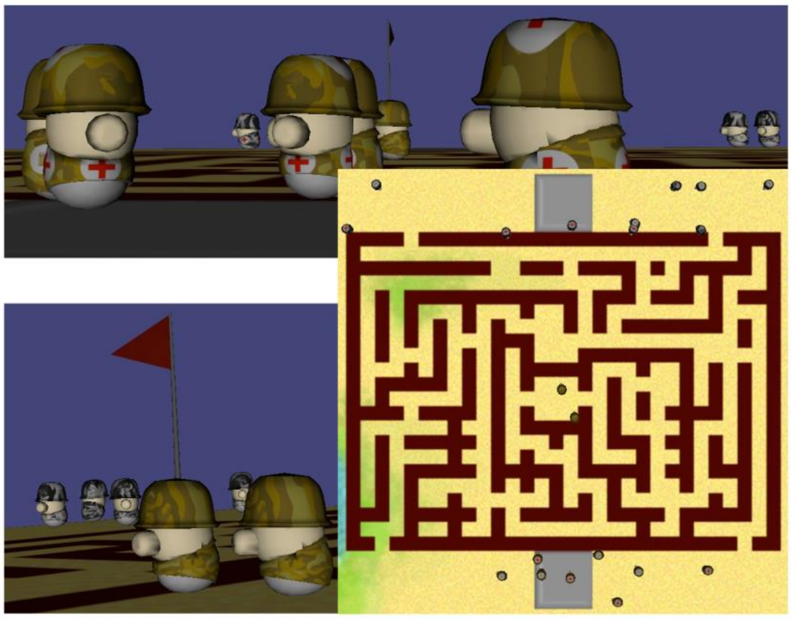
Grupo de Tecnología Informática
Inteligencia Artificial

JASON - JGOMAS

JASON-
JGOMAS
Manual



1. Introduction

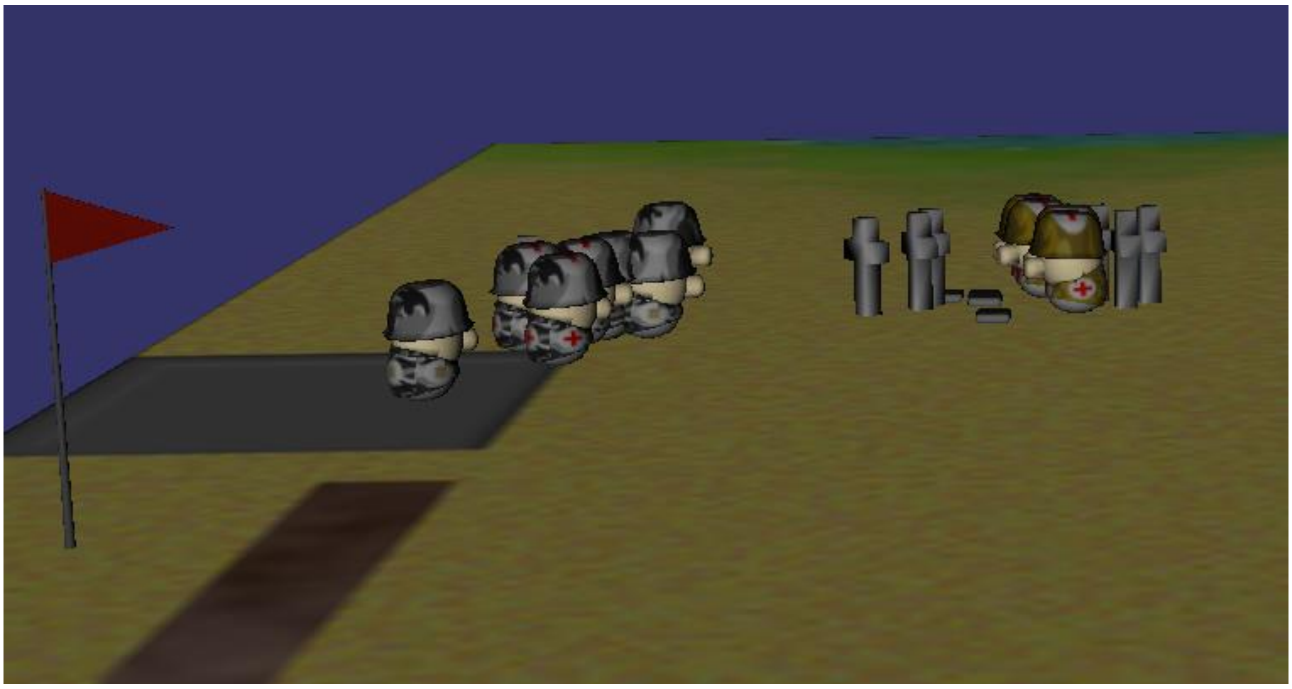
This paper presents a Multi - Agent social simulator System based on JADE and Jason that has been developed to meet different purposes. The principal is to serve as a starting point for a study of the feasibility of integration of Multi-Agent Systems (MAS) and Virtual Reality (VR). Thus, combines a 3D graphical display with a MAS which can be used in a qualitative evaluation of the current implementation of the MAS.

So, JGOMAS (Game Oriented Multi-Agent System based on JADE and Jason) is a test platform to study the integration between MAS and RV, also allowing it to be used as a validation tool for MAS.

A capture the flag kind of game has been selected as simulation of the environment in the MAS developed. In this kind of games, two teams (red and blue, allies and axis) must compete to capture the opponent's flag. This type of game has become a standard in almost all multi-player games that have have appeared since Quake game.

It is very easy and intuitive to implement MAS in such games, because every soldier can be seen as an agent. Moreover, a team of agents must cooperate to achieve the objective of the team, competing with the other team.

In our version of capture the flag, the two teams are the allies and axis. Allied agents must go to the base of the shaft, capture the flag and return to their base, in which case they will win the game. On the other hand, agents of the shaft must defend their flag and, if captured, must return it to the base. The game has a maximum time, after which, if the flag has not reached the Allied base, Axis team win the game.



2. Jason – JGOMAS Architecture

JGOMAS is primarily composed of two subsystems. On the one hand, there is a multi-agent system with two different kinds of running agents. One of these types of agents handling the current game logic, while the others belong to one of the two teams, and will be playing the full game. Actually, this subsystem is a layer that

runs on top of a platform for multi-agent specifically JADE and can take advantage of all the services provided by JADE.
Jason - JGOMAS, which is the current version of the JGOMAS platform allows the use of Jason agents to form teams of participating agents.
On the other hand, an ad-hoc graphical display (Render Engine) to display a 3D virtual environment has been developed. According to the specific requirements of (high computational cost for short periods) graphics applications, this graphic display has been designed as an external module (and not as an agent). It has been written in C++ using the OpenGL graphics library.
Figure 1 shows a JGOMAS architecture where all components and their relationships can be seen: the JADE platform as support for JGOMAS Multi-Agent System, which is comprised of agents, one of them acting as controller other agents, and as an interface for the application of graphical display.
The Multi-Agent System JGOMAS can be viewed as a kernel (basic package), which provides an interface for the graphical display to connect to the current game.

2.1. Agents Taxonomy

We can establish the following taxonomy of agents within JGOMAS according to their functionality:

- Internal agents: are those who form themselves the JGOMAS management platform. Their behaviors are pre-defined, and the user cannot change them. These agents are JADE agents, and there exist the following types:
    - Manager: This is a special agent. Its main objective is to coordinate the current game. In addition, you must answer requests from other agents. Another task that it is responsible for, is to provide an interface for graphics viewers. Therefore, any instance of the graphical display can be connected to current game and display the 3D virtual environment.
    - Pack: There are three different types of packs, the medic packs (used to give health to agents), ammo packs (used to give ammunition to agents) and objective packs, i.e. the flag to capture. All these agents are dynamically created and destroyed with the exception of the flag (there exists only one flag throughout the game and cannot be destroyed).
- External agents: They are the current players. They have a predefined set of basic behaviors that the user can modify or even add new ones. These agents must be developed in Jason. An agent can play a unique role in the current game. There are three roles defined, but the user can define new, each providing a unique service. Thus, these agents or troop agents, specialize in the following three roles:
    - Soldier: provides a *backup* service (the agent goes to help to his team-mates).
    - Medic: provides a *medic* service (the agent goes to give *medic packs*).
    - FieldOps: provides an *ammo* service (the agent goes to give *ammo packs*).

A domain like capture the flag enables a simple and entertaining way to establish a testbed either for algorithms and optimizations individually on each agent and for competitive and cooperative strategies between teams going.
External agents are integrated into the virtual environment, allowing the interaction between them (through the perception of peers / nearby enemies) which can lead to cooperation / coordination with partners of the same team. Also, being located in this virtual environment, must take into account the characteristics of the terrain in which they are located (difficulty moving, walls ...).

All communication that takes place between the agents that make up the platform is performed by passing messages according to protocols established by the FIPA ACL.

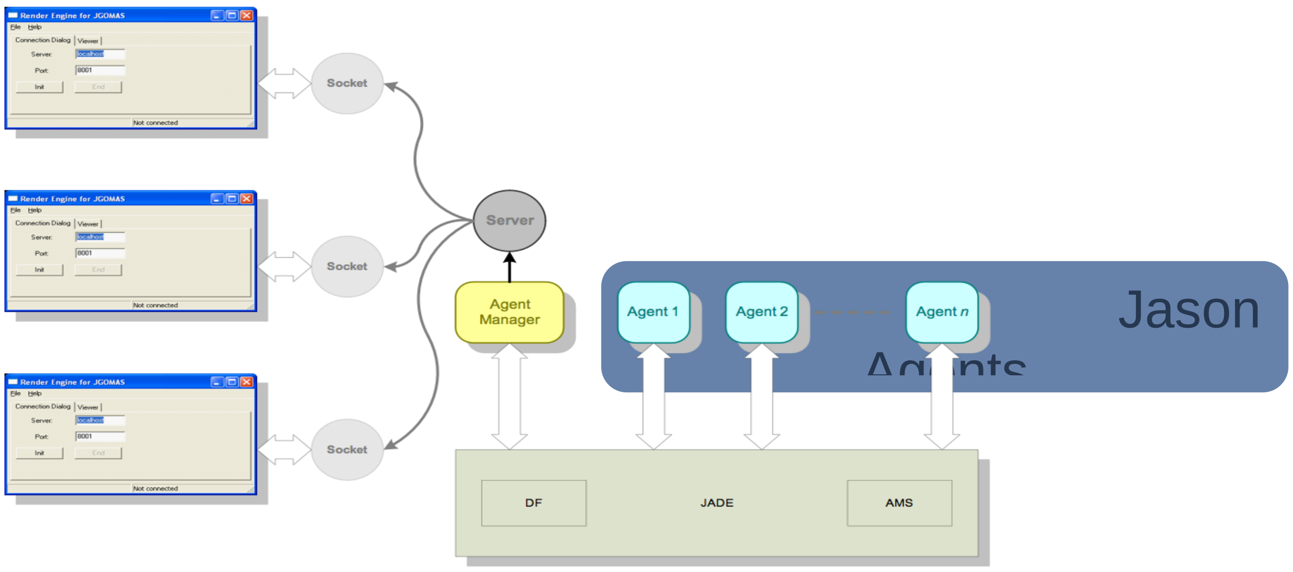


Figure 1: Jason-JGOMAS Architecture

2.2. Maps

JGOMAS can use different maps to define your virtual environment. These maps, by default, are of size 256 x 256, so that the position of the agents is given by its coordinates (x, y, z), where x, z take values between 0 and 255, while in the maps supplied, y is always equal to 0 (these maps do not have height).

Each agent has partial access to the map where the game is played, since despite having access to static information about it, can only perceive objects that are at a distance (within the "cone of vision").

The maps are stored in the **bin\data\maps** folder of the distribution. In this folder there is a subfolder for each map with the name **map_XX**, where **XX** is the number of the map. In this folder there are several files that define the map. For example, the content of the **map_04** folder is as follows:

- map_04_cost.txt: This file defines the walls of the map by means of * to indicate it. The content of this file is:

```
********************************
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                         **  **
*                         *    *
*               *         *    *
*               *         *    *
*               *         *    *
*             ********   **    *
*             *      *****     *
*             *          *     *
*   ***  ******          *     *
*             *                *
*                              *
*                        *     *
********************************
```

- map_04_terrain.bmp: This file defines the artistic look of the map, as you can see in the image on the left of Figure 2.
- map_04_cost.bmp: This file defines the walls of the map using a black and white image, where black represents the wall, as you can see in the image on the right of Figure 2.

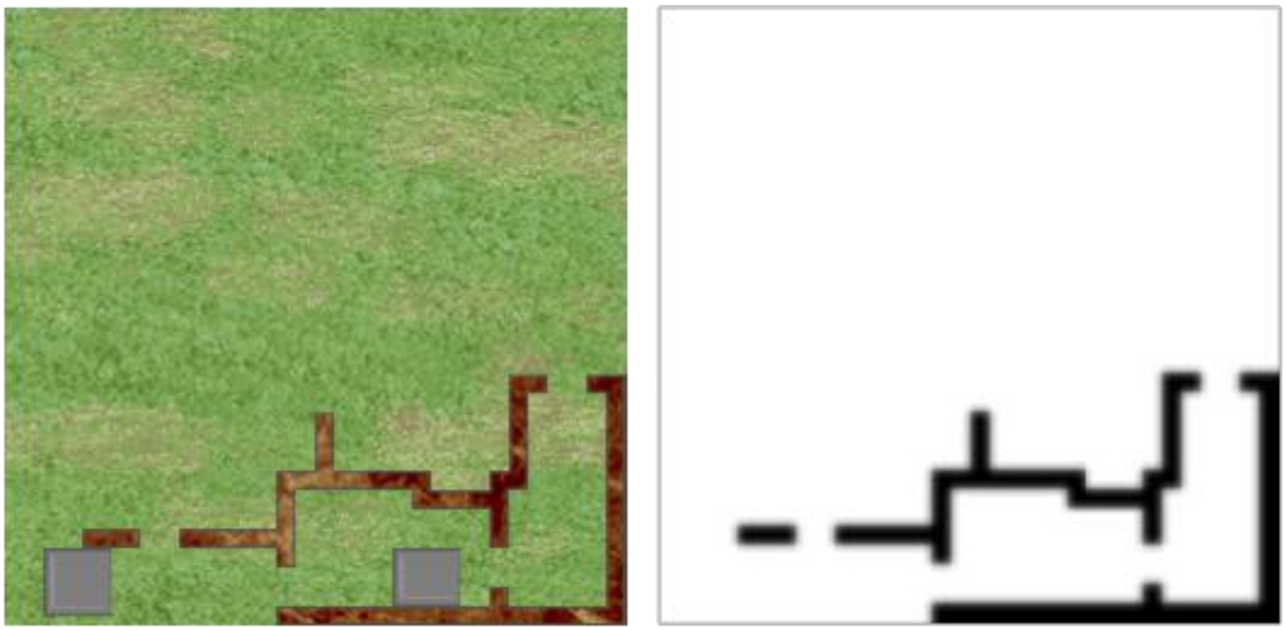


Figure 2: Left: map_04_terrain.bmp - Right: map_04_cost.bmp

- map_04.txt: This file contains the definition of different settings for the MAS and graphic display:
  - JADE_OBJECTIVE: Flag initial location.
  - JADE_SPAWN_ALLIED: Allied base location.
  - JADE_SPAWN_AXIS: Axis base location.
  - JADE_COST_MAP: Size and name of the cost file.
  - RENDER_ART_MAP: Size and name of the file containing the artistic look of the map.
  - RENDER_COST_MAP: Size and name of the file containing the artistic look of the map costs.
  - RENDER_HEIGHT_MAP: Size and name of the file containing the artistic look of the map heights.

  The content of this file is the following:

```
[JADE]
JADE_OBJECTIVE: 28 28
JADE_SPAWN_ALLIED: 2 28 4 30
JADE_SPAWN_AXIS: 20 28 22 30
JADE_COST_MAP: 32 32 map_04_cost.txt
[JADE]

[RENDER]
```

```
RENDER_ART_MAP: 256 256 map_04_terrain.bmp
RENDER_COST_MAP: 32 32 map_04_cost.bmp
RENDER_HEIGHT_MAP: 32 32 map_04_heightmap.bmp
[RENDER]
```

3. Tasks

A task is something that an agent has to perform in a particular position of the virtual environment. There are various types of tasks, which are according to the different actions that an agent can perform in the virtual environment, being the main ones:

- `TASK_GIVE_MEDICPAKS`: A medic must generate packets of medicine in a particular place (the position of the agent who requested it and who has agreed to go to give them).
- `TASK_GIVE_AMMOPAKS`: A fieldops must generate packets of ammo in a particular place (the position of the agent who requested it and who has agreed to go to give them).
- `TASK_GIVE_BACKUP`: A soldier should go to help a teammate to a particular place (the position of the agent who requested it and who has agreed to give it go).
- `TASK_GET_OBJECTIVE`: The agent, from the attacking team or ALLIED, must go to the starting position of the flag for it. If he manage to grab the flag, this task becomes going back to their home base.
- `TASK_GOTO_POSITION`: The agent must go to a specific location.

A task is associated in addition to its type, the agent that causes the task (the agent itself or the agent who requested that he should give something like life …), the position where it should be performed, priority and any possible contents additional. Always the highest priority task is launched. You can redefine the priority of each type of task.

The tasks the system is implemented, not the user, it can only add tasks to the list of active tasks the agent:

- JASON: a plan is used to add a task: **`add_task`**

  ```
  !add_task(task(TaskPriority, TaskType, Agent, Position, Content))
  !add_task(task(TaskType, Agent, Position, Content))
  ```

  Examples (two options):

  ```
  !add_task(task(1000, "TASK_GET_OBJECTIVE", M, pos(ObjectiveX, ObjectiveY, ObjectiveZ),
  ""));
  !add_task(task("TASK_GET_OBJECTIVE", M, pos(ObjectiveX, ObjectiveY, ObjectiveZ), “"));
  ```

  Such objectives trigger the plan creating the task. The second one assigns the priority defined by the agent.

4. Execution Loop

- Each JGOMAS external agent executes a FSM as the one in Figure 3:
  - `STANDING`: The agent does not have any triggered task.
  - `GO_TO_TARGET`: The agent has triggered a task and he is moving to the position where he has to do it.
  - `TARGET_REACHED`: The agent has reached the position where he has to do the triggered task, and he is doing the actions indicated in the task.

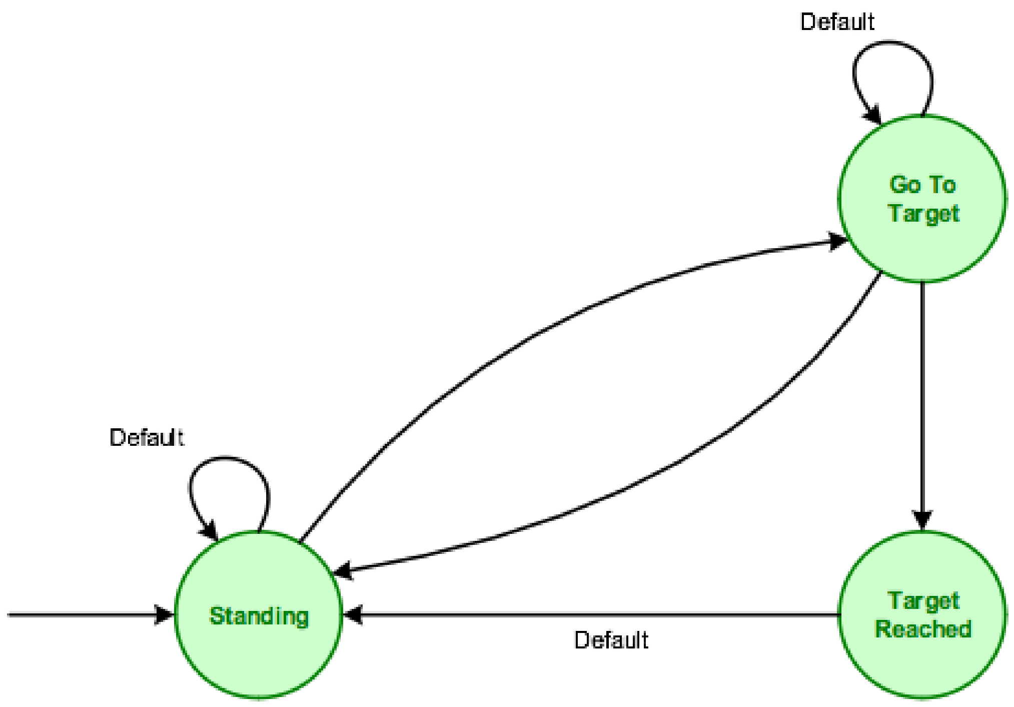


Figure 3: FSM defining JGOMAS external agents behaviour.

5. Interface (API)

The interface (API) for working with jGomas is composed of .asl files written in Jason that contain the different agents (each one with its own beliefs, goals, and behaviors). Inside these files, jgomas.asl includes the non-modifiable behaviors of the agent. To modify the behavior of the agent the files with a name following the pattern jasonAgent_TEAM_TYPE.asl have to be modified, where TEAM refers to the side of the agent (ALLIED, AXIS), and TYPE is the type of the agent (soldier, MEDIC, FIELDOPS). Therefore, there are 6 files covering all possible types of agents:

- `jasonAgent_ALLIED.asl`
- `jasonAgent_ALLIED_MEDIC.asl`
- `jasonAgent_ALLIED_FIELDOPS.asl`
- `jasonAgent_AXIS.asl`
- `jasonAgent_AXIS_MEDIC.asl`
- `jasonAgent_AXIS_FIELDOPS.asl`

Regarding agent beliefs, the main ones are:

- **`tasks(task_list)`:** Contains the list of active tasks of the agent. Ex:

```
tasks([task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),""),task(1001,"TASK_WALKING_PATH",
"A2",pos(204,0,228),"")])
```

In this example the list has two active tasks:

- `task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),"")` that indicates that has to get the objective (the flag) that is in the position pos(224,0,224). The priority assigned to this task is 1000.
- `task(2000,"TASK_GIVE_MEDIPAKS","A2",pos(204,0,228),"")` that indicates that has to go to the position pos(204,0,228) where the agent A2 is waiting for medicine packs (the agent with this active task is an agent). The priority assigned to this task is 2000.

- **`fovObjects(object_list)`:** Contains the list of objects currently seen by the agent. The structure of an object is `[#, TEAM, TYPE, ANGLE, DISTANCE, HEALTH, POSITION]`.

Example: [1,200,1,0.58,14.76,78,pos(214,0,219)], the object 1 of the team 200 (AXIS), type 1 (agent), with angle 0,58, with a distance of 14,76, a health 78 and its position is pos(214,0,219)
Note: The TEAM values are: 100 (Allied), 200 (Axis), 1003 (flag)

- **`state(current state)`:** This belief is used to indicate the state of the agent in his state machine: standing, selecting which task to do or waiting; go_to_target, going to its next target; target_reached, it has reached the target; quit, has to finish.

- **`my_health(X)`:** Stores the health of the agent. The initial, and maximum, value is 100. When this value reaches 0, the agent dies.
- **`my_ammo(X)`:** Stores the amount of bullets of the agent. The initial value is 100.
- **`my_position(X,Y,Z)`:** Stores the last position known by the agent.

The different modifiable plans of the agents are:

- **`!perform_look_action`**
  This object is invoked when the agent looks around and update the list of surrounding objects `fovObjects(L)`. It would be necessary to implement the plan associated to the creation of this event to be able look what is around.
- **`!perform_aim_action`**
  This objective is triggered if there is an enemy to aim, which can be used to take a decision about what to do with the aimed agent. An easy implementation of the plan is available. It is interesting to improve this plan to take a more refined decision about who to aim.
- **`!get_agent_to_aim`**
  This objective is invoked after `!perform_look_action`, and it would be used to decide if there is any enemy to aim. An easy implementation of the associated plan is available. It is interesting to improve the mentioned associated plan to take a more refined solution about who to aim.
- **`!perform_no_ammo_action`**
  This objective is triggered when the agent shoots and has no ammo. It is necessary to implement the associated plan to take a decision. For example, to run.
- **`!perform_injury_action`**
  This objective is triggered when the agent is shot. It is necessary to implement the plan associated to the creation of this event to take a decision. For example, run if the agent has a low life value.
- **`!performThresholdAction`**
  This objective is triggered when the agent has less life or bullets than the thresholds `my_ammo_threshold(X)` and `my_health_threshold(X)`. An easy implementation of the associated plan is available, which asks for help to medics or fieldops of its team. It is interesting to improve the plan to take a more refined solution.
- **`!setup_priorities`**
  This objective is triggered during agent initialization to fix agent task priorities. Each agent has its own priorities. A simple implementation is available. It is interesting to modify it to add new tasks or modify the priorities to have agents that behave in a different way.

- **`!update_targets`**
  This objective can be used to update the tasks and its priorities. It is invoked when the agent changes to the standing state and has to choose a new task among the available ones. It is necessary to implement the plan associated to the creation of this event.

6. Communication

It is necessary to highlight that an agent plays only one role during the whole game, if an agent A1 is created as a soldier of the type Soldier and belong to the AXIS side, it will be for the rest of the game.

Each role has different features and offers specific basic services that can be improved. Indicating the role and the basic services offered by an agent is carried out at the initialization, using a process known as Registration. Nevertheless, an agent can add new services during the development of the game.

6.1. Registration

A role has to register a service to allow the other role to request it. Registration is carried out using the internal function: **`.register( "JGOMAS", "type")`**

In this example, it is registered the service "`type`" by the agent who invoked the function. The registration is used to register in the DF a service of the type "type" by the agent who executed it.

Ex: `.register( "JGOMAS", "medic_AXIS");` registers the service "`medic_AXIS`" into the DF.

Exists a set of existing services depending on the role played by the soldier agent:

- In the case of ALLIED agents, they register in a transparent way that they belong to this side so the rest of agents of their team may know about that.
  ```
  .register( "TEAM", "ALLIED");
  ```
  In the case of allied soldier agents, it is registered the service "`backup_ALLIED`":
  ```
  .register( "JGOMAS", "backup_ALLIED");
  ```
  In the case of allied medic agents, it is registered the service "`medic_ALLIED`":
  ```
  .register( "JGOMAS", "medic_ALLIED");
  ```
  In the case of allied fieldops agents, it is registered the service "`fieldops_ALLIED`":
  ```
  .register( "JGOMAS", "fieldops_ALLIED");
  ```
- In the case of AXIS agents, they register in a transparent way so the rest of players in their team may know.
  ```
  .register( "TEAM", "AXIS");
  ```
  In the case of axis soldier agents, it is registered the service "`backup_AXIS`":
  ```
  .register( "JGOMAS", "backup_AXIS");
  ```

In the case of axis medic agents, it is registered the service "`medic_AXIS`":

```
.register( "JGOMAS", "medic_AXIS");
```

In the case of axis fieldops agents, it is registered the service "`fieldops_AXIS`":

```
.register( "JGOMAS", "fieldops_AXIS");
```

Once an agent has registered a service, it is posible for an agent of its same team to check which agents of its team offer a specific service.

How to know what services are available from an agent?

The following internal action is executed **`.my_team(type,list)`**

This action returns the list of agents of the team of the invoker agent that are available through the yellow pages (DF) from the type "`type`" (excluding itself).

In the example: `.my_team("medic_AXIS", E)`
Supposing that is executed by an agent of the "AXIS" team, it would return the list E that contains the alive doctors of the team.

In this way it is posible, for example, to know the name of the agents of the team. In order to do so, this internal function has to be invoked:

```
.my_team("AXIS", E);   ó
.my_team("ALLIED", E);
```

Then, using the list E, the agent can carry out the actions it considers as adequate.

Usage example

```
…
.my_team("AXIS", E1);
.my_name(Me);
.println("My team is: ", E1, " and I am: ", Me );
.length(E1, X);
 if (X==0) {  .println("I am alone");  }
```

In this case an agent checks the agents of its team. If the list is empty, it means that it is alone.

6.2. Coordination

JGOMAS is provided with mechanisms that allow agent coordination. It can be of one of these two types:

- No communication (implicit): It is achieved by sensing the environment. When an agent looks around itself the objective `!perform_look_action` is triggered. By rewriting the associated plan it can be decided what to do according to the perception.
- With communication (explicit): In this case it is used the message passing using the internal action `.send_msg_with_conversation_id`

```
.send_msg_with_conversation_id (Rec, Perf, Cont, ConvId)
```

where:

- Rec → receiver of the message (could be a list)
- Perf → performative (tell, untell, achieve…)
- Cont → content
- ConvId → Conversation Id (used in Jade)

Usage example: A1 wants to send a message to its team telling them to go to (to help, coordinate, regroup…)

The code would be the following:

```
…
?my_position(X,Y,Z);
.my_team("AXIS", E1);
.concat("goto(",X, ", ", Y, ", ", Z, ")", Content1);
.send_msg_with_conversation_id(E1, tell, Content1, "INT");
```

NOTE: "INT" is a made up conversation identifier, and its usage is recommended because there exist certain internal identifiers that may lead to an undesired agent behavior is they are used.

Following the same example, the rest of agents of the team will have a plan with this shape:

```
+goto(X,Y,Z)[source(A)]
 <-
    .println("Received a message of the type goto from ", A);
 .
```

In this case the agent that receives the message only prints the received message, but a more complex action may be carried out, such as change the task to make by the agent that receives the message. In this case, the code would be:

```
+goto(X,Y,Z)[source(A)]
 <-
    .println("Received a goto message from ", A);
    !add_task(task("TASK_GOTO_POSITION", A, pos(X, Y, Z), ""));
    -+state(standing);
    -goto(_,_,_) .
```

7. Appendix: Jason Eclipse Plugin

7.1. Instalation

In order to install the Jason plug-in for Eclipse you should follow the steps below and have Eclipse version 3.7.0 (Indigo) or greater.

Step 1

Download the latest version of Jason at the link: http://sourceforge.net/projects/jason/files/

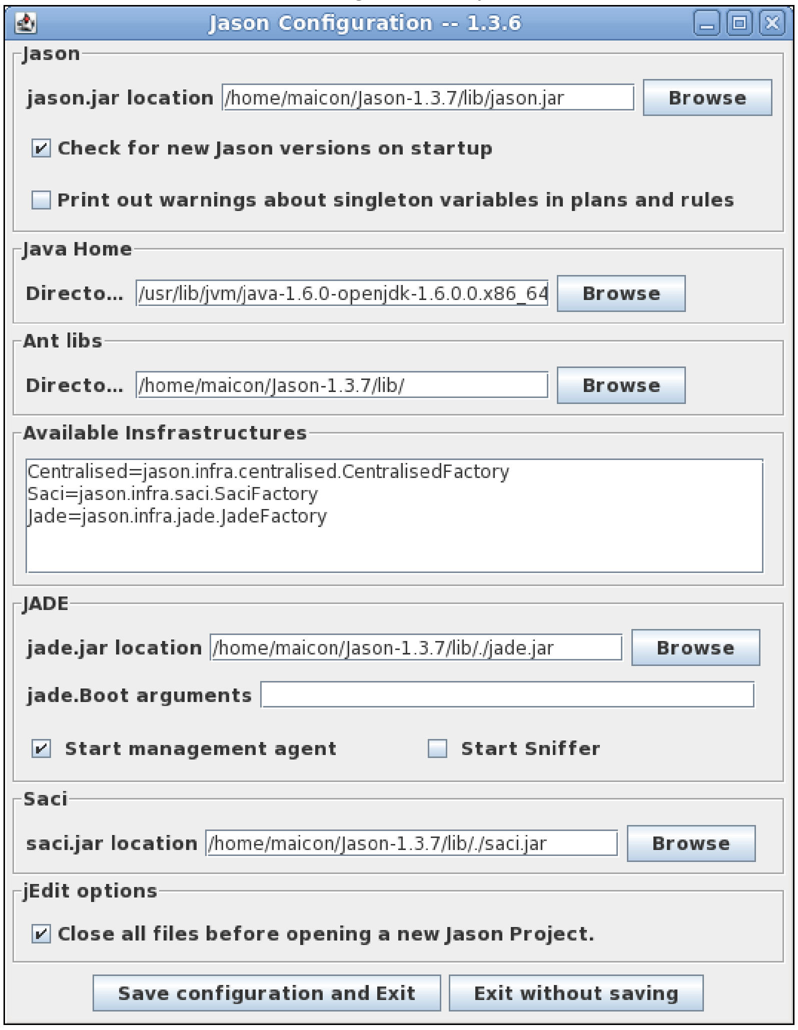
Step 2

After the download, unpack it in any directory of your machine.

Step 3

If you had never run Jason on your computer, execute the file "lib/jason.jar" by double clicking over it. You also could execute this file by with the following command: `java -jar lib/jason.jar`

The following figure shows the window that you have to see after you run the file jason.jar. Make sure about the directories of the libs. We suggest you only change the "Java Home" directory. The others are automatically filled out, but feel free to change them if you want to.


Jason Configuration -- 1.3.6

Jason
jason.jar location: /home/maicon/Jason-1.3.7/lib/jason.jar
[✓] Check for new Jason versions on startup
[ ] Print out warnings about singleton variables in plans and rules

Java Home
Directo...: /usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64

Ant libs
Directo...: /home/maicon/Jason-1.3.7/lib/

Available Insfrastructures
```
Centralised=jason.infra.centralised.CentralisedFactory
Saci=jason.infra.saci.SaciFactory
Jade=jason.infra.jade.JadeFactory
```

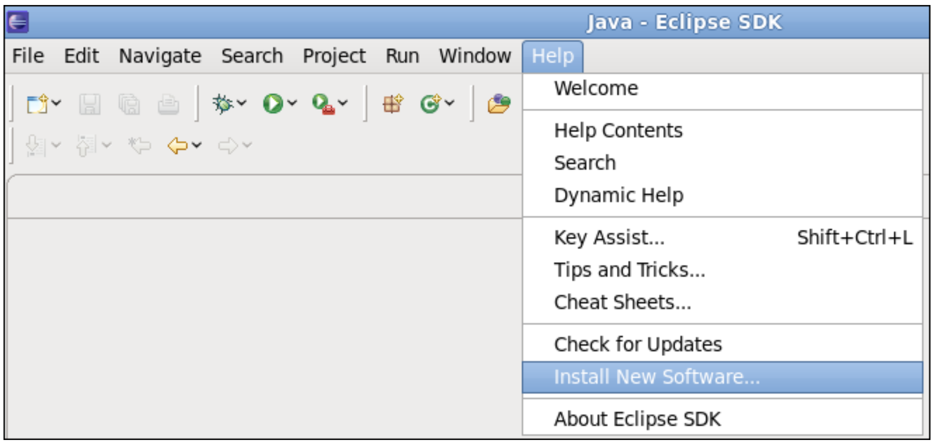
JADE
jade.jar location: /home/maicon/Jason-1.3.7/lib/./jade.jar
jade.Boot arguments:
[✓] Start management agent   [ ] Start Sniffer

Saci
saci.jar location: /home/maicon/Jason-1.3.7/lib/./saci.jar

jEdit options
[✓] Close all files before opening a new Jason Project.

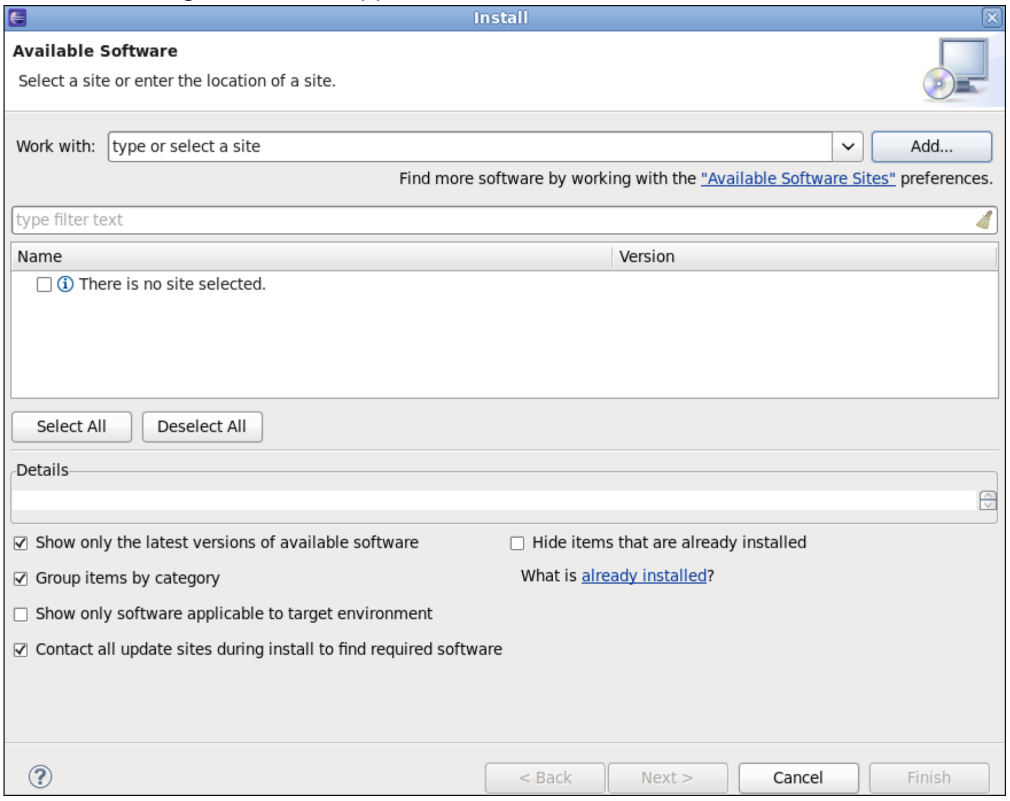
Step 4

Finally you could install the Jason plugin for Eclipse opening the Eclipse platform and going to the option "Install New Software..." at the "Help" menu:



Step 5

So, the following window will appear.


Install
Available Software
Select a site or enter the location of a site.
Work with: type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
There is no site selected.
Select All
Deselect All
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Cancel
Finish

Step 6

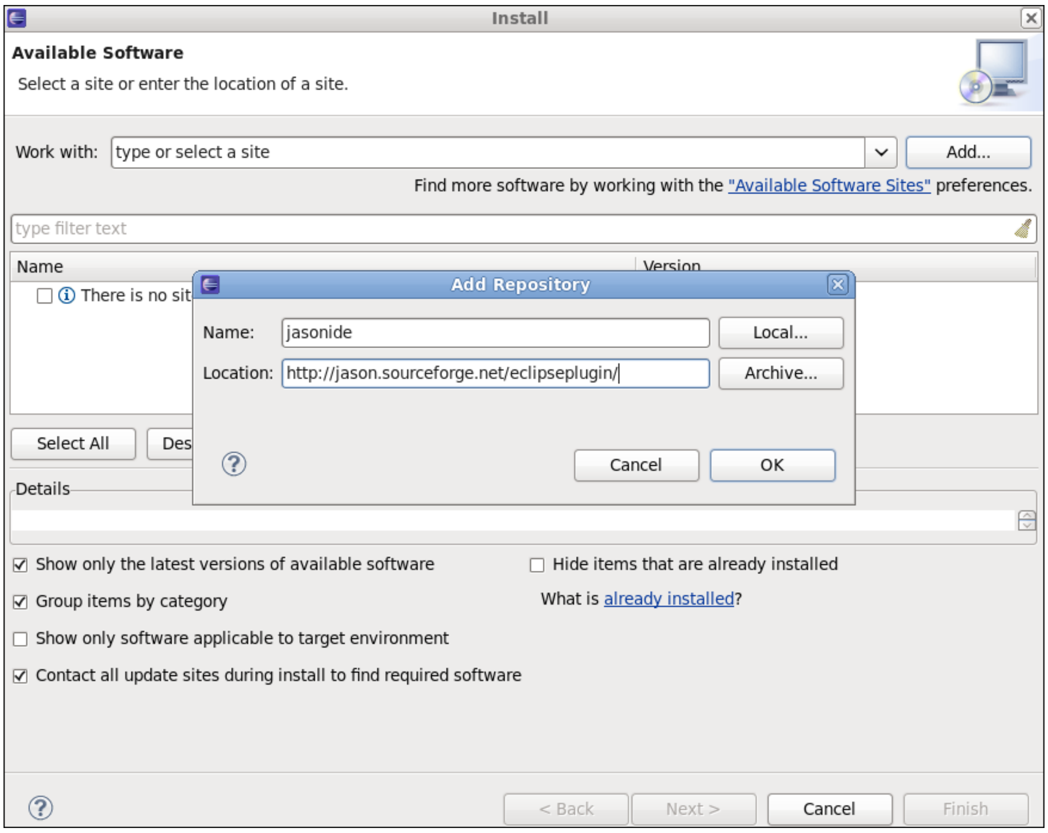
Click over the "Add" button, and fill out the form as shown in the next figure. The parameters are

Name: jasonide

Location (Eclipse Indigo): http://jason.sourceforge.net/eclipseplugin/

Location (Eclipse Juno/Kepler): http://jason.sourceforge.net/eclipseplugin/juno/

To finish, click on the "OK" button.



Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site | Add...

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name | Version

☐ There is no sit

Select All | Des

Details

Add Repository

Name: jasonide | Local...

Location: http://jason.sourceforge.net/eclipseplugin/ | Archive...

Cancel | OK

☑ Show only the latest versions of available software

☐ Hide items that are already installed

☑ Group items by category

What is already installed?

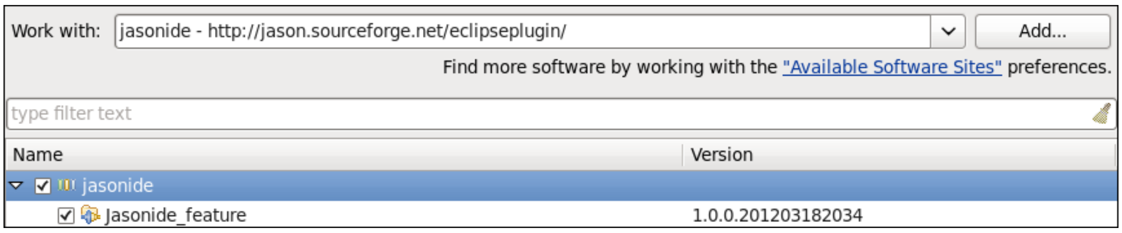
☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

< Back | Next > | Cancel | Finish

Step 7

Tick the option "jasonide" and then press the "next" button. So, you have to wait a moment while Eclipse search the dependences.



Work with: jasonide - http://jason.sourceforge.net/eclipseplugin/ | Add...

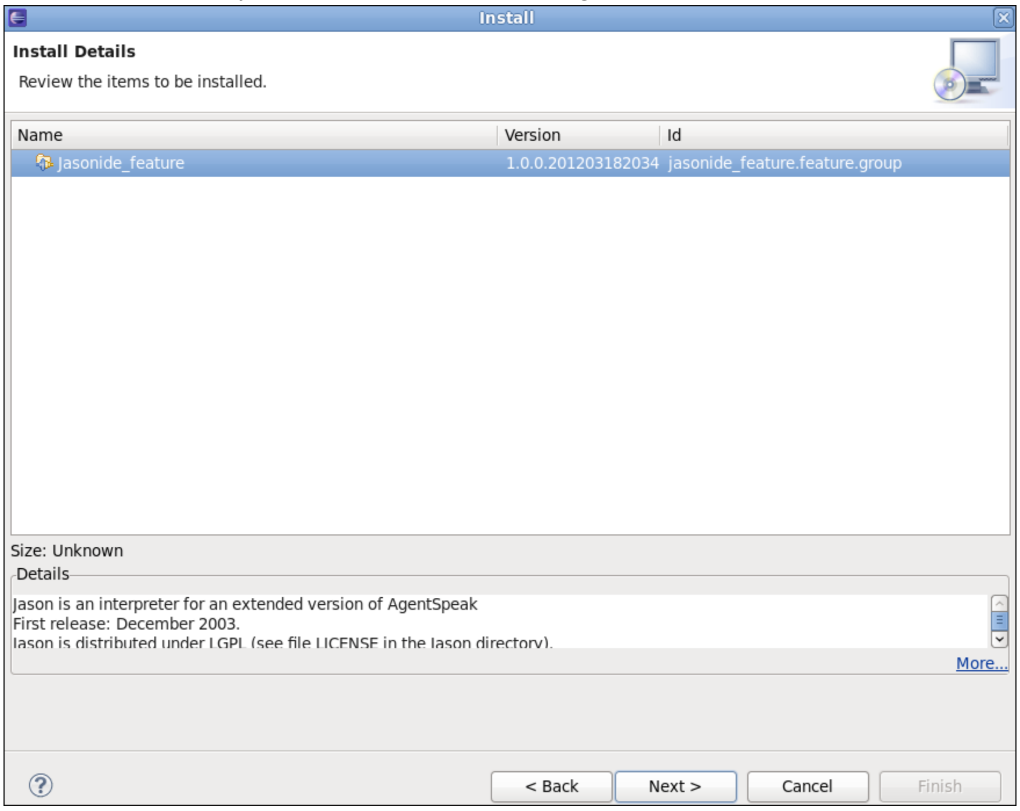
Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☑ jasonide	
☑ Jasonide_feature	1.0.0.201203182034

Step 8

In the next windows just press the "next" button again.



Install

Install Details

Review the items to be installed.

Name	Version	Id
Jasonide_feature	1.0.0.201203182034	jasonide_feature.feature.group

Size: Unknown

Details

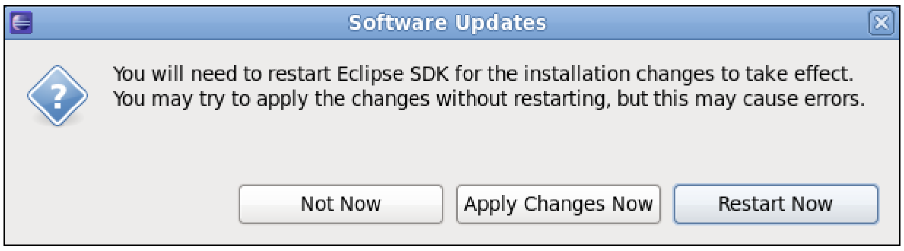
Jason is an interpreter for an extended version of AgentSpeak
First release: December 2003.
Jason is distributed under LGPL (see file LICENSE in the Jason directory).

Step 9

The last window that will be shown for you is about the license. Tick the option "I accept the terms of the license agreements" and then press the "finish" button. Then the installation is proceeded, it could take several minutes, so please wait.

Step 10

In the end of these process will be shown a window in order to complete the installation. Choose the option "Restart Now".

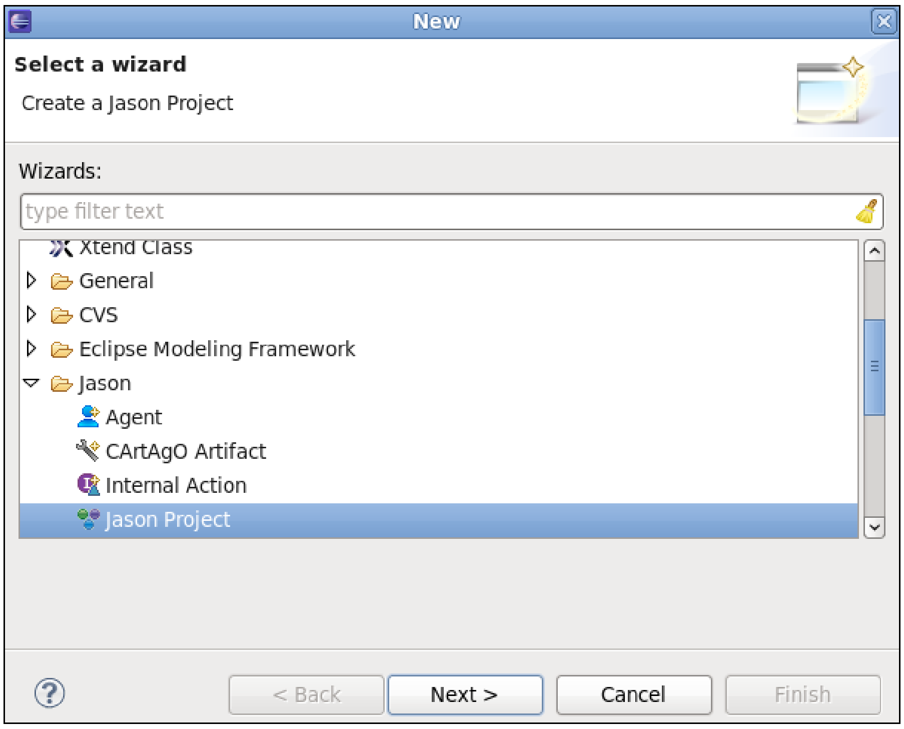


Software Updates

You will need to restart Eclipse SDK for the installation changes to take effect.
You may try to apply the changes without restarting, but this may cause errors.

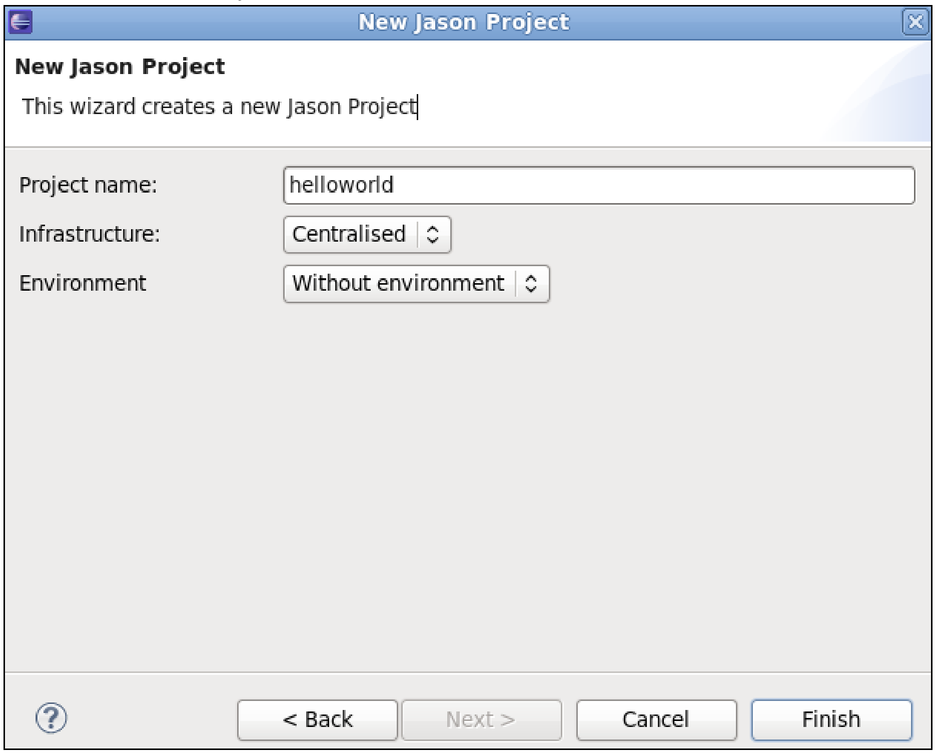
Step 11

Now you have all set. In order to test the installation of the plug-in, we suggest the creation of a simple hello world project. You could do it in the menu (File > New > Jason Project) or (File > New > Other > Jason > Jason Project).



Step 12

Fill out the field "Project name" and press the "Finish" button.



This wizard creates a new Jason Project

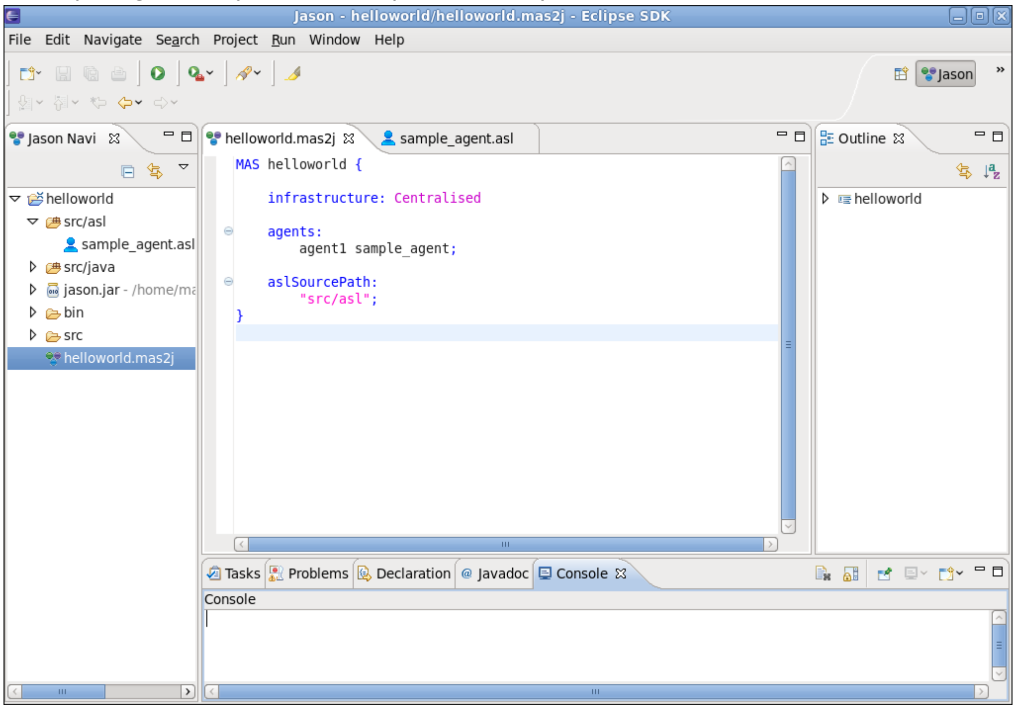
Project name: helloworld

Infrastructure: Centralised

Environment: Without environment

Step 13

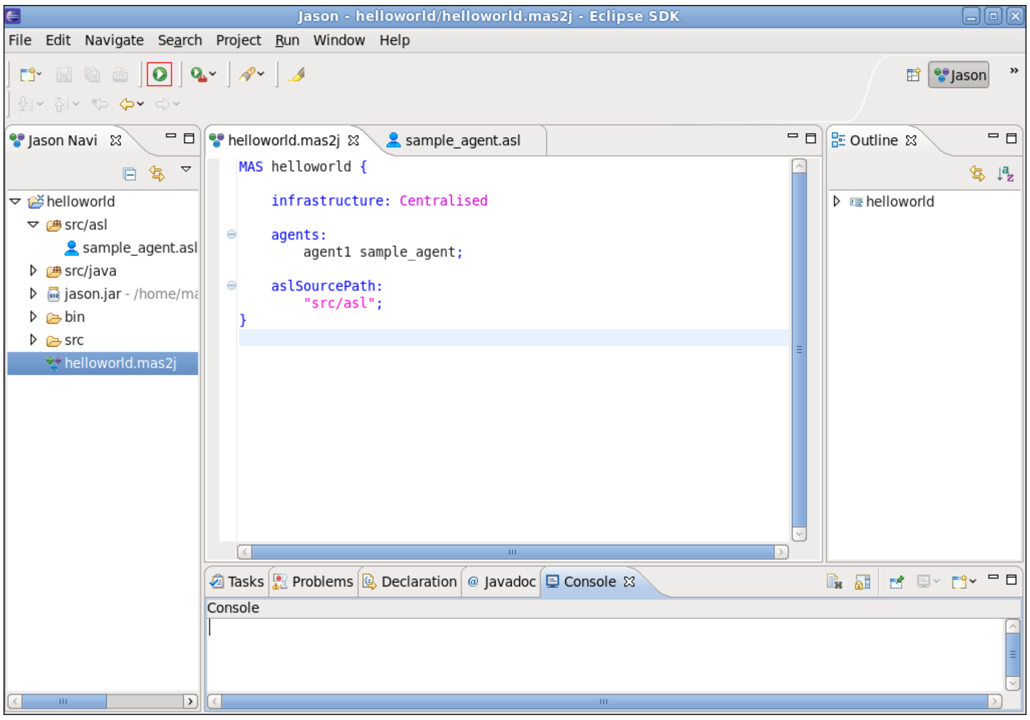
If everything is fine, you will have your first project created!



```
MAS helloworld {

    infrastructure: Centralised

    agents:
        agent1 sample_agent;

    aslSourcePath:
        "src/asl";
}
```

Step 14

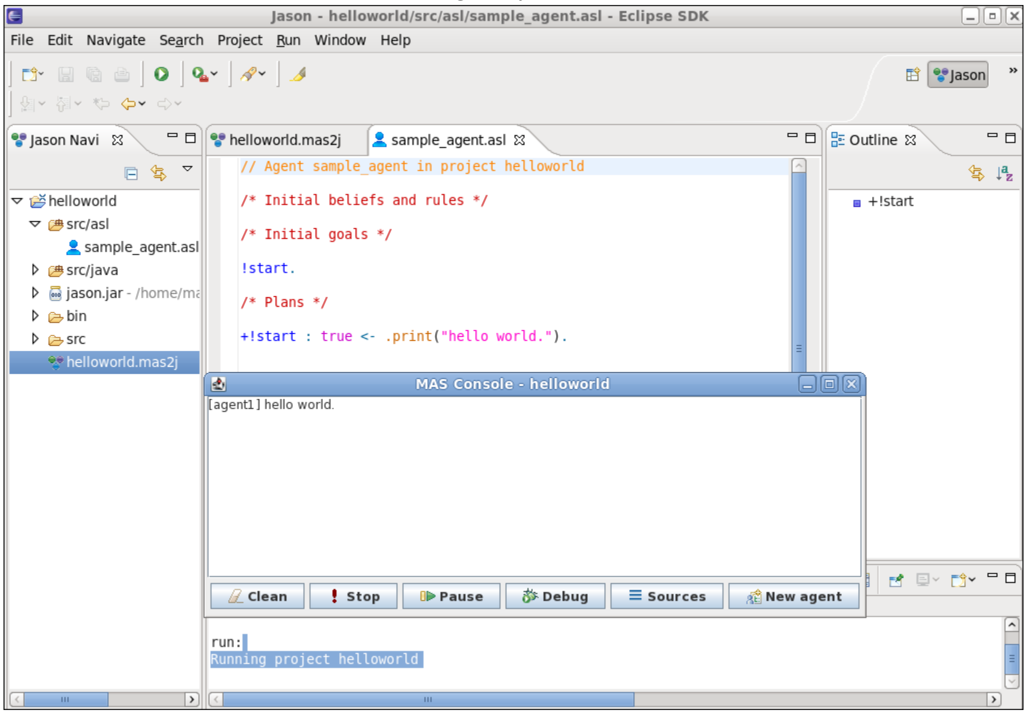
Now you can run the application by pressing the Run button.



```
MAS helloworld {

    infrastructure: Centralised

    agents:
        agent1 sample_agent;

    aslSourcePath:
        "src/asl";
}
```

Step 15

The result will be a "hello world" message in your screen.



```
// Agent sample_agent in project helloworld

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start : true <- .print("hello world.").
```

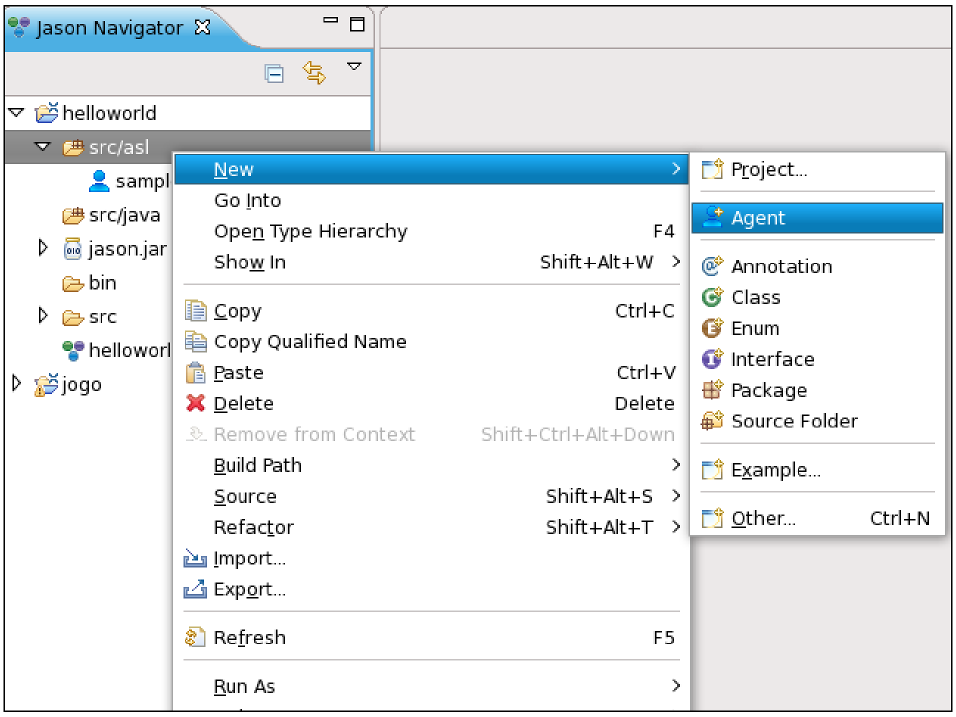
```
[agent1] hello world.
```

```
run:
Running project helloworld
```

7.2. How to créate an agent

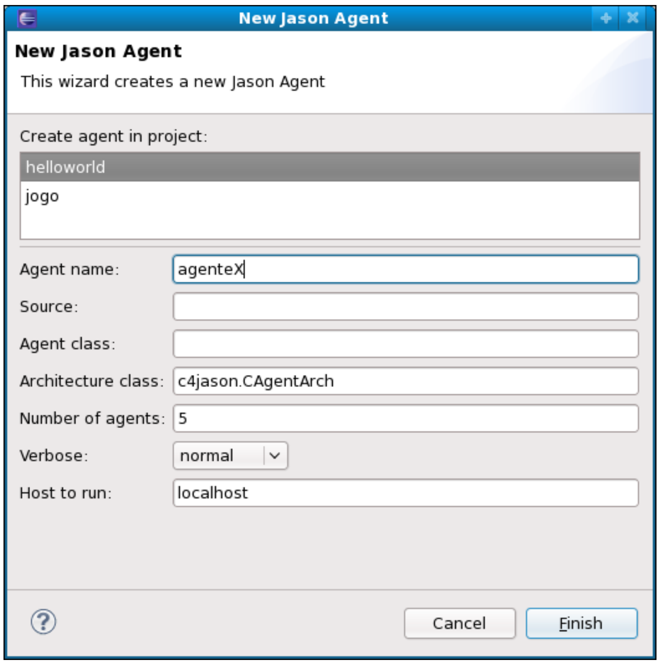
Step 1

Click on the source folder named "src/asl" using the right button and go to the option New > Agent.



Step 2

Fill out the form. The only required field is the name of the agent. After it, press the "Finish" button.



New Jason Agent

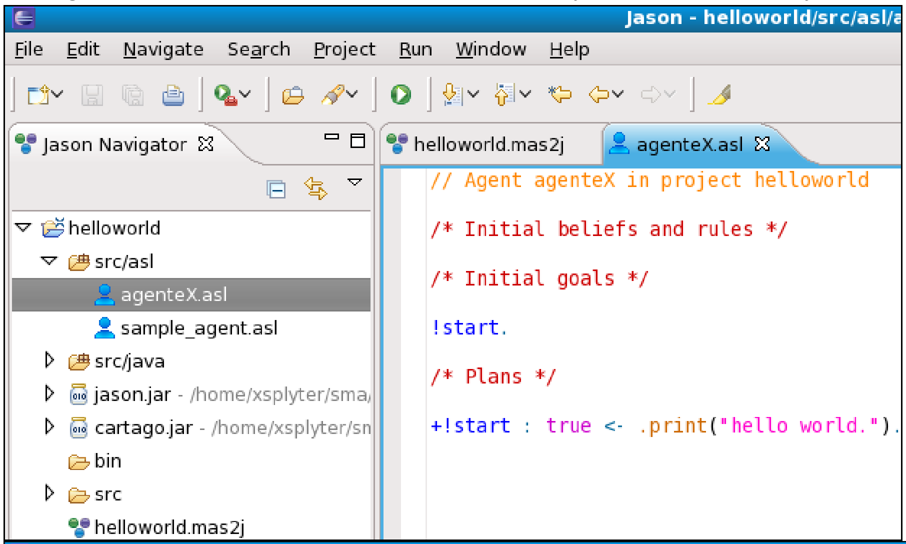
This wizard creates a new Jason Agent

Create agent in project:
- helloworld
- jogo

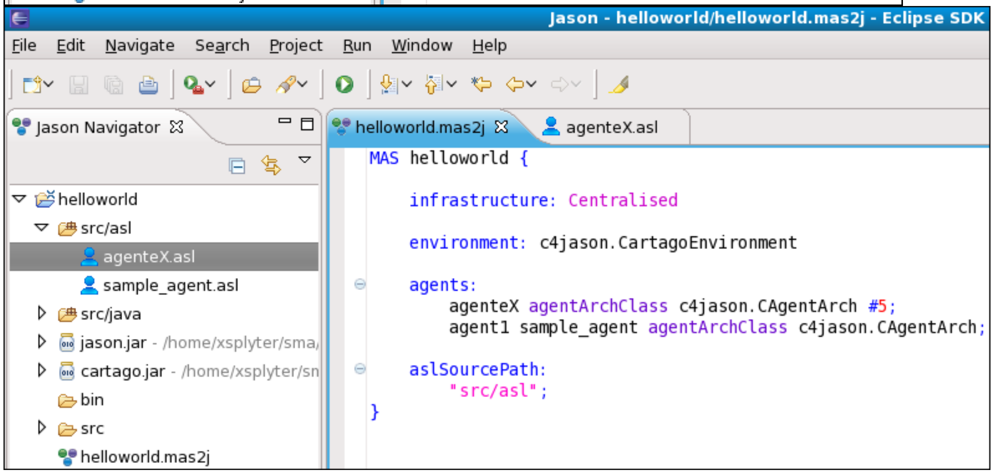
Agent name:	agenteX
Source:	
Agent class:	
Architecture class:	c4jason.CAgentArch
Number of agents:	5
Verbose:	normal
Host to run:	localhost

Step 3

The agent will be created and will be automatically added in the mas2j file.


Jason - helloworld/src/asl/a
File Edit Navigate Search Project Run Window Help
Jason Navigator
helloworld.mas2j
agenteX.asl
helloworld
src/asl
agenteX.asl
sample_agent.asl
src/java
jason.jar - /home/xsplyter/sma/
cartago.jar - /home/xsplyter/sn
bin
src
helloworld.mas2j

```
// Agent agenteX in project helloworld

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start : true <- .print("hello world.").
```

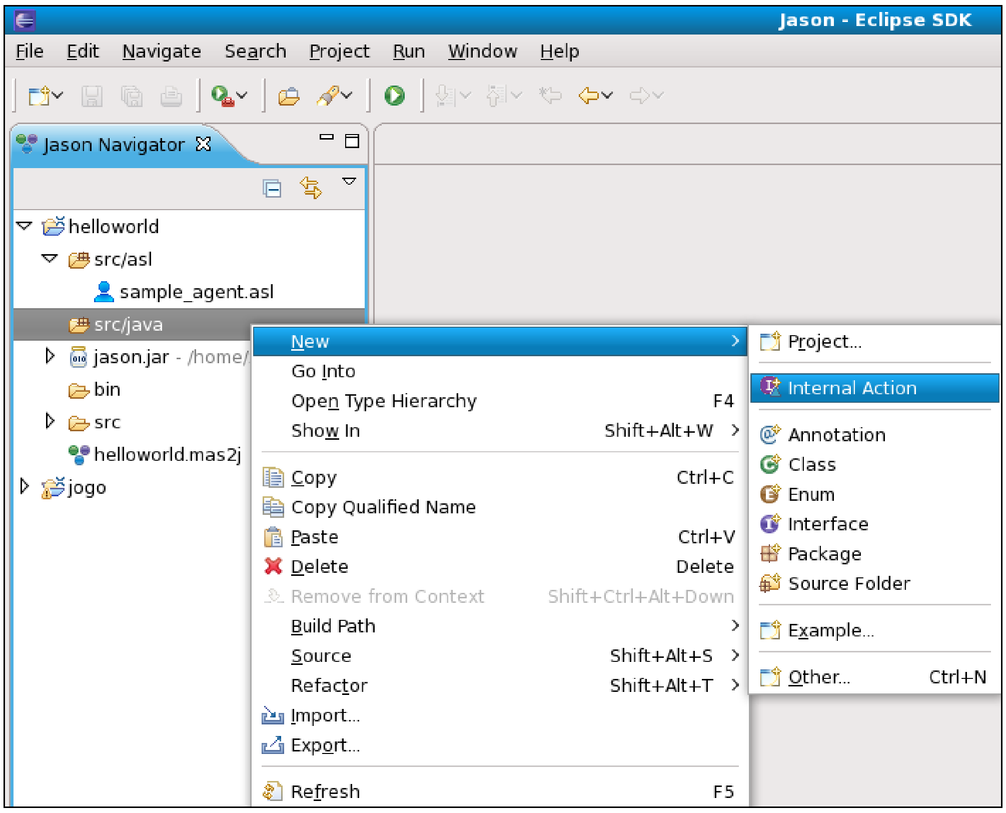

Jason - helloworld/helloworld.mas2j - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Jason Navigator
helloworld.mas2j
agenteX.asl
helloworld
src/asl
agenteX.asl
sample_agent.asl
src/java
jason.jar - /home/xsplyter/sma/
cartago.jar - /home/xsplyter/sn
bin
src
helloworld.mas2j

```
MAS helloworld {

    infrastructure: Centralised

    environment: c4jason.CartagoEnvironment

    agents:
        agenteX agentArchClass c4jason.CAgentArch #5;
        agent1 sample_agent agentArchClass c4jason.CAgentArch;

    aslSourcePath:
        "src/asl";
}
```

7.3. How to create an internal action

Step 1

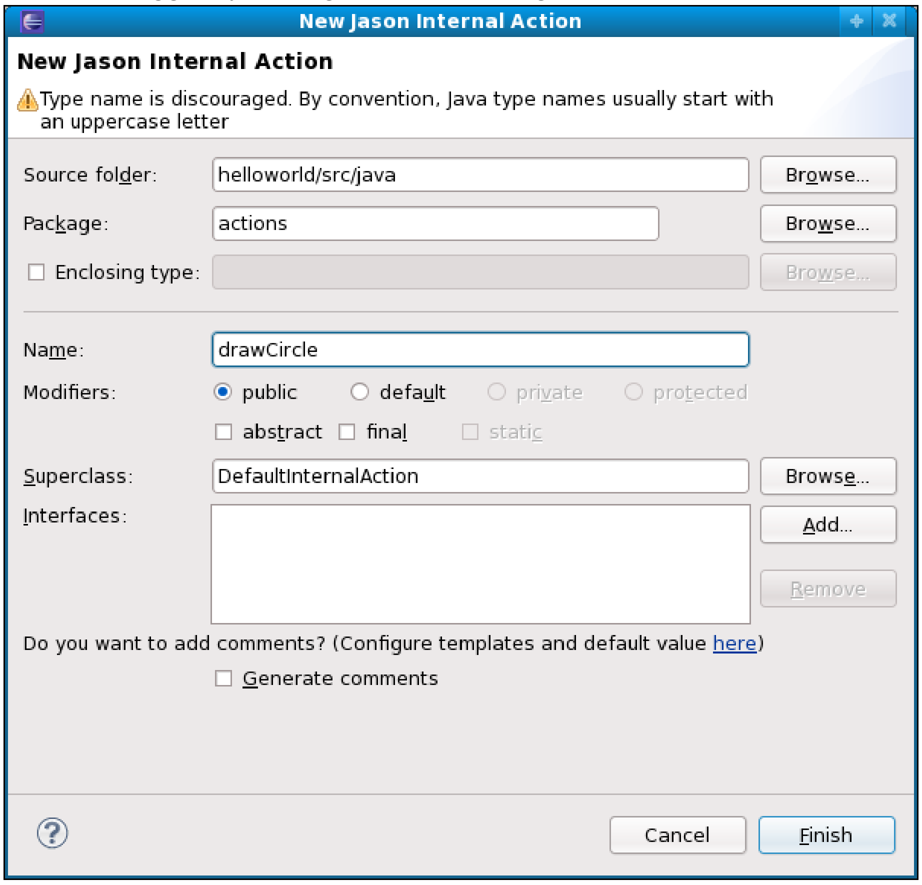
Click on the source folder named "src/java" using the right button and go to the option New > Internal Action.



Step 2

Fill out the form. An internal action is a java class, so the only required field is the name of the class.

Note: we suggest you to give a name using the first letter in lower case and also naming the package.



New Jason Internal Action

New Jason Internal Action

Type name is discouraged. By convention, Java type names usually start with an uppercase letter

Source folder:	helloworld/src/java	Browse...
Package:	actions	Browse...
☐ Enclosing type:		Browse...
Name:	drawCircle	
Modifiers:	● public ○ default ○ private ○ protected	
	☐ abstract ☐ final ☐ static	
Superclass:	DefaultInternalAction	Browse...
Interfaces:		Add... / Remove

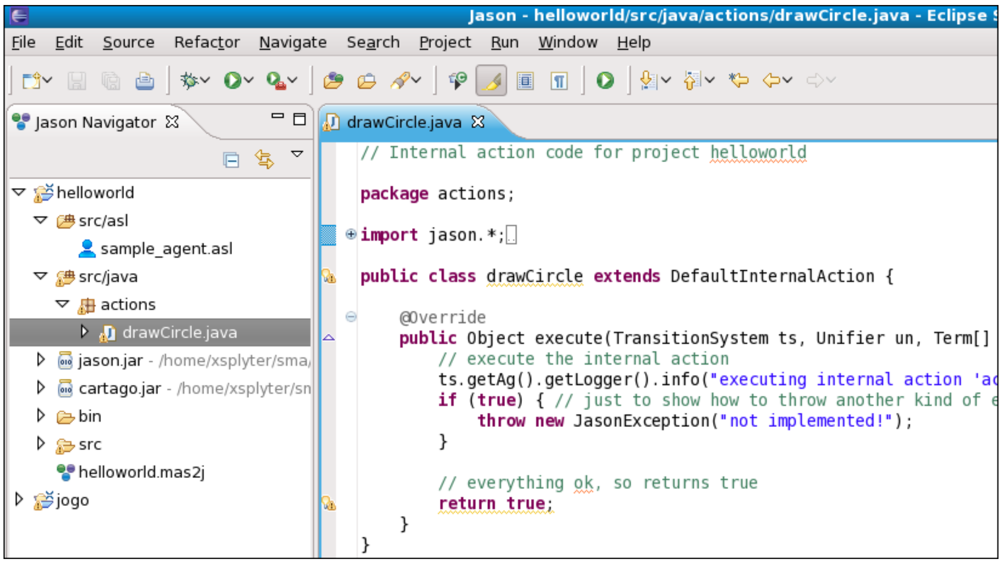
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Cancel Finish

Step 3

The internal action will be created.



Jason - helloworld/src/java/actions/drawCircle.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help
Jason Navigator
helloworld
src/asl
sample_agent.asl
src/java
actions
drawCircle.java
jason.jar - /home/xsplyter/sma
cartago.jar - /home/xsplyter/sn
bin
src
helloworld.mas2j
jogo
drawCircle.java

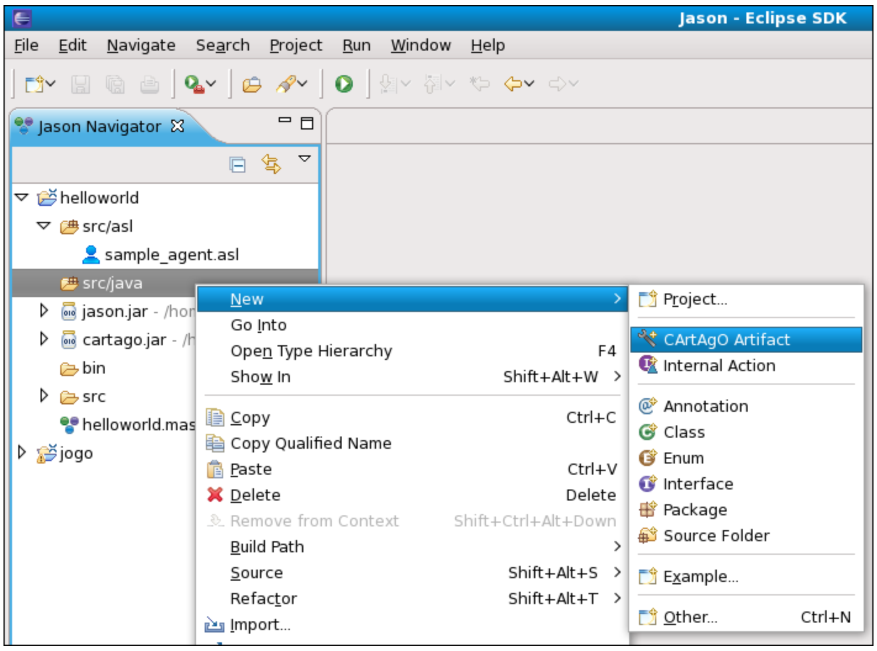
```
// Internal action code for project helloworld

package actions;

import jason.*;

public class drawCircle extends DefaultInternalAction {

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term[]
        // execute the internal action
        ts.getAg().getLogger().info("executing internal action 'ac
        if (true) { // just to show how to throw another kind of e
            throw new JasonException("not implemented!");
        }

        // everything ok, so returns true
        return true;
    }
}
```

7.4. How to create an artifact

Note: A CArtAgO artifact is only used if you are using the CArtAgO as an environment for your MAS.

Step 1

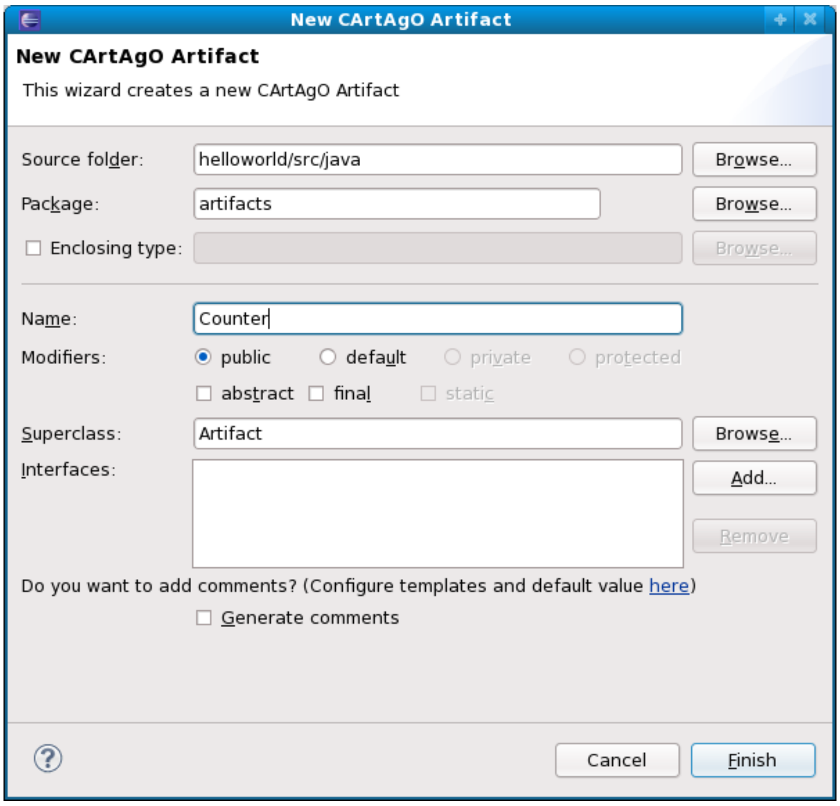
Click on the source folder named "src/java" using the right button and go to the option New > CArtAgO Artifact.



Step 2

Fill out the form. A CArtAgO artifact is a java class, so the only required field is the name of the class.

Note: in contrast to an internal action, in this case you could use a name with a first letter in upper case, and also we suggest you to name the package.



New CArtAgO Artifact

New CArtAgO Artifact

This wizard creates a new CArtAgO Artifact

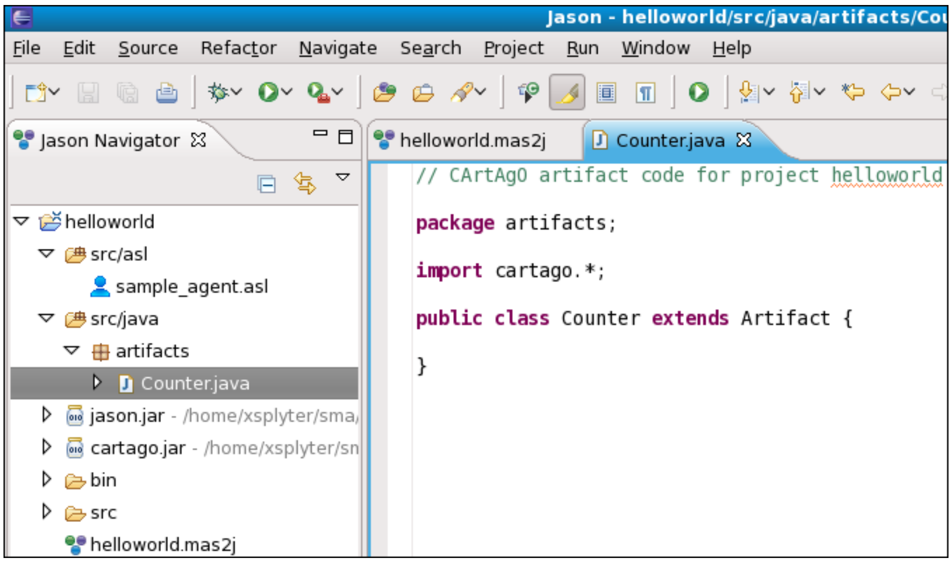
Source folder:	helloworld/src/java
Package:	artifacts
Enclosing type:	
Name:	Counter
Modifiers:	[✓ public] default private protected
	abstract final static
Superclass:	Artifact
Interfaces:	

Do you want to add comments? (Configure templates and default value here)

Generate comments

Step 3

The CArtAgO artifact will be created.

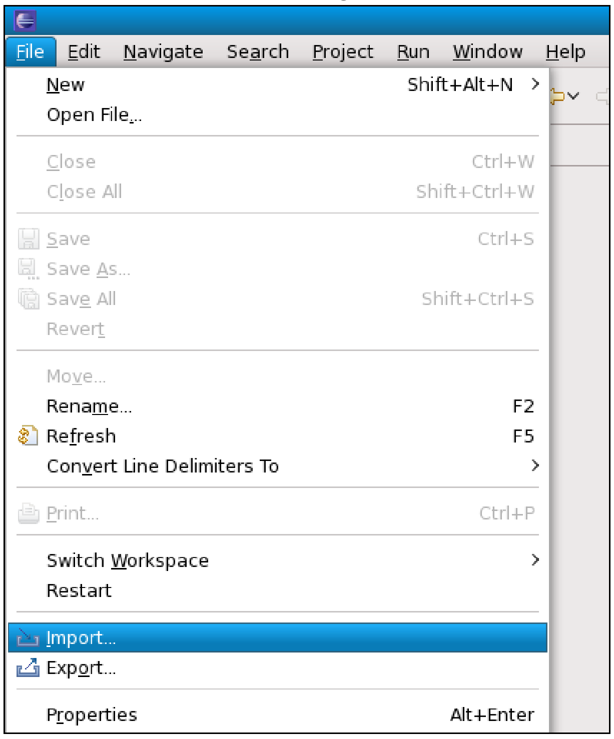

Jason - helloworld/src/java/artifacts/Cou
File Edit Source Refactor Navigate Search Project Run Window Help
Jason Navigator
helloworld.mas2j
Counter.java
helloworld
src/asl
sample_agent.asl
src/java
artifacts
Counter.java
jason.jar - /home/xsplyter/sma
cartago.jar - /home/xsplyter/sn
bin
src
helloworld.mas2j

```
// CArtAgO artifact code for project helloworld

package artifacts;

import cartago.*;

public class Counter extends Artifact {

}
```

7.5. How to import a Jason project

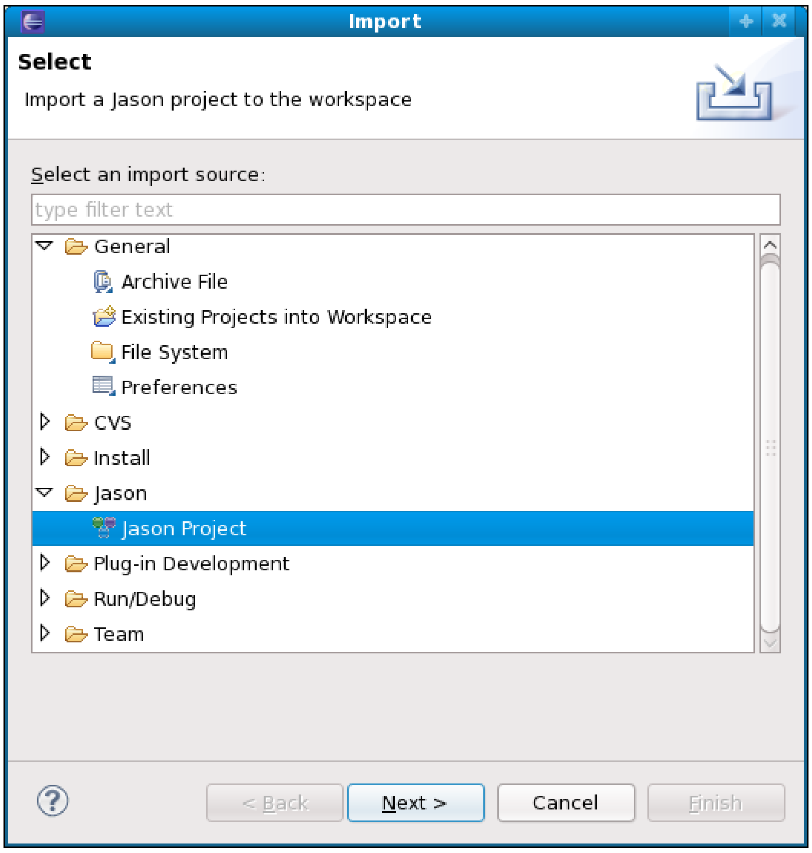
Step 1

Click on the "File" menu and go to the option "Import...".



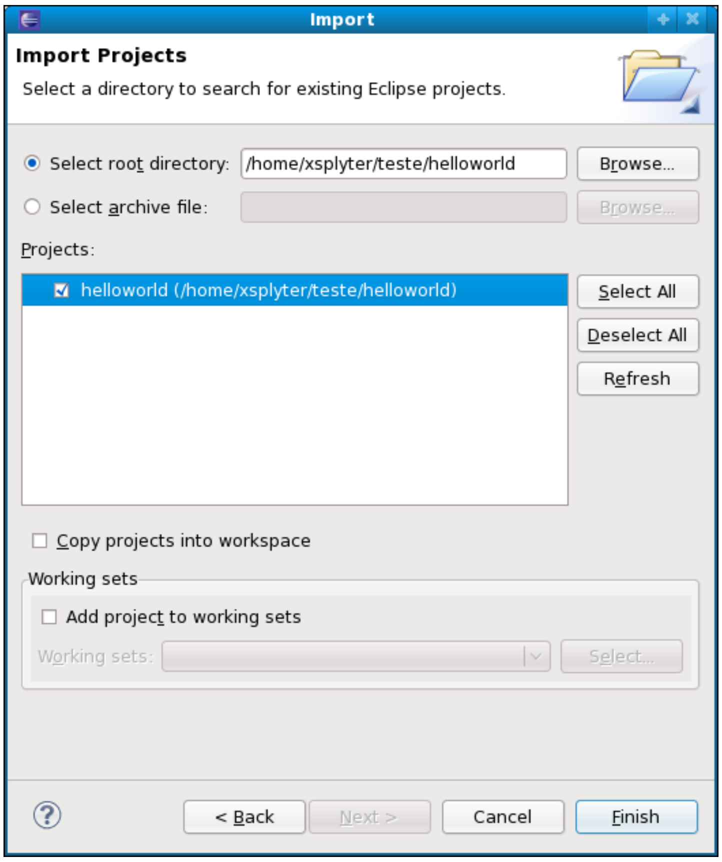
Step 2

Select the option Jason > Jason Project.



Step 3

Click on the "Browse" button and choose the directory of the project, tick the project that you wish to import and finally press the "Finish" button.


Import

Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: /home/xsplyter/teste/helloworld [Browse...]

( ) Select archive file: [Browse...]

Projects:

☑ helloworld (/home/xsplyter/teste/helloworld)

[Select All] [Deselect All] [Refresh]

☐ Copy projects into workspace

Working sets

☐ Add project to working sets

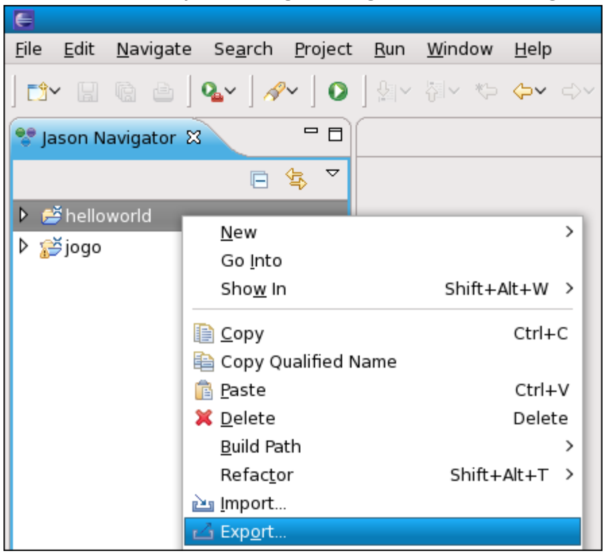
Working sets: [Select...]

[< Back] [Next >] [Cancel] [Finish]

7.6. How to export a Jason project

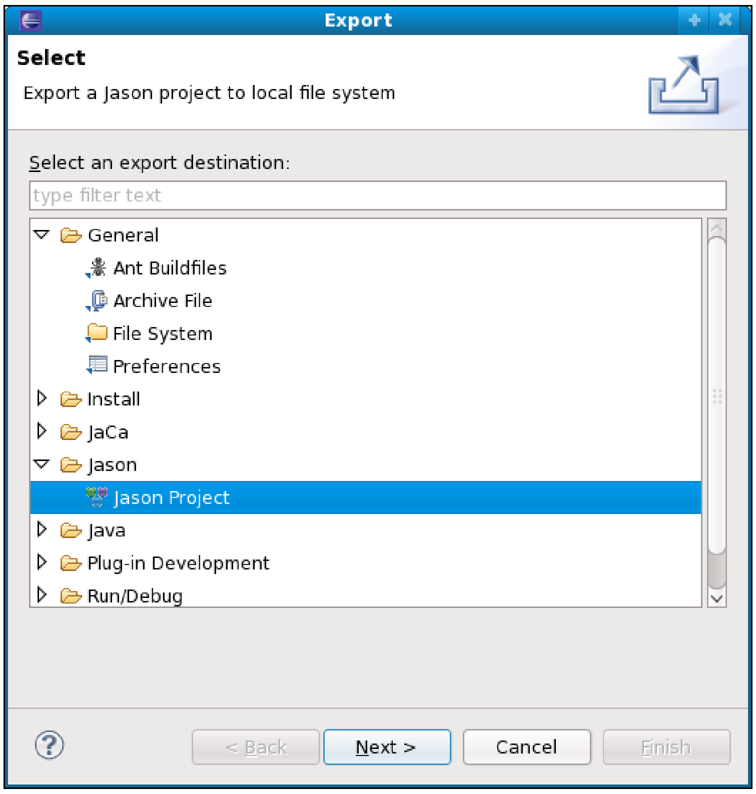
Step 1

Click on the project using the right button and go to the option "Export...".



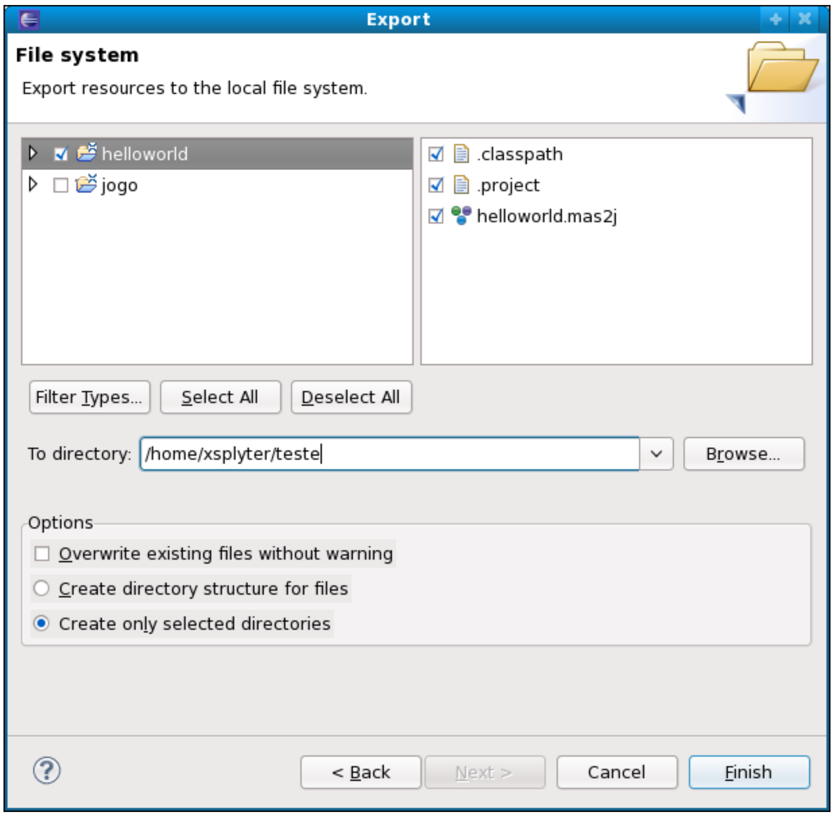
Step 2

Select the option Jason > Jason Project.



Step 3

Click on the "Browse" button and select the directory that you wish to export the project and press the "Finish" button.



To directory: /home/xsplyter/teste

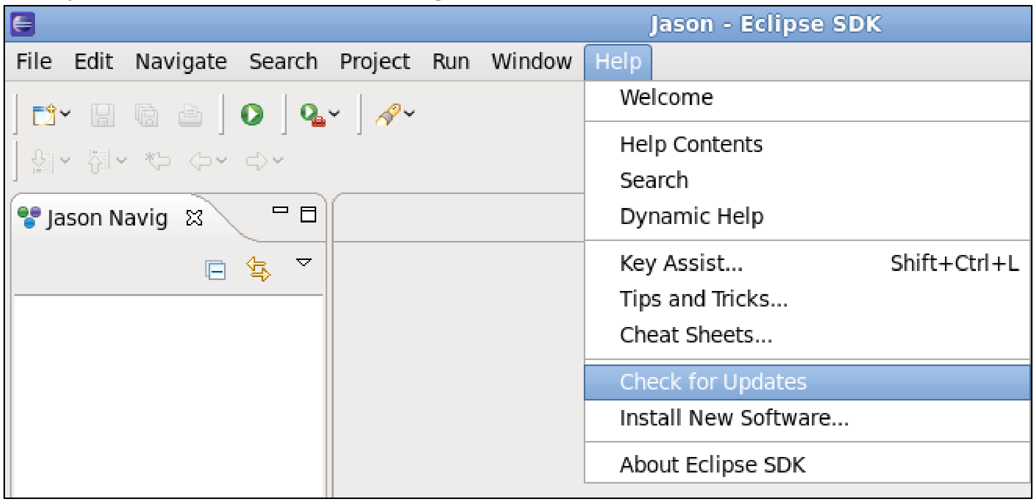
Options:
- [ ] Overwrite existing files without warning
- ○ Create directory structure for files
- [✓ Create only selected directories]

7.7. How to update the Jason plugin

You have two ways to update your Jason eclipse plugin.

First way

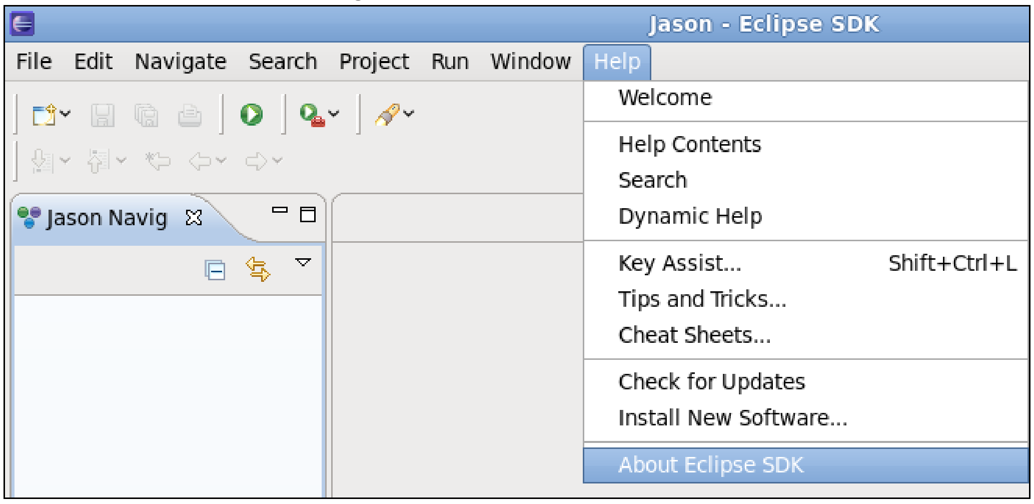
Simply click on the "Help" menu and go to the option "Check for Updates".



Second way

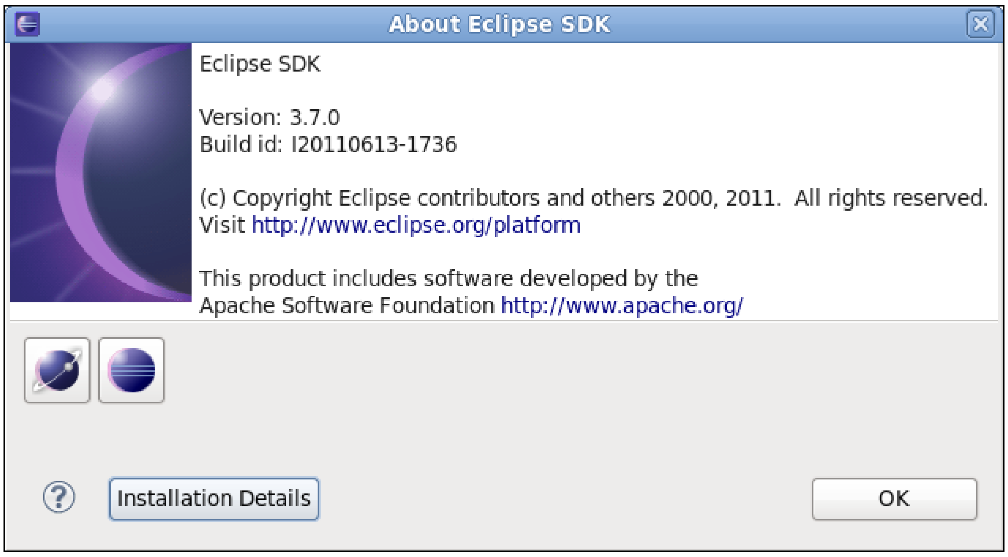
Step 1

Click on the "Help" menu and go to the option "About Eclipse SDK".



Step 2

Press the "Installation Details" button.



About Eclipse SDK

Eclipse SDK

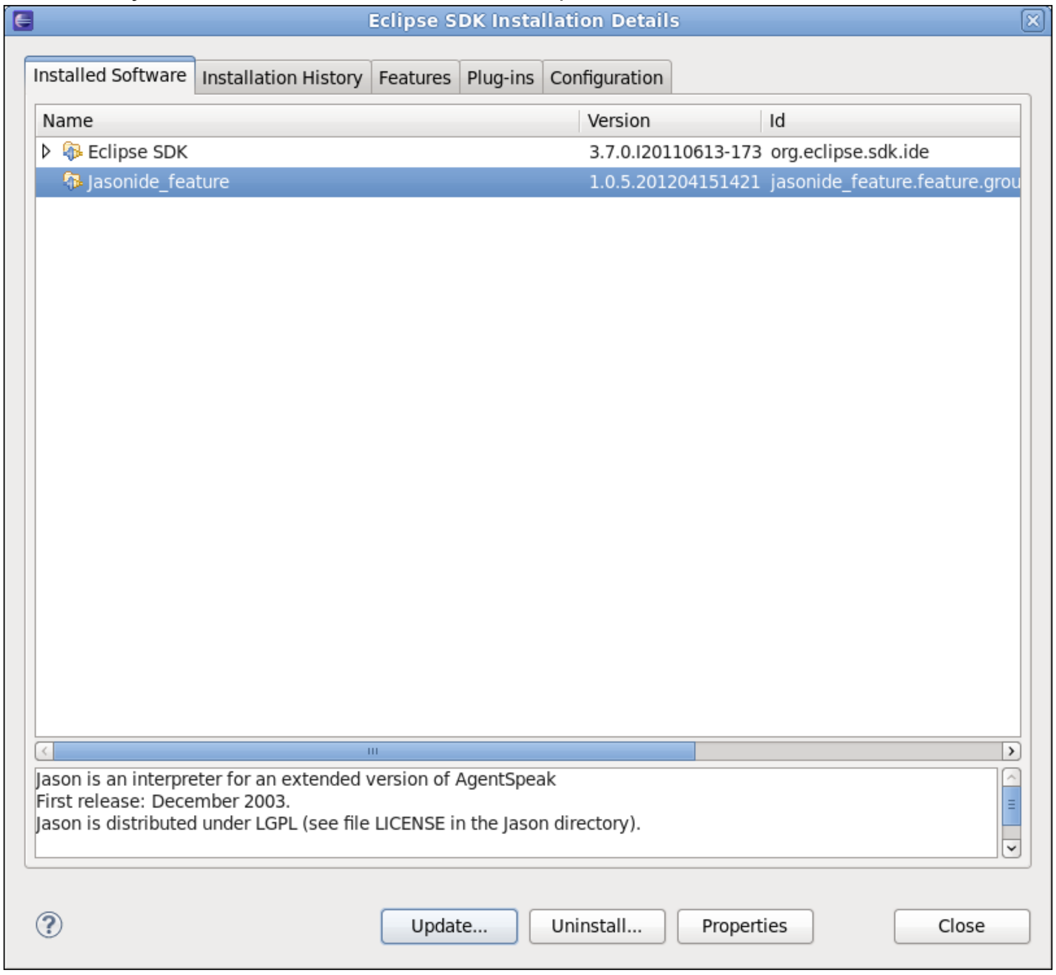
Version: 3.7.0
Build id: I20110613-1736

(c) Copyright Eclipse contributors and others 2000, 2011. All rights reserved.
Visit http://www.eclipse.org/platform

This product includes software developed by the
Apache Software Foundation http://www.apache.org/

Step 3

Select the "jasonide_feature" and click on the "Update..." button.



Eclipse SDK Installation Details

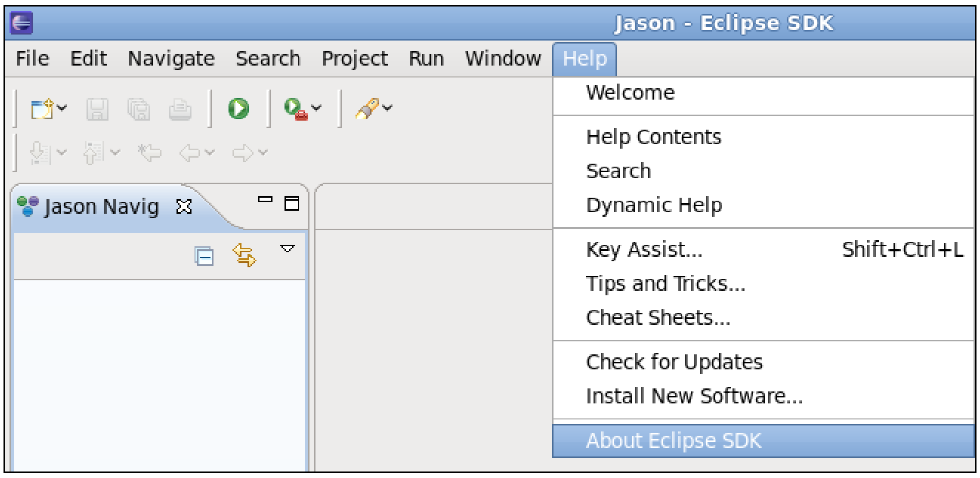
Name	Version	Id
Eclipse SDK	3.7.0.I20110613-173	org.eclipse.sdk.ide
Jasonide_feature	1.0.5.201204151421	jasonide_feature.feature.grou

Jason is an interpreter for an extended version of AgentSpeak
First release: December 2003.
Jason is distributed under LGPL (see file LICENSE in the Jason directory).

7.8. How to unistall the Jason plugin

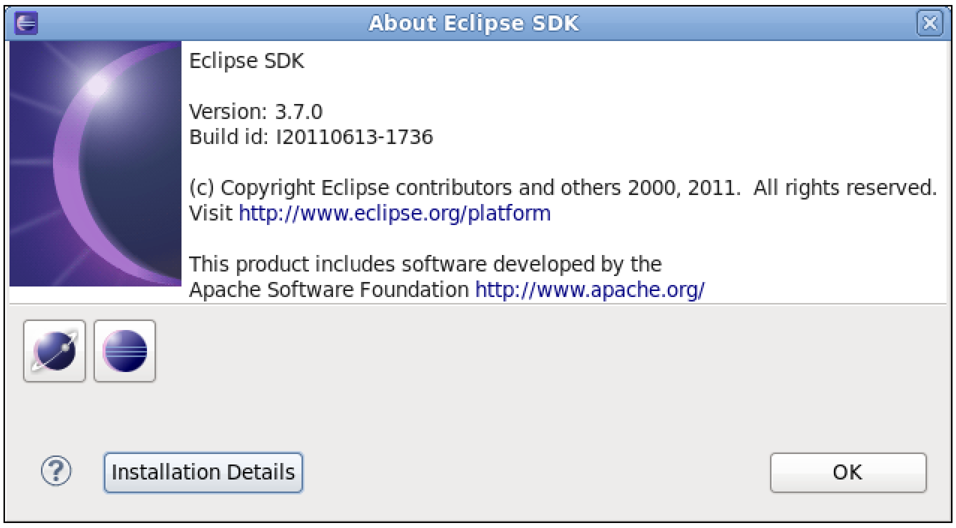
Step 1

Click on the "Help" menu and go to the option "About Eclipse SDK".



Step 2

Press the "Installation Details" button.



Eclipse SDK

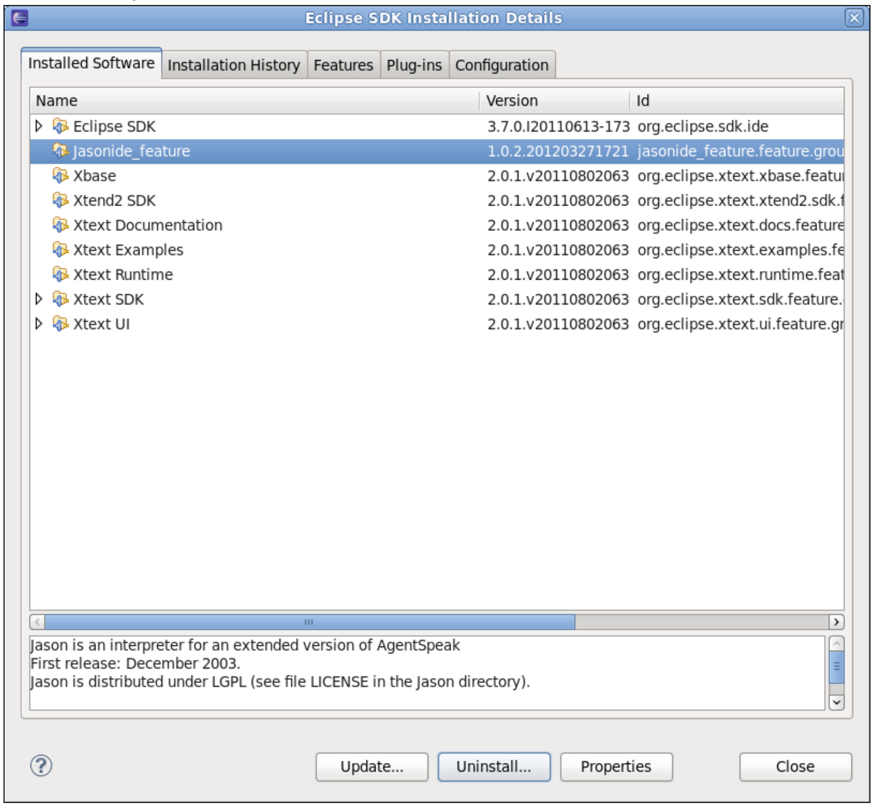
Version: 3.7.0
Build id: I20110613-1736

(c) Copyright Eclipse contributors and others 2000, 2011. All rights reserved.
Visit http://www.eclipse.org/platform

This product includes software developed by the
Apache Software Foundation http://www.apache.org/

Step 3

Select the "jasonide_feature" and click on the "Uninstall..." button.

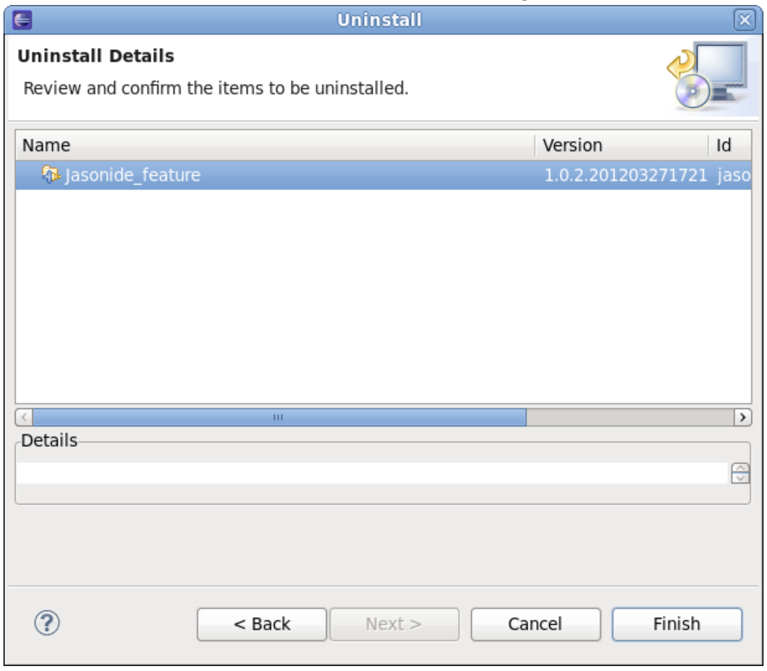


Name	Version	Id
Eclipse SDK	3.7.0.I20110613-173	org.eclipse.sdk.ide
Jasonide_feature	1.0.2.201203271721	jasonide_feature.feature.grou
Xbase	2.0.1.v20110802063	org.eclipse.xtext.xbase.featu
Xtend2 SDK	2.0.1.v20110802063	org.eclipse.xtext.xtend2.sdk.f
Xtext Documentation	2.0.1.v20110802063	org.eclipse.xtext.docs.feature
Xtext Examples	2.0.1.v20110802063	org.eclipse.xtext.examples.fe
Xtext Runtime	2.0.1.v20110802063	org.eclipse.xtext.runtime.feat
Xtext SDK	2.0.1.v20110802063	org.eclipse.xtext.sdk.feature.
Xtext UI	2.0.1.v20110802063	org.eclipse.xtext.ui.feature.gr

Jason is an interpreter for an extended version of AgentSpeak
First release: December 2003.
Jason is distributed under LGPL (see file LICENSE in the Jason directory).

Step 4

Confirm the process of uninstallation pressing the "Finish" button.



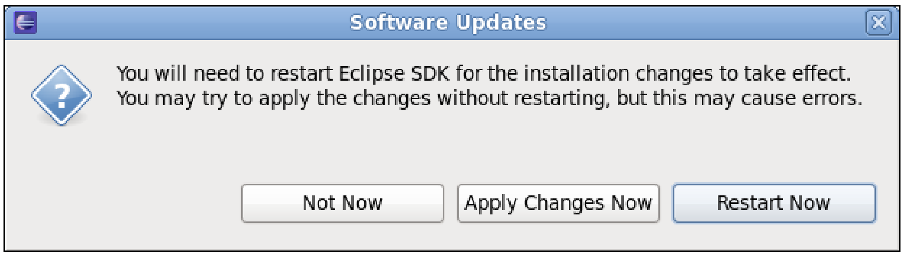
Uninstall Details

Review and confirm the items to be uninstalled.

Name	Version	Id
Jasonide_feature	1.0.2.201203271721	jaso

Step 5

In order to complete the uninstallation press the "Restart Now" button.



You will need to restart Eclipse SDK for the installation changes to take effect.
You may try to apply the changes without restarting, but this may cause errors.

You will find more information at http://jason.sf.net.

8. Appendix: Execution of JGOMAS

To execute JGOMAS it is first necessary to run the manager and then the agents.

- Run the manager: Execute the following command or the file `jgomas_manager.bat` (Windows) or `jgomas_manager.sh` (Linux):

```
java -classpath "lib\jade.jar;lib\jadeTools.jar;lib
\Base64.jar;lib\http.jar;lib\iiop.jar;lib
\beangenerator.jar;lib\jgomas.jar;student.jar;lib
\jason.jar;lib\JasonJGomas.jar;classes;." jade.Boot -gui
"Manager:es.upv.dsic.gti_ia.jgomas.Cmanager(<number_of_agents>,    <map>,    <re-
fresh_time>,<duration>)"
```

  Example:

```
java -classpath "lib\jade.jar;lib\jadeTools.jar;lib
\Base64.jar;lib\http.jar;lib\iiop.jar;lib
\beangenerator.jar;lib\jgomas.jar;student.jar;lib
\jason.jar;lib\JasonJGomas.jar;classes;." jade.Boot -gui
"Manager:es.upv.dsic.gti_ia.jgomas.Cmanager(4, map_04, 125,10)"
```

  The last line (in orange) represents the call to the manager agent, with the name `Manager`, the class `es.upv.dsic.gti_ia.jgomas.Cmanager` and the following parameters:

  - 4: number of agents
  - Map_04: name of the map
  - 125: refreshing time specified in milliseconds
  - 10: duration of the execution specified in minutes

- Run the agents: Execute the following command or the file `jgomas_launcher.bat` (Windows) or `jgomas_launcher.bat` (Linux):

```
java -classpath "lib\jade.jar;lib\jadeTools.jar;lib\Base64.jar;lib
\http.jar;lib\iiop.jar;lib\beangenerator.jar;lib
\jgomas.jar;student.jar;lib\jason.jar;lib\JasonJGomas.jar;classes;."
jade.Boot -container -host localhost
"<lista_de_agentes>"
```

  Example:

```
java -classpath "lib\jade.jar;lib\jadeTools.jar;lib\Base64.jar;lib
\http.jar;lib\iiop.jar;lib\beangenerator.jar;lib
\jgomas.jar;student.jar;lib\jason.jar;lib\JasonJGomas.jar;classes;."
jade.Boot -container -host localhost
"T1:es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_AXIS.asl);T2:es.
upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_AXIS_MEDIC.asl);A1:es.u
pv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_ALLIED_FIELDOPS.asl);A2:
es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_ALLIED.asl)"
```

  As in the case of the manager, the latest lines are used to define agents. In this case, there are defined the agents T1, T2, A1, and A2. For example, in the case of the agent T1, its name is represented with T1, its class with `es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch` and finally it is specified the .asl file where its code can be found, in this case `jason-Agent_AXIS.asl`.

- Finally, to visualize the graphic engine the file run_jgomasrender.bat is executed, containing the following instructions:

```
set OSG_FILE_PATH=../../../../data
JGOMAS_Render.exe --server <hostname> --port <integer>
```

9. Appendix: Available beliefs, actions, and objectives/plans available in Jason-JGomas application

9.1. Available beliefs

`tasks([])`
Contains the list of active tasks of the agent. Ex:
`tasks([task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),""),task(1001,"TASK_WALKING_PATH", "A2",pos(204,0,228),"")])`
In this example the list has two active tasks:

- `task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),"")` that indicates that has to get the objective (the flag) that is in the position pos(224,0,224). The priority assigned to this task is 1000.
- `task(2000,"TASK_GIVE_MEDIPAKS","A2",pos(204,0,228),"")` that indicates that has to go to the position pos(204,0,228) where the agent A2 is waiting for medicin packs (the agent with this active task is an agent). The priority assigned to this task is 2000.

`task_priority(Task, Y)`
Indicates the priority of each task of the agent. These values can be modified depending on what the agent decides. Ex: `task_priority("TASK_GET_OBJECTIVE",1000)` indicates that the priority of the task `TASK_GET_OBJECTIVE`" is 1000.

`manager("Manager")`
Indicates the name of the Manager agent. It is not required to be modified.

`team(X)`
Belief that indicates the team where an agent belongs to: “AXIS” or “ALLIED”

`type(X)`
Indicates the type of soldier that the agent is: "CLASS_SOLDIER", "CLASS_MEDIC" or "CLASS_FIELDOPS"

`patrollingRadius(X)`
Only to agents of the team “AXIS” . It is used to indicate the radius when patrolling. Initially is 64. The higher the value, the further the agents will be of the object to patrol around.

`fovObjects([])`
Contains the list of objects currently seen by the agent. The structure of an object is `[#, TEAM, TYPE, ANGLE, DISTANCE, HEALTH, POSITION]`.
Example: [1,200,1,0.58,14.76,78,pos(214,0,219)], the object 1 of the team 200 (AXIS), type 1 (agent), with angle 0,58, with a distance of 14,76, a health 78 and its position is pos(214,0,219)
Note: The TEAM values are: 100 (Allied), 200 (Axis), 1003 (flag)

`aimed("false")`
Indicates that the agent has no enemy to aim.
`aimed("true")`
Indicates that the agent has an enemy to shoot.

medicAction(on)
Indicates that the medic agent is going to help
medicAction(off)
Indicates that the medic agent will not help

fieldopsAction(on)
Indicates that the fieldops agent is decided to help
fieldopsAction(off)
Indicates that the fieldops agents is not decided to help

objectivePackTaken(on)
Indicates that the agent has the flag

state(State)
This belief is used to indicate the state of the agent in his state machine: standing, selecting which task to do or waiting; go_to_target, going to its next target; target_reached, it has reached the target; quit, has to finish.

current_task(Task)
Stores the current task. Ex: `current_task(task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),""))` the current task is to get the flag in position 222,0,224

my_health(X)
Stores the health of the agent. The initial, and maximum, value is 100. When this value reaches 0, the agent dies.

my_ammo(X)
Stores the amount of bullets of the agent. The initial value is 100.

my_position(X,Y,Z);
Stores the last position known by the agent.

objective(ObjectiveX, ObjectiveY, ObjectiveZ)
Stores the latest known position of the Guarda la posición última conocida del objetivo del agente

my_ammo_threshold(X)
Stores the threshold of bullets that the agent has to reach before asking for help or taking a decision. The initial value is 50.

my_health_threshold(X)
Stores the threshold of health that the agent has to reach before asking for help or taking a decision. Initially, it is 50.

debug(X)
Indicates the degree of verbosity of the agent between 1 and 3.

9.2. Acciones

`update_destination(Destination)`
This action modifies the destination of the agent when a new posible destination is calculated. For example, it is used when it is decided to change the destination to chase an enemy.

`move(Time);`
This action is used to indicate the movement action during a time Time. Its usage is internal, it is not necessary to use it.

`check_position(Position)`
It is used to check if a position is valid or not. After its execution, a belief is executed: position(X) where X can be valid or invalid.

`create_medic_pack`
This action is used by medic agents to generate medicine packs. Its usage is internal, it is not necessary to use it.

`create_ammo_pack`
This action is used by the fieldops agents to generate ammunition packs. Its usage is internal, it is not necessary to use it.

9.3. Added internal actions

`.my_team(type,list)`
This action returns the list of available agents of your team through the yellow pages (DF) that are of the type "type" . Ex: `.my_team("medic_AXIS", E)` executed by an agent of the "AXIS" returns the list E that contains the members of its team.
The available services by default are:

- "AXIS" , offered by all agents of the AXIS team
- "ALLIED" , offered by all agents of the ALLIED team
- "medic_AXIS" , offered by all medic agents of the AXIS team
- "fieldops_AXIS", offered by all fieldops agents of the AXIS team
- "backup_AXIS", offered by all soldier agents of the AXIS team
- "medic_ALLIED" , offered by all medic agents of the ALLIED team
- "fieldops_ALLIED", offered by all fieldops agents of the ALLIED team
- "backup_ ALLIED ", offered by all soldier agents of the ALLIED team

`.register( "JGOMAS", "type")`
It is used to register in the DF a service of the type "type" by the agent that executes it. Ex: `.register("JGOMAS", "medic_AXIS")` registers in the DF the service "medic_AXIS".

`.send_msg_with_conversation_id(Rec, Perf, Cont, ConvId)`
It is used to send a message with a conversation id (required by compatibility issues with the manager and the rest of agents). Ex: `.send_msg_with_conversation_id(ag1, tell, Content, "CFA")` sends a message to agent ag1, with the performative "tell" , with the content Content and with the conversation id "CFA" that means

Call for Ammo.

9.4. Objectives (Available objectives and plans that can be used but not modified)

`!look`
This objective is triggered for an agent to trigger the objects in its surroundings, and updates fovObjects(ObjList), which can be used to check the closer objects. Its usage is internal it is not necessary to use it.

`!shot(0)`
This objective is triggered when the agent decides to shoot.

`!add_task(task(TaskPriority, TaskType, Agent, Position, Content))`
`!add_task(task(TaskType, Agent, Position, Content))`
These two objectives are triggered to add a new task in the list of tasks of the agent.

9.5. Objectives (Available objectives and plans that may be modified)

`!init`
This objective is triggered once during agent initialization. Its associated plan can be used to introduce beliefs or new objectives required for the designed strategy.

`!update_targets`
This objective can be used to update the tasks and its priorities. It is invoked when the agent changes to the standing state and has to choose a new task among the available ones. It is necessary to implement the plan associated to the creation of this event.

`!perform_look_action`
This object is invoked when the agent looks around and update the list of surrounding objects `fovObjects(L)`. It would be necessary to implement the plan associated to the creation of this event to be able look what is around.

`!get_agent_to_aim`
This objective is invoked after `!perform_look_action`, and it would be used to decide if there is any enemy to aim. An easy implementation of the associated plan is available. It is interesting to improve the mentioned associated plan to take a more refined solution about who to aim.

`!perform_aim_action`
This objective is triggered if there is an enemy to aim, which can be used to take a decision about what to do with the aimed agent. An easy implementation of the plan is available. It is interesting to improve this plan to take a more refined decision about who to aim.

`!perform_injury_action`
This objective is triggered when the agent is shot. It is necessary to implement the plan associated to the creation of this event to take a decision. For example, run if the agent has a low life value.

`!perform_no_ammo_action`

This objective is triggered when the agent shoots and has no ammo. It is necessary to implement the associated plan to take a decision. For example, to run.

`!performThresholdAction`

This objective is triggered when the agent has less life or bullets than the thresholds `my_ammo_threshold(X)` and `my_health_threshold(X)`. An easy implementation of the associated plan is available, which asks for help to medics or fieldops of its team. It is interesting to improve the plan to take a more refined solution.

`!checkMedicAction` (Medics only)

This objective is triggered when a medic agent receives a help request. An easy implementation is available, which decides to always help generating the belief `medicAction(on)`. It is interesting to improve that plan to take a more refined solution and sometimes to generate the belief `medicAction(off)`, which makes the agent to not attend the petition because it is not interested.

`!checkAmmoAction` (Fieldops only)

This objective is triggered when a fieldops agent receives a help request. An easy implementation of the associated plan is available, which always decides to help, generating the belief `fieldopsAction(on)`. It is interesting to improve the plan to take a more refined decision and to sometimes generate the belief `fieldopsAction(off)`, which makes the agent to not attend the petition because it is not interested.

`!setup_priorities`

This objective is triggered during agent initialization to fix agent task priorities. Each agent has its own priorities. A simple implementation is available. It is interesting to modify it to add new tasks or modify the priorities to have agents that behave in a different way.

`+cfm_agree[source(M)]`

This objective is triggered when it is received a message stating that medic help is coming. Agent M will attend the request. It is not necessary to modify the plan but it could be interesting to modify it to refine agent behavior.

`+cfm_refuse[source(M)]`

This objective is triggered when a message that denies medical help is received. Agent M has denied the request. It is not necessary to modify the associated plan but it could be interesting to refine the agent behavior when help is denied.

`+cfa_agree[source(M)]`

This objective is triggered when it is received a message stating that ammo help is coming. Agent M will attend the request. It is not necessary to modify the plan but it could be interesting to modify it to refine agent behavior.

`+cfa_refuse[source(M)]`

This objective is triggered when a message that denies ammo help is received. Agent M has denied the request. It is not necessary to modify the associated plan but it could be interesting to refine the agent behavior when help is denied.