The A&A Programming Model and Technology for Developing Agent Environments in MAS

Alessandro Ricci, Mirko Viroli, and Andrea Omicini

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

Abstract. In human society, almost any cooperative working context accounts for different kinds of object, tool, artifact in general, that humans adopt, share and intelligently exploit so as to support their working activities, in particular social ones. According to theories in human sciences, such entities have a key role in determining the success or failure of the activities, playing an essential function in simplifying complex tasks and—more generally—in designing solutions that scale with activity complexity. Analogously to the human case, we claim that also (cognitive) multi-agent systems (MAS) could greatly benefit from the definition and systematic exploitation of a suitable notion of *working environment*, composed by different sorts of artifacts, dynamically constructed, shared and used by agents to support their working activities. Along this line, in this paper we introduce and discuss a programming model called A&A (Agents and Artifacts), which aims at directly modelling and engineering such aspects in the context of cognitive MAS. Besides the conceptual framework, we present the current state of prototyping technologies implementing A&A principles—CARTAGO platform in particular—, and show how they can be integrated with existing cognitive MAS programming frameworks, adopting the *Jason* programming platform as the reference case.

1 Introduction

"Artifacts play a critical role in almost all human activity [...]. Indeed [...] the development of artifacts, their use, and then the propagation of knowledge and skills of the artifacts to subsequent generations of humans are among the distinctive characteristics of human beings as a specie", Donald Norman, [8]

"The use of tools is a hallmark of intelligent behaviour. It would be hard to describe modern human life without mentioning tools of one sort or another", Robert Amant, [2]

In their articles, Norman [8] and Amant [2] remark—in different contexts—the fundamental role that *tools* and, more generally, *artifacts* play in human society. Artifacts and tools here could be understood as whatever kinds of device explicitly designed and used by humans so as to mediate and support their activities, especially social. Analogous observations are found in the work of Agre and Horswill in their *Lifeworld analysis* [1], as well as in the work of Kirsch [5,6]. Actually, such a perspective is central in theories developed in the context of human sciences, such as Activity Theory and Distributed Cognition, and currently taken as a reference by computer science related disciplines such as CSCW (Computer Supported Cooperative Work) and HCI (Human-Computer Interaction) [7]. There, a fundamental point is devising out the best kind of artifacts to populate humans' *fields of works*, and to organise them so as to improve as much as possible the performance of their activities, in particular coordinative ones [13,6].

Analogously to human society, we think that such a perspective is and will be fundamental also in the context of agent societies, and in particular for designing and programming complex software systems based on cognitive MAS. Quite provocatively, analogously to the human case, we think that the next evolution step in the development of cognitive MAS will *mandatorily* require the definition of MAS models and architectures with agents situated in suitable *working environments*. There, agents autonomously—besides speaking to each others—construct, share, and co-operatively use different kinds of artifact, designed either by MAS designers or by the agent themselves, to perform MAS activities. Indeed, this notion of environment is quite different with respect to the one traditionally adopted in mainstream cognitive agent theory: there, the environment is typically conceived as something "out of the MAS", then not a subject of design. On the contrary, the notion of "working environment" promotes MAS environment as an essential part of the MAS to be explicitly designed and fruitfully exploited by agents in their working activities.

Along this line, in this paper we introduce and discuss a first programming model called A&A (Agents and Artifacts) which aims at directly modelling and engineering working environments in the context of cognitive multi-agent systems. Such a perspective is strenghtened by recent efforts in AOSE (Agent-Oriented Software Engineering) that remark the fundamental role of the environment for the engineering of MAS [14]. The A&A approach can be considered an instance of such approaches, with some specific peculiarity: *(i) abstractions and generality*—the aim is to find a basic set of conceptual abstractions and related theory which, analogously to the agent abstraction, could be general enough to be the basis to define concrete architectures and programming environments, but specific enough to capture the essential properties of systems; *(ii) cognitive*—analogous to designed environment in human society, the properties of such environment abstractions should be conceived to be suitably and effectively exploited by cognitive agents, as intelligent constructors / users / manipulators of the environment.

Besides the abstract programming model, in this paper we describe also the concrete technologies developed to experiment the model: in particular we discuss CARTAGO technology, a platform for programming and supporting the execution of artifact-based working environments, developed on top of the Java platform, and its integration with the Jason agent programming environment.

The basic notion of artifact has been already introduced and published elsewhere [11,9,10], and the same applies for the first version of CARTAGO technology [12]. On the one side, besided purely conceptual papers such as [9], papers such as [11,10] can be considered first steps introducing the concept of artifact for programming MAS, without having a reference programming model defined here—called A&A—and related functioning technologies, i.e. CARTAGO, that can be integrated to existing platform. On the other side, the artifact programming model and its implementation in CARTAGO technologies has been substantially evolved with respect to the most recent one, described in [12]. In particular, the basic model of usage-interface and operation presented in [12] is quite simple and is not able to properly take into account the possibility to have the concurrent execution of operations on an artifact (the interested reader is forwarded to the paper for the details). Such a model has been completely revised, and this new version—which substantially change the way a MAS programmers can adopt to program artifact operations and behaviour—is described in detail in this work. Besides this, the work published in [12] is more oriented on the environment / infrastructure level: in this paper instead, besides describing in detail CARTAGO, we focus more on the A&A programming model and CARTAGO is described as an existing functioning technology supporting such a model.

The remainder of the paper is organised as follows. First, we provide a description of the basic concepts and principles of A&A (Section 2), by introducing an abstract model embedding such principles (Section 2.3). Then, we briefly present the current models and technologies that have been developed for concretely prototyping MAS applications in the A&A perspective (Section 3)—among which the one called CARTAGO—and discuss the issue of integration of such technologies with existing cognitive MAS programming platform, adopting *Jason* as our reference case study. Finally, we conclude the paper with some final remarks, and sketch some future line of work (Section 4).

2 Programming Model Building Blocks

2.1 Artifacts and Workspaces

A *working environment* in A&A is defined as the part of the MAS that is designed and dynamically constructed and used by agents to support their working activities. MAS programmers design and define the types of artifacts that agents will dynamically instantiate and cooperatively use.

A working environment is conceived as a dynamic set of *artifacts*, organised in *workspaces*. Workspaces are the logical containers of artifacts, useful to define the topology of the working environment. A workspace provides a notion of locality for agents: an agent can work only with artifacts belonging to the workspace where it is playing, but can be conceptually situated in multiple workspaces at the same time, possibly distributed on different Internet nodes. This concept can be used to define the distribution model of an application at an abstract level: a working environment—which corresponds to a possibly distributed application

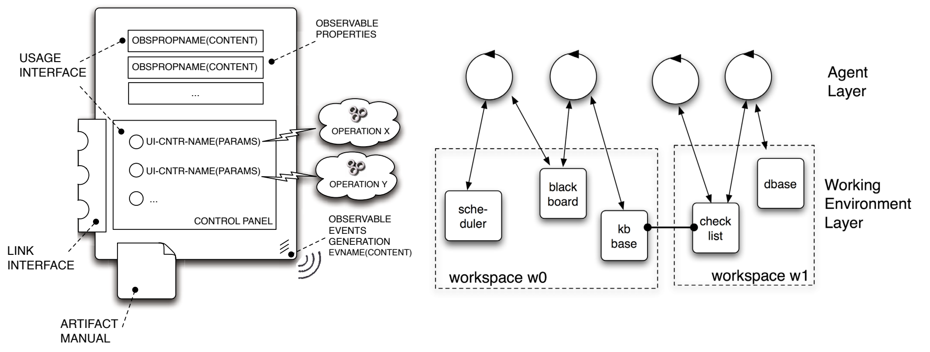

Fig. 1. *(Left)* Abstract representation of an artifact. *(Right)* Abstract representation of a working environment with two workspaces, with some artifacts of different kinds inside.

or MAS—can account for one or multiple workspaces, possibly spread among multiple network (Internet) nodes.

Current model does not explicitly take into the account security and organisation issues: for examples roles that can be defined in a workspace and the possible permits related to roles. These aspects are part of future work: we plan to adopt RBAC-like approach as a basic organisation and security model, as we did for the TuCSoN coordination model [15].

The notion of artifact is the core abstraction of the programming model: it is meant to represent any entity belonging to the working environment—hence existing outside the agent mind—that is created, shared & used (and eventually disposed) by agents to carry on their activities, in particular social ones. So, an artifact (type) is typically meant to be explicitly designed by MAS engineers so as to encapsulate some kind of *function*, here synonym of "intended purpose". An abstract representation of an artifact is shown in Fig. 1 and it is very similar to artifacts as found in human society. The functionality of an artifact is structured in terms of *operations*, whose execution can be triggered by agents through artifact *usage interface*. Analogously to usage interface of artifacts in our world (think, for example, of a coffee machine), an artifact usage interface in A&A is composed of a set of commands or *controls* that agents can use to trigger and control operation execution (such as the control panel of a coffee machine), each one identified by a label (typically equals to the operation name to be triggered) and a list of input parameters. The usage interface can change dynamically, according to state of the artifact; in other words, it is possible to design artifacts that expose a different usage interface according to their functioning stage. Besides the control to act, the usage interface might contain also a set of *observable properties* (think of the coffee machine display); that is, properties whose dynamic values can be observed by agents without necessarily interacting with (or operating upon) the artifact.

The execution of an operation upon an artifact can result both in changing the artifact's inner (i.e., non-observable) state, and in the generation of a stream of *observable events* that can be perceived by agents that are using or simply observing the artifact. Such a model strictly mimics the way in which humans use their artifacts: a simple example is the coffee machine, whose usage interface includes suitable controls—such as the buttons—and means to make (part of) the machine behaviour observable—such as displays—and to collect the results produced by the machine—such as the coffee can. It's worth remarking here the differences between observable properties and observable events. The former are dynamic and persistent attributes that belong to an artifact and that can be observed by agents without interacting with it (i.e. without using the ui-controls). Like the display of a coffee machine. The latter are non-persistent information, as signals carrying also an information content. Like the sound emitted by a coffee machine when the coffee is ready.

Artifacts can embed complex functionalities: accordingly, operations executed can be complex and an articulated operation model is provided for this purpose. Generally speaking, operation execution can be conceived as a process (from a conceptual point of view) combining the execution of possibly multiple *atomic guarded* operation *steps*. Multiple operations can be in execution upon an artifact by interleaving the execution of the operation steps. In order to avoid interferences, during the execution of a single operation step, the usage interface is disabled. This approach, in the overall, makes it possible to support the execution of multiple operations concurrently on the artifact, keeping mutual exclusion in artifact state access.

Analogously to artifacts in the human case, in A&A each artifact is meant to be equipped with a *manual* describing the artifact's function (i.e., its intended purpose), the artifact's usage interface (i.e., the observable "shape" of the artifact), and artifact's *operating instructions* (i.e., usage protocols or simply how to correctly use the artifact so as to take advantage of all its functionalities). A manual is meant to be inspected and used at runtime by agents, in particular intelligent agents, for reasoning about how to select and use artifacts so as to best achieve their goals. Accordingly, suitable formal languages and ontologies could be defined for manual description. Currently, no specific commitments towards specific technologies have as yet been made, as this is part of ongoing work.

Finally, as a principle of composition, artifacts can be linked together, in order to enable artifact–artifact interaction. This is realised through *link interfaces*, which are analogous to interfaces of artifacts in the human case (e.g. linking/connecting/plugging the earphones into an MP3 player, or using the remote control with a TV). In the overall link interfaces serve two purposes: on the one side, to explicitly define a principle of composability for artifacts, enabling the ruled construction of articulated and complex artifacts by means of simpler ones; on the other side, to realise distributed (composed) artifacts, by linking artifacts belonging to different workspaces.

2.2 Agent Bodies

In this overall picture, nothing is said about the specific cognitive model of the agent: actually A&A is meant to be orthogonal to this aspect: agents are simply conceived as autonomous entities executing some kind of working activities, either individually or collectively—typically in order to achieve some individual or social goal, or to fulfill some individual or social task. Such activities—from an abstract point of view—are seen as the execution of sequences of actions, which according to the A&A model can be roughly classified as: *(i)* internal actions, *(ii)* communicative actions, involving direct communications with one or more agents through some kind of ACL, and *(iii)* pragmatical actions, as interactions within working environments that concern construction, sharing, and use of artifacts.

Despite of their specific cognitive model / architecture, in order to execute actions over the artifact and perceive observable events, agents must be *situated* in a working environment: for this purpose, the notion of *agent body* is introduced. The agent body functions as the medium through which the agent *mind*—i.e. those parts that are designed and programmed according to a certain kind of cognitive model / architecture—can sense and affect a working environment. Such a notion is essential to decouple—for engineering purposes—the agent mind from the working environment in which the agent is situated, so as to be able to use A&A with different kinds of programming models for agent mind.

Agent bodies contain *effectors* to perform actions upon the working environment, and a dynamic set of *sensors* to collect stimuli from the working environment. Sensors in particular play here a fundamental role, that of *perceptual memory*, whose functionality accounts for keeping track of stimuli arrived from the environment, possibly applying filters and specific kinds of "buffering" policy. According to the specific interaction modality adopted for using and observing artifacts, as described later in this section, it might be useful to provide agents with basic internal actions for managing and inspecting sensors, as a kind of private memory. In particular, it could be useful for an agent to organise in a flexible way the perception of observable events, possibly generated by multiple different artifacts that an agent can be using for different, even concurrent, activities.

2.3 The Agent Programming Interface

In this subsection we provide an abstract description of the basic interface available to agent minds to play inside a working environment. Such an interface accounts for three basic groups of actions: (*i*) join and leave workspaces; (*ii*) use an artifact by acting on its usage interface and perceive observable events generated by artifacts; (*iii*) observe an artifact. Table 1 provides a synthetic view of the set of actions, grouped into the three main groups; as for the syntax, a pseudo-code first-order logic-like syntax is adopted, while semantics is described informally. Atoms `AName`, `WName` and `SName` are used to represent a unique name (identifier) for respectively artifact instances, workspaces and sensors. Following the semantics adopted in the cognitive agent-oriented programming approaches considered here, an action consists in the atomic execution of a statement which can result in changing the agent's state and/or interacting with the agent's environment, and can succeed or fail.

The first group of actions (labelled 1–2) are useful for managing a working session inside a workspace. Intuitively, `join` makes it possible to "enter" a

Table 1. Basic set of actions to interact with A&A work environments. + is used for optional parameters, ? for input parameters.

(1)	`joinWorkspace(WName,+Node)`
(2)	`quitWorkspace`
(3)	`use(AName,UIControlName(Params),+SName,+Timeout,+Filter)`
(4)	`sense(SName,?Perception,+Filter,+Timeout)`
(5)	`focus(AName,SName,+Filter)`
(6)	`stopFocussing(AName)`
(7)	`observeProperty(AName,PropertyName,?Property)`

workspace, whose name is specified as a parameter and `quit` to leave the current workspace. Since workspaces are meant to be distributed over a network, optionally the node where the workspace resides can be specified.

The second group of actions (labelled 3–4) concerns the *use* of artifacts. In particular `use` action accounts for using the artifact identified by `AName`, by acting on the `UIControl` usage-interface control, specifying some `Params` parameters, and optionally specifying a sensor `SName` to be used to collect the events generated by the artifact, a filter `Filter` and a timeout `Timeout`. The action succeeds if the specified artifact exists and its usage interface actually has the specified control, and as a result the related operation is triggered for execution. Then, if a sensor has been specified, every observable event subsequently generated by the artifact, as effect of the operation execution, is made observable to the agent as a stimulus `artifact_event(AName,Event)` collected in the sensor. The filter can be used to specify which kinds of events the agent is interested in perceiving. If the usage interface of the artifact is disabled when executing the action, for instance because the artifact is executing an operation (step), then the agent action is suspended until the usage interface is enabled again; the timeout specifies how long the agent can wait before considering the action as failed. Then, a `sense` action is provided to inspect the content of a sensor (i.e. the perceptual memory), so that the agent can become aware of any new percepts. In particular, the action succeeds if within `Timeout` time an event (stimulous) matching the specified filter `Filter` is found in the specified sensor `SName`. In that case, `Perception` is bound with such event. Both the timeout and the filter can be omitted. The same sensor can be used for collecting events of different usage interactions, possibly with different artifacts. It's worth remarking that the execution (and completion) of the `use` action is *completely asynchronous* with respect to the execution of the operation by the artifact and to the possible consequent generation of events. It is synchronous however with respect to the presence of the specified ui-control in the usage interface: if the action succeeds, then it means that such ui-control was part of the usage interface, that it has been "pressed" and that the related operation has been triggered for being executed (as soon as its guard is satisfied).

The third group of actions (labelled 5–7) concerns artifact observation, i.e. the capability of perceiving artifacts observable events and properties without directly interacting with them. The `focus` action can be used to start a continuous observation of an artifact (intuitively, to focus one's attention on that artifact so as to observe any changes that occur in it over time). The action succeeds if the `AName` artifact exists, and as an effect every observable event generated by the artifact (despite the specific operation that caused it, possibly executed by any other agent) is made observable to the agent as a stimulus `artifact_event(AName,Event)` collected in the specified sensor. Also for `focus`, a filter can be specified in order to select which kinds of event to actually observe. `stopFocussing` is used to stop observing the artifact. It's worth remarking here the differences between `focus` and `sense` actions: `sense` is an internal action, since it inspects a sensor (which is considered part of the agent); `focus`, instead, is external, enabling continuos observation of events that directly cause belief base update in the first modality, and sensor content update in the second modality. Finally `observeProperty` can be used to inspect the observable properties of the specified `AName` artifact, specifying the name `PropertyName` of the property to be observed. The action succeeds if the `AName` artifact exists and has a property with the specified name, and as result the current value of the property is bound to `Property`.

This abstract model is meant to be as much as possible orthogonal to the model(s) adopted for defining agent mind at the agent level. This should make it easier to integrate A&A concrete models and technologies with existing agent technologies—as described in Section 3.2.

2.4 The Artifact Programming Interface

Besides the API to use artifacts, a programming model for defining artifact types is given: Table 2 shows an abstract description of the main primitives used to define artifacts behaviour.

An artifact type or template defines the structure and behaviour of artifacts instances of such a type. For some extent, an artifact type is quite similar to the notion of class in Object-Oriented Programming, with artifacts analogous to objects, and to the notion of monitor, as defined in concurrent programming. As in the case of objects, the structure defines the artifact inner state and working machinery hidden to agent users. The behaviour is structured in a set of *operations*, which define in the overall artifact function. An operation encapsulates the computational and interaction behaviour—such as the update of the internal state and the generation of observable events—so as to provide some kinds of functionality. Operations execution is triggered and controlled by acting on the controls which are part of the artifact usage interface. During the execution of an operation, observable events (signals) can be generated by using a specific primitive, `signal` (label 1 in Table 2), specifying the event content as a labelled tuple.

Table 2. Basic API available for programming an artifact `+` is used for optional parameters, `?` for input / output parameters.

(1) `signal(Event)`
(2) `nextStep(OpStep(Params),+Guard(Params))`
(3) `switchToObsState(StateName)`
(4) `updateObsProperty(Property)`
(5) `readObsProperty(?Property)`

In order to enforce mutual exclusion in updating artifact state on the one side and allow for concurrent operations execution on the other side, artifact operations can be composed by one or multiple *operation steps*, which are meant to be executed atomically. At a given time only one operation (step) can be in execution: multiple operations can be executed concurrently by interleaving their steps. For each step *a guard* can be defined, which specifies when the step—once it has been striggered in the context of an operation—can actually be executed. Guards represent the condition that must be satisfied, as a predicate over the artifact state, to execute a step. So, an operation (step) is first triggered by agent user, the executed when the guard is satisfied. A step can trigger other steps, by means of the `nextStep` primitive (label 2 in Table 2), specifying the operation step to be executed and possibly the guard. An operation is considered completed when no more steps have been triggered. In the overall, this model makes it possible to design a complex articulated operation, that—for instance—can be controlled by using different controls in different times in the user interface before being completed, or—as another example—would need the execution of other operations to complete.

Besides the analogies with classes and monitors in object-oriented and concurrent programming, it's worth remarking here the deep difference with respect to those concepts, in particular for what concerns the interaction model: by virtue of agent autonomy, artifact operation (step) execution does not involve any control flow from the invoker agent to the invoked artifact, i.e. is not a method or function call.

The usage interface of an artifact can change according to artifact *observable state*, exposing different sets of operations and *observable properties* according to the specific functioning state of the artifact. The notion of observable state is adopted to structure the behaviour of an artifact in a set of labelled states, that can be recognised (observed) by the artifact users. For each artifact type a finite set of labelled observable states can be defined. Each artifact instance has a *current observable state*, that can be changed dynamically during artifact functioning by means of the `switchToObsState` primitive (label 3 in Table 2, specifying the label of the target observable state. For each observable state a different usage interface can be defined: this feature makes it possible to set the appropriate usage interface according to the functioning stage of the artifact. Dynamically, an agent can trigger the execution of an operation on an artifact if and only the operation is (in that moment) part of the usage interface; if the operation is not part the usage interface, the agent action fails. Observable properties can be defined as labelled tuple of information that can be observed by agents without directly using the artifacts. Basic primitives are available as part of artifact API for updating (`updateObsProperty`, label 4) and reading (`readObsProperty`, label 5) the value of an observable property: also properties are represented by labelled tuples and their access is meant to be associative.

3 Prototyping Technologies

Starting from the A&A abstract model, we developed some first concrete technologies, with the objective to have concrete frameworks for prototyping MAS-based applications engineered upon A&A basic abstractions, and for being integrated with existing agent technologies extended with the A&A support.

A primary technology is called CARTAGO (Common ARtifact Infrastructure for AGent Open environment), which is a framework providing essentially the capability to define new artifacts type, suitable API for agents to work with artifacts and workspaces, and a runtime supporting the existence and dynamic management of working environments. Another technology is called simpA (simple A&A programming environment), a framework extending CARTAGO with a support for defining and running agents (MAS) besides the working environments. While CARTAGO is meant to be integrated with existing (cognitive) agent models and technologies as a seamless support to define and create artifact-based working environments, simpA can be exploited alone to develop full-fledged applications engineered in terms of agents, artifacts and workspace. For lack of space, in this paper simpA is not described: the interested reader can refer to simpA web site[1]. Both technologies are based on Java and are available as open-source projects freely downloadable from the project web sites[2].

3.1 CARTAGO Overview

The CARTAGO architecture implements quite faithfully the abstract model described in Section 2.3. Pragmatically, we chose Java as the programming language to implement and map the programming model elements, adopting choices that would favour rapid prototyping, reusing as much as possible the support given by the Java Object-Oriented framework. In the following we briefly describe the three main parts of CARTAGO:

– *API for creating and interacting with artifact-based working environments* — These API are meant to extend the existing basic set of agent actions with

[1] `http://www.alice.unibo.it/simpa`
[2] `http://www.alice.unibo.it/cartago`

new ones, abstractly described in Section 2.3, essentially for creating and disposing artifacts, interacting with them through their usage interface—by executing operation and perceiving artifact observable states and events—reading artifact function description and operating instructions, managing sensors, and so on.

– *API for defining artifact types* — These API allow MAS programmers to develop new types of artifacts. An artifact type can be defined by extending the basic `Artifact` class provided in the API: at runtime, artifacts instances are instances of this class. Artifact structure (internal state) is defined in terms of instance fields of the class. Operations and operations steps body is defined by methods tagged by `@OPERATION` and `@OPSTEP`, where the operation (step) name and parameters are mapped onto the name and the parameter of the methods[3]. Guards are represented by boolean method annotated with the `@GUARD` annotation. Methods representing operations / operation steps have no return argument—a return argument would be meaningless in the A&A model. Observable events—which are the means to make agents perceive operation results—can be generated in the body of an operation by primitives of the kind `signal`, available as protected methods of the artifact. Events are collected by agent body sensors as stimuli, and then perceived by agents through `sense` action. Artifact function description and 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.

  A simple example of artifact definition is shown in Fig. 2: a simple type `MyArtifact` is defined, with an internal variable `m` and two operations, `op1` and `op2`, the former composed by two steps, the first one coinciding the first of the operation and the second one, `opStepA` with guard `canExecA` triggered by the first step by means of the `nextStep` CARTAGO primitive. The operation `op1` initialized the variable to 1 and then completes only when the variable value reaches the value 3, a condition that triggers the execution of the second step which generates the observable event `maxReached`. Each time the operation `op2` is executed, the variable is incremented and an observable event `newValue` generated. More complex and useful examples can be found in CARTAGO distribution.

– *Runtime environment and related tools* — This is the part actually responsible of the life-cycle management of working environments at runtime. 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 (see Fig. 3 for an abstract representation of a MAS application running on top of CARTAGO). Some tools are also made available in CARTAGO for online inspection of working environment state, in particular artifact state, and above the observation of artifact behaviour, in terms operation executed and events generated.

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

```
import alice.cartago.*;

class MyArtifact extends Artifact {
  private int m;

  @OPERATION void op1(){
    m=1;
    nextStep("opStepA", "canExecA");
  }
  @OPERATION void op2(){
    m++;
    signal("newValue",m);
  }
  @GUARD boolean canExecA(){
    return m == 3;
  }
  @OPSTEP void opStepA(){
    signal("maxReached");
    m = 1;
  }
}
```

```
...
ArtifactId id =
  createArtifact("myArtifact", MyArtifact.class);
SensorId sid =
  linkSensor(new DefaultSensor());

use(id, new Op("op1"));

while (true){
  use(id, new Op("op2"),sid);
  Perception p = sense(sid);
  if (p.getLabel().equals("maxReached")){
    break;
  }
}
...
```

Fig. 2. *(Left)* A simple type of artifact, with two operations, `op1` and `op2`, the former composed by two steps, the first one coinciding the beginning of the operation and the second one, `opStepA` with guard `canExecA` triggered by the first step. *(Right)* A Java fragment using CARTAGO API to create an artifact, link a sensor, execute the `op1` operation and then repeatedly execute the `op2` operation until a `maxReached` event is observed. `createArtifact` and `linkSensor` are auxiliary actions part of the Java CARTAGO implementation.

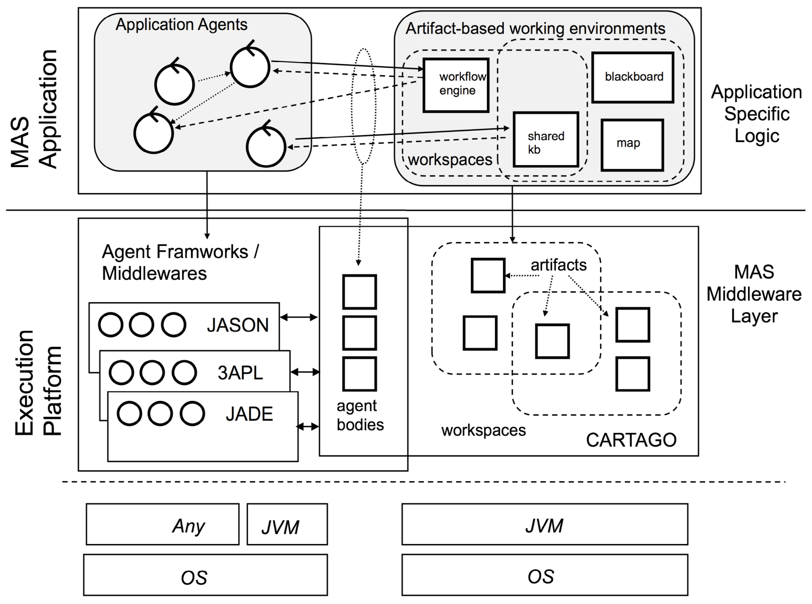

Fig. 3. Abstract representation of a MAS application exploiting CARTAGO

Further details about CARTAGO API and architecture, along with complete examples, can be found on the web site.

3.2 Integration with Existing MAS Programming Environments

As mentioned previously, an important aspect of A&A and of technologies such as CARTAGO is the possibility of integration with existing cognitive MAS architectures and models / languages / platforms, so as to extend them to create and work with artifact-based environments.

Actually, most available agent programming models and platforms for developing general-purpose applications—such as *Jason*, 3APL, Jadex, JACK, and others surveied in [3]—lack a true notion of environment, and when such a notion is accounted for, it is typically modelled and implemented in terms of low-level interfaces to the hosting VM or OS environment, or by considering a general monolithic abstraction of "Environment" and of "Event". This is perfectly reasonable according to the notion of environment as traditionally dealt with in agent theories. By integrating these platforms with A&A, the environment notion is seamlessly extended with the capability for cognitive agents written in existing programming environments to create, share and use artifacts according to the specific needs, with MAS designers directly programming artifacts so as to create the best working environments for supporting agent activities. Also, existing types of artifact can be reused, especially those providing general purpose functionalities such as the coordination artifacts. Furthermore, from a conceptual point of view it would be possible and interesting to build MAS applications composed by heterogeneous agent societies, made of cognitive agents programmed with different agent languages or architectures, working together in the same working environments, and interacting through the same mediating artifacts—besides communicating by means of the same ACL as usual.

In the following subsection we sketch the first results obtained with a concrete case, concerning the integration of CARTAGO with the *Jason* agent-oriented programming platform.

3.3 A Case Study: *Jason* Using CARTAGO

Jason is an interpreter written in Java for an extended version of AgentSpeak [4], a logic-based agent-oriented programming language that is suitable for the implementation of reactive planning systems according to the BDI architecture. *Jason* is taken here is as reference case: analogous considerations can be done using other platforms such as 3APL or Jadex.

By the integration then, it is possible to create a MAS application composed by a set of *Jason* agents working inside the same CARTAGO environment. By default, each *Jason* agent has an agent body inside the CARTAGO environment, and his basic set of external actions is extended to include the basic ones abstractly described in Section 2.3. In particular, a *Jason* agent can use an artifact by means of `use` and perceive artifact events collected by its sensors through `sense` action, and so on. In current simple integration model, percepts that are fetched by `sense` action are mapped to beliefs

```
MAS cartagoTest {

  infrastructure: Centralised

  environment: CartagoEnvironment

  agents:
    rosa philo.asl;
    beppo philo.asl;
    pippo philo.asl;
    maria philo.asl;
    giulia philo.asl;
    alfredo waiter.asl;
}
```

```
/* TABLE ARTIFACT */

import alice.cartago.*;

public class Table extends Artifact {
 private boolean[] chops;

  public Table(int nchops){
    chops = new boolean[nchops];
    for (int i = 0; i<chops.length; i++){
      chops[i]=true;
    }
  }
  @OPERATION(
    guard = "chopsAvailable"
  ) void getChops(int lc, int rc){
    chops[lc] = chops[rc] = false;
    signal("chops_acquired");
  }
  @GUARD boolean chopsAvailable(int lc,int rc){
      return chops[lc] && chops[rc];
  }
  @OPERATION void releaseChops(int lc, int rc){
      chops[lc] = chops[rc] = true;
      signal("chops_released");
  }
}
```

Fig. 4. *(Left)* Definition of a Jason MAS called `cartagoTest`, composed by five philosopher agents (`rosa`, `beppo`, `pippo`, `maria`, `giulia`) and a waiter agent (`alfredo`), sharing a CARTAGO environment. *(Right)* Definition of the `Table` artifact type

of the type `artifact_event(Type,SensorId,ArtifactId,EventTime)`, while exceptions regarding timeouts elapsed during sense actions to beliefs of the kind `sensing_timeout(SensorID)`.

Hello Philosophers! As a simple integration example, we consider the case "Hello philosophers" used here with analogous function of the "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

```
/* WAITER AGENT */

!live.
+!live : true
<- .print("Hello world!") ;
   .print("Preparing the table...") ;
   createArtifact(myTable,"Table",[5]);
   .print("The table is ready.") ;
   .print("Assigning the chopsticks");
   .send("rosa",tell,
       chops_assigned(myTable,0,1));
   .send("beppo",tell,
       chops_assigned(myTable,1,2));
   .send("pippo",tell,
       chops_assigned(myTable,2,3));
   .send("maria",tell,
       chops_assigned(myTable,3,4));
   .send("giulia",tell,
       chops_assigned(myTable,4,0));
   .print("Good luck.").
```

```
/* PHILOSOPHER AGENT */

!live.
+!live : true
  <-  .print("Hello world! Waiting to know my chopsticks...").

+chops_assigned(Table,C0,C1) : true
  <-  .print("I know my chopsticks, I can start my activity.");
      +my_chopsticks(Table,C0,C1) ;
      +wants_to_live_for_another_round.

+wants_to_live_for_another_round : true  <- !think.

+!think : not(hungry)
  <-  .print("Thinking.");
      -wants_to_live_for_another_round; +hungry.

+hungry :  my_chopsticks(Table,C1,C2) &
        not(got_chopsticks(C1,C2)) &
        not(chopsticks_requested(C1,C2))
  <-  .print("Got hungry, try to eat") ;
      use(Table,getChops(C1,C2),mySensor);
      +chopsticks_requested(C1,C2);
      sense(mySensor,8000).

+artifact_event(chops_acquired,mySensor,Table,EventTime) :
        chopsticks_requested(C1,C2)
  <-  .print("Got chopsticks, can eat.");
      -chopsticks_requested(C1,C2);
      +got_chopsticks(C1,C2); -hungry;
      use(Table,releaseChops(C1,C2),mySensor);
      sense(mySensor).
+sensing_timeout(mySensor) : chopsticks_requested(C1,C2)
  <-  .print("Starved, good bye world.");
      .myName(Me); .killAgent(Me).

+artifact_event(chops_released,mySensor,Table,_) :
        got_chopsticks(C1,C2)
  <-  .print("Chopsticks released.");
      -got_chopsticks(C1,C2);
      +wants_to_live_for_another_round.
```

Fig. 5. *Jason* implementation of waiter agents *(left)* and dining philosopher agents *(right)*

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. 5 shows the full executable implementation of the *Jason* project (available at CARTAGO web site). It accounts for a MAS file describing the multi-agent system initially composed by a `waiter` agent called `alfredo`, five `philo` agents called `rosa,beppo,pippo,maria,giulia`, working inside a common CARTAGO working environment. An artifact of type `Table` is dynamically created and exploited by the agents.

A brief description of the components follows. The `waiter` agent is responsible for creating a table identified by `myTable` with 5 chopsticks, and informing all the other agents which chopsticks should they use. 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 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). The source code of the philosopher in is quite intuitive: after receiving the information about the chopsticks to use, the philosopher starts a life-cycle interleaving thinking and eating. By thinking, a philosopher gets hungry. The belief to be hungry triggers the plan to eat: first, if it does not believe to own the chopsticks, then it suitably interacts with table to get them, by triggering the `getChops` operation and start observing the table. Note that the `use` action is not blocking: instead `sense` action can (optionally) block the agent control flow for a certain amount of time, waiting to get some stimuli on the specified sensor. If no perception are sensed in this amount of time, an sensing timeout belief is generated, and the philosopher sadly decides to die for starvation. When a philosopher perceives that the chopsticks have been acquired, then it can eat. After completing the eating activity, being no longer hungry, the philosopher releases the chopsticks by executing the `releaseChops` operation and starts thinking again.

3.4 Some Remarks

Some considerations are worth remarking. First the example is not meant to be as complex as real-world MAS working environments, and in particular it is not robust to possible failures as it should be—in particular to agent failures in giving back the chopsticks. Despite its simplicity, the example is useful to give an idea of the basic A&A philosophy in designing systems balancing the responsibilities among agents and artifacts. Simple alternative solutions to this one would account for having either an agent playing the role of the table, or avoiding a table and let agents coordinate through suitable conversation protocols.

With respect to the former one, the A&A approach makes it possible to avoid the need to design and implement parts of the system with "wrapper" agents though they are clearly not autonomous neither proactive. We think that this is very important both from a conceptual point of view—avoiding semantic gap between analysis and then design and implementation—but also a pragmatical point of view: it is intuitive that artifacts in general, despite specific cases, are entities largely more lightweight than agents: on the one side, artifacts are typically passive, with simple mechanisms to trigger and execute operations, and possibly changing the observable state and generating events; on the other side, agents typically encapsulate one or multiple control flows, and have complex machinery for managing knowledge, selecting pro-actively actions to do, and so on. So, adopting artifacts and not agents to represent automatised resources and tools can be effective from the point of view of maintenaince and performance—in using time and memory resources— in particular to scale with system complexity in terms of number of agents and artifacts involved.

With respect to the latter case, we think that the situation is pretty analogous to human working environments: not always the language and direct communication is the best way to coordinate the independent activities of individual. There are cases where well-designed coordination artifacts could be largely more effective, for instance enabling communication and coordination without requiring a strong temporal and spatial coupling between agents. Conversely, we think that the point is to find a way (models and theories) in MAS to use language—i.e.

direct communication—and artifacts in synergy, as happen in human contexts. Indeed, we consider this as one of the crucial points that would be worth investigating in future research on artifacts in MAS.

4 Conclusions and Future Works

The fact that the environment can play an important role in designing and programming MAS is now a well-known and accepted fact [16]: the point is now which kind of reference model we should adopt and systematically use to conceive and design a "good" environment for agent activities, so as to create cognitive MAS that suitably exploit such an environment to perform their individual and social activities. An important point here is *abstraction*: it is opinion of the authors that reference models should aim not merely at identifying mechanisms and/or architectures, but first of all at framing the issue of MAS environment in terms of new abstractions introduced with respect to the basic agent and MAS meta-model—and the related theoretical foundation.

By drawing inspiration from human cooperative working environments, in this paper we propose a general conceptual framework called A&A, which makes it possible to entail such design in terms of set of suitably designed artifacts, populating workspaces and constituting in the overall the MAS working environment. Then we discuss current technologies—among which the prominent one is called CARTAGO—that make it possible to prototype MAS applications exploiting artifact-based working environments. Finally, we consider the issue of integration of such technologies with existing MAS cognitive environments, by adopting as a reference case the *Jason* programming platform, towards scenarios in which MAS composed by intelligent agents—possibly developed with different agent programming languages or architectures—suitably share and exploit artifact-based working environments to interact and cooperate.

Indeed, in this paper we introduced and described just the basic—and somewhat simplest—points concerning A&A and the notion of artifacts in MAS: several other important points have not being considered either for lack of space or because they are still part of the future work. Among the many others, two main ones are worth to be pointed out here: *artifact composition* (linkability) and *intelligent use of artifacts*. Concerning this second point in particular, theoretical work on models and theories for the cognitive use of artifacts is still in its infancy. The objective here is to find on the one side suitable languages and theoretical frameworks to formally describe—in particular— the function of artifacts, their operating instructions and more generally artifact observable state, so as to make them useful and effective in agent reasoning; on the other side, revisiting agent reasoning model and techniques so as to exploit as much as possible the availability of working environments suitably designed to help their activities. Existing work on MAS in semantic web and the research work investigating human and autonomous agents reasoning on (real-world) tools [2] indeed could provide useful insights to face the problem.

References

1. Agre, P., Horswill, I.: Lifeworld analysis. Journal of Artificial Intelligence Reserach 6, 111–145 (1997)
2. Amant, R.S., Wood, A.B.: Tool use for autonomous agents. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI/IAAI 2005 Conference, Pittsburgh, PA, USA, July 9–13, 2005, pp. 184–189. AAAI Press / The MIT Press (2005)
3. Bordini, R., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica 30, 33–44 (2006)
4. Bordini, R.H., Hübner, J.F.: BDI agent programming in AgentSpeak using Jason. In: Toni, F., Torroni, P. (eds.) CLIMA 2005. LNCS (LNAI), vol. 3900, pp. 143–164. Springer, Heidelberg (2006)
5. Kirsh, D.: The intelligent use of space. Artif. Intell. 73(1-2), 31–68 (1995)
6. Kirsh, D.: Distributed cognition, coordination and environment design. In: European conference on Cognitive Science, pp. 1–11 (1999)
7. Nardi, B.A.: Context and Consciousness: Activity Theory and Human-Computer Interaction. MIT Press, Cambridge (1996)
8. Norman, D.: Cognitive artifacts. In: Carroll, J. (ed.) Designing interaction: Psychology at the human–computer interface, pp. 17–38. Cambridge University Press, New York (1991)
9. Omicini, A., Ricci, A., Viroli, M.: Agens Faber: Toward a theory of artefacts for MAS. Electronic Notes in Theoretical Computer Sciences 150(3), 21–36 (May 29, 2006), In: Proceedings of 1st International Workshop Coordination and Organization (CoOrg 2005), COORDINATION 2005, Namur, Belgium, (April 22, 2005)
10. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In: AAMAS 2004, vol. 1, pp. 286–293. ACM, New York (2004)
11. Ricci, A., Viroli, M., Omicini, A.: Programming MAS with Artifacts. In: Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F. (eds.) PROMAS 2005. LNCS (LNAI), vol. 3862, pp. 206–221. Springer, Heidelberg (2006)
12. Ricci, A., Viroli, M., Omicini, A.: CArtAgO: A framework for prototyping artifact-based environments in MAS. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2006. LNCS (LNAI), vol. 4389, pp. 67–86. Springer, Heidelberg (2007)
13. Schmidt, K., Simone, C.: Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. Computer Supported Cooperative Work 5(2/3), 155–200 (1996)
14. Viroli, M., Holvoet, T., Ricci, A., Schelfthout, K., Zambonelli, F.: Infrastructures for the environment of multiagent systems. Autonomous Agents and Multi-Agent Systems 14(1), 49–60 (2007)
15. Viroli, M., Omicini, A., Ricci, A.: Infrastructure for RBAC-MAS: An approach based on Agent Coordination Contexts. Applied Artificial Intelligence 21(4–5), 443–467 (April 2007) Special Issue: State of Applications in AI Research from AI*IA 2005
16. Weyns, D., Parunak, H.V.D. (eds.): Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Environment for Multi-Agent Systems 14(1) (2007)