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

Tristan M. Behrens

January 13, 2010

1 Introduction

This document's intent is to provide a tutorial for creating environment interfaces for arbitrary environments. Also it provides a template for documenting your environment-interfaces when making them available.

2 Environment Interface Implementation Guide

We would like to coin a new term: *EISification* is the process of taking a given environment, adapting it to support the Environment Interface Standard, and distributing the result.

The overall GOAL is to take your environment (let it be some already existing one or one that has to be developed), EISify it and then deploy it as a jar-file, for others to use it. EISification means creating an environment-interface-class – this is the *main-class* – that wraps your environment or connects to it.

These are the essential steps:

1. set up your project and add EIS to the class-path. The current version contained in `eis-0.2-lib.jar`.
2. create the main-class that either implements the standard-interface (`eis.EnvironmentInterfaceStandard`) **or** extends the default-implementation (`eis.EIDefaultImpl`).
3. create a jar-file from your classes and specify the main-class in the manifest-file.
4. make the jar-file available.

We recommend using the default-implementation over using the standard-interface.

You can add the jar-file to the class-path directly: Alternatively, if your project supports Maven[1], you can add EIS as a dependency to your `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>apleis</groupId>
    <artifactId>eis</artifactId>
    <version>0.2</version>
    <scope>compile</scope>
  </dependency>
  ...
</dependencies>
```

[1] http://maven.apache.org/

2.1 Using the default-implementation

The first thing you would do is create your main-class and let it extend class `eis.EIDefaultImpl`:

```
package yourproject;

import eis.*;

public class MyEnvironmentInterface extends EIDefaultImpl {
  // TODO implement the abstract methods
}
```

The default-implementation already implements all needed functions. You only have to implement methods that are specific to your environment-interface.

You have to implement the abstract method `isConnected`:

```
public boolean isConnected() {
  // TODO implement
}
```

This method is supposed to return `true` if the environment is connected to the environment-interface and `false` otherwise.

Also you have to implement `getAllPerceptsFromEntity`:

```
public LinkedList<Percept> getAllPerceptsFromEntity(String entity)
    throws PerceiveException, NoEnvironmentException {
  // TODO implement
}
```

This method should return all percepts of the entity `entity`.

Also, you should employ percepts-as-notifications wherever possible. Such percepts are supposed to be sent to agents via instances of `eis.AgentListener` that have been registered to the interface via its `attachAgentListener`-method. The default implementation provides the method `notifyAgents` that allow you to send a percept to one, several, or all agents. And the method `notifyAgentsViaEntity` allows for sending percepts to all agents that are associated to one, several or all entities.

Now we have to discuss the definition and executions of actions. For each action with a fixed named and fixed parameters you have to implement a method. For example assume that you have a `goto`-action with a parameter that determines the direction you would implement:

```
public Percept actiongoto(String entity, Ident dir) throws ActException,
    NoEnvironmentException {
  // TODO implement
}
```

Note that this is a convention. The action-name itself is mapped to a method-call via Java-reflection. However if you prefer your own custom mechanism for executing actions feel free to overwrite the `performAction`-method.

Before discussing other methods, we have to say something about the exceptions that can be thrown in the likely event that an action fails. An instance of `NoEnvironmentException` should be thrown if the environment-interface is not connected to an environment. If there is a connection an instance of `ActException` should be thrown. Please heed that that exception-class is typed, that is it communicates more detailed information about the action-failure by carrying a type that can be queried when the exception is caught. Do it like this:

```
// the syntax of the action is wrong
throw new ActException( ActException.WRONGSYNTAX );
```

The type `NOTSPECIFIC` is the default type of `ActException`. We strongly discourage you from using this one. We expect you to provide more detailed information about why the method has failed. `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 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.

The next method is `release` which is supposed to disconnect the environment-interface from the environment:

```
public void release() {
    // TODO release the environment
}
```

After invoking that method `isConnected` should always return `false`.

Finally, let us discuss managing the environment. You should allow for managing the environment by implementing the method `manageEnvironment`:

```
public void manageEnvironment(EnvironmentCommand command)
        throws ManagementException {
    // TODO implement environment-management
}
```

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 explicitly do note make obligatory that you should implement all environment-commands. If your environment for example does not support being paused than you do not have to implement the respective commands. A `NoEnvironmentException` is thrown when the environment-interface is not connected to an environment.

2.2 Using the interface

Instead of extending the default-implementation you can also implement the standard-interface. The first thing you would do is create your main-class and let it implement the interface `EnvironmentInterfaceStandard`:

```
package yourproject;

import eis.*;

public class MyEnvironmentInterface implements EnvironmentInterfaceStandard {

}
```

This is of course not everything. You have to implement the interface's methods. Quite some of them, to be honest. Please have a look at the default-implementation for some inspiration about how to implement those.

The first thing you should do is attaching and detaching environment- and agent-listeners. We assume that you internally store a list of registered listeners. For example you could use like in the default implementation `Vector<EnvironmentListener>` and `ConcurrentHashMap<String,HashSet<AgentListener>>` respectively. But feel free to use anything that fits your needs Environment-listeners are used to inform observers about the change of the state of execution of the environment. Every observer that is interested in such events should register via:

```
public void attachEnvironmentListener(EnvironmentListener listener) {
  // TODO store the listener internally
}
```

It should also be possible to remove an observer:

```
public void detachEnvironmentListener(EnvironmentListener listener) {
  // TODO remove the listener from the internal representation
}
```

For the agent-listeners it is almost the same. These are used to send percepts-as-notifications to the agents. Here you should store for each agent a set of listeners:

```
public void attachAgentListener(String agent, AgentListener listener) {
  // TODO store the listener internally
}
```

Again, removing the listener should also be allowed:

```
public void detachAgentListener(String agent, AgentListener listener) {
  // TODO remove the listener from the internal representation
}
```

Of course when implementing your specialized interface you should implement means to notify the listeners. Employ agent-listeners to send percepts to agents, and use environment-listeners to send environment events. Compare with the `notify*`-methods of the default implementation.

After that you should provide functionality that allow for (un)registering and unregistering agents, and for (dis)associating agents with entities. To begin it would be good to set-up an internal list of agents, another one for entities, and some map for associating agents and entities. For example you could use `LinkedList<String>` for the lists and `ConcurrentHashMap<String,HashSet<String>>` for the mapping.

Please allow for registering agents:

```
public void registerAgent(String agent) throws AgentException {
  // TODO store internally
}
```

You should throw an `AgentException` if the agent has already registered.

Then allow for unregistering:

```
public void unregisterAgent(String agent) throws AgentException {
  // TODO remove form internal representation
}
```

Here you should throw an `AgentException` if the agent has not registered.

Also make sure that the list of agents can be retrieved:

```
public LinkedList<String> getAgents() {
  // TODO return the list of agents
}
```

And make sure to to the same for entities as well:

```
public LinkedList<String> getEntities() {
  // TODO return the list of entities
}
```

Then associate an agent with an entity

```
public void associateEntity(String agent, String entity) throws RelationException {
  // TODO update the mapping
}
```

Make sure to throw an `RelationException` if associating the agent with the entity is not possible. This is the case for example when the agent or the entity are not stored in the internal lists.

After that allow for freeing an entity from all associations with agents:

```
public void freeEntity(String entity) throws RelationException {
  // TODO update the mapping
}
```

Here you should throw an `RelationException` if the entity could not be freed. That is when is not contained in the internal list of entities.

Do the same for an agent:

```
public void freeAgent(String agent) throws RelationException {
    // TODO update the mapping
}
```

Then remove a specific agent-entity-pair from the mapping:

```
public void freePair(String agent, String entity) throws RelationException {
  // TODO update the mapping
}
```

Again throw an `RelationException` if the operation fails.

After manipulating the agents-entities-relation it would be useful to allow for querying the data-structures. You should provide a method that returns the entities associated to an agent:

```
public HashSet<String> getAssociatedEntities(String agent) throws AgentException {
}
```

Make sure to throw an `AgentException` if the agent is not registered to the interface.

And you should provide the same for an entity:

```
public HashSet<String> getAssociatedAgents(String entity) throws EntityException {
}
```

Here you should throw an `EntityException` if the entity has not been added to the interface.

Finally return the list of free entities, that is a list of entities that are not associated:

```
public LinkedList<String> getFreeEntities() {
}
```

Also it is necessary to return the type of an entity:

```
public String getType(String entity) throws EntityException
```

Throw an `EntityException` if the entity does not exist.

Now we have to discuss acting and perceiving. Entities are the ones that provide agents with sensory and effectoric capabilities. Agents act and perceive through entities.

This is the first essential method:

```
public LinkedList<Percept> performAction(String agent, Action action, String...entities)
throws ActException, NoEnvironmentException {
  // TODO perform the action and return a percept
}
```

It should allow an agent to act through a set of his associated entities provided as an array. If the array is empty all associated entities should perform the action. Here you need to throw an `ActException` if the action fails, that is when one or more of the entities failed to execute the action or of one or several of the provided entities are not associated. And you need to throw an `NoEnvironmentException` of the environment-interface is not connected to an environment. Note that the return-value is also interesting. The method can also be used to facilitate active-sensing. Some actions might just return something simple like a `"success"`-Percept or something very sophisticated. Finally you should make sure to indicate the origin-entity of each percept via the `setSource`-method of `Percept`.

You should also implement this method for retrieving all percepts:

```
public LinkedList<Percept> getAllPercepts(String agent, String...entities)
    throws PerceiveException, NoEnvironmentException {
  // TODO return all percepts
}
```

This method is supposed to return all percepts of the entities that are associated with an agent. Again the associated entities are provided as an array. If the array is empty all entities are used for sensing. The method fails with an `PerceiveException` if perceiving through one or several entities is not possible, or of one or several of the provided entities are not associated. The `NoEnvironmentException` is thrown if no environment is connected.

```
what happens once you implement the interface? what do you have to do?
  you have to implement a couple of methods
    isConnected
    manageEnvironment
    release
what is which method supposed to do?
```

3 Environment Interface Documentation Policy

In order to ensure transparency and accessibility when publishing your environment-interfaces, you should provide a documentation. That documentation should contain all necessary information, including 1. a description of the environment, 2. the name of the jar-file that contains the environment-interface, 3. the names of the entities that populate the environment, 4. the types of those entities, 5. the actions of the entities, and 6. the percepts.

Please provide a description of the environment:

Environment description: the environment is a simple 3-dimensional world with a ground level. It is populated by jeeps that are not controllable entities. Controllable entities are unmanned vehicles, that should be used to locate the jeeps.

After that you should say, in which jar-file the environment-interface is contained, and optionally where to find that file:

Jar-file: `uv-simulation.jar`

Now you should give an overview of the different entities that populate the environment. Please provide their names and their characteristics:

Entities:

`uv1`,`uv2`,... are several unmanned ground vehicles. There are 100 in the simulation.

Please provide the types of entities as well:

`groundvehicle` these are unmanned ground vehicles.

`airvehicle` these are unmanned aerial vehicles.

Now, denote and describe the different actions that are supported. Please make sure to include the parameters of the actions and their meaning. And do not forget to mention, what kind of percepts an action would return, if it was a sensing action

Actions:

`move(Identifier)` moves the entity into a specific direction. Possible directions are: `north`, `east`, `south`, and `west`. Example: `move(east)`

`useCamera` uses the camera and returns the most prominent, visible object.

`liftOff` lifts an entity off the ground. Only available to aerial vehicles.

`land` lands and entity. Only available to aerial vehicles.

Now describe the different percepts. Again please describe the possible parameters and their meaning. Also explain how the different percepts are made available. Do you retrieve an agent via the `getAllPercepts`-method, by a notification, or by both?

Percepts:

`time(Numeral)` indicates the current time-stamp of the simulation. Returned by `getAllPercepts` and send as a notification every second.

`currentPos(Number,Number,Number)` denotes the current position of the vehicle in the three dimensions of space. Returned by `getAllPercepts` and send as a notification every second.

And finally make clear, which means for environment-management are supported:

Environment-management: the environment can be initialized with a parameter that denotes that map that should be used. All other environment-commands are not supported.