Programming MAS with Artifacts

Conference Paper *in* Lecture Notes in Computer Science · July 2005

DOI: 10.1007/11678823_13 · Source: DBLP

Alessandro Ricci
University of Bologna
245 PUBLICATIONS **3,679** CITATIONS

SEE PROFILE

Mirko Viroli
University of Bologna
287 PUBLICATIONS **4,147** CITATIONS

SEE PROFILE

Andrea Omicini
University of Bologna
439 PUBLICATIONS **6,086** CITATIONS

SEE PROFILE

Programming MAS with Artifacts

Alessandro Ricci, Mirko Viroli, and Andrea Omicini

DEIS, Alma Mater Studiorum, Università di Bologna
via Venezia 52, 47023 Cesena, Italy
a.ricci@unibo.it, mviroli@deis.unibo.it, andrea.omicini@unibo.it

Abstract. This paper introduces the notion of artifact as a first-class abstraction in MASs (multi-agent systems) and focuses on its impact on MAS programming. Artifacts are runtime devices providing some kind of function or service which agents can fruitfully use – both individually and collectively – to achieve their individual as well as social objectives. Artifacts can be conceived (and programmed) as basic building blocks to model and build agent (working) environments. Besides introducing a conceptual and modelling framework, the paper discusses the impact of this new notion on MAS programming, focussing in particular on MAS composed by cognitive agents. To make the discussion more concrete, we provide an example scenario featuring 3APL agents whose coordination activity is supported by TuCSoN tuple centres – an existing coordination model providing some of the basic properties of artifacts for MASs.

1 Introduction

Research on agent programming has been mainly focused so far on issues concerning *individual agents*, from theories to architectures, and programming languages. In particular, in the research contexts where a notion of strong agency is adopted, this attitude results in facing the basic systemic issues concerning MAS (Multi-Agent Systems) – such as coordination and organisation – mainly from the *subjective* perspective, i.e. exclusively relying on agent computational and communicative abilities. Such an approach has indeed some benefits in terms of uniformity, but has also some strong limits in scaling up with complexity, in particular when coordination activities are concerned [14]. On the one side, programming the glue – even the simple glue – still remains a challenging and complex task. Typically, simple coordination problems result in agents with high complexity, either in terms of the communication protocols or the reasoning capabilities that they must exhibit. On the other side, wrapping (and programming) any kind of useful environmental resource as an agent does not scale up with software systems complexity, in particular in MAS composed of cognitive agents.

A naive observation is that not every entity or abstraction in a MAS is suitably modelled as a goal-governed or goal-oriented system. They can of course be wrapped within an agent, but such a solution is more like a trick than a well-defined engineering choice. This point is simple and old: modelling and programming aspects of a system with abstractions that have not been conceived for this purpose has a dramatically negative impact, in particular as far as the system becomes complex and when the application domain requires forms of dynamic control and evolution.

In this paper we aim at tackling the problem at the foundation level. For this purpose, we introduce the notion of *artifact*, as a first-class abstraction used to design / program / build those aspects of a MAS for which the agent abstraction is not suitable for, i.e. everything that is not suitably modelled as a goal- or task-oriented system. In this paper we will focus in particular on the programming aspect – even though this issue affects every aspect of the agent paradigm, from theories to engineering methodologies.

By abstracting from specific mechanisms, artifacts are meant to be basic building blocks – along with agents – that MAS designers and programmers can design and program to build systems: agents and artifacts are meant to be first-class abstractions from design to runtime, supported by suitable infrastructures.

Generally speaking, artifacts can be used to program and build a suitable agent *working context or environment*: a set of objects (in the wide sense) and tools that agents can share and use to support their individual as well as social activities. Examples range from simple artifacts providing communication functionality, such as mail boxes and blackboards, to artifacts providing coordination services, such as workflow engines or auction-engines, or again artifacts representing general purpose shared resources, such as a shared memories.

Actually, artifacts and tools have been the focus of important theories studying the development of activities in human society. Main examples are Activity Theory and Distributed Cognition [11, 8]. According to such theories, most of the human activities are mediated by some kind of artifacts, and the design and use of such tools play a key role in activities development, heavily influencing their performance and their scalability with problem complexity. Also, the development of human societies itself is strictly related to the development of the tools constructed and used in such societies.

In this paper, we first briefly introduce the conceptual framework characterising the artifact abstraction and the relationships between agents and artifacts – generalising over previous works on coordination artifacts [20, 15] –, and then focus on the impact of using artifacts for programming MASs.

The rest of the paper is organised as follows: first we frame the artifact notion from a conceptual and theoretical point of view, providing a first model as well as some examples of artifacts (Section 2). Then, we introduce some issues related to artifact programming, providing some concrete examples using an existing coordination model – the tuple centre model [13] – which have some of the main properties of artifacts (Section 3). As an important point of the contribution, we consider then the impact of artifacts in agent programming, providing some basic examples using 3APL extended to deal with artifacts (Section 4). Related works (Section 5), conclusions and future works (Section 6) complete the paper.

2 A first theory of artifacts

By considering the conceptual framework described in [1], agents can be generally conceived as *goal-governed* or *goal-oriented* system. Goal-governed systems refer to the strong notion of agency, i.e. agents with some forms of cognitive capabilities, which make it possible to explicitly represent their goals, driving the selection of agents' actions. Goal-oriented systems refer to the weak notion of agency, i.e. agents whose behaviour is directly designed and programmed to achieve some goal, which is not to be explicitly represented. In both goal-governed and goal-oriented systems, goals are *internal. External goals* instead refer to goals which typically belong to the social context or environment where the agents are situated. External goals are sort of regulatory states that condition agent behaviour: a goal-governed system follows external goals by adjusting internal ones [1].

Then, there are systems or parts of a system that are better characterised as resources or tools that are *used* to achieve some goals, having neither internal goals nor a pro-active behaviour, but more simply some kind of functionality that can be suitably exploited, as a service. Here we refer to such basic entities as *artifacts*. Artifacts are non-goal governed / oriented entities explicitly designed to embody and provide a certain function, which is exploited by agents to achieve their individual as well as social goals – in other words, to support the execution of their individual as well as social tasks. By taking the human society as a reference, the distinction between agents and artifacts mirrors the distinctions between humans as autonomous entities and the artificial, non-autonomous tools they exploit everyday in their activities.

So, while the notions of goal and task are central for agents, the notion of *use* and *function*[1] – which is used here, quite roughly, as a synonym of service – are central for artifacts. As for the devices in human society, artifacts are used by means of a basic well-defined set of operations which define artifacts' interface. From a philosophical and conceptual point of view, there is a neat distinction between communication and use: more precisely, agents communicate with other agents but not with artifacts, which are instead *used* though their interface.

As for artifacts in human society, external goals can be attached to an artifact by its users, in spite of its designed function: in this case, the *destination* of the artifact is different from the purposes for which it has been built.

As remarked in Activity Theory and Distributed Cognition, despite their specific function, artifacts are always kind of *mediators* between agents and their objectives, i.e. instruments to transform agent objectives in outcomes. As discussed in next sections, such a mediation has different concrete forms: we can e.g. have mediation of agents interaction, as in the case of *coordination artifacts*, which are shared and used by multiple agents with the purpose of providing some kind of coordination service; or we can have mediation of an agent with respect to its organisational environment, as in the case of *boundary artifacts*,

[1] The term function here refers to a functionality or service, and should not be confused with the term function as used e.g. in functional languages

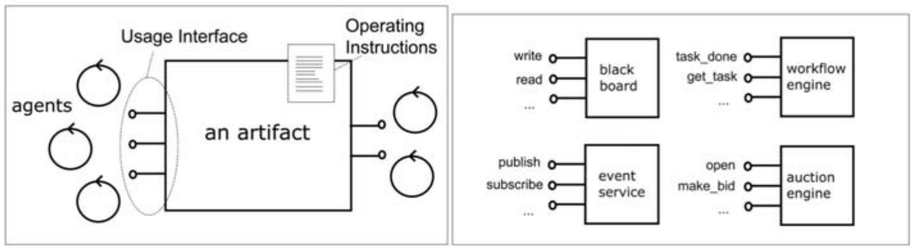

Fig. 1. An abstract representation of an artifact, along with some specific instances

which are used by a single agent with the purpose of constraining its action space according to some organisational rules.

An important distinction characterising agents / artifacts relationships concerns *use* and *use value* [1]. *Use value* corresponds to the evaluation of artifact characteristics and function, in order to *select* it for a (future) *use*. This distinction actually corresponds to two different kinds of external goals attached to an artifact by a user agent: *(i)* the *use value goal*, according to which the artifact should have the power of making its user agent achieve its objective by exploiting the artifact itself – such an external goal drives the agent actions concerning the selection of the artifact –; *(ii)* the *use goal*, which directly corresponds to the agent goal, which drives the usage of the artifact. From the agent point of view, when an artifact is selected and used it has then a *use value* goal which corresponds to its internal goal.

Finally, besides users, artifact *designers* and *programmers* play an important role in the picture, acting as the agents (either artificial or not) with the power of constructing, manipulating, adapting artifact behaviour, either for changing / expanding artifact function or for improving current behaviour without changing its function or interface.

2.1 A model

From the conceptual framework discussed above, we can devise out a first model for the artifact abstraction. An artifact for MAS can be defined as a device – a persistent and stateful runtime entity – designed to provide some kind of function or service, which agents can exploit to achieve their goals. An abstract representation of an artifact is depicted in Fig.1. We identified four basic elements to describe an artifact: the *usage interface*, the *operating instructions*, the *function*, and the *structure and behaviour*.

The *Usage Interface (UI)* is the set of the *operations* which agents can invoke to use the artifact and exploit its functionality. The invocation of an operation – as an agent external action – can result in the occurrence of events at some point(s) in the future, typically bringing some information about the result of the operation. Such events are perceived by the agent as external events (perceptions).

Operating instructions (OI) are a description of *how* to use the artifact to get its functionality. Operating instructions describe the possible *usage protocols*, i.e. sequences of operations that can be invoked on the artifact, in order to exploit its function. Besides a syntactic information, they can embed also some kind of semantic information that rational agents can eventually understand and exploit in their reasoning processes, to enable and promote the cognitive use of the artifact.

The *function* of an artifact is its intended purpose, i.e. the purpose established by the designer / programmer of the artifact, in other words *what* are the intended functionalities the artifact provides.

Finally, the *structure and behaviour* concerns the internal aspects of the artifact, that is how the artifact is implemented in order to provide its function. Such an aspect is typically hidden to users and resides in the domain of artifact designers and programmers.

Differently from agents, artifacts are not meant to be autonomous or exhibit a pro-active behaviour, neither to have social capabilities. Among the main properties, that are useful according to artifacts' purpose and nature, we have: *(i) inspectability and controllability*, i.e. the capability of observing and controlling artifacts structure (state) and behaviour at runtime, and of supporting their online management, in terms of diagnosing, debugging, testing; *(ii) malleability*, i.e. the capability of changing / adapting artifacts function at runtime (on-the-fly) according to new requirements or unpredictable events occurring in the open environment; and *(iii) linkability*, i.e. the capability of linking together at runtime distinct artifacts, for scaling up with complexity of the function to provide and as a mean to support dynamic reuse.

Also, differently from agents, artifacts can have a *spatial extension*, i.e. given a MAS with a topology, the same artifact can cover different nodes: in other words a single artifact can be both conceptually and physically distributed. For instance, a blackboard artifact can cover different Internet nodes, where agents use it by exploiting a local interface.

Given such a conceptual model of artifacts, three main aspects can be identified for characterising their relationships with agents: *(i)* use, *(ii)* selection and *(iii)* construction and manipulation. Such aspects are quite orthogonal, and involve different aspects of the artifacts on the one side, and different kinds of abilities of the agents on the other side. The usage interface and, possibly, the operating instructions are typically the only things an agent needs for using the artifact. Function is important, instead, for selecting what artifacts to use. Finally, construction and manipulation mainly touches the structure and behaviour of the artifacts. In Section 4 these aspects will be connected to different kinds of abilities requested to the agents to exploit artifacts at different levels.

2.2 Examples of artifacts

In order to make the discussion more concrete, we provide some basic examples of artifacts which frequently recur in MAS design and programming, here classified according to their purpose (see Fig.2). It is worth remarking that these examples

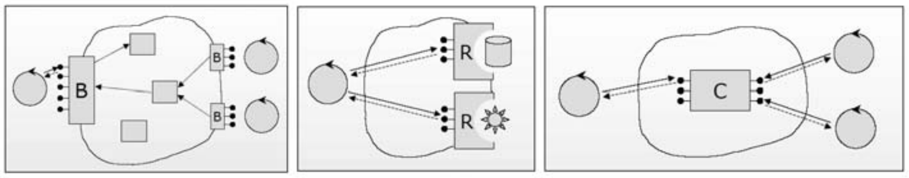

Fig. 2. Some basic types of artifacts: boundary artifacts (B), resource artifact (R) , coordination artifacts (C)

are not meant to be a rigorous taxonomy to partition artifacts: as happen for tools in our society, the same artifact can be classified in different ways according to the point of view. However, the following list is useful for pointing out some basic kinds of artifacts that frequently appear when engineering MAS.

Coordination artifacts – artifacts designed to provide a coordination service. Several mechanisms introduced in other computer science fields – concurrent system and software engineering in particular – and in foreign fields such as management science, can be understood as coordination artifacts. Examples at different levels of abstraction range from artifacts with communication functions (message boxes, blackboards, event services), to artifacts with a specific synchronisation function (schedulers, semaphores), up to general high-level coordination capabilities (workflow engines, auction-engines, normative systems, pheromone infrastructures). The notion of coordination artifact is similar to the concept of *coordination medium* developed in the context of coordination models and languages [18]: however, while in general coordination media have been conceived more for processes in concurrent / distributed systems, coordination artifacts (as kind of artifacts) have some basic features – such as Operating Instructions – that make them suitable for agents as a higher level of abstraction, in particular for goal-governed agents with cognitive capabilities. Also, coordination artifacts have properties which are not generally defined for coordination media, such as malleability, linkability, inspectability, controllability.

Any coordination artifact is a mediator of agent interaction, with both a constructive and normative aim: on the one side it is an enabler of agent interaction, as *the* place where the interaction occurs; on the other side, it constrains the agent interaction space to only the subspace which is correct according to the coordinating function it provides.

In management sciences a set of basic categories concerning coordination problems have been identified [9], classifying them according to the dependencies to be managed and then identifying for each category a set of possible mechanisms useful for this purpose [10]. Such a handbook of coordination knowledge can be ported to MASs, and the corresponding mechanisms implemented as coordination artifacts.

It is worth noting that coordination artifacts represent an *engineered* approach to coordination, which basically works when it is possible and useful to design *a priori* the solution to a coordination problem, and then to reify such a knowledge in suitable artifacts. Conversely, there are cases in which the solution cannot be established a priori by designers, but is either an outcome of agent reasoning, or it *emerges* with agent interaction: in such cases coordination artifacts can be used mainly as interaction enablers. In some cases however, the coordination knowledge acquired during agent interaction can be used to dynamically forge new coordination artifacts, typically to improve the effectiveness and efficiency of the coordination process. This reflects the role of artifacts in reifying the knowledge coming from agents' experience and history.

Boundary artifacts – a particular case of organisation artifacts – with an organisation and security function. They take inspiration from the Agent Coordination Context notion introduced in the context of coordination models and infrastructures [12]. A boundary artifact (BA) is an artifact used to characterise and control the presence (in its most abstract sense) of an agent inside an organisation context, reifying and enacting a *contract* between the agent and the organisation. In role-based environments, a BA embeds the contract for the role(s) the agent plays inside the organisation.

A BA is released to an agent when starting a working session inside an organisation, and then it constraints what the agent can do inside the organisation, in terms of the actions on other artifacts belonging to the the organisation and the communications to other agents. In other words, a BA can be conceived as the embodiment of a (boundary) ruled interface between the agent and the environment.

Resource artifacts – artifacts designed either to mediate access to a specific existing resource or to directly represent and embody a resource part of the MAS environment. An example is a database. This kind of artifact is important to bring at the agent level of abstraction all the computational (and physical) entities which can be useful for agents, from objects (in the OO sense) to services, such as a Web Service.

Currently, wrapper agents are typically used as a solution for this problem: such an approach, however, is useful and conceptually correct when the resource can be suitably and effectively represented and programmed as a goal-oriented or goal-governed system. In all the other cases, resources can be naturally represented as artifacts, as entities providing some kind of service that can be exploited by means of well-defined operations listed in the artifact interface.

It is worth remarking that, from an implementation point of view, artifacts are generally much more light-weight than agents, they more resemble objects (in the OO acceptation): they are typically passive entities managed by the infrastructure, with no structures – for instance – for dealing with task scheduling or reasoning. When engineering complex systems, with many agents and artifacts, this is clearly an issue affecting performance and scalability.

3 Programming Artifacts for MAS

In the following, we consider *tuple centres* as an example of an existing coordination model for MAS exhibiting some of the main features described for (coordination) artifacts. Actually, the design and development of models / infrastructures fully supporting the conceptual framework based on the general notion of artifact is part of our future works (Section 6).

3.1 The tuple centre example

A tuple centre is a programmable tuple space, i.e. a tuple space enhanced with the capability of programming its reacting behaviour to communication events in order to define any kind of coordination laws shaping agent interaction space [13]. TuCSoN is a coordination infrastructure providing tuple centres as run-time coordination services distributed among Internet nodes [16]. In the case of TuCSoN, the communication language adopted is based on logic tuples and the reactive behaviour can be specified as a set of reactions – always encoded as logic tuples – in the ReSpecT language. If the reaction specification is empty, a tuple centre behaves like a tuple space: coordination can be realised by suitably composing the basic coordination primitives to insert, retrieve, and read tuples. By programming the tuple centre with a reaction specification, a specific coordinating behaviour (and then the artifact function) is injected in the tuple centre. The detailed description of tuple centres, ReSpecT and TuCSoN are beyond the scope of the article: interested readers can read to reference articles listed in the bibliography.

So, tuple centres can be framed here as general purpose programmable coordination artifacts, whose coordinating behaviour can be programmed dynamically according to the coordination problem. More precisely, a tuple centre can be framed as a coordination artifact where:

– the usage interface is composed by the coordination primitives to insert (`out`), retrieve (`in`), read (`rd`) tuples, and to inspect (`get_spec`) and set (`set_spec`) tuple centre coordinating behaviour;
– the coordinating behaviour is expressed as a ReSpecT program;
– the operating instructions and the function description are not explicitly supported: they are implicitly described in ReSpecT programs defining specific artifact behaviour.

As for any other artifact operation, invocations are not blocking (the blocking behaviour has no meaning when dealing with artifacts and agents): after invoking an `in` operation on a tuple centre, the invoker agent continues to act according to its plan (which can include of course also waiting for the completion of the operation). When the `in` is satisfied, the operation completes and a completion event is notified to the agent, as a perception.

To exemplify the approach, here we consider a classic coordination problem: the dining philosopher [5]. The problem regards a number of philosophers eating

```
reaction(in(chops(C1,C2)), (pre, out_r(required(C1,C2)))).
reaction(out_r(required(C1,C2)),(
   in_r(chop(C1)),in_r(chop(C2)),out_r(chops(C1,C2)))).
reaction(in(chops(C1,C2)), (post, in_r(required(C1,C2)))).
reaction(out(chops(C1,C2)), (out_r(chop(C1)),out_r(chop(C2)))).
reaction(out(chops(C1,C2)), (in_r(chops(C1,C2)))).
reaction(out_r(chop(C1)), (
   rd_r(required(C1,C)),in_r(chop(C1)),in_r(chop(C)),out_r(chops(C1,C)))).
reaction(out_r(chop(C2)), (
   rd_r(required(C,C2)), in_r(chop(C)),in_r(chop(C2)),out_r(chops(C,C2)))).
```

Table 1. ReSpecT specification for coordinating dining philosophers

at the same round table, sharing chopsticks. Each philosopher alternates thinking with eating. In order to eat, a philosopher needs two chopsticks, which are shared with other two philosophers, sitting one at his left and one at his right. Coordination here is mostly needed to avoid deadlock, which can happen if each philosopher has taken a chopstick and is waiting for the other one, which is in turn taken by a waiting philosopher. In spite of its almost trivial formulation, the dining philosophers problem is generally used as an archetype for non-trivial resource access policies.

A solution to the problem according to our framework consists in using a suitable coordination artifact playing the role of the table, used by the philosopher agents to access the resources (chopsticks). The coordination artifact is here implemented with a tuple centre – called `table` – programmed to provide the coordinating behaviour which avoids deadlock. As an artifact, the table is characterised by:

– a usage interface, composed by the operation `acquireChops(C1,C2)` and `releaseChops(C1,C2)`. Using a tuple centre, the former operation is realised by an `in(chops(C1,C2))`, while the latter with an `out(chops(C1,C2))`;
– a function, informally described as *to dine*, which matches the the dining goal of the philosopher agent;
– operating instructions, which can be informally described as follows: "let `C1` and `C2` be the chopsticks you need, then first invoke `acquireChops(C1,C2)` operation. When the operation is completed, dining task can be scheduled. When the dining task finished, invoke `release(C1,C2)` operation". Such an informal description can be described more rigorously adopting a formal framework based on operational semantics, as discussed in [21].
– a coordinating behaviour to avoid deadlock. Using a tuple centre, the behaviour is provided by the ReSpecT specification described in Table 1 (for details concerning how the specification works refer to [13]).

Philosopher agents can be realised in any programming language: in Section 4 we show an implementation using 3APL. Basically, the philosopher agents' goal is to survive, interleaving thinking and dining behaviour. For the latter one, following the operating instructions, they need to get the chopsticks from the table and to give them back when dining has finished.

The main point here is that philosophers do not need to worry about how to coordinate themselves, or how the resources are represented: they simply need to know which chopstick pair to ask for, and then they can focus on their main tasks (thinking and eating).

4 Impact on Agent Programming and Reasoning

An important issue of our approach concerns how artifacts could be effectively exploited to improve agents' ability to execute individual as well as social tasks. Which reasoning models could be adopted by agents to use artifacts in the best way, simplifying their job? How could operating instructions be used in agent reasoning processes, in order to help them using artifacts and finally achieving their goal(s)? Or rather: how could an agent reason to select which artifacts to use? How could artifact function description be exploited for this purpose? And finally: how could agents reason to construct or adapt artifacts behaviour in order to be useful for their goals? All the above questions are strictly related to some of the main *foci* in the research in service-oriented (agent-based) architectures, i.e the description and discovery / brokerage of artifacts (services).

On the one side, the simplest case concerns agents directly programmed to use specific artifacts, with usage protocols directly defined by the programmer either as part of the procedural knowledge / plans of the agent for goal-governed systems, or as part of agent behaviour in goal-oriented systems. In spite of its simplicity, this case can bring several advantages for MAS engineers, exploiting separation of concerns when programming light-weight agents, without the burden – e.g. coordination burden – which is instead upon artifacts designed for this purpose. On the other side, in the case of fully open systems, the intuition is that operating instructions and function description can be the key for building MAS where intelligent agents dynamically look for and select which artifacts to use, and then exploit them accordingly, simplifying the reasoning required to achieve the goals with respect to the case in which artifacts are not available.

Actually, the conceptual framework discussed in Section 2 makes it possible to frame such abilities progressively, scaling with the openness and complexity of the domain context. Some levels can be identified, involving different kinds of artifact aspects and agents' abilities:

- *unaware use* – at this level, agents and agent programmers exploit artifacts without being aware of them. In other words, agents' actions never refer explicitly to the execution of operations on some kinds of artifacts.
- *programmed use* – at this level agents use some artifacts according to what has been explicitly programmed by the developer. In the case of cognitive agents, for instance, agent programmers can specify usage protocols directly as part of the agent plan. For the agent point of view, there is no need to understand explicitly artifacts' operating instructions or function: the only requirement is that the agent model adopted could be expressive enough to model in some way the execution of external actions and the perception of external events.

- *cognitive use* – at this level, the agent programmer directly specifies in the agent program some knowledge about what artifacts to use. However, how to exploit the artifacts is dynamically discovered by the agent, by reading the operating instructions. So, generally speaking the agent must be able to embed the procedural knowledge given by the operating instructions in the procedural knowledge defined in its plans. In this case the adoption of shared ontologies for operating instructions description / goal description is necessary.

  Focussing on this point, an interesting note comes from the studies on human behaviour using artifacts. According to Activity Theory, a hierarchy can be identified among activities, actions, and operations:
  - *Operations* – Operations are defined as routinised (interactive) behaviour of individuals, that require little conscious attention (e.g. rapid typing). Responsive of actual conditions, operations provide an adjustment of actions to current situations;
  - *Actions* – Actions are defined as behaviour that is characterised by conscious planning. There may be many different operations capable of fulfilling an action. Actions are directed toward *goals*, which are the objects of actions. Usually, goals are functionally subordinated to other goals, which may still subordinated to other goals and so forth. Actions must be understood within the frame of reference created by the activity;
  - *Activity* – Activity can be defined as the minimum meaningful context for understanding individual actions. An activity is directed toward a *motive*, which is the object which motivates the whole activity.

  Such a remark can be useful in our case for exploring two different ways to use an artifact:
  - *Conscious* – in this case any interaction with the artifact is under the direct control of the main reasoning process of the agent (e.g. main deliberation cycle), where the operating instructions have been embedded;
  - *Unconscious* – in this case the interaction with the artifact is not governed by the deliberation cycle of the agent, but realised by some automated procedure which executes directly – on the background of agent main reasoning – the operating instructions. Only in the case of a breakdown, the reasoning focus of the agent is shifted on the interaction with the artifact, by properly reacting to perceptions which represent the problems.

  The last case can be very interesting in order to devise out agents that very efficiently exploit artifacts in the background, while keeping the reasoning focus on other issues;
- *cognitive selection and use* – this case extends the previous one by conceiving agents that autonomously select artifacts to use, get operating instructions and use them. With respect to the previous case, agents must be able both to understand and embed the operating instructions, and also understand artifacts function / service description, in order to possibly decide to use the artifacts for their own goal(s). It is worth noting that such a selection process can concern also set of cooperative agents, interested in using a

```
PROGRAM "philosopher"

CAPABILITIES:
  { not hungry } think { hungry },
  { hungry } eat {not hungry },
  { not holding_chops } update_chop_belief(acquired) { holding_chops },
  { holding_chops } update_chop_belief(released) { not holding_chops },
  { left_chop(C1),right_chop(C2) } invoke_op(in(chops(C1,C2)),table) {},
  { left_chop(C1),right_chop(C2) } invoke_op(out(chops(C1,C2)),table) {}

BELIEFBASE:
  left_chop(...),
  right_chop(...)

GOALBASE:
  survive()

RULEBASE
  survive() <- not hungry | think,
  survive() <- hungry | dine,
  dine() <- not holding_chops | invoke_op(in(chops(C1,C2)),table),
  dine() <- holding_chops | eat ; invoke_op(out(chops(C1,C2)),table),
  <- op_completed(in(chops(C1,C2)),table,_) | update_chop_belief(acquired),
  <- op_completed(out(chops(C1,C2)),table,_) | update_chop_belief(released)
```

Table 2. A dining philosopher implemented in 3APL, using the tuple centre `table` as a coordination artifact

coordination artifact for their social activities. As in the previous case, shared ontologies are necessary, in this case both for operating instructions and function description;

– *construction and manipulation* – in this case the point of view is changed, considering agents playing the role of programmers of the artifacts. At this level agents are supposed to understand how artifacts work, and to adapt their behaviour (or build new ones from scratch) in order to make it more effective or efficient for other agents' goals. For its complexity, this level generally concerns humans. However, agents can e.g. be adopted to change artifact behaviour according to schema explicitly defined by the agent programmer.

4.1 An example using 3APL

In order to help the reader's intuition, in the following we describe a first example of MAS composed by a set of cognitive agents using a tuple centre as a simple kind of coordination artifact. Agents are implemented in 3APL [3], which is taken here as a reference example of agent-oriented programming language for goal-governed agents.

Actually, the basic 3APL model is extended to support the artifact framework. In particular the extension introduces external actions and perceptions (external events), as in the case of dMARS [6]. The extension is a generalisation of the work described in [4], where 3APL is extended to support communicative actions to send and receive FIPA ACL message, and to react to external events concerning the reception of messages. There, the authors define a message base as a new part of a 3APL agent state: communicative actions and external events alter the content of the message base. Practical rules with a guard are introduced for reacting to the presence in the message base of events related to the arrival of new messages.

Our extension consists first in modelling the execution of an operation on a specific artifact as a 3APL (external) action. For this purpose, we extend the set of possible 3APL goals with the action

`invoke_op(`*`O`*`,`*`A`*`)`

where *`O`* is a term representing the signature of the operation to be invoked, and *`A`* is a term used as identifier of the artifact. As an example, the action `invoke_op(get_token, synchroniser)` invokes the `get_token` operation on the `synchroniser` artifact. Another case is action `invoke_op(in(age('Bob',X)), dbase)`, which invokes the `in` operation on the tuple centre `dbase` in order to retrieve a tuple matching the template `age('Bob',X)`.

Second, the extension also models the perception of events generated by artifacts. To this end, the practical rule on message reception is generalised to consider also external events concerning the completion of an operation executed on an artifact. A new guard is introduced:

`op_completed(`*`O`*`,`*`A`*`,`*`R`*`)`

where *`O`* represents the signature of an operation previously invoked, *`A`* the source artifact, and *`R`* a result term carrying information related to the completion of the operation. An examples of rule is:

```
  <- op_completed(get_token, synchroniser, _) | do_critical_task()
```

This practical rule executes the goal `do_critical_task()` when the completion of the operation `acquire_lock` is perceived. The following rule executes the goal `update_info` when the `in` operation completes, retrieving a tuple from the tuple centre `dbase`:

```
<- op_completed(in(age('Bob',X)),dbase?in(age(_,Y)))|update_info(Y)
```

As an application example, we consider a solution to the dining philosopher problem, using 3APL agents as philosophers and exploiting the table as a coordination artifact. This is a a case of programmed use of artifacts, since the knowledge about how to use the artifact is directly encoded by the agent programmer among the practical rules of the agent. As a coordination artifact, we consider the tuple centre described in Section 3: in the overall we build up a solution with 3APL agents exploiting a TuCSoN tuple centre. The source code of the 3APL philosopher is shown in Table 2.

The agent goal is to survive. The plan to survive is described in the rule base, and involves thinking and dining activities. If the philosopher is not hungry, he can think: thinking activity is simplified into a simple action in the capabilities (line 4), whose effect is to make the philosopher hungry (`hungry` is inserted in the belief base). If the philosopher is hungry, then he plans to dine (line 20). In order to dine, the philosopher needs to have the chopsticks. If he believes to hold them (`holding_chops` is his belief base, line 22), then he can start the eating activity, again simplified into a simple action (line 5), whose effect is to make the agent not hungry. Instead, if the philosopher believes not to hold the chopsticks, then he interacts with the artifact table to get the chopsticks. In particular, he executes an external action to invoke an `in` operation on the tuple centre `table` to get a tuple `chops(C1,C2)` representing the chopsticks (line 21, 8). The information about the specific chopsticks to request are stored in the belief base in the form of the `left_chop` and `right_chop` beliefs. When the philosopher perceives the completion of the operation to get the chopsticks (line 23), the belief base is updated by means of an internal action asserting the `holding_chops` fact (line 6). Then, the plan of the agent is to release the chopsticks after eating. For this purpose an external action is executed (line 22, 9), which invokes an `out` operation on the same tuple centre, inserting back the tuple `chops(C1,C2)`. When the philosopher perceives that the operation to release the chopsticks has completed (line 24), the belief base is updated by means of an internal action asserting the `not holding_chops`.

5 Related Work

This work generalises and extends previous works on coordination artifacts [15].

The artifact abstraction brings in MAS ideas and concepts that have played a central role in other (un)related fields. From concurrent and distributed systems, coordination artifacts in particular can be considered the generalisation of traditional coordination abstractions, from low level ones such as semaphores, monitors, to high-level ones, such as tuple spaces and, more generally, coordination media as found in coordination models and languages [17]. Blackboards as defined in Distributed Artificial Intelligence context can be framed and modelled in MAS as coordination artifacts, toward the integration of the two different points of view (traditional multi-agent and blackboard systems) in designing collaborating-software engineering space [2].

Actually, artifacts can be exploited as an analytical tool for describing existing approaches based on some form of mediated / environment-based interaction. For instance, the environment provided by the pheromone infrastructure in [19] supporting stigmergy coordination can be interpreted as a coordination artifact exploited by ants to coordinate: as such, it provides operations for depositing and sensing pheromones, and the coordinating behaviour is given by the environmental laws ruling the diffusion, aggregation and evaporation of pheromones.

Also some coordination and organisation approaches developed in the context of intelligent / cognitive agents can be framed in terms of artifacts. A main example is is given by electronic institutions ([7] is an example), where agent societies live upon an infrastructure (middleware) which governs agent interaction according to the norms established for the specific organisation, representing both organisation and coordination rules. The institution then can be framed as a kind of shared artifact, characterised by an interface with operations that agents use to communicate, and providing a normative function on the overall set of agents.

6 Conclusion and Future works

In the paper we introduced the notion of artifact as first-class abstraction for MAS engineering. Artifacts are meant to be used as basic bricks to program MAS working environments, supporting agents in their individual and social activities. After providing some glances about artifact programming, in the paper we focused on the impact on agent programming, framing some levels related to artifact adoption.

Several directions characterise future works. An important one is devoted to deepen the investigation on how the artifact abstraction and its basic properties can be effective in supporting agent reasoning in achieving individual as well as collective goals. Another direction concerns the development of infrastructures and tools fully supporting the artifact abstraction and the basic kind of artifacts discussed in the paper, in particular integrating such infrastructures with existing MAS platforms for cognitive agents (3APL is an example). In particular, as in the case of service-oriented architectures, the infrastructure should provide services that agents can exploit for registering, discovering, locating artifacts, for retrieving their description and operating instructions (for agent using artifacts) and for their inspection and control (for human and agents managing artifacts). For this purpose, existing research literature on service description and discovery / brokerage will be considered among the reference sources.

Finally, our intuition is that the separation of concerns obtained by introducing artifacts could be important to make more tractable the verification / validation of formal properties of (open) MAS; accordingly, research studies will be devoted to define formal frameworks to specify artifacts function / behaviour semantics, and to explore how to use them for verification problems, both offline and on-line.

References

1. R. Conte and C. Castelfranchi, editors. *Cognitive and Social Action.* University College London, 1995.
2. D. D. Corkill. Collaborating software: Blackboard and multi-agent systems & the future. In *International Lisp Conference*, 2003.
3. M. Dastani, F. de Boer, F. Dignum, and J.-J. Meyer. Programming agent deliberation: an approach illustrated using the 3apl language. In *Proceedings of AAMAS '03*, pages 97–104. ACM Press, 2003.
4. M. Dastani, J. van der Ham, and F. Dignum. Communication for goal directed agents. In M.-P. Huget, editor, *Communication in Multiagent Systems, Agent*

Communication Languages and Conversation Polocies., volume 2650 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 2003.

5. E. Dijkstra. *Co-operating Sequential Processes.* Academic Press, London, 1965.
6. M. d'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge. The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 1:5–53, 2004.
7. M. Esteva, B. Rosell, J. A. Rodríguez-Aguilar, and J. L. Arcos. Ameli: An agent-based middleware for electronic institutions. In *Proceedings of AAMAS '04*, volume 1, pages 236–243, New York, USA, 19–23 July 2004. ACM.
8. D. Kirsh. Distributed cognition, coordination and environment design. In *Proceedings of the European conference on Cognitive Science*, pages 1–11, 1999.
9. T. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.
10. T. W. Malone, K. Crowston, J. Lee, B. Pentland, C. Dellarocas, G. Wyner, J. Quimby, C. S. Osborn, A. Bernstein, G. Herman, M. Klein, and E. O'Donnell. Tools for inventing organizations: Toward a handbook of organizational processes. *Management Science*, 45(3):425–443, 1999.
11. B. Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction.* MIT Press, 1996.
12. A. Omicini. Towards a notion of agent coordination context. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 187–200. CRC Press, 2002.
13. A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.
14. A. Omicini and S. Ossowski. Objective versus subjective coordination in the engineering of agent systems. In M. Klusch, S. Bergamaschi, P. Edwards, and P. Petta, editors, *Intelligent Information Agents: An AgentLink Perspective*, volume 2586 of *LNAI: State-of-the-Art Survey*, pages 179–202. Springer-Verlag, Mar. 2003.
15. A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, and L. Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *Proceedings of AAMAS '04*, volume 1, pages 286–293, New York, USA, 19–23 July 2004. ACM.
16. A. Omicini and F. Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, Sept. 1999. Special Issue: Coordination Mechanisms for Web Agents.
17. G. A. Papadopoulos. Models and technologies for the coordination of Internet agents: A survey. In *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 2, pages 25–56. Springer-Verlag, Mar. 2001.
18. G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998.
19. H. V. D. Parunak, S. Brueckner, and J. Sauter. Digital pheromone mechanisms for coordination of unmanned vehicles. In *Proceedings of AAMAS '02*, pages 449–450. ACM Press, 2002.
20. A. Ricci, A. Omicini, and E. Denti. Activity Theory as a framework for MAS coordination. In *Engineering Societies in the Agents World III*, volume 2577 of *LNCS*, pages 96–110. Springer-Verlag, Apr. 2003.
21. M. Viroli and A. Ricci. Instructions-based semantics of agent mediated interaction. In *Proceedings of AAMAS '04*, volume 1, pages 102–109, New York, USA, 19–23 July 2004. ACM.