Ann Math Artif Intell (2011) 61:261–295
DOI 10.1007/s10472-010-9215-9

Towards an environment interface standard for agent platforms

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

Published online: 3 November 2010

Abstract We introduce an interface for connecting agent platforms to environments. This interface provides generic functionality for executing actions and for perceiving changes in an agent's environment. It also provides support for managing an environment, e.g., for starting, pausing and terminating it. Among the benefits of such an interface are (1) standard functionality is provided by the interface implementation itself, and (2) agent platforms that support the interface can connect to any environment that implements the interface. This significantly reduces effort required from agent and environment programmers as the environment code needed to implement the interface needs to be written only once. We propose that the interface presented may be used as a standard that enables agents to control entities in environments. Our starting point for designing such a generic interface is based on a careful study of the various interfaces used by different agent programming languages to connect agent programs to environments. We discuss several case studies that use our interface (an elevator simulator, the well-known agent contest, and an implementation of the interface to connect agents to bots in Unreal Tournament 2004).

Keywords Agent · Environment · Interface

Mathematics Subject Classification (2010) 68T42

T. M. Behrens (✉) · J. Dix
Technical University of Clausthal, Julius-Albert Straße 4, 38678, Clausthal, Germany
e-mail: behrens@in.tu-clausthal.de

J. Dix
e-mail: dix@tu-clausthal.de

K. V. Hindriks
Technical University Delft, Mekelweg 4, 2628CD, Delft, The Netherlands
e-mail: k.v.hindriks@tudelft.nl

1 Introduction

Our objective is to *design and develop a generic environment interface standard* (EIS) that facilitates connecting agents programmed in various agent platforms to environments. We have initially concentrated on connecting agents programmed in agent programming languages to environments. The proposal presented here, however, is not limited to such languages. In principle the interface can be used to connect arbitrary agent platforms to environments. The interface is explicitly aimed at connecting *agents* to *environments*. It does not require any sophisticated property of an agent: A minimalistic notion of an agent will do.

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,
– an EIS allows for the easy distribution of existing and future environments (Multi-Agent Programming Contest, Unreal Tournament, 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.

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. Because a significant effort has been invested in developing interfaces to environments and the environments themselves it is crucial to reach these goals. Obviously, there is a trade-off between them. Our basic strategy for designing a generic environment interface is (1) to start with existing platforms, and (2) to try to merge this into a generic interface which is sufficiently close to these existing approaches.

In the first interface standard proposal below, we will not introduce any features that go beyond existing functionality for relating APLs to environments. With one exception, namely a feature that allows the connection between agents and environments to be a very dynamic one. We discuss this feature in more detail below and leave the discussion of other features to further research.

In order for the proposed interface to become a standard it is important that a significant number of agent platforms adopt it. We have been very careful to take into account the requirements that various agent platforms impose to facilitate easy adaptation. Our strategy in designing the interface has been to minimize the required effort for adapting to this new standard. In part this has been realized by developing an interface that provides support for various standard functions to connect to environments. Platform developers thus do not need to put effort themselves in providing this code.

The main assumption we make is that certain *minimal requirements* are satisfied. In more detail, the agent platform needs to support a *minimal agent-based abstraction*: *Actions and percepts are treated as first-class entities*. Most agent platforms satisfy this assumption already or they can be easily modified to fully support it.

The standard is based on

– a *meta-model for agent–environment interaction*, that defines the components of the standard and their interactions, and
– a *set of principles* that encode useful constraints for implementing the standard.

The paper is organised as follows. We first discuss related work in Section 2. In Section 3 we introduce a *meta model for agent–environment interaction* that describes the components, the relations between these components and functionalities provided by these components that are required to facilitate effective agent–environment interaction. The meta model provides a well-defined model on which we have based the design for the implementation of the interface. The interface implementation is described in Section 4. In Section 5 we present several case studies that illustrate the application and usefulness of the interface for connecting agent platforms to environments and show how the design facilitates implementing the interface for a number of different environments. Finally, Section 6 concludes the paper.

2 Related work

In the following, we compare our approach with existing and established ones. It is important to clarify and contrast our view of an environment with other views. For example, we are not concerned with the structure of the environments that we would like to connect to. Thus we do not assume or require anything about e.g., the topology of the considered environments. We intend to treat the environments as much as black-boxes as possible. Also, we do not adopt the view that agents are part of the environment and that such an environment facilitates the computation—that is for example by providing means for communication—of the agents. We take the perspective that there is a conceptual gap between agents and environments: We treat both as *separate components*.

The A&A model [22] has been proposed as a generic paradigm for modeling environments. In the A&A paradigm an application is composed of agents as well as *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 the form of the distributed middleware infrastructure CArtAgO [26].

We see EIS as a desirable complement to the above mentioned approach. E.g. one possible use of the EIS standard is to reduce the required implementation effort for connecting to e.g., virtual environments, as an already developed EIS based interface to a contest or game 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 a lot of potential in a combination of EIS and CArtAgO. Currently, there are particular bridges available for connecting agent platforms such as Jadex, *Jason* and 2APL to CArtAgO [25]. Implementing an EIS bridge for CArtAgO could lead to a universal implementation, which in turn could be used to connect CArtAgO to any kind of agent platform that supports EIS. In general, the EIS standard will enable connecting an agent platform to any kind of environment (A&A based as well as others).

Various other platforms have been connected to environments that we envisage EIS will support as well. One prominent example is Soar [18] that has been connected among other things to computer games. Soar is a general and flexible architecture for research in cognitive modeling across a wide variety of behavioral and learning phenomena and has proved to be useful for creating knowledge-rich agents that could generate diverse intelligent behavior in complex, dynamic environments [16]. SOAR has been used in computer games [15] and as a test bed for intelligent synthetic characters [17]. A SOAR-agent consists of a symbolic long-term memory, which itself consists of procedural memory, semantic memory, and episodic memory, and also of a symbolic short-term-memory. The long-term memory is changed via reinforcement-, semantic-, and episodic-memory respectively. The short-term memory is derived from the agent's perception and its long-term memory. Actions are motor commands encoded in a buffer in short-term memory. A decision procedure selects operators and impasses. On the lowest level, processing is matching and firing rules. SOAR has been inspired by human capabilities. As an example for SOAR's capabilities, we mention that it has been connected to UNREAL TOURNAMENT, which in turn has been extended to support design in complex, storytelling environments [19].

In Section 5.3 we present a case study and describe the implementation of the interface introduced here for the gaming environment UNREAL TOURNAMENT 2004 (UT2004). This case study is one of the more challenging case studies we have done. Part of the challenge has been to supply an adequate layer that allows agents to control bots in UT2004 at a reasonable level of abstraction. We do not believe it is possible to establish *the ultimate right level of abstraction* but some level of abstraction is needed in any agent-based approach. Although the environment interface introduced here does not assume anything specific about the level of abstraction, other work connecting agents to the gaming environment Quake has clearly shown the inefficiency of providing agents with the burden to control too many low-level details (such as providing low-level guidance as how to move to corridors; see e.g., [14]). This issue relates to the more generic topic of defining agent–environment interaction abstractly and is discussed below in the meta model section. The specific interaction layer for UT2004 is discussed in more detail in Section 5.3.

Various other projects are documented in the literature that connect an (agent) platform for controlling bots in UT2004. Most of these projects are built on top of Gamebots [12] or Pogamut [8], an extension of Gamebots: See e.g., [13, 23] which use Gamebots and [30] which use Pogamut.[1] Gamebots is a platform that acts as a UT2004 server and thus facilitates the transfer of information from UT2004 to the client (agent platform). The GameBots platform comes with a variety of predefined tasks and environments. It provides an architecture for connecting agents to bots in the UT2004 game while also allowing human players to connect to the UT2004 server to participate in a game. Pogamut is a framework that extends GameBots in various ways, and provides among others an IDE for developing agents and a parser that maps Gamebots string output to Java objects. We have built on top of Pogamut because it provides additional functionality related to, for example, obtaining information about navigation points, ray tracing, and commands that allow controlling the UT2004 gaming environment. The interface provided by Pogamut, however, has not been made available as is. Instead a behavioral layer that provides

[1] Jacobs et al. [11] is an exception, directly connecting READYLOG agents via TCP/IP to UT2004.

various abstractions of the functionality made available by Pogamut has been built to make a range of actions and percepts available to agents at a higher level of abstraction as that offered by Pogamut. The aim has been to provide an interface here for UT2004 that is adequate for BDI-based agents. These agents make decisions at the *knowledge level* [21] or *cognitive level* which requires a different abstraction than agents not based on the BDI metaphor.

Various projects aimed at a different abstraction level and do not aim to connect high-level BDI agents to UT2004. One example is the behavior-based framework called pyPOSH that has been connected to UT2004 using Gamebots [23]. The motivation has been to perform a case study of a methodology called Behavior Oriented Design [7]. The framework provides support for reactive planning and the means to construct agents using Behavior Oriented Design (BOD) as a means for constructing agents. BOD is strongly inspired by Behavior-based AI and is based on *"the principle that intelligence is decomposed around expressed capabilities such as walking or eating, rather than around theoretical mental entities such as knowledge and thought"* [23]. One other difference between the interface we have implemented and that of [7] is that our interface clearly separates the actions and behaviors that can be performed through the interface from the percepts that may be obtained from sensors provided by the environment.

Other projects that have been reported in the literature clearly demonstrate the duplication of effort that has resulted from implementing UT2004 interfaces that are dedicated to a specific platform. Tweedale et al. [29] briefly discusses an interface called *UtJackInterface* that allows JACK agents [1] to connect to UT2004. The effort has been motivated by the *"potential for teaming applications of intelligent agent technologies based on cognitive principles"*. The interface itself reuses components developed in the Gamebots and Javabots project to connect to UT2004. Some game-specific JACK code has been developed to *explore, achieve, and win* [29]. The interface provides a way to interface JACK agents to UT2004 but does not provide a design of an interface that facilitates reuse. Coming back to Soar (see above), we note that it has also been used to control computer characters. Soar provides so-called *operators* for decision-making. Soar has been connected to UT2004 via an interface called the Soar *General Input/Output* which is a domain independent interface [17]. Although the interface is domain independent it is specific to Soar and still does not support reuse between agent platforms. A similar effort is reported in [4] where a connection between the cognitive architecture ACT-R to Unreal Tournament is discussed. In this case, Gamebots is again used to develop an interface from Unreal Tournament to ACT-R. As a last example, we mention the work reported on connecting the high-level logic-based language ReadyLog (a variant of Golog) to UT2004 [11]. Similar issues are faced to provide an interface at the right abstraction level to ensure adequate performance, both in terms of responsiveness as well as in terms of being effective in achieving good game performance.

To summarize, various efforts exist that describe in more or less detail the interfaces developed for connecting an (agent) platform to UT2004. It is remarkable to note that these efforts basically have been independent projects that did not show any reuse of e.g., source code from earlier projects (other than Gamebots and Pogamut). It also seems that lessons learned have not been documented well enough to enable transfer of knowledge in this area. We believe that there is sufficient potential for such reuse which motivates our effort to introduce and apply a well-designed interface that explicitly aims at reuse capabilities. This provides a principled approach to reuse of our effort to facilitate control of Unreal bots. We believe our effort also facilitates comparison with other agent platforms that support the interface introduced here and thus contributes to the evaluation of agent platforms.

The HLA [28] (high level architecture) has some similarities with EIS. The HLA has been designed to be a general purpose architecture for distributed computer simulation systems. It is also a IEEE standard for communication between simulations. The HLA consists of three main components. (1) A *federate* models one or several entities or can be used for other purposes. For example each agent as well as the environment can be federates. Several federates combined form a *federation*. (2) The *federation model* defines the data-exchange between federates in a federation. And (3) the *runtime infrastructure* provides the communication between federates in a federation or between federations. These components already show a couple of differences to EIS. The first difference lies in the assumptions about the agents. In HLA an agent can be modeled as a federate, which already specifies the structure of such an agent to a certain extent. In EIS, we almost fully ignore what an agent is. The second difference lies in assumptions about environments. This is the same as for the agents. An environment is a federate, which already has some structure. The third difference lies in the fact that agents and environments are both federates. Since we do not make assumptions about both environments and agents, we certainly do not assume that both have some structure in common. The fourth difference is the communication between components. We only provide means for communication between environments and agents, not between agents themselves. Moreover we have a strict syntax for that communication. And the final difference is HLA's focus on simulations. Our focus is not only on simulations. We are interested in all sorts of environments.

3 A meta-model for agent–environment interaction

Before we design and describe the implementation of an interface for connecting agents to entities in an environment, we need a model describing the agent–environment interaction and related functionality that such an interface should support. We call this model a *meta model for agent–environment interaction*. As a first step towards this model, we compare different APL platforms with respect to how they facilitate accessing different environments. Figure 1 shows the meta-model which will be the result of our examinations.

3.1 APL platform comparison and summaries

We compare the APL platforms 2APL, Goal, Jadex, and *Jason* concentrating 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 should be available?

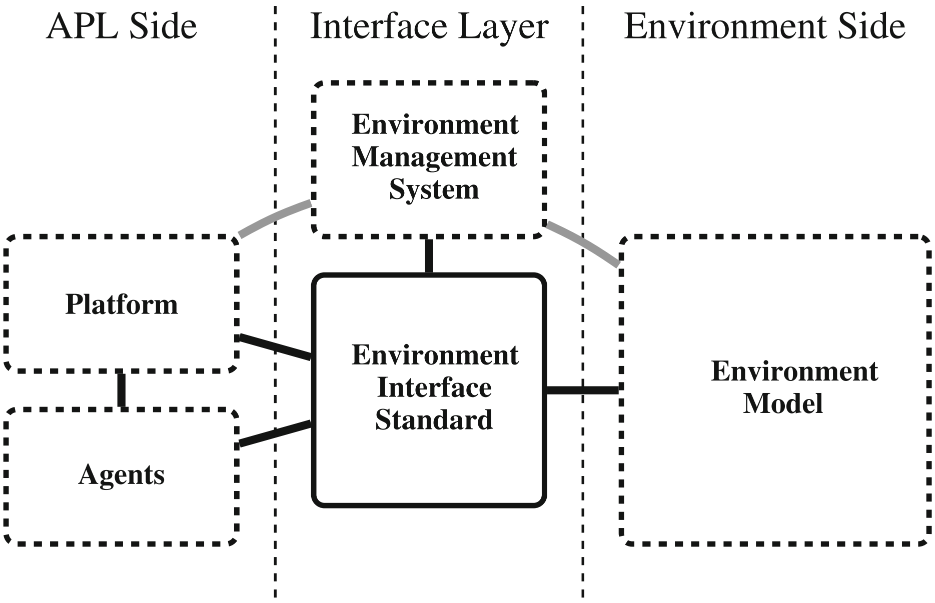


Fig. 1 The components of our proposal. We believe the platform-side that contains the agents should be separated from the environment model. The interface layer defined by EIS acts as a kind of glue layer that facilitates the interaction of the components

3.1.1 2APL environments

Creating new environments in 2APL means to implement a class `Env` [10], that extends `apapl.Environment` (see below). The package name defines the environment name. Environments are distributed 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`. This method 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(String name)` 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.

We note 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.

3.1.2 GOAL *environments*

To use a GOAL-environment, one 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 [24]. To work as an environment, a class has to implement the Java-interface with the name `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 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 otherwise. 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 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.

3.1.3 JADEX *environments*

In JADEX [6], 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. Here is a code-snippet of the garbage-collector-agent:

```
<agent [ \ldots ]>

  <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>

  [ \ldots ]

    </beliefs>

    [ \ldots ]

</agent>
```

The environment is stored in the belief-base as a Java-object.

3.1.4 Jason environments

To create a new environment a class has to be established extending the Java-class with the name `jason.environment.Environment` [5]. 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()` 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`.

We note that the environment allows for (external) control over action-execution strategies and provides logging-functionality (redirecting system interface `System.out`). Although the environment defines these functions, the two essential methods are `executeAction` and `getPercepts`, which provide a minimal agent interface. These methods are most interesting for our research, because we are about to separate agents from environments, and these methods define an interface between those two classes of components.

3.1.5 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* one has 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 properties 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

Table 1 Comparison-matrix to give an overview

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 environment object or sending a message to an environment agent	Invoking a method
Abstract environment functionality	Mapping from agent-names to agent-objects	No special functionality defined	No abstract environment	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 accessibility	Jar-files	Jar-files	Everything that is in the class-path	Jar-files

(in comparison to 2APL and GOAL) open-source.[2] One can implement environments without sticking to the instructions, but this does not seem to be the way intended by the developers (Table 1).

– *Perceiving* In 2APL/GOAL/*Jason* perceiving and acting means invoking special methods in the environment-class. 2APL allows for active and for passive sensing.[3] 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 a 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

[2]In the meanwhile 2APL's source-codes have been made available to the general public; when we started to work on EIS the code was not available. At the moment of writing, access to GOAL's source code is provided on request.

[3]Causally speaking, active sensing is sensing by performing a sense action and passive sensing is sensing without performing such an action, i.e., somehow automatically. We will come back to that later.

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, and notifiers for agents that the environment has changed. JADEX does not define any interface or abstract class for implementing 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.

3.2 Principles

We have identified the following principles that we think should be adhered to when designing an interface standard:

1. *Portability* We aim to facilitate the exchange of environments between platforms by (1) downloading the specific interface to an environment, (2) quickly adapting the MAS, and (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.
   Concerning 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. *Generality* 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 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

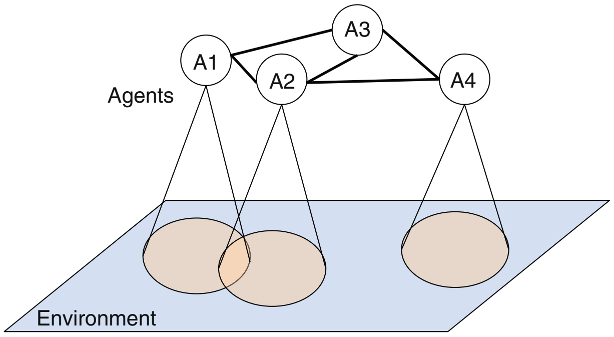


Fig. 2 Environment-MAS model

their states if agents act), and in which the environment will schedule the actions (like the *AgentContest* in which the environment can evolve on its own and schedules the agent's actions).
  - any assumptions on agent communication or on organizational 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).
  - 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*).
  - any assumptions about how an agent platform controls entities in an environment.
- 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*[4] 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).
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 emphasize again that we do *not* assume a one-to-one relation between agents and controllable entities.

[4]Of course there are two different notions about how agents and environments are related. Some people consider agents to be a *part of* the environment, others consider agents to be *separate from* environments. Although both stances are very different, there is no problem with assuming any one of them. We alert the reader not to confuse the two different point-of-views while reading this article.

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. Because different platforms in the general case come for example with different knowledge representation languages, it is necessary to provide a common ground that can be used by them. We intend to propose a standard based on special Java-objects, defining a language that represents each item as an abstract syntax tree.
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 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.

3.3 Meta model

Taking into account the related work discussed in Section 2 and the discussion of environment interface support provided by various agent programming platforms, we present here an abstract model of agent–environment interaction. This model also includes generic functionality for managing an environment. Such functionality enables an agent platform to manage the state of an environment generically. The model introduced below is called a *meta model* as it does not describe a specific agent–environment interface but is intended to describe generic functionality that should be available in any agent-based interface supporting agent–environment interaction.

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 [27] 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 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 particular about how these 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 Fig. 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, they are able to change the same parts of the 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 *AgentContest* 2008–2010,[5] 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 through EIS. As we will show later, agents and entities are not directly connected, they are connected via EIS.
– *Environment management system (EMS)* We assume this component to provide all the actions for managing an environment. Such actions might be: initializing

[5]http://www.multiagentcontest.org

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 (on environment- and/or on platform side), we would like to leave this issue "open" to a certain extent. We believe that this will be clarified after further practical experiments.
– *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).

Figure 1 presents the meta model schematically. It also depicts the relations that need to be supported between the various components. One of the most important relations that should be part of an agent–environment interface is what we call the *agents-entities-relation*. This relation associates agents with *controllable entities* in the environment. The relation is maintained to provide basic bookkeeping functionality. This bookkeeping functionality provides a key role as it determines which agents are allowed to control which entities and also determines which percepts should be provided to which agents.

To complete the overall picture, we also establish the following connections between

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

We now elaborate further on some essential details. For perceiving there are different models available. We also need to examine the components of our approach and their relationships.

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 this does *not relate* to the usual differentiation in robotics regarding active and passive sensors.[6]

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 *AgentContest* [9] for example, the connected APL platforms are not allowed to control the execution of the environment. Note, that although the EMS is conceptually on both sides, from the implementation point-of-view it is, like the action/percept-interface, a part of EIS itself and not a standalone-piece of software. This EMS-component is now more or less stable, but we can observe the need for a more sophisticated account, that would need solving soon (future work).

Also, 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 the *AgentContest*) 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. The last item (3) considers the case where EIS would be used for connecting to a new environment that still has to be developed. In that case, EIS should not put up any barriers for developing new environments.

4 Implementation

We now turn to the implementation of the interface and explain some of the design choices that we have made. We start by motivating the need for an *interface intermediate language*. This language facilitates the exchange of data between different

[6] "*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*" [20]. The term active sensor is not the same as active sensing. Active sensing is used to denote in a system that 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. Typical examples of data items that need to be exchanged are percepts, actions, and events. The interface intermediate language makes it possible to provide robust support for the exchange of these data items and, more importantly, is required to enable agent platforms to implement generic support for handling these items.

We then continue describing the functional architecture of the interface. The interface provides functions for

1. attaching, detaching, and notifying observers;
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.

4.1 Running example: multi-agent programming contest

The 2009 Edition tournament of our contest consisted of a series of simulations.[7] In each simulation two teams of agents competed in a grid-like world. There are virtual cowboys that can be controlled by agents. 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 grid itself is partially accessible: 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 push more cows into a specific corral than the opponent.

The simulation is discrete. In each step agents can perceive, have a fixed time to deliberate, and are then allowed to act. After a predefined number of steps the simulation is over. 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. Figure 3 shows a plot of a simulation-state. We use this scenario as a running example.

4.2 Interface intermediate language

An important design decision has been to define, as part of the environment interface, a convention for representing actions, percepts, and events. This convention is called the *interface intermediate language*, and supports the exchange of percepts and actions from and to environments. A conventional representation for actions, percepts, and events is required to be able to meet the second principle aiming at facilitating comparison of platforms and the fourth principle that aims at an 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

[7] We refer to the detailed paper in this special issue describing the *AgentContest*.

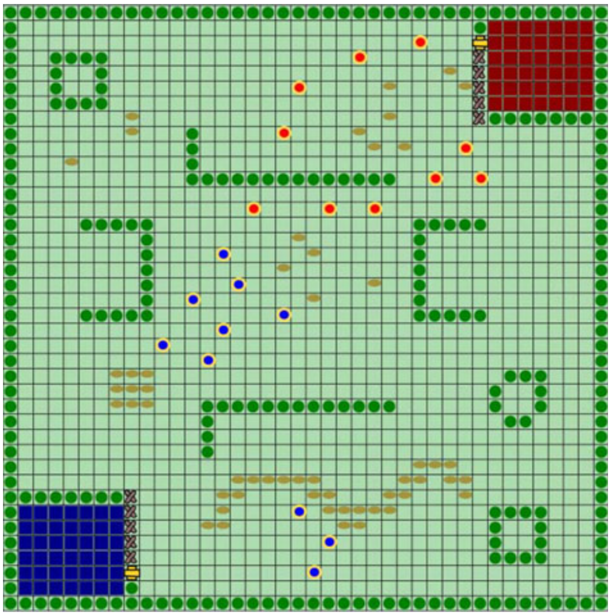


Fig. 3 Cowboys (*red* and *blue circles*) should scare cows (*brown ellipses*) into the corrals (*red* and *blue rectangles*)

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. Furthermore there 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 is an example[8] for a percept that informs an agent about the beginning of a simulation, including the position of the corral, the size of the grid, the visual range of the agents, the name of the opponent team and the number of steps of the simulation:

```
corral(0,0,20,20)grid(100,100)  id(1)
opponent(uglydozen)lineOfSight(8) steps(1400)
```

Here is an action that establishes a connection to a server at a given location, with a username and a password:

```
action(connect,agentcontest1,goodbadugly1,hh564kh)
```

[8]Note, however, that this is just an representation of elements of the interface intermediate language. Structurally each such element is an abstract syntax tree (which can easily be represented by a logic-programming token.

4.3 Functional point-of-view

What exactly 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.

Interactions are facilitated by function calls to the environment-interface, that can yield a return-value. For notifications we employ the *observer design pattern* (callback 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. 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 *AgentContest*).

Agents and entities The main principles assumed by the interface concerning the agent-entity relation are: (1) There is a set of agents on the agent 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. 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 and the environment-side.

Figure 4 shows the agents-entities-relation. The agents live on the agent-platform-side, they are known by the environment-interface by their identifiers. The entities

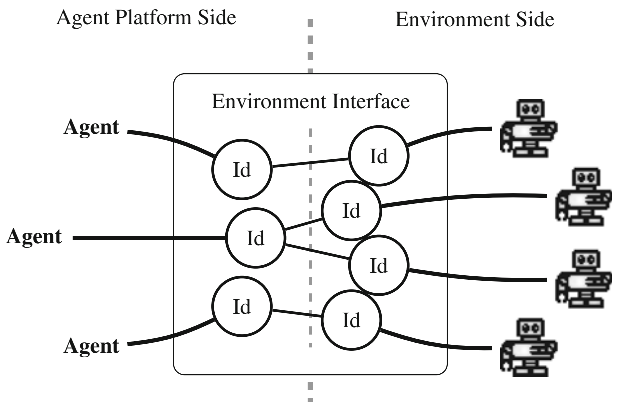


Fig. 4 The agents-entities-relation

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.

In the *AgentContest*, each cowboy is a controllable entity. Cows are entities as well but they are not controllable. Each agent can control only one single cowboy.

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, that is agent-events 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 are stored and managed.

Adding and removing entities Entities are added and removed in a similar fashion as 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 are notified via notifications about the respective events. This is done in order to allow components to react to the change of 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 effectoric and sensoric capabilities. The agents-entities-relation is manipulated by a set 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*, 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 decided to facilitate the two directions of flow by introducing two methods in order to have a unified approach.

There are two methods 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 location and with respective user-name and password, and eight for moving the cowboy. The method `getAllPercepts` retrieves the last percept sent by the server.

Managing the environment Although different environments provide different support to manage the initialisation, 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 functionality provided by the platform and the environment. This type of functionality is very useful and needed for debugging a MAS, and for inspecting an intermediate state of an agent-based simulation, for example.

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*, or *killing* the environment. Each command has an associated event. This is in particular useful, for example, if it is also possible that a user pauses an environment such as a game by means of the GUI of the game.

We propose as best practice that a corresponding event is returned each time an environment command is executed successfully. This concerns the implementation of the environment side of the interface; it facilitates graceful handling of environment management by an agent platform. The interface may be extended with additional commands and events that are not part of the convention yet, and provides support for implementing such additional commands and events.

It should be noted that there is no need to use and implement the environment management part of the interface for a particular environment. For example, in a testbed such as the *AgentContest* an agent platform cannot invoke a pause or reset command on the environment. Whenever an environment management command is invoked from an agent platform, as part of the environment management convention, we suggest that the interface returns a *not supported* event that is provided by the interface as one of the environment events.

We have chosen Java as an implementation-language for the interface because the platforms we have examined (that is 2APL, Goal, Jadex, *Jason*) are all Java-based. Note however, that the overall design is portable.

4.4 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 had 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 was not completely compatible with everything provided by the environment interface. It nevertheless proved very 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, but little effort was required to do so. We did have some issues with running a MAS that needs to be connected to an environment that only comes available after a user performs some initialization activities using a GUI. The philosophy of GOAL has been to collect all percepts (events) from an environment and process these just before selecting a next action. This approach does nicely match with the polling mechanism supported by the interface but does not match well with the event-based mechanism also supported by the interface. We have provided a simple mechanism to handle the percept notification mechanism, but a proper integration of this functionality will require additional work. Note that the main problem is that inconsistencies may easily arise. For example, when receiving a percept `on(a,b)` first and then `on(a,c)`, both via notifications, one cannot insert both into a belief base without introducing an inconsistency. Of course, providing a generic solution for this problem is not just an issue for GOAL but for all of the examined platforms.

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 glue 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 via fetching the corresponding space from their application context and use the EIS Java API directly for e.g., performing actions or collecting 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. With regard 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. With regard 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 observers mechanism available in EIS).

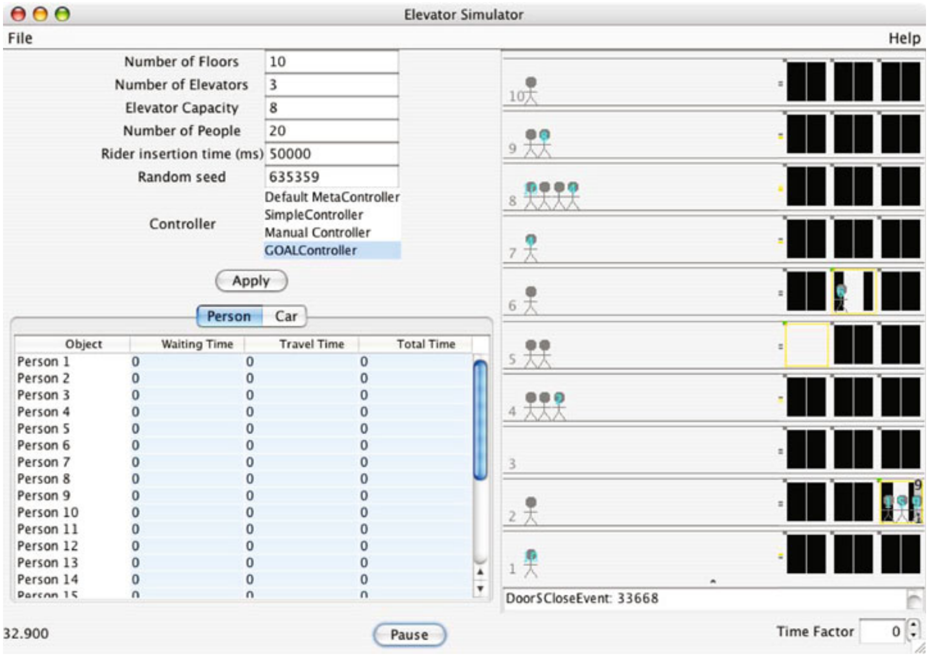
From all the concepts used in EIS, only that of "entities" is not supported by *Jason* as all actions and perceptions are relative to the overall environment and not to a particular entity. For percepts, the chosen solution was to add annotations to percepts that indicate the entity of origin. For actions, 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.

5 Case studies

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 discuss here three interface-enabled environments that can be used by any agent platform that supports the proposed environment interface.

5.1 Elevator environment

The elevator environment (see Fig. 5) is a good example of an environment that was not built specifically with agents in mind (it is available from [3]). 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.



Number of Floors	10
Number of Elevators	3
Elevator Capacity	8
Number of People	20
Rider insertion time (ms)	50000
Random seed	635359
Controller	Default MetaController SimpleController Manual Controller [✓ GOALController]

Object	Waiting Time	Travel Time	Total Time
Person 1	0	0	0
Person 2	0	0	0
Person 3	0	0	0
Person 4	0	0	0
Person 5	0	0	0
Person 6	0	0	0
Person 7	0	0	0
Person 8	0	0	0
Person 9	0	0	0
Person 10	0	0	0
Person 11	0	0	0
Person 12	0	0	0
Person 13	0	0	0
Person 14	0	0	0
Person 15	0	0	0

DoorSCloseEvent: 33668

32.900

Time Factor 0

Fig. 5 The elevator environment

The environment had been originally adapted for the Goal platform. The additional effort required to re-interface the environment to the environment interface was small. The main issue involved 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 these 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 2APL, Goal and *Jason*.

5.2 Agent contest environment

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. Furthermore, 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.

5.3 Unreal Tournament environment

EIS has also been used to implement an interface for connecting agents to the gaming environment Unreal Tournament 2004 (UT2004). We discuss this interface as a means for providing a high-level interface that supports relatively easy development of agent-controlled bots.

For this case study we have chosen to connect the agent programming language Goal to UT2004. UT2004 is a first-person shooter game that poses many challenges for human players as well as for computer-controlled players because of the fast pace of the game and because players only have incomplete information about the state of the game and other players. It provides a real-time, continuous, dynamic multi-agent environment and offers many challenges for developing agent-controlled bots. It thus is a suitable choice for putting an agent platform to the test. Kaminka et al. [12] argue that Unreal Tournament provides a useful testbed for the evaluation of agent technology and multi-agent research. Many modifications and additional maps are freely available for UT2004. It has, for example, also been used in competitions such as the RobocupRescue competition [2] which provides a high fidelity simulation of urban search and rescue robots using the Unreal engine. Using the Unreal Tournament game as a starting point to connect an agent platform to thus does not limit possibilities to one particular game but rather is a first step towards connecting an agent platform to a broad range of real-time environments.

We have used the behavioral control layer called *Pogamut* that extends *Gamebots* [8, 12] to bridge the gap that exists when trying to implement an interface oriented towards high-level cognitive control of a game such as UT2004.

One of the challenges of connecting BDI agents such as GOAL agents to a real-time environment is to provide a well-defined interface that is able to handle events produced by the environment, and that is able to provide sensory information to the agent and provides an interface to send action commands to the environment. Although Gamebots or Pogamut do provide such interfaces they do so at a relatively low level. The challenge here is to design an interface at the right abstraction level while providing the agent with enough detail to be able to "do the right thing". In other words, the "cognitive load" on the agent should not be too big for the agent to be able to efficiently handle sensory information and generate timely responses; it should, however, also be plausible and provide the agent with more or less the same information as a human player. Similarly, actions need to be designed such that the agent is able to control the bot by sending action commands that are not too finegrained but still allow the agent to control the bot in sufficient detail. Finally, the design of such an interface should also pay attention to technical desiderata such as that it provides support for debugging agent programs and facilitates easy connection of agents to bots. In the remainder of this section, we describe in more detail some of the design choices made and how EIS supported the development of the UT2004-GOAL interface.

The connection established using EIS between GOAL-agents, which are executed by the GOAL-interpreter, and UT2004 bots in the environment consists of several distinct components (see Fig. 6). The first component is GOAL's support for EIS. This component provides GOAL-specific support for EIS which will be implemented differently in different agent platforms. Basically this component in any agent platform needs to provide a sophisticated mechanism for launching multi-agent systems that instantiates agents (triggering the launch of agents e.g., based on events sent by EIS that indicate that a bot has become available in the environment); EIS then provides the mechanism for connecting the agent launched with the entity (bot). This component also needs to implement a mapping between, in this case, GOAL-percepts and -actions and those provided by EIS. The entities that agents are connected to, seen from the environment-interface-perspective, are instances of

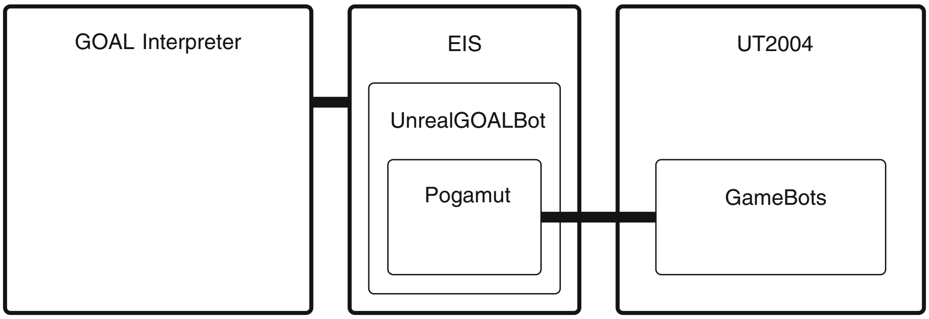


Fig. 6 A schematic overview of the implementation. The GOAL-interpeter connects to the EIS via Java-reflection. EIS wraps UnrealGOALBot, a heavy extension of Loquebot. UnrealGOALBot wraps Pogamut, which connects to GameBots via TCP/IP. GameBots is an Unreal-plugin

UnrealGOALBot, which is an extended modification of the *LoqueBot* developed by Juraj Simlovic. This LoqueBot again is built on top of Pogamut [8]. Pogamut itself is connected to *GameBots*, which is a plugin that opens UT2004 for connecting external controllers via TCP/IP. An entity in the terminology introduced here thus consists of three components: (1) An instance of UnrealGOALBot that allows access to UT, (2) a *action performer* which evaluates EIS-actions and executes them through the UnrealGOALBot, and (3) a *percept processor* that queries the memory of the UnrealGOALBot and yields EIS-percepts.

The instantiation of EIS for connecting Goal to UT2004 specializes percepts and distinguishes three classes of percepts. *Map-percepts* are sent only once to the agent and contain static information about the current map. This can be navigation-points (there is a graph overlaying the map topology), positions of all items (weapons, health, armor, power-ups et cetera), and information about the flags (the own and the one of the enemy). *See-percepts* on the other hand consist of what the bot currently sees. This can be visible items, flags, and other bots. *Self-percepts* consist of information about the bot itself. For example physical data (position, orientation and speed), status (health, armor, ammo and adrenaline), all carried weapons and the current weapon. Although these types of percepts are implemented specifically for UT2004, the general concepts of percepts that are provided only once, those provided whenever something changes in the visual field of the bot, and percepts that relate to status and can only have a single value at any time (e.g., current weapon) can be reapplied in other EIS instantiations. Here are some examples: `bot(bot1,red)` indicates the bot's name and its team, `currentWeapon(redeemer)` denotes that the current weapon is the Redeemer, `weapon(redeemer,1)`, indicates that the Redeemer has one piece of ammo left, and `pickup(inventoryspot56,weapon,redeemer)` denotes that a Redeemer can be picked up at the navigation-point `inventoryspot56`.

Actions are high-level to fit the BDI abstraction. The primitive behaviors that are used to implement these actions are based on primitive methods provided by the LoqueBot. Design-choices however were not that simple.

We have identified several layers of abstraction, ranging from (1) really low level interaction with the environment (the bot sees only neighboring waypoints and can use raytracing to find out details of the environment), over (2) making all waypoints available and allowing the bot to follow paths and avoid for example dodging attacks on its way, to (3) very high-level actions like *win the game*. The low level makes a very small reaction-time a requirement and is very easy to implement, whereas the high level allows for longer reaction times but requires more implementation effort.

We have identified the appropriate balance between reaction-time implementation effort to be an abstraction layer in which we provide these actions: `goto` navigates the bot to a specific navigation-point or item, `pursue` pursues a target, `halt` halts the bot, `setTarget` sets the target, `setWeapon` sets the current weapon, `setLookat` makes the bot look at a specific object, `dropweapon` drops the current weapon, `respawn` respawns the bot, `usepowerup` uses a power-up, `getgameinfo` gets the current score, the game-type and the identifier of the bot's team. Note that the first two actions take time to complete and are only initiated by sending the action command to UT2004. Durative actions such as goto and pursue may be interrupted. The agent needs to monitor the actions through percepts received to verify actions were successful.

EIS does support providing percepts as *return values* of actions but this requires blocking of the thread executing the action and we have chosen not to use this feature except if there is some useful *immediate* information to provide which does not require blocking. Special percepts were implemented to monitor the status of the goto action, including e.g., whether the bot is stuck or has reached the target destination. Moreover, the agent can control the route towards a target destination but may also delegate this to the behavioral control layer.

5.4 An example: the UNREAL-pill-collector

A GOAL-agent program consists of various components. The *belief base* is a set of beliefs, representing the current state of affairs. The *goal base* is a set of goals, representing in what state the agent wants to be. The *program section* is a set of action rules, that define a strategy or policy for action selection. The *action specification* is a specification of the conditions for each action available to the agent of when an action can be performed (precondition) and the effects of performing an action (postcondition). Finally, a set of *the percept rules* specify how percepts received from the environment modify the agent's mental state.

Figure 7 shows the agent-code of a simple GOAL-agent that performs two tasks: (1) Collecting pills and (2) setting a target for attack. The agent relies on dynamic

```
main: unrealCollector {
  beliefs{
    targets([]).
    moving(triple(0,0,0), triple(0,0,0), triple(0,0,0), stuck).
  }
  goals{
    collect. targets([all]).
  }
  program{

    if goal(collect), bel(pickup(UnrealLocID,special,Type))

       then goto([UnrealLocID]).

    if bel(targets([])) then setTarget([all]).
  }
  actionspec{
    goto(Args) {

      pre { moving(Pos, Rot, Vel, reached) }
      post { not(moving(Pos, Rot, Vel, reached)) }
    }
    setTarget(Targets) {
      pre { targets(OldTargets) }
      post { not(targets(OldTargets)), targets(Targets) }
    }
  }
  perceptrules{

    if bel( percept(pickup(X,Y,Z)) ) then insert(pickup(X,Y,Z)).

    if bel(percept(moving(Pos, Rot, Vel, State)), moving(P, R, V, S))
      then insert(moving(Pos, Rot, Vel, State)) + delete(moving(P, R, V, S)).
  }
}
```

Fig. 7 A very simple UNREAL-GOAL-agent collecting pills and setting targets

beliefs provided by the environment. The initial beliefs state that the agent has no target and also states the physical-state of the bot, that is the position, the rotation, the velocity and the state (the state could be `stuck`, `moving`, and `reached`).

The agent's goal base contains the single goal of collecting special items. The first rule in the program specification makes the bot go to the specific location of a special-item if the agent knows its position and has the goal of collecting those. The second one sets the targets from none to all bots.

The `goto` action in the actionspec-section allows the bot to move in the environment. The `setTarget` action sets the enemy bots that will be targeted if visible. These actions are quite different. The `goto` action takes more or less time to complete depending on the distance to be traveled. The `setTarget` action in contrast is executed instantaneously as it only changes a mode of operation (a parameter). This difference has important consequences related to specifying the pre- and postcondition of these actions. Whereas it is quite easy to specify the pre- and postcondition of the `setTarget action`, this is not the case for the `goto` action. As goto is a durative action that may fail (if only because an enemy bot may kill the bot) it is not possible to specify the postcondition uniquely. Moreover, some of the details of going somewhere as, for example, the exact route taken may (but need not be) delegated to the behavioral layer; this means that most of the time only through percepts the exact route can be traced. Therefore, it makes more sense in a dynamic environment that an agent relies on percepts that are made available by the environment to inform it about its state than on the specification of a postcondition. For this reason, when an action is selected, Goal does not block on this action until it completes. Instead, upon selection of an action, Goal sends the action command to the environment and then simply continues executing its reasoning cycle; this design explicitly allows for monitoring the results of executing the action command while it is being performed by the bot in the environment. For some actions, among which the goto action, the interface has been designed such that specific monitoring percepts are provided related to events that are relevant at the cognitive level. The moving state percepts stuck, moving, and reached are examples that illustrate how an agent may conclude the goto action has failed, is ongoing, or has been successful. The setup of sending an action command to the environment while continuing the agent's reasoning cycle also allows for interrupting the action if somehow that seems more opportune to the agent; it can simply select a `goto` action with another target to do so.

Though this agent is simple, the example shows that it is relatively simple to write an agent program using the interface that does something useful like collecting pills. Information needed to control the bot at the knowledge level is provided at start-up (e.g., where are pickup locations on the map). The code also illustrates that some of the tasks may be delegated to the behavioral layer. For example, the agent does not compute a route itself but delegates determining a route to pickup navigation point.

Here is an example to illustrate the coordination between the agent and the bot routines at lower levels. It concerns the precondition of the goto action. By defining the precondition as in Fig. 7 (which is a design choice not enforced by the interface), this action will only be selected if a previously initiated goto behavior has been completed, indicated by the `reached` constant.

In retrospective, we have faced several implementation challenges when connecting to UT2004 using EIS. EIS facilitated the design of a clean and well-defined separation of the agent (programmed in GOAL) and the behavioral layer (the UnrealGOALBot) to the Unreal-AI-engine. The strict separation of EIS between agents as percept-processors and action-generators and entities as sensor- and effector-providers made facilitated the design. We also had in mind right from the beginning that we wanted to use the UT2004-interface in order to provide means for comparing APL platforms in general. Since support for EIS is easily established on other platforms we have solved this problem as well, by making the interface EIS-compliant.

In summary, we have illustrated that EIS is able to support durative or hight-level actions, like in UT2004, as well as instant actions or low-level actions, like in the agent-contest. In the UT2004-scenario the specific EIS-interface connects to a high-level, behavioral layer on top of the game, that controls the entities, whereas the specific EIS-interface for the *AgentContest* allows for a direct control of the entities.

5.5 Evaluation summary

The relative ease with which the interface has been connected to four established 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 events, and actions (is the platform more logic-oriented or Java-based) and how environments before using the interface were connected to these platforms. The environment interface nevertheless could be connected to each of the platforms easily, thus providing evidence of its generality and comprehensiveness as well. Of course, we need and we have invited more agent platforms to use the environment interface, but we do not expect this will pose any fundamentally new issues. Initial experience with various environments has 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.

While we have discussed above the connection of *all three* agent platforms to an environment in just one case study, we are currently working on doing the same for the other case studies as well. However, in all case studies we connected at least one new platform: As this turned out to be so easy, we are convinced that connecting all the other platforms as well is straightforward. In the meantime, there has been also a rudimentary EIS-support established for the JIAC platform (conversation with Axel Heßler), which has shown again the ease of connecting to EIS.

6 Conclusion

In this paper we have developed, to the best of our knowledge for the first time, an environment interface standard (EIS).[9] This standard facilitates connecting agents programmed in various agent programming languages (APL) to arbitrary environments. The standard is based on a set of six principles. We have shown how several

[9]The software is available for download at http://sf.net/projects/apleis, where you will also find a tutorial on how to connect your platform or your environment to EIS.

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.

We believe that the agent community can greatly benefit from the standard proposed in this paper. As discussed, implementations of the interface for various environments make these environments available for all platforms that comply with the standard. By adopting the interface proposal future environments (implemented using this interface) can be used by any platform that supports the interface without any additional effort; there is no need to modify the functionality provided by an environment anymore. The interface proposal discussed here may also contribute to clarifying the connection that needs to be established between agents and environments.

6.1 Heterogenity

We believe that an interface standard also allows developing truly heterogenous multi-agent systems where agents developed using different agent platforms that comply to the standard may interact in one and the same environment. That is, agents that run on different agents platforms may connect to the same environment *simultaneously* (see Fig. 8) where the environment also provides the means for *agent-coordination* (i.e., communication). Although more research is needed, we sketch some ideas for future work that may achieve this goal.

The interface developed for the *MASSim*-server that implements the *Agent-Contest* already established a limited form of heterogeneity: It facilitates the

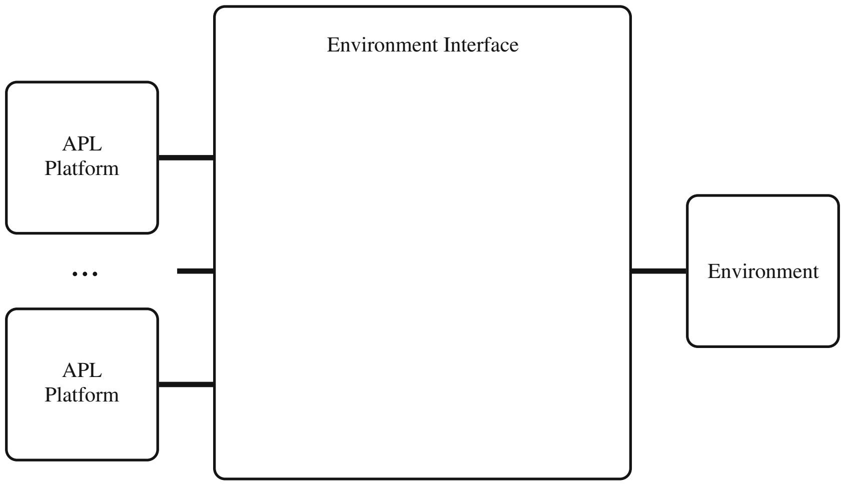


Fig. 8 Our goal: agents from several platforms connected

connection of agents that run on several different agent platforms to a single environment. The key element that is not yet available in this environment is support for coordination or agent communication. One solution to this problem may be to provide a special communication-action that is provided by the environment. On the other hand, we believe that the *MASSim*-server should not be used to help establishing true heterogeneity in general, i.e., using the *MASSim*-server to implement new or connect to already existing environments. The reason lies in the fact that the *MASSim*-server is specialized. It has been developed in order to allow competition between agent teams. Such provided functionality is not desired for general interfaces to environments, and thus renders the option of establishing heterogeneity via *MASSim* obsolete.

True heterogeneity should be established with functionality provided by an extended version of the EIS. The idea is to do this through an EIS-client-server architecture that is basically equivalent to the EIS as discussed in this paper. As a design-principle, the interface discussed here should not be fundamentally changed. Otherwise we would loose support for the easy transition from already existing interfaces to heterogeneous ones, and for keeping the already existing connections between agent platforms and the interfaces.

In accordance with our principles, Fig. 9 shows our proposed client-server-architecture. A heterogeneous environment-interface consists of one server that directly connects to the environment, and of one or several clients that connect to the agent platforms. The connection between each client and the server also separates the multiple processes.

It remains to discuss the client-server-connection. Interacting with the EIS is by calling methods and using-callbacks for exchanging data. Thus it is straightforward to employ Java-RMI to create the described client-server-architecture. Java-RMI is an application programming interface that allows calling methods on remote

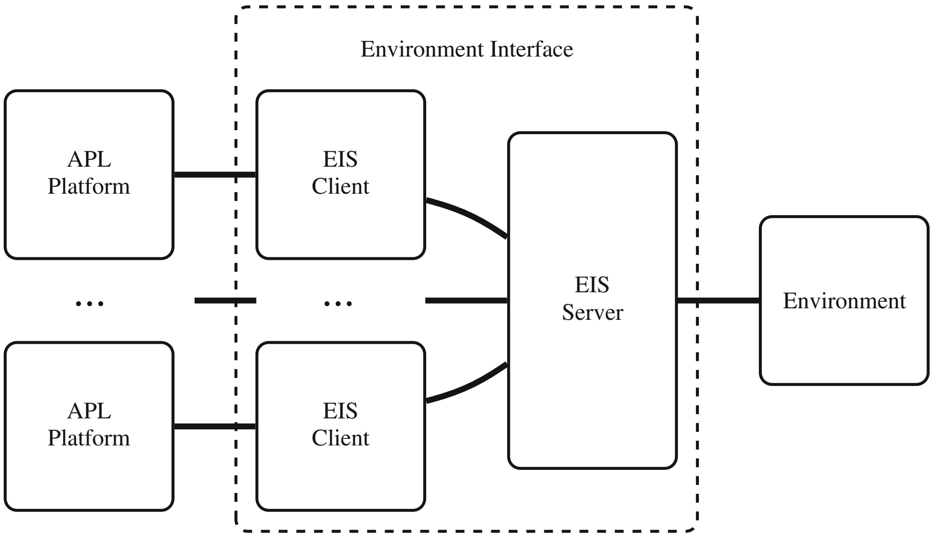


Fig. 9 Distributed EIS with several processes

applications. Mapping the already defined EIS-methods to RMI-supported ones would be the easiest way to establish the client-server-connections.

The difference between homogeneous and heterogeneous interfaces should only be noticed when implementing the respective Java-interface and nowhere else. The differences between implementing a homogeneous and a heterogeneous interface should be minimal.

Finally, to support true heterogeneity, it is essential to provide a means for agent-coordination. CArtAgO is a means to implement environments with arbitrary means for agent-coordination (called artifacts) including inter-agent communication. This has to be studied in the future.

Acknowledgements We thank Mehdi Dastani, Rafael Bordini, Lars Braubach, Jomi Hübner, and Alexander Pokahr for their contribution to our effort and their effort in implementing the interface for the agent programming platforms 2APL, *Jason*, and Jadex. We also like to thank Alessandro Ricci for useful discussions that helped us clarify some of the ideas presented in this paper. We also want to acknowledge the help of Wouter Pasman for implementing the EIS interface for Goal, and contributing to the evaluation of the interface.

We also thank three anonymous referees for their comments: they helped to improve the paper a lot.

The work of Tristan Behrens and Jürgen Dix was partly funded by the NTH School for IT Ecosystems. NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover.

References

1. JACK. Agent oriented software group. http://www.aosgrp.com/products/jack. Accessed 30 Jan 2010
2. RobocupRescue. http://www.robocuprescue.org. Accessed 30 Jan 2010
3. Elevator simulator homepage. http://sourceforge.net/projects/elevatorsim/
4. Best, B.J., Lebiere, C.: Teamwork, communication, and planning in ACT-R. In: Proc. of the IJCAI Workshop on Cognitive Modeling of Agents and Multi-Agent Interactions, pp. 64–72 (2003)
5. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology). Wiley, New York (2007)
6. Braubach, L., Pokahr, A., Lamersdorf, W.: Jadex: a BDI-agent system combining middleware and reasoning. In: Unland, V.R., Klusch, M., Calisti, M. (eds.) Software Agent-Based Applications, Platforms and Development Kits (2005)
7. Brom, C., Gemrot, J., Bida, M., Burkert, O., Partington, S.J., Bryson, J.: POSH tools for game agent development by students and non-programmers. In: Proc. of the 9th Computer Games Conference (CGAMES'06), pp. 126–133 (2006)
8. Burkert, O., Kadlec, R., Gemrot, J., Bída, M., Havlíèek, J., Dörfler, M., Brom, C.: Towards fast prototyping of IVAs behavior: Pogamut 2. In: Proceedings of the Seventh International Conference on Intelligent Virtual Humans (IVA'07) (2007)
9. Dastani, M., Dix, J., Novák, P.: Agent contest competition, 3rd edn. In: Dastani, M., Ricci, A., El Fallah Seghrouchni, A., Winikoff, M. (eds.) Proceedings of ProMAS '07. Revised Selected and Invited Papers. Lecture Notes in Artificial Intelligence, no. 4908. Springer, Honululu (2008)
10. Dastani, M., et al.: 2APL Manual. http://www.cs.uu.nl/2apl/
11. Jacobs, S., Ferrein, A., Ferrein, E., Lakemeyer, G.: Unreal GOLOG Bots. In: Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games, pp. 31–36 (2005)
12. Kaminka, G., Veloso, M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A., Scholer, A., Tejada, S.: Gamebots: a flexible test bed for multiagent team research. Commun. ACM **45**(1), 43–45 (2002)

13. Kim, I.C.: UTBot: A virtual agent platform for teaching agent system design. Journal of Multimedia **2**(1), 48–53 (2007)
14. Köster, M., Novák, P., Mainzer, D., Fuhrmann, B.: Two case studies for jazzyk bsm. In: Dignum, F., Bradshaw, J.M., Silverman, B.G., van Doesburg, W.A. (eds.) AGS. Lecture Notes in Computer Science, vol. 5920, pp. 33–47. Springer (2009)
15. Laird, J.E.: Using a computer game to develop advanced AI. Computer **34**(7), 70–75 (2001)
16. Laird, J.E.: Extending the soar cognitive architecture. In: Wang, P., Goertzel, B., Franklin, S. (eds.) AGI. Frontiers in Artificial Intelligence and Applications, vol. 171, pp. 224–235. IOS Press (2008)
17. Laird, J.E., Assanie, M., Bachelor, B., Benninghoff, N., Enam, S., Jones, B., Kerfoot, A., Lauver, C., Magerko, B., Sheiman, J., Stokes, D., Wallace, S.: A test bed for developing intelligent synthetic characters. In: Spring Symposium on Artificial Intelligence and Interactive Entertainment (AAAI'02) (2002)
18. Laird, J.E., Newell, A., Rosenbloom, P.: Soar: an architecture for general intelligence. Artif. Intell. **33**(1), 1–64 (1987)
19. Magerko, B., Laird, J.E., Assanie, M., Kerfoot, A., Stokes, D.: AI characters and directors for interactive computer games. In: McGuinness, D.L., Ferguson, G. (eds.) AAAI, pp. 877–883. AAAI Press/The MIT Press (2004)
20. Murphy, R.R.: Introduction to AI Robotics. MIT Press, Cambridge (2000)
21. Newell, A.: The knowledge level. Artif. Intell. **18**(1), 87–127 (1982)
22. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems **17**(3), 432–456 (2008)
23. Partington, S.J., Bryson, J.J.: The behavior oriented design of an unreal tournament character. In: Panayiotopoulos, T., Gratch, J., Aylett, R., Ballin, D., Olivier, P., Rist, T. (eds.) Intelligent Virtual Agents (IVA'05), pp. 466–477 (2005)
24. Pasman, W.: GOAL IDE User Manual. http://mmi.tudelft.nl/~koen/goal.php
25. Ricci, A., Piunti, M., Acay, L.D., Bordini, R., Hübner, J., Dastani, M.: Integrating artifact-based environments with heterogeneous agent-programming platforms. In: 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-08), pp. 225–232. IFAAMAS (2008)
26. Ricci, A., Viroli, M., Omicini, A.: CArtAgO: a framework for prototyping artifact-based environments in MAS. In: Weyns, D., Parunak, H.V.D., Michel, F. (eds.) Environments for MultiAgent Systems III, pp. 67–86. Springer (2007). doi:10.1007/978-3-540-71103-2_4
27. Russell, S.J., Norvig: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall (2003)
28. (SISC), S.I.S.C.: IEEE Standard for Modeling and Simulation (M & S) High Level Architecture (HLA)—Framework and Rules (2000). doi:10.1109/IEEESTD.2000.92296
29. Tweedale, J., Ichalkaranje, N., Sioutis, C., Jarvis, B., Consoli, A., Phillips-Wren, G.: Innovations in multi-agent systems. J. Netw. Comput. Appl. **30**(3), 1089–1115 (2007)
30. Wang, D., Subagdja, B., Tan, A.H., Ng, G.W.: Creating human-like autonomous players in real-time first person shooter computer games. In: Proc. of the 21st Conference on Innovative Applications of Artificial Intelligence (IAAI'09) (2009)