An Interface for Agent-Environment Interaction

Tristan M. Behrens[1], Koen V. Hindriks[2], Rafael H. Bordini[3], Lars Braubach[4], Mehdi Dastani[5], Jürgen Dix[1], and Jomi F. Hübner[6] Alexander Pokahr[4]

[1] Clausthal University of Technology, Germany {behrens,dix}@in.tu-clausthal.de
[2] Delft University of Technology, The Netherlands k.v.hindriks@tudelft.nl
[3] Federal University of Rio Grande do Sul, Brazil r.bordini@inf.ufrgs.br
[4] Hamburg University, Germany {braubach,pokahr}@informatik.uni-hamburg.de
[5] Utrecht University, Utrecht, The Netherlands mehdi@cs.uu.nl
[6] Federal University of Santa Catarina, Brazil jomi@das.ufsc.br

Categories and subject descriptors: I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*; I.6.3 [**Simulation and Modeling**]: Applications; I.6.7 [**Simulation Support Systems**]: Environments

General terms: Standardization

Keywords: Agent development techniques, tools and environments, and Case studies and implemented systems

Abstract. Agents act and perceive in shared environments where they are situated. Although there are many environments for agents – ranging from testbeds to commercial applications – such environments have not been widely used because of the difficulty of interfacing agents with those 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 have *designed and developed a generic environment interface standard*. Our design has been guided 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 might become an actual standard in the near future.

1 Introduction

Agents are situated in environments in which they perceive and act. From an engineering point of view, an issue that repeatedly has to be dealt with is how to connect agents to environments. Sometimes this issue is (partially) solved by the

physical sensors and actuators provided (e.g. in the case of a robot). But even if sensor and actuator specifications are available, the design and implementation of the interaction between the agents and the environment still require substantial effort. This is due in part to the fact that each environment is different but also because the platforms to build agents provide different support for agent-environment interaction.

By now, there exist many interesting environments which range from specialized testbeds for agent systems to industrial applications based on agent technology. In each of these applications, the interaction between agents and environments has to be addressed. This is particularly true in application areas for agent technology such as multi-agent based simulation and the use of agents in (serious) gaming [12,20,21]. In the former, agents need to be connected to computational models of real-world scenarios whereas in the latter agents are used to control virtual characters that are part of a game. The design of agent-environment interaction raises many interesting issues such as who is in control of particular features of the system and what would be the right level of abstraction of the interface that supports the interaction. Technically, there are also many challenges as witnessed by [11] who discuss an interface for connecting agents to the game Unreal Tournament 2004. This gaming environment has been identified as a potentially interesting testbed for multi-agent systems [6]. But without a suitable, generic interface that supports flexible agent-environment interaction such a testbed is unlikely to be widely used.

The availability of many interesting environments for applying agents does not mean that they are easily accessed by agents that are built using different platforms. To the contrary, in practice, it is often the case that agent developers rebuild very similar environments such as grid-like environments from scratch (one well-known toy example is the Wumpus environment [26] of which many implementations exist). Apart from the duplicate work of developing these environments, this also means that dedicated interfaces for agent-environment interaction are built: this makes it difficult to reuse existing environments. Instead, it would be much better to work with an *environment interface standard* which provides all the required functionality for connecting agents to an environment in a standardized way. If environments were developed using such a standard, they could be exchanged freely between agent platforms that support the standard and thus would make already existing environments widely available.

In this paper, we propose an *environment interface standard* that facilitates the sharing and easy exchange of environments for agents. Such a standard will facilitate the reuse of environments between agent platforms; it will support the easy distribution of environments such as the Multi-Agent Contest [15], Unreal Tournament,and many others. There are, however, many other benefits. An environment interface standard will provide a standardized and general approach for designing agent-environment interaction: this is important for the comparison of agent platforms as it would ensure that the same interface is used by each platform. Moreover, a generic interface will support the development of truly heterogeneous MAS, consisting of agents from several platforms. From a

more abstract point of view, the design of an interface standard will also increase insight and conceptual understanding of agent-environment interaction.

Our approach is to design an interface that is *as generic as possible*, and that facilitates *reuse as much as possible* from existing interfaces. Clearly, there is a trade-off between these two goals. Our 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 approaches. As agent-oriented programming platforms seem particularly suitable for developing agents, we have chosen to use four of the more well-known agent programming languages (APLs) as our starting point. The advantage of this choice is that each of these languages to some extent have already solved the problem of agent-environment interaction even though in ways more or less specific to the language. As a consequence, a second advantage is that it may be easier to adapt such platforms to the new interface standard, and we can evaluate the interface proposal by implementing it in these agent languages. The design outlined in this paper fits for current APLs and we are confident that our interface would also be suitable for other agent platforms. We have incorporated the same functionality as has been used to connect the selected APLs to environments in the past and improved upon those interfaces. Our current experience has shown that EIS eases the issue of connecting APLs to environments, when it comes both to time as well as structure, showing that standardization helped here.

The paper is organized as follows. The design of an environment interface requires a *meta model* of environments, agents, and agent platforms. In Section 2, the principles and requirements such a meta-model should satisfy are identified and the basic components of the model, their interrelations, and the functionalities provided are described. The meta model is used in Section 3 to define the proposed *environment interface standard*, the main contribution of this paper. Section 4 discusses related work and Section 5 evaluates the proposed standard.

2 Principles and Meta-Model

2.1 Principles

Two of the main motivations for introducing a generic environment interface are to facilitate the easy exchange of environments between agent platforms and to gain a more thorough understanding of the issues related to agent-environment interaction. The environment interface should allow for: (1) wrapping already existing environments, (2) creating new environments by connecting already existing applications, and (3) creating new environments from scratch. To this end, in this section we discuss and present requirements such an interface should satisfy. We do so by introducing various principles the interface should adhere to. We have analyzed the agent-environment support provided by four well-known agent programming languages: 2APL [14], GOAL [17], Jadex [9], and Jason [10]. Based on the principles, we then present a meta-model for an agent-environment

interface that is able to provide at least the support for agent-environment interaction already provided by existing agent platforms (Section 2.2).

In order to guide the design of the interface, and to ensure that the interface meets our objectives, we have identified a number of principles we believe a generic environment interface should meet. First, as we aim for a generic interface, the interface should impose as few restrictions on agent platforms and environments as possible. More specifically, we believe that an environment interface should *not* impose: (1) scheduling restrictions on the execution of actions (actions can be scheduled by the agent platform and/or by the environment), (2) restrictions on agent communication or organization structure (communication facilities may be provided by the agent platform as well as by the environment), (3) restrictions on what is controlled in an environment or how this control is implemented except for the fact that control is established by an agent performing actions , and (4) restrictions on how various components of the model should be implemented; for example, the interface should allow for different types of agent-environment connection (e.g. TCP/IP, RMI, JNI).

Second, as the interface is aimed at facilitating comparison of agent platforms, a strict separation of concerns is advocated: the interface should not make any assumptions about either the agent platform or the environments such platforms are connected to, except for the type of connection that is established and associated functionalities. In our meta-model, this will be represented by a clear distinction between agents and what we call controllable entities (i.e. "agents' bodies situated in the environment"). Technically, this means the environment interface must abstract from all implementation details concerning both agents as well as environment objects. Instead, the environment interface may only store identifiers to agents and entities and should administrate which agents are associated with which entities (i.e. "who controls which body"). This level of abstraction is required to ensure that no additional implementation effort is required once the agent platform has been adapted.

Finally, as a more technical requirement, the interface should support portability, i.e., the easy exchange of environments from one agent platform to another. As most agent platforms are implemented using JAVA it is at least possible to provide this kind of functionality for such platforms if certain fixed policies are adopted for initialising an environment.

2.2 Meta-Model

We have identified five components that are part of the meta-model on which we base the design of the proposed environment interface. This meta-model is illustrated in Fig. 1, and includes an *environment model*, an *environment interface* that consists of an *environment management system* and an *environment interface* component, an *agent platform* and *agents.*

Our *environment model* assumes the presence of a specific kind of entity. [7] defines an entity as "any object or component that requires explicit representation in the model." In the context of agent-environment interaction, the entities that we are interested in may be controlled by an agent. This means that the

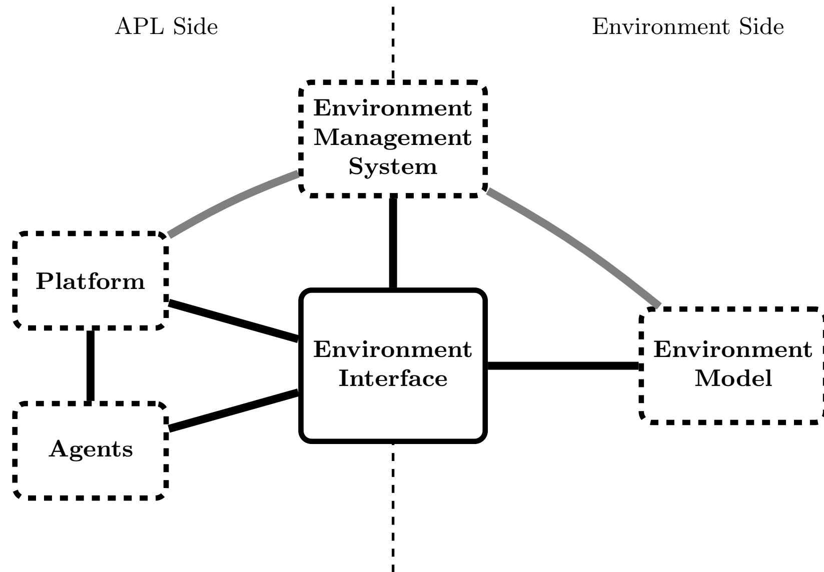

Fig. 1. The components of our environment meta-model. The platform and the agents are on the APL side. The environment management system is in between. The specific environment model is on the environment side. A specific environment model combined with the environment interface, yields a specific environment interface.

behavior generated by the entity can be controlled by an agent if the agent is properly connected to the entity. It is the task of the interface to establish such a connection. Entities in an environment that can be so controlled are called *controllable entities*.

Controllable entities facilitate the connection between the agents running on an agent platform and an environment by providing identifiers, *effecting* capabilities, and *sensory* capabilities to agents. An agent's identifier allows the environment to send percepts or events to agents by means of the interface. Moreover, the effecting and sensory capabilities specified by controllable entities allow the environment to contextualize an agent's action repertoire, the actions' effects, and which part of the environment can be sensed, thus establishing the *situatedness* of the agents.

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 agents are able to perform actions in the environment, sense the state of the environment and process such sensorial input, and receive and process events that are generated by the environment. We use the following very generic definition taken from [26] 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 program, although we are primarily motivated by existing agent-oriented programming languages.

An *agent platform* is the infrastructure that facilitates the instantiation and execution of individual agents. It is also assumed to facilitate connecting agents with environments and associating agents to controllable entities by means of the environment interface functionality. Other than that, nothing else is assumed about an agent platform.

The *environment interface* consists of two components: the *agent-environment interaction* component and the *environment management* component. The agent-environment interaction component manages the mapping and interaction between individual agents and the agent platform on one hand, and the environment and controllable entities on the other hand. The interaction between an agent platform and the agent-environment interaction component allows agents to act in an environment, sense its state, and receive events from it. We allow two ways of sensing: (1) active sensing through specific sense actions, and (2) passive sensing through a generic sense action embedded in the control cycle of the agents. Using the agent-environment interaction component, the platform can process different types of actions by calling different methods of this component and possibly wait for the return values which are subsequently passed on to the platform. The values returned can be either success/failure notifications or sense information if actions were (passive or active) sense actions. The environment interface can also interact with a platform by sending an event to a specified agent. The platform is then responsible to pass the event on to the specified agent (e.g., by adding the event to the agent's event base).

3 A Generic Environment Interface

In this section, we explain our ideas for a generic environment interface. First, we define an *interface intermediate language* that facilitates data-exchange (percepts, actions, events) between different components. Second, we assume a functional point-of-view of the interface architecture. The interface provides functions for:

1. attaching, detaching, and notifying observers (software design pattern);
2. registering and unregistering agents;
3. adding and removing entities;
4. managing the agents-entities-relation;
5. performing actions and retrieving percepts; and
6. managing the environment.

3.1 Motivating Example: Multi-Agent Contest

The Multi-Agent Programming Contest (MAPC) 2010 tournament consists of a series of simulations. In each simulation (see Fig. 2) two teams of agents compete in a grid-like world. There are virtual cowboys that can be controlled by agents.

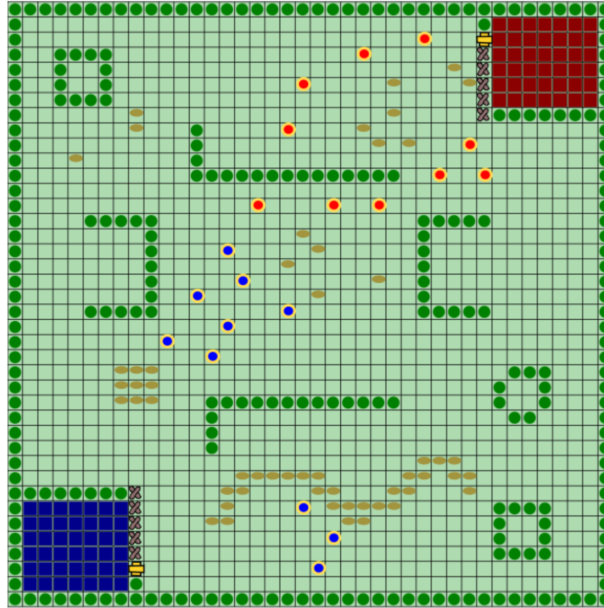

Fig. 2. A screen-shot of a simulation from MAPC 2010. Cowboys (red and blue circles) should scare cows (brown ellipses) into the corrals (red and blue rectangles). In this environment, acting is moving the cowboys, and perceiving is getting information about which objects are contained in each cowboy's square of visibility.

Agents have access to incomplete information, because the cowboys have a fixed sensor-range. Acting means moving a cowboy to a neighboring cell on the grid. There are no further actions. The environment contains obstacles: some cells can be blocked and thus are unreachable. The grid is also populated by virtual cows, that behave according to a simple flocking-algorithm. To win a simulation, an agent team has to herd more cows, and take them to their own corral, than the opponent team. The simulation proceeds through discrete time steps. In each step, agents can perceive, have a fixed amount of time to deliberate, and are then allowed to act. After a number of steps the simulation is finished. The tournament is run by the MASSim-server, which schedules and runs simulations. Agents are supposed to connect to the server as clients. Communication between clients and server is facilitated by exchanging XML-messages via the TCP/IP protocol.

We have given a very informal but adequate description of the environment-model. The environment is discrete in space (grid-world) and time (step-wise evolution). Platforms can interface with the MASSim-server by adhering to the defined communication-protocol. This has to be done for every platform, in a way specific to that platform. This is the case because, as we have observed, every platform has a specific way of connecting to environments. Every platform would have to use that connection-mechanism, parse XML-messages to evaluate

the percepts, and generate XML-messages for performing actions. Now, assuming that a platform would be EIS-compatible, you only have to go through the trouble of connecting to the MASSim-server once and create a specific environment-interface.

3.2 Interface Intermediate Language

An important design decision has been to define, as part of the environment interface, a convention for representing actions and percepts. This convention is called the *interface intermediate language* (IIL), and supports the exchange of percepts and actions from/to environments. A conventional representation for actions and percepts is required to be able to meet the second principle aimed at facilitating comparison of platforms and the fourth principle that aims at easy exchange of environments and portability. To meet these principles, the interface should be agnostic to any implementation details of either agent platform or environment; this can be achieved by an abstract intermediate language. The convention proposed here, however, imposes almost no restrictions (which is in line with our first principle of generality).

The language consists of: (1) *data containers* (e.g. actions and percepts), and (2) *parameters* for those containers. Parameters are *identifiers* and *numerals* (both represent constant values), *functions* over parameters, and *lists* of parameters. Data containers are: *actions* that are performed by agents, *results* of such actions, and *percepts* that are received by agents.

Syntactically each element of the IIL is an abstract syntax-tree (AST). Fig. 3 shows the relationship of the IIL-elements. Internally, each such element is stored as a tree of Java-Objects with the following structure:

- An `DataContainer` is either an `Action` or a `Percept`.
- An `Action` consists of (1) a string `name` that denotes the action's name, (2) an ordered collection `parameters`, containing instances of `Parameter`, representing the parameters of the action, and (3) an integer `timeStamp` encoding the exact time the action-object has been created.
- An `Percept` consists of (1) a string `name` that denotes the percepts's name, (2) an ordered collection `parameters`, containing instances of `Parameter`, representing the parameters of the percept, and (3) an integer `timeStamp` encoding the exact time the percept-object has been created.
- A `Parameter` is either a `Numeral`, a `Identifier`, a `ParameterList`, or a `Function`.
- A `Numeral` encapsulates a number `value`.
- An `Identifier` encapsulates a string `value`.

For the sake of readability each IIL-element can be printed either in a prolog-like or in a XML-notation. Note however, that these string representations are not supposed to be used on either the platform-side or the environment-side. They are for reading purposes only.

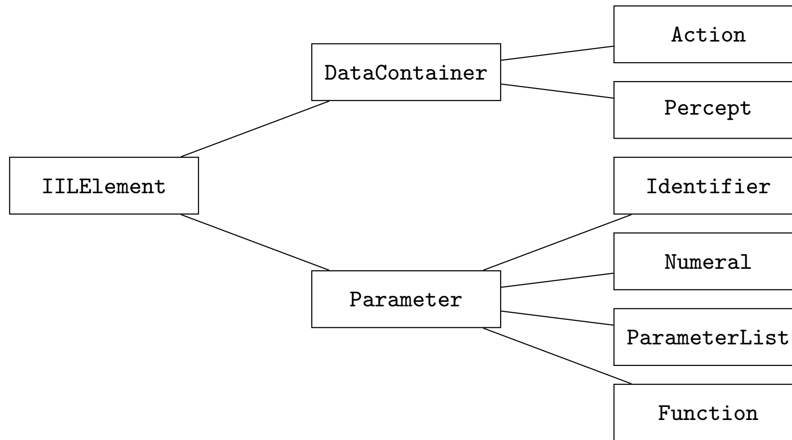

Fig. 3. The inheritance relation of the IIL-elements. Actions and percepts are data-containers. Each data-container consists of a name and an ordered collection of parameters.

Example 1 (a simple action with two atomic parameters). Assuming the existence of an entity that is capable of moving in a grid-like world, consider that this entity's action-repertoire includes an move-action that moves the entity to a position $[X, Y]$. This is the Prolog-like representation of such an action:

```
moveTo(2,3)
```

The action's `name`-field is `moveTo`. There are two parameters `2` and `3`. Both of the `Identifier`-type.

This is the XML-representation of the action:

```
<action name="moveTo">
  <actionParameter> <number value="2"/> </actionParameter>
  <actionParameter> <number value="3"/> </actionParameter>
</action>
```

□

Example 2 (another action using functions and a list). Consider now that the same entity is capable of performing a more complex action, that is following a path consisting of a sequence of positions at a given speed..

```
followPath([pos(1,1),pos(2,1)],speed(10.0))
```

The action's first parameter `[pos(1,1),pos(2,1)]` is a `ParameterList`. The list's elements are both instances of `Function`. Considering `pos(1,1)`, the function name is `pos`, both parameters are instances of `Numeral`. The second parameter of the action is a function, too.

Here is the XML-representation of the action:

```
<action name="followPath"><actionParameter>
    <parameterList>
      <function name="pos">
        <number value="1"/>
        <number value="1"/>
      </function>
      <function name="pos">
        <number value="2"/>
        <number value="1"/>
      </function>
    </parameterList>
  </actionParameter>
  <actionParameter>
    <function name="speed"><number value="10.0"/></function>
</actionParameter>
</action>
```

□

We do not need an explicit example for percepts, because syntactically percepts and actions are almost equivalent.

At this point, we have introduced the syntax of the IIL, and elaborated on it a bit by considering some examples. Now, we have to look at the semantics. The semantics of an action and/or a percept depends on the specific environment. Again, EIS does not make any assumptions here, except for the syntactical requirements.

After some experiments, a certain but trivial problem became evident. Some environments (e.g. UT 2004) provide identifiers that might be interpreted as variables on the platform side, thus rendering every IIL-expression that contains such identifiers unusable by the platform, causing errors that are difficult to deal with. To solve the problem we need to assume that none of the IIL-expressions that are distributed by a specific environment-interface contains interpreters that might be interpreted as variables.

3.3 Functional Point-of-View

What exactly is the correspondence between an environment-interface and the components (platform, agents, etc.)? We allow for a two-way connection via *interactions* that are performed by the components and *notifications* that are performed by the environment-interface.

Interactions are facilitated by function calls to the environment-interface that can yield a return value. For notifications we employ the *observer design pattern* (call-back methods, 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. The observer pattern is usually employed when a state-change in one object requires changes in 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 notifications 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 *AgentContest* example).

Agents and Entities: We make three assumptions: (1) there is a set of agents on the agent platform side (we do not know anything about them), (2) there is a set of controllable entities on the environments side (again we do not know anything about them), and (3) agents can control entities through the environment-interface. An important design decision that we had to make is to store in the environment-interface only identifiers to the agents, identifiers to the entities, and a mapping between these two sets. The reason for that decision is, as mentioned before, that we do not assume anything about the agent platform side or the environment side. Fig. 4 shows the agents-entities relation. The agents live on the agent platform side, they are known by the environment-interface through their identifiers. The entities live on the environment-side, and they are also known by their identifiers. The agents-entities relation is stored as a mapping between both sets of identifiers. In the *AgentContest*, each cowboy is a controllable entity. Cows are entities as well but they are not controllable. Each agent can control only a single cowboy.

In general, we allow the agents-entities relation to be arbitrary. For example, we also allow for one agent to be associated with several entities. This would be useful when using the agents&artifacts meta-model [23] to provide means for agent-coordination through the environment. An artifact would be an entity that can be controlled by several agents. Agents would perceive the state of the artifact and can act so as to change it.

Attaching, Detaching, and Notifying Observers: There are two directions for exchanging data between components and environment interfaces. One is via environment observers, which inform observers about changes in the environment or the environment interface. The second is via agent observers, which send percepts to agents. In order to facilitate sending events (i.e. percepts as notifications and environment events), the interface provides functions that allow for attaching and detaching observers, and for notifying components connected via observers. Listeners are useful when connecting to the *AgentContest* environment, since it is the simulator that actively provides agents with percepts.

Registering and Unregistering Agents: This step is the first to facilitate the interaction between agents and environments and establishing the agents' situatedness. It is necessary for the internal connection between agents and entities. The interface provides two methods: one for registering (`registerAgent`), and one for unregistering an agent (`unregisterAgent`). We note that the agents themselves are not registered to the interface; instead, identifiers as representatives

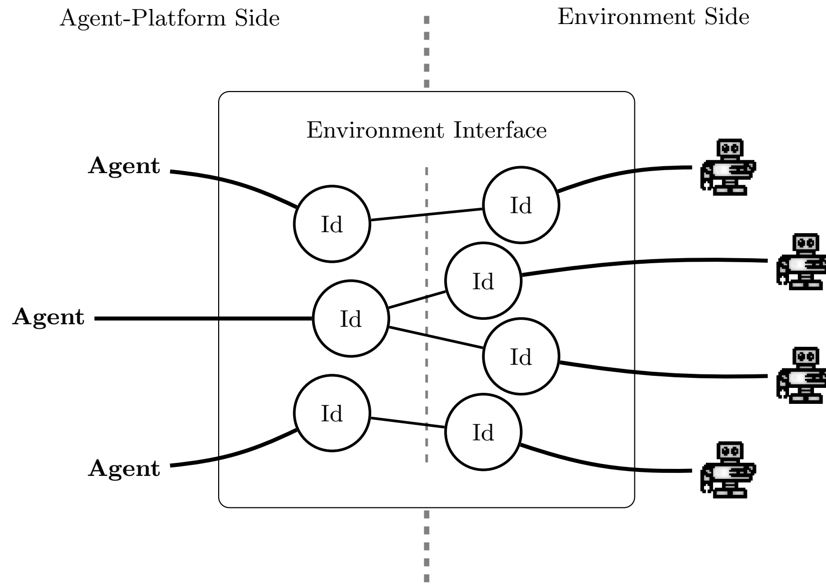

Fig. 4. The agents-entities relation. We distinguish between agents, which are platform-properties, and controllable entities, which are environmental properties. Agents have access to the entities effecting and sensory capabilities. In general the agents-entities-relation, depends on the specific environment-interface.

are stored and managed. We note that only identifiers representing the agents are stored and managed by the interface.

Adding and Removing entities: Entities are added and removed in a similar fashion to agents. Again identifiers representing entities are stored instead of the entities themselves. There are two methods: the first (`addEntity`) adds, and the second one (`deleteEntity`) removes an entity. Again this is necessary to facilitate the connection between agents and entities. Once an entity is added or removed, any observing components (platform and/or agents depending on the design of the platform) are notified about the respective events. This is done in order to allow components to react to changes in the set of entities in an appropriate manner.

Managing the Agents-Entities Relation: Associating an agent with one or several entities is the second and final step of establishing the situatedness of agents by connecting them to entities that provide effectory and sensorial capabilities. The agents-entities relation is manipulated by means of three methods. The first method (called `associateEntity`) associates an agent with an entity, the second one (`freeEntity`) frees an entity from the relation, and the third one (`freeAgent`), frees an agent. This can be done by the interface internally and by other components that have access to it as well. Restrictions on the structure

of the relation can be established by the interface. In the *AgentContest*, for example, one agent is supposed to control at most one virtual cowboy.

Performing Actions and Retrieving Percepts: The agents-entities relation is a connection between agents and the sensors and effectors of the associated entities. We establish two directions of information flow. Each direction corresponds to a typical step in common agent deliberation cycles. We have facilitated the management of the two directions of flow by following a unified approach whereby two methods are provided by the interface. The first one (`performAction`) allows an agent to act in the environment through the effectors of its associated entities. The second method (`getAllPercepts`) allows an agent to sense the state of the environment through the sensors of the associated entities. In the "cows and cowboys" scenario, nine actions are available. One for connecting to the server at a given IP address with valid username and password, and the other eight for moving the cowboy in each possible direction. The method `getAllPercepts` retrieves the last percept sent by the server.

Managing the Environment: Although different environments provide different support to manage the initialization, configuration, and execution of the environment itself, it is useful to include support for environment management in the environment interface. This allows agent platforms to provide this functionality by means of the interfaces that come with these platforms and relate environment functionality with similar functionality offered by the platform. For example, it is often useful to be able to "freeze" a running MAS simultaneously with the environment to which the MAS is connected by means of pause functionalities provided by the platform and the environment. As there is no common functionality supported by each and every environment, we have chosen to provide support for environment management by introducing a *convention* for labeling a set of *environment commands* and *environment events*. The commands that are part of the proposed environment management convention include *starting*, *pausing*, *initializing*, *resetting*, and *killing* the environment.

3.4 Implementation Details

The goal of developing an environment interface standard is to facilitate the easy exchange of environments. The interface would reduce the implementation effort of connecting agent platforms to environments. Of course, the effort of connecting to the environment through an environment interface should not substantially increase the effort needed for directly connecting agents to an environment. Below, we report on the experience we gained with adapting four agent platforms so that they support the environment interface as well as the experience gained with two environments that were adapted to support the environment interface.

In order to create an environment interface for a given environment, dedicated code that is specific to the environment is necessary. To that end, a particular Java interface has to be implemented. That interface enforces the functional contract introduced in subsection 3.3. Alternatively, the developer can inherit from

a class that contains a default implementation for all of the contract's methods. Whatever path the developers follow, they need to establish a connection to the environment.

Supported Agent Platforms To evaluate the ease of use and generality of the developed EIS concepts and components, we have connected four different APLs to example environments developed with the EIS. For 2APL, GOAL, Jadex, and Jason, a connection has been established with less than one day of coding effort each.

2APL proved to be compatible with EIS. In order to establish a connection a two-way converter for the interface intermediate language had to be developed. Furthermore, the environment loading mechanism of 2APL had to be replaced with the environment-interface loading mechanism provided by EIS. Percepts sent by EIS using the observer functionality are translated into 2APL events and handed over to the event-handling mechanism of the interpreter. Finally, special external actions have been added to facilitate the manipulation of the agents-entities relationship: (1) retrieving all entities, (2) retrieving all free entities, (3) associating with one or several entities, and (4) disassociating with one or several entities.

The original environment interface of GOAL did not fit with everything provided by the environment interface. It nevertheless proved quite easy to connect the interface to GOAL as most functionality provided by the interface is straightforwardly matched to that provided by the GOAL agent platform. Similar to 2APL, a two-way converter for the interface intermediate language had to be developed with little effort required. There were no percepts as notifications (like events in 2APL), prior to the adaptation to EIS. GOAL only allowed for retrieving all percepts in a distinct step of the deliberation cycle. Percepts as notifications are now collected and processed together in the step where all percepts are processed. Also, the MAS file specification of GOAL has been extended. Now one can use launch rules to connect specific agents with specific entities. This allows for instantiating agents even during runtime.

For connecting Jadex agents to EIS, it is sufficient to make all agents of one application aware of the concrete EIS object, implementing the current environment. In order to do this in a systematic way, the Jadex concept of *space* was used. A space may represent an arbitrary underlying structure of a MAS that is known by all agents. To support the EIS, a special `EISSpace` has been provided, which implements the required interfacing code for connecting to an EIS-based environment. Therefore, the participation in such an environment can now simply be specified in the Jadex application descriptor (".application.xml"). When such a defined application is started, the initial agents as well as the EIS environment will be created. Agents can then access EIS by fetching the corresponding space from their application context and use the EIS Java API directly for, e.g., performing actions or retrieving percepts.

Jason's integration with EIS was straightforward since almost all concepts used in the EIS are also available in Jason. The integration consists essentially

of: (1) the conversion of data types, and (2) the development of a class that adapts EIS environments to Jason environments. In regards to (1), all EIS data types have an equivalent in Jason. Although some data types in Jason (e.g., Strings) do not have a corresponding type in EIS, they can be translated to EIS identifiers. In regards to (2), the adaptor is a normal Jason Environment class extension that delegates perception and action to the EIS. The adaptor class is also responsible for registering the agents with the EIS as they join a Jason multi-agent system and wake them up when the environment changes (using the observer mechanism available in EIS). From all the concepts used in EIS, only that of "entities" is not supported by Jason as all actions and perceptions are relative to an agent and the overall environment rather than a particular entity therein. For sensing, the chosen solution was to add annotations to percepts that indicate the entity of origin. For actions, in case the agent is associated with exactly one entity, the action is simply dispatched to that entity. Otherwise, a special action that receives the relevant entity as a parameter must be used.

Implemented Environments The environment interface comes with several very simple examples of environments for illustrative purposes. These examples are mainly provided for clarifying some of the basic concepts related to the interface. We briefly discuss here two EIS-enabled environments, that may be used by any agent platform that supports EIS.

The *elevator environment* is a good example of an environment that was not built specifically with agents in mind, and is available from [1]. The environment is a simulator of arbitrary multi-elevator environments where the elevators are the controllable entities and the people using the elevators are controlled by the simulator. It comes with a graphical user interface (GUI) and a set of tools for statistical analysis. The environment had been originally adapted for the GOAL platform. The additional effort required to re-interface that environment to EIS was very little. The main issue was the event handling related to the initial creation of elevators, a functionality provided and supported by the environment interface which required some additional effort for adapting the environment to provide such events. The environment provides actions that take time (durative actions) instead of discrete one-step actions, which illustrates that the interface does not impose any restrictions on the types of actions that are supported. Similarly, elevators only perceive certain events but not, for example, whether buttons are pressed in other elevators. The percept handling related to this was easily established, illustrating the ease with which to implement a partially observable environment. We have successfully used the elevator environment with GOAL and 2APL.

Connecting to the MASSim-server turned out to be easy. As already mentioned, the entities in the *AgentContest*-environment are cowboys that herd cows. From the implementation point-of-view each connection to an entity is a TCP/IP connection. Acting is facilitated by wrapping the respective action into an XML-message and sending it to the server. Perceiving is done by receiving XML-messages from the server and notifying possible agent-listeners. Further-

more, for the sake of convenience, percepts are stored internally for a possible active retrieval. Much effort had to be invested in mappings from the interface intermediate language to the XML-protocol of the *AgentContest* and vice versa. We have shown that the interface does indeed not pose any restrictions on the connection between itself and environments.

Finally, it is worth mentioning that an interface to Unreal Tournament 2004 [18] is under developmentGrown out of the need for a more extensive evaluation of the application of logic-based BDI agents to challenging, dynamic, and potentially real-time environments, this EIS interface might help putting agent programming platforms to the test.

Evaluation Summary The relative ease with which the interface has been connected to four agent platforms and various environments already indicates that the interface has been designed at the right abstraction level for agent-environment interaction. The four agent platforms differ in various dimensions, regarding, for example, the functionality provided for handling percepts and actions (is the platform more logic-oriented or JAVA-based?) and how environments were connected to these platforms before using the interface. The environment interface nevertheless could be connected to each of the platforms easily, thus providing evidence of its generality and as well. Of course, we need more agent platforms to use the environment interface, and we have invited other platform developers to do so, but we do not expect this will pose any fundamental new issues. Initial experience with various environments has also shown that little to no restrictions are imposed on the types of environments that can be connected to an agent platform using the interface. The interface, for example, can support both real-time or turn-based environments, as well as environments that differ in other respects. Although we have mainly discussed software environments, there is no principled restriction imposed by EIS that would make it only applicable to such environments. It has been shown already in the past that it is possible to connect agent platforms to embedded platforms such as robots. EIS just provides another, more principled approach for doing so. In fact, it is planned to use EIS to connect to a robotic platform in the near future.

4 Related Work

The EIS was designed as a building block for an agent application, providing a standardized way of interfacing the agents with environmental components. In the context of agent applications, at least the following forms of environments can be distinguished: (1) environments in agent-based simulation models, (2) virtual environments such as testbeds or computer games, (3) real application components such as enterprise information systems, and (4) coordination infrastructures.

Agent-based simulation models can be used for performing experiments and analyzing the obtained result data. Agent simulation toolkits are specifically designed for this purpose and often employ custom agent models (e.g. simple

task-based agents) and a proprietary form of defining the environment behavior. Usually, there is a tight coupling between agents and the environment that is designed to support these toolkit-specific models. Therefore, simulation toolkits are closed in the sense that they do not support (and are not meant to) connecting external agents to simulated environments or simulated agents to external environments.

The specialized architecture Koko [27] provides a reusable and extensible environment, aiming at an enhanced user experience by linking independent applications. With our work we neither focus on human interaction with agents or the environment, nor are we exclusively interested in multiplayer and/or social games. We see these only as a single class of test-cases out of many ones.

Agent programming testbeds and contests, such as TAC, [5], RoboCup, [4] and the Multi Agent Contest [3], are specifically designed to offer open interfaces for connecting different types of agents to the provided test environment. Moreover, some network-based computer games with remote playing capabilities (e.g. Unreal Tournament) offer interfaces for controlling entities in the game environment which have been adapted to connect to software agents instead of human players [11]. All of these interfaces are quite specific with regards to the testbed or game they were created for, and therefore agent platform developers have to repeat the implementation effort of connecting their agents to each of these interfaces.

To connect agents to an environment composed of real application components, different options are available. Application-centered approaches would directly use available component interfaces or domain specific standards (such as HL7 in the healthcare domain) for the connection. Depending on the severity of the "impedance mismatch" between the component interface and the agent platform, this can become quite laborious and additionally has to be repeated for each platform and each application. Agent-centered approaches try to "agentify" the environment components, leading to a more seamless and straightforward connection. For example WSIG (Jade) [2] is an infrastructure that allows agents to interact with web services as if they were agents and vice-versa.

One well known approach for coordinating agents is by using blackboard approaches, which offer agents a possibility to decouple their interactions in terms of time and potential receivers. Besides passive blackboards acting as information stores only, also more advanced tuple spaces such as ReSeCT [22] have been devised with which one can also capture domain logic in terms of rules. The Open Agent Architecture (OAA) [13] is another form of coordination environment, in which the cooperation among agents and also humans is facilitated by automatic task delegation and execution. In contrast to EIS, these approaches focus on information exchange and problem solving and do not tackle the question of how environments could be generically interfaced.

Organizational or institutional approaches such as Islander [16] and MOISE [19] regulate agent behavior at high-level allowing designers and/or agents to define, monitor, and enforce certain kinds of organizational constraints (e.g. norms and group membership). The latest platform for MOISE is founded on the notion of

organizational environment where agents can perceive and act on their organization. This kind of environment can also contain artifacts specially developed to enforce some norms (e.g. a surgical room's door that forbids agents to enter if they do not play the role of doctor). Other approaches affect more directly agent behavior, for example biologically inspired approaches such as pheromone-based techniques to guide agent movement. While these approaches make use of the notion of environment, they are quite domain specific and do not allow for arbitrary environment development. In contrast, the A&A model [23] has been proposed as a generic paradigm for modeling environments. In the A&A paradigm, an application is composed of agents as well as so called artifacts. While the model makes no restricting assumptions with respect to the agents, the interface and operation of an artifact is intentionally quite rigidly defined. An implementation of the A&A model is available in form of the distributed middleware infrastructure CArtAgO [25].

We see EIS not as a competitor, but rather as a desirable complement to the above mentioned approaches. For example, one possible use of the EIS standard is reducing the required implementation effort for connecting agent to, say, virtual environments, as once an EIS-based interface has been developed for a contest or game, it can easily be reused by different agent platforms. Unlike FIPA-compliant approaches such as the WSIG, the focus of the EIS is providing a lean interface, i.e., when FIPA-compliant communication is not necessary, the EIS allows achieving similar openness and portability with much less effort. In particular, we see much potential in a combination of EIS and CArtAgO. Currently, there are specific bridges available for connecting agent platforms such as Jadex, Jason and 2APL to CArtAgO [24]. Implementing an EIS bridge for CArtAgO could lead to a universal implementation that could be used to connect CArtAgO to any agent platform (if it is already EIS-enabled). In general, the EIS standard will facilitate connecting any agent platform to all sorts of environments (A&A based as well as others).

5 Conclusion

The design and implementation of our proposal for an environment interface standard (see [8] for a more detailed exposition and more technical details) is motivated by the fact that it has been difficult to connect arbitrary agent platforms to many of the available environments. The design of the interface provides additional insight into the general problem of agent-environment interaction. At a conceptual level, the development of the environment interface has yielded insight, for example, into some of the distinguishing features of existing agent platforms. For example, where some platforms expect events initiated by the environment other platforms are based on a polling model for retrieving percepts.

The initial results of applying the interface to various agent platforms and environments have been very encouraging: they demonstrate the generality and usability of our interface. The environment interface standard allows the portability and reuse of application and testing environments across existing and newly

developed agent platforms. Furthermore, it provides a basis for heterogeneous agent applications composed of agents implemented in different agent platforms. The experience so far has also shown that connecting to and using the interface requires minimal effort and can be implemented easily.

Although the environment interface proposed here provides a solid basis for agent-environment interaction, there are some topics that require additional work. One of these topics involves the environment management system which has only been partly supported by most agent platforms; it facilitates combinations of agent platform and environment functionalities such as combined resetting of MAS and environment, but this requires additional investigation. We also need to gain more experience with the dynamic addition and removal of entities and the handling of such events by platforms. Related to the previous point, there is the issue of managing various types of entities. For example, how can the interface be extended to support the identification of these different types? Finally, we need to get more agent platforms, including platforms from multi-agent based simulation and other areas, involved and support the environment interface to establish our proposal as a genuine (de facto) standard.

References

1. Elevator simulator homepage. `http://sourceforge.net/projects/elevatorsim/`.
2. Java Agent DEvelopment Framework homepage. `http://jade.tilab.com/`.
3. Multi Agent Contest homepage. `http://www.multiagentcontest.org/`.
4. RoboCup homepage. `http://www.robocup.org/`.
5. Trading Agent Competition homepage. `http://www.sics.se/tac/`.
6. R. Adobbati, A. Marshall, A. Scholer, S. Tejada, G. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.
7. J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2009.
8. T. M. Behrens, J. Dix, and K. V. Hindriks. Towards an environment interface standard for agent-oriented programming. Technical Report IfI-09-09, Clausthal University of Technology, Sept. 2009.
9. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
10. L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI-agent system combining middleware and reasoning. In R. Unland, M. Klusch, and M. Calisti, editors, *Software agent-based applications, platforms and development kits*, 2005.
11. O. Burkert, R. Kadlec, J. Gemrot, M. Bda, J. Havlcek, M. Drfler, and C. Brom. Towards fast prototyping of IVAs behavior: Pogamut 2. In *Proceedings of 7th International Conference on Inteligent Virtual Humans*, 2007.
12. M. Buro. Call for AI Research in RTS Games. In *AAAI-04 AI in Games Workshop*, 2004.
13. A. Cheyer and D. Martin. The open agent architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, March 2001. OAA.

14. M. Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
15. M. Dastani, J. Dix, and P. Novák. Agent Contest Competition - 3rd edition. In M. Dastani, A. Ricci, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Programming Multi-Agent Systems 5th International Workshop, ProMAS 2007 Honolulu, HI, USA, May 15, 2007 Revised and Invited Papers*, number 4908 in Lecture Notes in Artificial Intelligence, Honululu, US, 2008. Springer.
16. M. Esteva, D. de la Cruz, and C. Sierra. Islander: an electronic institutions editor. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1045–1052, New York, NY, USA, 2002. ACM.
17. K. V. Hindriks and T. Roberti. Goal as a planning formalism. In *MATES 2009 Proceedings*, pages 29–40, 2009.
18. K. V. Hindriks, B. van Riemsdijk, T. Behrens, R. Korstanje, N. Kraaijenbrink, W. Pasman, , and L. de Rijk and. Unreal GOAL bots. In *Preproceedings of The AAMAS-2010 Workshop on Agents for Games and Simulations*, to appear.
19. J. F. Hübner, O. Boissier, R. Kitio, and A. Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents: "giving the organisational power back to the agents". *Journal of Autonomous Agents and Multi-Agent Systems*, 2009.
20. R. Z. Mili and R. Steiner. Modeling Agent-Environment Interactions in Adaptive MAS. In *Engineering Environment-Mediated Multi-Agent Systems International Workshop, EEMMAS 2007, Dresden, Germany, October 5, 2007. Selected Revised and Invited Papers*, pages 135–147. Springer, 2008.
21. J. Müller. Towards a Formal Semantics of Event-Based Multi-Agent Simulations. In *Multi-Agent-Based Simulation IX, International Workshop, MABS 2008, Estoril, Portugal, May 12-13, 2008, Revised Selected Papers*, Lecture Notes in Artificial Intelligence, pages 110–126. Springer, 2009.
22. A. Omicini. Formal ReSpecT in the A&A Perspective. *Electronic Notes of Theoretical Computer Science*, 175(2):97–117, 2007.
23. A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.
24. A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hübner, and M. Dastani. Integrating artifact-based environments with heterogeneous agent-programming platforms. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-08)*, pages 225–232. IFAAMAS, 2008.
25. A. Ricci, M. Viroli, and A. Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers*, volume 4389 of *Lecture Notes in Artificial Intelligence*, pages 67–86. Springer, 2007.
26. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2nd edition, 2003.
27. D. Sollenberger and M. Singh. Architecture for affective social games. In F. Dignum, J. Bradshaw, B. Silverman, and W. van Doesburg, editors, *Agents for Games and Simulations*, volume 5920 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin / Heidelberg, 2009.