The Environment Interface Standard for Agent-Oriented Programming Platform Integration Guide and Interface Implementation Guide

Tristan M. Behrens, Jürgen Dix, Koen V. Hindriks

IfI Technical Report Series IfI-09-10

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Michael Köster
Contact: michael.koester@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

The Environment Interface Standard for Agent-Oriented Programming Platform Integration Guide and Interface Implementation Guide

Tristan M. Behrens, Jürgen Dix, Koen V. Hindriks

Contents

1 Overview

Agents act and perceive in environments. Although there exist many existing environments for agents – ranging from testbeds to commercial applications – these environments have not been widely shared because of the difficulty of interfacing agents with these environments. A more generic approach for connecting agents to environments would be beneficial for several reasons. It would facilitate reuse, comparison, the development of truly heterogeneous agent systems, and increase our understanding of the issues involved in the design of agent-environment interaction. To this end, we *design and develop a generic environment interface standard*, called EIS. Our design has been guided to some extent by existing agent programming platforms. These platforms are not only suitable for developing agents but also already provide some support for connecting agents to environments. The interface standard itself is generic, however, and does not commit to any specific platform features. The interface proposal has been implemented and evaluated in a number of agent platforms. We aim at a de facto standard that will transform into a real standard in the near future.

EIS has already been implemented and tested with success. This document contains two implementation guides, which are companion-documents to the released-software:

1. the *platform integration guide*, which shows how to establish the connection between already existing platforms, and

2. the *interface implementation guide*, which explains how to *EISify* environments, that is making environments available that adhere to the standard.

2 Platform Integration Guide

2.1 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].

2.2 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. Alternatively, if your project supports Maven[2] 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>
```

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
}
```

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

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

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
}
```

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 the method `performAction` that accepts an optional array of strings (vararg language feature[3]) 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. The specific exception `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
}
```

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

The type `NOTSPECIFIC` denotes that the type of the exception has not been indicated specifically. Although we expect more detailed information about why the method has 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 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 specific 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 the respective method `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 raised 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.

2.3 Included environment-interfaces

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

2.3.1 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[4].

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

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)` instantiates a connection to the MASSim-server. The first identifier is the host-name 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 FIFO 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. An example would be: `gridSize(100,100).`

`simId(Identifier id)` denotes the id of the simulation. An example would be: `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)`

`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. An example would be: `currentScore(24)`

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

Environment-management: not supported.

2.3.2 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.

3 Interface Implementation Guide

3.1 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.

3.2 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 (refer to `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[5], 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>
```

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

3.3 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`.

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 conventions. 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.

3.3.1 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 the data-structure `Vector<EnvironmentListener>` and also the thread-safe mapping `ConcurrentHashMap<String,HashSet<AgentListener>>`. 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
}
```

After that you should provide functionality that allow for (un)registering and unregistering agent, 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 the thread-safe mapping `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 {
  // TODO return the associated entities
}
```

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 {
    // TODO return the associated agents
}
```

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() {
 // TODO return the free entities
}
```

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

```
public String getType(String entity)
  throws EntityException {
  // TODO return the type of the entity
  }
```

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.

3.4 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.

4 Conclusion

Up to now, several platforms have already established support for EIS: 2APL [6], GOAL [8], JADEX [7], and JASON [3] It has been proved that connecting these platforms is very easy, it could have been established in one day's time. A common trait of is the use of a two-way-converter between EIS-data-structures to platform-specific ones.

Also up to now, several environment-interfaces have been created: two connector to the Agent-Contest-server[5], one for 2009 and a second one for 2010, the elevator example[1], the well-known carriage-example[4], and the Wumpus-world[9] (will be contained in the upcoming GOAL-release). An environment-interface to the computer-game UnReal Tournament 2004[2] is on its way.

References

[1] Elevator simulator homepage. `http://sourceforge.net/projects/elevatorsim/`.

[2] R. Adobbati, A.N. Marshall, A. Scholer, S. Tejada, G.A. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[3] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[4] Nils Bulling and Wojciech Jamroga. Rational play and rational beliefs under uncertainty. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 257–264, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[5] Mehdi Dastani, Jürgen Dix, and Peter Novák. Agent contest competition - 3rd edition. In M. Dastani, A. Ricci, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Proceedings of ProMAS '07, Revised Selected and Invited Papers*, number 4908 in Lecture Notes in Artificial Intelligence, Honululu, US, 2008. Springer.

[6] Mehdi Dastani et al. *2APL Manual*. `http://www.cs.uu.nl/2apl/`.

[7] Braubach Lars, Pokahr Alexander, and Lamersdorf Winfried. Jadex: A BDI-agent system combining middleware and reasoning. In Von Rainer Unland,

Matthias Klusch, and Monique Calisti, editors, *Software agent-based applications, platforms and development kits*, 2005.

[8] Wouter Pasman. GOAL IDE user manual. `http://mmi.tudelft.nl/~koen/goal.php`.

[9] S. J. Russell and Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.