Multi-Agent Programming Contest EISMASSim Description (2015 Edition)

http://www.multiagentcontest.org/

Tobias Ahlbrecht Jürgen Dix Federico Schlesinger

May 18, 2015

New in 2015: Differences between the last year and 2015 are marked with boxes.

Contents

1 About EISMASSim

EISMASSim is based on EIS[1], which is an proposed standard for agent-environment interaction. It maps the communication between the *MASSim*-server and agents, that is sending and receiving XML-messages, to Java-method-calls and call-backs. On top of that it automatically establishes and maintains connections to a specified *MASSim*-server. Additionally it is intended to also gather statistics about the execution of your agents. EISMASSim and EIS both come as jar-files which are included in the software-package.

2 Using EISMASSim

In order to use EISMASSim with your project, you have to perform a couple of steps, which we will outline here.

1. Setting up the class-path: The first thing you have to do is to add EIS and EISMASSim to the class-path of your project. Please use the jar-files `eis-0.3.jar` and `eismassim-2.2.jar`. The first jar contains the *generic* environment-interface, the second one contains the *specialized* one.

New in 2015: EISMASSim has been updated to 2.2

2. Creating an instance of the environment interface: It is not intended to instantiate EIS-compliant environment-interfaces directly, that is calling the constructor of the respective class. Instead it is advised to use the *class-loader* `eis.EILoader`. Here is an example for instantiating the environment-interface-class via this very class-loader[2]:

```
EnvironmentInterfaceStandard ei = null;
try {
    String cn = "massim.eismassim.EnvironmentInterface";
    ei = EILoader.fromClassName(cn);
} catch (IOException e) {
    // TODO handle the exception
}
```

3. Registering your agents: Now that the environment-interface is instantiated you need to register your agents to it. That is, that you are required to register every single agent that is supposed to interact with the environment via the environment-interface using its name or any unique identifier. For each of your agents please do this:

[1] Available at `http://sf.net/projects/apleis/`.

[2] There is also a method called `fromJarFile`, which firstly add a jar-file to the class-path, secondly looks up the main-class attribute from the jar's manifest-entry, and thirdly instantiates the environment-interface. This works for EISMASSim as well.

```
try {
  ei.registerAgent(agentName);
} catch (AgentException e1) {
  // TODO handle the exception
}
```

4. Associating your agents with the vehicles: At this moment you have to associate your agents with the available entities. An entity is a connection to a vehicle, which is part of a simulation executed by the *MASSim*-server. You can associate one of your agents with an entity (vehicle) by using the entity's name. The names of the entities however are specified in the configuration XML-file (see below). As soon as you associate an agent with an entity, a connection to the *MASSim*-server is established. Here is an example how to associate an agent with an entity:

```
try {
    ei.associateEntity(agentName,entityName);
} catch (RelationException e) {
    // TODO handle the exception
}
```

5. Starting the execution: The next step is to start the overall execution. This is how it is done:

```
try {
    ei.start();
} catch (ManagementException e) {
    // TODO handle the exception
}
```

6. Perceiving the environment: Perceiving is facilitated either by 1. getting all percepts, that is calling the `getAllPercepts`-method or 2. by handling percepts-as-notifications, that is every time there is a new percept a listener's method is called in order to trigger a reaction to the percept. Note that this is EIS's usual policy about perceiving. Here is an example for retrieving all percepts[3]:

```
try {
    Collection<Percept> ret = getAllPercepts(getName());
    // TODO interpret the percepts
} catch (PerceiveException e)  {
    // TODO handle the exception
} catch (NoEnvironmentException e) {
    // TODO handle the exception
}
```

7. Acting: Executing an action means invoking the `performAction`-method and passing 1. the name of the agent, that intends to execute an action, and 2. an action-object that represents the action-to-be-executed. This is an exemplary execution of an action:

```
Action = new Action(...);
try {
    ei.performAction(agentName, action);
} catch (ActException e) {
    // handle the exception
}
```

[3]For an introduction on how to use percepts-as-notifications, see the manual that accompanies the EIS software package.

3 Configuring EISMASSim

The EISMASSim environment-interface can be configured using the configuration-file `eismassimconfig.xml` which is automatically loaded and evaluated when the environment-interface is instantiated. Fig. 1 shows an exemplary configuration-file for EISMASSim.

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaceConfig scenario="city2015" host="localhost" port="12300"
scheduling="yes" times="no" notifications="no" timeout="5000"
statisticsFile="no" statisticsShell="no">
  <entities>
    <entity name="vehicle1" username="a1" password="1" xml="yes" iilang="yes"/>
    <entity name="vehicle2" username="a2" password="1" xml="yes" iilang="yes"/>
    <entity name="vehicle3" username="a3" password="1" xml="yes" iilang="yes"/>
    <entity name="vehicle4" username="a4" password="1" xml="yes" iilang="yes"/>
    ...
  </entities>
</interfaceConfig>
```

Figure 1: An exemplary EISMASSim-configuration-file.

The attributes of the `<interfaceConfig>`-tag are:

- `scenario` specifies the Contest-scenario that is supposed to be handled. For 2015 the value that is accepted is `city2015`.
- `host` specifies the URL of the *MASSim*-server that runs the simulations. This can be for example `localhost`, a valid IP-address, or the fully-qualified hostname of one of our Contest-servers.
- `port` specifies the port-number of the *MASSim*-server.
- `scheduling` enables/disables scheduling. Enabled scheduling means that an action-message is not sent unless there is a valid action-id (see the protocol-description for details on the action-ids). This mechanism makes sure that a single action-id is used only once. Note that an attempt to send an action-message times out after 5 seconds (defined in `timeout`). The default value is `yes` for scheduling enabled. **Warning:** note, however, that disabling scheduling in the interface leaves you with the responsibility of scheduling, that is to ensure that the server is not strained with more than one action per connection and simulation-step.
- `times` enables/disables time-annotations. If enabled this will annotate each percept with a time-stamp, that indicates when the percept has been generated by the server (see the protocol-description for details on time-stamps).
- `notifications` denotes whether percepts are to be provided as notifications. The default-value is `no`.

- `timeout` specifies the number of milliseconds for the timeouts of the scheduling process. This is only used if scheduling is enabled. The default-value is `5000`.
- `statisticsFile` enables statistics-function, which computes the average response time of the agents and the percentual frequency of each kind of action. The results will be plotted to file.
- `statisticsShell` is similar to `statisticsFile`, just the result will be plotted to the shell. Both options' default value is no.

Each `<entity>`-tag specifies a single connection to the *MASSim*-server. The attributes are:

- `name` specifies the name of the connection. This is a requirement for acting and perceiving, and needs to be unique.
- `username` and `password` specify the credentials that are required by *MASSim*'s authentication-mechanism (provided either by the organizers, or specified in your very own server-configuration-file).
- `xml` enables/disables printing incoming/outgoing XML-messages to the console. This is useful for debugging-purposes. This is deactivated per default.
- `iilang` enables/disables printing percepts to the console. This is also useful for debugging-purposes. This is deactivated per default.

4 Scheduling

If scheduling is enabled the environment interface makes sure that the agents are properly synchronized with the *MASSim*-server. This is facilitated by ensuring that you can call the methods `getAllPercepts` and `performAction` only once per simulation step. In each step the server provides an action-identifier, which is a token that can be used only once. As long as there is no new action-identifier received from the server, `getAllPercepts` and `performAction` will block until a user-defined time-out is exceeded.

5 Actions and Percepts for the Logistics-Scenario

New in 2015: The actions have been updated to match the brand-new scenario.

Actions. In the following, we will elaborate on actions and percepts. Each action and each percept consists of a name followed by an optional list of parameters. The parameters have to be supplied as an `Identifier` representing a simple key-value list of the form `<key1>=<value1> <key2>=<value2> ...` in which pairs are separated by a single white space.

Below is the list of actions that can be performed in the course of each simulation. See the scenario-description for the precise semantics of the actions, as well as the parameters required/available for each one:

- `goto`
- `buy`
- `give`
- `receive`
- `store`
- `retrieve`
- `retrieve_delivered`
- `dump`
- `assemble`
- `assist_assemble`
- `deliver_job`
- `charge`
- `bid_for_job`
- `post_job`
- `continue`
- `skip`
- `abort`

Creating an action-object that is to be passed as a parameter to the method `performAction` is very straightforward:

```
Action a = new Action("goto", new Identifier("facility=shop1"));
```

Note: the use of `new Identifier("...")` to specify the list of parameters for an action is because of backwards compatibility. A more semantically correct way of specifying parameters is scheduled for a future update.

Percepts. In the following we will consider a list of percepts that can be available during a tournament. Note that during a simulation, data from the respective `sim-start`-message will be available as well as data from the current `request-action`-message (see the protocol description for details about such messages):

New in 2015: The percepts have been updated to match the brand-new scenario.

- `auctionJob(<Identifier1>,<Id2>,<Numeral3>,<Num4>,<Num5>, <Num6>,[item(<Id7>,<Num8>),...])` denotes a job that is currently auctioned.
  Parameters: 1. id 2. storage 3. begin step 4. end step 5. fine 6. maximum bid 7. item name 8. amount
- `batteryCapacity(<Numeral>)` denotes the maximum charge of an agent.
- `bye` indicates that the tournament is over.
- `charge(<Numeral>)` denotes the current (battery) charge of an agent.
- `chargingStation(<Identifier1>,<Numeral2>,<Numeral3>, <Numeral4>, <Numeral5>, <Numeral6>)` denotes a charging station.
  Parameters: 1. name 2. latitude 3. longitude 4. ch. rate 5. price 6. slots
- `deadline(<Numeral>)` indicates the deadline for sending a valid action-message to the server in Unix-time.
- `dump(<Identifier1>,<Numeral2>,<Numeral3>,<Numeral4>)` denotes a dump facility.
  Parameters: 1. name 2. latitude 3. longitude 4. price
- `entity(<Identifier1>,<Identifier2>,<Numeral3>,<Numeral4>, <Identifier5>)` denotes an entity (agent) in the simulation.
  Parameters: 1. name 2. team 3. latitude 4. longitude 5. role
- `fPosition(<Numeral>)` represents the queue-position the agent occupies in the facility (or -1)
- `id(<Identifier>)` indicates the identifier of the current simulation.
- `inFacility(<Identifier>)` denotes the facility the agent is in (or "none").

- `item(<Identifier1>,<Numeral2>)` an item the agent is carrying.
  Parameters: 1. item name 2. amount carried
- `jobPosted(<Identifier>)` denotes a job that has been posted by the agent's team.
- `jobTaken(<Identifier>)` denotes a job that has been taken by the agent's team (an auction that has been won).
- `lat(<Numeral>)` denotes the latitude of the agent's position (expect a double-value here)
- `load(<Numeral>)` denotes the current load of an agent.
- `loadCapacity(<Numeral>)` denotes the maximum load of an agent.
- `lon(<Numeral>)` denotes the longitude of the agent's position (see: lat).
- `lastAction(<Identifier>)` indicates the last action that was sent to the server.
- `lastActionParam(<Identifier>)` indicates the parameter of the last action that was sent to the server.
- `lastActionResult(<Identifier>)` indicates the outcome of the last action.
- `money(<Numeral>)` denotes the amount of money available to the vehicle's team.
- `pricedJob(<Identifier1>,<Id2>,<Numeral3>,<Num4>,<Num5>, [item(<Id6>,<Num7>,<Num8>),...])` denotes an active job both teams can work on.
  Parameters: 1. id 2. storage 3. begin 4. end 5. reward 6. item name 7. amount 8. delivered
  Note: Parameter 8 is only visible, if the job was posted by the agent's team.
- `product(<Identifier1>,<Numeral2>,[consumed(<Id3>,<Num4>), ...,tool(<Id5>,<Num6>),...])` denotes a product in the simulation.
  Parameters: 1. name 2. volume 3. item name 4. amount 5. item name 6. amount
  Note: If the list is empty, the product cannot be assembled.
- `ranking(<Numeral>)` indicates the outcome of the simulation for the vehicle's team, that is its ranking.
- `requestAction` indicates that the server has requested the vehicle to perform an action.
- `role` denotes the role as defined in the configuration.

- `route([wp(<Numeral1>,<Numeral2>,<Numeral3>),...])` denotes the route the agent is following.
  Parameters: 1. sequential id of the waypoint 2. latitude 3. longitude
- `routeLength(<Numeral>)` denotes the length of the route the system calculated, if the agent is currently following one.
- `shop(<Identifier1>,<Numeral2>,<Numeral3>,[item(<Id4>),...])` denotes some shop in the simulation.
  Parameters: 1. name 2. latitude 3. longitude 4. item name
- `shop(<Identifier1>,<Numeral2>,<Numeral3>,[availableItem(<Id4>, <Num5>,<Num6>,<Num7>),...])` denotes a nearby shop.
  Parameters: 1. name 2. latitude 3. longitude 4. item name 5. cost 6. amount 7. restock interval
- `simEnd` indicates that the server has notified the vehicle about the end of a simulation.
- `simStart` indicates that the server has notified the vehicle about the start of a simulation.
- `step(<Numeral>)` represents the current step of the current simulation.
- `steps(<Numeral>)` represents the overall number of steps of the current simulation.
- `storage(<Identifier1>,<Numeral2>,<Num3>,<Num4>,<Num5>,<Num6>, [item(<Id7>,<Num8>,<Num9>),...])` denotes a storage facility.
  Parameters: 1. name 2. latitude 3. longitude 4. price 5. total capacity 6. used capacity 7. item name 8. amount stored 9. amount delivered
- `timestamp(<Numeral>)` represents the moment in time, when the last message was sent by the server, again in Unix-time.
- `workshop(<Identifier1>,<Numeral2>,<Num3>,<Num4>)` denotes a workshop facility.
  Parameters: 1. name 2. latitude 3. longitude 4. price
- `visibleChargingStation(...,<Numeral7>)` denotes a charging station the agent can see. Contains same parameters plus an additional seventh representing the current size of the queue at this station.

Note, however, that the percepts look a little different, when annotations (see the section on configuring EISMASSim) are activated.