Construenda est CArtAgO: Toward an Infrastructure for Artifacts in MAS

Alessandro Ricci, Mirko Viroli, Andrea Omicini
DEIS, Alma Mater Studiorum, Università di Bologna
Via Venezia 52, 47023 Cesena, Italy
{a.ricci, mirko.viroli, andrea.omicini}@unibo.it

Abstract

Artifacts have been recently proposed as first-class abstractions to model and engineer general purpose computational environments for Multi-Agent Systems. In this paper, we first consider the role of infrastructures in supporting the artifact conceptual framework, discussing motivations and requirements, and then we propose an abstract model and architecture of a reference infrastructure for artifacts, here named CArtAgO.

1 Introduction

Artifacts have been recently proposed as first-class abstractions to model and engineer computational environments in software Multi-Agent Systems (MASs) [Ricci *et al.*, 2005; Viroli *et al.*, 2005]. The background vision—which is shared with other recent approaches in MAS literature (see [Weyns *et al.*, 2005a] for a survey)—is that the environment can play a fundamental role in engineering MAS: on the one hand it is a suitable locus where engineers can embed responsibilities, impacting on MAS design and development; on the other hand it is a source of structures and services that agents can suitably use at runtime to support and improve their activities, in particular social ones.

The artifact conceptual framework promotes a methodology for modelling and engineering such computational environments. Artifacts can be generally conceived as *function-oriented* computational devices—i.e. designed to provide some kind of *function*[1]—, that agents can exploit to support their individual and collective (social) activities [Ricci *et al.*, 2005]. Such a view directly impacts on the theories of interaction and activities in agency: a MAS is conceived as an (open) set of agents that develop their activities by *(i)* computing, *(ii)* communicating with each other, and *(iii) using* (constructing) shared artifacts, which embody their computational environment. Generally speaking, artifacts can be either the target (outcome) of agent activities, or the *tools* that agents use as media to support such activities, reducing the complexity of their tasks. For instance, *coordination artifacts* [Omicini *et al.*, 2004] are artifacts providing coordination functionalities (such as blackboards, tuple spaces or workflow engines).

The conceptual and theoretical background of one such framework stems from the theories developed in the context of human science, in particular Activity Theory [Nardi, 1996] and Distributed Cognition [Kirsh, 1999]. Also, this perspective shares the aims and the principles developed in existing research work in Distributed Artificial Intelligence about theories of interaction [Agre, 1995]—in particular with the work of Phil Agre and Horswill [Agre and Horswill, 1997]—, and in Computer Supported Cooperative Work, with the notion of *embodied interaction* by Dourish [Dourish, 2001].

From an engineering point of view, artifacts along with agents become basic building blocks to design and develop MASs (or applications as MASs): designers can use agents to model autonomous activities, typically goal / task oriented, and artifacts to model structures, objects, typically passive and reactive entities which are constructed, shared and used in the execution of such activities. The artifact abstraction provides then a natural way to model object-oriented (OO) and service-oriented abstractions (objects, components, services) at the agent level of abstraction, bridging the conceptual and semantic gaps between the paradigms. As in the case of objects and services, artifacts expose interfaces composed by operations that can be invoked by agents—though relying on a different semantics.

In order to stress the validity of the artifact conceptual framework, and to extend and evolve it, we consider useful to setup a first simple *infrastructure*—referred here to as CArtAgO—to be concretely used for engineering MAS applications exploiting such abstractions, embodying the basic concepts related to artifacts.

Infrastructures play an essential role for keeping abstractions alive from design to runtime [Gasser, 2001]. Agent infrastructures (or middleware) typically provide fundamental services for agent creation, management, discovery and (direct) communication: well-known examples are RETSINA [Sycara *et al.*, 2003] and JADE [Bellifemine *et al.*, 2001]. Analogously, CArtAgO is meant to be exploited for creating and sustaining the existence at runtime of computational environments engineered in terms of artifacts, providing on the one side basic services for agents to instantiate and use artifacts, on the other side a flexible way for MAS engineers (and possibly agents) to design and construct any useful kind

[1]The term function here should not be confused with its meaning in programming languages, and should refer to the meaning that is generally used in human sciences such as sociology and anthropology, as well as in some recent work in AI [Amant and Wood, 2005]

of artifact.

The objective of this paper is to define an abstract model and architecture for CArtAgO, which can be suitably exploited to drive a concrete reference implementation. Among the related works, worh mentioning are Odell and Parunak's general abstract model for environments [Odell *et al.*, 2003], Weyns, Holvoet and Schelfthout's work on software architectures to design environments in situated MAS [Weyns *et al.*, 2005b] and Ferber and Muller's works on models for situated MAS [Ferber and Müller, 1996]. A comprehensive survey can be found in [Weyns *et al.*, 2005a].

The abstract model and architecture of CArtAgO described in the remainder of the paper have been developed by considering some few basic requirements that we wanted for such an infrastructure: *minimality* and *extensibility*—we aim at identifying a minimal core by considering a first basic set of features and properties identified in the artifact conceptual framework, to have simple ways to construct, select, and use artifacts, but easily extensible to support more advanced features not considered in this paper, such as composition and on-line management; *neutrality*—the infrastructure should be as far as possible neutral with respect to agent models, architectures, platforms, in order to be integrated and exploited with (existing) heterogeneous agent infrastructures; *heterogeneity*—the infrastructure should be as far as possible open to different ways (models, languages, frameworks) adopted to implement artifacts. In particular, it should be natural to reuse and wrap existing object-oriented and (web) services technologies, enabling their explicit representation and flexible exploitation at the agent level.

2 CArtAgO Abstract Model

In this section we define an abstract model of the main aspects on which CArtAgO is based, starting with a model for actions and perceptions linking agents to their computational environment, then introducing an abstract model for artifacts—as basic bricks to engineer such environments—, and finally for workspaces, as logical contexts where artifacts and agents are situated.

2.1 Actions & Perceptions

As a premise to all the other aspects, the artifact framework calls for introducing a model (and a theory) of interaction different from the models generally adopted in software agent infrastructures—typically based solely on communicative acts—, and more similar to models defined in autonomous / situated agents. Agents interact with their computational environment by means of suitable *actions* provided for artifact construction, selection and usage, and by perceiving *observable events* generated from such artifacts: we refer to such a kind of actions here as *pragmatic acts*. Pragmatic and communicative actions constitute agent *cognitive acts*. The conceptual foundation of this notion of actions is inspired by the notion of cognition as defined in theories based on the concept of *autopoiesis* and *structural coupling* [Varela, 1981]—developed in theories concerning living systems, from biology to social sciences—where actions and perceptions are strictly connected. From this perspective, we consider agents as autopoietic entities, which interact with their environment—and artifacts in particular, as allopoietic entities— by executing operations that result in both changes in the environment and in the agents executing the actions, as a form of structural coupling. A primary example of action is the execution of an operation on a specific artifact, whose effects can be the generation of streams of events distributed in time.

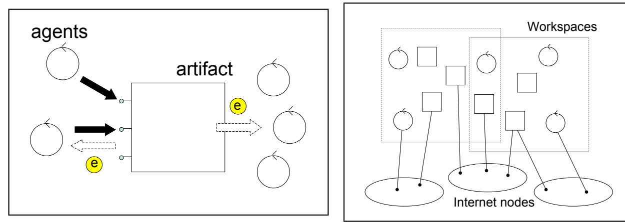

Figure 1: *(left)* Agents interact with artifacts by invoking operations on their usage interfaces and by perceiving observable events generated by them. *(right)* Agents and artifacts are situated in workspaces, spread over network nodes.

As in the classic agent model [Russel and Norvig, 2003], agents perceive events through *sensors*, as collectors of environment stimuli. In CArtAgO, sensors are structures provided by the infrastructure that agents can flexibly create and use to partition and control the information flow perceived from artifacts, possibly providing specific functionalities such as buffering, filtering and ordering, managing priorities. *Sensing* is the internal action that agents execute on their sensors to become aware (perceive) of the events (stimuli) collected by the sensors.

2.2 Artifacts

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

Identity

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

Usage Interface & Events

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

Differently from interfaces as found in OO or service-oriented paradigms, operations in usage interfaces have no return value: this aspect is modelled as observable

information (event) that can be perceived by agents after executing the operation. Also exceptions—as error conditions generated during operations execution—are modelled as observable events.[2]

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

Function Description & Operating Instructions

In order to support a rational exploitation of artifacts by intelligent agents, each artifact is equipped with a *function description*, i.e. an explicit description of the functionalities it provides, and *operating instructions*, i.e. an explicit description of *how* to use artifact to get its function, for instance in terms of the *usage protocols* that the artifact support. Such information is meant to be useful for cognitive agents that—by suitably inspecting and understanding them—can *(i)* dynamically reason about what artifacts can be selected to support their activities, and *(ii)* use operating instructions to support activity execution, making easier to set up plans and reason about expected behaviour of artifacts. We consider such issues of foremost importance, at the core of the notion of computational environments designed to support the activities of agents—in particular cognitive / rational agents. Actually, the research on these aspects, in particular on formal models and languages that can be used to specify function description and operating instructions, and their injection in existing agent reasoning architectures (such as BDI), is still in its infancy: first work can be found in [Viroli and Ricci, 2004].

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

Observable State

Artifacts are stateful reactive entities, with a state that can change according to the operations executed by agents.[3]

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

[2]It is worth remarking that operations are not methods as defined in the OO case, since agents encapsulate their control flows. In the OO paradigm, the execution of a method causes the control to flow with the data (parameters) from the object invoking the method to the object owning the method.

[3]Actually, also time- and location- aware artifacts can be considered, i.e. artifacts whose state can can change also also in reaction to—respectively—time passing (e.g. a clock) and artifact location changes.

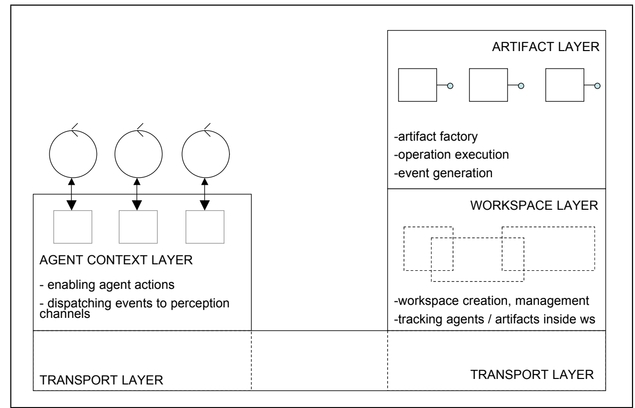

Figure 2: CArtAgO abstract architecture.

2.3 Workspaces

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

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

3 An Abstract Architecture

Based on the previous abstract model, the abstract architecture of CArtAgO can be described in terms of logic layers, distinguishing the agent side and the artifact side (see Figure 2).

On the agent side we have (top-down): an agent context layer and a transport layer. The *agent context layer* acts as a bridge between agents and CArtAgO computational environments, and is responsible for the allocation and management of *agent contexts*. The notion of

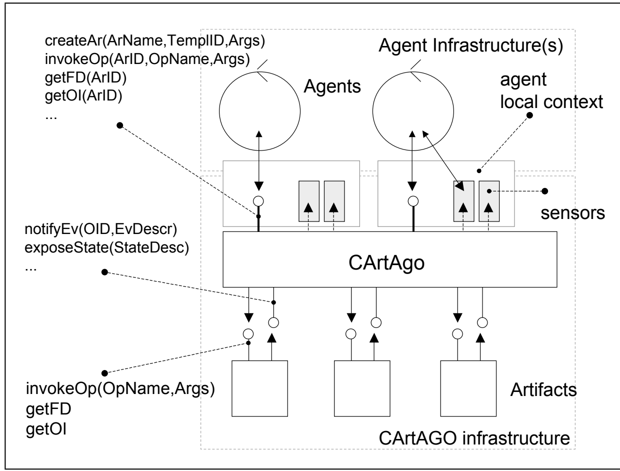

Figure 3: An overview of the interfaces realising the services.

`getContext(AgID,CtxDescr): CtxID`
`getWsID(WsName,{Location}):WsID`
`joinWS(WsID)`
`exitWS(WsID)`
`releaseContext(CtxID)`

`createAr(ArName,TemplID,[Args],{WsID}):ArID`
`getArID(ArName,{WsID}):ArID`
`registerAr(ArID,WsID)`
`deregisterAr(ArID,WsID)`
`disposeAr(ArID)`

`createSensor: SensorID`
`invokeOp(ArID,OpName,[Args],{SensorID}): OpID`
`sense({SensorID},Timeout): EventDescr`

`getFD(ArID): FdDescr`
`getOI(ArID): OIDescr`
`getUID(ArID): UIDDescr`
`getState(ArID): StateDescr`

`createWS(WsName,Location):WsID`
`addWSNode(WsID, Location)`
`removeWSNode(WsID, Location)`
`disposeWS(WsID)`

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

agent context is introduced to both explicitly represent a temporal notion of *session* (working session) of a specific agent, and as a local computational environment encapsulating CArtAgO structures that are private to the individual agents, such as sensors. The agent context acts as a bridge between the individual agent and the environment where the agent plays, providing the very basic interface to act inside CArtAgO, and in the overall defining the basic set of actions and perceptions allowed for such an agent. Agents contexts are useful to explicitly model the dynamics of agent working sessions: an agent starts its working session inside workspaces by first obtaining its context, which is dynamically constructed by the infrastructure, configured according to organisation and security issues (not dealt in this paper), and then released to the agent. Then, by means of the context the agent can execute actions and perceive events. So, the agent context layer on the one side provides the overall primitives (actions) to agents for exploiting CArtAgO, on the other side it is responsible of the dispatching of events to agents, through the perceptions channels in their agent contexts. The *transport layer* below the agent context layer is responsible of connecting agent contexts to workspaces and artifacts, dispatching actions, operations, events according to the topology, exploiting the available network infrastructure(s).

On the artifact side we can identify (top-down): an artifact layer, a workspace layer, and the same transport layer. The *artifact layer* is responsible of artifacts management. On the one side, the layer acts as *(i)* artifact factory, to create / dispose artifacts, and *(ii)* operation executor, executing operations requested by agents on artifacts; on the other side, it provides basic operations to be exploited by artifacts to generate events and to expose their observable state. The *workspace layer* is responsible to manage workspaces (creation, disposal, tracking agents and artifacts inside workspaces, etc.), providing basic services that are exploited by specific agent actions.

4 Basic Services

The abstract description of CArtAgO is completed by defining the abstract specification of the core services (API) provided by the infrastructure, on the one side used by agents to work with artifacts, on the other side useful for programming artifacts. Figure 3 shows the relationships between the abstract architecture and such interfaces.

4.1 Agent Side

Services are exploited by agents executing suitable cognitive actions, whose execution yields immediately a result and possibly a stream of events distributed in time, perceived by agents through their sensors. The abstract specification of the basic core of actions is reported in Table 1.

In order to start a working session, an agent must first execute a *getContext* action, obtaining its local context, enabling its participation to the computational environment. Once got a local context, an agent can join one or more workspace(s), in order to create or use artifacts situated there. For this purpose, *joinWS* action makes it possible to enter a workspace, by specifying the workspace id, and *exitWS* to exit. *getWsID* action can be executed to get the workspace identifier, specifying the workspace name and possibly one of the (network) nodes where the workspace is located.

The remainder of the actions are grouped according to the basic aspects that characterise agent / artifact relationships, i.e. artifact selection, use and construction.

Artifacts Selection

Agents can dynamically inspect the function description of an artifact to evaluate its possible usage in their activities. For the purpose, a basic action *getFD* can be used, specifying as a parameter the specific artifact whose function description should be read.

An important and articulated issue to be considered here—besides the retrieval of the function description of a *known* artifact— is artifact *discovery*, i.e. how agents can be aware of the available artifacts in their contexts (workspaces). A natural way to design such a discovery service accounts for instrumenting workspaces with suitable infrastructure agents and artifacts, available by

default, with some conventional names. This solution is common in agent infrastructures, with agent facilitators (such as the directory facilitator). By adopting a minimal approach, we instrument each workspace with a simple artifact called *artifacts_registry*, which can be used by agents to know what artifacts are actually present in the workspace. *artifacts_registry* could provide also basic services for artifact searching and match-making, by keeping track—beside the names and the ids— also of the function descriptions and possibly of the operating instructions. For flexible and articulated discovery services a good design strategy accounts for keeping *artifacts_registry* simple and introducing a suitable facilitator agent which would interact with agents requesting the services and use the registry accordingly.

Artifacts Use

These services account for using artifacts, i.e. invoking operations and processing events.

To execute an operation the action *invokeOp* is provided, specifying the artifact id, the operation name, the parameters, possibly the specific sensor where to collects events related to the operation execution (including the possible results of the operation). If no sensor is specified, then the events are collected in the default sensor of the agent. An identifier is generated by the infrastructure to uniquely denote the invocation executed, in order to be possibly used by the agent to get information about the state of this operation execution. The invocation of an operation can fail, for instance due to the unavailability of the artifact. This kind of failures should be distinguished from errors that can raise when executing the operation on the artifact and that depends on the specific semantics of the operation: such errors are made observable to the agents as events.

Specific services are provided to manage sensors, in particular to create a sensor (*createSensor*), to inspect it in order to sense and process the events (*sense*). It is worth remarking that sensors are meant to be local to the individual agent, not-shared among agents, and collected/managed in agent contexts.

Finally, in order to support a *cognitive* use of artifacts, actions are provided to inspect their operating instructions (*getOI*), their full usage interface (*getUID*) and also their dynamic observable state (*getState*). The results of these actions (if no errors occur) are machine-readable documents—XML-based, for instance— containing the specific description, according to some kind of (formal) semantics.

It is worth remarking the strong difference between communication taking places between agents through speech acts and interaction taking place between agents using artifacts: in particular, in the first case interaction is at the *knowledge level* [Simon, 1996], in the latter case the interaction is at the *operational level*, since specified and constrained by artifact interface.

Artifacts Construction & Manipulation

Artifacts are meant to be created and manipulated dynamically. Artifact creation is realised by the action *createAr*, specifying the logic name of the artifact, the template to be used for driving its creation, parameters needed for artifact creation and optionally the workspace id where the artifact should be created. The action result—if no errors occur—is the artifact identifier. The identifier of an existing artifact can be obtained by *getArID*, specifying the artifact name and (possibly) its location (workspace). By omitting the location, current workspace(s) where the agent is situated are considered. An explicit action is provided to dispose artifacts (*disposeAr*), specifying its id.

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

Besides creating and manipulating artifacts, some basic actions are provided to create and manage workspaces, defining dynamically the topology of the computational environment on top of network nodes. Workspace action is realised by *createWS* action, specifying a name for the workspace and possibly a network node where the workspace should be created, yielding as a result a workspace identifier. Since a workspace typically spans on multiple network nodes, *addWSNode* action is provided to dynamically extend an existing workspace on a specific network node. Conversely, *removeWSNode* remove a workspace from the specified network node. Workspace complete disposal is realised by *disposeWS* action.

Actually, artifact management includes several important aspects—including artifact initial configuration management, artifact online monitoring and debugging, artifact dynamic behaviour tuning and adaptation—which are not considered in this paper for lack of space.

```
getOpID: OpID
genEv(EventDescr)
genEv(OpID,EventDescr)
genEvWs(EventDescr)
exposeState(StateDescr)
```

Table 2: Basic services for artifact programming: observable event generation & observable state exposition.

4.2 Artifact Side

To support artifact programming CArtAgO provides some basic primitives useful for event generation (*genEv*) and (observable) state exposition (*exposeState*) (see Table 2), to be exploited in the body of operations.

Each operation request served by the artifact is labelled by a unique operation identifier (type *OpId* in the tables). An event can be then generated specifying the operation identifier to which it must be related, as observable effect of this operation (and of the agent action that caused it). If no *OpId* is specified, the event is considered related to the current operation request. The event is then dispatched to the agent that invoked the operation. An *OpId* can be retrieved by invoking the primitive *getOpID* during the execution of the operation (as part of its execution body). As a key aspect for designing artifact interactive behaviour, operation identifiers can be stored, collected, and flexibly used in *genEv* primitives when necessary, during artifact functioning, across operation executions.

If order to model the generation of events which are not directly related to any specific agent operation request, the primitive *genEvWs* is provided, generating an event which is observed by all the agents residing in the same workspace(s) of the artifact. Table 3 shows

```
artifact TupleSpace {
  Bag<Tuple> tuples;
  List<Query> pending_reqs;

  operation out(Tuple t){
    if (pending_reqs.isEmpty()){  tuples.add(t); }
    else {
      OpID id = remove_matching_req(pending_reqs,t);
      if (id != null){ genEv(id,tuple_removed(t)); }
    }
  }
  operation in(TupleTemplate tt){
    Tuple t = remove_matching_tuple(tuples,tt);
    if (t != null){ genEv(tuple_removed(t)); }
    else {
      OpId opId = getOpId();
      pending_reqs.add(query(opId,tt);
    }
  }
  operation rd(TupleTemplate tt){ ... }
}
```

Table 3: Pseudo-code of a simplified version of a tuple space artifact. The `out` operation inserts a tuple in the tuple space, `in` and `rd` respectively removes and reads a tuple matching a tuple template.

the object-oriented / Java-like pseudo-code of a simplified version of a tuple space implemented as an artifact, showing the use of *genEv* and *getOpID* primitives.

5 Concluding Remarks

In this paper we described in detail the abstract model and architecture for a basic infrastructure for supporting artifacts in MAS, focussing on basic core issues.

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

A possible roadmap for the development of the project CArtAgO could finally be sketched as follows: *(i)* setting up a first open-source Java-based reference implementation, to be used with existing agent platforms, such as JADE; *(ii)* developing some basic kind of general purpose artifacts, in particular bridges wrapping existing MAS coordination models / technologies, such as TuCSoN tuple centres [Omicini and Zambonelli, 1999], and standard specification / technologies, such as Web Services; *(iii)* establishing first models and ontology for defining function descriptions, operating instructions, and observable state description, possibly reusing existing research efforts on service description models and (standard) languages, such as OWL-S; *(iv)* extending the basic model to consider also organisation and security issues, for instance defining role-based models to explicitly define policies ruling agent access and use of artifacts, according to their role.

References

[Agre and Horswill, 1997] Phil Agre and Ian Horswill. Lifeworld analysis. *Journal of Artificial Intelligence Reserach*, 6:111–145, 1997.

[Agre, 1995] Phil Agre. Computational research on interaction and agency. *Artificial Intelligence*, 72(1-2):1–52, 1995.

[Amant and Wood, 2005] Robert St. Amant and Alexander B. Wood. Tool use for autonomous agents. In Manuela M. Veloso and Subbarao Kambhampati, editors, *20th National Conference on Artificial Intelligence / 17th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 2005)*, pages 184–189, Pittsburgh, PA, USA, 9–13 July 2005. AAAI Press / The MIT Press.

[Bellifemine *et al.*, 2001] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE: a FIPA2000 compliant agent development environment. In *5th International Conference on Autonomous Agents (Agents 2001)*, pages 216–217, Montreal, Quebec, Canada, 28 May–1 June 2001. ACM Press.

[Dourish, 2001] Paul Dourish. *Where the action is*. The MIT Press, 2001.

[Ferber and Müller, 1996] Jacques Ferber and Jean-Pierre Müller. Influences and reaction: a model of situated multiagent systems. In *2th International Conference on Multiagent Systems (ICMAS'96)*, pages 72–79, Kyoto, Japan, 1996. AAAI Press.

[Gasser, 2001] Les Gasser. MAS infrastructure: Definitions, needs, and prospects. In Thomas Wagner and Omer Rana, editors, *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, volume 1887 of *LNAI*, pages 1–11. Springer, 2001.

[Kirsh, 1999] David Kirsh. Distributed cognition, coordination and environment design. In *European conference on Cognitive Science*, pages 1–11, 1999.

[Nardi, 1996] B. A. Nardi. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996.

[Odell *et al.*, 2003] James J. Odell, H. Van Dyke Parunak, Mitch Fleischer, and Sven Breuckner. Modeling agents and their environment. In Fausto Giunchiglia, James Odell, and Gerhard Weiss, editors, *Agent-Oriented Software Engineering III*, volume 2585 of *LNCS*, pages 16–31. Springer, 2003.

[Omicini and Zambonelli, 1999] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.

[Omicini *et al.*, 2004] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, volume 1, pages 286–293, New York, USA, 19–23July 2004. ACM.

[Ricci *et al.*, 2005] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Programming MAS with artifacts. In Rafael P. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *3rd International Workshop "Programming Multi-Agent Systems" (PROMAS 2005)*, pages 163–178, AAMAS 2005, Utrecht, The Netherlands, 26 July 2005.

[Russel and Norvig, 2003] Stuart Russel and Peter Norvig. *Artificial Intelligence. A Modern Approach*. Prentice All, New Jersey, 2nd edition, 2003.

[Simon, 1996] Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, 3rd edition, October 1996.

[Sycara *et al.*, 2003] Katia Sycara, Massimo Paolucci, Martin van Velsen, and Joseph Giampapa. The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1-2), 2003.

[Varela, 1981] Francisco Varela. Describing the logic of the living: The adequacy and limitations of the idea of autopoiesis. In Milan Zeleny, editor, *Autopoiesis: A Theory of Living Organization*, pages 36–48, North Holland, New York, 1981.

[Viroli and Ricci, 2004] Mirko Viroli and Alessandro Ricci. Instructions-based semantics of agent mediated interaction. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 286–293, New York, USA, 19–23July 2004. ACM.

[Viroli *et al.*, 2005] Mirko Viroli, Andrea Omicini, and Alessandro Ricci. Engineering MAS environment with artifacts. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *2nd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2005)*, pages 62–77, AAMAS 2005, Utrecht, The Netherlands, 26 July 2005.

[Weyns *et al.*, 2005a] Danny Weyns, Van Dyke Parunak, Fabien Michel, Tom Holvoet, and Jacques Ferber. Environments for multiagent systems, state-of-the-art and research challenges. In *Environments for Multiagent Systems*, volume 3374 of *LNCS*. Springer Verlag, 2005.

[Weyns *et al.*, 2005b] Danny Weyns, Kurt Schelfthout, and Tom Holvoet. Exploiting a virtual environment in a real-world application. In *2nd International Workshop on Environments for Multiagent Systems (E4MAS), AAMAS'05*, 2005.