A General-purpose Programming Model & Technology for Developing Working 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 as Activity Theory, 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 describe 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, [9]

"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 [9] 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 Horsewil in their *Lifeworld analysis* [1], as well as in the work of Kirsch [6, 7]. 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) [8]. 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 [17, 7].

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 [18]. 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 environments 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. Actually, artifacts can be suitable used to design and program working environment in MAS with heterogeneous agent types, including agents not necessarily classified as intelligent or cognitive.

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 remainder of the paper is organised as follows. First, we provide a description of the basic concepts and principles of A&A (Section 3), by introducing an abstract model embedding such principles (Section 3.3). Then, we briefly present the current models and technologies that have been developed for concretely prototyping MAS

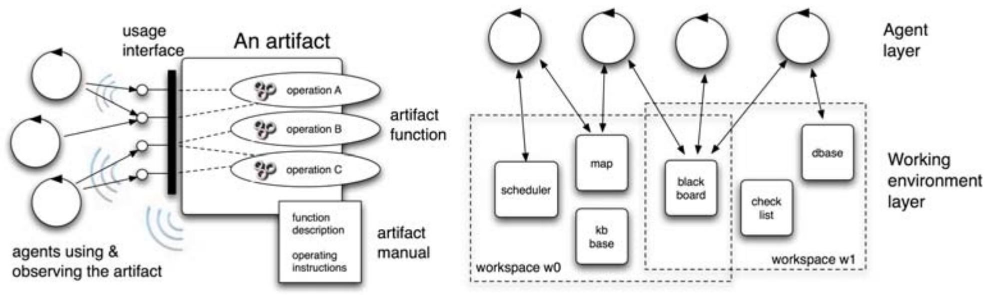

Fig. 1. *(Left)* Abstract representation of an artifact. *(Right)* Abstract representation of a working environment with two workspaces, with some artifacts of different kinds inside.

applications in the A&A perspective (Section 4), focussing on CARTAGO in particular. In Section 5 we discuss the issue concerning the integration of CARTAGO with existing MAS programming platforms, taking *Jason* as a case study. Finally, we conclude the paper with some final remarks, and sketch some future line of work (Section 6).

2 Related Works

Actually, the basic notion of artifact has been already introduced and published in previous papers [15, 11, 12], and the same applies for the first version of CARTAGO technology [16, 14]. On the one side—besides purely conceptual papers such as [11]—papers such as [15, 12] 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 platforms. In particular, in [15] the first model of artifact is implemented on top of ReSpecT tuple centres [10]. On the other side, the artifact programming model and its implementation in CARTAGO technologies has been substantially evolved with respect to the one described in [16]. In particular, the basic model of usage-interface and operation instructions presented in [16, 14] is quite simple and is not able to properly take into account the possibility to have the concurrent execution of operations on the same 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 changes the approach used by MAS programmers to program artifact operations and behaviour—is described in detail in this work. Besides this, the work published in [16] 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. Finally, in this is paper—as a new important contribution—a first concrete working example of integration with an existing MAS programming platform—that is *Jason*—is described.

3 A&A Programming Model Building Blocks

3.1 Artifacts and Workspaces

A *working environment* in A&A is defined as the part of the MAS that is designed and programmed by MAS programmers and dynamically instantiated and used by agents to support their working activities. 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 participate to one or multiple workspaces, and can work with those artifacts belonging to such workspaces only. 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 or MAS—can account for one or multiple workspaces, and an individual workspace can be either mapped onto a single node of the network or spread among multiple nodes.

The notion of artifact is the core abstraction of the programming model: it is meant to represent any passive 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. 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". More on this notion can be found in [15].

An abstract representation of an artifact is shown in Fig. 1 and it is very similar to artifacts as found in human society: artifact function—and related artifact behaviour—is partitioned in a set of *operations*, which agents can trigger by acting on artifact *usage interface*. The usage interface provides all the controls that make it possible for an agent to interact with an artifact, triggering and controlling the execution of operations and perceiving *observable events* generated by the artifact itself, as a result of operation execution and evolution of its state. 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.

Analogously to the human case, in A&A each artifact type can be equipped by the artifact programmer with a *manual* composed essentially by the *function description*—as the formal description of the purpose intended by the designer—, the *usage interface description*—as the formal description of artifact usage interface and observable states—, and finally the *operating instructions*—as the formal description of how to properly use the artifact so as to exploit its functionalities. Usage interface description is just a description of the controls—analogously to the description of which buttons and displays the coffee machine has—, while operating instructions describe the usage protocols, to exploit such controls. Such a manual is meant to be essential for creating open systems with intelligent agents that dynamically discover and select which kind of artifacts could be useful for their work, and then can use them effectively even if they have not pre-programmed by MAS programmers for the purpose.

It's worth remarking here the similarities and differences between the artifact abstraction and the general notion of *service*. On the one side, artifacts can be seen indeed as a natural way to implement services, without the need to *agentify* them as typically happens in service-based agent approaches. On the other side, typically services are conceived as a purely architectural concept: here instead, artifacts are meant to be the basic building block complimentary to the agent abstraction defining a new extended agent programming model. Then, typically a service can be designed and implemented on top of multiple artifacts and agents.

3.2 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 kinds of working activity, 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 the working environment that concern construction, sharing, and use of artifacts.

Despite of their specific model / architecture, in order to execute actions over an artifact and perceive observable events, agents must be *situated* in the 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. the part that is designed and programmed according to a certain kind of cognitive model / architecture—can sense and affect a working environment. Agent bodies are 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 model for the agent mind, including both intelligent agent and reactive / mobile / general agent models.

3.3 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. In the abstract model, each artifact is instance of some *artifact type*, which is the template specified by MAS programmers, defining artifact structure and behaviour. Each artifact instance has a unique logical name and a unique identifier, denoted in the following by AID. The logical name is given by the programmer, while the identifier is chose by the system. Also workspaces have a logic name and are identified by a unique workspace identifier, denoted by WspID.

Agent bodies are meant to contain *effectors* to perform actions upon the working environment, and a dynamic set of *sensors* to collect stimuli from the working environment. Different types and number of sensors can be attached to an agent body, each one identified by its own sensor identifier, denoted by SID.

The interface accounts for a (minimal) set of primitives that can be grouped as follows:

Artifacts construction, discovery and disposal

- createArtifact(Name,TypeID,Conf,WspID) — to create a new artifact called Name, of type TypeID, with a starting configuration represented by Conf, in the workspace identified by WsdID.
- getArtifactID(Name,WspID):AID — to get the identifier of an existing artifact, given its name
- disposeArtifact(AID) — to dispose an existing artifact

Workspace construction, discovery and disposal

- createWorkspace(Name) — to create a new workspace called Name in the working space
- getWorkspaceID(Name):WspID — to get the identifier of an existing workspace, given its logical name
- disposeWorkspace(WspID) — to dispose an existing workspace

Artifact use and observation

- execOp(AID,Op(Name,Params),SID) — to trigger the execution of an operation on an artifact given its identifier AID, specifying operation name and parameters, and the sensor, identified by SID, to collect corresponding events (if any) as generated by the artifact
- sense(SID,Filter,Timeout):Perception — to actively perceive a possible observable event collected by the sensor SID, applying some kind of filter Filter, for a timeframe of maximum Timeout time units;
- focus(AID,SID) — to start observing persistently an artifact identified by AID, collecting all the possible observable events generated by the artifact on the sensor identified by SID;
- unfocus(AID,ID) — to stop observing an artifact, previously focussed by a focus action.

Artifact inspection

- getFD(AID):FD — to retrieve the function description representation of a specific artifact;
- getOI(AID):OI — to retrieve the operating instructions representation of a specific artifact;
- getObsState(AID):ObsState — to retrieve the observable state representation of a specific artifact.

Sensors management

- linkSensor(SensorType):SID — to link a new sensor of type SensorType to the agent body, getting an identifier of it;
- unlinkSensor(SID) — to unlink a previously linked sensor from the agent body.

The primitives are modelled as *actions* available to the agent mind to pilot the agent body and, at the end, to play inside the working environment. All actions are meant to be atomic, i.e. executed as atomic steps of the agent activity, and in general can succeed—eventually with a result information—or fail. Note that 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 5.

3.4 The Artifact Programming Interface

Besides the API to use artifacts, a programming model for defining artifact types is given. An artifact type or template defines the structure and behaviour of artifact 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, instance variables (or fields) define 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.

In order to enforce mutual exclusion in updating artifact state on the one side—as in the case of monitors—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, and also the same operation can have multiple calls that are executed concurrently. For each step *a guard* can be defined, which specifies when the step—once it has been triggered 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 when an agent executes an execOp action, then it is actually executed when the guard is satisfied. A step can trigger other steps. An operation is considered completed when there are no more triggered steps whose execution is pending. 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, it's worth remarking here the strong, 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, since it is not a method or function call like in object-oriented programming. In other words, the execution of an execOp action for the agent over an artifact is never blocking: the concrete execution of operations (steps) is served not by a thread of control of the agent.

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 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*, whose value can change dynamically, during artifact functioning. 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 if the operation is (in that moment) part of the usage interface; if the operation is not part the usage interface, the agent action fails. It's worth remarking that such observable state notion can be used only when necessary: typically simple artifacts have a single implicit observable state, which needs not to be explicitly specified.

Finally, analogously to the agent API, some primitives are available to program artifact behaviour:

– genEvent(EventType, { Content }) — to generate an event of type EventType, possibly carrying an information content given by Content. The event is made observable to the agent that triggered the operation where the primitive is used, and to all the agents focussing (observing) the artifact.
– setObservableState(StateID) — to set current observable state of the artifact to be StateID.

The abstract model will be more clear by considering concrete examples provided in next section, describing first technologies implementing such a programming model.

4 Prototyping Technologies: CARTAGO

Starting from the A&A abstract model, some technologies have been developed, 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 fist one is is called CARTAGO (Common ARtifact Infrastructure for AGent Open environment), and it is a platform for programming and running artifact-based working environment. A second one is called simpA (simple A&A programming environment), and it is platform 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].

CARTAGO has been already introduced in previous papers [16, 14], and its model and technology—briefly described in the following—have been substantially evolved since such first steps.

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

The CARTAGO architecture implements quite faithfully the abstract model described in Section 3.3. 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 platform. CARTAGO is composed by three main parts:

– *API for creating and interacting with artifact-based working environments* — This API is meant to extend the existing basic set of agent actions with new ones, abstractly described in Section 3.3, 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* — This API allows 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`[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 `genEvent`, 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, a first one—triggered when the `execOp` action is invoked—and a second one—`opStepA`—triggered by the first with the invocation of the `nextOp` primitive and executed when the guard `canExecA` is true. 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. Some tools are also made available in CARTAGO for online inspection of working environment state, in particular arti-

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

```
import alice.cartago.*;

class MyArtifact extends Artifact {
  private int m;

  public MyArtifact(){
  }

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

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

execOp(id, new Op("op1"),sid);

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

Fig. 2. *(Left)* A simple type of artifact, with two operations, `op1` and `op2`, the former composed by two steps, a first one—triggered when the `execOp` action is invoked—and a second one—`opStepA`—triggered by the first with the invocation of the `nextOp` primitive and executed when the guard `canExecA` is true. *(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.

fact state, and above the observation of artifact behaviour, in terms operation executed and events generated.

The support for workspaces is still under development at the time of writing, and will be part of next CARTAGO release. Further details about CARTAGO API and architecture, along with complete examples, can be found on the web site.

5 Integration with Existing MAS Programming Frameworks

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]—do not directly provide high-level rich abstractions to model and design the notion of environment, in particular environment as a set of suitably resources available to agents. 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 (possibly cognitive) agents written in existing programming environments to create, share and use articulated working environments, enabling MAS engineers to directly design and program 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 coordination artifacts. Generally speaking, from a conceptual point of view it would be possible and interesting to build MAS applications composed by heterogeneous agent societies, made of 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.

A key component for the integration is played by the agent body, acting as an interface between the agent mind and CARTAGO artifact based working environment. So, conceptually the integration with any possible MAS programming framework could be possible in principle by extending the basic set of agent actions with those for accessing and controlling the agent body.

In the following subsection we consider a concrete case, which concerns the integration of CARTAGO with the *Jason* agent-oriented programming platform.

5.1 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 [5] or Jadex [13].

Jason natively supports a general notion of customisable environments—called "simulated environments"—, which can be programmed by users specifying which actions are available to agents, their effect, and the perceptions generated. In particular, a user can provide its own implementation of a simulated environment by defining a Java class that extends the *Jason* `Environment` class, overriding the methods `getPercepts` and `executeActions`, establishing the agent perceptions generated by the environments and the effects of action execution.

The integration with A&A concepts and CARTAGO technology in particular is then straightforward—both from a theoretical and pragmatical point of view—though quite useful. From a theoretical point of view, integrating CARTAGO accounts for extending the basic set of agents external actions and related perceptions, including the one listed in Section 3.3. From a pragmatical point of view, this is done simply by defining a class, called `CartagoEnvironment`, extending *Jason* `Environment` class and suitably overriding `getPerceptions` and `executeActions` methods so as to work as adapter for exploiting CARTAGO API. The class is not show here for lack of space: the interested reader can find it on CARTAGO web-site. `CartagoEnvironment` is a base class for defining application-specific environments, which specifies in its constructor the initial configuration of the working environment when booting the system, in particular in terms of the initial set of artifacts available to agents.

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 3.3. In particular, a *Jason* agent can create artifacts through `createArtifact` action, link / unlink sensors to its body through `linkSensor` and `unlinkSensor`, can use an artifact by means of `execOp` and perceive artifact events collected by its sensors through `sense` action, and so on. Perceptions that are fetched by `sense` action are mapped to beliefs of the type `artifact_perception(Type,SensorId,ArtifactId,EventTime)`, while exceptions regarding timeouts elapsed during sense actions to beliefs of the kind `sensing_timeout(SensorID)`.

5.2 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 and *the table*—managing the set of chopsticks—as the coordination artifact that agents share and use to perform their (eating) activities. It is quite natural to encapsulate in the table artifact the enactment of the social policy that makes it possible both to satisfy mutual exclusion for the access on the individual chopsticks, and—the most challenging part—to avoid deadlock situations. In particular deadlock is avoided since a chopstick is released to a philosopher if and only if (when) both the chopsticks required by the agent are available are available.

Fig. 3 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 `alfred`, 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). The source code of the philosopher 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 `execOp` 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 dies 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.

The example, despite being simple, is quite effective in showing the usefulness of designing suitable coordination artifacts in programming MAS. At a more conceptual level, it remarks that—analogously to what happens in human working environments—not always the language and direct communication are 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 co-ordination 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 happens 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.

As a last remark, a different solution with a table artifact providing operations to get and release chopsticks individually—so with `getChop` and `releaseChop` in the usage interface—could have been used, with a slightly different coordinating policy inside (based on allowing only N-1 philosophers to have access to chopsticks at a time, if N is the number of seats): despite of the specific implementation, the important point here is that you can model such issue with explicit suitable programmed abstraction.

6 Conclusions and Future Works

The fact that the environment can play an important role in designing and programming MAS is a quite accepted fact [19]: 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.

By drawing inspiration from human cooperative working environments, in this paper we proposed 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 discussed 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 considered 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 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 theyr still part of the future work. Among the many others, two main ones are worth to be pointed out here: *intelligent use of artifacts* and *artifact composition* (linkability).

About the former point, the 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.

For the latter point, introducing principles and models for artifact composition is essential if we aim at applying A&A to large complex systems. Again, human working environments are a great source of inspiration. In our society, most of the artifacts and tools in our working environments have been designed to be composable with other artifacts, to create more complex articulated artifacts. So, intuitively, besides a usage interface that is meant to be exploited by artifact cognitive users, artifacts can be equipped by an other kind of interface, a sort of *link interface*, which would make it possible to link the artifact with other ones, for I/O interactions. It is clear that the properties of such a linking interface are in principle different from the properties of the usage interface. Also for this issue, existing literature—in particular in the context of CSCW [17]—will provide useful insights to investigate the problem.

References

1. P. Agre and I. Horswill. Lifeworld analysis. *Journal of Artificial Intelligence Reserach*, 6:111–145, 1997.
2. R. S. Amant and A. B. Wood. Tool use for autonomous agents. In M. M. Veloso and S. Kambhampati, editors, *AAAI/IAAI'05 Conference*, pages 184–189, Pittsburgh, PA, USA, 9–13 July 2005. AAAI Press / The MIT Press.
3. R. Bordini, L. Braubach, M. Dastani, A. E. F. Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, pages 33–44, 2006.

4. R. Bordini and J. H ubner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *CLIMA VI*, volume 3900 of *LNAI*, pages 143–164. Springer, Mar. 2006.
5. K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4), 1999.
6. D. Kirsh. The intelligent use of space. *Artif. Intell.*, 73(1-2):31–68, 1995.
7. D. Kirsh. Distributed cognition, coordination and environment design. In *European conference on Cognitive Science*, pages 1–11, 1999.
8. B. A. Nardi. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996.
9. D. Norman. Cognitive artifacts. In J. Carroll, editor, *Designing interaction: Psychology at the human–computer interface*, pages 17–38. Cambridge University Press, New York, 1991.
10. A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.
11. A. Omicini, A. Ricci, and M. Viroli. *Agens Faber*: Toward a theory of artefacts for MAS. *Electronic Notes in Theoretical Computer Sciences*, 150(3):21–36, 29 May 2006.
12. A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, and L. Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *AAMAS'04*, volume 1, pages 286–293, New York, USA, 19–23July 2004. ACM.
13. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A bdi reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. Seghrouchn, editors, *Multi-Agent Programming*. Kluwer Book, 2005.
14. A. Ricci, M. Viroli, and A. Omicini. *Construenda est* CArtAgO: Toward an infrastructure for artifacts in MAS. In R. Trappl, editor, *Cybernetics and Systems 2006*, volume 2, pages 569–574, Vienna, Austria, 18–21 Apr. 2006. Austrian Society for Cybernetic Studies. 18th European Meeting on Cybernetics and Systems Research (EMCSR 2006), 5th International Symposium "From Agent Theory to Theory Implementation" (AT2AI-5). Proceedings.
15. A. Ricci, M. Viroli, and A. Omicini. Programming MAS with artifacts. In R. P. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3862 of *LNAI*, pages 206–221. Springer, Mar. 2006. 3rd International Workshop (PROMAS 2005), AAMAS 2005, Utrecht, The Netherlands, 26 July 2005. Revised and Invited Papers.
16. A. Ricci, M. Viroli, and A. Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for MultiAgent Systems*, volume 4389 of *LNAI*, pages 67–86. Springer, Feb. 2007.
17. K. Schmidt and C. Simone. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Computer Supported Cooperative Work*, 5(2/3):155–200, 1996.
18. M. Viroli, T. Holvoet, A. Ricci, K. Schelfthout, and F. Zambonelli. Infrastructures for the environment of multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):49–60, 2007.
19. D. Weyns and H. V. D. Parunak, editors. *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Environment for Multi-Agent Systems*, volume 14(1). Springer Netherlands, 2007.

```
MAS cartagoTest {

  infrastructure: Centralised

  environment: CartagoEnvironment

  agents:
    rosa philo.asl;
    beppo philo.asl;
    pippo philo.asl;
    maria philo.asl;
    giulia philo.asl;
    alfred 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;
    genEvent("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;
  }
}
```

Fig. 3. *(Left)* Definition of a Jason MAS called `cartagoTest`, composed by five philosopher agents—`rosa`, `beppo`, `pippo`, `maria`, `giulia`—and a waiter agent—`alfred`—, running on top of a CARTAGO environment, implemented by `CartagoEnvironment`.*(Right)* Definition of the `Table` artifact type.

```
/* 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") ;
      execOp(Table,getChops(C1,C2),mySensor);
      +chopsticks_requested(C1,C2);
      sense(mySensor,8000).

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

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

Fig. 4. *Jason* implementation of waiter agents *(left)* and dining philosopher agents *(right)*.