CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS

Alessandro Ricci, Mirko Viroli, and Andrea Omicini

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

Abstract. This paper describes CArtAgO, a framework for developing artifact-based working environments for multiagent systems (MAS). The framework is based on the notion of *artifact*, as a basic abstraction to model and engineer objects, resources and tools designed to be used and manipulated by agents at run-time to support their working activities, in particular the cooperative ones. CArtAgO enables MAS engineers to design and develop suitable artifacts, and to extend existing agent platforms with the possibility to create artifact-based working environments, programming agents to exploit them. In this paper, first the abstract model and architecture of CArtAgO is described, then a first Java-based prototype technology is discussed.

1 Introduction

Artifacts have been recently proposed as first-class abstractions to model and engineer agent *working environments* in software MAS (multiagent systems) [1]. The background view, shared with other recent approaches in MAS literature—see [2,3] for a survey—, is that the environment plays a fundamental role in engineering of MAS. On the one hand, environment is a suitable locus for engineers to embed responsibilities, impacting on MAS design and development; on the other hand, it is a source of structures and services that agents can suitably use at run-time to support and improve their activities—both individual and social ones. The specific notion of *working environment* is intentionally analogous to the notion of human cooperative working environments, as they are studied by disciplines and theories in human science, such as Activity Theory and Distributed Cognition, and recently adopted also in the context of CSCW (Computer Supported Cooperative Work) and HCI (Human-Computer Interaction) [4,5]. There, a working environment—also referred as *field of work*—is such part of the environment explicitly designed to support and realise agent working activities. Typically, it is modelled as set of objects, tools, more generally "artifacts", which are constructed, shared, and either cooperatively (or competitively) used by humans, so as to mediate and sustain their activities.

Analogously to human society, such a perspective is likely to be fundamental also in the context of agent societies, in particular for designing and programming

D. Weyns, H.V.D. Parunak, and F. Michel (Eds.): E4MAS 2006, LNAI 4389, pp. 67–86, 2007.

complex software systems based on MAS. Given that MAS are growing increasingly complex, one may easily foresee that the next step in the evolution of cognitive MAS will require MAS models and architectures to deal with agents situated within suitable working environments. There, agents would autonomously construct, share, and co-operatively use different kinds of artifact—designed either by MAS designers or by the agents themselves—to perform MAS activities. It is worth noticing that such a perspective shares the aims and principles developed by the research work in Distributed Artificial Intelligence about theories of interaction, environments and the role of tools [6,7].

The artifact abstraction is at the heart of this conceptual framework—which can be referred as A&A (agents and artifacts)—and promotes a methodology for modelling and engineering working environments, by introducing new concepts and elements that impact on system design, development and run-time management. Artifacts can be generally conceived as passive, *function-oriented* computational entities, explicitly designed to provide some kind of *function*, and then to be *used* by agents 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 [7], that is, the purpose for which the object 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. Artifacts could be either the targets (outcome) of agent activities, or the *tools* that agents use as means to support such activities: as such, they are useful to reduce complexity of task execution. For instance, *coordination artifacts* [8] are artifacts providing coordination functionalities—such as blackboards, tuple spaces or workflow engines.

In this paper we introduce and discuss CArtAgO (Common "Artifacts for Agents" Open framework), a framework for prototyping MAS applications with artifact-based working environments. Essentially, CArtAgO provides *(i)* the API to define any useful kind of artifacts, *(ii)* the API to be exploited by agents (agent programmers) for interacting with working environments populated by artifacts—in particular to instantiate, use, manipulate artifacts—, and *(iii)* a run-time environment supporting the existence and dynamic management of working environments. CArtAgO does not introduce any specific model or architecture for agents and agent societies: the framework is meant to be integrated and used with existing agent platforms, possibly characterised by heterogeneous kinds of agent architectures. From a conceptual point of view, CArtAgO makes it possible to build MAS composed by heterogeneous agent societies, made of reactive and cognitive agents programmed with different agent languages or architectures, sharing the same working environments, and interacting through suitable mediating artifacts—besides communicating via ACL as usual.

The rest of the paper is organised as follows: first, we describe the abstract model and architecture of CArtAgO (Sect. 2), focusing in particular on the core of API introduced by the framework (Sect. 3); then, we describe a first concrete implementation prototype (Sect. 4) developed in Java, implementing the core part of the abstract model previously defined.

2 CArtAgO Abstract Model and Architecture

In this section we describe the basic elements and structure of CArtAgO working environments, by taking as a reference the abstract architecture schema described in [3] and depicted in Fig. 1, useful to understand CArtAgO with respect to the other approaches. Accordingly, the abstract architecture of CArtAgO (and of CArtAgO working environments) is composed by three main building blocks (see Fig. 2): *(i) agent bodies*—as the entities that make is possible to *situate* agents inside the working environment; *(ii) artifacts*—as the basic building blocks to structure the working environment; and *(iii) workspaces*—as the logical containers of artifacts, useful to define the topology of the working environment.

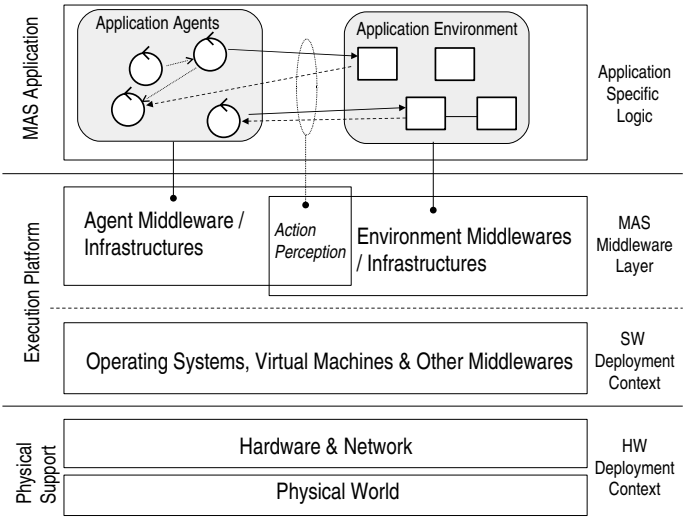

Fig. 1. Abstract representation of MAS layers with environment-based supports as depicted in [3]. Rectangles represent S/W and H/W tiers of the application at different levels. Agents are expressed as circles, environment abstractions as boxes. Arrows from agents to environment abstractions represent actions, dashed arrows in the opposite direction represent perceptions. Arrows between agents represent direct agent communications, while arrows between environment abstractions represent intra-environment interactions. Vertical lines represent the infrastructure supporting a concept at the MAS application level.

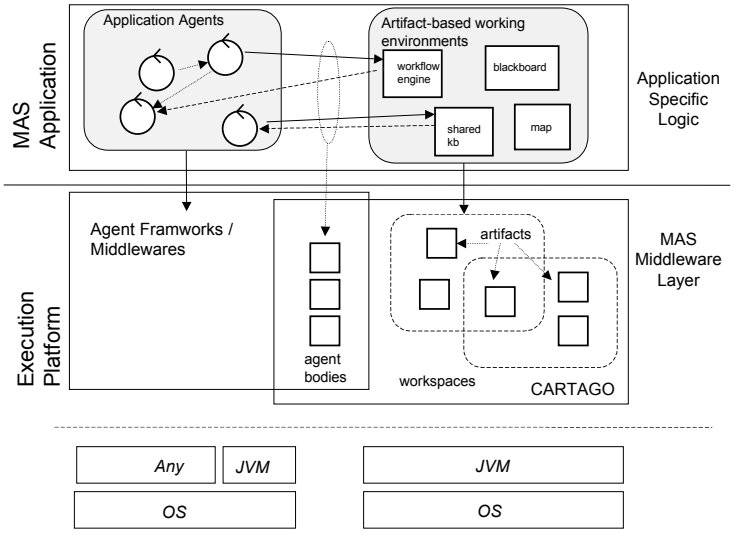

Fig. 2. MAS layers adopting CArtAgO support. Application environments are modelled in terms of artifact-based working environments. The CArtAgO middleware manage the life-cycle of working environments, composed by artifacts grouped in workspaces. Agent bodies are used to situate agents inside the working environments, executing actions upon artifact and perceiving artifacts observable state and events.

2.1 Agent Bodies

Agent bodies are what actually enable the coupling between an agent (mind) and a CArtAgO working environment. For each agent aiming at working inside a CArtAgO environment, an agent body is created. The agent body contains *effectors* to perform actions upon the working environment, and a dynamic set of *sensors* to collect stimuli from the working environment. The agent body is meant to be controlled by the agent, which actually plays the role of the "pilot" of the body. For the purpose, the agent body exposes a controlling interface that the agent mind could suitably exploit to interact with the environment.

By piloting their agent bodies, agents can interact with their working environment, executing *actions* provided for artifact construction, selection and usage, and perceiving *observable events* generated from such artifacts. Differently from the approach typically adopted in traditional agent architectures and more similarly to active perception [9], here perception is modelled as an intentional action referred as *sensing*. More precisely, environment observable events—generated by artifacts—are collected as stimuli by *sensors* which are part of the agent body. An agent can dynamically and flexibly link and unlink to its body different kinds of sensor, with different functionalities, such as buffering, filtering, ordering, and managing priorities. So, in CArtAgO sensing is the internal action that agents execute on their sensors to become aware (perceive) of the *stimuli* collected by the sensors. Stimuli typically concern observable events generated by artifacts.

2.2 Artifacts

Artifacts are the basic bricks managed by CArtAgO framework. Each artifact has a logic *name* specified by the artifact creator at instantiation-time, and an *id*, released by the framework, 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.

Usage Interface & Observable Events. Analogously to artifacts in our society, the basic model which characterises the interaction between agents and artifacts is based on a notion of use and observation. Agents can use an artifact by triggering the execution of operations listed in the artifact *usage interface*. An operation is characterised by a name and a set of typed parameters. The execution of an operation typically causes the update of the internal state of an artifact, and possibly the generation of one or multiple *observable events*— including error conditions— that can be possibly collected by agents sensor as they are generated, and perceived by agents through explicit sensing actions.

The usage interface of an artifact can change according to artifact *observable state*, exposing different sets of operations according to the specific functioning state of the artifact. The notion of observable state is adopted to structure the functioning behaviour of an artifact in a set of labelled states, which can be recognised (observed) by the artifact users. For each artifact type a finite set of labelled observable states can be defined. For each concrete instance, the notion of current observable state is defined, and its value can change dynamically, during artifact functioning. Then, for each observable state a different usage interface can be defined. This feature makes it possible to structure the overall usage interface of an artifact, providing the right interface according to the functioning stage of the artifact. In other words, an artifact can expose different set of operations according to its observable state.

Dynamically, an agent can trigger the execution of an operation on an artifact if and only if the operation is (currently) part of the usage interface; if the operation does not belong to the usage interface, the agent action fails.

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, 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: we forward the interested reader to [10] for the first results.

In CArtAgO, we provide a minimal enabling support to such issues, by modelling function description and operating instructions as flat strings, specified by artifact designers and dynamically inspectable (observable) by agents through suitable actions. Currently, there is no predefined syntax and semantics for such information (see future work for comments on this point).

2.3 Workspaces

Artifacts are logically located within *workspaces*, which can be used to define the topology of the working environment. A workspace can be defined as an open set of artifacts and agents creating and using them: artifacts can be dynamically added to or removed from workspaces, agents can dynamically enter (join) or exit workspaces. The same artifact can belong to multiple workspaces.

In CArtAgO, each workspace is created by specifying a logic name and is univocally identified by an id. By defining a topology of the environment, workspaces make it possible to structure agents and artifacts organisation and interaction, in particular functioning as scopes for event generation and perception, and artifact access and use. 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.

3 Core Primitives

After providing an overview of CArtAgO main components, in this section we describe the basic abstract set of core API provided by the framework on the one side to be used by agents (or agent programmers defining agent behaviour) to interact within working environments, and on the other side for defining artifact types, that is programming artifacts behaviour.

3.1 Agent Side

On the agent side, the API is represented by a set of primitives to control agent bodies and that eventually result in executing actions inside the working environment, making it possible basically to create and use artifacts, and perceive artifact observable state and events. Table 1 provides an abstract description of such primitives, grouped according to their functionalities:

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

Artifact construction and disposal	`createArtifact(Name,Template,Config,{WsID}):ArID` `disposeArtifact(ArID)`

Artifact selection & use	`getArtifactID(Name,{WsID}):ArID`
	`execOp(ArID,OpName,{Args},{SensorID})`
	`sense({SensorID},{Pattern},{Timeout}):Perception`
	`focus(ArID,SensorID)`
	`unfocus(ArID,SensorID)`

Artifacts inspection	`getFD(ArID): FDDescr`
	`getOI(ArID): OIDescr`
	`getUID(ArID): UIDDescr`
	`getState(ArID): StateDescr`

Sensor management	`linkSensor(SensorType,SensorConfig): SensorID` `unlinkSensor({SensorID})`

Workspaces management	`getWsID(WsName):WsID`
	`createWS(WsName):WsID`
	`disposeWS(WsID)`
	`registerArtifact(ArID,WsID)`
	`deregisterArtifact(ArID,WsID)`
	`joinWS(WsID)`
	`exitWS(WsID)`

Artifacts construction & disposal — 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, the initial configuration parameters needed for artifact creation and optionally the workspace where the artifact should be created. The action can fail if the template is unknown or the artifact instantiation is not completed due to some kind of problem (e.g. wrong initial configuration).

Artifact discovery & use — These primitives constitute the core of agent / artifact interactions, enabling an agent to use an artifact by executing operations and observing artifact state and events. To execute an operation, the action `execOp` is provided, specifying the artifact identifier, the operation name, the parameters, and (optionally) the specific sensor where to collect observable events generated by the artifact as a consequence of the operation execution. The action can fail either because the specified artifact is not available, or because the operation cannot be executed since it is not part of artifact usage interface. Action success means that the execution of the specified operation has been successfully triggered. The identifier of an existing artifact can be obtained by the `getArtifactID` primitive, specifying the artifact name and (possibly) its location (workspace).

After triggering the operation, an agent can observe related events through codesense primitives on the sensor specified in `execOp`. By executing sense

actions, an agent is made aware of the stimuli that are dynamically collected by a sensor. In particular, the effect of the action is to fetch (remove) a stimulus from the sensor and to return it to the agent as a perception. The action fails if no stimuli are available. Different types of sensors can provide different semantics establishing the order in which events are fetched. A time parameter can be optionally provided to indicate the duration for the sensing action: if no events are available in the sensor within the specified time-frame, the action fails. By default, the time-frame is zero.

In order to support forms of *data-driven* (or, equivalently, *filter-driven*) sensing, a pattern parameter can be specified acting as a filter for fetching (selecting) the perception. Conceptually, the pattern defines a set of perceptions: a perception is fetched if and only if is included in the set. Typically, the pattern can be represented by a Boolean function, establishing—given a perception—if either it is part or not of such a set. It is worth noting that specifying a pattern in a sense action is different from creating sensors that filter stimuli as they are collected. An available stimulus which is not fetched by a sense action because not satisfying the pattern is not removed from the sensor and can be possibly fetched by subsequent sense actions.

Finally, primitives for continuous observation are provided: by executing a `focus` action, an agent becomes a permanent observer of the artifact whose identifier is specified as a parameter. As a permanent observer, all the observable events generated by the artifact are automatically collected by the sensor specified as second parameter, as they are generated; `unfocus` stops the observation.

Artifacts 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.

Sensor management — Two basic primitives are provided to dynamically link and unlink sensors to the agent body: `linkSensor` links a sensor of the specified type and configuration to the body, returning the identifier to be used to refer the sensor; `unlinkSensor` unlinks a previously linked sensor.

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, `createWS` for directly creating a new workspace and `disposeWS` for completely removing a workspace.

Since the same artifact can be part of multiple workspaces, some basic primitives are provided to register (`registerArtifact`) / de-register (`deregisterArtifact`) an artifact in / from a workspace, specifying the workspace id.

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

Table 2. Basic primitives for artifact programming

Observable event generation	`genEvent({OpID},EventType,{EventContent})` `genEventInWsp(EventType,{EventContent})`
Operation management	`getOpID: OpID`
Observable state management	`setObservableState(ObsStateName)` `getObservableState:ObsStateName`

3.2 Artifact Side

On the artifact side, CArtAgO provides a support to define new types of artifact, defining artifact structure and behaviour. The specific programming model adopted to implement in Java artifact types is described in detail in Sect. 4. Here we report the basic set of abstract primitives which can be exploited when defining artifact behaviour (see Table 2), useful essentially for generating observable events and switching artifact observable state.

Observable events can be generated as either related or not to the specific execution instance of an operation. For the purpose, each operation triggered on the artifact is labelled by a unique operation identifier (type `OpId` in the tables). Such an identifier can be explicitly retrieved by the `getOpId` primitive 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 (this aspect will be clarified by a concrete example described in Subsection 4.2).

An event can be then generated using the `genEvent` primitive by specifying the operation identifier to which the event must 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 triggered. The effect of the execution of these primitives is the generation of an event which is eventually collected by the sensor (if specified) of the agent that triggered the operation and by all the agents that are observing—via focus—the artifact. To generate event unrelated to a specific operation execution, the primitive `getEventInWsp` is provided, which generates an event which is observed by all the agents focusing on the artifact.

Finally, a couple of primitives are provided to manage the current observable state of the artifact, in particular to set a new value with `setObservableState`, specifying a label identifying one of the possible set of observable states defined by the artifact type, and to retrieve current value with `getObservableState`.

Besides the events explicitly generated with the `genEvent` primitive, some other kinds of event are automatically generated by the framework and made observable to agents interacting with an artifact. In particular, an event is generated whenever the execution of an operation is completed, and whenever a new

observable state is set (details about the specific types of these events are provided when describing in next section).

4 A First Prototype

A first prototype implementing most of the functionalities described in the previous section has been developed in Java and is available for download at the CArtAgO project web site[1]. Our objective was to set up a first framework for prototyping and experimenting applications engineered upon the A&A metamodel, and so designed in terms of set of agents—possibly with heterogeneous models and architectures—situated in the same working environment, designed in terms of specific kind of artifacts. The framework itself is not meant to define or constrain the specific agent architecture adopted to define the behaviour of the individual agents: conversely, the framework is meant to be integrated and exploited with external agent frameworks or platforms, in particular with those that adopt Java as underlying implementation language, extending them so as to support the creation and use of artifact-based environment according to the A&A perspective.

As an example, simpA (simple A&A programming environment) is a full-fledged agent-oriented framework for prototyping general-purpose applications based on CArtAgO. Basically, simpA provides a support for developing MAS based on agents with an *activity-oriented* architecture, with a native support for creating and using artifact-based working environment, engineered upon CArtAgO. The interested reader is invite to refer to the simpA web site[2].

Based upon CArtAgO and simpA, simpA-WS is a framework for prototyping service-oriented application—in particular Web Service-based—in terms of agents and artifacts. There, artifacts are used on the client side as interfaces for user application agents to flexibly access and use Web Services, on the service side as interfaces for service agents to get Web-service messages and requests to be processed, and to provide responses. More information can be found at the simpA-WS web site[3]. Working in the first real-world application examples, simpA-WS is currently being investigated as an agent-based technology for prototyping service-oriented applications in the context of logistics [4].

4.1 Prototype Overview

The framework is composed by four main parts:

API for setup working environments — The entry point class of the framework is the `Cartago` class (sketched in Fig. 3), which mainly provides static services to create or get the reference to existing working environments

[1] CArtAgO web site: `http://www.alice.unibo.it/projects/cartago`
[2] simpA web site: `http://www.alice.unibo.it/projects/simpa`
[3] simpA-WS web site: `http://www.alice.unibo.it/projects/simpaws`
[4] `http://www.alice.unibo.it/projects/a4stil`

```
public class Cartago {

    public static synchronized ICartagoEnvironment
        getInstance(String name){...}
    public static synchronized ICartagoEnvironment
        getInstance(String name, ICartagoLoggerManager logger){...}
    public static String getVersion(){...}
}
```

Fig. 3. Entry point class for the CArtAgO framework. The class can be used to instantiate and get the reference to working environment.

```
public interface ICartagoEnvironment {

    IAgentBody getAgentBody(String name) throws AgentBodyAlreadyPresentException;
    ArtifactId createArtifact(String name, Class template, ArtifactConfig param)
                                                throws ArtifactAlreadyPresentException,
                                                       UnknownArtifactTemplateException,
                                                       ArtifactConfigurationFailedException;
    ArtifactId createArtifact(String name, Class template)
                                                throws ArtifactAlreadyPresentException,
                                                       UnknownArtifactTemplateException,
                                                       ArtifactConfigurationFailedException;
    void registerLogger(ICartagoLogger logger);
    void unregisterLogger(ICartagoLogger logger);
}
```

Fig. 4. Interface for working environments, providing services for creating agent bodies, and for directly creating artifacts, useful to setup the initial configuration of the environment

identified by a logic name. Once created or retrieved the reference to a working environment, it is possible to use the services provided by its interface—`ICartagoEnvironment`, sketched in Fig. 4—to setup the environment possibly creating an initial set of artifacts (besides the ones created dynamically by agents), and to create agent bodies, for enabling agents participation to the environment.

API for controlling agent bodies — From the agent point of view, the participation and interaction within a working environment takes place through an agent body. The creation of an agent body is provided as the `getAgentBody` provided by a working environment. Such a creation is typically done during agent initialisation. Once its agent body is created inside the environment, the agent—here conceived as the agent "mind"—can control it by suitably exploiting the `IAgentBody` interface implemented by the agent body, containing the core set of API described in Subsection 3.1. A sketch of the `IAgentBody` interface is reported in Fig. 5. It is possible to recognise the primitives for creating and disposing artifacts, for executing operations, sensing perceptions, managing sensors, and so on.

API for defining artifact types — A core part of the framework is given by the support provided to define new kind of artifacts, programming their structure and behaviour. We adopted a programming model that favours rapid prototyping of artifacts, exploiting as much as possible the support

```
public interface IAgentBody {

    ArtifactId createArtifact(String name, Class template, ArtifactConfig param)
                            throws ArtifactAlreadyPresentException,
                                     UnknownArtifactTemplateException,
                                     ArtifactConfigurationFailedException;
    ArtifactId createArtifact(String name, Class template)
                            throws ArtifactAlreadyPresentException,
                                   UnknownArtifactTemplateException,
                                   ArtifactConfigurationFailedException;
    void disposeArtifact(ArtifactId id) throws UnknownArtifactException;
    ArtifactId getArtifactId(String name) throws UnknownArtifactException;

    OpId execOp(ArtifactId id, Op op) throws OperationException;
    OpId execOp(ArtifactId id, Op op, SensorId sid) throws OperationException;

    Perception sense(SensorId sensorId)
                    throws NoPerceptionException;
    Perception sense(SensorId sensorId, IPerceptionFilter p)
                    throws NoPerceptionException;
    Perception sense(SensorId sensorId, int dt)
                    throws InterruptedException, NoPerceptionException;
    Perception sense(SensorId sensorId, IPerceptionFilter p, int dt)
                    throws InterruptedException, NoPerceptionException;

    void focus(ArtifactId aid, SensorId sid) throws SensorNotLinkedException;
    void unfocus(ArtifactId aid);

    SensorId linkSensor(AbstractSensor s);
    void unlinkSensor(SensorId id) throws CartagoException;
}
```

Fig. 5. Interface to control an agent body, including methods for triggering the execution of agent actions for artifact creation (`createArtifact`), artifact disposal (`disposeArtifact`), artifact discovery (`getArtifactId`), for triggering the execution of operation (`execOp`), for sensing perceptions (`sense`), for continuously observing artifacts (`focus`, `unfocus`), and for managing sensors (`linkSensor`,`unlinkSensor`)

```
public abstract class Artifact {
    ...
    protected final ArtifactId getId(){...}

    protected final OpId getOpId(){...}
    protected final OpRequestDescriptor getOpRequestDescriptor(){...}

    protected final void genEvent(String type) {...}
    protected final void genEvent(String type, Object content) {...}
    protected final void genEvent(OpId id, String type)
                                throws InvalidOpIdException {...}
    protected final void genEvent(OpId id, String type, Object content)
                                throws InvalidOpIdException {...}

    protected final void genEventInWsp(String type, Object content) {...}

    protected final void setObservableState(String state)
                                          throws UnknownArtifactStateException {...}
    protected final String getObservableState(){...}
}
```

Fig. 6. Base abstract class to define new artifact types. The basic set of primitives useful for programming artifact observable behaviour (in particular to generate observable events, to set and retrieve the observable state) are implemented as protected methods of this class.

given by the Java object-oriented environment. Accordingly, an artifact type can be defined by extending the basic `Artifact` class provided in the API: at run-time, artifacts instances are instances of this class. A sketch of the base class is shown in Fig. 6: the core set of the primitives described in Subsection 3.2 are available as protected methods provided by the class.

The artifact internal state is defined in terms of instance fields of the class, and the behaviour of operations can be defined by suitable instance methods of the class. In particular an operation `Op(Params)` can be implemented by a method of the kind:

```
@OPERATION(State1,State2,...) void Op(Params){...}
```

The annotation `@OPERATION`[5] is used to explicitly state that what follows is not to be interpreted as a normal method (meant to be invoked by other objects) but rather as the body of an artifact operation. It is worth remarking that methods representing operations have no return argument—a return argument would be meaningless in CArtAgO abstract model, as well as in the A&A general meta-model.

Currently, the concurrency model adopted for artifacts prevents operation execution requests to be served sequentially, so that only one operation at a time can be in execution on an artifact. Such a choice 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 future work will be devoted to explore further this issue.

As depicted in Fig. 6 and described in Subsection 3.2, observable events can be generated in the body of an operation by a family of primitives of the kind `genEvent`, specifying the event type, optionally an event content and the operation identifier to which the event must be related (`OpId` parameter). Events are collected by agent body sensors as stimuli, and then perceived by agents through `sense` action. Fig. 6 also includes the primitives that can be used to set and retrieve the current observable state of the artifact (`setObservableState` and `getObservableState`, respectively).

The manual of the artifact, containing information about function description, the operating instructions, as well as the list of the observable states, can be explicitly declared through the `@ARTIFACT_MANUAL` annotation preceding the artifact class declaration. If no states are declared, a single *default* state is defined. Defined the list of the observable states, an artifact programmer can specify the shape of the usage interface in relationship to the artifact observable state. This is possible by explicitly stating in the annotation of an operation what are the observable states in which the operation is meant to be visible (specifying `@OPERATION({State1,State2,...})`). If an operation has no states declared, then the operation is meant to be visible in all the states.

As a simple example, Fig. 7 shows the definition of an artifact type called `MyArtifact` (on the left), and an example of artifact use by an agent (on

[5] Annotations have been introduced along with the 5.0 version of Java.

```
\begin{verbatim}
@ARTIFACT_MANUAL(
  states = {"stateA","stateB"},
  start_state = "stateA",
  oi = @OPERATING_INSTRUCTIONS("..."),
  fd = @FUNCTION_DESCRIPTION("...")
) public class MyArtifact extends Artifact {

  private int count;
  private int max;

  public CounterArtifact(int max){
    this.max = max;
    count = 0;
  }

  @OPERATION({"stateA"}) void op1() {
    count++;
    genEvent("new_value",count);
    if (count >= max){
      setObservableState("stateB");
    }
  }

  @OPERATION({"stateA","stateB"}) void op2() {
    genEvent("value",count);
  }
}
```

```
...
ICartagoEnvironment env = Cartago.getInstance("...");
IAgentBody myBody = env.getAgentBody("...");

ArtifactId aid = myBody.getArtifactId("myArtifact");
SensorId sid = myBody.linkSensor(new DefaultSensor());

BasicFilter myFilter1 = new BasicFilter({"new_value"});
BasicFilter myFilter2 =
              new BasicFilter({"op_completed",
                               "state_changed"});

boolean state_changed = false;
while (!state_changed){
  try {
    myBody.execOp(aid,"op1",sid);

    // operation triggered:
    // sensing for one second for new_value events...
    Perception p = myBody.sense(sid,myFilter1,1000);
    log("current value: "+p.getContent());

    // observing next observable event,
    // which should be either
    // op_completed or state_changed
    Perception p = myBody.sense(sid,myFilter2,1000);

    String type = p.getType();
    if (type.equals("state_changed")){
      state_changed = true;
    }
  } catch (NoPerceptionException ex){
    // something wrong happened in the artifact
    // or simply artifact too slow in executing the op...
    break;
  } catch (OperationNotAvailableException ex){
    // inc was not part of artifact usage interface...
    break;
  }
}
...
```

Fig. 7. *(Left)* Complete definition of the `MyArtifact` type; *(Right)* A code fragment showing an example of use of a `MyArtifact` artifact

the right). As declared in the artifact manual, artifacts of sfMyArtifact kind have two possible observable states, labelled as `stateA` and sfstateB, with the former functioning as starting state. In the `stateA` state, the usage interface includes both the `op1` and `op2` operations, while in the `stateB` state the usage interface includes only `op2`. The execution of the `op1` operation causes the update of an internal counter of the artifact, whose new value is made observable by generating a `new_value` event. When the internal counter reaches a maximum value (provided with artifact initialisation), the artifact changes its observable state from `stateA` to `stateB`. The execution of the `op2` operation simply makes the current value of the internal counter observable, by generating an event of the kind `value`. As far as the artifact use is concerned, in the fragment—after creating an agent body inside the working environment where the artifact is located—`op1` operation is executed repeatedly, logging each time the value perceived by observing events generated as a consequence of the operation execution, until a change of artifact state is observed. In the example, two filters—instances of the class `BasicFilter`,

part of the utility class of CArtAgO—are used to select the perceptions. Using `BasicFilter`, a stimulus is selected if and only if its type description matches one of the descriptions provided as parameter of `BasicFilter` constructor (implemented as array of strings).

Run-time environment and related tools — This is the part actually responsible of the life-cycle management of working environments at run-time. Conceptually, it is the *virtual machine* where artifacts and agent bodies are instantiated and managed that is responsible of executing operations on artifacts and collecting and routing observable events generated by artifacts. Some tools are also made available in CArtAgO for on-line inspection of working environment state, in particular artifact state and behaviour, in terms operation executed and events generated.

4.2 A Complete Example: Hello Philosophers!

To illustrate a simple but complete example of MAS application exploiting artifact-based working environments, we consider the "Hello philosophers" example—listed among the basic examples in CArtAgO distribution—, which is used here analogously to the (in)famous "Hello world" example for traditional programming languages.

The example refers to the well-known problem introduced by Dijkstra in the context of concurrent programming to check the expressiveness of mechanisms and abstractions introduced to coordinate set of cooperating / competing computing agents. Briefly, the problem is about a set of N philosophers (typically 5) sharing N chopsticks for eating spaghetti, sitting at a round table (so each philosopher share her left and right chopsticks with a friend philosopher on the left and one on the right). The goal of each philosopher is to live a joyful life, interleaving thinking activity, for which they actually do not need any resources, to eating activity, for which they need to take and use both the chopsticks. The goal of the overall philosophers society is to share the chopsticks fruitfully, and coordinate the access to shared resources so as to avoid forms of deadlock or starvation of individual philosophers—e.g. when all philosophers have one chopstick each. The social constraint of the society is that a chopstick cannot be used simultaneously by more than one philosopher.

The problem can be solved indeed in many different ways. By adopting the A&A perspective, it is natural to model the philosophers as cooperative agents and the table—managing the set of chopsticks—as the coordination artifact that agents share and use to perform their (eating) activities. It is easy to encapsulate in the `table` artifact the enactment of the social policy that makes it possible to satisfy both mutual exclusion for the access on the individual chopsticks, and avoid deadlock situations.

Fig. 8 shows the complete application, with the `table` artifact implemented upon CArtAgO, the agent philosophers directly implemented as flat Java threads, without relying on a specific agent architecture.

The usage interface of the `table` artifact is composed by only two operations, `getChops` and `releaseChops`, which can be used respectively to get two chopsticks

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

public class Table extends Artifact {
  private boolean[] chops;
  private List<PendingReq> reqs;

  public Table(int nchops){
    chops = new boolean[nchops];
    reqs = new LinkedList<PendingReq>();
    for (int i = 0; i<chops.length; i++){
      chops[i]=true;
    }
  }

  @OPERATION void getChops(int c0, c1){
    if (chops[c0] && chops[c1]){
      chops[c0] = chops[c1] = false;
      genEvent("chops_acquired");
    } else {
      PendingReq req =
                  new PendingReq(c0, c1, getOpId());
      reqs.add(req);
    }
  }

  @OPERATION void releaseChops(int c0, int c1){
    chops[c0] = chops[c1] = true;
    Iterator<PendingReq> it = reqs.listIterator();
    while (it.hasNext()){
      PendingReq r = it.next();
      if (chops[r.c0] && chops[r.c1]){
        it.remove();
        chops[r.c0] = chops[r.c1] = false;
        try {
          genEvent(r.reqId,"chops_acquired");
        } catch (Exception ex){}
      }
    }
  }

  private static class PendingReq {
    public int c0,c1;
    public OpId reqId;
    public PendingReq(int c0, int c1, OpId id){
      this.c0 = c0; this.c1 = c1; reqId = id;
    }
  }
}
```

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

public class Philosopher extends Thread {
  private int lchop, rchop;
  private IAgentBody myBody;
  private String name;

  public Philosopher(String name, int c0, int c1,
                     String envName) throws Exception {
    this.name=name;
    lchop = c0;
    rchop = c1;
    ICartagoEnvironment env = Cartago.getInstance(envName);
    myBody = env.getAgentBody(name);
  }

  public void run() {
    try {
      ArtifactId tableId = myBody.getArtifactId("table");
      SensorId sid = myBody.linkSensor(new DefaultSensor());
      Op getOp = new Op("getChops",lchop, rchop);
      Op releaseOp = new Op("releaseChops",lchop, rchop);
      IPerceptionFilter myFilter =
                        new BasicFilter("chops_acquired");
      while (true){
        myBody.execOp(tableId,getOp,sid);
        try {
          myBody.sense(sid,myFilter,5000);
          eating();
        } catch(NoPerceptionException ex) {
          log("starved.");
          break;
        }
        myBody.execOp(tableId,releaseOp);
        thinking();
      }
    } catch (Exception ex){
    }
  }

  private void eating(){...}
  private void thinking() {...}
  private void log(String msg){...}
}
```

```
public class HelloPhilosophers {
    public static void main(String[] args) throws Exception {
        String envName = "restaurant";
        ICartagoEnvironment env = Cartago.getInstance(envName);
        env.createArtifact("table",Table.class,new ArtifactConfig(5));
        for (int i = 0; i<5; i++){
            new Philosopher("philo-"+i,i, (i+1)%nphilo,envName).start();
        }
    }
}
```

Fig. 8. Dining philosophers interacting within through a CArtAgO working environment called `restaurant`. Philosophers agents are simply implemented upon flat Java threads, while the table is implemented as a `table` artifact of class `Table`.

from the table and to give them back. The inner machinery of the `table` artifact ensures mutual exclusion on the access on chopsticks (an artifact executes one operation at a time, analogously to monitors) and deadlock avoidance (by releasing the chopsticks only if both are available, enqueueing the pending requests). It is worth noting the way in which observable events are generated: if the chopsticks are available when an instance of `getChops` operation is triggered, then the event `chops_acquired` is immediately generated, through the `genEvent` primitive; otherwise, the pending request is enqueued and the event is generated

as soon as the chopsticks become available with the execution of a `releaseChops` operation.

On the agent side, a philosopher gets an agent body during its initialisation, and then exploits it during its main activity—which is defined by the body of the method `run`. It is worth remarking that here we adopted such a simplistic implementation for agents just to make the description of CArtAgO usage and integration with agent platforms as simple and concise as possible. The same example using simpA agent framework can be found in simpA distribution. Agent main activity accounts for repeatedly alternate eating and thinking sub-activities, using the table artifact to get (and release) the chopsticks. In particular, to get the chopsticks the agent triggers the execution of the `getChops` operation on the table artifact, specifying a sensor (previously linked to its body) to collect stimuli related to this action. Then, it pro-actively observes the sensor for 5 seconds using the `sense` primitive, filtering stimuli that concern `chops_acquired` events. If no perception is sensed within 5 seconds, the philosopher starves and terminates. Otherwise, it performs its eating activity and then, after eating, it releases the chopsticks by executing a `releaseChops` operation on the `table` artifact, without specifying any sensor (since, in this simple implementation, it is not interested to observe the effects of such an action).

5 Related Works

The approach based on artifacts shares the same engineering aims introduced by Weyns and colleagues in [11], 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 authors 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 colleagues [12], where a general model for environments providing functionalities for *over-hearing* and *over-sensing* is presented. Our notion of artifact could 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 [9], 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* and *sense* actions 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 and link to their body 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). Similar to perceptual laws discussed above, sensor activity can be constrained according to laws enforced by the organisational and physical context where the agent is situated: this aspect will be explored in future work, with the introduction of an explicit support for organisation modelling on top of workspaces (see Sect. 6). Pattern-driven sensing described in Sect. 3 could 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 framework draw on the research work on *tuple centres* as programmable tuple-based coordination media and TuCSoN coordination model [13]. 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* [8], as artifacts designed to encapsulate programmable coordination services.

6 Concluding Remarks

In this paper we described CArtAgO, as a framework supporting the engineering of artifact-based working environment in MAS. First we described the abstract model and architecture of the framework, and then a first basic prototype technology implementing most of the 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 issues not currently faced in CArtAgO, and that will be part of our future work, we mention here *distribution*, *security*, and *organisation*.

As far as distribution is concerned, currently in CArtAgO there is no explicit account for the way in which workspaces—and possibly also artifacts—can be

distributed across multiple nodes of a networks. In current version, working environments are confined to a single CArtAgO virtual machine (node) and then agents can create and use artifacts that live in their local environment. Distribution is achieved by ad-hoc linking of artifacts, that is, by exploiting network connections that makes it possible the low-level communication among artifacts belonging to different workspaces and working environments. In future work we will focus on extending the framework towards a full-fledged *infrastructure*, providing a first-class support for such an aspect.

Security and organisation are related issues, and call for the introduction of an explicit support for organisation built on top of the basic CArtAgO abstractions, which would be effective also to model security aspects such as access control. By drawing on our previous research work about such aspects on TuCSoN infrastructure, in CArtAgO we plan to introduce a *role*-based model, inspired to RBAC (Role-Based Access Control) architectures [14], such as RBAC-MAS [15]. Such a model will be based on the notion of *workplace*. 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 may have or not the permission to use some artifacts or to execute some specific operations on some specific artifacts. So, workplaces would define an organisational layer—and, consequently, a security layer—on top of workspaces.

Besides the above three main points, future work will be devoted also: *(i)* to improve the development of the prototype, including all the missing features that are currently part of the abstract model—such as the support for workspaces—and future extensions—such as workplaces; *(ii)* to integrate existing services as kinds of artifact, in order to be easily re-used when engineering applications on top of CArtAgO: an example is given by artifacts wrapping TuCSoN tuple centres, providing agent coordination facilities; *(iii)* to define suitable formal models and ontology for describing 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. Weyns, D., Omicini, A., Odell, J.: Environment as a first-class abstraction in multi-agent systems. Autonomous Agents and Multi-Agent Systems **14** (2007) 49–60 Special Issue on Environments for Multi-agent Systems.

3. Viroli, M., Ricci, A., Holvoet, T., Shelfthout, K., Zambonelli, F.: Infrastructures for the environment of multiagent systems. Autonomous Agents and Multi-Agent Systems **14** (2007) 5–30 Special Issue on Environments for Multi-agent Systems.
4. Nardi, B.A.: Context and Consciousness: Activity Theory and Human-Computer Interaction. MIT Press (1996)
5. Kirsh, D.: Distributed cognition, coordination and environment design. In: European conference on Cognitive Science. (1999) 1–11
6. Agre, P.: Computational research on interaction and agency. Artificial Intelligence **72** (1995) 1–52
7. 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
8. 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
9. Weyns, D., Steegmans, E., Holvoet, T.: Towards active perception in situated multiagent systems. Applied Artificial Intelligence **18** (2004) 867–883
10. Viroli, M., Ricci, A.: Instructions-based semantics of agent mediated interaction. In: AAMAS'04. Volume 1., New York, USA, ACM (2004) 286–293
11. Weyns, D., Holvoet, T.: Formal model for situated multiagent systems. Fundamenta Informaticae **63** (2004) 125–158
12. Platon, E., Honiden, S., Sabouret, N.: Oversensing with a softbody in the environment: Another dimension of observation. In Kaminka, G.A., Pynadath, D.V., Geib, C.W., eds.: Workshop "Modeling Others from Observation" (MOO 2005), IJCAI-05, Edinburgh, Scotland (2005)
13. Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems **2** (1999) 251–269
14. Sandhu, R., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based control models. IEEE Computer **29** (1996) 38–47
15. Omicini, A., Ricci, A., Viroli, M.: An algebraic approach for modelling organisation, roles and contexts in MAS. Applicable Algebra in Engineering, Communication and Computing **16** (2005) 151–178 Special Issue: Process Algebras and Multi-Agent Systems.