Integrating Heterogeneous Agent Programming Platforms within Artifact-Based Environments

Alessandro Ricci
DEIS
Università di Bologna
Via Venezia 52
47023 Cesena (FC), Italy
a.ricci@unibo.it

Michele Piunti
ISTC-CNR, Roma
DEIS, Univ. di Bologna
michele.piunti@istc.cnr.it

L. Daghan Acay
DIS
The University of Melbourne
111 Barry Street Victoria
3010, Australia
lacay@pgrad.unimelb.edu.au

Rafael H. Bordini
Dept. of Computer Science
University of Durham
Durham DH1 3LE, UK
R.Bordini@durham.ac.uk

Jomi F. Hübner
ENS Mines Saint-Etienne
158 Cours Fauriel
42023 Saint-Etienne, France
Jomi.Hubner@emse.fr

Mehdi Dastani
Intelligent Systems Group
Utrecht University
3508 TB Utrecht, Netherlands
mehdi@cs.uu.nl

ABSTRACT

"Agents and Artifacts" (A&A) and CARTAGO are becoming increasingly popular as, respectively, a general-purpose programming model and a related infrastructure for developing shared computational environments in agent-based software systems. However, so far there has been no work on developing multiagent systems (MAS) where agents implemented and deployed in different agent-programming platforms can interact as part of the same MAS with a shared environment. Due to the generality of CARTAGO environments and its Java-based implementation, we have successfully implemented an *open* multi-agent system where heterogeneous agents developed with different platforms—namely *Jason*, 2APL as BDI-based approaches and simpA as an activity-oriented approach rather than BDI-based—work together in shared *workspaces* where they interact and cooperate by dynamically creating and using shared *artifacts*, analogously to human working environments. This paper shows how this was achieved by first presenting a general model for incorporating a theory of use and observation of artifacts in cognitive agents, then describing a general approach for developing such heterogeneous MAS using CARTAGO integrated with existing agent-oriented programming platforms.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems; D.1.m [**Programming Techniques**]: Miscellaneous; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Languages, Theory

Keywords

Artifact-based computational environments, Multi-agent systems platforms, CARTAGO

1. INTRODUCTION

Recent papers appearing in the agent literature have highlighted the important role that the notion of *environment* can play in designing and engineering Multi-Agent Systems (MAS), being a suitable locus to encapsulate services and functionalities in order to improve agent interaction, coordination, and cooperation (see [12] for a survey). The "Agents and Artifacts" (A&A) meta-model in particular has been proposed as a general-purpose approach to conceive computational environments in MAS, modelled as *workspaces* where agents can share, use, and manipulate *artifacts*. Artifacts are first-class entities in such design approaches, modelling tools and resources that agents can exploit to support their individual and social activities [8]. From a technological point of view, CARTAGO is an infrastructure for MAS supporting the creation and execution of artifact-based environments [8].

In this paper, we describe the integration of A&A and CARTAGO with existing agent-oriented programming languages and platforms, both cognitive/BDI—taking *Jason* [2, 1] and 2APL [4] as main representative examples—and non-cognitive/activity-oriented— here we consider the simpA platform [9]—enabling heterogeneous agents belonging to different platforms to participate in the same work environments, and then interact and cooperate by means of instantiating, sharing, and using artifacts. In fact, current BDI agent-oriented programming platforms recognise the benefits of providing some kind of support to define MAS computational environments. However, the provided support typically lacks clear conceptual foundations, and is typically realised by means of an API which enables the interaction between agents and entities belonging to a different level of abstraction with respect to agents, such as Java objects, without clear principles. Thanks to the integration described in this paper, some of the best-known agent platforms are extended with a general-purpose, high-level approach to model and implement those parts of the MAS which are not agents, such as passive and function-/service-oriented entities. Such entities are useful for building clear, sophisticated, *open* work environments (to be shared by multiple agents) which are built up by resources and tools that

are naturally part of the MAS, possibly wrapping resources developed in other languages (such as Java) or interfaces to the external MAS environment.

The remainder of the paper is organised as follows: Section 2 briefly provides background on A&A, including the notion of artifacts and their use as an abstraction, and on the CARTAGO infrastructure. Section 3 describes a general model of integration with existing agent-oriented programming models, both cognitive (BDI in particular) and non-cognitive agent programming platforms. Section 4 highlights some practical benefits of artifact use, describing a simple scenario, designed with CARTAGO, where heterogeneous agents (i.e. *Jason*, 2APL, simpA) can fully interact in an open system, where shared artifacts can be used for cooperation and to facilitate achieving goals. Finally, Section 5 concludes the paper, discussing future work and providing some final remarks.

2. BACKGROUND: THE A&A MODEL

A&A aims at introducing the notion of artifacts as first-class abstraction along with agents when modelling, designing, and developing MAS [8]. The main inspiration of A&A comes from Activity Theory [6], a psychological conceptual approach started in the Soviet Union at the beginning of the 20th century, and more recently was further developed, in northern Europe in particular, remarking the power and importance of culture and *artifacts* to enhance human abilities. One of the main concepts put forward by Activity Theory—along with Distributed Cognition and other movements within cognitive science—is that, in human societies, properly designed artifacts and tools play a fundamental (mediation) role in coping with the scaling up of complexity in human activities, in particular when social activities are concerned. As remarked in [7], tools (artifacts) shape the way human beings interact with reality, being designed for simplifying the execution of tasks, improving problem-solving capabilities, and for enabling efficient coordination and cooperation in social contexts. In fact, the concept of tool in Activity Theory is broad and embraces both technical tools, which are intended to manipulate physical objects (e.g., a hammer), and psychological tools, which are used by human beings to influence other people or themselves (e.g., the multiplication table or a calendar).

By analogy, the basic idea of A&A is to define a notion of *artifacts*, also in MAS, as basic building blocks to design and engineer those parts of the MAS that we call here *work environment*. Such building blocks can be programmed by MAS designers and dynamically instantiated, discovered, shared, used and manipulated by agents while performing their tasks. In other words, artifacts represent the *functional part* of the MAS environment, while agents represent the *goal-* or *task*-oriented part. As noted by Norman [7] on the relationship between (cognitive) artifacts and humans, artifacts change the way a task gets done. In particular, artifacts can: (*i*) distribute the actions across time (precomputation); (*ii*) distribute the actions across agents (distributed cognition); (*iii*) change the actions required of the individuals doing the activity. So, adopting an A&A approach to design a MAS means to design and program agents, as well as computational work environments in terms of specific types of artifacts, to help in agent activities. Different kinds of functionalities can be encapsulated in suitably programmed artifacts: communication and coordination functionalities (e.g., a blackboard, a map), purely synchronising functionalities (e.g., a clock, a calendar, a workflow engine), resource access functionalities (e.g., a database, a wrapper for a web-service) and so on. Then, by generalising existing indirect communication models —tuple spaces, as a main example— artifacts extend agent communication and cooperation, besides direct language-based communication, towards forms of mediated interaction (with artifacts functioning as mediators).

Besides artifacts, the notion of *workspace* plays an important role in A&A. A workspace is a logic container of agents and artifacts, and can be used to structure the overall sets of entities, defining a topology of the work environment and providing a way to frame and rule the interaction within it. More specifically, the notion of workspace has a twofold purpose. First, it is useful to explicitly define the *distribution model* of the MAS. A complex MAS can be organised in a dynamic set of workspaces, typically distributed among multiple nodes of the network, with agents possibly joining in and working simultaneously at multiple workspaces. Second, a workspace is meant to be the conceptual locus where to define the set of security and organisational rules constraining agent access to the workspace and to artifacts belonging to it. For this purpose, a Role-Based Access Control model (RBAC) is adopted [10]. In particular, for each workspace a dynamic set of roles and related control policies ruling artifact usage can be defined and changed at runtime, both by human and agent administrators, the latter by using specific pre-defined artifacts. This makes it possible to develop truly open systems on the one hand, being the set of agents, roles, and policies fully dynamic, and to avoid related security problems on the other hand, by means of access control policies enforcement.

2.1 A Programming Model for Artifacts

The development model introduced with A&A and applied in related technologies such as CARTAGO aims at both capturing the *function-oriented* nature of artifacts and the notion of artifact *use*, as well as being sufficiently general-purpose to be used in programming any type of artifact that might be useful in MAS applications. An artifact is essentially a passive, dynamic, stateful entity, designed to encapsulate and provide some sort of function. The functionality of an artifact is structured in terms of *operations*, whose execution can be triggered by agents through an artifact's *usage interface*. Analogously to usage interface of artifacts in the real world (think, for example, of a coffee machine), an artifact usage interface in A&A is composed of a set of *operation controls* that agents can use to trigger and control operation execution (such as the control panel of a coffee machine). Each operation control is identified by a label (typically equal to the operation name to be triggered) and a list of input parameters. The usage interface can change dynamically, according to state of the artifact; in other words, it is possible to design artifacts that expose a different usage interface according to their functioning stage. Besides the operation control, the usage interface might contain also a set of *observable properties* (think of the coffee machine display); that is, properties whose dynamic values can be observed by agents without necessarily interacting with (or operating upon) the artifact.

An *operation* is the basic unit upon which artifact func-

tionality is structured. The execution of an operation upon an artifact can result both in changes in the artifact's inner (i.e., non-observable) state, and in the generation of a stream of *observable events* that can be perceived by agents that are using or simply observing the artifact. It is worth remarking here the differences between observable properties and observable events. The former are (dynamic, persistent) attributes that belong to an artifact and that can be observed by agents without interacting with it (i.e., without using the operation controls)—for example, the display of a coffee machine. The latter are non-persistent information, as signals carrying also an information content—as the sound emitted by a coffee machine when the coffee is ready, for example.

Operation execution can be conceived as a process (from a conceptual point of view) combining the execution of possibly multiple guarded operation steps, where guards relate to the inner artifact state. In order to avoid interferences, the execution of a single operation step is *atomic*. This approach, overall, makes it possible to support the execution of multiple operations concurrently within the artifact, maintaining mutual exclusion to access the artifact state.

Analogously to artifacts in the human case, in A&A each artifact is meant to be equipped with a "manual" describing the artifact's function (i.e., its intended purpose), the artifact's usage interface (i.e., the observable "shape" of the artifact), and the artifact's *operating instructions* (i.e., usage protocols or simply how to correctly use the artifact so as to take advantage of all its functionalities). An artifact manual is meant to be inspected and used at runtime by agents, in particular intelligent agents, for reasoning about how to select and use artifacts so as to best achieve their goals. This is a fundamental feature for developing open systems, where agents cannot have *a priori* knowledge of all the artifacts available in their workspaces since new instances and types of artifacts can be created dynamically, at runtime. Currently, no commitments towards specific models, ontologies, and technologies to be adopted for manual description have as yet been made, as this is part of ongoing work.

Finally, as a principle of composition, artifacts can be linked together, in order to enable artifact–artifact interaction. This is realised through *link interfaces*, which are analogous to interfaces of artifacts in the real world (e.g., linking/connecting/plugging the earphones into an MP3 player, or using a remote control for the TV). Linking is supported also for artifacts belonging to distinct workspaces, possibly residing on different network nodes.

The programming model of A&A includes also a set of predefined artifacts available by default in each workspace, providing essential functionalities for managing artifacts and workspaces themselves. Among others, we mention here: a *factory* artifact, providing basic functionalities for dynamically creating and disposing of artifacts; a *registry* artifact, for looking up and discovering the set of artifacts currently available in the workspace; a *security-registry* artifact, for managing the set of roles and related access control policies currently defined in the workspace.

2.2 The CARTAGO Infrastructure

CARTAGO (Common ARtifact infrastructure for AGent Open environment) is a platform/infrastructure for programming and executing artifact-based work environments for MAS [8]. It provides: (*i*) an API for defining new types of artifacts, using the A&A programming model; (*ii*) an API to be used in agent-oriented programming platforms as an interface to interact with CARTAGO environments; (*iii*) a runtime environment, supporting the execution of possibly distributed work environments, by managing workspace and artifact lifecycles; (*iv*) a library with a predefined set of general-purpose artifacts, providing different kinds of functionalities, such as coordination functionalities (blackboards, message boxes, tuple spaces, tuple centers, etc.), GUI functionalities (GUI frames and components can be modelled in CARTAGO as artifacts, mediating human–agent interactions), resources (databases, legacy system wrappers, etc.), and so on. CARTAGO[1] technology is *open source* and completely based on Java.

3. INTEGRATION WITH AGENT PROGRAMMING PLATFORMS

The main contribution of this work is the definition and implementation of a general model for integrating A&A and CARTAGO with existing cognitive agent-oriented platforms so as to extend their programming model for MAS in order to have a uniform high-level general purpose programming model for: (*i*) implementing MAS environments in general, enabling, mediating, and controlling the access, for instance, to a real physical or computational environment; and (*ii*) designing and programming suitable work environment for agents, with artifacts and tools useful to organise complex applications. An important result of the integration is introducing a new simple way of enabling basic interoperability among agents of different platforms, by creating distributed applications with agents from different platforms participating in the same workspaces and interacting with the same artifacts.

In order to make it easier to define such integration, in spite of the specific agent model (BDI or not) considered, we introduce the notion of an *agent body* as that part of an agent which conceptually belongs to a workspace, containing effectors to act upon artifacts and sensors to perceive artifacts' observable events. Sensors in particular play here a fundamental role: that of *perceptual memory*, whose functionality accounts for keeping track of stimuli arrived from the environment, possibly applying filters and specific kinds of "buffering" policy. According to the specific interaction modality adopted for using and observing artifacts, as described later in this section, it might be useful to provide agents with basic internal actions for managing and inspecting sensors, as a kind of private memory. In particular, it could be useful for an agent to organise in a flexible way the perception of observable events, possibly generated by multiple different artifacts that an agent can be using for different, even concurrent, activities.

The notion of agent body makes it easier to realise the integration also from an engineering point of view, by providing a clear way to separate the part of an agent governed by a cognitive agent platform (which we could call the "agent mind") and the part that is managed by CARTAGO (i.e., the agent body, situated in workspaces); from a development point of view, the integration accounts for defining a suitable low-level interface enabling the mind to control the body and perceive stimuli collected by body's sensors.

---

[1]Available at `http://www.alice.unibo.it/cartago`.

```
(1)  lookupWsp(WName,?Wid,+Node)
(2)  joinWsp(Wid)
(3)  quitWsp(Wid)
(4)  createWsp(WName,+Node)
(5)  removeWsp(WName,+Node)
(6)  lookupArtifact(AName,?Aid)
(7)  use(Aid,OpCntrlName(Params),+Timeout,+Filter)
(8)  use(Aid,OpCntrlName(Params),Sid,+Timeout,+Filter)
(9)  sense(Sid,?Perception,+Filter,+Timeout)
(10) makeArtifact(Aid,+ATemplate,+AConfig)
(11) disposeArtifact(Aid)
(12) focus(Aid,+Filter)
(13) focus(Aid,Sid,+Filter)
(14) stopFocussing(Aid)
(15) observeProperty(Aid,PFilter,?AProperty)
```

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

3.1 Incorporating a Theory of Use and Observation of Artifacts into Cognitive Agents

Realising the integration aimed in this work means, first of all, incorporating a theory of use and observation of artifacts into cognitive agents. At the base (enabling) level, such a theory is based on a new set of basic actions that makes it possible for an agent to: (*i*) dynamically create, join, leave, remove workspaces; (*ii*) use an artifact, by acting on its usage interface and perceive observable events generated by artifacts; (*iii*) observe an artifact. Table 1 provides a synthetic view of the set of actions, grouped into three main groups. As for the syntax, a pseudo-code first-order logic-like syntax is adopted, with the semantics described informally. Following the semantics adopted in the cognitive agent-oriented programming approaches considered here, an action consists in the atomic execution of a statement which can result in changing the agent's state and/or interacting with the agent's environment, and can succeed or fail.

The first group (labelled 1–5) is composed by actions for managing workspaces, and starting and finishing a working session inside a workspace. Intuitively, `lookupWsp` obtains a workspace unique identifier given its name and possibly its location; `joinWsp` makes it possible to "enter" logically a workspace, whose identifier is specified as a parameter; `quitWsp` to leave a workspace; `createWsp` and `removeWsp` to respectively create a new workspace, specifying its name and location, and remove an existing one, specifying its identifier.

The second group of actions (labelled 6–11) concerns the *use* of artifacts. Two basic interaction modalities are supported, relating to the ways in which observable events generated by artifacts are perceived and processed by agents. In the first modality, events generated by an artifact are made observable to agents as new beliefs about the occurrence of the event, without the mediation of sensors. The `use` action labelled (7) supports this modality. The action accounts for using the artifact identified by `Aid`, by acting on the usage-interface operation control (`OpCntrlName`), specifying some parameters (`Params`), and optionally specifying a filter (`Filter`) and a timeout (`Timeout`). The action succeeds if the specified artifact exists and its usage interface actually has the specified control, and as a result the related operation is triggered for execution. Then, every observable event subsequently generated by the artifact, as effect of the operation execution, is made observable to the agent as a new belief `artifact_event(Aid,Event)` in the agent's belief base. The filter can be used to specify which types of events the agent is interested in perceiving. If the usage interface of the artifact is disabled when executing the action, for instance because the artifact is executing an operation (step), then the agent action is suspended until the usage interface is enabled again; the timeout specifies how long the agent can wait before considering the action as failed.

In the second modality, events generated by an artifact do not cause the direct creation of new beliefs, but are collected instead in agent sensors. The `use` action labelled (8) supports this modality. It has the same semantics of the previous one, with the difference that the agent specifies the identifier `Sid` of the sensor to be used to collect the events generated by the artifact. In this case, the belief base of the agent *is not* updated. However, in this modality, a `sense` action is provided (9) to inspect the content of a sensor (i.e., the perceptual memory), so that the agent can become aware of any new percepts (hence updating the belief base). In particular, the action succeeds if within `Timeout` time an event (stimulus) matching the specified `Filter` is found in the specified sensor `Sid`. In that case, `Perception` is unified with such event. Both the timeout and the filter can be omitted. The same sensor can be used for collecting events of different usage interactions, possibly with different artifacts.

It's worth noting that in both modalities the execution (and completion) of the `use` action is *completely asynchronous* with respect to the execution of the operation by the artifact and to the possible consequent generation of events. It is synchronous, however, with respect to the interaction with the specified operation control in the usage interface: if the action succeeds, then it means that such operation control was part of the usage interface, that it has been activated (as when we "press a button"), and that the related operation has been triggered for being executed (as soon as its guard is satisfied). More details about this point can be found in [8].

Besides `use` and `sense` actions, the other actions of this group are useful for getting the identifier of an artifact given its name (`lookupArtifact`), for creating new instances of an artifact (`makeArtifact`) specifying the name and the template, and for disposing of an existing one (`disposeArtifact`). Actually, these actions are not primitive but realised on top of the basic `use` and `sense` actions working on predefined artifacts: in particular, `makeArtifact` concerns the use of *factory*, and `lookupArtifact` and `disposeArtifact` of *registry*. The same applies also for lookup, create, and dispose actions of the first group (labels 1, 4, and 5).

The third group of actions (labelled 12–15) concerns continuous observation, i.e., the capability of perceiving artifacts observable events without interacting with them. Also in this case the two modalities are supported, with and without sensors. The `focus` actions can be used to observe an artifact (intuitively, to focus one's attention on that artifact so as to observe any changes that occur on it over time). The first `focus` action (label 12) corresponds to the first modality: the action succeeds if the `Aid` artifact exists, and as an effect every observable event generated by the artifact (despite the specific operation that caused it, possibly executed by any other agent) is made observable to the agent as a new belief `artifact_event(Aid,Event)` in agent's belief

Gameboard Usage Interface
Operations and Observable Events possibly generated:

```
move(X,Y,X1,Y1):
  {move_ok(X,Y,X1,Y1),wrong_move(X,Y,X1,Y1),
   winner(Who)}
getContent(X,Y):{cell_content(X,Y,{white|black|empty})}
isMoveAllowed(X,Y,X1,Y1):{yes,no}
```

No Observable Properties

Lamp Usage Interface
No Operations
Observable Properties:

```
turn({black|white|none})
```

Table 2: Usage interface of the gb game-board (top) and of the lamp artifact (bottom).

base. In the second case, the event is instead collected in the specified sensor. Also for `focus`, a filter can be specified in order to select which types of event to actually observe. `stopFocussing` is used to stop observing the artifact. It is worth mentioning here the differences between `focus` and `sense` actions: `sense` is an internal action, since it inspects a sensor (which is considered part of the agent); `focus`, instead, is external, enabling continuos observation of events that directly cause belief base update in the first modality, and sensor content update in the second modality. Finally, `observeProperty` concerns the capability to inspect observable properties of artifacts without interacting with them, specifying a filter for selecting the property to observe. Analogously to events, a property is represented as a tuple of possibly typed information.

3.2 Integration with Existing Agent-Oriented Programming Platforms

In the remainder of this section we briefly describe how such general integration model has been applied in the context of two well-known cognitive agent-oriented programming languages and platforms based on the BDI model, namely *Jason* [2, 1] and 2APL [4], as well as an agent-oriented framework used for programming concurrent applications, called simpA [9]. To give a taste of the integration, for each one we provide excerpts of the code where most of the CARTAGO actions are used. The excerpts refer to a toy but quite illustrative example, the game *Othello* (also called *Reversi*), programmed with agents and artifacts. The game is modelled as a workspace with two agents, playing the role of game players, and two artifacts, a game board called `gb` and a lamp `lamp`, used to model respectively the game board and a coordination artifact defining the player's turn for the next move. The game board artifact is designed with the purpose of keeping the (shared) state of the game, providing an interface for agents to play, encapsulating and enforcing the rules of the game. Here we consider a simple usage interface (see Table 2, top). It provides a `move` operation to make a move. The operation can generate different kinds (and numbers of) events: `move_ok` if it is a valid move, `wrong_move` if the move is not valid. When the move succeeds, another event might be generated: `winner(Who)` specifying the winner, if there is one. The `getCellContent` operation control is used to inspect the content of a specific cell, creating an observable event `cell_content(X,Y,PieceType)`, and `isMoveAllowed` to check if a move is valid. The usage interface has no observable properties; as a design alternative, we could

have provided a set of `cell_content(X,Y,PieceType)` observable properties, then using the `observeProperty` action to inspect their values, without the need of having `getCellContent` operation.

The artifact `lamp` is used to indicate the player's turn. Its usage interface (see Table 2, bottom) accounts for just an observable property `turn(Turn)`, that indicates the current turn: `black`, `white`, or `none`. The game board and the lamp are linked so that every time a new turn is established, the observable property `turn` of the lamp is updated and the related event generated. Essentially, player agents observe the lamp to know when it is their turn and interact with the game board to make their move.

In a more articulated version of the example, multiple matches can take place in different workspaces, created, and destroyed dynamically, with a tournament manager agent (role) in charge of running the tournament, configuring the workspaces and starting the matches.

3.2.1 Integration with Jason and 2APL

The integration with the *Jason* and 2APL platforms has been quite straightforward, since both platforms are fully Java based, and both provide direct support for: (*i*) extending the basic set of agent actions, and (*ii*) for defining new customised environments (as a source of events). In the *Jason* case, new internal actions can be implemented extending the `InternalAction` class, while customised environments can be realised by extending the `Environment` class, both provided by the core API. Similarly, in 2APL new internal actions can be implemented extending the `Plan` class provided in the core package, and customised environments can be defined by extending the `Environment` class.

Then, CARTAGO actions have been integrated by defining new (internal) actions, since these actions are meant to be independent of the specific (CARTAGO) environment used, and can be thought as basic capabilities provided natively to agents. For *Jason* in particular, internal actions are prefixed by a library name and a dot (e.g., `cartago.use`).

The following code excerpts provide a brief overview or the use and observation of artifacts both in *Jason* and in 2APL, taking the game Othello as example. The game player starts observing the `lamp` artifact by means of a `focus` action; `gb` is the game board artifact. As soon as it observes an event `property_changed(turn(Who))` with `Who` matching its colour, it selects and performs a move. The *Jason* version is as follows:

```
myColour(white). // an initial belief

+!play
  <- ...
     cartago.focus(lamp);
     ...

+artifact_event(lamp,property_changed(turn(Who)))
  : myColour(Who)
  <- ...
     // inspect the gameboard at X,Y
     cartago.use(gb,getContent(X,Y),s0) ;
     cartago.sense(s0,cell_content(X,Y,Content),1000)) ;
     ...
     <decide to place a piece in some position MX,MY>
     ...
     // note that MX and MY are now already bound
     !perform_move(Who,MX,MY).
```

In the excerpt above, in order to select a move the agent may need to interact with the game board in order to retrieve the content of cells. This is realised by `use`, acting

on the `getContent` operation control and a related `sense`, using `s0` as sensor. Note that this example shows the use of both modalities of observation, the first with artifact events which becomes implicitly part of the belief base of the agent, and the second where such observable events are managed through sensors (i.e., the perceptual memory as previously described). The 2APL version is as follows:

```
Beliefs:
  myColour(white).

PC-rules:
play <-  true |
{ ...
   focus(lamp);
   ...
}
event(lamp,property_changed(turn(Who))) <- myColour(Who) |
{ ...
   // inspect the gameboard at X,Y
   use(gb, getContent(X,Y), s0);
   sense(s0, cell_content(X,Y,Content), 1000);
   ...
   <decide to place a piece in some position MX,MY>
   ...
   // note that MX and MY are now already bound
   perform_move(Who,MX,MY)
}
```

Since both agent languages are logic-based, the integration style is very similar.

Then, in order to move, the agent performs a simple `move` action. Here is the *Jason* version:

```
+!perform_move(C,X,Y)
  <- ...
     cartago.use(gb,move(C,X,Y)).

+artifact_event(gb,wrong_move(C,X,Y)
  <- ...
     <select another move>
     ...
+artifact_event(gb,winner(Who))
  :  myColour(Who)
  <- cartago.use(console,print("I won!")).

+artifact_event(gb,winner(Who))
  :  not myColour(Who)
  <- cartago.use(console,print("Damn it.")).
```

Note that no sensors are specified: in this case we want to process observable events as beliefs about perceptual information added to the belief base (in this case, `wrong_move`, `move_ok`, and `winner(Who)`). As soon as the agent observes a `winner` event, it uses a console artifact to print a message. The 2APL version is similar:

```
PC-rules:
perform_move(C,X,Y) <- true |
{ ...
   use(gb,move(C,X,Y))
}

event(gb,wrong_move(C,X,Y)) <- true |
{   <select another move> }

event(gb,winner(Who)) <- myColour(Who) |
{ use(console,print("I won!") }

event(gb,winner(Who)) <- not myColour(Who) |
{ use(console,print("Damn it.")) }
```

3.2.2 *Using Artifacts in* simpA

simpA is a framework for programming concurrent/multi-core applications introducing an agent-oriented abstraction layer based on the A&A programming model, on top of the

basic object-oriented layer. Currently, it is implemented as a framework on top of Java, and it provides a simple API to program agents and artifacts as basic building blocks for structuring a concurrent software system. The model of agents supported in simpA is *activity-oriented* (not cognitive or BDI): a programmer defines the agent's active behaviour by specifying the agenda of activities that the agent has to do at runtime. As a long-term memory, each agent has a *memo* space, an associative data structure where agents can dynamically store/remove useful information for their activities, in terms of "memos" (which are called tuples of data), possibly partially specified. simpA natively supports CARTAGO actions. Details about the simpA programming model can be found in [9] and in the simpA website[2]. Here we use simpA as an example of non-cognitive agent-oriented platform, so as to emphasise the integration of heterogeneous types of agents working together in shared workspaces.

Differently from the cognitive platform cases, here we support only one interaction modality, based on sensors. In the following excerpt, the same Othello player is shown. Following the simpA model, the behaviour of the player agent is defined by a main activity, composed by a `init` subactivity, a persistent `play` activity, which starts as soon as the `init` activity is completed, and it is executed repeatedly until a memo `won` or `lost` is found in the memo space. In the `init` activity, the agent uses the `focus` action to start observing the lamp artifact:

```
public class OthelloPlayer extends Agent {
  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="init"),
    @TODO(activity="play", pre="completed(init),
                 !memo(won),!memo(lost)", persistent=true),
    @TODO(activity="enjoy", pre="memo(win)"),
    @TODO(activity="cry", pre="memo(lost)"),
  }) void main(){}

  @ACTIVITY void init() throws ActivityFailure {
    memo("myColour","white");
    focus("lamp","s0");
    ...
  }
```

Similarly to the behaviour of *Jason* and 2APL agents, playing means waiting for the turn and then selecting and performing a move. In the `waitTurn` activity, the agent performs a `sense` action on the sensor used for observing the lamp, waiting to perceive an event indicating its turn. As soon as such event is perceived, in the `selectMove` activity the agent interacts with the game board artifact to check the content of cells (by executing `getCellContent`).

```
  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="waitTurn"),
    @TODO(activity="selectMove", pre = "completed(waitTurn)"),
    @TODO(activity="performMove",
                        pre = "memo(new_move_selected,_,_))")
  }) void play(){}

  @ACTIVITY void waitTurn() throws ActivityFailure {
    Memo m = getMemo("myColour");
    sense("s0",new PropertyChanged("turn",m.stringValue(0)));
  }

  @ACTIVITY void selectMove() throws ActivityFailure {
    ...
    use("gb",new Op("getCellContent",x,y),"s1");
    try {
      ...
      Perception p = sense("s1","cell_content",1000);
```

```
    String content = p.stringContent(2);
    ...
    memo("new_move_selected",x,y);
  } catch (NoPerceptionException ex){ ... }
}
```

In simpA, the sense action directly returns the event fetched from the sensor (in the case of a successful action) as an object of type Perception. In fact, it can be used just as a synchronisation point: in the waitTurn activity the sense action suspends the execution of the activity until an event matching property_changed(turn(Who)) is found in the sensor, where Who stands for the agent's color.

As soon as a memo about the new move selected is created, the performMove activity starts and makes the move by using the move operation. Differently from *Jason* and 2APL, also in this case we use the sense action to retrieve observable events generated by the move.

```
@ACTIVITY void performMove() throws ActivityFailure {
  Memo move = delMove("new_move_selected");
  String turn = getMemo("myColour").stringContent(0);
  int x = move.intContent(0);
  int y = move.intContent(1);
  use("gb",new Op("move",turn,x,y),"s2");
  Perception p = sense("s2","move_ok|wrong_move|winner");
  if (p.getLabel().equals("winner")) {
    String winnerName = p.stringContent(0);
    if (winnerName.equals(myName)){
      memo("won");
    } else {
      memo("lost");
    }
  } else if (p.getLabel().equals("wrong_move")) {
    ...
  }
}
@ACTIVITY void enjoy() throws ActivityFailure {
  use("console",new Op("print","I won!"));
}
@ACTIVITY void cry() throws ActivityFailure {
  use("console",new Op("print","Damn it."));
}
}
```

Similarly to the previous cases, a console artifact is used to print out a message as soon as a memo about winning (win) or losing (lost) is added to the agenda.

4. BENEFITS OF ARTIFACTS IN PRACTICE: AN OPEN SYSTEM EXAMPLE

In order to provide an example explaining some of the practical benefits in integrating agent platforms within a CARTAGO infrastructure, we here briefly describe a simple experiment that we have conducted. Built with CARTAGO technology, the *RoomsWorld* scenario realises an open system where heterogeneous agents have the possibility to join, test and interact with CARTAGO environments. The work environment is composed by a number of rooms and corridors separated by walls and doors. Once a room is entered, agents should achieve the goal to find and clean trash objects which may appear in the rooms with arbitrary frequencies. To find a trash, an agent has to enter a given room and then perform epistemic actions which are assumed to allow agents to know the exact location of the trash (provided in the form of percepts). Besides, a set of particular artifacts are supplied to agents in order to support their activities. In the example described here, agents have the possibility to use *checklist* artifacts which are placed at the entrance of each room. Also, agents may observe a *watch* artifact,

which provides them with a symbolic record of the ongoing simulated time.

In this experiment, we deploy in the RoomsWorld agents with two different strategies for achieving their goals. The "normal cleaners" simply look for trash exploring the rooms randomly. Once they get the percept of a trash, they reach its location and adopt a "clean" goal. The second type of agents use a more complex strategy, taking advantage of artifacts placed in the environment. This strategy assumes that, once some agent has cleaned a room, it puts a time record in the related checklist, by retrieving the actual time from the watch. This gives subsequent agents the possibility to use the information contained in the checklist, to avoid cleaning rooms that have recently been cleaned by other agents.

For simplicity we here show a part of the code for the cleaner agent—written in *Jason* in this case—using the checklist strategy:

```
+!useChecklist
   <- ?target_check_list(N);
       goToChecklist(N);
      -target_check_list(N);
      cartago.use(checklist(N), readLastNote,s0);
      cartago.observeProperty(watch,current_time(_),s0);
      cartago.sense(s0,last_note(LastTime)),
      cartago.sense(s0,current_time(CurrentTime)),
      !decide(N,LastTime,CurrentTime).

+!decide(N,LastTime,CurrentTime)
  :  threshold(D) & (D < CurrentTime - LastTime)
  <- cartago.use(console,"I'VE DECIDED TO ENTER!");
     !clean(N).

+!decide(N,LastTime,CurrentTime)
  <- cartago.use(console,"INTENTIONS RECONSIDERED: EXPLORING!");
     !explore.
```

Once a cleaner has decided which room it is interested in, it first reaches the related checklist (determined by the target_check_list belief, which is removed after the agent locates the checklist), and uses it to read the latest time record. This is done by executing the readLastNote operation and perceiving the last_note event through a sensor (s0), carrying information about the last time-stamp left by some other agent at the end of its cleaning activity. In order to know the current time, the agent observes the watch artifact, getting the content of the current_time(Time) property.

Once the agent has both information about the checklist time stamp and the current time, it can decide what to do. If the difference between the actual time and the time-stamp retrieved in the checklist is beyond the threshold (say, a day's time), it keeps its intention of going into the room that was previously selected, otherwise it reconsiders its intentions and goes on to access another checklist.

The experiment shows that cognitive agents using checklist outperform normal agents in terms of achieved goals. We compared the performances of two teams of agents situated in environments with four rooms [3]. Whereas normal agents in the first team waste time and resources looking for trash in rooms that have just been cleaned, the team of agents exploiting the checklists is better able to balance activities aimed at achieving the clean goals.

This simple case study makes it possible to highlight some of the benefits of using artifacts as mediating tools for interactions. In general terms, the use of tools is a suitable

---

[3]The experiment is available at CARTAGO website

strategy to simplify choices and ease computational burden. From the point of view of an outside observer, the checklist plays the role of enhancing and distributing information. The knowledge which is acquired at the subjective level by agents during their tasks can be shared by checklists, which allow the overall society to benefit from that information. From the point of view of the agents, artifacts allow them to externalise activities in order to reach desired states [3]. Their use changes the task as well as enhances agent's cognitive abilities, while the means to achieve goals is dramatically simplified [7]. In particular, checklists can be viewed by agents as useful tools for representing the problem space in their situated contexts. Agents may rely on checklist information to improve their practical behaviour, thus saving exploration and epistemic activities [5]. In so doing, agents can take advantage of the information left by other agents without the need for mutual presence within location and time, nor the need for message broadcasting.

Checklists organise and make available relevant information as a permanent modification of their state, persisting even when an agent who produced it decides to leave the environment. This has a multifaceted importance in the context of an open system, where different agents may asynchronously operate, with interleaved presence in specific rooms. With respect to a solution based on message exchange, here agents locally recur to information concerning their *actual* purposes. Although a message based broadcast to the whole society would allow agents to maintain an updated knowledge of the entire environment, this would require agents to waste their computational resources to continually process messages which are not strictly relevant for their current needs. Messages would allow agents to know exactly what is happening in the overall environment, but the information coming from far rooms is unlikely to be relevant for agents. Moreover, an open-system assumption may imply the presence of agents with arbitrary architectures, hence without the necessary abilities for message passing/broadcasting, nor a shared semantic to understand their contents.

5. CONCLUDING REMARKS

In this paper we described a general approach for integrating the A&A programming model with existing agent-oriented platforms—cognitive/BDI in particular—enabling MAS engineers to (*i*) have a high-level and general-purpose programming model to develop agent environments, and (*ii*) to design and develop MAS in which (possibly heterogeneous) agents dynamically create, share, and use artifacts to improve their work, in particular their cooperation.

We think that this integration could be a first step towards a novel form of interoperability in open multi-agent systems, based on agents working in the same workspace(s), and sharing and using the same artifacts with a common understanding of such use (possibly using artifact "manuals" if necessary). Currently, CARTAGO lacks a reference model and ontologies for defining the machine-readable content of artifact manuals, and this partially limits the level of inter-operability and openness that can be achieved. Accordingly, some planned future work is in this direction: towards the definition of a common model for manuals, with a shared semantics of the description of artifact purpose and usage. For this purpose, existing work on related research domains, such as the Semantic Web, will be an important

reference [11].

Besides open systems and interoperability, the integration presented here makes it possible to easily create testbeds for benchmarking and comparing different agents and MAS models and their design solutions, possibly developed with different agent-oriented languages and platforms. Future work will also account for exploring how the different cognitive models may differ in their performances given their different reasoning processes and problem-solving styles.

6. REFERENCES

[1] R. Bordini and J. Hübner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *CLIMA VI*, volume 3900 of *LNAI*, pages 143–164. Springer, Mar. 2006.

[2] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.

[3] A. Clark and D. Chalmers. The extended mind. *Analysis*, 58: 1:7–19, 1998.

[4] M. Dastani and J.-J. Meyer. A practical agent programming language. In *Proceedings of the fifth International Workshop on Programming Multi-agent Systems (ProMAS'07)*, 2007.

[5] V. Kaptelinin, B. A. Nardi, and C. Macaulay. Methods & tools: The activity checklist: a tool for representing the "space" of context. *interactions*, 6(4):27–39, 1999.

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

[7] D. Norman. Cognitive artifacts. In J. Carroll, editor, *Designing interaction: Psychology at the human–computer interface*, pages 17–38. Cambridge University Press, New York, 1991.

[8] A. Ricci, M. Viroli, and A. Omicini. The A&A programming model & technology for developing agent environments in MAS. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Post-proceedings of the 5th International Workshop "Programming Multi-Agent Systems" (PROMAS 2007)*, volume 4908 of *LNAI*, pages 91–109. Springer, 2007.

[9] A. Ricci, M. V. Viroli, and G. Piancastelli. simpA: A simple agent-oriented java extension for developing concurrent applications. In M. Dastani, A. E. F. Seghrouchni, J. Leite, and P. Torroni, editors, *Workshop on Languages, Methodologies and Development Tools for Multi-Agent Systems (LADS'007)*, pages 176–191, Durham University, U.K, Sept. 2007.

[10] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[11] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.

[12] D. Weyns, A. Omicini, and J. Odell. Environment as a first-class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, Online First, July 2006. Special Issue: Environment for Multi-Agent Systems.