Environment Interface Standard for Agent-Oriented Programming
Platform Integration Guide for EIS v0.2

Tristan Behrens

December 14, 2009

1 Introduction

This document's intent is to give an overview on how to integrate EIS with your platform. This is supposed to be a document complemental to the EIS-javadoc. Things that you will not find in this document, you will find in the javadoc. For the general motivation behind EIS please refer to our technical report[1].

2 Integration

In this section we will provide an tutorial about how to integrate EIS into your (agent programming) platform. We suppose that you have already downloaded the complete package.

The first thing that you have to do is adding EIS to the class-path of your project. The package contains the file `eis-0.2-lib.jar`, which includes all the Java-interfaces and -classes that are necessary for using environment-interfaces. Add this file to the class-path.

The next thing that you have to do is to employ the jar-loading-mechanism that comes with EIS. Specific environment-interfaces are distributed as jar-files. The class `eis.EILoader` should be used to load an environment-interface from a jar-file and instantiate it. Use the method `fromJarFile`:

```
EnvironmentInterfaceStandard ei = null;
try {
    ei = EILoader.fromJarFile(new File(jarFileName));
} catch (IOException e) {
    // TODO handle the exception
}
```

Note that you have to handle exceptions that are potentially thrown by the invocation of the method. Possible causes for failure could be that the file does not exist or that the version (EIS has a versioning-system) does not match the required one.

Now that you have successfully instantiated an environment-interface you have to register your agents. Since EIS is very agnostic when it comes to the type/structure/architecture of your agents you only have to register your agents by providing their names. The reason for this is the desired generality. So let us assume that you have some agent, which is represented by its name. You can register the agent like this:

```
String agentName = ... ; // the name of your agent
try {
    ei.registerAgent(agentName);
} catch (AgentException e) {
  // TODO handle the exception
}
```

[1] `http://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi0909behrens.pdf`

Again, you have to handle possible exceptions. The invocation fails if an agent with the same name has already registered. You can also unregister an agent if you want to cut it off from the environment-interface:

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

Here the invocation fails if the agent has not been registered to the interface.

But why do you have to register your agents to the environment-interface? You have to do so because the next thing we are going to do is associating agents with entities. We differentiate between agents, which we only assume to be software-agents, and controllable entities, which provide agents with sensory and effectoric capabilities. Due to the fact that we do not assume anything about the nature of an environment-interface, we have to make this distinction. Agents act and perceive through entities, entities facilitate the situatedness of agents. A nice example for an entity is a simulated elevator. An agent that controls that elevator-entity can make it move to a specific floor and perceive its current floor. In order to have an agent control an entity both have to be associated. Similar to agents, entities are only represented by their name (again a String). So assuming that you know that there is an entity with the name `"car1"`, you can make your agent control it like this:

```
try {
    ei.associateEntity(agentName, "car1");
} catch (RelationException e) {
  // TODO handle the exception
}
```

The invocation could fail, so you have to handle the exception. Note that this a very naive way to associate your agent with an entity, because it assumes that you know the name of the entity beforehand. You can however query the interface for all free entities and associate your agent with the first one:

```
LinkedList<String> ens = ei.getFreeEntities();
try {
    ei.associateEntity(agentName, ens.removeFirst());
} catch (RelationException e) {
  // TODO handle the exception
}
```

Which policy you apply here is your decision. There are more methods for manipulating the agents-entities-relationship (see the javadoc). Note that you can also query the type of an entity with the method `getType`. This could be useful for example if you want to instantiate different types of agents for different types of entities.

Now that you have your agent registered and associated with an entity, or you have already iterated the process and associated several agents with several entities, you want to make them act and perceive. Acting is quite simple. You have to invoke the method `performAction` like this:

```
Action action = ... // this has to be an EIS-action
try {
    Vector<Percept> ps = eis.performAction(agentName, action);
} catch (ActException e) {
  // TODO handle the exception
} catch (NoEnvironmentException e) {
  // TODO handle the exception
}
```

The action must be an instance of `eis.iilang.Action`. You could for example instantiate an action like this:

```
Action action = new Action("goto", new Ident("up"));
```

It might be necessary to implement a mapping from your definition of what an action is to EIS-actions. Note that performing an action could return a percept. This is necessary for active sensing. Make sure that such return-values are handled properly.

Note that you can sometimes (depending on the environment-interface) associate a single agent with several entities. This can be reflected by `performAction` that accepts an optional array of strings (vararg language feature[2]) as the third parameter. The array should contain a subset of the set of entities that are associated with the agent:

```
try {
  Vector<Percept> ps = eis.performAction(agentName, action,"entity1","entity2");
} catch (ActException e) {
  // TODO handle the exception
} catch (NoEnvironmentException e) {
  // TODO handle the exception
}
```

If the array is empty, all entities will be taken into account. Note that you can determine the source (e.g. the entity) of each percept via the `getSource`-method.

You definitely are advised to handle the exceptions. `NoEnvironmentException` is thrown if the environment-interface is not properly connected to an environment. `ActException` is thrown if the action could not be executed. Possible reasons for that are reflected by the type of the exception:

```
try {
  ei.performAction(agentName, action,entities);
} catch (ActException e) {
  if( e.type == NOTYPE ) {
    // TODO handle the exception
  } else if( e.type == NOTREGISTERED ) {
    // TODO handle the exception
  } else if( e.type == NOENTITIES ) {
    // TODO handle the exception
  } else if( e.type == WRONGENTITY ) {
    // TODO handle the exception
  } else if( e.type == WRONGSYNTAX ) {
    // TODO handle the exception
  } else if( e.type == FAILURE ) {
    // TODO handle the exception
  }
} catch(NoEnvironmentException e) {
  // TODO handle the exception
}
```

The type `NOTYPEGIVEN` denotes that the type of the exception has not been indicated. Although we expect more detailed information about why the method as failed, we do not enforce this. `NOTREGISTERED` indicates that the agent has not registered to the environment-interface. `NOENTITIES` on the other hand communicates that the agent has no associated entities. `WRONGENTITY` denotes that at least one of the provided entities is not associated with the agent. `NOTSUPPORTEDBYTYPE` indicates that the type of the entity does not support

[2] `http://java.sun.com/developer/JDCTechTips/2005/tt0104.html`

the execution of the action. `WRONGSYNTAX` indicates that the syntax of the action is wrong. That is the case when the name of the action is not available and when the parameters do not match (number of parameters or their types and structure). And `FAILURE` indicates that the action has failed although it matched all mentioned requirements. For example `goto(up)` could fail if the path is blocked in the respective direction.

Now let us talk percepts. There is a method to retrieve all percepts. This has been shown to be very useful for some APL platforms. You can do this:

```
try {
  LinkedList<Percept> percepts = ei.getAllPercepts(agentName);
  // TODO process the percepts
} catch (PerceiveException e) {
  // TODO handle the exception
} catch (NoEnvironmentException e) {
  // TODO handle the exception
}
```

After the invocation you have to make sure that the percepts are processed in a proper manner. Also a `PerceiveException` is thrown if perception fails, that is if the agent is not registered or has no associated entities. An instance of `NoEnvironmentException` is thrown if there is no environment. Similar to `performAction` the method `getAllPercepts` supports a vararg for restricting the call to a subset of the associated entities.

Now let's talk about the third and final way to get percepts from the environment-interface: percepts-as-notifications. EIS supports sending percepts to the agents on special occasions without a request to do so. That is, environments sending percepts. In order to allow your agents to receive such percepts, your platform has to implement the interface `eis.AgentListener` and its method `handlePercept(String agent, Percept percept)`. Furthermore you have to register the listener to the environment-interface. The string `agent` of the `handlePercept`-method indicates the recipient of the percept `percept`. Note that it is your responsibility to make sure that the percept is passed to the respective agent.

You can establish percepts-as-notifications like this:

```
class YourPlatform implements AgentListener {

  EnvironmentInterface Standard ei;

  ...

  public void init() {
    eis.attachAgentListener(agentName, this);
  }

  public void handlePercept(String agent, Percept percept) {
    // TODO pass the percept to the agent
  }

}
```

Now we will discuss *environment-events*. Such events are generated if 1. the set of entities changes or is modified, an 2. if the executional-state of the environment changes. Again you have to implement the interface `eis.EnvironmentListener` and its methods:

```
class YourPlatform implements EnvironmentListener {

  EnvironmentInterface Standard ei;
```

```
  ...

  public void init() {
    eis.attachEnvironmentListener(this);
  }

  public void handleFreeEntity(String entity) {
    // TODO handle event
  }

  public void handleNewEntity(String entity) {
    // TODO handle event
  }

  public void handleDeletedEntity(String entity) {
    // TODO handle event
  }

  public void handleEnvironmentEvent(EnvironmenEvent event) {
    // TODO handle event
  }

}
```

The method `handleNewEntity` is called when there is a new entity, wheras `handleFreeEntity` is called when an entity is freed, and `handleDeletedEntity` is called when an entity is deleted. Again, you have to come up with your own platform-specific policy for new/free/deleted entities. We will come back to `handleEnvironmentEvent` in a minute.

Finally we will discuss methods of environment-management. For managing the environment you can use the method `manageEnvironment`:

```
EnvironmentCommand command = ...;
try {
    ei.manageEnvironment(command);
} catch (ManagementException e) {
    // TODO Auto-generated catch block
} catch (NoEnvironmentException e) {
    // TODO Auto-generated catch block
}
```

An environment-command can either be: starting the environment, killing it, pausing its execution, resetting it, and initializing it with parameters. A `ManagementException` is thrown when the command passed as a parameter is not supported. We do not assume that all environments support all environment-commands (if any at all). A `NoEnvironmentException` is thrown when the environment-interface is not connected to an environment.

The environment-interface can also notify about the change of the state of the execution of the environment. Such an event can either be that the environment has been started, killed, paused, reset, or initialized. Note that we do not assume that all environment-interfaces notify about such events.

3 Included environment-interfaces

Wherein we elaborate on environment-interfaces that are included in the EIS-package.

3.1 Agent Contest Connector 2009

In order to run the contest environment you have to download the package including the MASSim-server from the Multi-Agent Contest homepage[3].

Environment description: the environment is a grid-like, partially-accessible world. Cowboys are steered by agents. The goal is to push cows into a corral by frightening them. More information is available at the contest homepage.

Jar-file: `eis-acconnector2009-0.2.jar` (included in the EIS-package).

Entities:

`connector1,...,connector10` each one is a connector to a single cowboy in the environment.

Types of entities: this interface does not take into account different types of entities.

Actions:

`connect(Identifier, Numeral , Identifier, Identifier)` connects to the MASSim-server. The first identifier is the hostname of the server. The numeral is its port. The second identifier denotes the user-name, the final one denotes the password. This action has to be performed successfully in order to allow for other actions. Example: `connect("139.174.100.201",12300,"agentred1","dfkj39")`.

`move(Identifier direction)` moves the cowboy to a specified direction. Possible actions are `north`, `northeast`, `east`, `southeast`, `south`, `southwest`, `west`, and `northwest`. Example `move(east)`

`skip` has no effect.

Percepts: all those percepts are both propagated as notifications and returned by the `getAllPercepts`-method. Note that the interface implements a queue of percepts that is filled every time a message from the MASSim-server is received and whose first entry is retrieved every time the `getAllPercepts`-method is called. The interface does not hold a world-model.

`connectionLost` indicates that the connection to the server has been lost.

`simStart` indicates that the simulation has begun.

`corralPos(Numeral, Numeral, Numeral, Numeral)` is the position of the corral the first two numbers indicate the upper-left- the last two ones indicate the lower-right-corner. Example: `corralPos(1,1,10,10)`.

`gridSize(Numeral, Numeral)` represents the size of the grid. The first value is the width, the second one is the height. Example:`gridSize(100,100)`.

`simId(Identifier id)` denotes the id of the simulation. Example: `simId("cowSkullMountain")`

`lineOfSight(Numeral num)` indicates how far the respective entity can see. Example: `lineOfSight(8)`

`opponent(Numeral name)` denotes the opponent in the current match. Example: `opponent("StampedeTeam")`

`steps(Numeral num)` indicates how many steps the simulation lasts. Example: `steps(1000)`

`simEnd` indicates that the current simulation is over.

`result(Identifier)` represents the result of the simulation. Values could be either `win`, `lose`, or `draw`. Example: `result(win)`

`finalScore(Numeral)` represents the final-score of the simulation. Example: `finalScore(42)`

[3] `http://www.multiagentcontest.org`

`bye` indicates that the overall tournament is over.

`cell(Numeral, Numeral, Identifier)` denote the content of a cell. The numerals represent the position relative to the cowboy's current position. The identifier represents the object. Possible values are `agentally`, `agentenemy`, `switch`, `fenceopen`, `fenceclosed`, `cow`, `obstacle`, `empty`, and `unknown`. Example: `cell(-1,-1,cow)`

`pos(Numeral x, Numeral y)` denotes the current position of the cowboy. Example: `pos(10,15)`

`currentScore(Numeral)` denotes the current score. Example: `currentScore(24)`

`currentStep(Numeral num)` indicates the current step of the simulation. Example: `currentStep(123)`

Environment-management: not supported.

3.2 Carriage Example

Environment description: there is a carriage on a circular-track. That track has three distinct locations for the carriage two be on. On each side of the carriage is a robot. Both robots can push the carriage. The environment evolves in a step-wise manner.

Jar-file: `eis-carriage-0.2.jar` (included in the EIS-package).

Entities:

`robot1`,`robot2` are the two robots in the environment.

Types of entities: this interface does not take into account different types of entities.

Actions:

`push` pushes the carriage. If both robots push at the same time, the carriage will not move. If only one robot pushes the carriage will.

`wait` has no effect.

Percepts:

`step` indicates that current step of the environment. Propagates via notifications

`currentPos(Number)` denotes the current position of the carriage, either `0`, `1`, or `2`. Returned by `getAllEntities`.

Environment-management: not supported.