# Distributed Agent Environments in the Ambient Event Calculus

Stefano Bromuri and Kostas Stathis
Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey, TW20 0EX, UK
{stefano,kostas}@cs.rhul.ac.uk

## ABSTRACT

We study the development of distributed agent environments as distributed event-based systems specified in the *Ambient Event Calculus* (AEC). The AEC is a logic-based formalism that is developed here to support the representation of a distributed agent environment as a persistent composite structure evolving over time. Such a complex structure supports the interaction between *agents*, *objects*, and *containers*, entities that have their own external observable state and can be distributed over a network. Interactions between these entities are specified in terms of *events* that represent actions executed by agents on objects and other agents in the environment. When events happen they are stored in containers and are notified to agent sensors that subscribe to event descriptions and as a result perceive the interactions. The AEC formalism also allows changes caused by events to be delivered across distributed containers, according to the topology of the application environment. We illustrate the use of AEC and we show how to specify interactions within the GOLEM agent platform applied to a specific agent scenario.

## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming:—*Distributed Programming*; I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence:—*Multiagent Systems*

## General Terms

Research Paper

## Keywords

MAS, Distributed agent environments, Event Calculus

## 1. INTRODUCTION

There is an increasing demand for complex distributed systems that are capable of operating in open and dynamic environments, that are composed of loosely-coupled interacting components with autonomous behaviour, that are flexible, adaptive, and scalable to support what people do at work, home, or while on the go. There are many applications that can be thought of as distributed systems of this kind, consider for example autonomic workflow systems deployed in business organisations, computer games entertaining millions of people at home, and ubiquitous and ambient intelligence applications that allow people to interact and communicate efficiently and effectively wherever they are.

In order to meet the challenges of complex distributed systems, the notion of *agency* has often been proposed as a development metaphor for distributed systems applications. A software agent is a computer system that is capable of flexible autonomous action in dynamic and, possibly, unpredictable or open environments [30]. The characteristics of dynamic and open environments suggest that improvements on traditional computing models and paradigms are required. Thus, the need for some degree of autonomy, to enable agents to respond dynamically to changing circumstances while trying to achieve their goals, is seen as fundamental. Many observers therefore believe that agents represent the most important new paradigm for software development since object orientation [17].

Typically, software agents are not monolithic systems, but they interact and collaborate with other agents to support a specific application. A multi-agent system (MAS) is a system composed of multiple interacting agents that can be used to solve problems which are difficult or impossible for an individual agent to solve. To support the deployment of a MAS for practical applications, early agent platforms such as JADE [2] and JACK [12], became available to support the development activity and provide the necessary middleware. These platforms supported the engineering of MAS applications by enabling agents to interact with each other in their environment by sending and receiving messages [13, 20]. However, attempts to model the agent environment as a message transport system (or broker infrastructure) have been criticized to be inadequate for complex applications [29]; these criticisms further suggest that the agent environment should be a first-class entity [21, 28].

### 1.1 Motivation

We are concerned with specifying and implementing agent environments for software agents with cognitive capabilities [14], with the aim to make agents more intelligent through the use of logical reasoning. In many of these environments [23, 4], an application does not only use agents but


*DEBS'09,* July 6–9, Nashville, TN, USA.

also additional entities where agents are contained in or interact with. We are particularly interested in agent environments that can be distributed over a network. For such environments it makes sense that the evolution of the state of the environment is modelled via occurrences of events that capture the interactions of agents at the application level [4]. It is natural then to consider Distributed Event-Based Systems (DEBS) as the underlying implementation model for our study [3]. More specifically, our main concerns are:

**Event Declaration:** how are events declared in the agent environment? What is their structure and how can the agent environment deal with different types of events?

**Event Notification:** how is the occurrence of events notified to the entities situated in the environment? Should entities in the agent environment receive an event that happened in the past or should an event be notified immediately as it happens? How does the topology of the distributed agent environment affect the notification mechanism?

**Dynamic Event Binding:** how can an agent dynamically subscribe to observe different types of events?

**Full Delivery:** how can we notify all the components that are interested in an event within an agent environment?

**Event subscription:** how can entities in the environment subscribe for events produced within a portion of the agent environment?

**Event filtering:** how can entities filter the information that they receive from the agent environment?

**Delivery semantics:** how are events propagated in a distributed agent environment?

**Event Persistence:** how and where are the event stored within the distributed agent environment?

Taking these issues as a starting point, this work studies the modelling of a distributed agent environment implemented as a DEBS, to support the implementation of agent environments as first-class complex objects.

## 1.2 Contribution

The contribution of this work is that it develops a new formalism, that we call Ambient Event Calculus (AEC), to deal with events in a distributed agent environment seen as a distributed event-based system. We use the AEC to show how we can deal with the subscription, notification and production of events happening within the context of a complex topology. More specifically, we illustrate how agents and objects populating a distributed agent environment can be seen as publishers and subscribers to events, which the agents can perceive using their sensors and effectors, and the object can process using their triggers and emitters. We exemplify the discussion by re-specifying the interactions of an existing agent environment implemented in the GOLEM platform [4]. The significance of the new version of the GOLEM platform is that we apply the concept of distributed event systems to deal with distributed event notification and perception, allowing us to model the evolution of the distributed agent environment as a first-class abstraction, and thus showing how to apply DEBS techniques to build MASs.

## 1.3 Organisation

The reminder of this paper is organised as follows. Section 2 presents a background on the GOLEM agent infrastructure, introducing its main components, and a scenario that is already presented in [4] but extended here to illustrate how an application can be distributed over a computer network. Section 3 introduces the Ambient Event Calculus, which is the logic formalism on which the new version of GOLEM will be based upon, including example interactions from the application scenario. The implementation of the framework is discussed in Section 4. After Section 5 which evaluates the AEC performance in a distributed setting, Section 6 discusses the related work. We conclude in Section 7 where we also outline our plans for future work.

# 2. THE GOLEM AGENT ENVIRONMENT

We provide here the background of an existing agent environment whose main components have been reported in [24], the specification of the implemented platform has been described in [4], and its use to develop a distributed systems application in [5]. To provide continuity with this existing work we also present an extended version of a scenario discussed in [4] in order to show how to distribute an application in practice. Our aim is then to use the scenario to exemplify our discussion for the rest of this paper.

## 2.1 Background on GOLEM

The GOLEM agent platform [24, 4] allows the deployment of three main entities: *agents*, *objects* and *containers*. Agents are active and cognitive entities that can interact with other agents and objects in the agent environment. Agents are composed by a declarative mind, a component that supports the reasoning abilities of the agent such as planning, decision making, and temporal reasoning. The mind is situated in the agent environment via another component that is called the agent body. This component contains sensors for the agent to be able to perceive the environment and effectors to be able to affect the environment. In other words, sensors and effectors represent the interface between the agent environment and the agent mind.

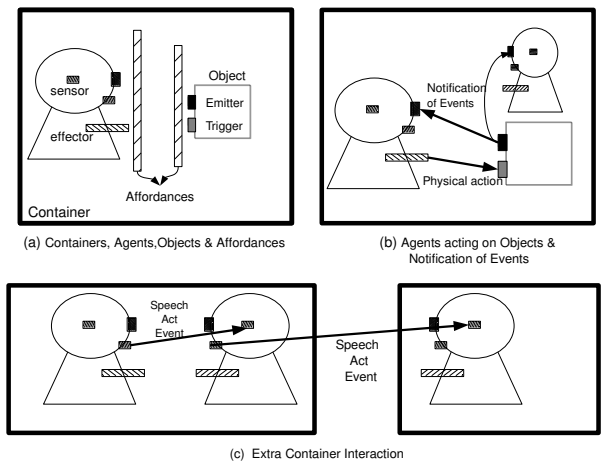



**Figure 1: Agent Architecture in GOLEM**

Unlike agents, objects are reactive entities. Objects have a

trigger to receive events from the environment and an emitter to produce reactions to such events. The trigger and the emitter ensure that the interaction between the object and the environment is completely asynchronous. An object is composed by an external object which is connected to the trigger and the emitter, and its purpose is to hide the complexity of an internal object, which could represent an external resource. The general idea behind the internal object is that it wraps in it a resource of the external environment, thus hiding from the agents the complexity of interfacing with the external resource. In other words the object abstraction can be a virtual entity or a virtualisation of an external resource of the external real environment, for example, a printer, a web service, or a database.

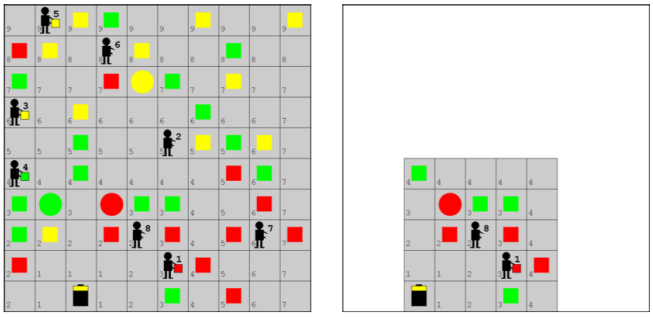
Central to the GOLEM agent platform is the concept of affordances: normally this concept is taken to describe "all the action possibilities latent in the environment, objectively measurable, and independent of an agent's ability to recognise those possibilities" [11]. As with research in HCI [19], GOLEM relies upon *perceived affordances* where entities of an environment "suggest" to agents (whether artificial or human) how they should interact with them. In other words the concept affordances in GOLEM represent the external observable state of an agent or of an object as shown in Fig. 1(a).

Agents and objects are situated within *containers*. A container represents a portion of the distributed agent environment and it works as a mediator for the interaction taking place between agents and objects. Events describe what happens in the agent environment as a result of actions being executed by effectors. According to the happening of an event the agent environment notifies those sensors capable of perceiving the action of the event. In GOLEM three types of acts are embedded in an event: *speech acts* - to allow agents to communicate with other agents and users; *sensing acts* - to allow an agent to perceive the environment actively; and *physical acts* - to allow the agent to interact with other entities, in particular objects, but also agents as well. To simulate these acts GOLEM relies upon different kinds of sensors and effectors the agent should possess in the agent environment. Fig. 1(b) shows how the result of a physical act of an agent on an object is perceived not only by the agent itself but also by another agent, using the affordances of these entities within the container in which they are situated.

In the first version of the platform, the only interaction that was possible to perform from one container to another was communication between agents. In the scope of this paper we want to extend the model to have agents that can interact across containers and assuming a distributed topology of containers. Figure 1(c) illustrates the kind of extra-container interaction that we wish to provide for the GOLEM agent environment using a DEBS model.

## 2.2 An Example Agent Environment

In order to illustrate the specific features of the DEBS model on a MAS application, we use as a motivating example the Packet-World scenario [27]. As shown in Fig. 2, the basic setup of the Packet-World consists of a number of differently coloured packets inside a rectangular grid, whose destination is a circle with the same colour. Each agent living in the Packet-World has a battery that discharges as the agent moves in different locations in the grid. The battery can be recharged using a battery charger. This charger emits a gradient whose value is larger if the agent is far away from the charger and smaller if the agent is closer to the charger. To locate the battery charger an agent must follow the direction of decreasing gradient values. The agents have the goal to bring the packets to the collection points and can communicate with other agents to create collaborations or to ask information about the position of the collection points.



(a) A Packet-World of 10x10. (b) Local view of agent 8.

**Figure 2: The Packet-World.**

As shown in Fig. 3, here we consider a version of the Packet-World, where areas of the grid are distributed to multiple containers that run on different hosts.

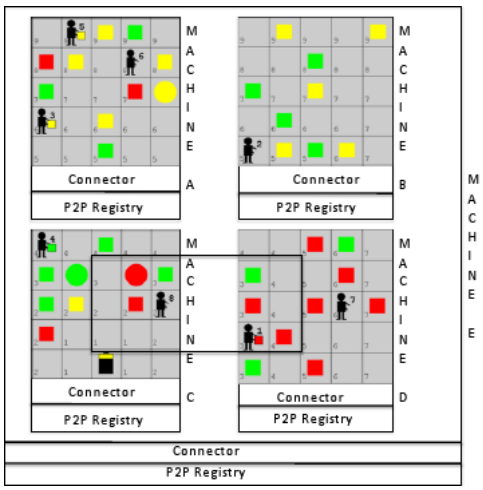



**Figure 3: Distributed Packet-World in GOLEM**

Every different host is responsible for the agents and the packets deployed within it. In such a setting one requirement is that the agent can perceive what happens in a location that is logically nearby, but distributed in another host. Another requirement is the definition of the propagation of the events and how the agents can interact with packets or destinations that are logically near, but physically distributed elsewhere.

# 3. THE AMBIENT EVENT CALCULUS

The Ambient Event Calculus (AEC) is a formalism that allows the specification of complex events happening in the

complex and evolving state of a distributed agent environment. The calculus supports *intra-container* interactions, i.e. interactions within a container, and provides mechanisms for subscription and notification of events in this context. It also supports the specification of *extra-container* interactions, that is interactions across containers, and provides mechanisms for subscriptions and notifications of distributed containers. Thus, in AEC the agent environment is specified in four main parts: intra-container persistence and evolution; intra-container action execution and perception; extra-container persistence and evolution; extra-container action execution and perception. In what follows, we are presenting examples of our formulation, using the GOLEM platform and the Packets-World scenario.

## 3.1 Evolution of the Agent Environment

To represent the state of a container and the entities it contains we use the object-based notation used by C-logic [7], a formalism that allows the description of complex objects. By state we mean the complex data structures required to represent the perceived affordances of the entities within a container, including the container itself. For example the description:

```
container:c1[address ⇒ "container://one@134.219.7.1:13000",
             laws ⇒ physics:pw1,
             type ⇒ open,
             entities ⇒ {picker:ag1,
                         packet:p1,
                         packet:p2,
                         destination:d1,
                         battery:b1}
            ]
```

describes a container whose affordances include an address attribute whose value is container://one@134.219.7.1:13000, a laws attribute pointing another object pw1 of class physics, the type attribute has the value open, meaning that any agent can enter the container, and the entities contained in the container is a set (multiple values attribute) of one picker agent (ag1), two packets (p1, p2), a destination for packets (d1), and a battery (b1). Entities within a container can have their own perceived affordances, for example, the perceived affordances of the picker ag1 are shown below:

```
picker: ag1[ understands ⇒ ontology:o1,
             sensors ⇒ {sight:s1, hearing:s2, smell:s3},
             effectors ⇒ {speak:ef1, arm:ef2, arm:ef3},
             position ⇒ square:sq3,
             activity ⇒ idle
           ]
```

The advantage of this representation is that agents can perceive directly these affordances and reason about them, as these descriptions have a translation to first-order logic, see [7] for the details.

To represent how phenomena change the state of a GOLEM container we use the object-based event calculus (OEC) described by Kesim and Sergot in [15]. The OEC assumes an object-based data model and describes how instances of complex terms, such as containers, evolve over time. A subset of the clauses describing the OEC is given below:

```
(C1) holds_at(Id, Class, Attr, Val, T)←
        happens(E, Ti), Ti ≤ T,
        initiates(E, Id, Class, Attr, Val),
        not broken(Id, Class, Attr, Val, Ti, T).

(C2) broken(Id, Class, Attr, Val, Ti, Tn)←
        happens(E, Tj), Ti < Tj ≤Tn,
        terminates(E, Id, Class, Attr, Val).

(C3) holds_at(Id, Class, Attr, Val, T)←
        method(Class, Id, Attr, Val, Body),
        solve_at(Body, T).

(C4) attribute_of(Class, X, Type)←
        attribute(Class, X, Type).

(C5) attribute_of(Sub, X, Type)←
        is_a(Sub, Class),

        attribute_of(Class, X, Type).

(C6) instance_of(Id, Class, T)←
        happens(E, Ti), Ti ≤ T,
        assigns(E, Id, Class),
        not removed(Id, Class, Ti, T).

(C7) removed(Id, Class, Ti, Tn)←
        happens(E, Tj), Ti < Tj ≤ Tn,
        destroys(E, Id).

(C8) assigns(E, Id, Class)←
        is_a(Sub, Class),
        assigns(E, Id, Sub).

(C9) terminates(E, Id, Class, Attr, _)←
        attribute_of(Class, Attr, single),
        initiates(E, Id, Class, Attr, _).

(C10) terminates(E, Id, _, Attr, _)←
        destroys(E, Id).

(C11) terminates(E, Id, _, Attr, IdVal)←
        destroys(E, IdVal).
```

Clauses C1-C2 provide the basic formulation of OEC deriving how the value of an attribute for a complex term holds at a specific time. Clause C3 describes how to represent derived attributes of objects treated as method calls computed by means of a solve_at/2 meta-interpreter as specified in [16]. C4-C5 support a monotonic inheritance of attributes names for a class limited to the subset relation. As C1-C2 describe what holds at a specific time, C6-C7 determine how to derive the instance of a class at a specific time. The effects of an event on a class is given by assignment assertions; the clause C8 states how any new instance of a class becomes a new instance of the super-classes. Finally, deletion of objects is catered for by clauses C9-C11. C9 deletes single valued attributes that have been updated, while C10-C11 delete objects and dangling references.

Event descriptions themselves are specified as complex terms. For example, the event description below:

```
do:e14 [actor ⇒ ag1, act ⇒ move:m1 [destination⇒ sq3]].
```

represents a physical action of agent ag1 who tries to move from one location to another. We will see later, how such an action is executed by the agent that causes the event to happen. For the time being, we will assume that the event has happened and we will show next how the affordances in the agent environment will evolve as a result of the happening of this event. To do this we need to define domain specific initiates and terminates clauses, as shown below:

```
initiates(E, picker, A, position, Pos) ←
        do:E [actor ⇒ A, act ⇒ move:M [destination⇒ Pos]].
```

In this way the new position of the agent has been initiated as a result of the move. The complete description of the event's effects also requires to terminate the attribute holding the old position of the agent; this is handled in OEC by the general rule described in clause C9.

## 3.2 Intra-container action and perception

In an agent environment the producers of events are agent effectors and object emitters, while consumer of events are agent sensors and object triggers. The issue then becomes how sensors and triggers subscribe to events, how events are notified to sensors once they happen, and how the notifications trigger objects make agents to perceive these events.

### 3.2.1 Event Subscription

To show how sensors subscribe to specific events in the AEC we need to understand how affordances of effectors and sensors are specified in the environment. To illustrate event subscription, consider the definition of the hand effector for a picker agent in Packet-World. This is described using the following C-Logic term:

```
hand:ef1[
        abilities ⇒ {pick, drop, throw}
        status ⇒ free
       ]
```

While the status of the hand is free, the agent body that is equipped with this effector can produce three kind of events: pick, drop, and throw. Events produced by effectors must be consumed by sensors. We link sensors and effectors at the level of sensor affordances (if it is an agent) and trigger affordances (if it is an object). For example, to express that an agent sensor listens to every kind of speech act we define listener sensors as:

```
listener:s1[
          senses ⇒ speech_acts,
          status ⇒ open
         ]
```

In order to specify which events a sensor can perceive, we need to allow agents (or objects) to change the affordances of their sensors (or triggers) by opening and closing the sensor to the event notification. To deal with the way sensors subscribe/unsubscribe to events we use subscribes/3 predicates written as:

```
(C12) subscribes(E, S, T)←
          holds_at(S, sensor, senses, ActClass, T),
          ActSubClass: E,
          is_a(ActSubClass, ActClass),
          holds_at(S, sensor, status, open, T).
```

The above definition specifies that a sensor S subscribes to an event E if the class of act in the event is a subclass of the actions that can be sensed by the sensor and that the status of the sensor is open. Otherwise, if the sensor is not anymore interested in events of a certain class, the agent must close the sensor and the sensor will not subscribe to events for the time it is closed (non-subscription can then be deduced from subscribes/3 definitions and the use of negation as failure [8]). Also, the is_a/2 use above relies on a hierarchy of events to optimize the number of clauses needed for event subscriptions. The event schema we use for the Packet-World environment is shown in Fig. 4.

More detailed definitions of subscriptions can be defined that take into account topological information within a container. Consider for example agents facing each other. However, we abstract away from these definitions for simplicity of presentation. Similarly, we need to specify definitions for triggers (the sensors of objects). The details we omit as these subscriptions are simpler in that the agent producing the event has to specify also the object identifier to which the action is directed to.

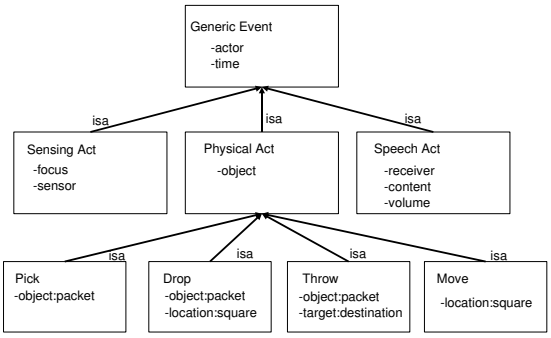



**Figure 4: Event Schema for Packet-World**

### 3.2.2 Local Action Execution

In AEC we represent actions within an agent environment as attempts. An attempt represents an action that is going to occur within the agent environment. Attempts are described by the assertion of events at a specific time, provided that such an attempt is possible within the physics of the agent environment. For instance, an agent that is attempting to pick an object at a certain time could be specified as follows:

```
attempt(e20,100)
do:e20[actor⇒ag1,act⇒ pick,object⇒obj1]
```

Such an attempt happens in the environment if the action is possible in the state of that environment. We specify this as:

```
(C13) happens(Event,T)← attempt(Event,T), possible(Event,T).
```

This definition requires that we describe the preconditions in which an event is possible at a certain time within the environment before an event can happen. Consider, for example, how to define when it is possible for an agent to move to a square in the grid:

```
possible(E, T)←
    do:E [actor ⇒ A, act ⇒ move:M [destination⇒ PosB]],
    holds_at(A, _, position, PosA, T),
    adjacent(PosA, PosB),
    not occupied(PosB, T).
```

The possible/2 rules mediate and constraint agent interaction happening within a container. Similar rules need to be defined to deal with an agent trying to move outside the area of the environment. We will show later, in Section 3.3, how to specify environment reactions as part of the physics.

### 3.2.3 Local Passive Perception

When an event happens, the agent environment notifies instantaneously all the types of sensors that are capable of detecting it, according to the type of sensors and the physics. Here the agent environment works as a mediating event dispatcher for event subscribers represented as agents and objects. The definition of the event notification has to take into consideration the affordances of the sensors and triggers. For passive perceptions, as in a fire alarm, we define the notification of events in AEC as:

```
(C14) notify(E, S, T) ← happens(E, T), detectable(E, S, T).
```

The key to notification is the detectable/3 predicate. For an event E, this checks if a sensor S subscribes to that event at time T, before E notified to the sensor. We define this as:

(C15) detectable(E,S,T) ← subscribes(E,S,T), not interfered(E,S,T).

Detectability of events by sensors does not only depend on subscription of events but also on some high level filtering that is due to the physical laws of the environment. We capture this via the notion of interference specified as rules named interfered/3. In Packet-World an agent cannot perceive an event of type pick, drop, throw or move if an obstacle is between the agent and the location where the event is generated. We represent this as:

```
interfered(E, S, T) ←
    physical_act:E [actor ⇒ Actor],
    holds_at(Actor,entity,position,XYe,T),
    holds_at(A, picker, sensor, S, T),
    holds_at(A, picker, position, XYa, T),
    holds_at(Entity, entity, position, XYent, T),
    in_between(XYent, XYa, XYe).
```

In other words, a notification of a pick, drop, throw or move act is interfered only when there is an entity between the position of the agent and the location in which the event happened.

### 3.2.4 Local Active Perception

Agents in AEC are enabled to actively perceive other entities deployed in the agent environment. Such perceptions are performed by an agent sensor that makes the call and specifies the conditions of what needs to be perceived. We specify this as:

```
(C16) perceive(E, S, T) ←
        happens(E, T),
        sensing(E),
        not interfered(E, S, T),
        E [filter ⇒ Focus],
        solve_at(Focus, T).
```

Here we assume that the agent has attempted to generate a sensing event successfully, there is nothing that interferes with the perception, the perception filters a set of properties of interest defined in the Focus, and these properties are retrieved by means of an asynchronous call to the environment via the solve_at/2 predicate. The call will return the variable substitutions to Focus; if there are no variables the call will verify whether the query holds or not.

## 3.3 Extra-container action and perception

Modelling an agent environment often requires designing an application as a complex topology of containers representing different parts of the environment. Such a topology could require containers to be embedded within containers or containers to be adjacent to other containers. One way to handle these requirements is to have a centralised component that knows about the affordances of every entity in the topology. However, the main issue with this solution is that the whole system is difficult to scale and will have a single point of failure. We present next an alternative solution where containers are distributed on different machines. We also discuss the issues of how to decide what holds in the distributed environment and how to distribute action execution, notification, and perception.

### 3.3.1 Distributing Containers

The first feature that we need to support is containers within containers. The issue then becomes how to allow for querying the state of a container with complex structure. We define the notion of a property holding locally, to a single, possibly structured container as follows:

```
(C17) locally_at(CId, Path, Path*, Id, Class, Attr, Val, T)←
        holds_at(CId, container, entity_of, Id, T),
        holds_at(Id, Class, Attr, Val, T),
        append(Path, [CId], Path*).

(C18) locally_at(CId, Path, Path*, Id, Class, Attr, Val, T)←
        instance_of(SubCId, container, T),
        holds_at(SubCId, container, super, CId, T),
        append(Path, [CId], NewPath),
        locally_at(SubCId, NewPath, Path*, Id, Class, Attr,Val,T).
```

The definition of locally_at/8 states that the affordances of an entity can be inferred either from the top-level container (C17) or from a sub-container (C18) and at the same time keeping the path of the visited containers (to be used in a while). In this way containers can be recursively embedded inside other containers as objects, according to the topology needed, and implemented on different hosts, if necessary.

In many applications the topology of the environment requires that containers are next to each other, for example, the distributed Packet-World in Fig. 3. As a result, we will need to define a region of the environment that agents will interested in, where an agent that belongs to one container needs to access part of another one. For this purpose we define containers that are linked with other containers to be neighbours, if one of the containers does not contain (or is contained by) the other container. We define querying neighbouring containers as follows:

```
(C19) neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
        Max > 0,
        locally_at(C, Path, Path*, Id, Cls, Attr, Val, T).

(C20) neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
        holds_at(C, container, neighbour, N, T),
        not member(N, Path),
        Max* is Max -1,
        append(Path, [C], New),
        neighbouring_at(N, New, Path*, Max*, Id, Cls, Attr, Val, T).
```

Now we need to keep the maximum number of containers (Max) that are part of the region and which containers have been visited so far (Path) with a resulting path (Path*), if the query succeeds. Clause C19 then looks for the property of the object locally, while C20 looks for the property in neighbouring containers not already visited.

One of the problems of neighbouring_at/9 is that when we look at a region, if the query fails, we do not check the super-container of the container that we fired the query from. To address this problem we introduce the regionally_at/9 predicate so that to query large areas of a distributed topology within the agent environment.

```
(C21) regionally_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
        neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T).

(C22) regionally_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
        holds_at(C, container, super, S, T),
        Max* is Max - 1,
        Max* >= 0,
        append(Path, [C], New),
        regionally_at(S, New, Path*, Max*, Id, Cls, Attr, Val, T).
```

Assuming a large Max, clauses C21 and C22 above visit all the nodes of the topology until a solution is found. Complex queries about the attributes of an entity are obtained as a combination of locally_at/8 and neighbouring_at/9 definitions. Similar rules are defined for the generalisation of instance_of/3 that are defined as local_instance_of/6, neighbouring_instance_of/6, and regional_instance_of/6. Their defi-

nitions we omit, as these simply rely on calling the predicates of clauses C17...C22, instead of holds_at/5.

### 3.3.2 Distributed Physics

When we deal with distributed containers, we need to enhance the definition of possible/2 to use what holds in regions. As a consequence, to specify that it is possible for an agent to pick up a packet, this must be adjacent. We define this as follows:

```
possible(E,T)←
    do:E[actor⇒A, act⇒ pick, object⇒Id],
    holds_at(A, agent, effector, Ef, T),
    instance_of(Ef, hand, T),
    holds_at(Ef, effector, state, free, T),
    holds_at(A, agent, position, PosA, T),
    neighbouring_at(this, [], _, 1, Id, packet, position, PosB, T),
    adjacent(PosA, PosB).
```

Such a definition means that it is possible for an agent to pick a packet if it has an effector of type hand, this is free and the agent resides in a position that is logically adjacent to the position of the packet. The call to neighbouring_at/9 allows us to check that the packet is in the right position, but possibly in a different container, thus allowing for a fully distributed physics of the agent environment.

From the perspective of a distributed event-based system, this means that the *event publication* is mediated by the physics of the agent environment. If the event is conformant with the physics, this will be published, otherwise the physics will prevent its execution.

After an agent attempts to move by generating a moving event in this way, such an event will cause a reaction by the environment. We support environment reactions in terms of attempts from the environment that are 'physically' necessary. We specify this as:

```
(C23) happens(E, T)← attempt(E, T), necessary(E, T).
```

In other words, attempt/2 captures the reaction event produced by the environment, while the necessary/2 specification is part of the physics. In the distributed Packet-World, we model an agent moving from one container to another in this way. For this purpose we define necessary/2 as:

```
necessary(E, T)←
  happens(E*, T),
  E*[actor⇒A, move ⇒ Pos],
  outside_borders(this, Pos),
  neighbouring_at(this, [], [C], 1, Id, square, position, Pos, T),
  E[move_to⇒(C, Id) agent⇒A,type⇒physical_act].
```

The definition above specifies that the environment should move an agent A who is in **this** current container to the square Id of container C with position Pos, as a consequence of the agent trying to move outside the boarders of **this** container. The event E is an event generated by the agent environment.

(C23) is used also to capture a process in the agent environment. The main role of such an abstraction is to support ongoing activities within a container, making a container an active entity through a relationship of causality. An example of a process is a force applied to an object: since there is an evolution over the time of an action, the process related to the force produces events accordingly to the intensity of the force applied to the object. A more detailed discussion on this topic is beyond the scope of this work.

### 3.3.3 Distributed Passive Perception

We have already seen that the detectable/3 and the subscribes/3 predicates specify the *delivery semantics* of an agent environment. However, to deal with a distributed agent environment we have to modify such predicates to use the AEC local, neighbouring, and regional versions for the notification of the events. For this purpose we change subscribes/3 as follows:

```
subscribes(E,S,T)←
    Class:E,
    forwards(E, Max, T),
    neighbouring_instance_of(this, [], _, Max, S, sensor, T),
    neighbouring_at(this, [], _, Max, S, sensor, senses, Class, T),
    neighbouring_at(this, [], _, Max, S, sensor, status, open, T).
```

This new definition of subscriptions identifies first the class of event, finds the maximum number of containers Max that the event can be forwarded, and then finds all the sensors within this neighbourhood that can sense the event. The forwarding of an event is specified according to the policies of the distributed environment for a certain type of event. For example, the forwarding of a speech act is specified as:

```
forwards(E, I, T)←
    Class: E [intensity⇒I],
    is_a(Class, speech_act).
```

The number of containers notified in the case of the speech act depends on the intensity attribute of the event. For instance, if a speech act event has to be detected only in the adjacent and local container, the agent will produce an event with intensity I equal to 1.

To complete the definition of what is detectable in the agent environment we need to provide domain constraints for interference. For the distributed Packet-World, we define these as:

```
interfered(E, S, T) ←
  E[actor⇒ Actor, type⇒ Type],
  holds_at(Actor, entity, position, XYe, T),
  is_physical_action(Type),
  forwards(E, Max, T),
  neighbouring_at(this, [], _, Max, A, picker, sensor, S, T),
  neighbouring_at(this, [], _, Max, A, picker, position, XYa, T),
  neighbouring_at(this, [], _, Max, Id, entity, position, XYent, T),
  in_between(XYent, XYa, XYe).
```

In this way, the delivery semantics of an application can be defined according to the requirements we have for the distributed agent environment. The AEC allows the developer to customise the agent environment according to these needs.

## 4. IMPLEMENTATION ISSUES

To accommodate the distributed environment model of the AEC presented here we need to extend the GOLEM model presented in [4] with a new reference model shown in Fig. 5. As with [4], we have a *notification* module, a *passive/active perception* module, an *attempts* module, a container *connector* interface, a *process* module specifying the environment's reactions, and a *physics* module. Unlike [4], however, we have now a new *AEC state*, a *mobility* module (within the connector interface) to support agents moving between containers, a *context daemon* to deal with distributed Prolog queries, and a new *synchroniser* module to synchronise the various containers. We outline next the need to implement the three new modules of this extended COLEM in AEC: the mobility module, the context daemon, and the synchroniser module.

## 4.1 The Mobility Module

One of the issues of a distributed environment is that an agent may have to move from one container to another (destination container) in order to access resources and interact with them. For this purpose a *mobility* module has been introduced within the connector interface of GOLEM. This enables the serialisation of an agent body that is implemented as a set of Java classes, and the serialisation of the agent mind implemented using SWI-Prolog [1]. Since both Java and SWI-Prolog support serialisation, our infrastructure supports strong mobility between containers. The deployment context and the modules within the physics are implemented as Prolog modules in tuProlog [26]. This Prolog engine was chosen because it allows the registration of Java objects within the Prolog context, has reflection mechanisms to translate Prolog calls to Java calls, and provides a programmable medium of interaction to constrain and enable the coordination between agents.

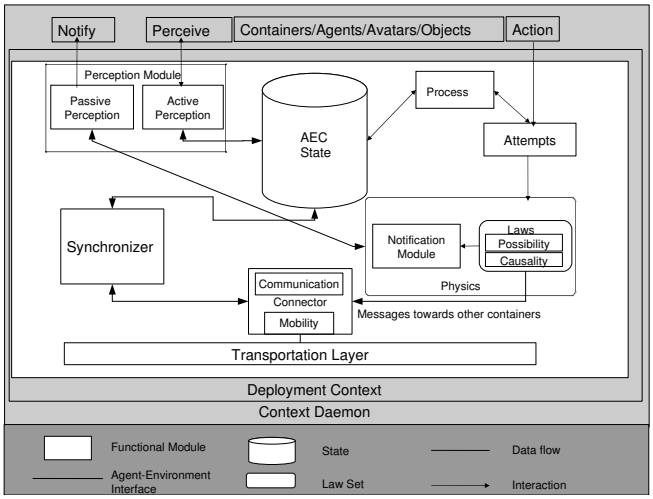



**Figure 5: The New GOLEM Reference Model.**

## 4.2 The Context Daemon

On top of the deployment context which handles the interaction within a container, we have added a *context daemon* that wraps the deployment context handling the AEC queries coming from other containers. To access the context of a container B from a container A we use the proxy pattern that allows us to represent container B in the container A. The predicates of the AEC in the container A will then make use of the proxy to query the context daemon of container B. Put another way, the proxy is a stub towards the context daemon and allows for remote method invocation on it. The methods here are predicates such as holds_at/5 and locally_at/8. The important point here is that the proxy allows for distributed backtracking via the transportation layer, which makes the whole approach quite powerful. Another important point to make is that the Context Daemon allows for multiple queries to be performed on a single containers working as a Web server: the declarative context of the GOLEM container is copied to perform a query and destroyed when not needed anymore. This avoids deadlocks and prevents an external entity to write in the original context of the container.

Another aspect of the implementation is that it seeks to minimise the number of messages exchanged between containers in the query phase. For this we added optimisation predicates in the AEC module, where rather than asking for just one attribute at a time, we ask for multiple attributes instead. For example we define locally_at_many/1 to query multiple attributes as follows:

```
locally_at_many([]).
locally_at_many([Oterm | Oterms])←
    locally_at_one(Oterm),
    locally_at_many(Oterms).

locally_at_one((Cld, Id, Class, Attr, Val, T))←
    locally_at(Cld, Id, Class, Attr, Val, T) .
```

In this way, we optimise the number of queries to the Context Daemon but we still need to deal with distributed backtracking. We deal similarly with the other predicates of the AEC, such as neighbouring_at/9 and regionally_at/9.

## 4.3 The Synchroniser Module

Another issue that we had to take into consideration in our implementation is that of *synchronisation*. A *synchroniser* component has been introduced to alleviate the issue of keeping an ordering between events generated in different containers. Every container in the distributed topology synchronises with its super-container using a PTP (Precision Time Protocol) described in the IEEE standard 1588. In this protocol a super-container sends a synchronisation message – SYNC message – with an estimated value of the time cyclically to the connected sub-containers. Parallel to this, the time at which the message leaves the sender is measured as precisely as possible, if possible at the nanosecond by the Java clock. The super-container then sends this actual exact transmission time of the corresponding SYNC message to the sub-containers in a second (follow-up) message. The sub-containers also measure the reception time of these messages as precisely as possible and correct the offset value to the one of the super-container. The sub-container clock is then corrected by this offset. If the transmission line were to have no delay, both clocks would be synchronized. We do not take into consideration any addition delays here and further consideration of this issue is beyond the scope of this work.

# 5. EVALUATION

The AEC formalism works as a distributed backtracking algorithm that moves from one container to another until a solution is found. The main purpose of AEC is to link the physics of containers as the distributed Packet-World presented here. However, to support such distributed applications we need to ensure a certain level of efficiency.

In this section we evaluate the performance of our approach on a medium scale network of containers where we show the behaviour of the neighbouring_at/9 predicate. The evaluation has been performed on a Intel Centrino Core 2, with 1GB of RAM and 1.9GHz. The most suitable topology for AEC is that of spanning trees as shown in Fig. 6. The AEC can also deal with graphs, but due to the fact that AEC implements a distributed backtracking, a node could be part of a solution multiple times, leading to performance issues.

The evaluation, done in GOLEM, is based on a spanning-tree topology of 50 containers where we register approxi-

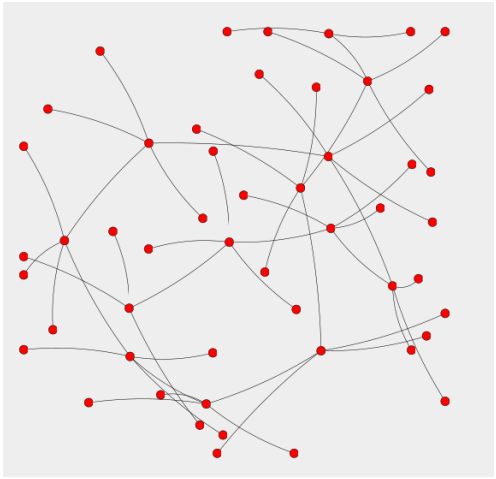


**Figure 6: Agent Environment Topology.**

mately 100 objects per container in the topology randomly. The attributes of such objects are then checked using neighbouring_at/9 queries. The evaluation of the neighbouring_at/9 predicate is shown in Fig. 7.

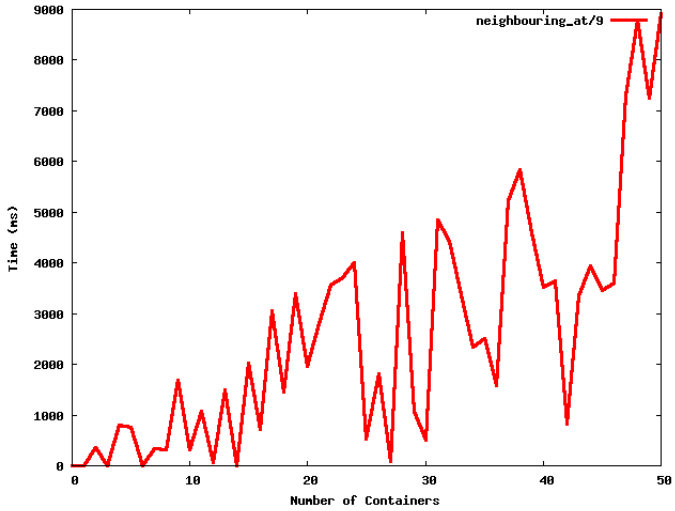



**Figure 7: Neighbouring_at/9 Results.**

Since objects are registered randomly, querying an object near the container from where the query is performed requires less time. The result shows that if the topology gets bigger, also the time to perform the neighbouring_at/9 query gets longer, following a linear asymptote. This result tells us that it is acceptable to use the AEC to link the physics of adjacent containers, but tells also that the AEC is not suitable for discovery of entities in large networks, because the time to retrieve the affordances of an entity can be long. Note also that the evaluation of neighbouring_at/9 is representative for the other predicates of the AEC (C17,C18,C21,C22) because they have the same behaviour from a network standpoint, except that these other predicates consider super and sub-containers rather than neighbour containers, meaning that rather than moving horizontally in the topology they move vertically.

## 6. RELATED WORK

The first work that we consider relevant to ours is the one presented by Gazzotti et al. [10], where the architecture of an event-based middleware is modelled for agents that interact by means of events. Their infrastructure may deploy different coordination laws using the notion of *local interaction context*, which defines the agent perceivable world, as a place where the interaction occurs. One advantage of this work is that it allows us to have a neat separation between inter-agent and intra-agent interactions since an agent's goal is specified within the internal reasoning of the agent, while the coordination between the agents is specified in the local interaction context. Such a context takes the form of a programmable coordination medium with a set of primitives operations that let the agents access it and a set of laws which define the internal behaviour in response to interaction events that the agents can generate or subscribe to. The coordination medium represented by the local interaction context works as an event dispatcher that receives, redistributes and elaborates events according to the behaviour programmed in it. Events are notified to the proper agent according to the signature of the event to which the agents are subscribed. Particular emphasis is given to the fact that the agents can move in the network across local interaction contexts. As a consequence the mobility is intended as a movement across organisations and not just across locations of the distributed infrastructure.

In our work, a single local interaction context can be seen as a container representing a portion of the distributed environment. Our main advantage is that events and primitives about subscription, notification, and delivery of events are defined in the AEC, whereas in the model proposed by Gazzotti et al. the structure of the events and the primitives of interaction remain undefined. Similarly to the model of Gazzotti et al., our new implementation of GOLEM in AEC supports mobility of agents and subscription to events through sensors, although the emphasis of AEC is not on the mobility part of an infrastructure, but on the interactions between the containers defining the distributed agent environment. In addition, the AEC formalism deals with event persistence implicitly, offering an easy way to give historical information about the interaction when needed. Moreover, the AEC allows our containers to define a distributed but connected topology where the rules can be shared between containers as well as being specific to the local context. Furthermore, GOLEM defines the concept of affordances to describe declaratively the producers and subscribers within the system, allowing an heterogeneous environment where agents and objects can coexist.

Natali et al [18] define a Java-based framewok for the development of component-based software systems. Such a framework focuses both on specifying the logic of interaction at the component level and on specifying the glue between the components deployed within the system. The work relies on first-order logic as the computational reference model for describing and defining the logic of interaction. In particular the framework specifies the logic of interaction at the component level, defining the interactive capabilities of individual components, and at the system level specifying the laws that define and govern the interaction that characterise the overall ensemble of the components together. Two main

entities of the system are used: *Actors* and *Kernels*. Actors are components meant to provide some kind of task or service which can be added or removed dynamically from a kernel. Kernels provide actors with specific services for supporting their interaction, working as the *environment* for the components. As a consequence the system is composed by a kernel and a dynamic set of actors linked through the same kernel. Components are characterised by the set of *interaction signals* that they can generate and receive. The kernel role is to act as a glue which enables mediates and controls the generation of output interaction signals. Of particular importance in the system are the interaction laws and the interaction vetoers. The former specify how the event listening happens, allowing a dynamic set of listeners to observe a specific interaction signal generated by a specific emitter or actor, the latter can specify cases when a component could prevent an event to be dispatched to another actor. More complex models can be created composing the primitives provided in the basic model.

There are several similarities between the model proposed by Natali et al. and an agent environment specification in AEC. First of all, the kernel abstraction can be seen as a container whose set of physical laws constrains the interaction in the system. Making use of sensors/triggers and emitters/effectors we can specify the capabilities of an entity deployed in the system. Our specification of the new GOLEM infrastructure has several advantages on the prototype proposed by Natali et al. First our use of AEC makes the specification of the interaction explicitly via notification, delivery, subscription, and distribution, including the implicit treatment of event persistence. Using AEC, GOLEM allows for a set of containers to interact and being controlled by coordination laws mediated in a distributed manner.

The JEDI platform [9] is an example of a distributed event-based infrastructure that could support agents. Such a platform is not explicitly based on the concept of agent, the authors prefer to abstract away an talk about *active objects*, autonomous component, each with its own thread of control, performing application specific tasks. In JEDI, events are represented as ordered set strings, which are generated by active objects and sent to a component called event dispatcher (ED). The role of the ED function is to notify events to active objects. In order to declare the classes of events the active object is interested in, it has to subscribe the class of events in the event dispatcher. Active objects can also arbitrarily unsubscribe their own interests. A hierarchical architecture of *Dispatcher Servers* is also proposed that can be interconnected to create a dissemination tree for the notification of events. The JEDI infrastructure proposes a synchronisation strategy called *causal ordering* to ease the issue of keeping an ordering between the events in a distributed infrastructure. In such a strategy, the events that are a consequence of other events are always delivered after the events that caused them. Such an infrastructure allows for mobility through serialisation of active objects, where the active objects are allowed to subscribe and unsuscribe their interest from the dispatcher server according to the position where they are in the topology.

With respect to JEDI, GOLEM deals explicitly with active entities that are cognitive agents and object that are reactive. The specification of an event dispatcher in AEC can be based on a distributed agent environment. Since there is an explicit model for the topology of the distributed environment, the dispatching and notification of events in AEC can be modified according to the particular topology of the environment and according to the event produced, while in JEDI this issue is not considered. Like JEDI the new GOLEM implementation presented here support the mobility of software entities through the serialisation of Java classes, and the dynamic event binding is realised registering sensors and effectors of the agents in the new container. GOLEM also supports the discovery of described entities through their affordances which JEDI abstracts away from.

HERMES [22] is another DEBS infrastructure with an advanced distributed event-based system model. The system is based on the concept of *event broker*, which works as a server for event publishers and event subscribers. The event brokers are distributed in the system by means of a *distributed hash table* created on top of the event types advertised in the system by event publishers. HERMES provides two routing protocols that are *type-based* and *type-based routing with inheritance* to notify event subscribers, where the difference between the two is given by how the event dissemination trees are created. HERMES also introduces the concept of *rendezvous nodes*, which are in charge of the definition of dissemination trees. Differently from other platforms, HERMES allows the definition of events as objects, or concepts of an ontology, with inheritance of the attributes from the top-level of the hierarchy to the bottom level. To enhance the robustness of the approach, HERMES handles the situation where an event broker leaves the network by mean of *heartbeat messages* exchanged by event brokers in order to update the dissemination trees periodically. If a rendezvous node is failing, another one can take over.

The main difference between GOLEM and HERMES is that the agent environment does not work just as an event dispatcher, but also as a mediator for the interaction happening in it. Similarly, HERMES and in general DEBS do not consider the entities to be situated in a particular environment were they interact. Due to the fact that there is no need to provide mediation systems like HERMES and JEDI, what happens is that modelling heterogeneous systems, like the one proposed in this paper, is very hard because of lack of coordination mechanism (like the physics of GOLEM) to avoid inconsistent states and due to the fact that there is no support to perceptions and physical actions. To model a MAS as a DEBS system, we need to take into consideration more factors than just how to dispatch events from the point of view of interaction. Since GOLEM resides at a higher level of abstraction than JEDI and HERMES, an advantage of the GOLEM infrastructure as defined here is flexibility: mediating the interaction of agents and objects allows us to potentially define a different dispatching policy according to the particular event. We can achieve this by modifying/adding predicates to the AEC or by adding a new service to the agent environment in charge to receive a selected number of types of events (or all of them) produced in the environment and dispatch them according to the policy defined by the developer, without having to change the whole infrastructure.

# 7. CONCLUSION

We have studied the development of distributed agent environments as distributed event-based systems specified in the *Ambient Event Calculus* (AEC). The AEC is a logic-based formalism that has been developed to support the

representation of a distributed agent environment as a persistent composite structure evolving over time. Such a complex structure can support the interaction between *agents*, *objects*, and *containers*, entities that have their own external observable state and can be distributed over a network. We have shown how interactions between these entities are specified in terms of *events* that represent actions executed by agents on objects and other agents in the environment. When events happen they are stored in containers and are notified to agent sensors that subscribe to event descriptions and as a result perceive the interactions. The AEC formalism also allows changes caused by events to be delivered across distributed containers, according to the topology of the application environment. We have illustrated the use of AEC and we have shown how to specify interactions within the GOLEM agent platform applied to a specific agent scenario. We have also implemented a prototype of the platform and experimented with the scenario.

One of the contributions of the AEC is the hierarchical definition of a distributed agent environment defined through containers which are recursively deployed in other containers, where the issues of event dispatching and delivery semantics are handled according to the topology and physical laws of the environment. The resulting framework is a powerful mechanism for developing distributed multi-agent systems capable of mediating the interaction of the entities deployed in it, and supports both dynamic binding of agents to events and mobility between containers.

The Packet-World example has been chosen here for simplicity of presentation. More sophisticated applications of GOLEM, in the way described here, include how to apply multi-agent systems to support applications for (a) web-service discovery, selection, and negotiation [6] and (b) 3D virtual environments [5]. Future work involves testing the AEC as used in GOLEM with application on ubiquitous computing and ambient intelligence [25] and show how the system can deploy context aware agents in small devices.

## 8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and Robin Bennett for their comments on a previous version of this paper. This work was partially supported by the IST-FP6-035200 ArguGRID project.

## 9. REFERENCES

[1] SWI-Prolog. http://www.swi-prolog.org/, February 2009.

[2] F. Bellifemine, A. Poggi, and G. Rimassa. JADE: a FIPA2000 compliant Agent Development Environment. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 216–217. ACM Press, May 2001.

[3] R. Blanco, J. Wang, and P. Alencar. A Metamodel for Distributed Event-based Systems. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 221–232, New York, NY, USA, 2008. ACM.

[4] S. Bromuri and K. Stathis. Situating Cognitive Agents in GOLEM. In *Engineering Environment-Mediated Multiagent Systems (EEMMAS'07)*. Springer, Oct 2007.

[5] S. Bromuri, V. Urovi, P. Contreras, and K. Stathis. A Virtual E-retailing Environment in GOLEM. In *Intelligent Environments (IE'08)*. IET, July 2008.

[6] S. Bromuri, V. Urovi, M. Morge, F. Toni, and K. Stathis. A Multi-Agent System for Service Discovery, Selection and Negotiation. In *Procedings of the Eighth International Conference in Autonomous Agents and Multi Agent Systems AAMAS09*, Budapest, Hungary, May 2009. Demonstration.

[7] W. Chen and D. S. Warren. C-logic of Complex Objects. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 369–378, New York, NY, USA, 1989. ACM Press.

[8] K. L. Clark. Negation as Failure. In *Logic and Data Bases*, pages 293–322, 1977.

[9] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI Event-based Infrastructure and its Application to the Development of the OPSS WFMS. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[10] M. Gazzotti, M. Mamei, and F. Zambonelli. A Programmable Event-based Middleware for Pervasive Mobile Agent Organizations. In *In 11th IEEE EUROMICRO Conference on Parallel, Distributed, and Network Processing*, pages 517–525, 2003.

[11] J. J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, 1979.

[12] JACK. The Jack Development Environment. Home Page: http://www.agent-software.com/shared/products/index.html.

[13] JADE. Java Agent DEvelopment framework. Home Page: http://jade.tilab.com.

[14] A. C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of Agency. In *Proceedings of the 16th European Conference of Artificial Intelligence*, pages 33–37, Valencia, 2004.

[15] F. N. Kesim and M. Sergot. A Logic Programming Framework for Modeling Temporal Objects. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):724–741, 1996.

[16] N. Kesim. *Temporal Objects in Deductive Databases*. PhD thesis, Imperial College, 1993.

[17] O. S. M. Luck, P. McBurney and S. Willmott. Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing). In *AgentLink*, 2005.

[18] A. Natali, E. Oliva, A. Ricci, and M. Viroli. A Framework for Engineering Interactions in Java-based Component Systems. In C. Carlos and M. Viroli, editors, *Proceedings of FOCLASA 2005, 4th Intl. Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2005)*, CONCUR 2005, San Francisco (CA), USA, 27Aug. 2005.

[19] D. A. Norman. Affordance, Conventions, and Design. *Interactions*, 6(3):38–43, 1999.

[20] H. S. Nwana, D. T. Ndumu, L. C. Lee, and J. C. Collis. ZEUS: A Toolkit for Building Distributed Multiagent Systems. *Applied Artificial Intelligence*, 13(1-2):129–185, 1999.

[21] J. Odell, H. V. D. Parunak, M. Fleischer, and S. Brueckner. Modeling Agents and their

Environment: The Physical Environment. *Journal of Object Technology*, 2(2):43–51, 2003.

[22] P. R. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[23] A. Ricci, M. Viroli, and A. Omicini. CArtAgO: A Framework for Prototyping Artifact-based Environments in MAS. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 67–86. Springer, Feb. 2007.

[24] K. Stathis, S. Kafetzoglou, S. Papavasilliou, and S. Bromuri. Sensor Network Grids: Agent Environments combined with QoS in Wireless SensorNetworks. In *The Third International Conference on Autonomic and Autonomous Systems (ICAS07)*, Jun 2007.

[25] K. Stathis, R. Spence, O. D. Bruijn, and P. Purcell. Ambient Intelligence: Human - Agent Interaction in Connected Communities. In P. Purcell, editor, *Networked Neighbourhoods: The Connected Community in Context*. Springer-Verlang, July 2006.

[26] tuProlog. tuProlog. Home Page: `http://www.alice.unibo.it:8080/tuProlog/`.

[27] D. Weyns, A. Helleboogh, and T. Holvoet. The Packet-World: a Test Bed for Investigating Situated Multi-Agent Systems. Software Agent-Based Applications, Platforms and Development Kits, Whitestein Series in Software Agent Technology., 2005.

[28] D. Weyns, A. Omicini, and J. Odell. Environment as a First Class Abstraction in Multi-agent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.

[29] D. Weyns, H. V. D. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for Multiagent Systems State-of-the-Art and Research Challenges. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *E4MAS*, volume 3374 of *Lecture Notes in Computer Science*, pages 1–47. Springer, 2004.

[30] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.