JaCaMo-Unity integration using VEsNA Framework

A Guide Map in the Rabbit Hole

~ Andrea Gatti ~
DIBRIS, Università degli Studi di Genova, Italy

Introduction

Agent

We will consider *agents* written in AgentSpeak and interpreted by Jason or, more widely, by JaCaMo. These agents are **Belief-Desire-Intention** (BDI), agents that act with *intentions* based on their *belief*s to fulfill their *desires*.

A small recap of AgentSpeak Syntax

We will now introduce a small example of AgentSpeak code:

```
friend( alice ).

!say_hello( alice ).

+!say_hello( Ag )
  :   friend( Ag )
  ←   .print( "Hello, ", Ag, "!").
```

In this example, the agent has a *belief* `friend( alice )`, a *desire* `!say_hello( alice )` and a *plan* `+!say_hello( Ag )`. The agent wants to greet Alice: its *intention* is to use the available plan to fulfill its *desire*.

Let's analyze the syntax of the plan. We are saying that to fulfill (`+`) the desire (`!say_hello( Ag )`) we should follow the plan. The plan has a *context* in which it is applicable. The context follows the `:`. In this case, the agent can greet `Ag` if exists a belief `friend( Ag )`. If the context is satisfied then the agent will follow the instructions after the ←.

> **NOTE**
> *Note that if the agent hadn't the belief `friend( alice )`, the **intention** would have failed: the agent has the intention to greet but does not have a plan applicable in the context.*

From this small example we can highlight a few more important concepts. Words that start with lowercase letter are **ground**: they are values, truths. Words that start with uppercase letter instead are **variables**: they need to be **unified**. We could spend many words on the concept of *unification*. Let's just say that in the case of the example the variable `Ag` is *unified* with the term `alice`.

Another important point is the `.print()`. This *action* (notice that it is an action and not a function because it is performed by an agent) is a *DefaultInternalAction*. Jason provides a set of actions that are predefined and that the agent is able to perform.

In order to launch the code you will need also a `.jcm` file. This file defines the configuration of the multi-agent system.

```
mas example {

  bob:agent.asl
  alice:agent.asl

}
```

In this example there are two agents with name *alice* and *bob* that will use the code inside `agent.asl` to live.

> **ALERT!**
> *Qui mancano tutte le notazioni* `?`*,* `+`*, ecc ecc*

VEsNA

VEsNA (Virtual Environments via Natural language Agents) is a framework that provides different tools to create *embodied agents*.

To make an agent a VEsNA agent it is sufficient to modify the `.jcm` file setting the `ag-class` parameter as follows:

```
mas example {

  bob:agent.asl {
    beliefs:  address( localhost )
              port( 8080 )
    ag-class: vesna.VesnaAgent
  }

}
```

The agent also has two initial beliefs that tells the agent where to connect. The agent implements a WebSocket client, the environment should implement a WebSocket server.

VEsNA Agent

`VesnaAgent` class extends the default `Agent` class from Jason. It overrides `loadInitialAS` to create the connection with the body before the server starts. If the connection is not available the agent is killed. Note these lines (61-74):

```
// Connect the two handle functions to the client object
client.setMsgHandler( new WsClientMsgHandler() {
    @Override
    public void handle_msg( String msg ) {
        vesna_handle_msg( msg );
    }

    @Override
    public void handle_error( Exception ex ) {
        vesna_handle_error( ex );
    }
}  );
// Connect the body
client.connect();
```

VEsNA Internal Actions (VIA)

VEsNA provides also a set of additional *DefaultInternalActions*:

- `vesna.walk`
- `vesna.stop`
- `vesna.rotate`
- `vesna.jump`
- `vesna.says`

These actions are things the agent knows how to do. In practice they are all actions in the environment so they send a message to the body with all the needed data. We will now briefly describe the API.

All the messages from VEsNA actions are JSON formatted and follow this structure:

{ data }

```
{
  "sender": "alice",
  "receiver": "body",
  "type": "walk",
  "data": {}
}
```

Vesna Walk

Can take different number of arguments:

- `vesna.walk()`: performs a step

{ data }

```
{
  "type": "step"
}
```

- `vesna.walk( n )`: performs a step of length n

{ data }

```
{
  "type": "step",
  "length": n
}
```

- `vesna.walk( Target )`: goes to the Target

{ data }

```
{
  "type": "goto",
  "target": Target
}
```

- `vesna.walk( Target, Id )`: goes to the Target with Id (this is useful in cases in which there are multiple objects with the same name but different id)

{ data }

```
{
  "type": "goto",
  "target": Target,
  "id": Id
}
```

Vesna stop

This action takes no argument.

Vesna rotate

This command can take different arguments:

- `vesna.rotate( Direction )` where `Direction` is one of `left|right|forward|backward`: the agent rotates 90 degrees in that direction

{ data }

```
{
  "type": "direction",
  "direction": Direction
}
```

- `vesna.rotate( Target )` where `Target` is an element in the environment

{ data }

```
  {
    "type": "lookat",
    "target": Target
  }
```

- `vesna.rotate( Target, Id )`. Same as `walk`, you can also specify the `id` of an object in the environment if necessary

{ data }

```
  {
    "type": "lookat",
    "target": Target,
    "id": Id
  }
```

Vesna Jump

This action takes no argument.

Vesna Says

> **NOTE**
>
> *This is function is supposed to be used with the official KQML communication protocol. Agents can communicate through*
>
> ```
> .send( Performative, To, Msg )
> ```
>
> *where the Performative can be `tell`, `askHow`, `askOne`, `achieve`, etc. This is also the reason for the arguments order (consistent). Look at the official documentation for more concepts.*

This action can take different arguments:

- `vesna.says( Msg )` where `Msg` is the message to be displayed.

{ data }

```
  {
    "msg": Msg
  }
```

- `vesna.says( To, Msg )` where `Msg` is the message to be displayed and `To` is the recipient

{ data }

```
  {
    "recipient": To,
    "msg": Msg
  }
```

- `vesna.says( Performative, To, Msg )` where `Msg` is the message to be displayed, `To` is the recipient and `Performative` is the *performative* the user used

{ data }

```
{
  "performative": Performative,
  "recipient": To,
  "msg": Msg
}
```

VEsNA Plans

Environment