# CArtAgO: An Infrastructure for Engineering Computational Environments in MAS

Alessandro Ricci, Mirko Viroli, and Andrea Omicini

DEIS, Università di Bologna
via Venezia 52, 47023 Cesena, Italy
{a.ricci,mirko.viroli,andrea.omicini}@unibo.it

**Abstract.** Artifacts have been recently proposed as first-class abstractions to model and engineer general-purpose computational environments for multiagent systems. In this paper, we consider the design and development of an infrastructure called CArtAgO, directly supporting the artifact notion for the engineering of multiagent applications. We first propose an abstract model of the infrastructure, and then describe an implementation prototype of it.

## 1 Introduction

*Artifacts* have been recently proposed as first-class abstractions to model and engineer computational environments in software MASs (multiagent systems) [1, 2]. The background view—which is shared with other recent approaches in MASs literature (see [3] for a survey)—is that the environment can play a fundamental role in engineering MASs. On the one hand, it is a suitable locus where engineers can embed responsibilities, impacting on MASs design and development; on the other hand, it is a source of structures and services that agents can suitably use at runtime to support and improve their activities—both individual and social ones. The artifact notion promotes a methodology for modelling and engineering such computational environments, by introducing concepts and elements that impact on system design, development and runtime management.

Artifacts can be generally conceived as *function-oriented* computational devices, namely, they are designed to provide some kind of *function*, that agents can exploit to support their individual and collective (social) activities [1]. The notion of "function" here refers to the meaning that is generally used in human sciences such as sociology and anthropology, as well as in some recent work in AI [4], that is, the purpose for which the device has been designed for—for an artifact, to support agent activities.

This view directly impacts on the foundation of interaction and activity in agency: a MAS is conceived as an (open) set of agents that develop their activities by *(i)* computing, *(ii)* communicating with each other, and *(iii) using* and possibly constructing shared artifacts—which embody the MAS computational environment. Generally speaking, artifacts can be either the target (outcome) of agent activities, or the *tools* that agents use as means to support such activities: as such, they are useful to reduce the complexity of their tasks' execution. For instance, *coordination artifacts* [5] are artifacts providing coordination functionalities—examples are blackboards, tuple spaces or workflow engines.

The conceptual and theoretical background of this framework stems from the theories developed in the context of human science, in particular Activity Theory

[6] and Distributed Cognition [7]. Also, this perspective shares the aims and the principles developed in existing research work in Distributed Artificial Intelligence about theories of interaction [8, 9] and in Computer Supported Cooperative Work, with the notion of *embodied interaction* [10].

From an engineering point of view, by following this approach artifacts along with agents become the basic building blocks (i.e. abstractions) to design and develop MASs: designers can use *(i)* agents to model autonomous activities, which are typically goal / task oriented, and *(ii)* artifacts to model structures, objects, typically passive and reactive entities which are constructed, shared and used during the execution of such activities. The artifact abstraction provides then a natural way to model object-oriented (OO) and service-oriented abstractions (objects, components, services) at the agent level of abstraction, bridging the conceptual and semantic gaps with the agent-oriented paradigm. On the one hand, as in the case of objects (components) and services, artifacts expose interfaces composed by operations that can be invoked by agents—though relying on a different semantics. On the other hand, differently from the case of object-oriented and service-oriented models, the invocation of an operation on an artifact *never results in a transfer of control*: agents fully encapsulate their control flow(s). If this feature can be found in several *patterns* developed in the context of OO and service-oriented software engineering, in our case it becomes one of the fundamental principles which underly the artifact model, and that eventually imposes a strict discipline in system design and implementation.

In order to stress the validity of the artifact model, and as a basis to extend and evolve it, we find it useful to setup a prototype *infrastructure*, referred here to as CArtAgO (Common "Artifacts for Agents" Open infrastructure). This is to be concretely used for engineering multiagent applications, providing designer with the artifact notion and all the related concepts.

Infrastructures play an essential role for keeping useful abstractions alive from design to runtime [11]. Agent infrastructures (or middleware) typically provide fundamental services for agent creation, management, discovery and (direct) communication: well-known examples are RETSINA [12] and JADE [13]. Analogously, CArtAgO is meant to be exploited for creating and sustaining the existence at runtime of computational environments engineered in terms of artifacts. It provides basic services for agents to instantiate and use artifacts, as well as a flexible ways for MAS engineers to design and construct any useful kind of artifact.

In this paper, we first discuss the basic aspects that characterise CArtAgO from an abstract point of view—its abstract model (Sect. 2) and a core of abstract API (Sect. 3); then, we describe a first concrete prototype (Sect. 4), implementing the core part of the abstract model previously defined.

## 2 CArtAgO Abstract Model

The abstract model of CArtAgO concerns three basic parts, which will be developed in the following subsections: *(i)* the model adopted for characterising actions and perceptions linking agents to their computational environment, *(ii)* the abstract model adopted for artifacts—as basic bricks to engineer such an environment—, and finally *(ii)* the model adopted for workspaces, as logical contexts where artifacts and agents are located.

### 2.1 Agent Side: Actions & Perceptions

As a premise to the following discussion, the artifact notion calls for introducing a model (and a theory) of interaction which is different from the models generally adopted in software agent infrastructures, which are typically based solely on communicative acts. Rather, the artifact model of interaction is more similar to models defined for autonomous / situated agents. Agents interact with their computational environment by means of suitable *actions* provided for artifact construction, selection and usage, and by perceiving *observable events* generated from such artifacts: we refer to this kind of actions as *pragmatic acts*. The term *pragmatic* is adopted to remark the differences with respect to *communicative* actions, which are the main—we should say the only—type of actions that are typically modelled and supported in the most-diffused languages, architectures and infrastructures for software agents, such as e.g. FIPA-compliant platforms. Kinds of pragmatic action are pervasive, instead, in all the approaches proposing an explicit notion of environment in MASs (which, however, cannot be considered part of the mainstream in current agent community, yet). So, in our case, differently from communicative actions that have agents as targets—targets of the message sent—, pragmatic actions as defined here are directed to an artifact. A primary example of pragmatic action is the execution of an operation, whose effects can be the generation of streams of events distributed in time.

The distinction between communicative and pragmatic actions is fundamental also from a theoretical point of view, in particular if we consider the context of intelligent agents. The semantics of a *speech acts* is typically defined in terms of preconditions and effects on either the mental state of the communicative agents or the social relationships that link such agents. On the other hand, such semantics is not meaningful when we consider the interaction between agents and artifacts, for artifacts are not suitably characterised in terms of mental or social attitudes (belief, desires, intentions, goals, commitments, etc.), but rather by function-oriented features. Actually, lots of research work has been developed to characterise from a theoretical point of view communicative actions, trying to defined a semantics which could be fruitfully exploited to support agent reasoning. Analogously, we think that research work on theoretical aspects of pragmatic actions for software agents—which is still in its infancy—is fundamental to conceive agent models and architectures that support reasoning about the computational environment and its exploitation.

As in the classic agent model [14], agents perceive events through *sensors*, as collectors of environment stimuli. In CArtAgO, sensors are structures provided by the infrastructure: agents can flexibly create and use them to partition and control the information flow perceived from artifacts, possibly providing specific functionalities such as buffering, filtering, ordering, and managing priorities. *Sensing* is the internal action that agents execute on their sensors to become aware (perceive) of the events (stimuli) collected by the sensors.

### 2.2 Artifacts

Artifacts are the basic bricks managed by CArtAgO infrastructure. In the following abstract model we define some basic features and properties which are essential in their construction, manipulation and use, independently from the specific implementation models.

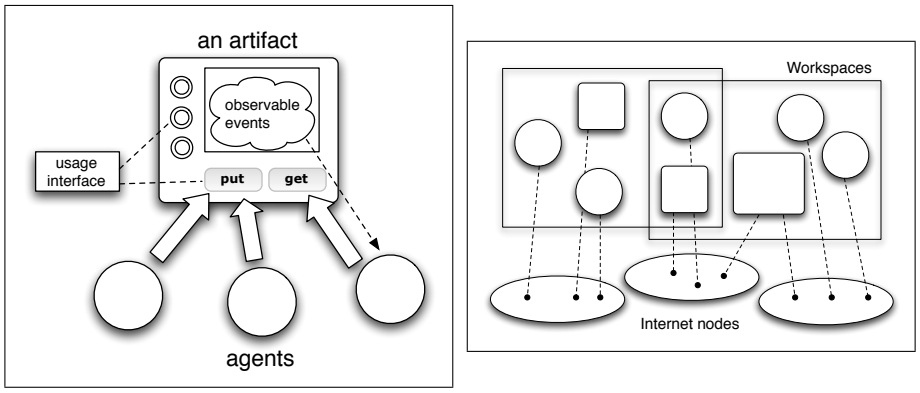



**Fig. 1.** *(left)* An abstract representation of an artifact, used by agents through its usage interface. In the picture the usage interface is represented metaphorically as the set of buttons that agents press to interact with the artifact, and an output area where containing observable events. *(right)* An abstract representation of workspaces, used to represent from a logical point of view the topology of the system. In the picture two workspaces are represented, mapped onto three Internet nodes. Agents and artifacts can belong to multiple workspaces (that is, workspaces can intersect). Also, the same artifact can be distributed on multiple nodes.

**Identity** Each artifact has a logic *name* specified by the artifact creator at instantiation-time, and an *id*, released by the infrastructure, to univocally identify the artifact. The logic name is an agile way for agents to refer and speak about (shared) artifacts, while the id is required to identify artifacts when executing actions on them. The *full name* of an artifact includes also the name of the workspace(s) where it is logically located. Since an artifact can be located in multiple workspaces, the same artifact can be referenced by multiple full names.

It is worth remarking that an artifact can be both logically and physically distributed, that is the same artifact—i.e. an artifact with the same identity and state—can have a spatial extension, being located in multiple locations according to the chosen topology. This characteristic is not typically found in the agent abstraction: agents, even if can have a distributed implementation, from a logical point of view are always situated in a specific point of the topological space, which can e.g. dynamically change in the case of mobile agents.

**Usage Interface & Events** Each artifact has a *usage interface* that agents exploit in order to interact with it, i.e to *use* it. A usage interface is defined as a set of *operations*: agents interact with artifacts by invoking operations and observing *events* generated from them, perceived through sensors. An operation is characterised by a *name* and a set of *parameters*. Parameters, as well as the event generated, are meant to have a type.

Operations have no return value concept, differently from e.g. methods in the OO paradigm: any information, generated due to the operation execution and which can be of interest for the invoker agent, is modelled as an observable event generated by the artifact and later received by agents as a perception. Also, exceptions

(error conditions) generated *during* the execution of an operation are modelled as observable events.

As mentioned in the introduction section, a radical difference with respect to the notion of interface as found in OO and component-oriented paradigms is the *control uncoupling*. In such paradigms, the execution of a method causes the control to flow with the data (parameters) from the object invoking the method to the object owning the method: in other words, there is *no encapsulation* of the control. In order to avoid this kind of control flow, some support for *threads*[1] must e.g. be used, as typically found in some well-known OO patterns for managing concurrency [15]. By considering agents and artifacts, such an aspect is supported instead at the very foundation level, and there is no need to introduce any explicit multi-threading support.

The usage interface of an artifact can depend on its state, analogously to GUI interfaces of applications: in other words, an artifact can expose different set of operations according to its state. This is a simple and direct way to structure the interface of artifacts, directly supporting what is typically implemented as a pattern in the context of object-oriented paradigm, where interfaces are typically fixed.

**Function Description and Operating Instructions** In order to support a rational exploitation of artifacts by intelligent agents, each artifact is equipped with a *function description*, i.e. an explicit description of the functionalities it provides, and *operating instructions*, i.e. an explicit description of *how* to use the artifact to get its function—for instance in terms of the *usage protocols* that the artifact support. These descriptions are meant to be useful for cognitive agents that—by suitably inspecting and interpreting them—can *(i)* dynamically reason about which artifacts can be selected to support their activities, and *(ii)* get instructions to support activity execution, making it easier to set up plans and to reason about the expectation of using an artifacts. We consider such issues of foremost importance, at the core of the notion of computational environments designed to support the activities of agents—in particular cognitive / rational agents. Actually, the research on these aspects, in particular on formal models and languages that can be used to specify function description and operating instructions, and their injection in existing agent reasoning architectures (such as BDI), is still to be fully developed: first work can be found in [16].

In CArtAgO, we provide a minimal but enabling support to these issues, by modelling function description and operating instructions as simple textual documents that can be specified for any artifact by its designer, providing basic services for their dynamic inspection. We do not fix the specific model and formal semantics for such documents: just to promote a first naive form of interoperability, we consider the use of XML as representation language.

**Observable State** Artifacts are stateful reactive entities, with a state that can change according to the operations executed by agents. Actually, in the case of time- and location-aware artifacts, state and location of an artifact can change also in reaction to time passing (e.g. a clock). Reaction to time passing is a simple way

[1] We remark here that the thread concept is per se a foreign one in OO foundations.

to realise artifacts that exhibit a form of *active* behaviour, even if actually they do not encapsulate any control flow, at least in the sense that we have seen for the agent abstraction, where *pro*-active behaviour—instead—is considered. Pheromones or pheromone environments are an effective example of computational environments that can be designed and implemented as time-aware artifacts, where the state—the pheromone intensity in this case—changes (also) according to time passing.

As for artifacts in human society, we consider it useful to explicitly define a notion of *observable state*, as dynamic information (such as a set of properties) exposed by an artifact, that can be dynamically observed by agents without necessarily interacting through its usage interface. Such an observable state includes also the usage interface description, whose shape can change according to the state of the artifact, as mentioned previously. Also artifact observable state is sensed by agents as events perceived from the environment, uniformly to the perception model introduced previously. Analogously to function description and operating instructions, in this first model of CArtAgO we do not consider specific models and semantics for describing artifact observable state, and we just consider a flat textual XML description.

### 2.3 Modelling Topologies: Workspaces

Artifacts (and agents) are logically located in *workspaces*, which can be used to define the topology of the computational environment. A workspace can be defined as an open set of artifacts and agents: artifacts can be dynamically added to or removed from workspaces, agents can dynamically enter (join) or exit workspaces. A workspace is typically spread over the nodes of an underlying network infrastructure, such as the Internet. In CArtAgO, each workspace is created by specifying a logic name and is univocally identified by an id released by the infrastructure. Workspaces make it possible to define topologies to structure agents and artifacts organisation and interaction, in particular as scopes for event generation and perception. On the one side, a necessary condition for an agent to use an artifact is that it must exist in a workspace where the agent is located. On the other side, events generated by the artifacts of a workspace can be observed only by agents belonging to the same workspace.

Intersection and nesting of workspaces are supported to make it possible to create articulated topologies. In particular, intersection is supported by allowing the same artifacts and agents to belong to different workspaces.

### 2.4 Modelling Organisation and Security Issues: Toward Workplaces

The adoption of explicit organisational models is a crucial aspect for managing the complexity of the MAS structure, in particular for open scenarios [17]. As largely discussed in literature (electronic Institutions are a primary example [18]), organisational models can be fruitfully exploited also to specify and manage some aspects that concern security inside the systems—access control in particular—by defining the organisational policies and norms that rule the relationships among the system parts.

By drawing on our previous research work on such aspects on TuCSoN infrastructure [19], in CArtAgO we introduce a *role*-based model, inspired to RBAC (Role-Based Access Control) architectures [20]. According to the results coming from dif-

| | |
|---|---|
| Artifacts use | `invokeOp(ArID,OpName,[Args],{SensorID}): OpID`<br>`sense({SensorID},Timeout): EventDescr`<br>`sense({SensorID},{Pattern},Timeout): EventDescr`<br>`createSensor(SensorType,SensorConfig): SensorID` |
| Artifacts construction and manipulation | `createArtifact(Name,Template,Config,{WsID}):ArID`<br>`getArtifactID(Name,{WsID}):ArID`<br>`disposeArtifact(ArID)`<br>`registerArtifact(ArID,WsID)`<br>`deregisterAr(ArID,WsID)` |
| Artifacts selection and inspection | `getFD(ArID): FdDescr`<br>`getOI(ArID): OIDescr`<br>`getUID(ArID): UIDDescr`<br>`getState(ArID): StateDescr` |
| Workspaces management | `getWsID(WsName,{Location}):WsID`<br>`joinWS(WsID)`<br>`exitWS(WsID)`<br>`createWS(WsName,Location):WsID`<br>`addWSNode(WsID, Location)`<br>`removeWSNode(WsID, Location)`<br>`disposeWS(WsID)` |

**Table 1.** Actions available to agents to manage artifacts and workspaces.

ferent disciplines, role-based organisational models are among the most effective approaches to specify the structure of complex and articulated systems (human / social systems as well as artificial ones). RBAC models—mainly adopted for engineering security in complex information systems—adopt roles as the basic abstraction to encapsulate user permissions to access system resources: that is, a user can access system resources according to the permissions granted to the role the agent is playing within the organisation.

In CArtAgO we inject such concepts with the notion of *workplaces*. Workplaces define an organisational layer on top of workspaces. A workplace defines the set of roles and related organisational rules or *contracts* being in force in a workspace. The contracts defines, in particular, the norms and policies that rule agent access to the artifacts belonging to the workspace. For example, depending on the role(s) that an agent is playing inside the workplace, it can have or not the permission to use some artifacts or to execute some specific operations on some specific artifacts.

By following the RBAC approach then, agents aiming to participate to a workplace must first be assigned to the role(s) that they aim at playing, and after that, such roles can be dynamically activated / deactivated by the agents according to the need. Both role assignment and activation are subject to organisational rules, defining which (authenticated) agents can be assigned to which roles, and when agents can dynamically activate/de-activate such roles.

These aspects are not discussed further in this paper though they are important, and will be considered in details in future works.

## 3 Core Primitives

In order to exploit the infrastructure services, a basic abstract set of API (Application Programming Interface) has been defined, related both to the agent side—that is, to be used by agents (or agent programmers defining agent behaviour) to exploit artifacts—and the artifact side, i.e. used for programming artifacts.

### 3.1 Agent side

On the agent side, the API is represented by a set of primitives that can be thought as actions available to the agent, and that basically make it possible to create, locate, manipulate, and use artifacts. Table 1 provides an abstract description of such primitives, grouped according to their functionalities:

**Artifacts Construction & Manipulation** — Basic primitives are provided to create (*createArtifact*) and dispose (*disposeArtifact*) artifacts dynamically. To create an artifact, a logic name must be specified, along with the *template* that identifies the type of the artifact to be created, possible configuration parameters needed for artifact creation and optionally the workspace where the artifact should be created. The identifier of an existing artifact can be obtained by the *getArtifactID* primitive, specifying the artifact name and (possibly) its location (workspace). By omitting the location, the current workspace(s) where the agent is situated are considered.

The same artifact can be part of multiple workspaces: accordingly, basic actions are provided to register (*registerAr*) / de-register (*deregisterAr*) an artifact in / from a workspace, specifying the workspace id.

**Artifact Use** — These primitives constitute the core of agent / artifact interactions, enabling an agent to invoke operations and sensing events. To execute an operation the action *invokeOp* is provided, specifying the artifact id, the operation name, the parameters, and the specific sensor where to collect events possibly generated by the artifact in relation to this specific operation execution. The invocation of an operation can fail, due for instance to the unavailability of the artifact. This kind of failures should be distinguished from errors that can raise when executing the operation on the artifact and that depends on the specific semantics of the operation: such errors are made observable to the agents as events.

For perceiving the events collected by sensors, a set of *sense* primitives are provided. By executing a sense action, an agent is made aware of the events (stimuli) that have been eventually collected by a specific sensor. The effect of the action is to fetch (remove) an event from the sensor and to return it to the agent as a perception. Different types of sensors can provide different semantics establishing the order in which events are fetched.

A time parameter is specified to indicate a maximum duration for the sensing action: if no events are available in the sensor within the specified timeframe, the action fails.

Besides this simple form of sensing, a *pattern-driven* sensing is supported: a pattern parameter can be specified acting as a filter for fetching the events. Again, specific types of sensor can support specific models for pattern matching.

| | |
|---|---|
| Event generation | `genEvent(EventDescr)`<br>`genEvent(OpID,EventDescr)`<br>`genEventWs(EventDescr)` |
| Operation management | `getOpID: OpID` |
| State exposition | `exposeState(StateDescr)` |

**Table 2.** Basic primitives for artifact programming

Finally, a primitive *createSensor* can be used to flexibly create instances of sensors, specifying their types.

**Artifacts Selection & Inspection** — In order to support a *cognitive* use of artifacts, a basic set of primitive is provided to inspect the function description (*getFD*), the operating instructions (*getOI*), the usage interface (*getUID*), and the dynamic observable (exposed) state (*getState*) of the artifact. Such information is provided to the agent as documents encoded in a machine-readable format, such as XML, possibly equipped with a formal semantics defining the content of the documents with respect to some ontologies. OWL[2], for instance, can be a good candidate for this purpose.

**Workspace manipulation** — Finally, a basic set of primitives is provided to manipulate the logical topology of the environment, modelled through workspaces. Such primitives range from *joinWS* and *exitWS* to join and leave a workspace, to *getWsID* for getting a workspace identifier given its name and possibly one of the (network) nodes where the workspace is located, *createWS* for directly creating a new workspace and *disposeWS* for completely removing a workspace. Since a workspace typically spans on multiple network nodes, *addWSNode* is provided to dynamically extend an existing workspace on a specific network node, and conversely *removeWSNode* to remove a workspace from the specified network node.

Most of these core services have been implemented in the prototype described in Sect. 4.

### 3.2 Artifact side

Table 2 shows the minimal set of abstract primitives provided on the artifact side. Such primitives are meant to be exploited by programmers defining artifact structure and behaviour, and are useful essentially for generating observable events and expose the artifact observable state.

Despite the specific programming model adopted for defining artifacts, artifact behaviour is meant to be structured in operations, characterised by a name and a set of parameters. Each operation request served by the artifact is labelled by a unique operation identifier (type *OpId* in the tables). An event can be then generated using the *genEvent* primitive by specifying the operation identifier to which the event must

[2] `http://www.w3.org/TR/owl-features/`

be related, as observable effect of this operation (and of the agent action that caused it). If no *OpId* is specified, the event is considered related to the current operation request. The effect of the execution of these primitives is the generation of an event that is dispatched to the agent that invoked the operation.

The *OpId* can be retrieved by invoking the primitive *getOpID* during the execution of the operation (as part of its execution body). Operation identifiers are meant to be manageable as normal data structures, for instance, creating list of operation identifiers and then generating events related to these operation when necessary, during artifact functioning, across operation executions.

In order to model the generation of events which are not directly related to any specific agent operation request, the primitive *genEventWs* is provided, generating an event which is not related to a specific operation execution and that is observed by all the agents residing in the same workspace(s) of the artifact.

## 4 A First Prototype

A first prototype implementing some of the functionalities described in the previous section has been developed in Java and is available for download at the CArtAgO project web site[3]. The prototype can be used to implement both concurrent and distributed applications designed and structured in terms of agents and artifacts, programmed in Java. A simple programming model called simpA[4] has been adopted to code agent and artifact structure and behaviour, to create agent and artifact templates.

In this first prototype a CArtAgO application is simply a Java application exploiting CArtAgO and simpA libraries (provided by the `alice.cartago` and `alice.simpa` packages) containing the API to program agents and artifacts. The main loop of the application is responsible to setup the CArtAgO environment (runtime) and to populate the environment with the initial set of agents and artifacts constituting the booting configuration of the application. Specific primitives are provided by the CArtAgO environment at this stage to spawn agents and create artifacts.

With respect to the full model presented in previous sections, in this prototype workspaces are not supported, and then the topology is defined directly in terms of the infrastructure nodes. More precisely, each launched CArtAgO application—running on a Java virtual machine—represents a node of the infrastructure, so multiple nodes can reside on the same Internet host. Each node (application) can contain a dynamic set of agents and artifacts.

### 4.1 Architecture

Figure 2 shows the abstract layered architecture of a CArtAgO node (application). At the core of the architecture there is the *kernel*, which acts as the glue between agents and artifacts, providing the core services described in previous sections, exploited through the API. Among the various functionalities, the kernel keeps track of the agents and artifacts in execution on the nodes (through maps), exploits agent and artifact registered *factories* to dynamically create agents and artifacts given their

[3] CArtAgO project web site: `http://www.alice.unibo.it/cartago`
[4] simpA project web site can be found at `http://www.alice.unibo.it/simpa`

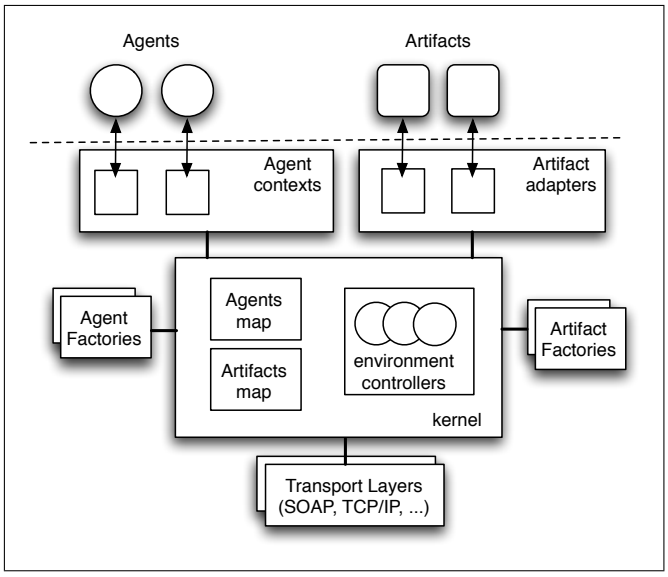



**Fig. 2.** Abstract architecture of a CArtAgO infrastructure node

templates, collects and serves operation execution requests generated by agents, and dispatches events generated by artifacts, possibly using the available *transport layers* when agents and artifacts reside on different nodes. The operation execution requests on artifacts are served by a pool of environment controllers, acting as worker threads—it is worth remembering here that artifacts, differently to agents, do not encapsulate any control flow.

Agent access to kernel functionalities is mediated by *agent contexts*. The notion of agent context is introduced to explicitly represent the local computational environment encapsulating CArtAgO structures that are private to the individual agents, such as sensors. Generally speaking, an agent context acts as a bridge between the individual agent and the environment where the agent plays, providing the very basic interface to act inside CArtAgO, and in the overall defining the basic set of actions and perceptions allowed for the agent. So, the agent context is responsible on the one side to route agent requests to the kernel and on the other side of dispatching the events posted by the kernel to the agent, through the sensors managed by the agent contexts.

Actually, an agent context can be fruitfully exploited to represent a *working session* of the agent inside the system, and then to embed and enact possible organisational and security policies related to the role(s) that the agent is playing. When an agent starts a working session, an agent context is created and configured so as to enable (filter) only those actions and sequence of actions that the agent can execute according to his role(s).

### 4.2 Programming Artifacts and Agents

The simpA programming model provides a simple way to implement agents and artifacts as patterns on top of Java, adopting CArtAgO model to define agent / artifact interaction.

An artifact in simpA is modelled as a reactive entity exposing an interface with a basic set of operations. As described in previous sections, such operations have no return value: the information flow from an artifact to its environment (agents) is modelled through events explicitly generated by the artifact during operation execution.

In simpA, a new artifact (template) can be defined by extending the `alice.simpa.Artifact` class provided in the library `alice.simpa`. The operations supported by the artifact can be programmed as methods of the class, with no return value and with the method parameters representing operation parameters. In the body of the operations (methods), the basic CArtAgO primitives for event generation, operation management and state exposition can be exploited, available as protected method of the `alice.simpa.Artifact` class. The private fields of the class can be exploited to model artifact hidden state.

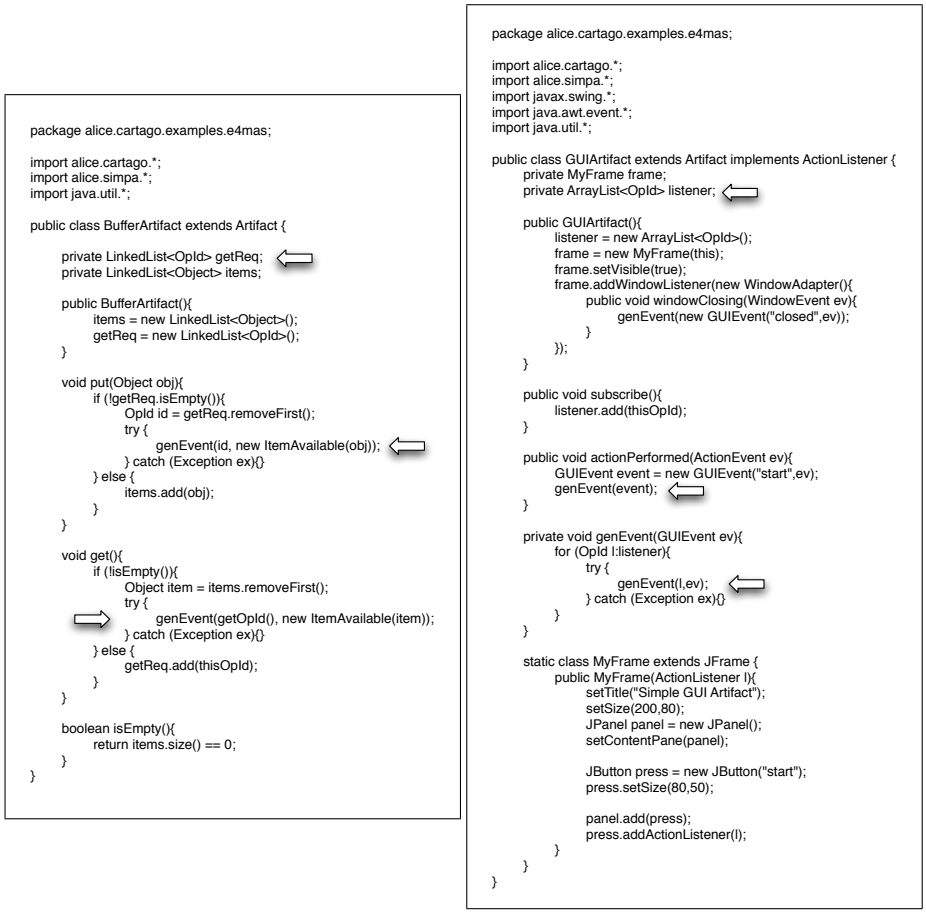
Currently, the concurrency model adopted for artifacts constrain operation execution requests to be served sequentially, i.e. only one operation at a time can be in execution on an artifact. Such a choice—which resembles to the strategy adopted for the monitor abstraction in the context of concurrent programming, with the difference that in our case the control flow of the invoker is not blocked—is quite effective in avoiding basic problems related to concurrent use of artifacts by agents (and in particular concurrent updates of artifact internal state). At the same time, this choice limits quite strongly the concurrency in artifact use: so, in the future we plan to relax this constraint, by allowing multiple operations to be in execution at a time on an artifact, relying on basic constructs for defining critical sections (such as synchronisation blocks in Java) in order to avoid inconsistencies.

Figure 3 (*left*) shows a simple example of artifact, a shared buffer that will be used in next subsection. The description of artifact structure and behaviour is reported in the caption of the figure.

It is worth remarking here that this approach makes it simple and effective to wrap and reuse any kind of object (resource, services) as artifact, so as to be suitably exploited by the agents. For instance, even GUIs can be suitable modelled as artifacts, mediating the interactions between human agents and software agents, both acting upon such artifacts and observing events generated by them. Figure 3 (*right*) shows a simple GUI—with a single button—modelled as an artifact: by invoking the `subscribe` operation, agents register themselves to be notified with the events occurring in the GUI artifact, sensed as perceptions.

As another example, a Web Service can be wrapped as an artifact, with the usage operations wrapping the operations provided by the service. So we think that this is an effective solution to bridge the agent and service / object worlds, keeping the agent level of abstraction and, in particular, encapsulation of control.

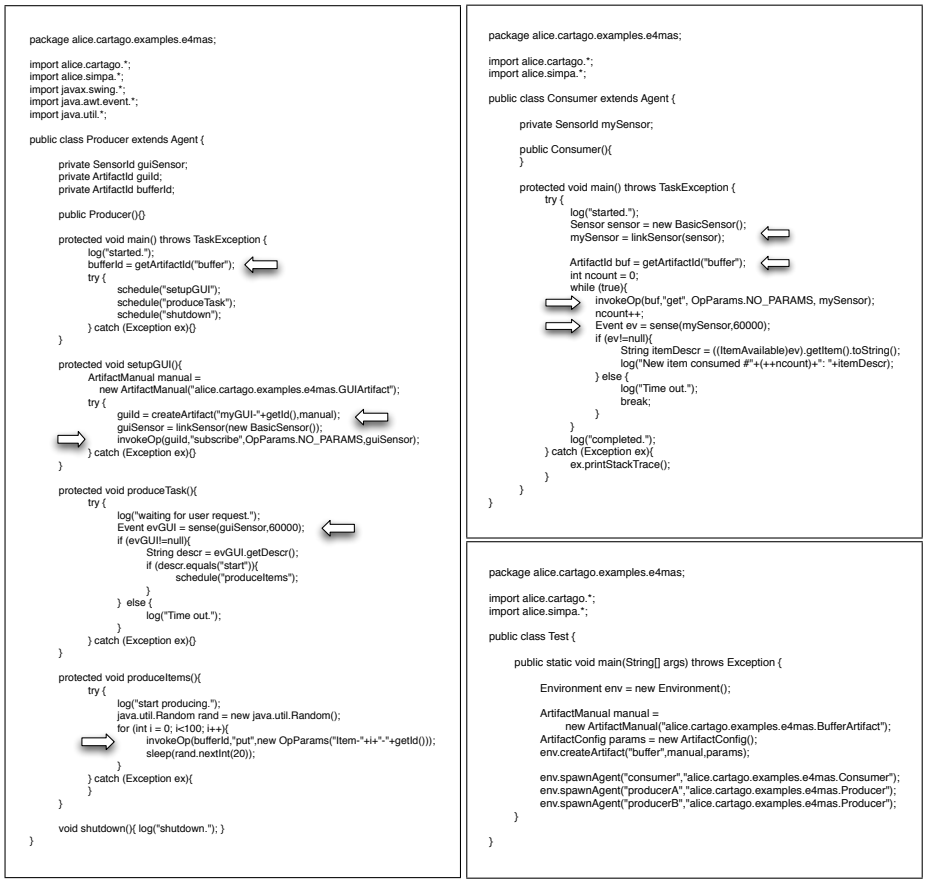
On the other side, an agent in simpA is modelled as a pro-active entity autonomously executing the set of tasks defined by the agent programmer. A simpA agent can interact with its social and computational environment either by means of direct communication with other agents—using a simple speech act-like model—, or



```
package alice.cartago.examples.e4mas;

import alice.cartago.*;
import alice.simpa.*;
import java.util.*;

public class BufferArtifact extends Artifact {

    private LinkedList<OpId> getReq;
    private LinkedList<Object> items;

    public BufferArtifact(){
        items = new LinkedList<Object>();
        getReq = new LinkedList<OpId>();
    }

    void put(Object obj){
        if (!getReq.isEmpty()){
            OpId id = getReq.removeFirst();
            try {
                genEvent(id, new ItemAvailable(obj));
            } catch (Exception ex){}
        } else {
            items.add(obj);
        }
    }

    void get(){
        if (!isEmpty()){
            Object item = items.removeFirst();
            try {
                genEvent(getOpId(), new ItemAvailable(item));
            } catch (Exception ex){}
        } else {
            getReq.add(thisOpId);
        }
    }

    boolean isEmpty(){
        return items.size() == 0;
    }
}
```

```
package alice.cartago.examples.e4mas;

import alice.cartago.*;
import alice.simpa.*;
import javax.swing.*;
import java.awt.event.*;
import java.util.*;

public class GUIArtifact extends Artifact implements ActionListener {
    private MyFrame frame;
    private ArrayList<OpId> listener;

    public GUIArtifact(){
        listener = new ArrayList<OpId>();
        frame = new MyFrame(this);
        frame.setVisible(true);
        frame.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent ev){
                genEvent(new GUIEvent("closed",ev));
            }
        });
    }

    public void subscribe(){
        listener.add(thisOpId);
    }

    public void actionPerformed(ActionEvent ev){
        GUIEvent event = new GUIEvent("start",ev);
        genEvent(event);
    }

    private void genEvent(GUIEvent ev){
        for (OpId l:listener){
            try {
                genEvent(l,ev);
            } catch (Exception ex){}
        }
    }

    static class MyFrame extends JFrame {
        public MyFrame(ActionListener l){
            setTitle("Simple GUI Artifact");
            setSize(200,80);
            JPanel panel = new JPanel();
            setContentPane(panel);

            JButton press = new JButton("start");
            press.setSize(80,50);

            panel.add(press);
            press.addActionListener(l);
        }
    }
}
```

**Fig. 3.** A buffer artifact *(left)* and a GUI artifact *(right)*, programmed using the simpA model, on top of CArtAgO. The buffer usage interface is composed by the `put` and `get` operations, to insert and remove an item. In the `put` operation, the item passed as parameter is inserted in `enqueued` only if no pending requests are present, collected by the `getReq` list. If there is a pending request, then the item is dispatched to the requestor through the generation of an event. In the `get` operation, if an item is available, then it is dispatched to the requester by generating an event; conversely, the request is enqueued, by inserting the operation id of current request in `getReq`. For what concerns the GUI Artifact, it is worth noting the `subscribe` operation is used to register agents to be notified for events occurring inside the artifact, internally generated by the Swing event-listener mechanism, but externally made observable to subscribed agents through `genEvent`.



```
package alice.cartago.examples.e4mas;

import alice.cartago.*;
import alice.simpa.*;
import javax.swing.*;
import java.awt.event.*;
import java.util.*;

public class Producer extends Agent {

        private SensorId guiSensor;
        private ArtifactId guiId;
        private ArtifactId bufferId;

        public Producer(){}

        protected void main() throws TaskException {
                log("started.");
                bufferId = getArtifactId("buffer");
                try {
                        schedule("setupGUI");
                        schedule("produceTask");
                        schedule("shutdown");
                } catch (Exception ex){}
        }

        protected void setupGUI(){
                ArtifactManual manual =
                   new ArtifactManual("alice.cartago.examples.e4mas.GUIArtifact");
                try {
                        guiId = createArtifact("myGUI-"+getId(),manual);
                        guiSensor = linkSensor(new BasicSensor());
                        invokeOp(guiId,"subscribe",OpParams.NO_PARAMS,guiSensor);
                } catch (Exception ex){}
        }

        protected void produceTask(){
                try {
                        log("waiting for user request.");
                        Event evGUI = sense(guiSensor,60000);
                        if (evGUI!=null){
                                String descr = evGUI.getDescr();
                                if (descr.equals("start")){
                                        schedule("produceItems");
                                }
                        } else {
                                log("Time out.");
                        }
                } catch (Exception ex){}
        }

        protected void produceItems(){
                try {
                        log("start producing.");
                        java.util.Random rand = new java.util.Random();
                        for (int i = 0; i<100; i++){
                                invokeOp(bufferId,"put",new OpParams("Item-"+i+"-"+getId()));
                                sleep(rand.nextInt(20));
                        }
                } catch (Exception ex){
                }
        }

        void shutdown(){ log("shutdown."); }
}
```

```
package alice.cartago.examples.e4mas;

import alice.cartago.*;
import alice.simpa.*;

public class Consumer extends Agent {

        private SensorId mySensor;

        public Consumer(){
        }

        protected void main() throws TaskException {
                try {
                        log("started.");
                        Sensor sensor = new BasicSensor();
                        mySensor = linkSensor(sensor);

                        ArtifactId buf = getArtifactId("buffer");
                        int ncount = 0;
                        while (true){
                                invokeOp(buf,"get", OpParams.NO_PARAMS, mySensor);
                                ncount++;
                                Event ev = sense(mySensor,60000);
                                if (ev!=null){
                                        String itemDescr = ((ItemAvailable)ev).getItem().toString();
                                        log("New item consumed #"+(++ncount)+": "+itemDescr);
                                } else {
                                        log("Time out.");
                                        break;
                                }
                        }
                        log("completed.");
                } catch (Exception ex){
                        ex.printStackTrace();
                }
        }
}
```

```
package alice.cartago.examples.e4mas;

import alice.cartago.*;
import alice.simpa.*;

public class Test {

        public static void main(String[] args) throws Exception {

                Environment env = new Environment();

                ArtifactManual manual =
                        new ArtifactManual("alice.cartago.examples.e4mas.BufferArtifact");
                ArtifactConfig params = new ArtifactConfig();
                env.createArtifact("buffer",manual,params);

                env.spawnAgent("consumer","alice.cartago.examples.e4mas.Consumer");
                env.spawnAgent("producerA","alice.cartago.examples.e4mas.Producer");
                env.spawnAgent("producerB","alice.cartago.examples.e4mas.Producer");
        }

}
```

**Fig. 4.** A producer agent *(left)* and a consumer agent *(right)* in simpA. The behaviour of the producer agents is composed by the following tasks, scheduled to be executed sequentially: first, the agent creates the GUI artifact for interacting with the human user, by exploiting the CArtAgO `createArtifact` service (API); then, when the human user presses the button on the GUI, this event is perceived by the agent that starts to produce and insert items in the buffer artifact, shared with the consumer, by invoking through the `invokeOp` API the `put` operation belonging to the buffer usage interface. Finally, after having inserted 100 items, the agent shutdowns. On the other side, a consumer agent simply collects the items from the buffer, by invoking the `get` operation and processing the information on items embedded in events generated by the buffer and perceived by the agent.

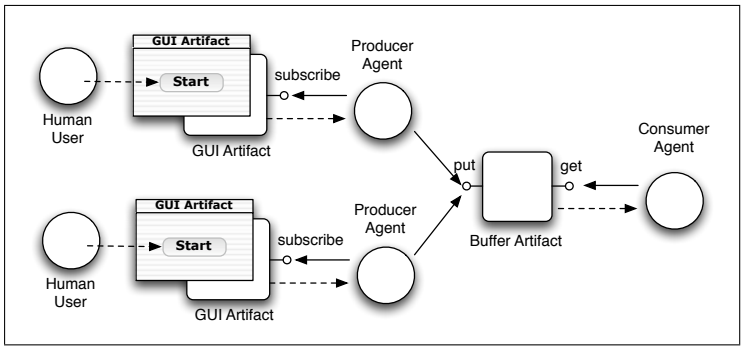



**Fig. 5.** Agents and artifacts involved in the application described in Subsection 4.2

by means of pragmatic actions as defined in CArtAgO, to create, use, and inspect artifacts. A new agent (template) can be defined by extending the `alice.simpa.Agent` class provided in the library `alice.simpa`. Actually, such a class contains all the machinery that makes it possible to encapsulate the control flows to execute agent tasks, realising agent pro-activity. Classes adopted for representing simpA agents are meant to have no public interface. Agent tasks are specified as protected methods of the class: in such methods the programmer explicitly defines the procedure—as sequence of actions—to run when the execution of the task is requested. The basic set of native actions available to the agent—implementing the basic abstract API described in Sect. 3—are provided as protected method of the `alice.simpa.Agent` class. Such actions include both internal and external actions. The former are mainly used to manage agent agenda, for scheduling new tasks to execute, removing scheduled tasks, spawning the execution of parallel tasks, and so on. The latter are mainly used to speak with other agents and to exploit CArtAgO core primitives, in particular to create artifacts, invoke artifact operations, sense events, and so on. The private field of the class can be used to model agent hidden state.

As an example, Fig. 4 *(left)* shows a simple producer agent using the GUI and the buffer artifacts described previously. Basically, the agent creates an instance of the GUI artifact to interact with the user. When the human user presses the button, the agent senses the event and starts using the buffer, by inserting some information items. Figure 4 *(right)* shows a simple consumer agent, observing and retrieving information from the buffer, and logging such items in standard output. Figure 4 *(bottom-right)* shows the main details of a simple application (graphically represented in Fig. 5), where two producers and one consumer are spawned, accessing the same buffer artifact, created in the main.

## 5 Related Works

The approach based on artifacts shares the same software engineering perspective introduced by Weyns and colleagues in [21], where they identify a general model and an architecture that can be (re-)used to engineer environments in MAS, despite of the specific application domain. The model presented by the author is *concern-based*:

the environment is modelled as a set of modules that represent different functional concerns of the environment. A similar focus, but in some sense less general, can be found also in the work of Platon and collegues [22], where a general model for environments providing functionalities for *over-hearing* and *over-sensing* is presented. Our notion of artifact can be compared at a first glance with the notion of functional modules describe by Weyns and colleagues. The main difference is that artifacts are conceived to be first-class abstractions both for the engineers designing and programming agent environment *and for the agents using such an environment*: agents do not perceive the environment as a single entity providing a set of functionalities (which are internally engineered upon a set of modules), but directly create, share, use, manipulate, destroy artifacts, each designed to encapsulate some kind of function.

The model for perception and sensing described in the paper shares many points with the model—more general—discussed in [23], introducing the notion of *active perceptions*. Such a model decomposes perceptions into a succession of three functionalities: sensing, interpreting and filtering. First, sensing maps the state of the environment to a representation. The agent can select a set of *foci*, that enable the agent to sense specific type of data in the environment. The representation of the state is composed according to a set of *perception laws*, that can be used by designers to enforce specific constraints on perceptions. Then, agents interpret representations by means of *descriptions*, that are blueprints that map representation onto percepts, modelled as expressions that can be understood by the internal machinery of the agent. Finally, agents can select a set of *filters*, to restrict the perceived data according to specific context relevant selection criteria.

In our model, *sensors* provide some of the functionalities discussed above. In particular, by following the meaning introduced by the authors, each sensor can be used as a specific focus: the idea is that an agent can dynamically create, customise and use different kind of sensors, with distinct features (such as buffering, filtering, etc), to partition the perceptions from the environment, in our case related to artifacts (even if the model can be extended to consider also perceptions directly related to other agents). Sensor activity can be constrained according to laws enforced by the organisational and physical context where the agent is situated, similar to perceptual laws discussed above: in our approach such rules are meant to be embedded and enforced by the *agent contexts*—where sensors are collected, described in subsection 4.1—, including policies that constrain agent action space, i.e. set of actions that are allowed for an agent playing some specific role(s). Pattern-driven sensing described in Sect. 3 can be framed as a simplified form of filtering as defined in active perceptions, with some points that concern also interpretation: patterns act as simple filters that agents can specify to fetch in a data-driven way the data collected by sensors, and require that an explicit description is adopted for describing the events or stimuli posted to sensors.

Finally, the artifact abstraction and CArtAgO infrastructure draw on the research work on *tuple centres* as programmable tuple-based coordination media [24] and on TuCSoN coordination infrastructure [25]. Artifacts can be framed as a generalisation of the notion of tuple centre: more precisely, tuple centres can be conceived as a type of *coordination artifacts* [5], as artifacts designed to encapsulate programmable coordination services.

## 6 Concluding Remarks

In this paper we first described in detail the abstract model and architecture of a basic infrastructure for supporting artifacts in MAS, and then we provided a first basic prototype implementing some core functionalities.

Among the issues not considered for lack of space—and that can be found in the artifact conceptual framework—we mention here: *(i)* artifact *composition*—support for linking together existing artifacts to dynamically compose complex artifacts, by defining and exploiting artifact *link interfaces*; *(ii)* artifact *management*—support for inspecting, controlling, testing artifact state and behaviour, by defining and exploiting artifacts *management interface*, besides usage interface.

Among the main points of a possible roadmap for the development of the project CArtAgO, we consider important: *(i)* improving the development of the prototype, supporting all the features presented in the abstract model, by implementing—in particular—a first support for workspace and workplace, i.e. topology and organisation / security, as described in the paper; *(ii)* modelling and integrating existing services as kind of artifacts, in order to be easily re-used when engineering applications on top of CArtAgO. Two examples are: artifacts wrapping TuCSoN tuple centres, providing agent coordination facilities, and artifacts wrapping Web Services, to raise agent interaction with Web Services at the artifact level; *(iii)* establishing first models and ontology for defining function descriptions, operating instructions, and observable state description, possibly reusing existing research efforts on service description models and (standard) languages, such as OWL-S.

Finally, existing and ongoing research in environment for MAS will be important to improve the theoretical foundation of CArtAgO, concerning the notion of artifact and related concepts: for instance, the research work on active perceptions can be important to improve and extend the model of sensing currently adopted.

## References


1. Ricci, A., Viroli, M., Omicini, A.: Programming MAS with artifacts. In Bordini, R.P., Dastani, M., Dix, J., El Fallah Seghrouchni, A., eds.: 3rd International Workshop "Programming Multi-Agent Systems" (PROMAS 2005), AAMAS 2005, Utrecht, The Netherlands (2005) 163–178
2. Viroli, M., Omicini, A., Ricci, A.: Engineering MAS environment with artifacts. In Weyns, D., Parunak, H.V.D., Michel, F., eds.: 2nd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2005), AAMAS 2005, Utrecht, The Netherlands (2005) 62–77
3. Weyns, D., Parunak, V.D., Michel, F., Holvoet, T., Ferber, J.: Environments for multiagent systems, state-of-the-art and research challenges. In: Environments for Multiagent Systems. Volume 3374 of LNCS., Springer Verlag (2005)
4. Amant, R.S., Wood, A.B.: Tool use for autonomous agents. In Veloso, M.M., Kambhampati, S., eds.: AAAI/IAAI'05 Conference, Pittsburgh, PA, USA, AAAI Press / The MIT Press (2005) 184–189
5. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In: AAMAS'04. Volume 1., New York, USA, ACM (2004) 286–293
6. Nardi, B.A.: Context and Consciousness: Activity Theory and Human-Computer Interaction. MIT Press (1996)

7. Kirsh, D.: Distributed cognition, coordination and environment design. In: European conference on Cognitive Science. (1999) 1–11
8. Agre, P.: Computational research on interaction and agency. Artificial Intelligence **72** (1995) 1–52
9. Agre, P., Horswill, I.: Lifeworld analysis. Journal of Artificial Intelligence Reserach **6** (1997) 111–145
10. Dourish, P.: Where the action is. The MIT Press (2001)
11. Gasser, L.: MAS infrastructure: Definitions, needs, and prospects. In Wagner, T., Rana, O., eds.: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems. Volume 1887 of LNAI. Springer (2001) 1–11
12. Sycara, K., Paolucci, M., van Velsen, M., Giampapa, J.: The RETSINA MAS infrastructure. Autonomous Agents and Multi-Agent Systems **7** (2003)
13. Bellifemine, F., Poggi, A., Rimassa, G.: JADE: a FIPA2000 compliant agent development environment. In: 5th International Conference on Autonomous Agents (Agents 2001), Montreal, Quebec, Canada, ACM Press (2001) 216–217
14. Russel, S., Norvig, P.: Artificial Intelligence. A Modern Approach. 2nd edn. Prentice All, New Jersey (2003)
15. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley & Sons (2000)
16. Viroli, M., Ricci, A.: Instructions-based semantics of agent mediated interaction. In: AAMAS'04. Volume 1., New York, USA, ACM (2004) 286–293
17. Ferber, J., Gutknecht, O.: A meta-model for analysis and design of organizations in multi-agent systems. In: Proceedings of ICMAS '98, IEEE Press (1998)
18. Esteva, M., Rosell, B., Rodríguez-Aguilar, J.A., Arcos, J.L.: Ameli: An agent-based middleware for electronic institutions. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: AAMAS 2004. Volume 1., New York, USA, ACM (2004) 236–243
19. Omicini, A., Ricci, A., Viroli, M.: RBAC for organisation and security in an agent coordination infrastructure. Electronic Notes in Theoretical Computer Science **128** (2005) 65–85
20. Sandhu, R., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based control models. IEEE Computer **29** (1996) 38–47
21. Weyns, D., Holvoet, T.: Formal model for situated multiagent systems. Fundamenta Informaticae **63** (2004) 125–158
22. Platon, E.e.a.: Oversensing with a softbody in the environment: Another dimension of observation. In G. Kaminka, G., Pynadath, D., Geib, C., eds.: Proceedings of the "Modeling Others from Observation" Workshop, International Joint Conference on Artificial Intelligence, Edinburgh, Scotland (2005)
23. Weyns, D., Steegmans, E., Holvoet, T.: Towards active perception in situated multiagent systems. Applied Artificial Intelligence **18** (2004) 867–883
24. Omicini, A., Denti, E.: From tuple spaces to tuple centres. Science of Computer Programming **41** (2001) 277–294
25. Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems **2** (1999) 251–269