TU Clausthal

Clausthal University of Technology

Towards an Environment Interface Standard for Agent-Oriented Programming

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

IfI Technical Report Series IfI-09-09

Department of Informatics
Clausthal University of Technology

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

Towards an Environment Interface Standard for Agent-Oriented Programming

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

Contents

Abstract

Our aim is to *design and develop a generic environment interface standard* (EIS) that facilitates connecting agents programmed in various agent programming languages (APL) to environments. We aim at a de facto standard that eventually transforms into a real standard in the near future.

1 Introduction

Our objective is to *design and develop a generic environment interface standard* (EIS) that facilitates connecting agents programmed in various agent programming languages (APL) to environments. We aim at a de facto standard that, hopefully, becomes a real standard in the near future. Our motivation is based on the following considerations:

- implementing an EIS makes already working environments widely available (short-term goal),
- an EIS allows for the easy distribution of future environments (Multi-Agent Contest, Unreal, ORTS,...),
- an EIS allows the direct comparison of APL platforms, and
- an EIS enables the development of a truly heterogeneous MAS, consisting of agents from APL platforms that adhere to the standard of the EIS (long-term goal).

Our approach takes the following goals into account: to design an interface that is *as generic as possible*, and to *reuse as much as possible* from existing interfaces. Obviously, there is a trade-off between these two goals. Our basic strategy for designing a generic environment interface is (1) to start with what is currently "out there" in existing platforms, and (2) to try to merge this into a generic interface which is sufficiently close to these existing approaches.

Initially, in the first interface standard proposal, we will not introduce new features that go beyond existing functionality for relating APLs to environments. However, there is one exception: a feature that allows the connection between agents and environment to be a very dynamic one. We discuss this feature in more detail below. We leave the discussion of other features (that may be useful and related to connecting to environments) to future extensions of this paper.

But even for such an incremental approach, it is important that **everyone needs to adapt to the new standard**. Our strategy is to minimize the required effort for adapting to the new standard.

The paper is organised as follows. In Section 2 we introduce a meta model that describes the basic components, their interrelations and functionalities between them. This model is used in Section 3 to define our *environment interface*

standard (EIS), the main result of this paper. Finally, in Section 4, we show with several examples how to connect particular environments to environment interfaces.

2 The Meta-Model

In this section, we compare different APL platforms with respect to how they facilitate accessing different environments. We also propose a meta-model that provides the starting point for defining a generic interface. Finally, we suggest a set of six principles that will guide the development of the EIS in Section 3.

2.1 Summaries

We give an overview of the APL platforms 2APL, GOAL, JADEX, and JASON. We will concentrate on the following questions:

1. How can agents be connected to environments?
2. How can agents act and perceive in an environment?
3. What other useful functions are available?

The answers to these questions will be used to define a meta-model and to derive principles for defining an environment interface standard.

2.1.1 2APL Environments

Creating new environments in 2APL means to implement a class `Env` [3], that extends `apapl.Environment` (see below). The package name defines the environment name. Environments are distributes as jar-files. Agents can be associated with several environments, jars have to be in the user-directory. From an implementation point of view, there is a class `APAPLBuilder` that is used to parse MAS-files, in order to load and run agents and environments. Furthermore there is a derivate of the class `apapl.Executor` that executes agents.

Here is the abstract environment-class:

```
public abstract class Environment {

    private HashMap<String,APLAgent> agents = new HashMap<String,APLAgent>();
    public final void addAgent(APLAgent agent) { ... }
    public final void removeAgent(APLAgent agent) { ... }
    protected abstract void addAgent(String name);
    protected abstract void removeAgent(String name);
    protected final void throwEvent(APLFunction e, String... receivers) { ... }
    public final String getName() { ... }
    public void takeDown() { ... }

}
```

- `addAgent(APLAgent agent)` adds an agent to the environment and stores its name and object in the hash-map. It is called by the class `APAPLBuilder`. Cannot be overridden.
- `removeAgent(APLAgent agent)` removes an agent from the environment. It is called by the class `APAPLBuilder`. Cannot be overridden.
- `addAgent(String name)` should be overwritten while inheriting from the environment. It is called by the environment itself.
- `removeAgent` should be overwritten while inheriting from the environment. It is called by the environment itself.
- `throwEvent` sends an event to a set of agents. Cannot be overwritten.
- `getName` returns the name of the environment. Cannot be overridden.
- `takeDown` is called to release the resources of the environment.
- For implementing external-actions you have to implement for each such action a method with the signature `Term actionName(String agent, Term... params)`. These methods are called by the agent-executor.

An observation is that the environment stores agents as objects. Furthermore there is a format for exchanging data (perceive/act) between agents and the environment, based on the class `apapl.data.Term`: `APLIdent` for constants, `APLNum` for numbers, `APLFunction` for functions, and `APLList` for lists.

2.1.2 GOAL Environments

To use a GOAL-environment, the user has to copy a jar-file or a folder with class-files to a convenient location (e.g. the folder containing the MAS-description) and adapt the MAS-file [7].

To function as an environment, a class has to implement the Java-interface `goal.core.env.Environment` and implement the methods defined therein.

Agents are executed by a scheduler that invokes the mentioned methods.

Here is the environment-interface:

```
public interface Environment {

    public boolean executeAction(String pAgent, String pAct) throws Exception;
    public ArrayList<Percept> sendPercepts(String pAgentName) throws Exception;
    public boolean availableForInput();
    public void close();
    public void reset() throws Exception;

}
```

- `executeAction` is called by the scheduler in order to execute an action. The first parameter is the respective agent's name, the second one is a string that encodes the action. The method returns true if the action has been recognized by the environment, false if not. An exception is thrown if the action has been recognized by the environment but its execution has failed.
- `sendPercepts` is called by the scheduler to retrieve all observations of an agent. The parameter is the respective agent's name. The method returns a list of percepts. It throws an Exception if retrieving the observations has failed.
- `availableForInput` is called by the scheduler to determine whether the environment is ready for accepting input or not.
- `close` is called by the platform-manager to shut down the environment.
- `reset` is called by the platform-manager to reset the environment. It throws an exception if the reset has failed.

Note that the IDE user manual explicitly states that `executeAction` needs not to be thread-safe, i.e. the scheduler is supposed to ensure thread-safety.

2.1.3 Jadex Environments

In JADEX [5], agents are composed of beliefs, goals and plans, that are Java-objects. XML-based Agent Definition Files glue together initial instances of those mental attitudes.

Associating an agent with an environment is usually done by putting the environment into the belief base, either as a set of facts representing the environment-state, or as a single environment-object encapsulating the state. The environment objects are typically part of the agents' beliefs and when they change the agents automatically notice this (via property changes).

There are two ways of associating several agents with a single environment: (1) sharing a singleton environment-object, or (2) implementing an agent, that manages the state and the evolution of the environment and allows other agents to act and perceive by message-passing. A singleton environment is shared by the agents and accessed via normal method calls. These calls are synchronized within the environment object. An environment agent manages the environment object. This allows a system distribution as actions/percepts are transferred via messages to/from the environment agent.

In JADEX the normal Java class-path is used for loading all kinds of resources, i.e. if the class file is contained in a jar and that jar file is in the class-path.

Since JADEX does have a strict policy when it comes to connecting to environments we will show an example. This is a code-snippet of the garbage-collector-agent:

```
<agent [...]>

    <beliefs>
        <!-- Environment object as singleton.
             Parameters are name and type of agent for adding it
            No clean solution but avoids registering of agents.-->
        <belief name="env" class="Environment">
            <fact>
              Environment.getInstance(Environment.COLLECTOR, $scope.getAgentName())
            </fact>
         </belief>

        <!-- The actual position on the grid world. -->
        <belief name="pos" class="Position" evaluationmode="push">
            <fact language="clips">

                ?agent = (agent (agent_has_localname ?agentname))
                ?rbel_env = (belief (element_has_model ?mbel_env) (belief_has_fact ?env))
                ?mbel_env = (mbelief (element_has_name "env"))
                ?env = (jadex.bdi.examples.garbagecollector.Environment (
                     getPosition (?agentname) ?ret))
            </fact>
         </belief>

  [...]

    </beliefs>

    [...]

</agent>
```

As we can see, the environment is stored in the belief-base as a Java-object.

2.1.4 Jason Environments

To create a new environment a class has to be established extending the class `jason.environment.Environment`[1]. Environments are distributed as jar-files. Each MAS has at most one environment. The jar-file has to reside in the user directory. Agents are executed using infrastructures (e.g. centralised of Jade). Infrastructures also load agents and environments.

Here is the class:

```
public class Environment {

    private static Logger logger = Logger.getLogger(Environment.class.getName());
    private List<Literal> percepts =
      Collections.synchronizedList(new ArrayList<Literal>());
    private Map<String,List<Literal>>  agPercepts =
      new ConcurrentHashMap<String, List<Literal>>();
    private boolean isRunning = true;
    private EnvironmentInfraTier environmentInfraTier = null;
    private Set<String> uptodateAgs = Collections.synchronizedSet(new HashSet<String>());
    protected ExecutorService executor;

    public Environment(int n) { ... }
    public Environment() { ... }
    public void init(String[] args) { ... }
    public void stop() { ... }
```

```
    public boolean isRunning() { ... }
    public void setEnvironmentInfraTier(EnvironmentInfraTier je) { ... }
    public EnvironmentInfraTier getEnvironmentInfraTier() { ... }
    public Logger getLoger() { ... }
    public void informAgsEnvironmentChanged(Collection<String> agents) { ... }
    public void informAgsEnvironmentChanged() { ... }
    public List<Literal> getPercepts(String agName) { ... }
    public void addPercept(Literal per) { ... }
    public boolean removePercept(Literal per) { ... }
    public int removePerceptsByUnif(Literal per) { ... }
    public void clearPercepts() { ... }
    public boolean containsPercept(Literal per) { ... }
    public void addPercept(String agName, Literal per) { ... }
    public boolean removePercept(String agName, Literal per) { ... }
    public int removePerceptsByUnif(String agName, Literal per) { ... }
    public boolean containsPercept(String agName, Literal per) { ... }
    public void clearPercepts(String agName) { ... }
    public void scheduleAction(final String agName, final Structure action,
      final Object infraData) { ... }
    public boolean executeAction(String agName, Structure act) { ... }

}
```

- `Environment(int n)` and `Environment()` instantiate the environment with n threads to execute actions.
- `init(String[] args)` initializes the Environment. The method is called before the MAS execution. The arguments come from the MAS-file.
- `stop()` stops the environment.
- `isRunning()` checks whether the environment is running or not.
- `setEnvironmentInfraTier(EnvironmentInfraTier je)` and `getEnvironmentInfraTier()` set and get the infrastructure tier (saci, jade, centralised,...).
- `getLoger()` [sic!] gets the logger (not used).
- `informAgsEnvironmentChanged(Collection<String> agents)` informs the agents that the environment has changed.
- `informAgsEnvironmentChanged()` informs all agents that the environment has changed.
- `getPercepts(String agName)` returns the percepts of an agents. Includes common and individual percepts. Called by the infrastructure tier.
- `addPercept(Literal per)` adds a percept to all agents. Called by the environment.
- `removePercept(Literal per)` removes a percept from the common percepts. Called by the environment.

- `removePerceptsByUnif(Literal per)` removes all percepts from the common percepts, that match the unifier `per`.
- `clearPercepts()` clears the common percepts.
- `containsPercept(Literal per)` checks for containment.
- `addPercept(String agName, Literal per)` adds a percept to an agent. Called by the environment.
- `removePercept(String agName, Literal per)` removes a percept.
- `removePerceptsByUnif(String agName, Literal per)` removes all percepts matching the unifier `per`.
- `containsPercept(String agName, Literal per)` checks for containment.
- `clearPercepts(String agName)` clears all percepts.
- `scheduleAction(final String agName, final Structure action, final Object infraData)` is used to schedule an action for execution.
- `executeAction(String agName, Structure act)` executes an action `act` of the agent `agName`.

An observation is that the environment allows for (external) control over action-execution strategies and provides logging-functionality (redirecting system interface `System.out`). Note that although the environment defines these functions the two essential methods are `executeAction` and `getPercepts`, which provide a minimal agent interface.

2.1.5 Comparison

- **Restrictiveness/portability:** From the point of view of an agent-/environment-/MAS-developer, 2APL and GOAL are most restrictive, JASON is moderately restrictive and JADEX is not restrictive at all. To create agents in 2APL/ GOAL/JASON you have to provide jar-files (or also compiled Java-classes in the case of GOAL), that contain the environment. In the case of 2APL and JASON, creating an environment boils down to creating a class that inherits from an abstract environment-class, in GOAL on the other hand one has to implement an environment-interface. JADEX is absolutely open, one can plug-in almost everything. This is true to a certain degree for JASON as well, because JASON is (in comparison to 2APL and GOAL) open-source. One can implement environments without sticking to the instructions, but this does not seem to be the way intended by the developers.

Criterion	2APL	GOAL	Jadex	Jason
Portability	jar-files	jar-files	everything	jar-files
Perceiving	sense-actions and external events	getting all percepts via a provided method	accessing environment-objects or requesting percepts from an environment-agent	getting all percepts via a provided method
Acting	invoking methods	invoking a method	manipulating an enviro-ment object or sending a message to an environment agent	invoking a method
Abstract En-vironment Functionality	mapping from agent-names to agent-objects	no special functionality	no abstract environment defined	logging and action-scheduling
Formats	logical terms and atom encoded as Java-objects	strings	java-objects	logical literals and structures encoded as Java-objects
Java accessi-bility	jar-files	jar-files	everything that is in the class-path	jar-files

Table 1: Comparison-matrix to give an overview.

- **Perceiving:** In 2APL/ GOAL/JASON perceiving and acting means invoking special methods in the environment-class. 2APL allows for active and for passive sensing. An agent can perform a sense-action to get percepts, or the environment can send percepts by throwing events. In GOAL and JASON the only way to get percepts is to call special methods. This is usually done in the reasoning cycle of each agent.

  JASON also differentiates between individual (available to one agent) and global percepts (available to all agents). It also allows for switching between active and passive sensing in the MAS-specification files. In JADEX perceiving means either querying an environment that is stored as an object in the agents' belief-base, or by communicating with an agent that functions as an environment-agent.

- **Acting:** Acting in 2APL/ GOAL/JASON is done by calling special methods. In GOAL and JASON the action to be performed is a parameter of a special method, in 2APL the action-name is also the name of the special method. Executing an action-method in 2APL can have two outcomes. Either a return-value (an object) indicating success is returned, that might be non-trivial (e.g. list of percepts in the case of an sense-action) or terminate with an exception indicating action-failure. In GOAL invoking the execute-action-method might have tree outcomes. Either the return-value `true` indicating success, `false` indicating that the action has not been recognized, or an exception indicating that the action has failed. The JASON execute-action-method returns a `boolean`. In JADEX acting means either updating an environment that is stored as an object in the agents' belief-base, or again by communicating with an agent that functions as an environment-agent.

- **Functionality of the abstract environments:** The GOAL interface implements no standard functionality. The abstract environment-class of 2APL only implements a mapping from agent-names to agent-objects. The abstract environment-class of JASON on the other hand implements more sophisticated functionality, like support for multi-threaded action-execution, dealing with the environment infrastructure tier, notifiers for agents that the environment has changed . . . JADEX does not define any interface or abstract class for implenting environments.

- **Formats:** 2APL actions/percepts/events are instances of derivatives of the class `Term`. A GOAL-percept is an instance of the class `Percept`, an action is a Java-string. A JASON-percept is an instance of the class `Literal`, an action is an instance of `Structure`. In JADEX actions/percepts/events are arbitrary Java-objects.

- **Java-accessibility:** Accessing Java code in 2APL/ GOAL/JASON is possible through jar-files. In comparison to 2APL and GOAL, JASON allows for

internal-actions stored in a jar-file that does not contain an environment. Accessing Java code in JADEX is easy.

2.2 Meta Model

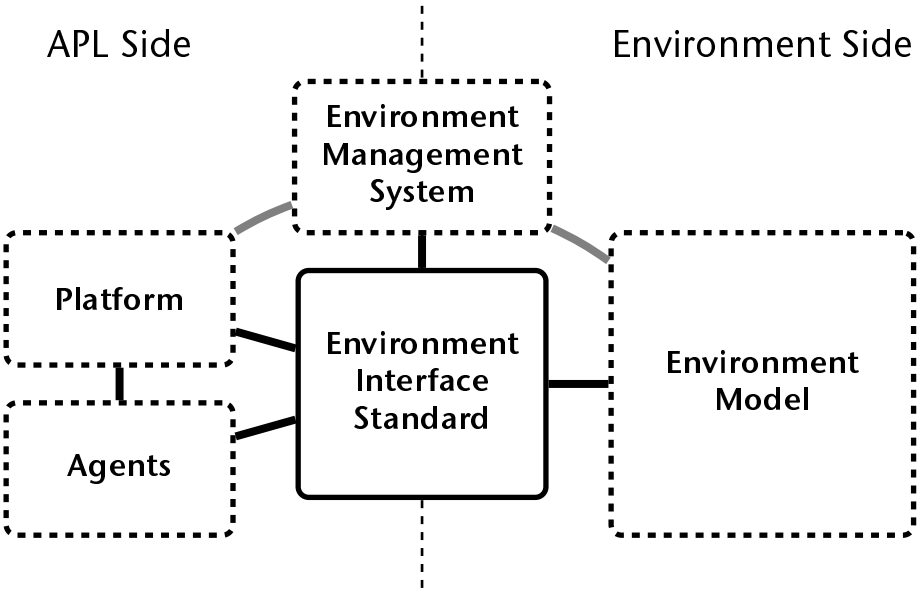

Figure 1: The components of our proposal.

In this section, we present an abstract model of what we would like to achieve. We intend this meta-model to be as general as possible. The main goal is to allow establishing connections between APL platforms and environments, by means of the functionality provided by the EIS. This functionality has yet to be defined. The main feature that we believe should be included in the standard at this point is an *agents-entities-relation*. This relation associates agents with what we have labeled *controllable entities* in the environment, i.e. it lets agents control entities in the environment.

We identify five components from a software engineering perspective (see Fig. 1):

- **Agent:** The objective of defining an environment interface standard is to provide a generic approach for connecting *agents* to environments. Agents may refer to almost any kind of software entity but the stance taken here is that these entities are able to act and process percepts. We use the following very generic definition taken from [8] that includes

precisely these two aspects: *An agent is anything that can be viewed as **perceiving** its **environment** through sensors and **acting** upon that **environment** through effectors.* We do not intend to restrict our proposal to any specific kind of agent, although we are primarily motivated by the AOP-perspective.

- **Environment model:** We assume an environment to contain *controllable entities*. Controllable entities establish the connection between agents and the environment by providing (1) *effectoric* capabilities and (2) *sensory* capabilities to agents, thus facilitating the *situatedness* of these agents.

  Such entities may be controlled from outside of the environment (by agents) and are capable of performing actions in the environment to change the state of that environment. We assume that each entity has its own repertoire of actions, and we do not assume anything about how the actions are performed in the environment.

  Similarly, we assume each entity to receive percepts that may be specific to that entity. See the paragraph about *perceiving* below for more details on different modes of perception that are supported. Note that we allow other active entities in the environment that are not controlled by agents. Finally, we assume that controllable entities can be created or removed from the environment.

  Controllable entities may be linked one-to-one to concrete Java-objects at the code level but need not be so. That is, entities may be *implicit* and we do *not* require that entities can be matched to particular Java-objects that are part of an environment. Entities thus primarily are used conceptually and refer to abstract containers for actuators and sensors to which agents can connect. The only representation that is obligatory for each controllable entity is an identifier.

  The model of the environment is illustrated in Figure 2. We assume (possibly) intersecting spheres of influence of multiple agents acting in an environment. The sphere of influence of an agent is defined by the effectoric and sensory ranges of its associated controllable entites. Note that we do *not* assume a one-to-one relation between agents and controllable entities. Different perspectives may be taken towards these spheres of influence: (i) an action perspective (agents may interfere with each other, change same parts of environment) and (ii) a perception perspective (agents may have different views on the environment).

  Controllable entities can be something very simple like thermostats or something quite complex like a robot. In the Multi-Agent Contest 2009, the cowboys are the controllable entities. The sensory capabilities are limited to some sort of camera, that provides agents with a limited visual range. The effectoric capabilities consist of moving the cowboy in different directions.

Finally, although it is natural to talk about states of the environment this should *not* be taken to imply that we impose any additional structure on an environment being e.g. a discrete state system. The environment model is generic and can be instantiated to all kinds of specific environments.

- **Platform:** We assume the platform to be responsible for instantiating and executing agents. Furthermore we assume that it facilitates connecting agents with environments, and associating agents to controllable entities in environments.
- **Environment management system (EMS):** We assume this component to provide all the actions for managing an environment. Such actions might be: initializing an environment using a configuration file, releasing the resources of the environment and kill it, furthermore actions like pausing, unpausing, and resetting. The environment management system may be run independently from an APL platform. However, our main concern is to define this component in an abstract way as a means to allow platforms to exert some control over the environment. Note that we propose the EMS to potentially be on both sides (we would like to leave this issue "open" to a certain extent). We believe that this will be clarified during practical experiments. Note also that we propose the EMS and its functions, but we do not define anything about it to be obligatory.
- **Environment interface standard (EIS):** The environment interface standard is the layer that connects the platform, the environment management system, and the agents with the environment(s).

Fig. 1 shows the meta-model. Connections are:

1. between the agents to the EIS (allows for acting and perceiving) (see below for an explanation on different modes of sensing),
2. between the platform to the EIS (allows for manipulating the agents-entities-relation),
3. between the EMS to the EIS (allows for controlling the execution of the environment),
4. between the EMS and the environment (allows for direct control over the environment's execution),
5. between the EIS and the environment (facilitates the already mentioned functionalities on the environment side), and
6. between the platform and the agents (e.g. for controlling the agents' execution, or creating/removing agents).

Perceiving: We allow for three different ways of perceiving: (1) active sensing through sensing actions, (2) passive sensing, and (3) perceptions sent by the environment automatically. Sensing actions are actions that are selected by the agent to perform next. In this sense, these actions represent a choice of the agent to inspect its environment by means of some sensory equipment. These actions should be part of the agent program. In contrast, passive sensing should not involve a choice of the agent, but is embedded in the control cycle of the agent. Note that our distinction does not relate to the usual differentiation in robotics regarding active and passive sensors: *"A sensor is often classified as being either passive sensor or active sensor. Passive sensors rely on the environment to provide the medium for observation, e.g., a camera requires a certain amount of ambient light to produce a useable picture. Active sensors put out energy in the environment to either change the energy or enhance it. A sonar sends out sound, receives the echo, and measures the time of flight."*[6] The term active sensor is not the same as active sensing. Active sensing is used to denote in a the system when effectors are used to dynamically position a sensor for a "better look". A camera with a flash is an active sensor; a camera on a pan/tilt head with algorithms to direct the camera to turn to get a better view is using active sensing.

Components: The platform and the agents are on the APL side. The environment is on the environment side. The EMS on the other hand has a special role, it can be on both sides. We do not wish to impose a restriction by requiring that the EMS is to be a component of the APL platform. In the case of the Multi-Agent Contest[2] for example, it is not allowed to control the execution of the environment through the connected APL platforms.

The platform may be equipped with further functionality like graphical user-interaction and integrated development of MAS, but we do not require that functionality.

Note that we assume the components – except for the EIS – to be implicit. Each object that is associated to the EIS via the agents-to-EIS connection qualifies as an agent, each object that uses the platform-to-EIS connection qualifies as the platform, and so on.

The connection between the EIS and the environment is arbitrary. It could be facilitated for example by Java programming constructs (methods, buffers,...) in the case that the connection to a Java-environment is to be established, Java-RMI if a distributed application is desired, JNI if the connection to a C/C++ application is desired, or TCP/IP (compare with Multi-Agent Contest) for a general, distributed solution.

Our aim is to allow specific interfaces to specific environments to be distributed. The EIS should allow for: (1) wrapping already existing environments (e.g. 2APL's blocksworld), (2) creating new environments by connecting already existing applications (e.g. Unreal Tournament), and (3) creating new

environments from scratch.

2.3 Principles

In order to define an environment interface standard, we have identified the following principles that we think should be adhered to when designing the interface standard:

1. **Portability:** We aim to facilitate the easy exchange of environments between platforms by (1) downloading the specific interface to an environment, (2) quickly adapting the MAS, (3) executing. We believe that using jar-files – following the examples of 2APL/ GOAL/JASON – facilitates the desired portability. Therefore we need a solid policy for locating the environment entry-points in the jar-files. We do not want to make the use of jar-files obligatory, however.

   When it comes to MAS-configurations, we are open for any suggestions. An environment is something arbitrary that agents connect to through the environment interface. If it is intended to instantiate several environments, this can be done using one environment interface each. Naming environments and resolving naming issues should not be our concern. These can be tackled by the APL platform programmers.

2. **Environment Interface Generality:**ă The interface should be generic and impose only minimal restrictions on the platform or environment. That is (compare with Fig. 1):

   - The interface should not impose scheduling restrictions when it comes to the execution of actions, actions can be either scheduled by the platform/agents or by the environment itself. The interface standard is supposed to plainly provide the functionality to connect to the environment. We expect that there will be cases in which the agents schedule actions (environments that do only change their states if agents act), and in which the environment will schedule the actions (like the Multi-Agent Contest in which the environment can evolve on its own and schedules the agent's actions).
   - It does not impose any assumptions on agent communication or on organization structure. Communication can be on both sides of the interface (i.e. facilitated by the agent/platform components or by the environment model in the meta-model).
   - It does not impose any assumptions about what is controlled in an environment, except for the fact that *controllable entities* are able to perform actions (and it is possible they do so by being "instructed" by an agent).

- It also does not impose any assumptions about how an agent platform controls entities in an environment.
- The interface does not require the components of the meta-model to be implemented explicitly.
- The interface should not limit the use of various technical options: the environment interface can be used for different types of connections (TCP/IP, RMI, wrapping Java code, JNI).
- The interface should not prohibit the use of several environments in a MAS.

3. **Separation of concerns:** We assume the agents to be separated from the environment(s) (see Fig. 2). We distinguish between APLs and agent programs on the one hand (agents are action-generators, and percept-processors) and an environment model and controllable entities on the other hand (entities can be instructed to perform actions, and the environment provides these entities with percepts which can be provided to agents through our interface).

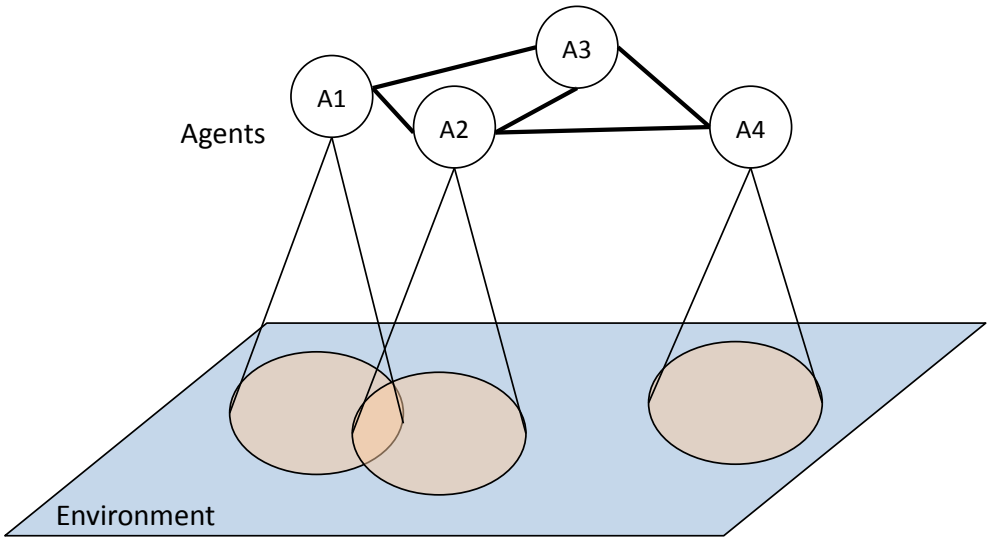

Figure 2: Environment-MAS Model.

>From the implementation point-of-view we disallow agent-objects being stored on the environment side and entity-objects being stored on the APL side. Rather we suggest that the environment interface stores

identifiers to both agents and entities and the relation (who-controls-whom) as a mapping. We repeat here that we do *not* assume a one-to-one relation between agents and controllable entities.

4. **Unified connections:** The environment interface standard should provide unified means for the connections between the agents, the platform and the environment management system on one side and the environment on the other. It should not restrict any existing approaches. The interface should facilitate acting, active and passive sensing, and events sent by the environment, by providing a set of *agent-methods*. The interface should facilitate creating, removing entities and assigning entities to agents, by providing a set of *platform-methods*. Finally the interface should facilitate controlling the execution of the environment(s), by providing a set of *environment-management-system-methods*.

5. **Standards for actions/percepts/events/etc.:** The environment interface standard has to provide a convention for actions, percepts, events, and other concepts of that kind, that does not restrict any existing approach. We intend to propose a standard based on special Java-objects.

6. **Support for heterogeneity:** The interface standard needs to facilitate heterogeneity. Currently, we think the easiest way of establishing heterogeneity that conforms with all other principles would be this: (1) set up and run a central application that contains the environment, and (2) provide a jar-file based on EIS that connects the platforms to the environment (TCP/IP, RMI, wrapping Java code, JNI).

   As an example, we would like to mention the multi-agent contest again. Here, heterogeneity would be established by (1) providing an environment interface that connects to the MASSim-server, and (2) providing a new action that allows for inter-agent communication.

3 The Proposed Standard: EIS

In this section, we present our proposal for an *environment-interface standard*. After defining a language for data exchange between interfaces and other components, we propose an abstract class for defining environment-interfaces.

This can be viewed as a semi-complete documentation to the software-packages. For a complete documentation we refer to the accompanying javadoc.

3.1 Interface Immediate Language

We design an *interface immediate language*, that is, a language for exchanging data between interfaces and other components via Java-objects. This corre-

sponds to principle 5 (see Section 2) for data-exchange between the environment-interface and different components. However, a convention is necessary that does not restrict existing and future approaches. We have decided to go for java-objects. They can be handled and integrated into existing approaches easily.

The language consists of 1. *data containers* (e.g. actions and percepts), and 2. *parameters* to those containers.

3.1.1 Parameters

Parameters are: identifiers, numbers, functions over parameters, and lists of parameters.

These are the Java-classes representing parameters:

- `eis.iilang.Identifier` represents an identifier,
- `eis.iilang.Numeral` represents a number,
- `eis.iilang.Function` represents a function over parameters, and
- `eis.iilang.ParameterList` represents a list of parameters.

3.1.2 Data Containers

Data containers are: actions that are performed by agents, results of such actions, percepts that are received by agents, and events that are sent to agents by the interface. Furthermore they are: environment commands that are for example issued to control the execution of the environment, and events that are sent to notify about changes of the state of execution.

Each of these data containers consist of (1) a name, and (2) a set of parameters. Here are the respective classes:

- `eis.iilang.Action` represents an action.
- `eis.iilang.ActionResult` represents the result of an action. This can be something very simple that just indicates the success of an action, or something more complex like the results of a sensing action.
- `eis.iilang.Percept` is a percept.
- `eis.iilang.EnvironmentCommand` is a command that is sent to the interface for example to control the execution of the environment.
- `eis.iilang.EnvironmentEvent` is an event that is sent by the interface to notify about changes in the environment.

When it comes to environment-commands and environment-events we impose a *convention*. An environment-command can either be *starting*, *pausing*, *initializing*, *resetting*, or *killing* the environment. Additionally we provide a way for implementing further commands. The same holds for environment-events. There are events that indicate that the environment has been *started*, *paused*, et cetera.

Note however that the use of these commands and events is not obligatory. There will be application-cases in which it makes sense to have the environment released via a respective environment command. But in the agent-contest, such a thing does not make sense. The same holds for all the other commands as well. Also it is not obligatory that environment-interfaces send environment-events. And finally, we expect this convention to be extended in the future.

Appendix A contains several examples.

3.2 Environment Interface Standard

We now elaborate on the main package, that contains an abstract class that is supposed to be used to implement specific environment-interfaces and a listener that is to be used to allow the environment-interface to notify other components about certain events.

The first point that we would like to elaborate on is the correspondence between an environment-interface and components (platform, agents). We allow for a two-way connection via *interactions* that are performed by the components and *notifications* that are performed by the environment-interface (see Fig. 3).

Interactions are facilitated by function calls to the environment-interface, that can yield a return-value. For notifications we employ the *observer design pattern*[4] (known as *listeners* in Java). The observer pattern defines that a *subject* maintains a list of *observers*. The subject informs the observers of any state change by calling one of their methods. This way distributed event handling is facilitated. The observer pattern is usually employed when a state-change in one object requires changing another one. This is the reason why we made that choice. The subject in the observer pattern usually provides functionality for *attaching* and *detaching* observers, and for *notifying* all attached observers. The observer, on the other hand, defines an *updating* interface to receive update notification from the subject.

We allow for both interactions and notifications, because this approach is the least restrictive one. This clearly corresponds to the notions of *polling* (an agent performs an action to query the state of the environment) and *interrupts* (the environment sends percepts to the agents as in the agent-contest).

The second point that we would like to elaborate on is the agents-entities-relation. We make three assumptions: 1. there is a set of agents on the APL

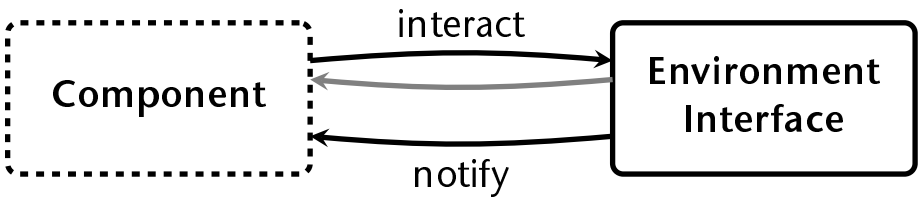

Figure 3: We provide two means for transfering data: interacting (polling) and notifying (interrupts).

platform side (we do not know anything about those), 2. there is a set of controllable entities on the environments side (again we do not know anything), and 3. agents can control entities through the environment-interface. A important design decision that we had to made was to store in the environment-interface only identifiers to the agents, identifiers to the entities and a mapping between those two sets. The reason for that decision is, as we have mentioned before, that we do not assume anything about the APL-platform-side and the environment-side.

Fig. 4 shows the agents-entities-relation. The agents live on the APL-platform-side, they are known by the environment-interface by their identifiers. The entities live on the environment-side, they are also known by their identifiers. The agents-entities-relation is stored as a mapping between both sets of identifiers.

Finally, these are the main classes of the environment-interface standard. We will elaborate on their functions later:

- `eis.EnvironmentInterfaceStandard` represents an interface. This is an abstract Java-class, that should be inherited from in order to create a specific interface. It represents the subject of the observer pattern. It contains all the functionality that allows for connecting platforms/agents/etc. to environments.
- `eis.EnvironmentListener` establishes a connection from the interface to other components. The interface is used for call-backs. Classes that implement that interface are the observers of the design pattern. Informs about changes in the environment.
- `eis.AgentListener` establishes a connection from the interface to agents. It sends percepts to agents.

Before we explain the services that are provided by the EIS, we assume a functional point of view. The environment-interface standard provides functions for

1. attaching, detaching, and notifying listeners;

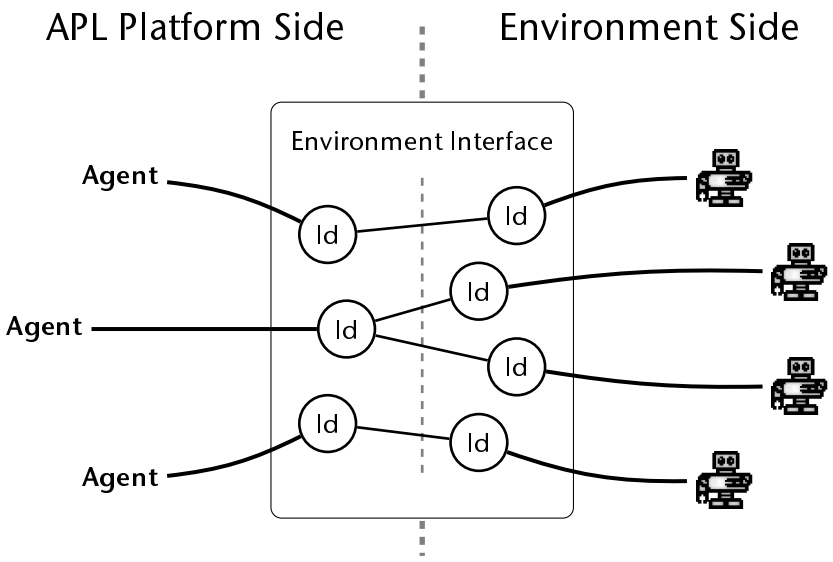

Figure 4: The agent-entities-relation.

2. registering and unregistering agents;
3. adding and removing entities;
4. managing the agents-entities-relation;
5. performing actions and retrieving percepts;
6. managing the environment;
7. loading environment-interfaces from jar-files;
8. miscellanea.

We will now consider these functions.

3.2.1 Attaching, Detaching, and Notifying Listeners

There are two directions for sending events to components. One is via environment-listeners, which inform observers about changes in the environment or the environment interface. The second is via agent-listeners, which send percepts to agents. In order to facilitate sending events, that is agent-events and environment-events, `eis.EnvironmentInterfaceStandard` provides functions that allow for attaching and detaching listeners, and for notifying components connected via listeners.

Classes that are supposed to function as the observer in the observer-pattern have to implement the interface `eis.EnvironmentListener` and/or the interface `eis.AgentListener` and their methods.

The `AgentListener`-interface has these methods:

- `void handleEnvironmentEvent(EnvironmentEvent event)` handles an environment-event.
- `void handleFreeEntity(String entity)` handles the incident that the environment-interface notifies about a free entity.
- `void handleDeletedEntity(String entity)` handles the incident that the environment-interface notifies about a deleted entity.
- `void handleNewEntity(String entity)` handles the incident that the environment-interface notifies about a new entity.

The `AgentListener`-interface has this method:

- `void handlePercept(String agent, Percept percept)` handles a percept. Is supposed to hand the event over to the specific agent.

These are the respective methods of `eis.EnvironmentInterfaceStandard` when it comes to environment-listeners:

- `public final attachEnvironmentListener( EnvironmentListener listener)` attaches an environment-listener.
- `public final detachEnvironmentListener( EnvironmentListener listener)` detaches a environment-listener.
- `protected final void notifyFreeEntity(String entity)` Notifies about a free entity. Invokes `handleFreedEntity` of all environment-listeners.
- `protected final void notifyNewEntity(String entity)` Notifies about a new entity. Invokes `handleNewEntity` of all environment-listeners.
- `protected final void notifyDeletedEntity(String entity)` Notifies about a deleted entity. Invokes `handleDeletedEntity` of all environment-listeners.
- `protected final void notifyEnvironmentEvent( EnvironmentEvent event)` Notifies about an environment-event. Invokes `handleEnvironmentEvent` of all environment-listeners.

Note that some functions are protected. They are not supposed to be called from outside the environment-interface.

These are the respective methods of `eis.EnvironmentInterfaceStandard` when it comes to agent-listeners:

- `public final void attachAgentListener(String agent, AgentListener listener)` attaches an agent-listener for a specific agent.
- `public final void detachAgentListener(String agent, AgentListener listener)` detaches an agent-listener.
- `protected final void notifyAgents(Percept percept, String...agents)` notifies a list of specific agents about a certain event. Invokes `handlePercept` of all listeners. If the list of agents is empty, all agents will be used.
- `protected final void notifyAgentsViaEntity(Percept percept, String...pEntities)` Looks up those agents that are associated to the given list of entities and notifies them. Invokes the method `handlePercept` of all listeners. If the list of entities is empty, all entities will be used.

Note that the notifying-methods are final.

3.2.2 Registering and Unregistering Agents

These methods provided by `eis.EnvironmentInterfaceStandard`: facilitate registering and unregistering agents.

- `public final void registerAgent(String agent) throws AgentException` registers an agent. An exception is thrown if an agent of the same name has already been registered.
- `public final void unregisterAgent(String agent) throws AgentException` unregisters an agent. Throws an exception if the agent has not been registered. Also updates the agents-entities-relation.
- `public final LinkedList<String> getAgents()` returns the list of registered agents.

3.2.3 Adding and Removing entities

Adding and removing entities is facilitated by these methods of the Java-interface `eis.EnvironmentInterfaceStandard`:

- `protected final void addEntity(String entity) throws EntityException` adds an entity. An exception is thrown if an entity of the same name has already been added.
- `protected final void deleteEntity(String entity) throws EntityException` deletes an entity. Throws an exception if the entity does not exist. Also updates the agents-entities-relation.

Note that these methods are not visible. Only the environment is supposed to add and remove entities.

3.2.4 Managing the Agents-Entities-Relation

Managing the agent-entities-relation is facilitated by these methods of the Java-interface `eis.EnvironmentInterfaceStandard`:

- `public void associateEntity(String agent, String entity) throws RelationException` associates an agent with an entity. Throws an exception if the agent has not been registered and/or the entity has not been added. Can be overridden to restrict the agents-entities-relation.
- `public final void freeEntity(String entity) throws RelationException` frees an entity. An exception is thrown if the entity does not exist.
- `public final void freeAgent(String agent) throws RelationException` frees an agent. An exception is thrown if the agent as not been registered.
- `protected final HashSet<String> getAssociatedEntities( String agent) throws AgentException` returns the set of entities associated with a specific agent. Throws an exception if the agent has not been registered.
- `protected final HashSet<String> getAssociatedAgents( String entity) throws EntityException` returns the set of agents associated with a specific entity. Throws an exception if the entity does not exist.
- `public final LinkedList<String> getFreeEntities()` returns a list of free entities.

3.2.5 Performing Actions and Retrieving Percepts

These methods of the Java-class`eis.EnvironmentInterfaceStandard` facilitate performing actions and retrieving percepts:

- `public final LinkedList<ActionResult> performAction( String agent, Action action, String...entities) throws PerceiveOrActFailureException,NoEnvironmentException` performs an action of an agent through the entities in the given list. If the list is empty the agent will act through all its entities. Throws an exception if the agent has not been registered, or the list of entities is inconsistent with the list of associated entities of the agent, or if the interface is not connected to an environment. Looks up a method that corresponds to the action's name and its parameters via Java-reflection.

- `public final LinkedList<Percept> getAllPercepts(String agent, String...entities) throws PerceiveOrActFailureException,NoEnvironmentException` lets an agent sense through its associated entities. Senses either through all associated entities if the list is empty or through the specific entities. Throws an exception if the agent has not been registered, or the list of entities is inconsistent with the list of associated entities of the agent, or if the interface is not connected to an environment. Calls the next function for each associated entity.

- `public abstract LinkedList<Percept> getAllPerceptsFromEntity(String entity)` has to be overridden to provide sensing (sensing is interface-specific; compare to performing actions). This method is called by the previous one.

The method `performAction` processes the given action and invokes a respective method using Java-reflection. For example, if an action with the name `moveto`, with two parameters is to be executed, the interface will attempt to look up the method `actionmoveto(String entity, Parameter p1, Parameter p2)`, where `entity` is the identifier of an associated controllable entity. The naming of action-methods is a convention the string `action` is to be followed by the actual name of the action for the sake of readability.

Note that we do not support *identity management*. That is, we do not provide means to ensure that the caller of the method `performAction` and/or `getAllPercepts` is either the respective agent or an instance acting on its behalf. In our point of view, the developer of the agent-platform is responsible to ensure the right identity management. We have to stress that we assume a very non-restrictive position with the EIS. Especially when it comes to agents. We do not want to take anything about agents for granted. Nothing about their structure or how they are implemented. The only point that me made obligatory is that each agent that wants to act/perceive has to be represented in the interface by its id. In summary we provide means for storing agents and entities represented by their identifiers and their relation in the environment-interface.

Furthermore note that we use arrays as parameters (here and in other methods). Note however if you would like to use collections instead of arrays you can easily transform to collections using the method `asList` of the Java-class `java.util.Arrays`.

3.2.6 Managing the Environment

Managing the environment is facilitated by this method of the Java-interface `eis.EnvironmentInterfaceStandard`:

- `public abstract void manageEnvironment(EnvironmentCommand command, String... args) throws ManagementException` processes an environment-command. Throws an exception if this fails. Must be overridden.

We refer to Subsection 3.1.2 for the convention for environment-events and environment-commands.

3.2.7 Loading environment-interfaces from jar-files

This method of `eis.EnvironmentInterfaceStandard` facilitates loading environment-interfaces from jar-files:

- `public static EnvironmentInterfaceStandard fromJarFile(File jarFile) throws IOException` loads a specific environment interface from a given jar-file.

The name of the file without the extension defines the class that is loaded and instantiated. For example, given the file `carriageexample.jar` the class `carriageexample.EnvironmentInterface` will be loaded.

3.2.8 Miscellanea

The connection between the environment-interface and the environment can be terminated using this method of `eis.EnvironmentInterfaceStandard`:

- `public abstract void release()` will terminate the connection and release all resources. After that the environment-interface cannot be used anymore. A programmer that has to implement such a connection has to make sure that these requirements are fulfilled.

Note that there is a difference between calling the `release()`-method and sending a kill-environment-command. The first disconnects the environment-interface from the environment, not necessarily releasing the environment. The second releases the environment.

The state of the connection between the environment-interface and the environment can be queried using this method of the Java-interface `eis.EnvironmentInterfaceStandard`:

- `public abstract boolean isConnected()` returns `true` if the connection is valid, `false` otherwise. Again the programmer has to ensure this requirement.

4 How to Connect to the EIS

Given an environment, what needs to be done to connect this environment to our interface?

We suggest the following steps (they are not obligatory) that can be used to connect environments to an interface:

1. Incorporate a mechanism for loading environment-interfaces.
2. Implement transformations for mapping the data-containers (actions, percepts, et cetera) to APL-platform-specific data-structures and vice versa.
3. Prepare the platform for callbacks (agent-events, environment-events, et cetera).
4. Implement a policy for registering agents to the environment-interface.
5. Implement a policy for associating your agents with the controllable entities provided by the environment.
6. Incorporate acting and perceiving.

Some of these steps can be skipped for some environments, but not for others. Some of these steps can be implemented in a 'minimal' sense (like always automatically connecting some agent to an entity when it appears).

We will now elaborate on three examples: (1) the platform example shows how an APL platform could be connected to an environment-interface, (2) the carriage example shows how an environment could be connected to an environment-interface by wrapping it, (3) the MASSim connector shows how the MASSim server (used for the annual agent contest) can be easily connected to an environment-interface.

4.1 The Platform Example

The platform example shows how a simple APL platform could be connected to an environment-interface. Please compare with package `eis.examples.platform`.

First, the platform class is defined:

```
public class Platform implements EnvironmentInterfaceListener { ... }
```

The platform-class implements the listener-interface in order to make it receive events.

4.1.1 Mechanism for Loading Environment Interfaces

Given the filename of a jar-file that contains an environment-interface, it can be loaded like this:

```
EnvironmentInterfaceStandard ei = null;
try {
    ei = EnvironmentInterfaceStandard.fromJarFile(new File(jarFileName));
    System.out.println("Environment interface loaded.");
} catch (IOException e) {
    System.out.println("Jar-file could not be loaded.");
    System.exit(0);
}
```

The static method `fromJarFile` is invoked to load an environment from a jar-file. The exception-handling is very important at this position, since loading an interface could fail.

4.1.2 Preparing for Callbacks

Now that the environment-interface has been loaded, the platform has to be connected in order to function as an observer:

```
    ei.attachListener(this);
```

4.1.3 Agent Registering

The string `agent` represents the id of an agent. The agent can be registered as follows:

```
try {
    ei.registerAgent(agent);
    System.out.println("Added agent " + agent);
} catch (AgentException e) {
    System.out.println("Agent " + agent + " could not be added.");
}
```

Again, some exception handling is necessary, because registering could fail.

4.1.4 Agent Associating

Now we will associate agents with entities. `agent` is a string representing the id of an agent, and `entity` is a string representing the id of an entity.

Associating could look like this:

```
try {
    ei.associateEntity(agent, entity);
    System.out.println("Associated " + agent + " with " + entity);
} catch (RelationException e) {
    System.out.println("Failed to associate " + agent + " with " + entity);
}
```

4.2 The Carriage Example

The rules of the game are as follows:

- The environment contains a carriage on a circular track (see Fig. 5). There are three distinct positions for the carriage on the track.
- The environment also contains two robots, which are the controllable entities.
- An agent that controls a robot can make it push the carriage or wait. If both robots push or if none of them pushes, nothing will happen. If only one robot pushes the carriage will be moved to the next position.

The package `carriageexample` consists of these classes:

- `carriageexample.Environment` contains the environment.
- `carriageexample.EnvironmentInterface` contains the interface to the environment
- `carriageexample.EnvironmentWindow` contains the environment window.
- `carriageexample.Main` contains a test scenario with two agents.

The environment provides these methods:

- `public int getCarriagePos()` returns the position of the carriage.
- `public long getStepNumber()` returns the current step of the environment.
- `public void robotPush1()` lets the first robot push the carriage in the current step.
- `public void robotPush2()` lets the second robot push the carriage in the current step.
- `public void robotWait1()` lets the first robot wait in the current step.

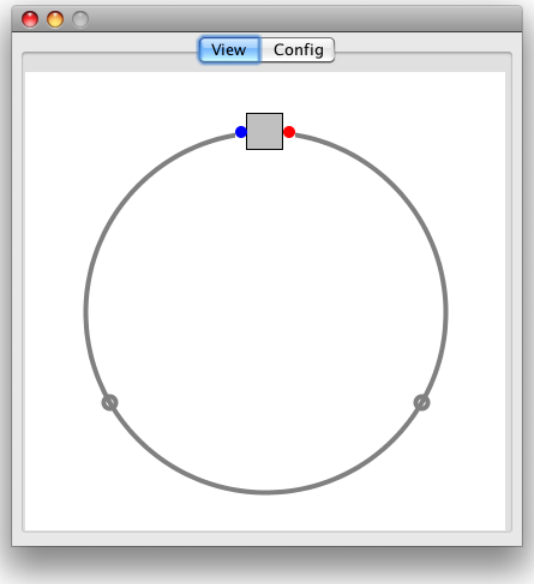

Figure 5: Screenshot of the carriage example.

- `public void robotWait2()` lets the second robot wait in the current step.
- `public int getRobotPercepts1()` returns the percepts of the first robot, that is its perceive carriage position.
- `public int getRobotPercepts2()` returns the percepts of the second robot, that is its perceive carriage position.

In its constructor the environment-interface adds the two entities:

```
public EnvironmentInterface() {

    try {

        this.addEntity("robot1");
        this.addEntity("robot2");

    } catch (EntityException e) {
        e.printStackTrace();
    }

[...]
```

The interface has a thread that notifies the attached listeners once per second about the entities that are free and the current step-number:

```
// notify about free entities
for( String free : this.getFreeEntities() )
    this.notifyFreeEntity(free);

// tell current step
long step = env.getStepNumber();
Percept p = new Percept("step", new Numeral(step) );
for( String entity : this.getEntities() )
    try {
        this.notifyAgentsViaEntity(p, entity);
    } catch (EnvironmentInterfaceException e1) {
        e1.printStackTrace();
    }
```

To facilitate perception the method `getAllPerceptsFromEntity` had to be overridden:

```
    @Override
    public LinkedList<Percept> getAllPerceptsFromEntity(String entity) {

        LinkedList<Percept> ret = new LinkedList<Percept>();

        if( entity.equals("robot1"))
            ret.add(
                    new Percept(
                            "carriagePos",
                            new Numeral(env.getRobotPercepts1())
                            )
                    );
        else if( entity.equals("robot2"))
            ret.add(
                    new Percept(
                            "carriagePos",
                            new Numeral(env.getRobotPercepts2())
                            )
                    );

        return ret;

    }
```

Since the are only two actions – `push` and `wait` – two methods had to be implemented, in order to have them invoked via Java-reflection:

```
    public ActionResult actionpush(String entity) {

        // push
        if( entity.equals("robot1") )
            env.robotPush1();
        if( entity.equals("robot2") )
            env.robotPush2();

        return new ActionResult("success");

    }

    public ActionResult actionwait(String entity) {

        // push
        if( entity.equals("robot1") )
            env.robotWait1();
        if( entity.equals("robot2") )
            env.robotWait2();
```

```
        return new ActionResult("success");

    }
```

Finally the connection is terminated using like this:

```
    @Override
    public void release() {

        env.release();

        env = null;

    }
```

The method releases the environment itself and then the connection.

4.3 The MASSim-Connector

The MASSim-connector is an environment-interface to the MASSim-server. The MASSim-server is the application that manages and executes the annual Multi-Agent Contest[1]. For us the most important aspect is: The MASSim-server contains an environment with which agents can interact.

The environment is discrete in space and time. The world is a grid, the simulation is executed in a step-wise way. There are two teams of cowboys. Each cowboy is a controllable entity that can move in the environment. The environment is only partially accessible, an entity can only perceive the cells of the grid that are in the visibility range. There are also non-controllable entities in the environment: cows move according to a special movement-algorith. The goal is to use the cowboys to push cows in to the corrals.

>From a technical point-of-view, the server allows and manages TCP/IP connections: One connection for each controllable entity. Agents are executed remotely and interact with the server by exchanging XML-messages. The simulation in each step sends a message containing the current perception (what is visible) of each cowboy and waits for a reaction from the cowboys. After a specific time-out the environment evolves by one step.

The implementation consists of these classes:

- `acconnector2009.Connection` contains a connection between the interface and the MASSim-server that contains the environment. Each entity can be associated with one connection.
- `acconnector2009.ConnectionListener` is used to handle incoming messages, that is messasges sent by the MASSim-server.
- `acconnector2009.EnvironmentInterface` contains the environment-interface.

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

- `acconnector2009.Main` contains a sample application that shows a single agent connecting to and interacting with the MASSim-server.

Actions are:

- `public ActionResult actionconnect(String entity, Identifier server, Numeral port, Identifier user, Identifier password) throws ActException` implements the authentification-step of the communication protocol. A server location (IP-adress or URL) and a port have to be provided. If the connection cannot be established an exception is thrown.
- `public ActionResult actionmove(String entity, Identifier direction) throws ActException, NoEnvironmentException` sends an action message encapsulating a movement-action to the MASSim-server. If the action cannot be performed and/or there is no valid connection to the MASSim-server an exception is thrown.
- `public ActionResult actionskip(String entity) throws ActException, NoEnvironmentException` sends an action message encapsulating a skip-action to the MASSim-server. If the action cannot be performed and/or there is no valid connection to the MASSim-server an exception is thrown.

There are two ways of perceiving: incoming messages that contain percepts are transformed into the IIL and then 1. sent to the respective agents via the listener-interface and 2. stored for the use of `getAllPercepts`.

For details we refer to the source-code and its javadoc.

5 Conclusion

In this paper we have developed, to the best of our knowledge for the first time, an environment interface standard (EIS). This standard facilitates connecting agents programmed in various agent programming languages (APL) to arbitrary environments. The standard is based on a set six principles. We have shown how several of the currently employed platforms in the agent community (2APL, GOAL, JADEX, JASON) can be easily connected to our EIS. In addition, we have indicated a general methodology how to connect an arbitrary environment to our EIS.

We are currently testing our EIS with the agent contest, an annual contest on comparing and evaluating multi-agent systems on a grid where entities have to cooperatively solve a particular goal.

A standard like the one proposed in this paper can have a huge influence in the agent community. Not only are working environments made available for

all platforms that comply to the standard, but also future environments (e.g. for the next agent contest) can be dealt with without changing the underlying functionality. Even more importantly, such a standard allows for a truly heterogenous MAS, whose agents can belong to completely different APL platforms (that comply to the standard).

References

[1] 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.

[2] 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.

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

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[5] 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.

[6] Robin R. Murphy. *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA, 2000.

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

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

A Interface Immediate Language Examples

We will now show some examples. Internally data containers are stored as Java-objects. We assume that they can be printed as XML-strings and strings in predicate-form. Note, however that the Java-objects are intended to be the main means for data exchange.

Example 1 *This action is moving to a specific position:*

```
new Action(
    "moveTo",
    new Numeral(2),
    new Numeral(3)
)
```

XML-printout:

```
<?xml version="1.0" encoding="UTF-8"?>
<action name="moveTo">
  <actionParameter>
    <number value="2.0"/>
  </actionParameter>
  <actionParameter>
    <number value="3.0"/>
  </actionParameter>
</action>
```

Predicate-form-printout:

```
action(moveTo,2.0,3.0)
```

Example 2 *This action is following a specific path (a list of coordinates) at a given speed.*

```
new Action(
    "followPath",
    new ParameterList(
        new Function("pos", new Numeral(1), new Numeral(1)),
        new Function("pos", new Numeral(2), new Numeral(1)),
        new Function("pos", new Numeral(2), new Numeral(2)),
        new Function("pos", new Numeral(3), new Numeral(2)),
        new Function("pos", new Numeral(4), new Numeral(2)),
        new Function("pos", new Numeral(4), new Numeral(3))
        ),
    new Function("speed", new Numeral(10.0))
)
```

XML-printout:

```
<?xml version="1.0" encoding="UTF-8"?>
<action name="followPath">
  <actionParameter>
    <parameterList>
      <function name="pos">
        <number value="1.0"/>
        <number value="1.0"/>
```

```
      </function>
      <function name="pos">
        <number value="2.0"/>
        <number value="1.0"/>
      </function>
      <function name="pos">
        <number value="2.0"/>
        <number value="2.0"/>
      </function>
      <function name="pos">
        <number value="3.0"/>
        <number value="2.0"/>
      </function>
      <function name="pos">
        <number value="4.0"/>
        <number value="2.0"/>
      </function>
      <function name="pos">
        <number value="4.0"/>
        <number value="3.0"/>
      </function>
    </parameterList>
  </actionParameter>
  <actionParameter>
    <function name="speed">
      <number value="10.0"/>
    </function>
  </actionParameter>
</action>
```

Predicate-form-printout:

```
action(
  followPath,
  [pos(1.0,1.0),pos(2.0,1.0),pos(2.0,2.0),pos(3.0,2.0),pos(4.0,2.0),pos(4.0,3.0)],
  speed(10.0))
```

Example 3 *This percept represents a red ball that is made of rubber:*

```
new Percept(
    "sensors",
    new ParameterList(
        new Function("red", new Identifier("ball")),
        new Function("rubber", new Identifier("ball"))
    )
  );
```

XML-printout:

```
<percept name="sensors">
  <perceptParameter>
    <parameterList>
      <function name="red">
        <identifier value="ball"/>
      </function>
      <function name="rubber">
        <identifier value="ball"/>
      </function>
    </parameterList>
  </perceptParameter>
</percept>
```

Predicate-form-printout:

```
percept(sensors,[red(ball),rubber(ball)])
```

Example 4 *This environment command tells the environment-interface to pause the execution of the environment:*

```
new EnvironmentCommand(
    EnvironmentCommand.PAUSE
    );
```

XML-printout:

```
<?xml version="1.0" encoding="UTF-8"?>
<environmentCommand type="pause">
</environmentCommand>
```

Predicate-form-printout:

```
environmentcommand(pause)
```

Example 5 *This environment command tells the environment-interface to initialize the environment with a config-file:*

```
new EnvironmentCommand(
    EnvironmentCommand.INIT,
    new Identifier("/home/groucho/eisexamples/config.txt")
);
```

XML-printout:

```
<?xml version="1.0" encoding="UTF-8"?>
<environmentCommand type="init">
  <environmentCommandParameter>
    <identifier value="/home/groucho/eisexamples/config.txt"/>
  </environmentCommandParameter>
</environmentCommand>
```

Predicate-form-printout:

```
environmentcommand(init,/home/groucho/eisexamples/config.txt)
```

Example 6 *This environment command requests the current time of the environment:*

```
new EnvironmentCommand(
    "request",
    new Identifier("time")
);
```

XML-printout:

```
<?xml version="1.0" encoding="UTF-8"?>
<environmentCommand name="request" type="misc">
  <environmentCommandParameter>
    <identifier value="time"/>
  </environmentCommandParameter>
</environmentCommand>
```

Predicate-form-printout:

```
environmentcommand(misc,request,time)
```

Example 7 *This environment-event has the meaning that the execution of the environment has been paused:*

```
new EnvironmentEvent(
    EnvironmentEvent.PAUSED
);
```

XML-printout:

```
<?xml version="1.0" encoding="UTF-8"?>
<environmentEvent type="paused">
</environmentEvent>
```

Predicate-form-printout:

```
environmentevent(paused)
```

Example 8 *This environment-event transports the current time of the environment:*

```
new EnvironmentEvent(
    "environmentTime",
    new Numeral(System.currentTimeMillis())
);
```

XML-printout:

```
<?xml version="1.0" encoding="UTF-8"?>
<environmentEvent name="environmentTime" type="misc">
  <environmentEventParameter>
    <number value="1200020201"/>
  </environmentEventParameter>
</environmentEvent>
```

Predicate-form-printout:

```
environmentevent(misc,environmentTime,1200020201)
```