Environment programming in multi-agent systems: An artifact-based perspective

Article *in* Autonomous Agents and Multi-Agent Systems · September 2011

DOI: 10.1007/s10458-010-9140-7 · Source: DBLP

3 authors:

Alessandro Ricci
University of Bologna
274 PUBLICATIONS **4,902** CITATIONS
SEE PROFILE

Michele Piunti
Reply, Inc.
45 PUBLICATIONS **738** CITATIONS
SEE PROFILE

Mirko Viroli
University of Bologna
327 PUBLICATIONS **5,958** CITATIONS
SEE PROFILE

Auton Agent Multi-Agent Syst (2011) 23:158–192
DOI 10.1007/s10458-010-9140-7

Environment programming in multi-agent systems: an artifact-based perspective

Alessandro Ricci · Michele Piunti · Mirko Viroli

Published online: 23 June 2010

Abstract This article introduces the notion of *environment programming* in software multi-agent systems (MAS) and describes a concrete computational and programming model based on the *artifact* abstraction and implemented by the CArtAgO framework. Environment programming accounts for conceiving the computational environment where agents are situated as a *first-class abstraction* for programming MAS, namely a part of the system that can be designed and programmed—aside to agents—to encapsulate functionalities that will be exploited by agents at runtime. From a programming and software engineering perspective, this is meant to improve the modularity, extensibility and reusability of the MAS as a software system. By adopting the A&A meta-model, we consider environments populated by a dynamic set of computational entities called *artifacts*, collected in *workspaces*. From the agent viewpoint, artifacts are first-class entities of their environment, representing resources and tools that they can dynamically instantiate, share and use to support individual and collective activities. From the MAS programmer viewpoint, artifacts are a first-class abstraction to shape and program functional environments that agents will exploit at runtime, including functionalities that concern agent interaction, coordination, organisation, and the interaction with the external environment. The article includes a description of the main concepts concerning artifact-based environments and related CArtAgO technology, as well as an overview of their application in MAS programming.

Keywords Environment programming · Multi-agent systems programming · Artifacts · CArtAgO · Agent programming languages · Jason

A. Ricci (✉) · M. Piunti · M. Viroli
DEIS, Alma Mater Studiorum—Università di Bologna, Via Venezia 52, Cesena, Italy
e-mail: a.ricci@unibo.it

M. Piunti
e-mail: michele.piunti@unibo.it

M. Viroli
e-mail: mirko.viroli@unibo.it

1 Introduction

The notion of *environment* is a primary concept in agent and multi-agent systems, being the computational or physical place where agents are situated, and providing the basic ground for defining the notions of agent perception, action and then interaction. Fundamental features of the agent abstraction are directly or un-directly related to the environment: reactivity is an obvious example, but also pro-activeness, being the notion of *goal*, which is actually the essential aspect of pro-active behaviours, typically defined in terms of states of the world that an agent aims to bring about.

Actually two main different perspectives can be adopted when defining the concept of environment in MAS: a classical one rooted in AI, and a more recent one grown in the context of Agent-Oriented Software Engineering (AOSE) [25]. In the classical AI view [46], the notion of environment is used to identify the external world (with respect to the system, being a single agent or a set of agents) that is perceived and acted upon by the agents so as to fulfill their tasks. In contrast to this view, recent works in the context of AOSE introduced the idea of environment as a *first-class abstraction* for MAS engineering [54], namely a suitable place where to encapsulate functionalities and services to support agents activities (the interested reader can consult [55,56] for a survey of the research works developed in this context). In the latter view, the environment is no longer just the target of agent actions and the container and generator of agent percepts, but a part of the MAS that can be suitably designed so as to improve the overall development of the system. Accordingly, in this case the environment can be defined from the MAS and MAS engineers point of view as *endogenous*, being part of the software system to be designed—in contrast to classic AI environments which can be defined, dually, as *exogenous*.

The responsibilities and functionalities of endogenous environments can be summarised by the following three different levels of support, identified in [54]: (i) *a basic level*, where the environment is exploited to simply enable agents to access the *deployment context*, i.e. the given external hardware/software resources which the MAS interacts with (sensors and actuators, a printer, a network, a database, a Web service, etc.); (ii) *abstraction level*, exploiting an environment abstraction layer to bridge the conceptual gap between the agent abstraction and low level details of the deployment context, hiding such low level aspects to the agent programmer; (iii) *interaction-mediation level*, where the environment is exploited to both regulate the access to shared resources, and mediate the interaction between agents. These levels represent different degrees of functionality that agents can use to achieve their goals.

In this article we bring this perspective from MAS design to *MAS programming*, devising the environment as a first-class abstraction when programming multi-agent systems, to be integrated with existing agent programming languages. By adopting this view, the environment becomes a *programmable* part of the systems: accordingly we will refer to this aspect of MAS programming as *environment programming*. Analogously to agent programming, a main issue in this case concerns the definition of general-purpose computational and programming models and related technologies (languages, frameworks) to program the environment, and their integration with existing agent programming languages and frameworks. Accordingly, in this article we describe a concrete computational and programming model based on the A&A (Agents and Artifacts) meta-model [28,43] and implemented by CArtAgO technology [44]. This approach allows for designing and programming an environment in terms of a dynamic set of first-class computational entities called *artifacts*, collected in localities called *workspaces*. Artifacts represent resources and tools that agents can dynamically instantiate, share and use to support their individual and collective activities [28,43].

On the one side, they are first-class abstractions for MAS designers and programmers, who define the types of artifacts that can be instantiated in a specific workspace, defining their structure and computational behaviour. On the other side, artifacts are first-class entities of agents world, which agents perceive, use, compose, and manipulate as such.

By exploiting the artifact abstraction, CArtAgO provides a direct support to all the three levels identified before. At the basic level, artifacts can be used to wrap and enable access to resources in the deployment context. At the abstraction level, artifacts can be used to define a new abstraction layer both hiding the low-level details of the deployment context and possibly containing computational resources that are fully *virtual*, independent from the deployment context. At the interaction-mediation level, artifacts can be designed to encapsulate and enact coordination mechanisms, so as to realise forms of environment-mediated interaction and coordination [29].

The remainder of the paper is organised as follows. In Sect. 2 we outline the main aspects that concern environment programming in MAS, abstracting from specific models and technologies. Then, in Sect. 3 we informally describe the artifact-based computational/programming model and its implementation in CArtAgO technology; after that, in Sect. 4 we discuss how the approach can be applied in practice to improve MAS programming, in particular to develop mechanisms useful for agent interaction, coordination, and organisation. We conclude the paper discussing related works in Sect. 5 and providing concluding remarks including a sketch of ongoing and future works in Sect. 6.

2 Environment programming in multi-agent systems

At a first glance, the basic idea behind environment programming can be sumarised by the equation

$$\text{programming MAS} = \text{programming Agents} + \text{programming Environment}$$

where implicitly we refer to software MAS and endogenous environments. In this view, the environment is a programmable part of the system, orthogonal to—but strongly integrated with—the agent part. Orthogonality means *separation of concerns*: on the one side, agents are the basic abstraction to design and program the autonomous parts of the software system, i.e. those parts that are designed to fulfill some goal/task[1]—either individual or collective—encapsulating the logic and the control of their action. On the other side, the environment can be used to design and program the computational part of the system that is *functional* to agents' work, i.e. that agents can dynamically *access* and *use* to exploit some kind of functionality, and possibly *adapt* to better fit their actual needs. As already stated in the introduction, such functionalities can range from enabling and easing agents' access to the external (deployment) environment, to introducing computational structures properly designed to help agent work, up to mediating and ruling agents interaction for organisation and coordination purposes.

As a simple example, consider the implementation inside a multi-agent program of a blackboard as a mechanism to enable uncoupled communication among agents. Without the above-mentioned separation of concerns, a blackboard must be implemented as an agent, creating then a mismatch between the design and implementation, since a blackboard is typically not designed to fulfill pro-actively and autonomously some goal, but rather to be used by other agents to communicate and coordinate. By adopting environment programming, the

[1] Here the concept of task and goal are used as synonyms.

blackboard is implemented as an environment resource, accessed by agents in terms of actions and percepts. The example can be generalised, considering any possible computational entity properly designed to help agent work and interaction.

2.1 Programming models for environment programming

To program environments we need to adopt some general-purpose computational and programming model, defining respectively what model can be adopted to define the structure and computational behaviour of the environment, and what kind of programming abstractions and constructs can be used by MAS developers to design and program environments. By following well-known principles in computer programming and software engineering, some desiderata on programming/computational models can be identified:

Abstraction: The model adopted should preserve the agent abstraction level, i.e. the main concepts used to program environment structure and dynamics should be consistent with agent concepts and their semantics. Examples include the notion of actions, percepts, events, tasks/goals.

Orthogonality: The model should be as much orthogonal as possible to the models, architectures, languages adopted for agent programming, so as to naturally support the engineering of heterogeneous systems.

Generality: The model should be general and expressive enough to allow for developing different kinds of environment according to different application domains and problems, exploiting the same basic set of concepts and constructs.

Modularity: The model should introduce concepts to modularise environments, avoiding monolithic and centralised views.

Dynamic extensibility: The model should support the dynamic construction, replacement, extension of environment parts, in an *open system* perspective.

Reusability: The model should promote the reuse of environment parts in different application contexts/domains.

Among the others, the point about *abstraction* is important to remark that existing programming paradigms can be re-used to define the environment programming model only by bridging the abstraction gap that exists with respect to the agent abstraction level. For instance, the notion of object as defined in the context of object-oriented programming (OOP) cannot be re-used "as it is" as first-class environment abstraction: on the one side, in OOP objects interact with each other by means of method invocation and no action/perceptions concepts are defined; on the other side, method invocation is not defined in the context of agent-oriented programming and in the semantics of agent programming languages, consequently it is not meaningful to simply enable "agent–object interaction" in terms of method invocation. This holds also when considering agent frameworks based on OO programming languages, such as Jade [1] (which is based on Java): objects (classes) are used to implement agents, not to create environments shared by agents to enhance their coordination and cooperation—agents are meant to interact solely by means of message passing based on FIPA ACL. In other words: objects are not first-class entities of the agent world, they are the basic construct to implement agents. So, also in this case a further abstraction layer is necessary to wrap objects, defining the semantics of agent–object interaction.

2.2 Main aspects

Given these general requirements, in the following we identify and discuss main aspects that—we argue—characterise a general-purpose programming/computational model for environment programming, namely (i) the action model, (ii) the perception model, (iii) the environment computational model, (iv) the environment data model, and finally (v) the environment distribution model.

2.2.1 Action model

This aspect concerns how agents affect the state of the environment—hence, the very notion of external action—and includes what kind of semantics is adopted for defining action success/failure, which action execution model is adopted, and finally how the action repertoire is defined/programmed.

The most common action success/failure semantics adopted in current agent programming languages is that the success of an action, on the agent side, means that the action has been successfully executed by agent effectors. That is, the action has been accepted by the environment. However this does not imply anything about action completion and effects: in order to know if the execution of an action in the environment has been completed with success, an agent must check its percepts. The environment programming perspective makes it possible to define and exploit richer semantics, in which, for instance, the success of an action on the agent side means that not only the action has been accepted by the environment, but also that its execution has been completed and related effects on the environment produced. More generally, in endogenous environments the set of actions can be considered part of the *contract* that the environment provides to the agents that are logically situated in it.

Concerning the action execution model, the semantics typically adopted in current agent programming languages models actions as *events*, i.e. as a single atomic transition (from the agent viewpoint) changing/inspecting the state of the environment. In this case it is like to say that actions have zero time length and the execution of two actions cannot overlap in time. Also in this case, programmable environments make it possible to introduce richer semantics, modelling action execution as a *process*, i.e., a sequence of two or more events, including the event representing the starting of the action execution and the event representing the completion of the execution. This allows for easily representing long-term, possibly concurrent actions and also to define actions useful for agent synchronisation—this aspect will be clarified in Sect. 4.

2.2.2 Perception model

This aspect concerns how the environment can be perceived by agents, the definition of the stimuli generated by the environment and the corresponding agent percepts as result of the perception process. Along with actions, these can be considered part of the contract provided by the environment as well.

Essentially, two basic semantics can be adopted when defining the perception model, that we refer here as *state-based* and *event-based*. In the former, stimuli are information about the *actual state* of the environment and are generated when the agent is engaging the perception stage of its execution cycle.[2] In the latter, stimuli are information about *changes*

[2] Here we refer to the agent control loop which is typically found—with different characterisations—in agent architectures, such as BDI (Belief-Desire-Intention), composed by a sense stage—in which inputs from the

occurred in the environment, dispatched to agents when such changes occur, independently from agents' execution state. For instance, the model adopted to define agent abstract architecture [57], where a *see* function is introduced to model the perception process mapping the environment actual state E into a set of percepts P, is state-based. In this case, percepts are a snapshot of the current state of the observable part of the environment: in BDI architectures, for instance, such a snapshot is used to update the current state of the belief base. Referring to concrete agent programming languages and their formal operational semantics, this approach is adopted—for instance—in Jason [5], where the update of the belief base generates events which may trigger plans. It is worth noting that Jason architecture allows for fully customise also this aspect, so this is just the *default* semantics and other semantics can be injected. A concrete example of agent programming language treating percepts directly as events is 2APL [10]: in that case in order to keep track of the observable state of the environment in terms of beliefs, the programmer is forced to explicitly define the rules that specify how to change the belief-base when a percept is detected.

As in the case of the action model the chosen semantics for percepts and perception can have a strong impact on the dynamics of MAS program execution. For instance, by adopting a state-based approach, if the environment changes multiple times between two subsequent occurrences of the perception stage of an agent execution cycle, such changes are not perceived by the agent.

2.2.3 Environment computational model

This aspect concerns how to program environment functionalities, that is the structures defining its inner state and its computational behaviour, both including those computations that are directly caused by agent actions and those that represent internal processes.

A first main aspect concerns the model adopted to decompose the overall computational state/behaviour of the environment. The simplest one is the monolithic approach, in which the computational structure and behaviour of the environment is represented by a single computational object, with a single state. In this case, this object is the entry point for defining the effect of actions and the set of stimuli generated. For instance, 2APL, GOAL [18], and Jason natively adopt this approach, by providing a Java based API to program the environment as a class. A more modular approach accounts for explicitly defining first-class structures (and finally abstractions) to decompose and modularise the functionalities of the environment [50]: a main example in this case is provided by artifact-based environments, that will be described in next section. Other examples—that will be surveyed in the related works section—include a generic notion of *object* as adopted in GOLEM [7] and MadKit [17]. Depending on the structure adopted, the action model may include or not the actions to create/dispose/replace environment structures at runtime.

A related aspect concerns the *concurrency model* adopted, that is how many threads or control flows are exploited to execute environment computational processes, and, in the case of multiple threads, what is the mapping with respect to the environment computations and how concurrency problems, such as race conditions and interferences, are avoided. Clearly, this aspect strongly impacts on the performance of the system.

Footnote 2 continued
environments are collected, a plan stage, in which the internal state of the agent is updated and a new action to perform selected, and an act stage, in which the action is executed.

2.2.4 Environment data model

This aspect concerns the types of the data exchanged by agents and the environment, which is used in particular to encode action parameters, action feedbacks, the content of stimuli (percepts), and their representation. This completes the basic contract that the environment exposes to agents. Here we need to face the same *interoperability* issues that arise when integrating in the same program parts that have been developed with different programming languages/frameworks, in this case agent programming languages on the one side and environment programming languages/frameworks on the other side.

To tackle these issues, first, an environment *data model* can be introduced, defining explicitly the possible types of data structures involved in actions and percepts, adopting some data representation language (such as XML-based ones) or even the object model of an object-oriented programming language. Then, on the agent programming language side, a form of *data-binding* must be specified, defining how the types of data defined in the data model can be translated into the specific data model adopted by the agent language and vice-versa.

A further issue that must be faced in the case of open systems is the definition of environment data model that allows for describing proper *ontologies* so as to explicitly define the semantics of the data involved in agent–environment interaction. To this purpose existing work in the context of Semantic Web and the models/languages adopted for describing ontology (such as OWL) can be suitably exploited.

2.2.5 Environment distribution model

Finally, this aspect concerns how to handle distribution, i.e. how to program environments that need to be distributed (or can be opportunistically distributed) among multiple network nodes. To this end, a distributed environment model may introduce an explicit notion of *place* (locality) to define a non-distributed portion of the computational environment and then define if/how places—and related environment computational structures—are connected and eventually interact. On the agent side, the environment distribution model adopted affects the repertoire of agent action, possibly including also actions to enter into a place or move from place to place.

Actually, the environment distribution model affects also the *time model* which can be adopted inside the MAS. For distributed MAS it is not feasible—both from a theoretical and practical point of view—to have a *single* notion of time inside the system, to time-stamp events and then define total orders among such events. This is a main issue, since many formalisations of agent systems in different contexts—such as e-Institutions, normative systems, agent organisations—typically are based on a global notion of time. By subdividing an environment into proper sub-environments—often called *places*—it is possible to recover the notion of time at the level of the single place.

After devising the main aspects that should be addressed by general-purpose computational/programming models for environment programming, in next sections we describe how these aspects and issues are handled in artifact-based environments and CArtAgO technologies.

3 Artifact-based environments and CArtAgO

By drawing inspiration from Activity Theory [23], the notion of *artifact* in MAS has been introduced the first time in [38] in the context of MAS coordination, in particular to define

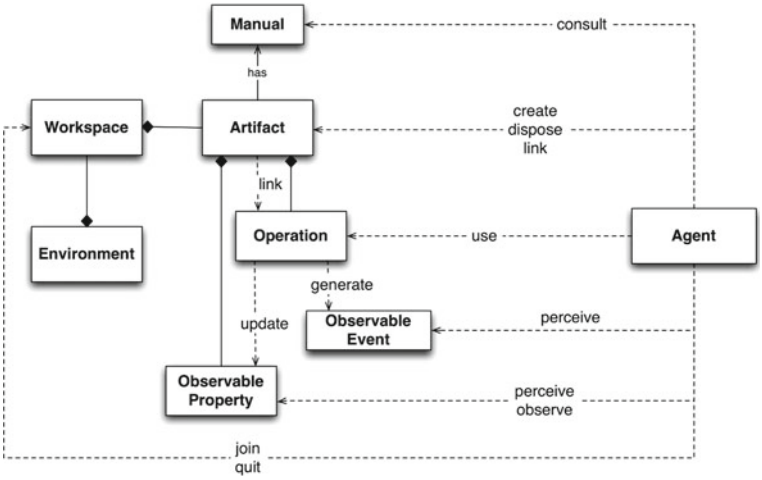


Fig. 1 A&A meta-model expressed in the Unified Modelling Language (UML)-like notation

the basic properties of first-class coordination abstractions enabling and managing agent interaction, generalising the notion of coordination media [29]. The concept has been then generalised besides the coordination domain, leading to the definition of the A&A (Agents and Artifacts) meta-model [28,43] and the development of a computational framework—CArtAgO [41,44]—to support the development and execution of environments designed and programmed upon the notion of artifact.

In the following we provide a concise and informal description of the main concepts underlying A&A and CArtAgO technology—revisited and extended with respect to previous publications so as to include the most recent improvements [39,41,43,44]—in the environment programming perspective in particular. First we provide an abstract overview of the basic concepts underlying the artifact abstraction and the basic actions on the agent side to work with artifacts, and then we briefly describe how they are concretely provided by CArtAgO technology.

3.1 Basic concepts

Figure 1 provides an overview of the main concepts characterising artifact-based environments. The environment is conceived as a dynamic set of computational entities called *artifacts*, representing in general resources and tools that agents working in the same environment can share and exploit. The overall set of artifacts can be organised in one or multiple *workspaces*, possibly distributed in different network nodes. A workspace represents a place (following the terminology introduced in Sect. 2), the locus of one or multiple activities involving a set of agents and artifacts.

From the MAS designer and programmer viewpoint, the notion of artifact is a *first-class abstraction*, the basic *module* to structure and organise the environment, providing a general-purpose programming and computational model to shape the functionalities available to agents. Actually, MAS programmers define *types* of artifacts, analogously to classes in OOP, which define the structure and behaviour of the concrete instances of those types. Each workspace is meant to have a (dynamic) set of artifact types that can be used to create instances. From the agent viewpoint, artifacts are the *first-class entities* structuring, from a

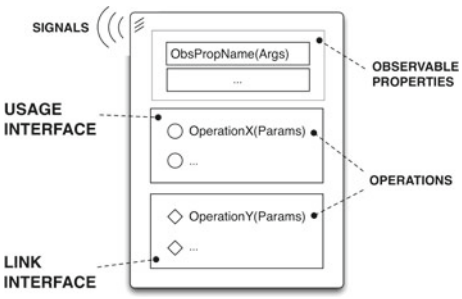


Fig. 2 The abstract representation of an artifact used in the paper. In evidence the usage interface, the observable properties and the link interface

functional point of view, the computational world where they are situated and that they can create, share, use, perceive at runtime.

To make its functionalities available and exploitable by agents, an artifact provides a set of *operations* and a set of *observable properties* (see Fig. 2). Operations represent computational processes—possibly long-term—executed inside artifacts, that can be triggered by agents or other artifacts. The term *usage interface* is used to indicate the overall set of artifact operations available to agents. Observable properties represent state variables whose value can be perceived by agents;[3] the value of an observable property can change dynamically, as result of operation execution. The execution of an operation can generate also *signals*, to be perceived by agents as well: differently from observable properties, signals are useful to represent non-persistent observable *events* occurred inside the artifact, carrying some kind of information. Besides the observable state, artifacts can have also an hidden state, which can be necessary to implement artifact functionalities.

From an agent viewpoint, artifact operations *represent external actions provided to agents by the environment*: this is a main aspect of the model. So in artifact-based environments the repertoire of external actions available to an agent—besides those related to direct communication—is defined by the set of artifacts that populate the environment. This implies that the actions repertoire can be dynamic, since the set of artifacts can be changed dynamically by agents themselves, instantiating new artifacts or disposing existing artifacts. Observable properties and events constitute instead agent percepts. In BDI architectures, like the ones found in Jason, 2APL or GOAL, percepts related to the value of observable properties can be directly modelled inside agents as beliefs about the actual state of the environment. Actually, to scale up with the environment complexity, in artifact-based environments an agent can dynamically select which are the artifacts to observe, so as to perceive the observable properties and events of only that part of the environment that the agent is interested in.

As a principle of composition, artifacts can be *linked* together so as to enable one artifact to trigger the execution of operations over another linked artifact. To this purpose, an artifact can expose a *link interface* which, analogously to the usage interface for agents, includes the set of operations that can be executed by other artifacts—once the artifacts have been linked together by agents, as clarified in next sections. The semantics of link operation execution is the same of operations executed by agents: the operation request executed by the linking artifact is suspended until the operation on the linked artifact has been executed, with success or failure. Link operations cannot be accessed by agents, but only by linking artifacts. Linkability makes it possible to realise distributed environments, linking together artifacts possibly belonging to different workspaces in different network nodes. It is worth noting

[3] Actually by those agents that are observing the artifact, as will be clarified later on in the section.

that this mechanism allows for treating artifacts as components in the context of component-oriented software engineering, providing and requiring interfaces that can be connected together.

Finally, an artifact can be equipped with a *manual*, a machine-readable document to be consulted by agents, containing a description of the functionalities provided by the artifact and how to exploit such functionalities (that is, artifact *operating instructions* [51]). Such a feature has been conceived in particular for open systems composed by intelligent agents that dynamically decide which artifacts to use according to their goals and dynamically discover how to use them. Actually, the notion of manual can be extended from artifacts to workspaces [40]: in that case manuals may contain the description of usage protocols that can involve multiple kinds and instances of artifacts.

3.2 Actions to work with artifacts

First, to work within a workspace an agent must *join* it (and eventually *quit* from it as soon as it completed its work); an agent can work simultaneously in multiple workspaces, possibly distributed among different network nodes. Then, within a workspace the set of actions available to agents to work with artifacts can be categorised in three main groups: (i) actions to create/lookup/dispose artifacts; (ii) actions to use artifacts, execute operations and observe properties and signals; (iii) actions to link/unlink artifacts. In the following, we describe these basic sets of actions more in detail. The syntax Name(Params):Feedback is used to define action signature, which includes the action name, parameters and optionally the action feedback. The action feedback represents some kind of data—which depends on the specific action—which can result from action execution, carrying information related to the success or failure of the action.

3.2.1 Creating and discovering artifacts

Artifacts are meant to be created, discovered and possibly disposed by agents at runtime: this is a basic way in which the model supports dynamic extensibility (besides modularity) of the environment. Three basic kinds of action are provided to this purpose: makeArtifact, disposeArtifact and lookupArtifact. The action makeArtifact(ArName,ArTypeName,InitParams):ArId instantiates a new artifact called ArName of type ArTypeName inside a workspace. The logic name identifies the artifact inside a workspace: artifacts belonging to different workspaces can have the same logic name, so besides the logic name each artifact has also a unique identifier generated by the system—returned as action feedback. Dually to makeArtifact, disposeArtifact(ArId) allows for removing an artifact from a workspace. It is worth noting that proper access control mechanisms can be adopted here to avoid that—for instance—a generic agent could be allowed to dispose any possible artifact inside a workspace. In CArtAgO model and technology for instance—described in Sect. 3.3—a Role-Based Access Control (RBAC) approach [47] is adopted, which allows to group agents in roles and, for each role, to define policies specifying what actions on which artifacts agents interpreting such roles are allowed/not allowed to perform.

Artifact discovery concerns the possibility of retrieving the identifier of an artifact located in a workspace given either its logic name or its type description. A couple of actions are provided to this end: lookupArtifact(ArName):ArId which retrieves an artifact unique identifier given its logic name, and lookupArtifactByType(ArTypeName):{ArId} which retrieves the (possibly empty) set of artifacts that are instances of the specified type.

3.2.2 *Using and observing artifacts*

Using an artifact for agents involves two aspects: (1) being able to execute operations actually listed in the artifact usage interface and (2) being able to perceive artifact observable information, in terms of observable property and events.

For the first aspect, a single action use(ArId,OpName(Params)):OpRes is provided (see Fig. 3), specifying the identifier of the target artifact and the details about the operation to be executed (name and required parameters). The action succeeds if the operation completes with success; conversely, the action fails if either the specified operation is not currently included in the artifact usage interface or if some error occurred during operation execution, i.e. the operation itself failed. By successfully completing its execution, an operation may generate some results that are returned to the agent as action feedback. By performing a use, current agent activity/plan is suspended until an event reporting the completion of the action (with success or failure) is received. By receiving the action completion event, the action execution is completed—and the related activity/plan reactivated. However, even if one activity is suspended, the agent is not blocked: the agent cycle can go on processing percepts and executing actions related to other plans/activities.

This semantics finally results in considering artifact operations directly *as agent actions*, or rather the set of operations provided by artifacts as an extension of agents' action repertoire. Accordingly, since the operations executed by artifacts can be (long-term) processes, use actions triggering the execution of the operations can be long-term as well, not completing (with success or failure) immediately. So, following the classification described in Sect. 2, the action-model adopted is process-oriented. We will see this aspect in practice in Sect. 3.3, and in Sect. 4 we will show how such a semantics could be exploited to create effective mechanisms for agents' action synchronisation.

For the second aspect, i.e. observation, an agent can start perceiving observable properties and signals of an artifact by doing a focus(ArId,{Filter}) action (see Fig. 4), specifying the identifier of the artifact to observe and optionally a filter to further select the subset of events the agent is interested in. In agent programming languages based on the BDI model, observable properties are mapped directly into beliefs in the belief base. An agent can focus (observe) multiple artifacts at the same time. Dually to focus, stopFocus(ArId) action is provided to stop observing an artifact. Referring to the classification introduced in Sect. 2, the perception model adopted is event-based: every time an observable property is changed or a signal generated, a related observable event is notified to all the agents observing the artifact. In BDI-based languages, the first kind of events—observable property update—makes

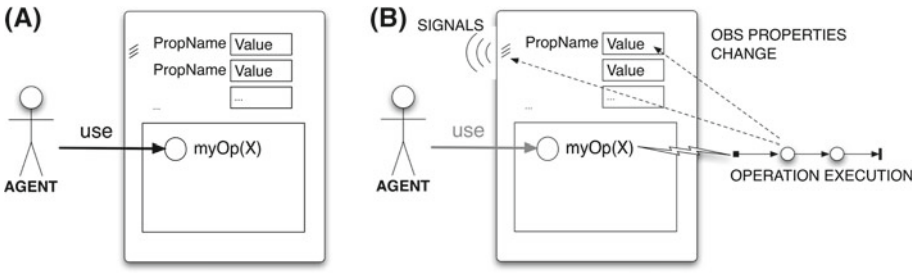


Fig. 3 *Left*: An agent invoking an operation listed in the usage interface of an artifact. *Right*: By executing the operation the observable property of the artifact can be changed and signals can be generated as observable events (for those agents observing the artifact)

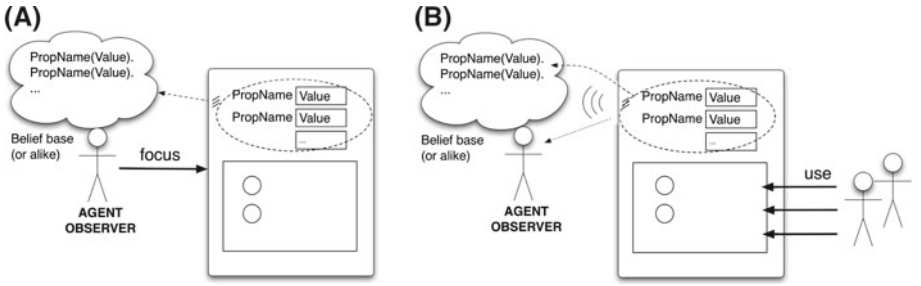


Fig. 4 By doing a focus action on an artifact, an agent will eventually perceive the updated value of observable properties as percepts—mapped into agent beliefs or knowledge about the environment—and all the signals generated by the artifacts

it possible on the agent architecture side to automatically update the beliefs keeping track of the observable properties. Signals, instead, are not related to observable properties: they are like messages generated on the artifact side that are asynchronously processed on the observing agents side. Concrete examples of these aspects will be provided using CArtAgO in next section.

It's worth noting that an agent can use an artifact without observing it, if it is not interested in the properties of the artifact or the events that it generates. Conversely, if an agent executes an operation on an artifact that the agent is observing, then it will eventually have percepts about the artifact while the operation is in execution (and while the use action has not completed yet) and it can react accordingly.

We conclude the basic set of actions to use artifacts with the observeProperty (PropName):PropValue action, which reads the actual value of a specific observable property. In this case no percepts are involved: the value of the property is retrieved as action feedback. This is useful when an agent does not need to be continuously aware of the observable state by an artifact, but to know the value of such properties when needed.

3.2.3 Linking and unlinking artifacts

Linking artifacts accounts for connecting two artifacts together so as to allow one artifact (the linking one) to execute operations over another artifact (the linked one). More precisely, by linking two artifacts the execution of an operation on the linking artifact may trigger the execution of operations on the linked artifact(s). To this end two basic actions are provided, linkArtifacts(LinkingArId,LinkedArId{,Port}) and unlinkArtifacts(LinkingArId,LinkedArId), respectively to link and unlink two artifacts together. This makes it possible for agents to dynamically compose complex artifacts by linking together simple ones, creating networks of artifacts—which can be distributed in different workspaces—and changing the links according to the need. The Port parameter is needed when linking the same artifact to multiple artifacts: in that case the linking artifact must provide multiple labelled *ports* and a linked artifact can be attached to a specific port.

A note is worth about the interface compatibility when linking two artifacts. Currently we adopt the simplest solution, allowing any artifact to be linked to any other artifact; this, however, could lead to runtime operation failures in the case that in the body of an artifact operation a linked operation is executed and either no artifacts are actually linked or the requested operation is not provided by the actual linked artifact. For this latter case, a better solution accounts for checking the compatibility of linking when linkArtifacts action is

executed—and eventually the introduction of a formal notion of link interface. This is part of future work.

3.3 CArtAgO technology

CArtAgO (Common ARtifact infrastructure for AGent Open environments) is a framework and infrastructure for programming and executing artifact-based environments implementing the model described informally above. As a *framework*, it provides a Java-based API to program artifacts and the runtime environment to execute artifact-based environments, along with a library with a set of pre-defined general-purpose artifact types. As an *infrastructure*, it provides the API and the underlying mechanism to extend agent programming languages/frameworks so as to program agents to work inside CArtAgO environments.

To exemplify artifact programming, in the following we provide an informal description of some main aspects concerning the API: further details can be found in CArtAgO documentation available in CArtAgO open-source distribution.[4]

3.3.1 Artifact programming model

The Java-based API makes it possible to program artifacts in term of Java classes and basic data types, without the need of using a new special-purpose language to this end. In the following we give an overview of the basic features of the programming model using some simple examples.

An artifact (type) is programmed directly by defining a Java class extending the library class `cartago.Artifact`, and using a basic set of Java annotations and inherited methods to define the elements of artifact structure and behaviour.[5] The type defines the structure and behaviour of the concrete instances that will be instantiated and used by agents at runtime. Figure 5 shows a simple example of artifact type definition named `Counter`, implementing a simple counter, providing a single observable property called `count` keeping track of the counter value, and an operation `inc` to increment the value of the count. The `init` method is used by convention to specify how the artifact must be initialised at creation time—if the method generates an error, the artifact is not created and the agent action fails. Observable properties are defined by means of the `defineObsProperty` primitive specifying the name of the property and the initial value (which can be any tuple of data objects).[6] Other two primitives are available to retrieve the current value of the property (`getObsProperty`) and to change its current value (`updateObsProperty`). Instance fields of the class are used to implement artifact internal non-observable state (no internal variables are defined in the `Counter` example).

Operations are defined by methods annotated with the `@OPERATION` tag and `void` return value, using method parameters as both input and output operation parameters. In the example the counter artifact has an `inc` operation, which simply increments the value of the observable property `count`. Output operation parameters can be used to spec-

[4] http://cartago.sourceforge.net.

[5] The annotation framework is a feature introduced with Java 5.0 that makes it possible to tag some elements of a class description on the source code—including methods, fields, and the class definition itself—with some descriptors that can be accessed at runtime by the program itself, through the reflection API. Annotations are represented by symbols of the kind `@ANNOT_NAME`, possibly containing also attributes `@ANNOT_NAME(attrib=value,...)`.

[6] Actually the values of an observable property can be objects of any type—in other words it is not possible to specify or constrain the type of the values that an observable property can feature.

```
class Counter extends Artifact {

  void init(){
    defineObsProp("count",0);
  }

  @OPERATION void inc(){
    int c = getObsProperty("count").intValue();
    int newc = c + 1;
    updateObsProperty("count", newc);
  }
}
```

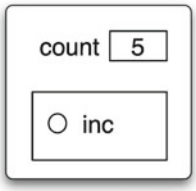


Fig. 5 *Left*: Definition of a simple `Counter` artifact, exposing a usage interface with a single operation `inc` and an observable property `count`. *Right*: An abstract representation of the `Counter` artifact, with in evidence the usage interface and the observable property

```
public class MyArtifact extends Artifact {

  int internalCount;

  @OPERATION
  void opWithResults(double x, double y,
                     OpFeedbackParam<Double> sum,
                     OpFeedbackParam<Double> sub){
    sum.set(x+y);
    sub.set(x-y);
  }

  @OPERATION void structuredOp(int ntimes){
    internalCount = 0;
    signal("step1_completed");
    await("canExecStep2", ntimes);
    signal("step2_completed",internalCount);
  }

  @GUARD boolean canExecStep2(int ntimes){
    return internalCount >= ntimes;
  }

  @OPERATION void update(int delta){
    internalCount+=delta;
  }
}
```

```
class Clock extends Artifact {

  boolean working;
  final static int TICK_TIME = 1000;

  void init(){
    working = false;
  }

  @OPERATION void start(){
    if (!working){
      working = true;
      execInternalOp("work");
    } else {
      failed("already_working");
    }
  }

  @OPERATION void stop(){
    working = false;
  }

  @INTERNAL_OPERATION void work(){
    while (working){
      signal("tick");
      await_time(TICK_TIME);
    }
  }
}
```

Fig. 6 *Left*: Definition of an artifact providing operations with feedbacks (`opWithResults`) and an operation structured in steps (`structuredOp`). *Right*: An artifact with an internal operation (`work`), indirectly controlled by the other two operations (`start` and `stop`)

ify the operation results and related action feedback: an example is given by the operation `opWithResults` provided by `MyArtifact` artifact shown in Fig. 6. The parameters `x` and `y` are input parameters, while `sum` and `sub` are output parameters (represented by the parametrised class `OpFeedbackParam`), whose value is set by the operation execution.

Operations can be composed by one or multiple *atomic* computational steps. In the examples, `inc` in `Counter` and `opWithResults` in `MyArtifact` are composed by a single atomic computational block. The execution of atomic steps inside an artifact is mutually exclusive: only one atomic computational step can be in execution at a time inside an artifact. So no interferences can occur if multiple single-step operations are requested concurrently by agents.

Operations composed by multiple steps are essential for implementing long-term operations and, as will be shown in Sect. 4, efficient coordination mechanisms. An example is given by the `structuredOp` operation provided by `MyArtifact`, whose execution is composed by two steps. The `await` primitive is the basic mechanism that can be used to

break down the execution in steps, completing a step and then waiting for some condition or event before executing the next step. In the example the first step simply resets an internal counter and generates an observable event `step1_completed` by means of the `signal` primitive, which is part of the CArtAgO API. Such an observable event (or signal) will be eventually perceived by all the agents focussing the artifact. The `signal` primitive allows for specifying the type of the signal—`step1_completed` in the example—and optionally an information content (which can be a tuple of values). The second step is executed as soon as the internal counter has reached the value specified as argument of the operation. The API allows for specifying this by implementing proper boolean methods—annotated with `@GUARD`—which define the condition to be specified in the `await` primitive. In multi-step operations, each step is executed atomically: only a single operation step can be in execution inside an artifact at a time. However, multi-step operations can be executed concurrently, by interleaving the steps—single-step operations, instead, are always executed in a mutually exclusive way.

Operations can themselves trigger the asynchronous execution of *internal* operations, that are operations which are not part of the usage interface. The definition of internal operations is the same of normal operations except for the annotation to be used, which is `@INTERNAL_OPERATION`. To trigger the execution of internal operations the `execInternalOp` primitive is provided, specifying the name of the operation and the parameters. The `Clock` artifact shown in Fig. 6 (on the right) has an internal multi-step operation `work` which is triggered by the `start` operation. The operation `work` is multi-step and repeatedly emits a signal `tick`—in this case the `await_time` primitive is used specifying, as event that triggers the execution of the next step, the amount of time (in milliseconds) that must elapse. The process goes on until the `stop` operation is executed, resetting the `working` flag. Note that `execInternalOp` does not execute directly the specified operation (as a method call): it just triggers its execution, which actually starts as soon as the overall condition of the artifact allows that.

The `Clock` example shows also the possibility to specify that an operation (and then the related action, from the agent viewpoint) has failed, by means of the `failed` primitive: in the `start` operation first we check if the clock is already working: if so, then the operation fails specifying also information about the failure. The `failed` primitive generates an exception that interrupts the control flow in the execution of the method.

Linkability is supported by properly annotating with `@LINK` annotation those operations of an artifact that can be linked by other artifacts—these operations constitute the *link interface* of the artifact. Then, an artifact can execute a link operation listed in the link interface of another artifact by means of a specific primitive, `execLinkedOp`. The semantics is analogous to normal operations and `execLinkedOp` succeeds (or fails) as soon as the

```
class LinkedArtifact extends Artifact {
  ...
  @LINK void linkedOp(String s,
                      OpFeedbackParam<Integer> r){
    r.set(s.length());
  }

}
```

```
class LinkingArtifact extends Artifact {
  ...
  @OPERATION void myOp(String s){
    ...
    OpFeedbackParam<Integer> res =
                    new OpFeedbackParam<Integer>();
    execLinkedOp("linkedOp",s,res);
    int v = res.get();
    ...
  }
}
```

Fig. 7 *Left*: An artifact exposing an operation (`linkedOp`) which can be triggered by other linked artifacts; *Right*: An example of an artifact with an operation triggering the link operation by means of the `execLinkedOp`

linked operation completes with success or failure. In the example shown in Fig. 7, the artifact `LinkedArtifact` provides a `linkedOp` link operation, which can be triggered as shown in the `myOp` operation of `LinkingArtifact` artifact. It's worth remarking that an artifact can trigger the execution of a link operations over another artifact if such artifacts have been previously linked by an agent, by means of the linkArtifacts action. A second note is that the same method can be marked with both `@LINK` and `@OPERATION` to represent an operation which is part of both the usage interface and the link interface.

As a final note, the environment data model adopted in CArtAgO is based on the Java platform, hence on the Java object model: so the data types used in operation parameters, observable properties and signals are either Java primitive data types or objects instances of some class. When integrated with agent programming languages, the primitive data types are mapped onto the APL's ones, while a basic set of internal actions is introduced to manage Java objects and classes.

3.3.2 Integration with agent programming languages

CArtAgO is orthogonal to the specific technology adopted for programming the agents working within artifact-based environments. In principle, it has been conceived so as to be integrated with any agent programming language and platform, so as to allow for creating heterogeneous systems in which agents implemented using different agent programming languages and technologies—and running on different platforms—could work together in the same MAS, sharing common artifact-based environments [39].

Technically, by integrating an agent programming language/framework with CArtAgO, the repertoire of agent actions is extended with the new set of actions discussed in previous section. On the one side it includes actions representing CArtAgO basic primitives—such as makeArtifact, lookupArtifact, focus, etc. On the other side, by means of the use mechanism, every operation provided by an artifact actually located in the environment can be considered by an agent—and an agent programmer—as an external action part of its repertoire. On the perception side, the set of possible agent percepts are extended with observable properties and signals generated by artifacts. In particular, observable properties are mapped into agent beliefs (or knowledge, if the notion of belief is not supported) about the state of the environment (artifacts), instead signals are mapped to beliefs about the occurrence of observable events. The concrete realisation of actions and percepts can vary, depending on the specific agent programming platform [39]; however their semantics—which constitutes the interface between the agent layer and environment layer—is the same. In this section and in the following one we will take Jason as reference agent programming language to describe examples, but the mapping that we used to integrate CArtAgO with Jason can be considered valid, more generally, for any intelligent agent architecture and language.

To give a taste of agent and environment programming here we consider a toy example with two Jason agents—whose source code is shown in Fig. 8—that create and cooperatively use three artifacts of type `Counter`, `Clock` and `MyArtifact` defined in Sect. 3.3.1, executing artifact operations and perceiving observable properties and events. To ease the understanding of the agent source code here we report a brief description of Jason syntax. An agent program in Jason is defined by an initial set of beliefs, representing agent's initial knowledge about the world, a set of goals, and a set of plans that the agent can dynamically instantiate and execute to achieve such goals. Agent plans are described by rules of the type `Event : Context ← Body` (the syntax is Prolog like), where Event represents the specific event triggering the plan—examples are the addition of a new belief (`+b`), a goal (`+!g`), the failure of a plan (`-!g`). The plan context is a logic formula on the belief

```
// userA agent

!test_tools.

+!test_tools
  <- !first_test;
     !second_test.

+!first_test
  <- make_artifact("myclock","Clock");
     start;
     make_artifact("mycount","Counter");
     inc;
     inc [artifact_name("mycount")];
     observe_property("mycount",count(V));
     println("[userA] count value: ",V).

+!second_test
  <- make_artifact("myartifact","MyArtifact");
     opWithResults(4,3,X,Y);
     println("[userA] results: ",X," ",Y);
     focus("myartifact");
     structuredOp(10);
     println("[userA] done").

+step1_completed
  <- println("[userA] first step completed.").
```

```
// userB agent

!test_tools.

+!test_tools
  <- !discover("myclock",Clock);
     focus(Clock);
     !discover("mycount",Count);
     focus(Count);
     !discover("myartifact",_);
     !use_myart(4).

+count(V)
  <- println("[userB] count value: ",V).

+tick
  <- println("[userB] new tick event perceived.").

+!use_myart(NTimes) : NTimes > 0
  <- update(6);
     println("[userB] used myartifact");
     !use_myart(NTimes - 1).

+!use_myart(0)
  <- println("[userB] no more using myartifact.");
     ?count(V);
     println("[userB] final count value: ",V);
     stop_focus("mycount");
     stop_focus("myclock").

+!discover(ArtName,Id)
  <- lookup_artifact(ArtName,Id).
-!discover(ArtName,Id)
  <- !discover(ArtName,Id).
```

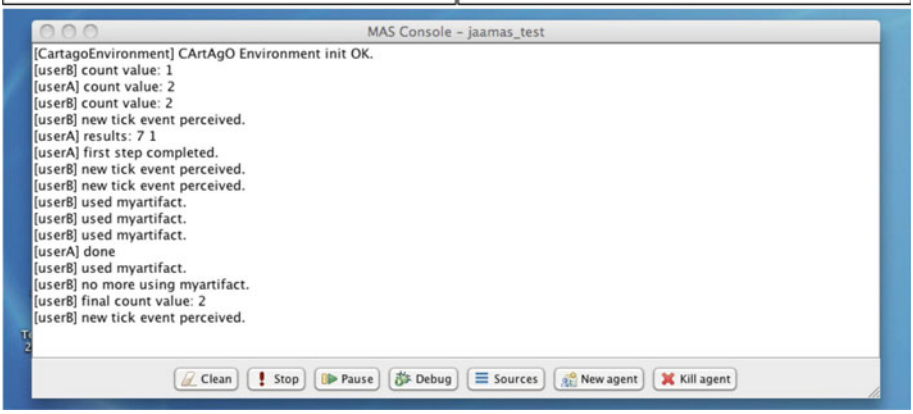


Fig. 8 A toy example with two Jason agents cooperatively using two shared artifacts

base—a belief formula—asserting the conditions under which the plan can be executed. The plan body includes basic actions to create subgoals to be achieved (`!g`), to update agent inner state—such as adding a new belief `+b`—and external actions to act upon the environment.

When integrating Jason with CArtAgO, the basic CArtAgO actions as well as the operations provided by artifacts are implemented as external actions. The `userA` agent (source code shown in Fig. 8, on the left) has a single initial goal `test_tools`. A plan to achieve the goal is specified, which is triggered by the `+!test_tools` goal addition event. The plan creates two further subgoals to achieve sequentially, `first_test` and `second_test`. The plan for `first_test` first creates an instance of a `Clock` called `myclock` and it performs a `start` external action which results in executing the `start` operation provided by the artifact. By executing `start`, `myclock` starts a counting process, generating `tick` signals. Then `userA` creates a `Counter` called `mycount` and increments it by executing

twice the `inc` operation. The second `inc` shows the possibility to explicitly specify, by means of annotations (`artifact_name` in this case), the artifact that provides the operation—in the case that the same operation is provided by multiple artifacts currently located in the workspace. Both the artifacts are observed by the second agent, `userB` (source code shown in Fig. 8, on the right), which waits for artifacts to be available by exploiting the CArtAgO action `lookup_artifact`—that fails if the specified artifact is not found—and then starts observing them by performing a `focus` action. In Jason, by focussing an artifact, observable properties are mapped into beliefs, automatically updated as soon as the value of observable properties changes. Signals are not automatically stored as beliefs: they are directly mapped onto events possibly triggering the execution of plans. In this case, by focussing `mycount`, `userB` has a new belief about the current value of the observable property `count`; by focussing `myclock` the agent perceives `tick` events eventually generated by the artifact. To test this, two plans are included in `userB`, one reacting to changes to the value of `count` and accordingly printing the new value on the console, and one triggered each time a `tick` event is perceived and printing a log. In Fig. 8 in the bottom a screenshot of the Jason console is provided, showing a run of the system. It is possible to recognise the messages logged on the console by the `userB` agents reacting to percepts.

The `first_test` plan of `userA` concludes by reading the actual value of the observable property `count` by means of the CArtAgO built-in `observe_property` action and printing it on standard output—`println` action is an operation provided by a console artifact, available by default in the workspace. The second plan (to achieve `second_test` goal) shows first the execution of an operation with output parameters and then the execution of a long-term operation. In the former case, an instance of `MyArtifact`—described in Sect. 3.3.1—is created and the `opWithResult` operation executed. By successfully completing this action, operation results—represented by the third and fourth parameter—are received as action feedback, referenced by `X` and `Y` variables.

Then, by performing an action that results in the execution of an operation inside an artifact, the plan is suspended until the action completes (with success or failure). In the example, by executing the multistep `structuredOp` operation the plan is suspended until the `userB` agent would have used `myartifact`—updating its state by means of the `update` operation—so that the second step of the multistep operation can be executed and the operation complete. While the plan of an agent is suspended, the agent can anyway carry on other plans or react to percepts. We show this in the example. `userA`, before executing `structuredOp`, starts observing `myartifact` by doing a `focus`; then, before the completion of the operation (action), by means of a proper plan the agent reacts to the `step1_completed` event signaled by the artifact at the end of the first step, and prints a message on the console. This message will be printed always before the one the agent prints on the console *after* the completion of the `structuredOp` operation.

It is worth noting that mapping observable properties as beliefs—when focusing an artifact—implies that their value can be dynamically inspected by the agent simply by accessing the belief base. In the example, `userB`, after executing the `update` operation on the artifact four times and before stopping observing the artifact by means of a `stop_focus`, retrieves the final value of the count by means of `?count(V)`, which is the internal action in Jason to inspect the belief base.

In the remainder of the paper we will adopt Jason as reference language to program agents in the examples. In [34] the interested readers can find examples of CArtAgO exploited with

Jadex[7] agents [36], and in [39] a more general discussion including heterogeneous agent programming languages.

3.3.3 Openness and security

In previous sections we remarked how the approach supports the development of multi-agent programs exhibiting some degrees of openness: in particular heterogeneous agents can dynamically join and leave workspaces, and inside a workspace artifacts can be created and disposed dynamically by agents. Given this possibility, it is then important to introduce proper security models and strategy to control such openness. In CArtAgO a Role-Based Access Control (RBAC) model [47] is adopted for specifying and managing security aspects at workspace level, ruling agent entrance and exit, and agent access and interaction with artifacts. In particular, for each workspace a (dynamic) set of roles can be defined, and for each role policies can be specified to constrain the overall set of actions permitted to agents playing that role, including agent entrance/exit from the workspace and agent use/observation of artifacts. So the policies can specify—for instance—which artifacts an agent playing some specific role is permitted to use or which operations (actions) is allowed/not allowed to execute. The set of roles and of the policies associated to roles can be inspected and changed dynamically, both by human administrators and by agents themselves. Agents can do it by using a proper artifact—called `security-registry`—which is available in each workspace and provides a usage interface to create new roles/policies, to change existing policies, etc. Of course, also the use of this artifact is not necessarily allowed to every agent, but can be controlled in the same way.

4 Application to MAS programming

From the agent programming language/framework perspective, CArtAgO can be conceived as a general-purpose tool to be exploited in all application domains or problems in which the introduction of a suitably designed environment could be useful and effective for developing MAS [41]. In the following, first we provide a general discussion of the applicability of the approach from a methodological point of view. Then, we consider some specific contexts and problems where artifacts and CArtAgO can be exploited to improve current practice in MAS programming.

4.1 Applicability of the approach in general

From a methodological point of view, CArtAgO enriches the set of design and programming choices available to MAS developers to conceive solutions and programming systems, independently from the specific application domain. Some entities or concepts like "shared data objects", "shared resources", and "communication/coordination services", that quite inevitably appear when using MAS methodologies—an example can be found in [32], about Prometheus methodology—can be directly implemented as artifacts, whereas current approaches adopt less effective techniques. They either use dumb wrapper/mediator agents, which do not exhibit any autonomous or pro-active behaviour, or they instead break the agent abstraction level by introducing hooks to lower level mechanism, such as access to shared Java objects—as would happen in agent programming platforms like 2APL and Jadex.

[7] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/.

Besides this aspect, the new abstraction layer enhances the space of solutions that could be devised at the design (and programming) stage, adding new solutions that are possible when a notion of environment is available other than just agents. For instance, consider *stigmergic coordination* in MAS, as an example of coordination strategy that does not involve communication, but is chosen as an effective architecture in various problem domains [21,48]. It would not be possible to think and develop this strategy and architecture with only agents as design (and programming) abstractions, and without including a notion of environment modelling a pheromone ground as described in literature [33]—embedding functionalities such as aggregation and diffusion of pheromones. Moreover, the availability of a *general-purpose* computational and programming model, like the one provided in CArtAgO, makes it possible to conceive specialised mechanisms, beyond pheromone infrastructures, engineered upon properly designed artifact(s).

Actually, the benefits of exploiting first-class abstractions to model resources and tools shared and used by agents are not only related to effectiveness of the abstraction, but also to *efficiency*, thanks to the basic features provided by artifact computational and interaction model. In particular, observability and controllability—that are common properties for resources and tools—are not first-class features of agents, like instead autonomy and proactiveness, and so they must be reconstructed inefficiently on top of agent inner mechanisms and communication models.

Finally, compared to the first model introduced in [42], CArtAgO has evolved so as to make the usability of artifact-based environments by agents as seamless as possible. The one-to-one mapping between agent externs actions and artifact operations and the mapping of observable properties onto beliefs (in the case of BDI-like architectures) are the two most important features to this purpose.

In the remainder of this section we will discuss some specific problems that are relevant in MAS programming. We start from agent coordination and organisation mechanisms, which are important in particular when a multi-agent perspective is adopted, and conclude the section with resource programming, which is important also for a single-agent perspective.

4.2 Agent coordination

Agent coordination is a main aspect of programming MAS, and indeed one of the most challenging. Since artifacts are entities shared and concurrently used by agents, they can be straightforwardly exploited to provide coordinating functionalities, that is functioning as *coordination artifacts* [29]. Coordination artifacts are particularly useful in all those contexts or problems where it is useful to adopt an *objective* approach to coordination [27], i.e. encapsulating the state and the rules defining the coordination policy in some proper controllable medium, out of the interacting agents. Objective coordination is particularly useful when (i) the coordination laws are stable and the objective is to automate the coordination process as much as possible, without the need of negotiation among the participants which are even not required to know or understand the overall coordination strategy; (ii) the coordination rules must be enforced besides the individual behaviour of the participants (prescriptive coordination), but without violating their autonomy (i.e. control of their behaviour). This is possible since the enforcement of coordination rules is not a responsibility of the agents but of the medium or media that agents are using to achieve coordination. In our case, this is achieved by designing proper coordination tools as artifacts that agents create, share and use at runtime.

Also, agents' capability to replace artifacts at runtime, or to inspect and possibly change/adapt artifact behaviour, makes the approach interesting for those contexts in which the overall coordination policies need to be changed at runtime, possibly without requiring changes in participant agents.

To give a more concrete idea about the benefits of coordination artifacts, in the following we consider some well-know basic coordination problems and their solutions based on artifact-based environments.

4.2.1 Uncoupled and mediated communication

Uncoupled communication accounts for enabling communication among loosely coupled agents—i.e. agents that do not know each others or each others' location and/or they are running in different temporal contexts. Tuple spaces and related Linda language [14] are a well-known example of coordination model and language that supports uncoupled communication. To communicate, agents exploit a shared data space (the tuple space) where they insert, associatively read and retrieve structured data chunks called tuples. The communication primitives used to insert (out), read (rd), and remove (in) tuples provide a basic synchronising behaviour—in and rd complete only when a tuple matching the specified template is found in the space. In spite of its simplicity, the model makes it possible to effectively solve coordination problems by developing suitable coordination protocols based on the basic primitives. Environment programming allows for integrating in a clean and effec-

```
public class SimpleTupleSpace extends Artifact {

  TupleSet tset;

  void init(){
    tset = new TupleSet();
  }

  @OPERATION void out(String name, Object... args){
    tset.add(new Tuple(name,args));
  }

  @OPERATION void in(String name, Object... params){
    TupleTemplate tt = new TupleTemplate(name,params);
    await("foundMatch",tt);
    Tuple t = tset.removeMatching(tt);
    bind(tt,t);
  }

  @OPERATION void rd(String name, Object... params){
    TupleTemplate tt = new TupleTemplate(name,params);
    await("foundMatch",tt);
    Tuple t = tset.readMatching(tt);
    bind(tt,t);
  }

  @GUARD boolean foundMatch(TupleTemplate tt){
    return tset.hasTupleMatching(tt);
  }

  private void bind(TupleTemplate tt, Tuple t){
    Object[] tparams = t.getContents();
    int index = 0;
    for (Object p: tt.getContents()){
      if (p instanceof OpFeedbackParam<?>){
        ((OpFeedbackParam) p).set(tparams[index]);
      }
      index++;
    }
  }
}
```

```
// philosopher agent in dining philosophers
// problem solved with a tuple space artifact

!living.

+!living
 <- !thinking;
    !eating;
    !!living.

+!eating
 <- !acquireRes;
    !eat;
    !releaseRes.

+!acquireRes :
  my_left_fork(F1) & my_right_fork(F2)
  <- in("ticket");
     in("fork",F1);
     in("fork",F2).

+!releaseRes:
  my_left_fork(F1) & my_right_fork(F2)
 <-  out("fork",F1);
     out("fork",F2);
     out("ticket").

+!thinking <- ...
+!eat <- ...
```

Fig. 9 *Left*: The skeleton of a tuple space implemented as an artifact. *Right*: A philosopher agent in the dining philosopher problem exploiting the tuple space to coordinate with the other philosopher agents

tive way tuple spaces in the context of MAS programming, by conceiving tuple spaces as environment abstractions providing coordination functionalities that agents can exploit by means of a basic set of actions representing the coordination primitives. In particular, using artifact-based environments, a tuple space can be suitable programmed as an artifact shared and used by agents, where the coordination primitives are mapped onto artifact usage interface. Figure 9 shows an almost complete cut-out of a simple tuple space implementation, with in evidence the `out`, `in` and `rd` operations. The synchronising behaviour of `in` and `rd` operations is realised by means of multi-step operations. Distributed implementation of tuple spaces can be realised by exploiting multiple artifacts, representing different portions of the tuple space, and linkability. As an example of concrete usage, Fig. 9 shows also (on the right) the implementation of a philosopher agent to solve the dining philosopher problem [11] using a MAS, exploiting the tuple space to coordinate with the other philosopher agents. To eat the agent first retrieves a tuple `ticket`—if the number of philosophers is N, then $N-1$ `ticket` tuples are available in the space—then the two tuples `fork(F)`, representing the resources needed to eat (the left fork and the right fork). The solution guarantees both mutual exclusion in accessing the resources and the absence of deadlock. It's worth noting here that the agents exploit `out`, `in`, `rd` operations provided by a tuple space directly as external actions, with input/feedback parameters, and this makes the agent code quite clean and compact.

Besides tuple spaces, uncoupled communication is used in different kinds of architectures introduced in the engineering of concurrent systems. A main and common one is *producers–consumers*, that involves the coordinated activities of N producer agents that produce some information items that need to be processed by M consumer agents. Each one of the N agents repeatedly produce information items that can be consumed and processed by *any* of the M consumers. N and M values can possibly change at runtime. To

```
public class BoundedBuffer extends Artifact {
  private LinkedList<Item> items;
  private int nmax;

  void init(int nmax){
    items = new LinkedList<Item>();
    defineObsProperty("n_items",0);
    this.nmax = nmax;
  }

  @OPERATION(guard="bufferNotFull")
  void put(Item obj){
    items.add(obj);
    int ni = items.size()+1
    updateObsProperty("n_items",ni);
  }

  @GUARD boolean bufferNotFull(Item obj){
    return items.size() < nmax;
  }

  @OPERATION(guard="itemAvailable")
  void get(OpFeedbackParam<Item> res){
    Item item = items.removeFirst();
    int ni = items.size()-1;
    updateObsProperty("n_items",ni);
    result.set(item);
  }

  @GUARD boolean itemAvailable(){
    return items.size() > 0;
  }
}
```

```
// generic producer agent skeleton

!produce.

+!produce: true <-
  ?nextItemToProduce(Item);
  put(Item);
  !!produce.

+?nextItemToProduce(Item) : true
  <- // compute next item ...
```

```
// generic consumer agent skeleton

!consume.

+!consume: true <-
  get(Item);
  !consumeItem(Item);
  !!consume.

+!consumeItem(Item) : true
  <- // process item ...
```

Fig. 10 *Left*: Implementation of an artifact functioning as a bounded buffer in producers–consumers architectures. *Right*: Skeletons of a generic producer and a generic consumer agent concurrently exploiting the buffer

coordinate the activities of producers and consumers, a *bounded buffer* is typically introduced [2], where producers insert items and consumers retrieve them. Besides functioning as a shared data store uncoupling the work of producers/consumers—so that a producer has not to wait a consumer to be available for consuming the item—the buffer synchronises agent activities, in particular the action of a consumer retrieving an item from the buffer is suspended until at least one item is available. Viceversa, the action of a producer inserting an item is suspended until the buffer is not full. By adopting artifact-based environments, a bounded-buffer can be effectively implemented as an artifact (see Fig. 10, left), providing operations to insert (`put`) and remove (`get`) items and encapsulating the mechanisms needed to synchronise the agents. Note that, given the semantics adopted for action/operation execution, a consumer agent—for instance—waiting to get a new item from an empty bounded buffer is not blocked: it has just a suspended intention—which will be eventually resumed as soon as the `get` action has completed—so that the agent can react to other percepts generated by the environment and carry on other intentions.

4.2.2 Task synchronisation

Both the tuple space and the bounded buffer examples involve a form of synchronisation of agent actions. Generalising this point, environment programming can be used to implement specific coordination mechanisms to effectively synchronise agent activities without necessarily using communication protocols and without requiring agents to know each others. As an example, suppose that we need to synchronise the activities of N agents so that before executing their next task T_2 they must wait each other to complete tasks T_1. A simple but expensive solution purely based on message passing accounts for each agent sending a message to each other $N-1$ agents as soon as it reaches the meeting point and then waiting to receive $N-1$ messages before proceeding. This solution involves the exchange of $N(N-1)$ messages. By adopting a solution with a mediator agent functioning as coordinator, the number of messages is reduced to $2N$. An alternative, more efficient solution using the environment accounts for introducing an artifact functioning as a *barrier*, providing a single `synch` operation (see Fig. 11, left). In this case, in order to synchronise, agents need to do a single `synch` action (see Fig. 11, right), so only N actions are required in the overall. More complex examples of task synchronisation include the development of coordination artifacts such as shared task schedulers and workflow engines [49].

```
public class Barrier extends Artifact {

  int nSynchToWait;

  void init(int nParticipants){
     nSynchToWait = nParticipants;
   }

  @OPERATION  void synch(){
    nSynchToWait--;
    await("allReady");
  }

  @GUARD boolean allReady(){
    return nSynchToWait == 0;
  }
}
```

```
// one of the N agents that need to synch


+!synchronisedTask: synchTool(Barrier) <-
  ...
  !task_T1;
  synch [artifact_name(Barrier)];
  !task_T2;
  ....
```

Fig. 11 *Left*: Implementation of an artifact functioning as a synchronisation barrier. *Right*: Skeleton of a generic agent using the barrier to achieve a synchronisation point with other agents

4.3 Social and organisation mechanisms

Being the conceptual place defining agent actions and percepts, the environment can be the natural locus where to encapsulate and enforce rules constraining agent actions and interactions, according to some design objective of the agent system in the overall. This can be exploited so as to implement those social and organisation mechanisms that can be found, for instance, in electronic institutions [24]. Without the availability of environment abstractions, this kind of systems is typically implemented by introducing a layer of mediator agents, collecting and processing any action request from participant agents. In that case, any action of participant agents that is meant to be ruled by the laws of the system must be realised as a communicative action towards mediator agents. For instance, in AMELI [12]—a middleware for building e-Institutions—a *governor* agent is introduced for each participant agent, mediating its communicative actions. Environment programming allows for another perspective, which accounts for directly designing and programming first-class environment computational entities embedding and enforcing those social and organisational mechanisms and laws constraining and ruling agent actions. By adopting artifact-based environments, in particular, those entities are mapped onto one or multiple linked artifacts, exposing a usage interface that corresponds to agent actions and exploiting artifact computational behaviour to implement and enforce the rules. As a toy example, consider the development of a game like TicTacToe as a MAS. Agents can be straightforwardly used to implement artificial players of the game. The game-board, which defines and enforce the game rules, can be quite intuitively programmed as an artifact, used by the player agents. Figure 12 shows a sketch of an artifact implementing a game board and the skeleton of the player agents using it. The board defines the possible actions (moves) that the agent can do depending on the stage of the game—in this case `newGame` for starting a game and `move` to play during a game—and encapsulates the rule of the game, making the observable state of the game perceivable as observable properties—in particular, `pos(X,Y,S)` keeps track of the cells content, `state` represents the state of the game (playing or finished) and `turn` indicates which player has to move next. The agent player observing the game board chooses a new move as soon as it perceives that it is its turn, and do the move by using the game-board. As soon as it perceives a signal about who is the winner, it prints a message on the console.

By exploiting this principle, it is quite straightforward to design environments that make it possible to monitor the actions performed by agents, keep track of violations—possibly making them observable so as to trigger the reaction of agents in some specific institutional role—and even apply sanctions—by changing the set of actions that the agent is allowed to do. Following this line, a concrete example of artifact-based environments applied to the context of *Organisation Oriented Programming* is the ORA4MAS proposal [19]. Organisation Oriented Programming is concerned with the introduction of proper organisation modelling languages and middleware for supporting organisations and their members [3]. One of the challenges in this context is to conceive and design proper infrastructures (i.e. middleware) for enacting roles, norms and global goals defined by the organizational models without violating agent autonomy, and, at the same time, supporting *open* organisations. To this end, ORA4MAS infrastructure exploits artifact-based environments to embody an Organisational Management Infrastructure—specified in this case with the MOISE+ organisation modelling language [20]. The organisation is deployed in agent environments in terms of a set of *organisational artifacts*, which are, on the one side, used by organisation members to access services, and, on the other side, controlled by organisational agents aimed at further

```
public class CheckerBoard extends Artifact {

  void init(int nmax){
    for (int i = 0; i < 3; i++){
      for (int j = 0; j < 3; j++){
        defineObsProperty("pos",i,j,"empty");
      }
    }
    defineObsProperty("state","no-play");
    defineObsProperty("turn","-");
  }

  @OPERATION void newGame(){
    updateObsProperty("state","playing");
    resetGameBoard();
    updateObsProperty("turn","cross");
  }

  @OPERATION void move(int x, int y, String sign){
    if (isFree(x,y) && rightTurn(sign)){
      updateObsProperty("pos",x,y,sign);
      if (won(sign)){
        updateObsProperty("state","terminated");
        signal("winner",sign);
      } else {
        updateObsProperty("turn",oppositeSign(sign));
      }
    } else {
      failed("wrong_move");
    }
  }

  boolean notPlaying(){...}
  boolean playing(){...}

  private void resetGameBoard(){...}
  private boolean rightTurn(String sign){...}
  private boolean isFree(int x, iny y){...}
  private boolean won(String who){...}
  private String oppositeSign(String s){...}
}
```

```
// agent player skeleton

winning_pos(Sign,0,0) :-
          (pos(0,1,Sign) & pos(0,2,Sign)) |
          (pos(1,1,Sign) & pos(2,2,Sign)) |
          (pos(1,0,Sign) & pos(2,0,Sign)).

winning_pos(Sign,1,0) :-
          (pos(0,0,Sign) & pos(2,0,Sign)) |
          (pos(1,1,Sign) & pos(1,2,Sign)).
...

+turn(Sign) : my_sign(Sign)
  <- !choose_move(X,Y);
     move(X,Y,Sign).

+winner(Winner) : my_sign(Winner)
  <- println("I'm the winner!").

+winner(Winner) : not my_sign(Winner)
  <- println("sob.").

+!choose_move(X,Y) : my_sign(S) &
                     winning_pos(S,X,Y).
+!choose_move(X,Y) : adv_sign(S) &
                     winning_pos(S,X,Y).
+!choose_move(X,Y) <- ...
```

Fig. 12 *Left*: Implementation of the board of a game (TicTacToe in this case) as an artifact. *right*: Skeleton of a player agent, using and observing the board to play

controlling system dynamics. In other terms, this approach reifies the organisation as a set of distributed first-class entities populating the agent world, namely organisational artifacts, which agents can create and cooperatively use.

4.4 Programming resources

Finally, besides being used to implement coordination and organisation facilities, environment programming can be used to create an abstraction layer to model any kind of (non-agent) resource used by agents, either internal resources that are functional to agent activities (such as a large knowledge base, a calculator, a personal agenda) or wrapping external resources existing in the deployment context or outside the system (such as a Web Services, a data-base, a legacy system). In the following we discuss some main examples.

4.4.1 Agent libraries

A first simple kind of resources that we consider are *libraries* extending the actions of an agent so as to include new functional capabilities. From a software engineering perspective, the possibility of modularising and extending agent capabilities, possibly at runtime, is a main issue. Different kinds of solution have been proposed in the context of agent programming, introducing new constructs to group, encapsulate and reuse in well-defined modules agent features, that can vary according to the architecture or model adopted—a main example is the concept of *capability* adopted in BDI agent programming platforms such as Jack [8] and

```
public class Agenda extends Artifact {

  @OPERATION
  void schedule(String todo, long when){
    execInternalOp("genAlarm",todo,when);
  }

  @INTERNAL_OPERATION
  void genAlarm(String todo, long dt){
    await_time(dt);
    signal("todo",todo);
  }
}
```

```
// an agent using the agenda
...
+!setupAgenda :
  task1_date(T1) & task2_date(T2)
  <- make_artifact("my_agenda","Agenda",Id);
     focus(Id);
     schedule("task1", T1);
     schedule("task2", T2).

/* my scheduled activities*/

+todo("task1"): true
<- // activities related to task1

+todo("task2"): true
<- // activities related to task2
```

Fig. 13 A sketch of a personal agenda artifact and a usage example by an agent

Jadex [6]. Environment programming allows for a complementary approach, in which agent libraries are *externalised* into properly programmed artifacts, exploiting artifacts as agent modules or rather *personal tools* extending agent capabilities. Figure 13 shows a simple example, a *personal agenda* artifact that can be used by an agent to annotate (schedule) tasks to perform some time in the future. The artifact will eventually generate a `todo` signal at the specified time, so as to allow the agent to react with a proper plan and perform the task. First investigations about this perspective are reported in [40].

Actually, among the tools here we include those artifacts wrapping and providing a clean agent-level access to existing libraries written in other programming languages, such as Java, C or C++. The point here is not simply integrating low-level mechanisms provided by foreign languages, but rethink those functionalities at the agent abstraction level. Graphical user interface (GUI) toolkits are an example. By exploiting CArtAgO, it is possible to program GUIs inside a multi-agent program as artifacts mediating the interaction between humans and agents. To this end, a basic abstract artifact `GUIArtifact` is provided among the basic tools of CArtAgO, to be extended in order to create concrete GUI. Actually this possibility is provided by almost any agent programming platform based on Java, for instance, exploiting Java GUI toolkits such as Swing: however we argue that using the artifact programming model to define essential aspects of GUI as part of the agent environment makes the approach cleaner and with a stronger separation of concerns—the agent source code is not "polluted" with low-level OO code. Figure 14 shows a simple example, in which an agent uses a GUI to repeatedly get some inputs from the user and to visualise some output. In particular, the agent creates a GUI artifact called `MySimpleGUI`, providing one button and one edit text. The structure of the GUI—based on Java Swing library—is defined by the `MyFrame` class, as it would be in a traditional OO program. An instance of this class is created inside `MySimpleGUI` and events generated by the GUI components are linked to internal operations of the artifact by means of a set of predefined methods implemented in `GUIArtifact`. In particular an action event generated by `frame.okButton` causes the execution of the internal operation `ok`, which generates an observable event `ok`. The enter key stroke event generated by `frame.text` causes the execution of `updateText`, which updates the value of an observable property `value` with the current value of the edit text; and finally the window closing event is mapped onto the `closed` operation, which generates a `signal` event. The agent who is observing the GUI reacts to every change of the observable property and to `ok` and `close` observable events, respectively logging the new value on the console, acting upon the GUI to set the value incremented by one and shutting down. The screenshot shows the dynamics and the behaviour of the agent: first the agent perceives and prints on the console the initial value of the observable

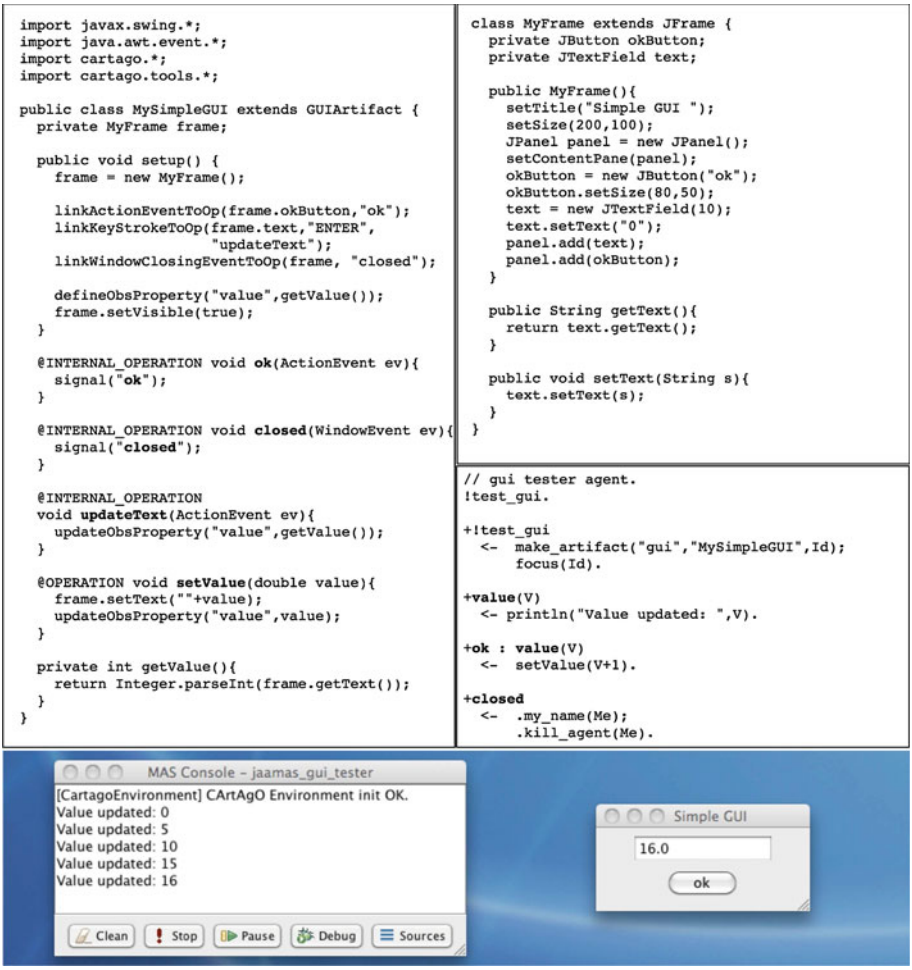


Fig. 14 An example of a GUI realised as an artifact (`MySimpleGUI`) and an agent exploiting it to interact with the user

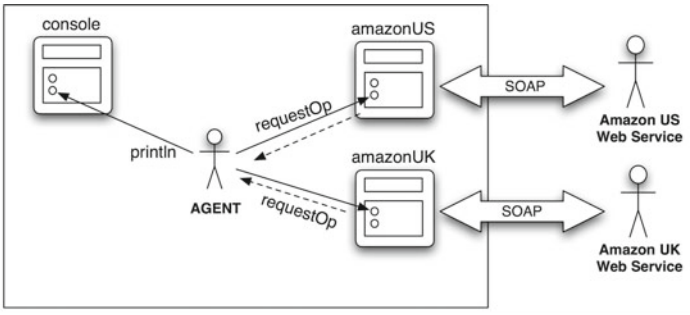
property `value`, which is zero. Then the user inserts the numbers five and ten pressing enter each time, so the internal artifact operation `updateText` is executed and the observable value updated. The agent reacts to the updates and logs them on the console. Finally the user inserts the value 15 and presses the button: the internal operation `ok` is executed, generating the signal `ok` which is then perceived by the agent which executes the operation `setValue` setting the value to the one perceived plus one (so 16).

Summarising the outcomes, externalization allows for: (i) extending agent action repertoire without the need to extend agent architectures/languages; (ii) reducing the computational burden on the agent side—agents do not waste time and computational resources for the execution of the operation and processes related to the externalised functionalities, which are instead executed inside artifacts; (iii) enhancing reusability—tools (artifacts) can be flexibly re-used among heterogeneous agents, even developed with different agent programming languages. (iv) dynamic extensibility—tools can be created/disposed at runtime by need.

4.4.2 External resources

The integration and interaction with existing systems and technologies is a very common issue when implementing real applications with MAS. An example among the others is the integration of agent-based systems with Web based technologies (used as case study in many agent books, such as [5,31]). A typical solution to this problem accounts for either extending the set of agent actions with ad hoc new actions, which enable the interaction with external systems, or introducing wrapper agents that play the role of mediators, encapsulating the machinery required to interact with the external system—as the WSIG gateway agent proposed in FIPA platform for mediating the interaction with Web Services [16].

By exploiting environment as a first-class abstraction, an alternative solution is designing a proper environment providing agents with the functionalities needed to access and interact with the resources. In particular, besides enabling and ruling the access, artifacts allow for directly representing those resources as first-class entities of the MAS, which can be dynamically configured/managed/adapted by agents. An example is described in [37], in which a basic set of artifacts is introduced to interact with Web Services and to implement Web




```
!test_ws.

+!test_ws : true <-
  make_artifact("amazonUS","cartagows.WSInterface",
                "http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl");
  make_artifact("amazonUK","cartagows.WSInterface",
                "http://webservices.amazon.com/AWSECommerceService/UK/AWSECommerceService.wsdl");
  // configure the tools to access the services
  setWSAddressingSupport(false)[artifact_name("amazonUS")];
  setWSAddressingSupport(false)[artifact_name("amazonUK")];

  // do requests
  !do_requests.

+!do_requests <-
  Req    = "<ItemSearch xmlns='http://ecs.amazonaws.com/AWSECSCommerce/onca/soap'>
              <AWSAccessKeyId>18HK8VQHV5AZMHSPHC82</AWSAccessKeyId>
              <Request>
              <SearchIndex>Books</SearchIndex>
              <Keywords>Charles%20Bukowski%20Tales%20Madness</Keywords>
              </Request>
              </ItemSearch>";
  xml_lib.buildDOM(Req,Params);

  // make the requests to the services
  requestOp("ItemSearch",Params)[artifact_name("amazonUS")];
  requestOp("ItemSearch",Params)[artifact_name("amazonUK")].

// process the responses
+ws_response(Result,RelatedTo)[source(percept),artifact_name(_,Service)]
  xml_lib.getDOMElemValue(Result,"ItemSearchResponse.Items.TotalResults",Value);
  println("Hits in service ",Service,": ",Value).
```

Fig. 15 An agent interacting with two Web Services by exploiting two `WSInterface` artifacts

Services in the Service-Oriented Architecture (SOA) context. A simple case taken from that work is shown in Fig. 15, where an agent uses two instances of an artifact called `WSInterface` concurrently to interact with external Web Services. The artifact is created by specifying a WSDL and provides a usage interface to invoke the operations provided by the Web Service—for instance, the `requestOp` operation in the example—and also to configure aspects related to the quality of the service—e.g. the `setWSAddressingSupport` operation. So, the agent exploits the functionality of a Web Service by using a properly configured `WSInterface`, to interact with multiple Web Services at the same time multiple instances of this artifact are used. In the example in Fig. 15, a Jason agent interacts with two Amazon Web Services, the one of US and the one of UK, by invoking the `ItemSearch` operation to search for books matching the keywords `Charles Bukowski Madness`. Responses in this case are generated asynchronously as `ws_response` observable events, which are perceived and processed by the agent to print out the results.

Another example is about using agent technologies for automating the management of virtual resources—virtual machines in particular—running on top of a virtualisation infrastructure (examples are VMWare, Virtual Box, Xen). The idea is to experiment the use of agent programming languages to implement autonomic systems that monitor the dynamic behaviour of pools of virtual machines and autonomously apply administration policies so as to improve the overall efficiency of the system—for instance by switching off/on virtual machines, moving virtual machines from a physical node to another one, etc. To this end, we designed a set of artifacts that wrap existing virtualisation technologies and make them usable from agents—the library (called CArtAgO-VM) is available on CArtAgO Web Site. Among the other, a `VirtualMachine` artifact is introduced to represent and control an existing system virtual machine,[8] providing a usage interface to both control the virtual machine—to switch on and off the machine, to freeze/unfreeze its state—and observable properties to make current VM state—CPU load, the memory used, and so on—observable to interested agents. Another kind of artifact, `DataCenter`, provides functionality to manage the pool of virtual machines, for instance to create a new machine or to move a virtual machine from a pool to another pool. Then, an agent-based autonomic system in this case is made of agents that autonomously manage pools of virtual machines by observing related artifacts and reacting as soon as some event or condition (such as CPU overload, machine shutdown) needs some kind of action (such as load balancing, machine reset).

Summing up, the approach provides a principled way to reuse in agent languages existing technologies and libraries, typically developed in mainstream languages such as Java or C, suitably wrapped in artifacts and represented as first-class entities in the agent world.

5 Related works

Many research works have been developed so far exploiting a notion of environment in the context of MAS in general (see [55,56] for a survey). To the authors' knowledge, no works have been developed, however, in the specific context of *MAS programming*, to explore what kind of impact the notion of environment as a first-class programming abstraction can have in programming multi-agent systems, in particular using existing agent programming languages. Consequently, also the issue of defining general-purpose computational/programming models for environment programming is new. Besides ours, two works that indirectly consider

[8] Virtual machines can be of two kinds: *application* virtual machines—executing programs compiled in the machine language of the virtual machine (examples are the Java Virtual Machine and Microsoft CLR)—and *system* virtual machines—emulating the hardware of a physical machine.

this issue are MadKit [17] and GOLEM [7]. MadKit has been one of the first general-purpose Java-based framework for developing multi-agent systems, implementing the action model devised by Ferber and Müller in [13]. Even if not explicitly introducing a computational and programming model for the environment, in practice the framework allows for programming the environment in terms of *objects* embedding some computational behaviour. Actually, this programming support has been exploited in particular for defining the behaviour of the environment in MAS-based simulations, implemented on top of MadKit. GOLEM is a recent logic-based framework that allows for representing an agent environment declaratively, as a composite structure that evolves over time, including two main classes of entities—agents and *objects*—organised in *containers*. Besides being described in a logic-based framework, the features of the objects and containers strongly resemble those of artifacts and workspaces. Interactions between these entities inside a container are specified in term of events whose occurrence is governed by a set of physical laws specifying the possible evolutions of the agent environment, including how these evolutions are perceived by agents and affect objects and other agents in the environment.

Outside the context of MAS programming, most of the research works related the environment in MAS engineering concerns the definition of environment-based mechanisms (see [35] for a survey) useful to solve specific class of problems—such as agent communication and coordination. A wider perspective is adopted by Weyns and Holvoet in [53], where a reference model of a generic environment architecture is proposed. It consists of a set of *modules* that represent core functionalities of the environment and describes the functional relationships in terms of flows between these modules. The decomposition is primarily driven by the way agents interact with the environment. An agent can sense the environment to obtain a percept (i.e. a representation of its vicinity), an agent can perform an action in the environment (i.e. attempting to modify the state of affairs in the environment), and it can exchange messages with other agents. Besides the architectural description, also a formal model has been proposed [52], essentially keeping the same decomposition perspective. The action model adopted by Weyns and Holvoet is an extension of the *influences* and *reactions* model proposed by Ferber and Müller originally in [13].

Artifact-based environments introduce a different kind of modularisation with respect to the one promoted by the reference architecture introduced by Weyns and Holvoet. In the reference architecture described in [53], modularisation is from the point of view of MAS engineers: modules represent basic blocks of the software architecture of the environment, encapsulating some functions that developers can reuse and specialise when developing concrete environments for specific applications. From the agent view point, the environment is still perceived as a monolithic entity, providing actions to act upon it and producing stimuli. In artifact-based environments, artifacts are modules both from the MAS engineer viewpoint and the agent viewpoint. As in the case of the reference model, artifacts as modules encapsulate some kind of *function*, however not from the MAS engineer point of view as in Weyns and Holvoet work, but from the agent point of view. For this reason, the agent–environment interaction model in artifact-based environments can be refined beyond actions and percepts as in the reference model in [53], introducing the basic set of actions described in Sect. 3. As a consequence of this different kind of modularisation, in our perspective the modules (the artifacts) are dynamically instantiated and disposed (by agents), and their type changes according to their specific functionalities; in the reference model and related specific models implementing it, typically modules are static, even if customised according to the specific application needs. In turn, our modularisation is orthogonal to the one in the reference model, for a single artifact typically manages most aspects dealt with by all modules of the reference model. Another consequence concerns the role of the computational and programming

model for the environment: in our case, it is explicitly defined and its basic features are crucial, since they strongly influence the way in which the application environment is programmed and engineered. In Weyns and Holvoet's reference model perspective—which is more akin to software architectures—no computational/programming model is needed for the environment: it can be possibly introduced by specific applications.

Related to environment programming are the works concerning coordination models and languages introduced in MAS. The notion of mediated interaction and the introduction of proper *coordination media* as first-class abstractions to design the agent interaction space is a cornerstone of the research on coordination models and languages [9,15]. The *tuple space* model and the related *Linda* coordination language [14] are main examples, exploited today by industrial-strength technologies for the development of distributed systems. These research works strongly influenced A&A and CArtAgO, in particular—besides Activity Theory—the artifact abstraction was inspired by *programmable coordination media*, like tuple centres [26] adopted in the TuCSoN coordination infrastructure [30]. In fact, in earlier works, tuple centres have been used as concrete model to implement the concept of *coordination artifact* [29], which has been generalised then into the notion of artifact [28], then implemented in CArtAgO, with the introduction of specific computational and programming models uncoupled from tuple centres. Compared to the basic notion of (programmable) coordination medium, the artifact abstraction can be considered, on the one side, a *generalisation* beyond the coordination purpose, introducing to this end new key concepts such as the usage interface, a notion of observable state, and the manual; on the other side, it can be considered a *specialisation* of the concept in the context of MAS and agent-oriented programming, with specific features that are thought to be exploited by agents as defined in existing agent programming languages/architectures, not simply by processes of a distributed system.

Finally, compared to our previous works, this paper provides a revised and extended discussion of aspects initially discussed in earlier papers, from the first papers on CArtAgO [44] and A&A [43] to recent contributions to environment programming [41], and includes also recent improvements in CArtAgO computational/programming model and technology.

6 Conclusion

By conceiving the environment as a programmable part of the MAS besides agents, designers and programmers have a new dimension to devise solutions to complex problems, and to build effective systems. When the *programming* perspective is assumed, the availability of *general-purpose* computational and programming models is a main concern. The computational/programming model introduced by artifact-based environments and implemented by CArtAgO is meant to play this role, providing those features that are important from a software engineering point of view, in particular: (i) *abstraction*—it preserves the agent abstraction level, since the main concepts used to define application environments, i.e. artifacts and workspaces, are first-class entities in the agents world, and the interaction with agents is built around the agent-based concepts of action and perception (use and observation); (ii) *modularity and encapsulation*—it provides an explicit way to modularise the environment, where artifacts are components representing units of functionality, encapsulating a partially-observable state and a set of operations; (iii) *extensibility and adaptation*—it provides a direct support for environment extensibility and adaptation, since artifacts can be dynamically constructed (instantiated), disposed, replaced, and adapted by agents; (iv) *reusability*—it promotes the definition of types of artifact that can be reused as tools in

different application contexts, such as in the case of coordination artifacts empowering agent interaction and coordination, such as blackboards and synchronisers.

Indeed, the work developed so far can be considered just the stepping stone to explore further aspects that are important when the engineering dimension is considered. We mention here two main issues among the others: the first is devising a rigorous definition of the aforementioned notion of *type* for artifacts—related to the functionality they provide—so as to both exploit it to detect errors in agent programs at compile time and to investigate aspects related to artifact extendibility, reusability, and substitutability. The second is devising methodologies for testing and validating artifact-based environments and investigating how, more generally, this could impact on approaches that are currently proposed for MAS testing and validation, including model-checking [4]. To this end, the definition of a formal model for the artifact abstraction and artifact-based environments appears an essential step. First explorations about this point can be found in [45].

Then, the orthogonality of CArtAgO with the agent model/architecture adopted makes it possible, on the one side, to take advantage of these features from existing agent programming platforms—Jason and Jadex are two main examples—and, on the other side, to foster the programming of *heterogeneous* MAS, with agents developed using different languages working together inside the same artifact-based environment. Indeed, to achieve full support for *openness*, further work is needed to find out an effective model for describing, in particular, the artifact manual. The definition of a proper model and language for the manual is fundamental also for another main issue which is part of our future work, i.e. the *cognitive selection* and *use* of artifacts, which concerns the development of MAS where agents dynamically select which artifacts to use/create according to their goals and dynamically discover and learn how to use them, by consulting the manual. First investigations about this point are reported in [34]. For these issues, existing works on ontologies and related languages and reasoning frameworks—such as the ones developed in the context of Semantic Web, like OWL—will be a main reference.

An aspect not developed in this paper, which is actually a main issue when the programming perspective is of concern, is the definition of specific *languages* for programming the environment, embodying the computational and programming models. Being conceived as an annotation-based framework on top of the Java language and platform, CArtAgO does not introduce a truly new programming language. This has immediate advantages from the point of view of the deployability and (re-)usability of the framework, but also introduces some drawbacks, such as the verbosity of the source code, the weak support in finding errors at compile time, and, more generally, the abstraction gap between the programming model and the language used to implement it. So a main part of our future work will be the definition of a specific language to program artifact-based environments, still based on object-oriented languages such as Java to define the environment data model but allowing the definition of artifacts by means of first-class constructs of the language.

Then, the availability of an environment abstraction layer for designing and programming MAS leads to an extension of the notion of *interoperability* (which is typically related only to communication): agents inter-operate not only by exchanging messages using common agent communication languages and ontologies, but also by being situated and working in the same computational environment, sharing resources and tools. Accordingly, a main issue to be considered in future research works about environment design and programming—valuable in particular in the AOSE perspective—is the definition of common ontologies and standards related to the environment, analogously to what FIPA has done so far for agent communication languages and platforms.

Finally, a main issue for both AOSE and MAS programming is bridging the gap between designing MAS and programming MAS, or rather between agent-oriented methodologies and agent programming languages: in this context, we believe that the basic concepts promoted by environment programming and, more specifically, artifact-based environments, could be useful even at the design level, besides the programming level, so as to conceive in current agent methodologies—such as Prometheus [32]—a support for *environment design* aside to agent design. First exploration in this direction have been done in the SODA methodology [22].

References

1. Bellifemine, F. L., Caire, G., & Greenwood, D. (2007). *Developing multi-agent systems with JADE*. Chichester: Wiley.
2. Ben-Ari, M. (2006). *Principles of concurrent and distributed programming*. Boston: Addison-Wesley.
3. Boissier, O., Hübner, J. F., & Sichman, J. S. (2007). Organization oriented programming: From closed to open organizations. In G. O'Hare, O. Dikenelli, & A. Ricci (Eds.), *Engineering societies in the agents world VII (ESAW 06)*, volume 4457 of *LNCS* (pp. 86–105). Berlin/Heidelberg: Springer.
4. Bordini, R. H., Fisher, M., Visser, W., & Wooldridge, M. (2006). Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems, 12*(2), 239–256.
5. Bordini, R., Hübner, J., & Wooldridge, M. (2007). *Programming multi-agent systems in agentspeak using Jason*. Hoboken: Wiley-Interscience.
6. Braubach, L., Pokahr, A., & Lamersdorf, W. (2005). Extending the capability concept for flexible BDI agent modularization. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni (Eds.), *Programming multi-agent systems*, volume 3862 of LNAI (pp. 139–155). Berlin/Heidelberg: Springer.
7. Bromuri, S., & Stathis, K. (2008). Situating cognitive agents in GOLEM. In D. Weyns, S. Brueckner, & Y. Demazeau (Eds.), *Engineering environment-mediated multi-agent systems*, volume 5049 of LNCS (pp. 115–134). Berlin/Heidelberg: Springer.
8. Busetta, P., Howden, N., Rönnquist, R., & Hodgson, A. (2000). Structuring BDI agents in functional clusters. In N. Jennings & Y. Lespèrance (Eds.), *Intelligent agents VI*, volume 1757 of LNAI (pp. 277–289). Berlin/Heidelberg: Springer.
9. Ciancarini, P. (1996). Coordination models and languages as software integrators. *ACM Computing Surveys, 28*(2), 300–302.
10. Dastani, M. (2008). 2APL: a practical agent programming language. *Autonomous Agent and Multi-Agent Systems, 16*(3), 214–248.
11. Dijkstra, E. W. (1971). Hierarchical ordering of sequential processes. *Acta Informatica, 1*(2), 115–138.
12. Esteva, M., Rodríguez-Aguilar, J. A., Rosell, B., & Ameli, J. L. (2004). An agent-based middleware for electronic institutions. In N. R. Jennings, C. Sierra, L. Sonenberg, & M. Tambe (Eds.), *Proceedings of the 3rd international joint conference on autonomous agents and multi-agent systems (AAMAS'04)*, New York, USA (pp. 236–243). ACM.
13. Ferber, J., & Müller, J.-P. (1996). Influences and reaction: A model of situated multi-agent systems. In *Proceedings of the 2nd international conference on multi-agent systems (ICMAS'96)*. AAAI Press.
14. Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems, 7*(1), 80–112.
15. Gelernter, D., & Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM, 35*(2), 96.
16. Greenwood, D., Lyell, M., Mallya, A., & Suguri, H. (2007). The IEEE FIPA approach to integrating software agents and web services. In *Proceedings of the 6th international joint conference on autonomous agents and multi-agent systems (AAMAS'07)*, Honolulu, Hawaii (pp. 1–7). ACM.
17. Gutknecht, O., & Ferber, J. (2000). The MADKIT agent platform architecture. In *Agents workshop on infrastructure for multi-agent systems* (pp. 48–55).
18. Hindriks, K. V. (2009). Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni (Eds.), *Multi-agent programming: Languages, platforms and applications* (Vol. 2, pp. 3–37). Berlin/Heidelberg: Springer.
19. Hübner, J. F., Boissier, O., Kitio, R., & Ricci, A. (2009). Instrumenting multi-agent organisations with organisational artifacts and agents: "Giving the organisational power back to the agents". *Autonomous Agents and Multi-Agent Systems*. doi:10.1007/s10458-009-9084-y.

20. Hübner, J. F., Sichman, J. S., & Boissier, O. (2007). Developing organised multi-agent systems using the MOISE+ model: Programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering, 1*(3/4), 370–395.
21. Mamei, M., & Zambonelli, F. (2009). Programming pervasive and mobile computing applications: The TOTA approach. *ACM Transactions on Software Engineering and Methodology, 18*(4), 1–56.
22. Molesini, A., Omicini, A., Denti, E., & Ricci, A. (2005). SODA: A roadmap to artefacts. In O. Dikenelli, M.-P. Gleizes, & A. Ricci (Eds.), *Engineering societies in the agents world*, volume 3963 of LNCS (pp. 49–62). Berlin/Heidelberg: Springer.
23. Nardi, B. (Ed.). (1996). *Context and consciousness: Activity theory and human-computer interaction*. Cambridge: MIT Press.
24. Noriega, P., & Sierra, C. (2002). Electronic institutions: Future trends and challenges. In M. Klusch, S. Ossowski, & O. Shehory (Eds.), *Cooperative information agents VI*, volume 2446 of LNAI. Berlin/Heidelberg: Springer.
25. Odell, J., Parunak, H. V. D., Fleischer, M., & Brueckner, S. (2003). Modeling agents and their environment. In *Agent-oriented software engineering III*, volume 2585 of LNCS (pp. 16–31). Berlin/Heidelberg: Springer.
26. Omicini, A., & Denti, E. (2001). From tuple spaces to tuple centres. *Science of Computer Programming, 41*(3), 277–294.
27. Omicini, A., & Ossowski, S. (2003). Objective versus subjective coordination in the engineering of agent systems. In M. Klusch, S. Bergamaschi, P. Edwards, & P. Petta (Eds.), *Intelligent information agents: An agentlink perspective*, volume 2586 of LNAI, (pp. 179–202). Berlin/Heidelberg: Springer.
28. Omicini, A., Ricci, A., & Viroli, M. (2008). Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems, 17*(3), 432–456.
29. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., & Tummolini, L. (2004). Coordination artifacts: Environment-based coordination for intelligent agents. In *Proceedings of the 3rd international joint conference on autonomous agents and multi-agent systems (AAMAS'04)*, New York, USA, 19–23 July 2004 (Vol. 1, pp. 286–293). ACM.
30. Omicini, A., & Zambonelli, F. (1999). Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems, 2*(3), 251–269.
31. Padgham, L., & Wiknikoff, M. (2004) *Developing intelligent agent systems: A practical guide*. Chichester: Wiley.
32. Padgham, L., & Winikoff, M. (2003). Prometheus: A methodology for developing intelligent agents. In F. Giunchiglia, J. Odell, & G. Weiss (Eds.), *Agent-oriented software engineering III*, volume 2585 of LNCS (pp. 174–185). Berlin/Heidelberg: Springer.
33. Parunak, H. V. D., Brueckner, S., & Sauter, J. A. (2002). Digital pheromone mechanisms for coordination of unmanned vehicles. In *Proceedings of the 1st international joint conference on autonomous agents and multi-agent systems (AAMAS'02)*, Bologna, Italy (pp. 449–450). ACM.
34. Piunti, M., Ricci, A., Braubach, L., & Pokahr, A. (2008). Goal-directed interactions in artifact-based mas: Jadex agents playing in CARTAGO environments. In *Proceedings of the 2008 IEEE/WIC/ACM international conference on Web intelligence and intelligent agent technology (IAT'08)* (Vol. 2). IEEE Computer Society.
35. Platon, E., Mamei, M., Sabouret, N., Honiden, S., & Parunak, H. V. (2007). Mechanisms for environments in multi-agent systems: Survey and opportunities. *Autonomous Agents and Multi-Agent Systems, 14*(1), 31–47.
36. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, & A. E. F. Seghrouchni (Eds.), *Multi-agent programming: Languages, platforms and applications*. Berlin: Springer.
37. Ricci, A., Denti, E., & Piunti, M. (2010). A platform for developing SOA/WS applications as open and heterogeneous multi-agent systems. *Multiagent and Grid Systems International Journal (MAGS)*, *6*(2). Special Issue about "Agents, Web Services and Ontologies: Integrated Methodologies" (to appear).
38. Ricci, A., Omicini, A., & Denti, E. (2003). Activity theory as a framework for MAS coordination. In P. Petta, R. Tolksdorf, & F. Zambonelli (Eds.), *Engineering societies in the agents world III*, volume 2577 of LNCS (pp. 96–110). Berlin/Heidelberg: Springer.
39. Ricci, A., Piunti, M., Acay, L. D., Bordini, R., Hübner, J., & Dastani, M. (2008). Integrating artifact-based environments with heterogeneous agent-programming platforms. In *Proceedings of 7th international conference on agents and multi agents systems (AAMAS08)*.
40. Ricci, A., Piunti, M., & Viroli, M. (2010). Externalisation and internalization: A new perspective on agent modularisation in multi-agent system programming. In M. Dastani, A. El Fallah-Seghrouchni, J. Leite, & P. Torroni (Eds.), *Languages, methodologies and development tools for multi-agent systems*, volume 6039 of LNAI (pp. 36–55). Berlin/Heidelberg: Springer.

41. Ricci, A., Piunti, M., Viroli, M., & Omicini, A. (2009). Environment programming in CArtAgO. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah-Seghrouchni (Eds.), *Multi-agent programming: Languages, platforms and applications*, (Vol. 2, pp. 259–288). Berlin/Heidelberg: Springer.
42. Ricci, A., Viroli, M., & Omicini, A. (2006) *Construenda est* CArtAgO: Toward an infrastructure for artifacts in MAS. In R. Trappl (Ed.), *Cybernetics and systems 2006*, Vienna, Austria, 18–21 April 2006 (Vol. 2, pp. 569–574). 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.
43. Ricci, A., Viroli, M., & Omicini, A. (2007). The A&A programming model & technology for developing agent environments in MAS. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, & M. Winikoff (Eds.), *Programming multi-agent systems*, volume 4908 of LNAI (pp. 91–109). Berlin/Heidelberg: Springer.
44. Ricci, A., Viroli, M., & Omicini, A. (2007). CArtAgO: A framework for prototyping artifact-based environments in MAS. In D. Weyns, H. V. D. Parunak, & F. Michel (Eds.), *Environments for multiagent systems III*, volume 4389 of LNAI (pp. 67–86). Berlin/Heidelberg: Springer.
45. Ricci, A., Viroli, M., & Piunti, M. (2009). Formalising the environment in mas programming: A formal model for artifact-based environments. In L. Braubach, J.-P. Briot, & J. Thangarajah (Eds.), *Programming multi-agent systems*, volume 5919 of LNAI. Berlin/Heidelberg: Springer.
46. Russell, S., & Norvig, P. (2003). *Artificial intelligence, a modern approach* (2nd ed.). Englewood: Prentice Hall.
47. Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-based access control models. *IEEE Computer, 29*(2), 38–47.
48. Valckenaers, P., Hadeli, K., Germain, B. S., Verstraete, P., & Brussel, H. V. (2007). MAS coordination and control based on stigmergy. *Computer Industry, 58*(7), 621–629.
49. Viroli, M., Denti, E., & Ricci, A. (2007). Engineering a BPEL orchestration engine as a multi-agent system. *Science of Computer Programming, 66*(3), 226–245.
50. Viroli, M., Holvoet, T., Ricci, A., Schelfthout, K., & Zambonelli, F. (2007). Infrastructures for the environment of multiagent systems. *Autonomous Agents and Multi-Agent Systems, 14*(1), 49–60.
51. Viroli, M., Ricci, A., & Omicini, A. (2006). Operating instructions for intelligent agent coordination. *The Knowledge Engineering Review, 21*(1), 49–69.
52. Weyns, D., & Holvoet, T. (2004). A formal model for situated multi-agent systems. *Fundamenta Informaticae, 63*(2–3), 125–158.
53. Weyns, D., & Holvoet, T. (2007). A reference architecture for situated multiagent systems. In *Environments for multiagent systems III*, volume 4389 of LNCS (pp. 1–40). Berlin/Heidelberg: Springer.
54. Weyns, D., Omicini, A., & Odell, J. J. (2007). Environment as a first-class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems, 14*(1), 5–30.
55. Weyns, D., & Parunak, H. V. D. (Eds.). (2007). Special issue on environments for multi-agent systems. *Journal of Autonomous Agents and Multi-Agent Systems, 14*(1), 1–116.
56. Weyns, D., Parunak, H. V. D., Michel, F., Holvoet, T., & Ferber, J. (2005). Environments for multiagent systems: State-of-the-art and research challenges. In D. Weyns, H. V. D. Parunak, F. Michel, T. Holvoet, & J. Ferber (Eds.), *Environment for multi-agent systems*, volume 3374 (pp. 1–47). Berlin/Heidelberg: Springer.
57. Wooldridge, M. (2002). *An introduction to multi-agent systems*. Chichester: Wiley.