Multi-Agent Programming Contest (2012 Edition) EISMASSim Description

http://www.multiagentcontest.org/2012/

Tristan Behrens Jürgen Dix Jomi Hübner
Michael Köster Federico Schlesinger

April 17, 2012

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 a 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-1.0.jar`. The first jar contains the *generic* environment-interface, the second one contains the *specialized* one.

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*

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

`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:

```
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
}
```

[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.

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 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.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaceConfig scenario="mars2012" host="localhost" port="12300"
scheduling="yes" times="no" notifications="no" timeout="5000"
statisticsFile="yes" statisticsShell="yes">
  <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"/>
    <entity name="vehicle5" username="a5" password="1" xml="yes" iilang="yes"/>
    <entity name="vehicle6" username="a6" password="1" xml="yes" iilang="yes"/>
    <entity name="vehicle7" username="a7" password="1" xml="yes" iilang="yes"/>
    <entity name="vehicle8" username="a8" password="1" xml="yes" iilang="yes"/>
    <entity name="vehicle9" username="a9" password="1" xml="yes" iilang="yes"/>
  </entities>
</interfaceConfig>\todo{here must be 20}
```

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 the time being the only value that is accepted is `"mars2012"`.
- `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. 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 Mars-Scenario

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. A parameter is either an identifier (`<Identifier>`), that is a String, or a numeral(`<Numeral>`).

Here 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):

- `attack(<Identifier>)` attacks a vehicle.
- `buy(<Identifier>)` buys an item.
- `goto(<Identifier>)` moves to a vertex.

- `inspect` inspects some visible vehicles.
- `parry` parries all attacks.
- `probe` probes the current vertex.
- `recharge` recharges the vehicle.
- `repair(<Identifier>)` repairs a vehicle.
- `skip` does nothing.
- `survey` surveys some visible edges.

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

```
Action attack = new Action("attack", new Identifier("a2"));
```

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):

- `achievement(<Identifier>)` denotes an achievement.
- `bye` indicates that the tournament is over.
- `deadline(<Numeral>)` indicates the deadline for sending a valid action-message to the server in Unix-time.
- `edges(<Numeral>)` represents the number of edges of the current simulation.
- `energy(<Numeral>)` denotes the current amount of energy of the vehicle.
- `health(<Numeral>)` indicates the current health of the vehicle.
- `id(<Identifier>)` indicates the identifier of the current simulation.
- `lastAction(<Identifier>)` indicates the last action that was sent to the server.
- `lastActionParam(<Identifier>)`[4] indicates the parameter of the last action that was sent to the server.
- `lastActionResult(<Identifier>)` indicates the outcome of the last action.

[4] This is new in this version.

- `lastStepScore(<Numeral>)` indicates the score of the vehicle's team in the last step of the current simulation.
- `maxEnergy(<Numeral>)` denotes the maximum amount of energy the vehicle can have.
- `maxEnergyDisabled(<Numeral>)` denotes the maximum amount of energy the vehicle can have, when it is disabled.
- `maxHealth(<Numeral>)` represents the maximum health the vehicle can have.
- `money(<Numeral>)` denotes the amount of money available to the vehicle's team.
- `position(<Identifier>)` indicates the current position of the vehicle. The identifier is the vertex's name.
- `probedVertex(<Identifier>,<Numeral>)` denotes the value of a probed vertex. The identifier is the vertex's name and the numeral is its value.
- `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.
- `score(<Numeral>)` represents is the overall score of the vehicle's team.
- `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.
- `strength(<Numeral>)` represents the current strength of the vehicle.
- `surveyedEdge(<Identifier>,<Identifier>,<Numeral>)` indicates the weight of a surveyed edge. The identifiers represent the adjacent vertices and the numeral denotes the weight of the edge.
- `timestamp(<Numeral>)` represents the moment in time, when the last message was sent by the server, again in Unix-time.

- `vertices(<Numeral>)` represents the number of vertices of the current simulation.
- `visRange(<Numeral>)` denotes the current visibility-range of the vehicle.
- `visibleEdge(<Identifier>,<Identifier>)` represents a visible edge, denoted by its two adjacent vertices.
- `visibleEntity(<Identifier>,<Identifier>,<Identifier>,<Identifier>)` denotes a visible vehicle. The first identifier represents the vehicle's name, the second one the vertex it is standing on, the third its team and the fourth and final one indicates whether the entity is disabled or not.
- `visibleVertex(<Identifier>,<Identifier>)` denotes a visible vertex, represented by its name and the team that occupies it.
- `zoneScore(<Numeral>)` indicates the current score yielded by the zone the vehicle is part of.
- `zonesScore(<Numeral>)` indicates the current score of the vehicle's team yielded by zones, that is the sum of scores of all zones.

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