The Knowledge Engineering Review, Vol. 34, e2, 1–28. © Cambridge University Press, 2019
doi:10.1017/S026988891800005X

Dimensions in programming multi-agent systems

OLIVIER BOISSIER[1], RAFAEL H. BORDINI[2], JOMI F. HÜBNER[3] and ALESSANDRO RICCI[4]

[1]*University of Lyon, IMT Mines, Saint-Etienne, CNRS, Laboratoire Hubert Curien UMR 5516, France;*
e-mail: olivier.boissier@emse.fr;
[2]*School of Technology, PUCRS, 90619-900 Porto Alegre, RS, Brazil;*
e-mail: rafael.bordini@pucrs.br;
[3]*Department of Automation and Systems Engineering, Federal University of Santa Catarina, Florianópolis, SC 88040-900, Brazil;*
e-mail: jomi.hubner@ufsc.br;
[4]*Department of Computer Science and Engineering, University of Bologna, Cesena Campus, 47521 Cesena (FC), Italy;*
e-mail: a.ricci@unibo.it

Abstract

Research on Multi-Agent Systems (MAS) has led to the development of several models, languages, and technologies for programming not only agents, but also their interaction, the application environment where they are situated, as well as the organization in which they participate. Research on those topics moved from agent-oriented programming towards multi-agent-oriented programming (MAOP). A MAS program is then designed and developed using a structured set of concepts and associated first-class design and programming abstractions that go beyond the concepts normally associated with agents. They include those related to environment, interaction, and organization. JaCaMo is a platform for MAOP built on top of three seamlessly integrated dimensions (i.e. structured sets of concepts and associated execution platforms): for programming belief desire intention (BDI) agents, their artefact-based environments, and their normative organizations. The key purpose of our work on JaCaMo is to support programmers in exploring the synergy between these dimensions, providing a comprehensive programming model, as well as a corresponding platform for developing and running MAS. This paper provides a practical overview of MAOP using JaCaMo. We show how emphasizing one particular dimension leads to different solutions to the same problem, and discuss the issues of each of those solutions.

1 Introduction

Current trends in computer science are facing up to the challenges of building distributed and open software systems operating in dynamic and complex environments. In this context, multi-agent technologies can provide concepts and tools that support possible answers to the challenges of practical development of such systems by taking into consideration issues such as autonomy, decentralization, interaction, and flexibility.

Within the broad field of research in multi-agent systems (MAS), various techniques and concepts related to autonomous agents led to concrete programming models[1]. These are concerned with agent-oriented programming languages, interaction and protocol languages, environment infrastructures, and agent organization model and management systems. The results produced so far indicate the importance of these concepts and abstractions for the development of multi-agent applications.

Nevertheless, and perhaps a bit surprisingly, as we discussed in Boissier *et al.* (2013), the engineering of MAS has been hampered by the use of programming approaches that are mainly focused on subsets of

[1] We invite the reader to refer to the proceedings of the EMAS Workshop series and its predecessors for a broad overview of the area.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

those concepts available at design time, but then not available for programming, or not consistent with those used in programming. Because of this, developers miss the benefits of a comprehensive approach that suitably integrates all these concepts in a way that keeps these abstractions coherent from the design to the programming and execution.

Multi-agent-oriented programming (MAOP), as proposed in Boissier *et al*. (2013), aims at supporting the MAS paradigm at the programming level. It provides a structured approach based on three integrated dimensions of concepts that are useful for designing such complex systems: the *agent dimension* that is used to program the individual (interacting) autonomous entities, the *environment dimension* used to develop shared resources and connections to the real world, and, finally, concepts from the *organization dimension* allow the structuring and regulation of interrelations between the autonomous agents, and between the autonomous agents and the shared environment. We put forward a particular approach to MAOP that is supported by an existing fully fledged platform called JaCaMo[2] (Boissier *et al*., 2013). One of the most important aspects of JaCaMo is that the platform supports programming constructs that match each of the design abstractions of MAOP.

This paper provides an overview and discussion of the main practical programming aspects that concern MAOP, using JaCaMo as a reference platform. To this end, we take as a starting point a conceptual model resulting from the integration of the three main dimensions that structure all these concepts.

After presenting this conceptual model (in Section 2), we illustrate the approach by presenting the development of illustrative programs explaining incrementally how to use and compose these different concepts in order to develop MAS. We first show how to program agents and the shared environment where they are situated (Section 3), then we discuss the programming of coordinated behaviour among those autonomous entities exploiting direct communication, shared environments (Section 4), and agent organizations (Section 5). All along these sections, we discuss how alternative solutions result from different synergies between the available dimensions, that is, an emphasis on a particular dimension generates a particular solution for the given problem. We discuss the benefits and limitations of each of those solutions to approach the same problem. The JaCaMo platform is used as a development tool for that exercise. This way, we illustrate MAOP from a practical point of view and discuss the development of a simple (abstract) system integrating the agent, environment, and organization dimensions. Before concluding, we briefly discuss some related work (Section 6).

2 Concepts

In this section, we discuss the concepts underlying the abstractions that are used in the MAOP approach that is promoted in this paper. We first give a global view on a JaCaMo MAS introducing briefly some of the concepts that are important in this approach. After that, we discuss the programming abstractions that are essential for MAS designers and programmers and which will be extensively used in the remainder of the paper. As mentioned previously, the concepts are organized into three separate dimensions, and this is reflected in the structure of this section.

2.1 Multi-agent system

A JaCaMo MAS is composed of a dynamic set of *agents* interacting within a shared, possibly distributed, *environment* that consists of a dynamic set of *artefacts* (Boissier *et al*., 2013). Agents are goal-oriented autonomous entities, encapsulating a logical thread of control, that pursue their goals by perceiving and acting upon artefacts and by communicating with other agents. Artefacts are used to model any kind of resource or tool that agents can use and possibly safely share, to achieve their goals. An agent can perceive the observable state of an artefact, reacting to events related to that state change, and act by performing actions that correspond to operations provided by an artefact's *usage interface*. Environments can be decomposed into one or multiple *workspaces*, which are artefact containers representing a logical space,

[2] The acronym JaCaMo comes from combining the names of the three platforms on which JaCaMo is based, namely Jason, Cartago, Moise.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

defining a notion of *locality*[3]. An agent can join (and work with artefacts of) multiple workspaces at the same time. Workspaces can be located on different network nodes, in case of physically distributed environments.

The agent *organization* abstraction level makes it possible to specify and handle as a first-class aspect the coordinated and organized activity taking place in the system, resulting from the concurrent and complex tasks performed by *groups of agents* interacting with each other or acting within the environment. Changes in the state of the environment may lead agents to react and they may also affect the state of the organization. In order to support the joint work of agents, an organization can regulate and coordinate agent activities. For example, if there is a dependency between the tasks of two agents, when the first task is perceived as concluded through the state of the environment, the organization can require or expect the agent responsible for the second task to engage in the action it has previously committed to execute. The possibility to specify rules at the organizational level regulating and coordinating agent activities can significantly reduce the amount of work developers need to do when implementing agents for complex applications.

Of course, as agents are autonomous, this can only happen if they actively choose to participate in, and to comply with, one or more of the currently existing organizations in the system.

A MAS combines and inter-connects several different life cycles, each consisting of the creation, execution, and destruction of entities such as artefacts, workspaces, agents, and organizations. In this global picture, it is also important to stress the notion of *autonomy*, making clear the difference between agents and artefacts. Even though at some level both entities may be considered as processing/acting entities, only agents are meaningfully described as autonomous entities. Agents are deemed able to make reasoned/motivated decisions about the courses of action that they will take. This implies, as it will be explained below, that agents do not act upon other agents as they do with artefacts; rather, they *interact* with each other: the acceptance of information or the adoption of tasks from received messages are controlled by a decision-making process of the receiver. The set of abstractions expressed in agent organizations are targeted towards the agents in order to regulate and control their autonomy.

2.2 Programming dimensions

In order to support the design and programming of JaCaMo MAS, we propose three sets of (programming) abstractions. Each of these sets of abstractions, to which we refer as *dimensions*, take part in the creation of a multi-agent oriented program: the *agent*, *environment*, and *organization* dimensions.

In this paper, we focus on some of the abstractions and concepts from each dimension and we only discuss the main relations between these dimensions (cf. Figure 1). We give references for the interested reader where all the abstractions and details of our approach can be found. The objective here is to look only at the most foundational concepts and relations so that the reader can get the overall picture of our approach. Even though a few other concepts are introduced in later sections with a more practical view, the description of all the concepts in MAOP and JaCaMo in particular is not possible in this paper due to space constraints.

It should be noted that a MAS may lead to distributed execution of a number of heterogeneous agents, artefacts, and organizations; one of the reasons for *heterogeneity* is that those entities are potentially programmed by different people or indeed representing the interest of different, possibly competing, companies. Moreover, this programming approach give strong support for changes in and reorganization of the system code while the system remains running.

For example, through run-time creation of organization and artefact instances, agents joining and leaving organizations through the roles they choose to adopt, and so forth. This is useful for many of the features needed in modern computer systems pertaining, for instance, to the Internet of Things (IoT) domain.

2.3 Agent dimension

The *agent dimension* gathers all the programming abstractions that are used to program agents as autonomous software entities that have their own thread of control. Agents constantly perceive and act on the

[3] In fact, it is also *physically* a locality, since a workspace runs on a single host.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

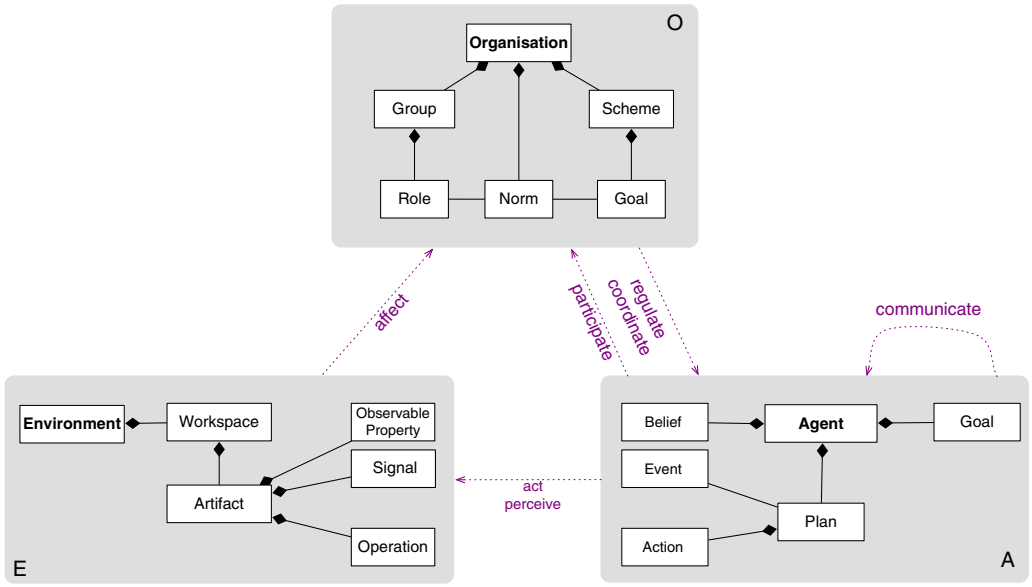


Figure 1 Multi-agent-oriented programming dimensions—agent (A), Environment (E), Organization (O) —with some of their inter-relations and with some of the concepts they use. In this figure, dashed purple links show the relations and inter-dependencies between the dimensions, for example, the agent dimension is related to the environment dimension through acting and perceiving

environment (i.e. on artefacts), interact with other agents, and take part in higher-level entities of the agent organization. In order to exhibit the property of autonomy as explained above, the programming abstractions that are used to program an agent allow it to reason about *what* to achieve by means of symbolically represented *goals* and, also importantly, *how* to do so by means of *plans*, given the current understanding of the perceived state of the system represented by means of *beliefs*.

The plan construct of the programming language is used to define sequences of *actions* and the manipulation of *beliefs* and *goals* when necessary. These are the 'recipes' for action that provide knowhow to the agent. Changes in beliefs and goals may happen by means of plan execution but also as a consequence of interaction with other agents and the environment. Furthermore, the changes in both these types of mental attitudes are recorded internally in an agent as *events*. It is the events that then lead to the execution of particular plans depending on the current state of environment, the agents beliefs, possibly the other intentions (i.e. other plans the agent is already committed to execute), and so forth.

All these abstractions make it possible for agents to be *proactive*, that is, to act so as to achieve their goals, but also to be *reactive*, that is, to react to changes in beliefs about the environment, the agent organization, or other agents. It should be noted that agents may interact with other agents through particular actions that are called *communicative* actions[4].

2.4 Environment dimension

The *environment dimension* offers a set of programming abstractions to represent the shared environment. These programming constructs concern first the notion of *artefact*, the basic environment entity that encapsulates computing or other forms of resources. Artefacts are used by agents through a set of *operations* that agents can perform on the artefact. It offers also a partial view on its state through a set of *observable properties* and on its activities through a set of *signals*. Both observable properties and signals are perceived by the agents only after they *focus* on the corresponding artefact (thus potentially avoiding information overload by only focussing the attention on the artefacts that matter to the agent). It should be

[4] More details about the concepts in the agent dimension can be found in Bratman (1987), Rao (1996), Labrou *et al.* (1999), Bordini *et al.* (2007).

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

noted that the connections between the agent and the environment dimensions happen through the transformation of observable properties and signals issued from the artefacts into beliefs/events of the agents and to the transformation of actions within executing plans of the agents into calls to operations on artefacts.

Note that artefacts can make direct references to real-world environments, to various forms of existing computational resources, or simply support agent coordination through shared resources in an implicitly controlled way. This dimension offers a second important programming abstraction, used to structure the activities of the situated agents by defining topological or symbolic regions called *workspaces*. Artefacts are situated in a workspace. Agents may dynamically *join* or *leave* one or several workspaces. Joining a workspace allows agents to perceive all artefact activity taking place in it[5].

2.5 Organization dimension

Finally, the *organization dimension* gathers the abstractions that are used to structure and guide how the activities resulting from the actions and interactions of the agents within the environment should be coordinated. In an organization, a *group* can be used to provide social structure by means of *roles* that agents may decide to adopt within that group. When taking part in an organization, an agent plays at least one role in a particular group. Norms are used by MAS designers to express the behaviour that is expected from agents playing a particular role in the context of a particular group.

These programming constructs are used to state that agents playing specific roles will have certain obligations or permissions within the group activities in order to achieve collective goals. The individual behaviours are part of collective plans, called *schemes*, that provide a structured way to achieve a collective goal in which various different agents will have to work on different parts (sub-goals) of that activity by using some resources deployed in some configuration of the environment[6].

The development of a JaCaMo application starts by the definition of instances of the abstractions from each of these dimensions and their connections. Since some concepts from different dimensions are aligned (e.g. organizational goals and agent goals, environment operations and agent actions, observable properties and agent beliefs), developers currently use the same identifier to link the dimensions. For instance, if a norm obliges and agent to achieve goal `g34`, the agent has to have, at runtime, a plan to achieve goal `g34`[7]. In the next section, these definitions and their integration are concretely illustrated through code excerpts.

3 Programming agents and shared environments

In this section, we will explore how to program systems of multiple situated agents, interacting with artefacts and also with other agents. *Situatedness* is a basic concept in MAOP. It means that agent's decisions are based on their perception of their surrounding environment and also that the result of their reasoning are actions to be taken upon that environment. To illustrate this relation between the agent and environment dimensions, we start by showing how to use JaCaMo to program agents and artefacts, how to program their interactions, how to distribute them on multiple hosts, and finally how agents can directly interact with each other.

From a methodological point of view, the availability of the agent and environment dimensions promotes a clean separation of concerns about what in a system can be better modelled/programmed as

[5] More details about the concepts in the environment dimension can be found in Weyns *et al*. (2007), Omicini *et al*. (2008), Ricci *et al*. (2009).

[6] More details about the concepts in the organization dimension can be found at Demazeau and Rocha Costa (1996), Lemaître and Excelente (1998), Hubner *et al*. (2002), Boissier *et al*. (2007), Rocha Costa and Dimuro (2009).

[7] This assumption can be relaxed by defining an 'interaction' language (not only for agent communication but also for perception and action), some sort of common language or application ontology that defines a set of goals, actions, and percepts that are translated into environment observable properties and operations, into agent beliefs and actions, and into agent and organization goals. It can also be relaxed if agents are capable of learning, possibly from the application ontology, what `g34` means. For the particular example of plans for required social goals, it is worth mentioning that in MAOP agents often acquire plans at runtime through interaction with other agents or a planner (e.g. encapsulated as an artefact). This kind of sophistication, however, is not considered in this paper and we assume the developer uses matching identifiers to link the dimensions.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

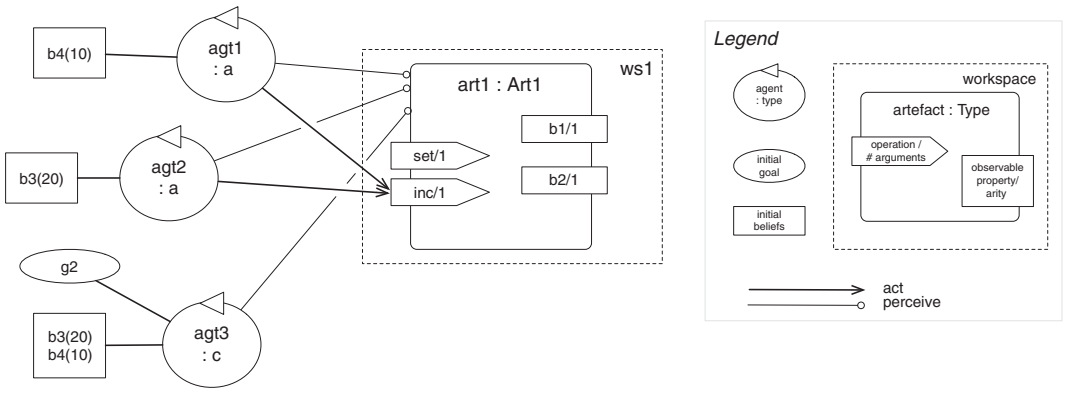


Figure 2 Situated agents interacting with an artefact

task/goal-oriented entities (agents), designed to pro-actively achieve such goals and what can be better modelled/programmed instead as resources/tools (artefacts) useful to achieve those goals. Examples of the latter case span from artefacts modelling I/O devices and OS services (e.g. stdin/stdout, graphical user interface, files) to shared data, coordination media (shared knowledge bases, blackboards, message services, etc.), and artefacts representing legacy resources or systems.

3.1 Agent–environment programming

We first show a simple program where three agents (named `agt1`, `agt2`, and `agt3`) perceive one artefact (named `art1`), as shown in Figure 2. The first two agents have the same initial program (file `a.asl`), however, they have different initial beliefs. To start a MAS with such an initial state, a JaCaMo project file as shown below is used[8]. After the initialization, new agents and artefacts can be created by the agents themselves (we discuss this later).

```
mas ker_h1 {
    agent agt1: a.asl {   // a.asl is the source code file for agent agt1
        beliefs: b4(10)   // initial belief for agt1
        focus  : ws1.art1 // initial focus on artifact art1 situated in workspace ws1
    }
    agent agt2: a.asl {   // a.asl is also the source code for agent agt2
        beliefs: b3(20)
        focus  : ws1.art1
    }
    agent agt3: c.asl {   // agent agt3 has a different source code file: c.asl
        beliefs: b4(10), b3(20)
        goals  : g2       // initial goal for the agent agt3
        focus  : ws1.art1
    }

    workspace ws1 {       // creation of a workspace with art1 inside
        artifact art1: tools.Art1(5,0) // art1 is created from tools.Art1 artifact type
    }
}
```

3.2 Agent–environment interaction

All three agents in Figure 2 *perceive* the artefact situated in workspace `ws1` by focussing on it and thus will have beliefs representing the current state of the observable properties `b1` and `b2` (these properties have one assigned value, as indicated by /1 in Figure 2). For instance, if the value of `b1` is 10 in artefact `art1`, the agents have a belief `b1(10)[source(percept)]` in their belief base and this belief is automatically updated every time the value changes in the artefact. Agents `agt1` and `agt2` will also *act* on `art1`

[8] The code for the examples used in this paper is available at `http://jacamo.sf.net/ker2017`. The technical documentation of the JaCaMo platform with all the programming constructs that can be used to write a MAOP is available on the JaCaMo website at `http://jacamo.sf.net/`

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

executing the operation `inc`. The square brackets following a belief predicate are used to add *annotations* to a belief. They may be omitted in the code when there are no constraints on particular annotations of a belief but all beliefs in the belief base will have at least the annotations about the origin of the beliefs.

Agents can *react* to changes in the environment using plans. For instance, the program `a.asl` of agents `agt1` and `agt2` contains the following plan (`planB1`) which is triggered when the value of `b1` changes[9]:

```
@planB1 // label of the plan
+b1(X) : X < 10 <- inc(X/2).
```

That can be read as 'Whenever the belief about the value of `b1` changes, provided the value of `b1` is currently less than 10, the agent may execute the operation `inc`. to increase that value by half'. The part of the plan between: and <– is called *context*. It states the conditions under which the course of actions in the plan *body* is appropriate for handling the *triggering event* (the initial part of the plan preceeding:).

An agent program could be as simple as a line like the one above alone. The developer does not need to explicitly program the perception of the environment (this is done automatically in the agent reasoning cycle based on what artefacts the agent has focussed on) nor handle how the action request is passed on for execution by the artefact.

Beyond reactive behaviour as introduced above, *proactive* behaviour can be programmed by creating goals and reacting to changes in the agent's goals. For instance, the new plan `planB1` below creates a goal `g5` given the same context conditions of the last example. The plan to achieve that goal may have many steps, including the creation of other goals, addition/deletion of beliefs, and execution of actions (i.e. operations provided by artefacts).

This plan may take a long time to finish—for instance, depending on how difficult it is to achieve all the required goals and execute all the required actions—so an observer of the behaviour of the agent could perceive it as a long-term proactive behaviour.

```
@planB1
+b1(X) : X < 10 <- !g5.

@planG5
+!g5 : ... <- ... // the plan to achieve g5
```

On the environment side, the artefact `art1` is an instance of the artefact type `Art1` and is programmed in Java with some provided classes and annotations that support artefact programming. For instance, the implementation of the operation `inc`., which increments the value of the observable property `b2`, is

```
public class Art1 extends Artifact { // program for the Art1 artifact type
    void init(int b1, int b2) {  // creation of observable properties
        defineObsProperty("b1", b1);
        defineObsProperty("b2", b2);
    }
    ...
    @OPERATION void inc(double v) {
        ObsProperty prop = getObsProperty("b2"); // get a reference for b2
        prop.updateValue(prop.intValue()+v);     // and increments it by v
    }
}
```

It is worth noting that, as operations are atomically executed in an instance of an artefact, developers do not need to handle concurrency issues such as when several agents trigger the `inc`. operation at the same time. Furthermore, the platform takes care of suspending a particular agent intention when it requires an action to be executed in the environment. The intention remains suspended until the corresponding operation in the artefact is completed. In a plan body such as … `a1`; `a2`; … , the programmer can safely assume that action `a2` will only be executed after the execution of `a1`. Suspending intentions that are waiting for the execution of environment actions allows agents to carry on executing intentions related to other aspects of the environment or to its internal reasoning. The agent continues its reasoning cycles, that

[9] The syntax of AgentSpeak, the language that inspired our agent language, was in turn inspired by Prolog and thus identifiers starting in upper case are variables.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

is, perceiving and acting on the environment, so as to maintain its *reactivity* to other possible changes in the environment.

3.3 Changing the environment

In the code examples we introduced so far, the environment is fully defined in the project file by the application developer through the definition of workspaces, artefacts, and the agents that join/focus on them. In that case, the environment is fully created and configured at the launching of the system. However, in some cases it may be important to reconfigure and create new parts of the environment while the system is running. For instance, creating artefacts in reaction to some particular situation or from some deliberation of one or several agents. This means that the agents themselves should be able to create and change the environment configuration at run-time. This is another important feature of JaCaMo that enables agents to control the whole environment life cycle by creating workspaces, joining them, creating new artefacts, focussing on them, using them, and destroying them. As an example, the following agent program excerpt illustrates the use of this feature to achieve goal `g4` within plan `planG4`.

```
@planG4
+!g4 <-
   makeArtifact(art4,"tools.Art1",[5,0], ArtId); // creates a new artifact that can be
                                                 // referenced with variable ArtId,
   focus(ArtId);                                 // and focuses on it
   inc(10)[artifact_id(ArtId)];  // uses the artifact (by doing operation "inc")
   .send(agt3,tell,myart(art4)); // sends the name of the new artifact to agt3
   .wait( b2(X) & X > 20 );      // waits for the observable property b1 to be
                                 // greater than 20
   disposeArtifact(ArtId).       // removes the artifact
```

The possibilities for agents to dynamically change the environment include also other programming constructs that can be used in the development of a MAOP. For instance, an agent can create or destroy other agents while executing. Furthermore, we discuss later in this paper how agents can deploy and reconfigure organizations at runtime.

3.4 Distribution

Another important feature of MAOP is that it allows us to distribute the execution of agents and artefacts in a straightforward way. In JaCaMo, this is done as follows. Let us consider that agents of type `c`, in order to achieve goal `g2`, need to perform a heavy computation and several machines should be used to distribute the load. Considering further that such computation is better implemented in Java. We could implement it in an artefact, hence externalizing (i.e. moving outside of the agent program) that computation as shown in Figure 3. In JaCaMo, workspaces and agents can be *distributed* on multiple machines; the simplest way of doing this is using project files[10]. In the example below, two project files would be needed. The first project file ker_h1 runs in host1 and the following lines are added to the previous version of this project file:

```
mas ker_h1 {
    ... // configuration of agt1 and agt2 as before

    agent agt3: c.asl {
        beliefs: b4(10), b3(20)
        goals  : g2
        focus  : ws1.art1
    }

    workspace ws1 { // creation of a workspace with art1 and art2 inside
        artifact art1: tools.Art1(5,0)
        artifact art2: tools.Art2()
    }
}
```

[10] Note that this can also be done in an agent program as shown in the previous section with a particular language construct called internal actions. We do not show it here due to space constraints.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

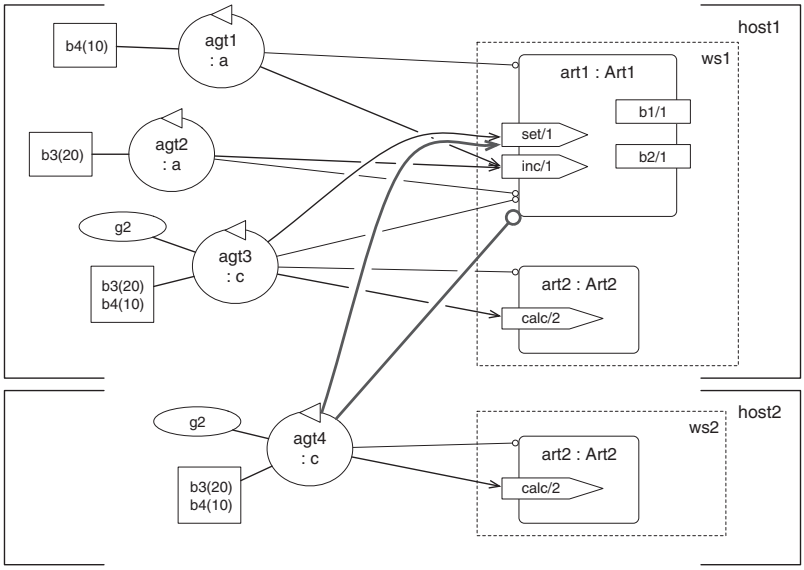


Figure 3 Distributed agents and workspaces

The second project file, to be executed in host2, should be as follows:

```
mas ker_h2 {
    agent agt4: c.asl {
        beliefs: b4(10), b3(20)
        goals  : g2
        focus  : ws1.art1  // ws1 is running on another host
                 ws2.art2
    }

    workspace ws2 {
        artifact art2: tools.Art2()
    }
}
```

The JaCaMo infrastructure manages the communication between the distributed components (workspaces, agents) so that we can deploy the system differently by just changing the project files (i.e. different distributed deployments do not require changing the code of the agents or artefacts).

In this setup, there are two artefacts with the same name (`art2`) but in different workspaces, one is used by `agt3` and the other by `agt4`. These artefacts have the calc operation that returns some value instead of changing some observable property as done, for example, in the operation inc. of artefact `Art1`. The `calc` operation is implemented as follows:

```
@OPERATION void calc(double a, OpFeedbackParam<Double> b) {
    int r = ...; // a complex computation is done here
    b.set(a+r);
}
```

Note that return values are modelled as *action feedback*, and there can be more than one. The Java API exploits `OpFeedbackParam` argument types to represent such feedback.

The agents do not need to know where artefacts are running to use their operations. For example, agents of type `c`, based on which artefacts they are focussing, can use distributed artefacts in plans as follows[11]:

```
@planG2
+!g2 : b4(X) & X mod 5 == 0 // 'mod' is the remainder operator
   <- calc(X,Y); // operation of artifact art2 @ host 2
      set(Y).    // operation of artifact art1 @ host 1
```

[11] If an agent requests an operation which is provided by multiple artefacts (located in the workspaces that the agent has joined) and without explicitly stating which one is meant, no error occurs and one is selected non-deterministically by the infrastructure.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

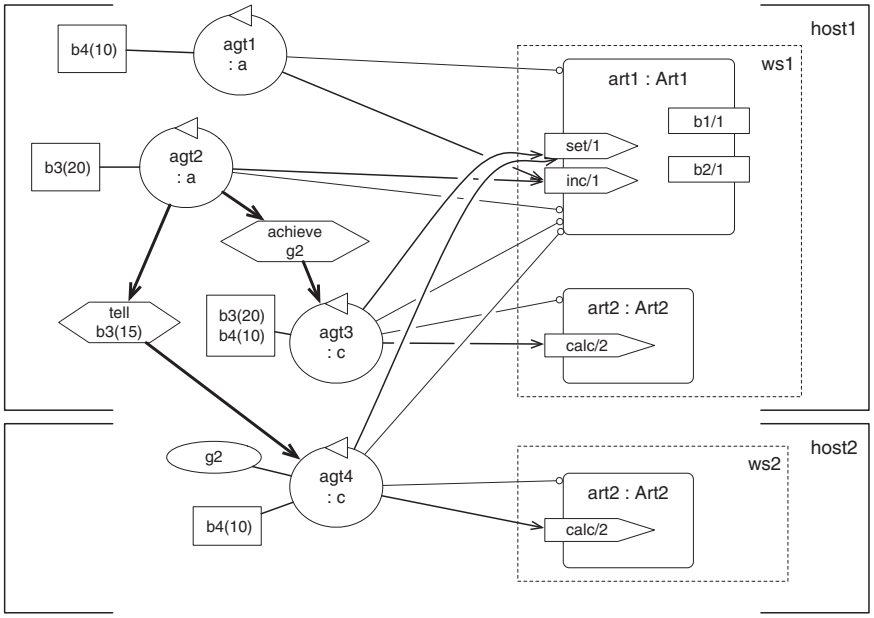


Figure 4 Agent communication

Although we do not exemplify this here, often artefacts are used to connect human users to the system or to connect the system to elements of a real-world environment. The examples included in the JaCaMo distribution exemplify artefacts used for graphical user interfaces and various other uses of artefacts.

3.5 Agent–agent interaction

Besides the interaction that agents can have with or through the artefacts in the environment, direct agent–agent interaction is also possible. It is based on *speech acts*. To illustrate this kind of interaction, two changes are made in the program of `agt2` and project file as follows: (i) the goal `g2` of `agt3` will not be set in the project file anymore, it will be delegated by `agt2` instead, and (ii) the belief `b3` of agent `agt4` will be informed by `agt2` , as shown in Figure 4.

To program the first change, agent `agt2` uses the `.send` (internal) action to send an *achieve* message to `agt3`. This action has three parameters: the name of the receiver, the performative (stating how the sender intends the message to affect the receiver), and the content. The performative used in this example, `achieve`, asks the receiver to achieve a new goal corresponding to the content of the message. The code used by `agt2` to send the message is thus the following:

```
.send(agt3,achieve,g2)
```

When `agt3` receives this message, the default interpretation for the message (as implemented in the platform but user customization is possible) leads to the creation of a new goal `g2`. The program for agent `agt3` does not need to be changed since it already has a plan to react to the creation of a goal `g2` (previously created from the project file instead).

To implement the second change, `agt2` uses the *tell* performative:

```
.send(agt4,tell,b3(15))
```

The default interpretation of a *tell* message on the receiver side leads to the creation of a belief corresponding to the content of the message (`b3(15)` in that case). Again, the code for agent `agt4` does not need to be changed. In JaCaMo, the source of an agent's beliefs is either external (through the initial project file, from tell messages sent by other agents, or perception of the environment in particular through observable properties of artefacts) or internal (i.e. created with a belief addition operator '+' within some executed plan in

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

the agent program). Similarly, goals can be created externally (through the initial project file or achieve messages sent by other agents[12]) or internally (using the goal addition operator '`!`' in the agent program).

3.6 Wrapping up

In this section, we showed how to program a set of agents situated in a shared environment populated with artefacts. The following features have been demonstrated through programming constructs from the agent and environment dimensions:

Situatedness: using the environment dimension and the ability to join workspaces where artefacts have been deployed. Artefacts can be individual computational tools, multi-agent coordination tools, graphical interfaces, or interfaces with the external world. Agents can perceive this environment and represent part of it through beliefs. They can also act on it through the current set of actions associated with the operations that are exposed by the currently instantiated artefacts.

Autonomy: agents have a reasoning cycle, with support to reactive and proactive behaviour through programming constructs such as goals, plans, intentions, etc.

Interaction: agents can interact through communicative actions and the shared environment.

Openness: agents can enter and leave workspaces and also create and change the environment structure and configuration at run time, by dynamically creating and disposing artefacts.

Distribution and deployment: the JaCaMo project files makes it possible to specify the initial deployment of multi-agent applications that could involve multiple workspaces running on different nodes (hosts).

4 Programming coordinated agents in a multi-agent system

The abstractions provided by the different dimensions can be flexibly combined in order to define the strategies to solve problems, for example, coordination problems. In this section we see this point by considering a classic coordination mechanism, the Contract Net Protocol (Foundation for Intelligent Physical Agents (FIPA), 2002). We show here how to program it with the agent dimension (i.e. using direct agent interaction as commonly done in MAS) and based also on the environment dimension using environment constructs (such as artefacts and workspaces) as a coordination medium. In the next section, we will look at another approach, based on the organization dimension, in which organization specifications are used to express the strategies of this coordination mechanism.

The example, used throughout this section and the following, considers an initiator agent that announces a call for proposals (cfp) asking for some agent (a contractor) to perform a task. Several participant agents may answer the call with their proposals or refusal to do the task. The protocol ends with the initiator agent selecting a winner and announcing its decision to the participant agents.

4.1 CNP with a message-based implementation

The CNP can be programmed using message passing between agents, as illustrated in Figure 5. The *askOne* performative is used by the initiator (agent `agt1`) to consult the participants (agents `agt2`–`agt4`) about their price for task `t1`. The default behaviour for a receiver of such a message is to consult its belief base for the message content (the query is `task(t1,Price)` in this case) and to send back a *tell* message with the result of the query (e.g. `send(agt1,tell,task(t1,300))`. It should be noted that the initiator simply chooses the lowest-price bid.

In this section, we do not intend to program agents with sophisticated reasoning, so their programs, presented below, are the simplest we can write. Comments are used to explain them. Code *initiator.asl* for agent *agt1* contains an initial goal aiming at achieving allocate(t1) and the plan labelled `allocateTask` that achieves this goal[13].

[12] This assumes full cooperation. However, also different scenarios can be handled, where, for example, an agent can reject a goal delegated by another agent, or one which the organization forbids.

[13] We use the `.broadcast` command instead of `.send`, which is used to send a message to all agents in the system.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

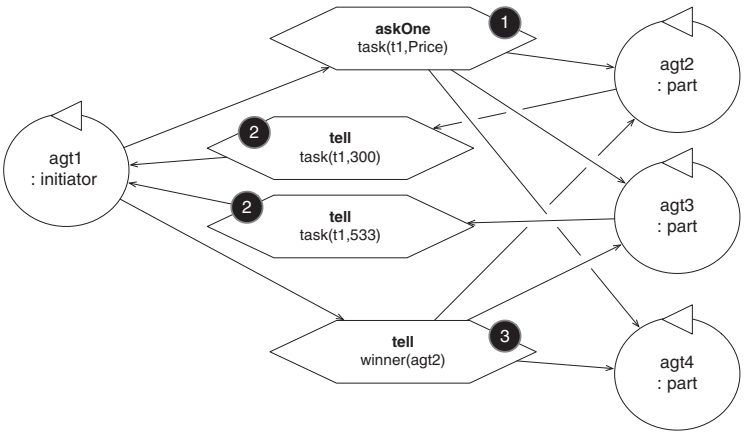


Figure 5 CNP: message-based version

```
!allocate(t1).  // initial goal: allocate task t1

@allocateTask   // plan that implements the protocol on the initiator side
+!allocate(T)
   <- .broadcast(askOne,task(T,P)); // announces task t1 to all agents
      .wait(2000);                  // waits 2 seconds for the proposals
      // places all received bids into list L
      .findall(bid(P,A),task(T,P)[source(A)],L);
      .print("Bids: ",L);
      .min(L,bid(WOf,WAg));    // selects the best bid
      .print("Winner is ",WAg," with ",WOf);
      .broadcast(tell,winner(WAg)). // announces the winner of the allocation
```

Code *agt2.asl* for a participant agent has an initial belief where a price is fixed for the task `task(t1,300)`. It also has two plans, one for handling the case of winning an allocation and another for performing task t1.

```
   task(t1,300).                    // initial belief (used to respond to askOne)
   +winner(A) : .my_name(A) <- !t1. // if I am the winner, I have to do the task
   +!t1 <- ...                      // plan to achieve t1
```

Code *agt3.asl* used for another participant agent does not fix the price with an initial belief but with a rule generating a random price for task `t1`. It also has the two plans for the winning case and for doing task `t1`.

```
task(t1,P) :- P = math.random * 100 + 500. // rule that sets a random value for task t1
+winner(A) : .my_name(A) <- !t1.           // if I am the winner, I have to do the task
+!t1 <- ...                                // plan to achieve t1
```

Code *agt4.asl* for another participant agent is more sophisticated in the sense that the agent will only answer to the *askOne* price query if it comes from a friend and if it has a plan for task `t1` in its plan library (see plan `fixingPrice`). As we can see this code also has a plan for the winning case and one for completing a task. However, this plan is for task `t2` and not for task `t1` for illustration purposes, as explained below in the execution steps.

```
   friend(agt1).          // agt1 is my friend

   @fixingPrice
   +?task(T,P)[source(A)] // plan for answering a query goal issued by another agent A
      : friend(A) &       // the query comes from an agent that is my friend
       .relevant_plans({+!T},Plans,_) &  // getting the list of all plans for task T
       Plans \== []
   <- P=300.              // answer with fixed price (300) if I have a plan for T

   +winner(A) : .my_name(A) <- !t2. // if I am the winner, I have to do the task

   +!t2 <- ...            // plan to achieve t2
```

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press



Figure 6 MindInspector of agent `agt1` after the call for proposals

When we launch these agents using a JaCaMo project, the execution produces the following steps:

1. `agt1` broadcasts to all agents a query for the price of task `t1` (line 5) and waits 2 seconds for the answers (line 6);
2. `agt2` answers with a fixed price according to its initial belief `task(t1,300)`;
3. `agt3` answers with a random value following the rule in its belief base;
4. `agt4` does not answer given that it does not have a plan for task `t1` (it only has a plan for task `t2`);
5. `agt1` receives the answers as beliefs in its belief base (contents of messages with a tell performative are interpreted as belief additions) as illustrated by the screenshot of the JaCaMo mind inspector in Figure 6;
6. `agt1` collects the bids (line 8) into a list based on a query `task(T,P)[source(A)]` (i.e. the bids it has received from agents `agt2` and `agt3`), selects the best bid in line 10 (the agent who placed the bid with the lowest price for the task), and broadcasts the name of the winner to all agents (line 12);
7. `agt2` starts the task as soon as it receives the message saying it has been awarded the contract for that task.

The output of the execution of these agents on the console is as follows:

```
[agt1] Bids: [bid(553.278486221895,agt3),bid(300,agt2)]
[agt1] Winner is agt2 with 300
[agt2] Doing task t1
```

We now briefly discuss this implementation of the CNP and highlight some of its drawbacks:

1. The *initiator* agent uses `.broadcast` for the announcements (i.e. cfp and winner), so all agents, even those not interested in the allocation process, receive the message and have their belief bases updated with the new belief (unless they are customized to ignore specific messages).
2. In *open systems*, where new agents can enter and leave at any time, if an agent arrives after the cfp announcement (and before the deadline), it will miss the opportunity to participate in the ongoing allocation process.
3. In usual implementations of CNP, *participant* agents do not know the bids placed by the other agents and thus are not able to decide whether to make a better offer. Allowing that to happen in the CNP implementation described above would require all bids to be *broadcast*, which would cause a significant increase in the number of exchanged messages that would be required for allocating the tasks.

Even though we could improve the agent programs to avoid some unnecessary messages, in the next section we avoid these drawbacks using artefacts instead.

4.2 CNP with an artefact-based implementation

The artefact-based implementation, as illustrated in Figure 7, slightly changes the previous protocol to better highlight the possibilities brought about by the use of artefacts. Instead of broadcasting the cfp, `agt1` creates an artefact that shows, as an observable property, the required task and the current best proposal. Participant agents interested in that allocation process can focus on that artefact and, if they have a better proposal (although this is not allowed in the original 'sealed bid' CNP protocol), they can place further bids using the operation bid. After the deadline, `agt1` checks the best proposal and announces the `winner` using again the artefact (the operation `set_winner`). Agents then perceive who is the winner based on the winner observable property of that artefact.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

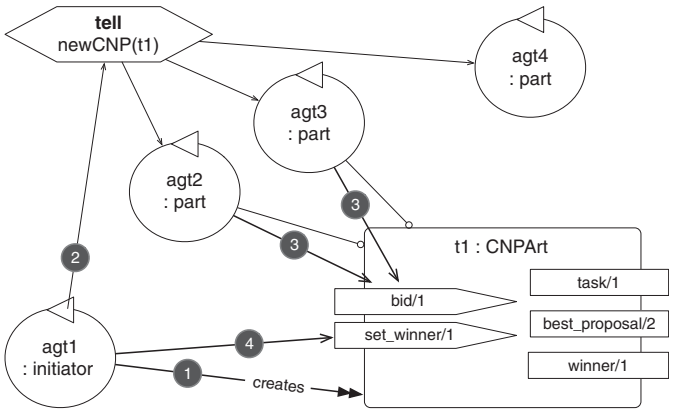


Figure 7 CNP: artefact-based version

To implement this version, the program of the initiator `agt1` is changed to[14]:

```
!allocate(t1).  // initial goal: allocate task t1

@allocateTask
+!allocate(T)
   <- // creates the CNP artifact and initializes it with the task to be allocated "t1"
      makeArtifact(t1,"protocols.CNP", ["t1"], ArtId);
      focus(ArtId);                  // focuses on this artifact
      // broadcasts the name of the CNP artifact to announce its creation to others
      .broadcast(tell, newCNP(t1));
      .wait(2000);                   // waits an amount of time for the proposals
      ?best_proposal(WAg,WOf);       // consults the belief base for the best proposal
      .print("Winner is ",WAg," with ",WOf);
      set_winner(WAg).               // sets the winner on the CNP artifact
```

Comparing the code of plan allocateTask to the one in the previous section, it can be noted that the program is shorter, since part of the previous code was externalized in the CNP artefact, for example, the determination of the best proposal. We can also notice that the belief base of `agt1` is not populated with all the proposals as before. It only gets the updates of the *current* `best_proposal` belief from the CNP artefact.

On the participant side, the code *agt2.asl*, *agt3.asl*, and *agt4.asl* are extended with the following two plans:

```
@focusCNP
+newCNP(T) : task(T,_) <- lookupArtifact(T,ArtId); focus(ArtId).

@bidding
+best_proposal(_,X) : task(t1,MyPrice) & MyPrice < X <- bid(MyPrice).
```

In the `focusCNP` plan, the agent reacts to receiving the announcement of the creation of the CNP artefact by looking for the artefact with the given name (variable *T*) and focussing on it if it has a belief `task` for the announced task handled by the CNP artefact. The `bidding` plan reacts to changes in the `best_proposal` observable property: if the agent's price is better than the current proposal, the agent places a new bid through the CNP artefact using the bid operation available on CNP artefacts. It is worth noting that agents now have the possibility to improve their bids compared to the bids from other participants throughout the execution of the biding phase. More importantly, agents can retrieve information about all available artefact instances when joining a workspace. The broadcast in the code above alerts agents currently in the system about the new CNP artefact so that agents do not need to be constantly looking for newly created artefacts. Therefore, agents that just entered the system do not miss the opportunity to participate in allocations that had been already announced when they joined the system and that are still within the deadline. Note also that many instances of the same artefact type can be created at the same time,

[14] In the agent code, `t1` is the name of the artefact, whilst `ArtId` is the identifier of the artefact whose name is `t1`. The value of `ArtId` is determined by the underlying infrastructure that manages artefacts.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

so several CNP processes for various different tasks can be running in parallel. The code of the CNP artefact is as follows:

```
public class CNP extends Artifact {
    boolean open = true;

    // creates two observable properties for the task given
    // as argument when the artifact instance is created
    void init(String task) {
        defineObsProperty("task", task);
        defineObsProperty("best_proposal", "none", Integer.MAX_VALUE);
    }

    @OPERATION void bid(double p) {
        if (!open)
            return;
        // update of observable property for the best proposal
        ObsProperty prop = getObsProperty("best_proposal");
        int best = prop.intValue(1);
        if (p < best) {
            prop.updateValue(0, new Atom(getOpUserName()));
            prop.updateValue(1, (int)p);
        }
    }

    @OPERATION void set_winner(String w) {
        // add a new observable property to the artifact
        defineObsProperty("winner", new Atom(w));
        open = false;
    }
}
```

The execution of these agents has the following steps:

1. `agt1` creates the artefact (line 6) and broadcasts the information about the newly created task allocation to all agents (line 9) so that they can focus on the relevant artefact if interested;
2. `agt3` focusses on the artefact and places a bid;
3. `agt2` focusses on the artefact and places a better bid;
4. `agt4` does not focus on that artefact since it is not interested in task `t1`. As shown in Figure 8, its belief base does not include the observable properties from the CNP artefact.
5. `agt1` looks for the best proposal in line 11 (the `best_proposal` observable property) and simply sets the agent that placed it as the winner in line 13 (of course, `agt1` could use more complex criteria to decide the winner);
6. `agt3`, the winner, does the task.

The output of the execution is the same, but the belief bases of the agents are quite different: (i) agent `agt4` does not focus on the CNP artefact and so does not have the belief about the current best proposal for that task; (ii) agent `agt1` does not store all proposals, only the current best one. The state of the belief bases at the end of the execution is shown in Figure 8.

An important point here is that the programmers choices in regards to the use of the programming dimensions may have impacts on the properties of the resulting program. In this section, we have addressed the limitations listed at the end of Section 4.1 by using an artefact-based implementation in which:

1. Agents not interested in the allocation processes (e.g. `agt4`) simply do not focus on the CNP artefact.
2. Since the current best proposal is an observable property, agents can have better strategies to choose their own bids[15]. For instance, the following plan implements a strategy that always wins the CNP (unless the deadline finishes right before its latest proposal arrives):

```
+best_proposal(_,X) <- bid(X-1).
```

[15] Of course we could change the protocol in the message-based version too, but it would require a lot of extra message passing that are avoided by the features provided by artefacts.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press



Figure 8 MindInspector of agents `agt1`, `agt2`, and `agt4` after the execution of the CNP

3. As mentioned before, even though in this version agent `agt1` broadcasts an announcement of the newly created CNP instance, agents entering the system can search, for instances, of CNP artefacts available in the system, focus on them according to the tasks that they can handle or are interested in, and place their proposals, provided they are not too late with respect to the deadline that agent `agt1` has in its `allocateTask` plan.

The decision to externalize some operation into an artefact (such as the determination of the current best proposal in our example) should be carefully considered during the design of the system, as it may transfer some decision making from an agent to an artefact, possibly limiting the autonomy of the agents (in this case, the determination of the best proposal). For some applications, in particular open systems where the developer does not know what the incoming agents will do, such a limitation in the autonomy could be useful against malicious agents. Note, however, that, as with all design choices about which dimension to use for implementing a particular feature, this should be carefully considered by designers of a MAS.

5 Programming agent organizations

This section goes a step further in the programming of MAS using a MAOP approach. It introduces elements from the organization dimension in order to structure the set of agents interacting in the system and to set up coordinated collective activities within it. While the first subsection explains this with the use of a JaCaMo project file, the second subsection shows how agents can deploy it on-the-fly. The last subsection presents how to use the organization dimension to regulate collective activity, which is of great importance in open systems.

5.1 Shaping an agent organization

The organization is mainly used to shape the set of agents into groups and roles (structure), to specify the coordinated achievement of goals (function), and to assign them to agents in the system by means of roles and norms. By assigning goals to roles using norms, we provide a way to abstract away from the agents and to define the functioning of the system independently of the particular agents that will be available at runtime. As soon as agents start to play roles, the organization becomes instantiated, bringing obligatory, permitted, or forbidden goal achievements to the agents in the context of their roles and participation in groups. These organizational abstractions provide support for handling the overall complexity of the system.

As we did with the programming of agents and environment in Section 3, we start with a simple organization program exemplifying the structural, functional, and normative components of an agent organization. In JaCaMo, the organization is specified in a XML file, `org.xml` in this case. Figure 9 depicts a simplified graphical representation for the content of this file. The structural specification states that for an organization to comply with this specification it must be composed of one group instance of the root group `orgGroup`. All agents taking part in that organization plays the `member` role. The g roup named `orgGroup` contains subgroups of type `taskExecutionGroup` , where participating agents can play three roles: `hirer` (at least one and at most one agent playing it), `contractor1`

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

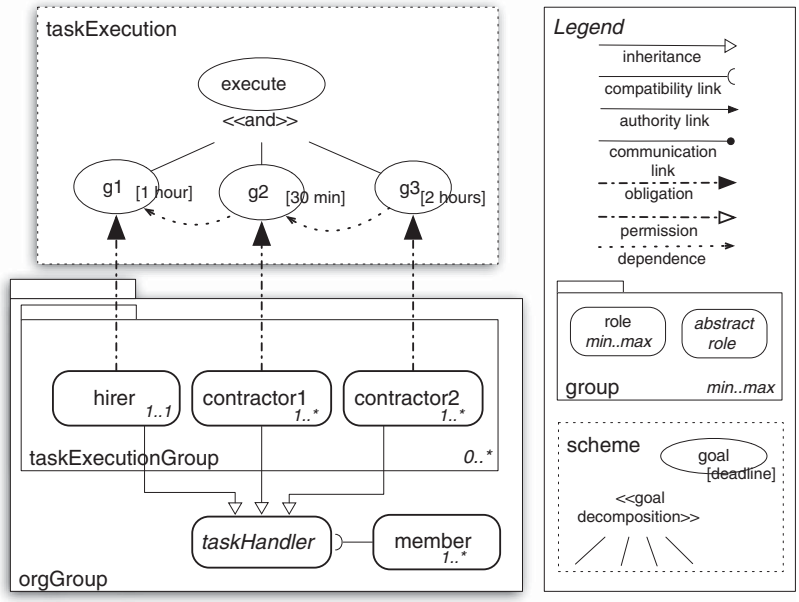


Figure 9 Simplified graphical view of the `org.xml` organization specification. Only the necessary elements for understanding the organization specification are shown

(at least one agent playing it), or `contractor2` (at least one agent playing it). An agent cannot play more than one role in this subgroup because there are no compatibility relation between the roles in the specification. These three roles inherit from the abstract role `taskHandler` the compatibility with the role `member` (i.e. an agent can play the `member` role while playing the `hirer`, `contractor1`, or `contractor2` roles). As role `taskHandler` is an abstract role, no agent can adopt it directly.

For this example, the functional specification has only one social scheme (`taskExecution`). This scheme is composed of a goal decomposition tree stating that by achieving goals `g1`, `g2`, and `g3` the goal `execute` is achieved (operator `and`). Due to their dependencies, they have to be achieved in sequence. As defined by the normative specification, the norms linking the `taskExecution` scheme to the roles of the `taskExecutionGroup` state the duties of the agents when participating in the group. In this example any agent is obliged to achieve `g1` when playing the `hirer` role, `g2` when playing the `contractor1` role, and `g3` when playing the `contractor2` role. As these goals are part of the scheme, they can only be achieved by agents that are obliged or permitted to do so through the roles they are playing.

We now describe the agents that participate in this organization. An important principle in this programming approach is that agent may achieve goals `g1`, `g2`, and `g3` required by the organization in their own way. For simplicity here, the same agent program (file `d.asl`) is used for all agents who participate in this simple organization:

```
+!g1 <- inc(2).
+!g2 <- inc(3).
+!g3 : b1(X) <- calc(X,W); set(W).
```

This program simply reuses the actions of the artefacts as defined in Section 3 to achieve the organizational goals. In this first implementation, agents are obedient, that is, they obey obligations, permissions, and prohibitions as determined by the norms of the organization[16].

[16] JaCaMo provides a library of plans that enable the agents to react to the state of the norms managed by the organization. For example, this library has a plan that creates goal *g* for the agent whenever the agent is obliged by the organization to achieve *g*. The use of this library is optional, as in other applications agents may prefer to decide weather to adopt the goal or not based on their own reasons.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

In order to deploy such an organization when the system execution is initiated, the JaCaMo project file introduced in Section 3 is enriched with a set of instructions for creating group and scheme instances, and to assign roles to agents:

```
agent agt1: d.asl {
   focus  : ws1.art1                    // NB: no initial goals anymore
}
agent agt2: d.asl {
   focus  : ws1.art1
}
agent agt3: d.asl {
   focus  : ws1.art1, ws2.art2
}
agent agt4: d.asl {
   focus  : ws1.art1, ws2.art2
}

workspace ws1 {
   artifact art1: tools.Art1(5,0)
}
workspace ws2 {
   artifact art2: tools.Art2()
}

organisation o1 : org.xml {             // org.xml contains the specification
  group torg1: orgGroup {
    players: agt1 member                // agt1 plays role member in torg1
             ...                        // same for agt2, agt3, agt4
    group tag1: taskExecutionGroup {    // tag1 subgroup of torg1
      players: agt1 hirer               // agt1 plays role hirer in tag1
               agt2 contractor1         // agt2 plays role contractor1 in tag1
               agt3 contractor2         // agt3 plays role contractor2 in tag1
               agt4 contractor2         // agt4 plays role contractor2 in tag1
      responsible-for: s1               // tag1 is responsible for scheme s1
    }

    group tag2: taskExecutionGroup {
      players: agt1 hirer
               agt2 contractor1

               agt3 contractor1
               agt4 contractor2
      responsible-for: s2
  }
  scheme s1: taskExecution
  scheme s2: taskExecution
}
```

As we can see, group `tag1` is in charge of (`responsible-for`) the execution of scheme `s1` of type `taskExecution` while group `tag2` is in charge of scheme `s2`.

Participating in the organization means for the agents to adopt roles and to commit to achieve the goals under their duties as stated by the norms. The organization management infrastructure in JaCaMo interprets organization specifications in order to coordinate collective agent activities at runtime. It will start the coordinated execution of a scheme as soon as the group instance that is in charge of it is *well formed* (i.e. to say that the minimal numbers of agents that are required to play the roles have already adopted those roles) and that the scheme is also well formed (which in this case means to say that all agents have committed to performing the corresponding goals under their responsibility). The developer does not need to implement coordination mechanisms. The infrastructure lets the agents know whenever they have goals to achieve according to the scheme specification. The scheme specification defines the order in which the goals have to be achieved. For example, the scheme specifies that goal `g2` has to be pursued after goal `g1` has been achieved (due to a `g2` dependence relation); agent `agt1` is committed to `g1` and agent `agt2` to `g2`. In this case, `agt2` has to wait for `agt1` to achieve g1 before starting to pursue `g2` . The infrastructure itself checks if new goals have been enabled to be pursued because its dependencies have been achieved by other agents. The agent committed to a goal that become enabled can then start acting on

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

that particular goal. This coordinated execution within an organization is a distributed process taking place in each of the group and scheme instances that are running in the current state of the organization.

The execution of this system follows these steps:

1. agents, environment, and organization are instantiated as defined in the project file;
2. `agt1` is obliged to commit to goal `g1` in scheme `s1`, because it plays role `hirer` in group `tag1`, which is responsible for `s1`;
3. similarly, ag1 commits to goal `g1` in scheme `s2`, `ag2` commits to goal `g2` in schemes `s1` and `s2`, `ag3` commits to goal `g2` in scheme `s2` and goal `g3` in scheme `s1`, `ag4` commits to goal `g3` in schemes `s1` and `s2`;
4. the schemes are now well formed and the goal `g1` is enabled in both;
5. agents committed to `g1` are thus obliged to achieve it and they do so;
6. since `g1` is satisfied, `g2` is now enabled and agents committed to it are obliged to achieve it and they do so using their own plans;
7. `g3` is now enabled and agents committed to it are obliged to achieve it, and they do so.

As can be seen in this example, designers can structure both the set of agents and the collective activity taking place among them. Changing the organization deployment in a JaCaMo project file or the specification in files such as `org.xml` makes it possible to change the structure and the functioning of the system respectively at the organization instance or at the organization specification levels. For example, even if the system is composed of the set of agents with the previous simple code to achieve each of the goals `g1`, `g2`, and `g3`, the collective activity may differ depending on the way the agent organization is created. By changing the scheme and dependence between goals, we are changing the collective behaviour of the agents, and we can do this without changing any code of the agents and without interrupting the system execution. In the same way, changing the minimum or maximum number of agents playing roles leads to different coordinated activities in the system by limiting or opening the possibility for more or less agents to participate in the execution process. It should also be noted that the system has a boundary to the specification of the organization: only agents being part of the instance of `orgGroup` have the possibility to participate in some task execution process by adopting either the `hirer` or `contractor1` roles in the appropriate group instance. For example, any agent that is not part of the organization does not receive information about the organization activities. Besides this possibility, organizations introduce means for coordination, expectations on the behaviour of the other agents participating in the same social scheme, modularity in terms of sets of agents (e.g. several group instances of `taskExecutionGroup` may exist), and in terms of activities (e.g. `s1` and `s2` in the JaCaMo project file following the defined schemes) attached to different groups of agents.

5.2 Changing the organization

Now that the elements to program organizations have been introduced and we have shown how an initial organization is deployed by the designer through project files, we can discuss how agents themselves can further instantiate their own organization on-the-fly[17]. For instance, an agent playing the hirer role may launch a new coordinated task for the agents belonging to its group instance by creating a new instance of the appropriate `taskExecution` scheme (line 6 below).

```
// plan to react to the event that group GrArtId is well formed,
// it has thus enough committed agents
+formationStatus(ok)[artifact_id(GrArtId)]
    : // It is me playing role hirer in that group
      .my_name(Me) & play(Me,hirer,GrName)[artifact_id(GrArtId)]
   <- .concat("sch",Me,SchName);                          // create a new scheme name
      createScheme(SchName, taskExecution, SchArtId);  // create a new scheme instance
      focus(SchArtId);                                        // focus on this scheme
      addScheme(SchName)[artifact_id(GrArtId)].          // add this new scheme to the
                                                  // responsibilities of group GrArtId
```

[17] For space reasons, we do not explain how agents adapt and redefine the *specification* of their organization, although this is possible and can be used for interesting forms of self-organizing systems.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

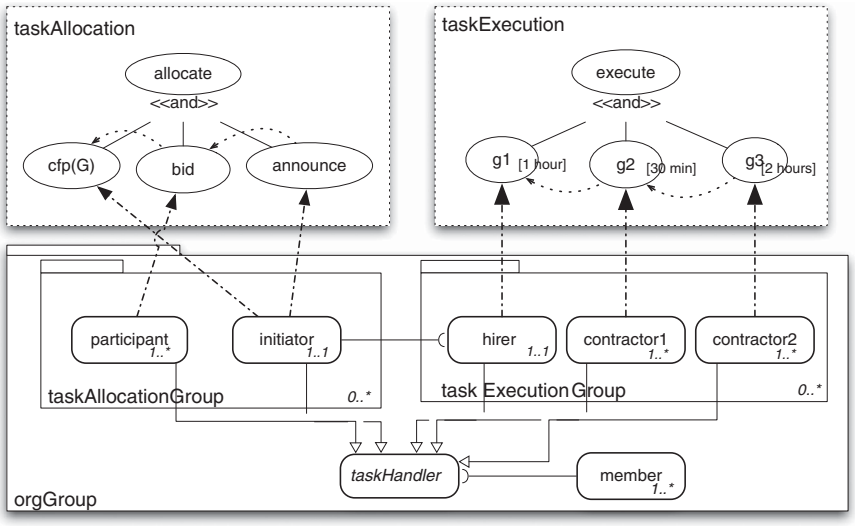


Figure 10 Simplified graphical view of the new version of the `org.xml` organization specification. Only elements necessary for understanding the organization specification are presented here. Note the compatibility relation added between `initiator` and `hirer` in order to allow an agent to play both roles in the context of the same `orgGroup` instance

Going a step further in this ability for agents to instantiate their organization, we add a new scheme to the previous organization specification. This new `taskAllocation` scheme aims at coordinating the recruitment of agents to execute tasks `g2` and `g3` following the CNP introduced in Section 4. It defines a sequence of the `cfp`, `bid`, and `announce` goals. The goal `cfp` has one argument `G` which is used to enable the agent that will achieve this goal to specify the task that is the target of the allocation process (e.g. `g1`, `g2`, or `g3`) for which agents will have to bid.

As specified by norms and by the social scheme (`taskAllocation`), any agent playing role `initiator` is obliged to manage the task allocation—that is the `cfp`—and to announce the winner (`announce`), while agents playing role `participant` are obliged to make some proposal (`bid`).

Using this new organization specification, agent `agt1` defines two group instances of `taskAllocationGroup` in which it adopts the `initiator` role. Both groups are responsible of a social scheme following the `taskAllocation` specification: one for allocating `g2` and one for allocating `g3`. When the scheme is successfully finished (i.e. goal allocate is achieved), using the plan below, agent `agt1` creates a group instance of `taskExecutionGroup` where it adopts the role hirer and asks the winner of each task allocation process (`cf. org.xml` and Figure 10) to play the `contractor1` or `contractor2` roles in the `TaskExecutionGroup` instance.

```
+!create_ex_group(SchG2,SchG3)    // SchG2 is the scheme that selected an agent for g2
                                  // SchG3 is the scheme that selected an agent for g3
   <- createGroup(nteg, taskExecutionGroup, GrArtId);           // create a new group
      ?play(_,member,PGrName);                 // retrieves the name of the root group
      setParentGroup(PGrName)[artifact_id(GrArtId)];    // linking to the parent group
      focus(GrArtId);
      adoptRole(hirer)[artifact_id(GrArtId)];
      ?winner(W2)[artifact_id(SchG2)];                  // retrieve the winner for g2
      ?winner(W3)[artifact_id(SchG3)];                  // retrieve the winner for g3
      .send(W2,achieve,enter(GrName,contractor1)).     // ask these agents to enter in
      .send(W3,achieve,enter(GrName,contractor2)).                  // the new group
```

Using organization specification and programming as shown above, agents can instantiate functional schemes (i.e. social plans), groups, and norms that coordinate and structure the activities taking place in the MAS. The task-allocation process that was hidden and hard coded in the shared coordination artefact of the previous section is made explicit and declarative. Agents have the possibility to reason about and change the protocol, the way the task should be executed, and so forth. Furthermore, openness is facilitated, that is,

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

agents can discover and understand how to participate in interactions following these coordination patterns specified in the functional, structural, and normative specifications.

5.3 Regulated organizations

Besides making it possible for agents to discover and use the groups, roles, and the coordinated activities that could take place, agent organizations also allow agents to regulate their own functioning. This section illustrates this kind of regulation and how openness can be managed through organization programming using norms. Consider that some participants in the `taskExecution` did not achieve the goals allocated to them, even though they were obliged to do so. In the execution, the scheme does not progress to the next goal in the sequence (goal `g3`), nor the execution of the scheme is finished, as the organization is waiting for the execution of other goals: goal `g2` is *enabled* and goal `g3` is *waiting* (see Figure 11).

Considering the deadline for goals `g2` and `g3`, after the corresponding elapsed time the platform that manages the current state of the scheme and the corresponding norms raises a signal informing the agents committed to the scheme that some obligations were not fulfilled.

```
+oblUnfulfilled( obligation(Ag,_,done(Sch,Goal,Ag),_ ) )[artifact_id(AId)]
```

Any agent participating in the scheme can then react to such events. For instance, one of the other agents could choose to add the violating agent into a black list and setting the outstanding goal as satisfied in order to allow the scheme to progress. For that, agents can use a `BlackList` artefact for sanctioning agents, as exemplified below. A unique instance of this artefact is used to gather the names of the agents that have been blacklisted in previous task allocation processes for being bad contractors (their names is accessible through an observable property containing a list of names of the agents known to have not fulfilled their obligations). The artefact then provides an operation addBadContractor for blacklisting agents. The following plan illustrates the use of that operation:

```
+oblUnfulfilled( obligation(Ag,_,done(Sch,Goal,Ag),_ ) )[artifact_id(AId)]
    // the hirer agent sanctions contractor agents that failed their tasks
    : .my_name(Me) & play(Me,hirer,_)
   <- .print("Participant ",Ag," didn't achieve ",Goal," on time in ", Sch);
      addBadContractor(Ag);  // add Ag in the blacklist managed by some artifact
      // make the execution of the scheme progress by marking the goal as satisfied
      .concat("goalSatisfied(",Goal,")",Cmd);
      admCommand(Cmd)[artifact_id(AId)]. // exectution of Cmd on AId
```

From this solution, we could go further and use the elements of the organization dimension to enrich the organization specification by including the explicit regulation. Briefly, in JaCaMo sanctions are not directly expressed as an element of the norm formula. However, the normative specification may contain norms that are sanctions or rewards based on the (un)fulfilment of other norms. For instance, a norm can be written to express that any agent playing the `hirer` role has the obligation to sanction (i.e. to achieve a `blacklisted` goal) any agent playing `contractor1` or `contractor2` that do not fulfil their obligations.

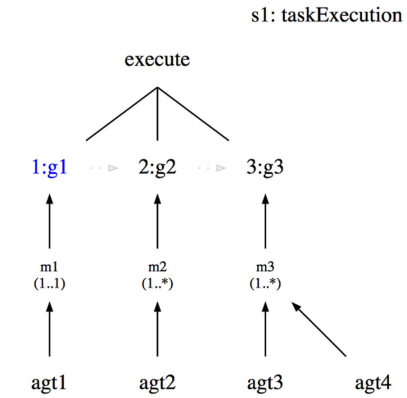
5.4 Wrapping up

In this section, we showed how to program organizations structuring and coordinating the activities of agents situated in a shared environment populated with artefacts. From these various programs, we can highlight the following features:

Boundaries and modularity: using the explicit organization model representation and the ability to adopt roles, to create groups, to create schemes, and to assign them to particular groups, we can structure and coordinate the interactions and deployment of the agents in the system. It is also possible to have several and various agent organizations in the same system. Agents can perceive this organizational shaping and represent it through beliefs, as well as to conduct some reasoning about the organizations. Agents can also act on them through the current set of actions that are provided by the underlying

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

Responsible groups: tag1



Players

- agt1 committed to m1
- agt2 committed to m2
- agt3 committed to m3
- agt4 committed to m3

goal	state	committed / achieved by	arguments	plan : dependencies
execute	waiting	[]/[]		g1, g2, g3 : {g3}
g1	satisfied	[agt1]/[agt1]		
g2	enabled	[agt2]/[]		{g1}
g3	waiting	[agt3,agt4]/[]		{g2}

Normative State

state	agent	maintenance condition	aim	deadline	done at	annotations
active	agt2	enabled(s1,g2)	**O** done(s1,g2,agt2)	now + 2 seconds		created = ["2017-10-3 13:26:14"] norm = [ngoal, [["A",agt2], ["S",s1],["D","1969-11-3 21:0:10"],["G",g2],["M",m2], ["What",done(s1,g2,agt2)]]]
unfulfilled	agt2	enabled(s1,g2)	**O** done(s1,g2,agt2)	now - 7 seconds		created = ["2017-10-3 13:26:4"] unfulfilled = ["2017-10-3 13:26:14"] norm = [ngoal, [["A",agt2], ["S",s1],["D","1969-11-3 21:0:10"],["G",g2],["M",m2], ["What",done(s1,g2,agt2)]]]

Figure 11 Organization inspector showing scheme s1 for `taskExecution`

organization management infrastructure. In JaCaMo, this infrastructure is deployed by dedicated artefacts providing distributed organization management and uniform interaction means (through agent *actions'*) between agents and their organization and environment.

Autonomy and regulation: even though organizations have norms, agent behaviour is not regimented in our approach. The MAOP approach that is promoted here allows the agents to keep their thread of decisions with respect to their own set of goals and actions as well as those coming from the organizations and their norms. However, the allowed autonomy is monitored by the organization management mechanisms that trigger events signalling violations. These events may, in turn, lead some agents to making decisions about applying appropriate sanctions.

Adaptation and reorganization: given an organization specification, the examples in this section show how agents can deploy and create various organization instances on-the-fly, and to change them by creating/deleting groups and schemes. This feature of adaptation at the organization-instance level may be extended to the organization specification also. Due to lack of space, it was not possible to

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

illustrate this feature in this paper. Nevertheless, agents do have the possibility to create and change their own organization specification at run time as they do for the organization instances.

6 Related work

The idea of adopting a multi-dimensional approach in modelling, programming, and engineering MAS was used in the Vowels decomposition paradigm (Demazeau, 1995) more than two decades ago and supported by two proposals of multi-agent platforms: Volcano (Ricordel & Demazeau, 2002) and MASK (Occello *et al.*, 2002). In particular, Vowels is a methodology for developing MAS—from analysis to deployment—based on four basic 'bricks': Agents (A), Environment (E), Interactions (I), and Organizations (O). The accompanying tools are Vowels-oriented multi-agent platforms designed to support the building process with both methodological and software tools, for various kinds of applications and domains.

According to the Vowels paradigm, given a problem to solve (or a system to simulate) in a particular domain, the user chooses the agent models, an environment model, the interaction models, and the organization models to be instantiated. Agents range from simple fine-grained automata to complex coarse-grained knowledge-based systems. The environments are most usually spatial but there is no constraint about them. Interaction structures and languages range from physics-based interactions to speech acts. Organizations range from static to dynamic ones, and follow any kind of structure from hierarchies to markets.

Vowels is guided by three principles for 'gluing the bricks'. The first principle (declarative) says that a MAS is composed of several agents, an environment, a set of possible interactions, and possibly at least one organization. The second principle (functional/computational) says that the functions that are performed by the MAS are those of the agents enriched by the ones resulting from the added value generated by the MAS itself, usually known as collective intelligence. Finally, the third principle (recursion) says that a MAS could be potentially considered as agent entities at higher-level MAS. A key point of the approach is to see the A, E, I, O bricks as not atomic components, but as elements that can be composed of several parts, and these parts are distributed across the MAS when it is deployed. An architecture definition language called Madel (Multi-Agent DEscription Language) is used to describe the main parts of the system under construction, in the transition between development and deployment (Ricordel & Demazeau, 2002).

Vowels and Volcano were among the first developments showing the advantages of a methodology and a platform for adding, mixing, and reusing models of different aspects of MAS (Agent, Environment, Interaction, Organization), in terms of a high level of reuse, and the high abstraction level of the manipulated concepts (Demazeau, 1995). Recent work has also highlighted the importance of the dimensions of MAS for software development (Baldoni *et al.*, 2016a).

The MAOP approach proposed in this paper and supported in practice by the JaCaMo platform shares the same high-level principles of that work. The main difference concerns how such principles are put in practice. While Vowels and Volcano refer explicitly to a *component-oriented* approach to developing systems, JaCaMo instead achieves the integration of the dimensions by integrating specific programming languages and programming frameworks, setting up a single set of specific first-class programming concepts, used from design to runtime. Another difference is the place for the I dimension, which in JaCaMo is spread along the A, E, and O dimensions. We have discussed the agent–agent (i.e. communicative actions) and agent–artefact interactions (i.e. perception/action), however, the interaction in the context of an organization is not explored in detail in this paper. The interaction, as considered in the MAS domain, includes the specification of interaction protocols as first-class entities that *regulate* how agents and artefacts interact (Baldoni *et al.*, 2016b; Zatelli *et al.*, 2016). This feature allows the developer to specify the interaction outside the agents, as protocol specifications. The specification of protocols outside the agent is a requirement when we want to use them to regulate agent behaviour. Another characteristic of this approach is that changes in the interaction protocols of a system requires changes in the protocol specification but not in the code of the agents. This feature is under development in the JaCaMo platform (Zatelli *et al.*, 2016).

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

Besides Vowels and Volcano, many specific agent programming languages and platforms have been developed in the last two decades[18]. Our work is related in particular to BDI-based platforms that are explicitly oriented to practical development and in particular including an explicit support not only for the agent level, but also for the other dimensions, environment, and organization in particular. In fact, to the best of our knowledge, JaCaMo is the first fully fledged platform based on programming languages for the MAS dimensions that allow programmers to explore the synergies between those dimensions of concepts and programming constructs.

Other existing approaches consider either the agent–organization dimensions or the agent–environment dimensions only, or do not have fully functional development platforms. Starting with the agent–organization dimensions, JACK (Winikoff, 2005) has the concept of team (similar to the JaCaMo concept of group). Although teams can be used to structure agent roles, in this paper we highlighted the benefits of using organizations to coordinate and regulate MAS, features not available in JACK. A closer-related work is 2OPL ('Organization-Oriented Programming Language') (Dastani *et al.*, 2008). This platform consists in a rule-based language that allows the programming of multi-agent organizations in terms of *norms* and which is meant to be exploited in synergy with agent programming languages—2APL in particular (Dastani, 2008). So far, the work has been given solid theoretical foundations but lacks a clear description of how the approach integrates with the agent level from a practical programming point of view. The main differences with respect to JaCaMo are: (i) the set of concepts in the organization dimension and (ii) the conception of the environment. The organization dimension in JaCaMo is not only concerned with regulation (through norms), it also addresses coordination with concepts like schemes and groups.

For the agent–environment dimensions, several languages consider their integration by *ad hoc* solutions (e.g. Jason and 2APL consider the environment as a Java object that provides perception to the agents). An initiative for a standard integration of these two dimension is environment interface standard (EIS) (Behrens *et al.*, 2012). Using EIS, languages like Jason (Bordini *et al.*, 2007), 2APL (Dastani, 2008), and GOAL (Hindriks, 2009) can share the same implementations of environments. Compared to JaCaMo, the main objective of EIS is to provide a common interface for different agent programming languages to interact with *existing* (software/hardware) environments—for example, using EIS to program the bot (artificial players) of a game engine platform (Hindriks *et al.*, 2011). Instead, JaCaMo—through of CArtAgO—provides the means to create (and execute, interact with) new environments, possibly distributed, based on the A&A meta-model. Besides, it allows agents to dynamically *reconfigure* them, by creating new artefacts at run-time (as shown in Section 4.2). In spite of this main difference, there are strong affinities. In EIS, the concept of *controllable entity* is introduced to represent on the environment side those entities establishing a connection between agents and the environment, by providing effectoric capabilities and sensory capabilities to agents. In JaCaMo, one can develop artefacts that are meant to provide an interface to an existing environment—these are called in literature *boundary artefacts* (Ricci *et al.*, 2006). Besides, artefacts can be used to model/represent also the 'body' of agents, so as to be perceived by other agents immersed in the same workspace.

Another important related work is the Golem agent platform (Bromuri & Stathis, 2008). It allows the programming of both cognitive agents and computational environments, structured as non-cognitive objects which are organized into 'containers'. Recent work on that approach presented the initial steps to extend the platform with norms for developing norm-governed MAS (Urovi *et al.*, 2010). Other agent programming languages also provide some support for environments and some organizational notions such as roles, but without including a fully fledged organizational model or first-class environment abstractions. We do not aim to cover all that work in this section, please refer to the surveys cited above for further details.

On the modelling side, existing work in the Agent-Oriented Software Engineering (AOSE) literature consider the use of Organization and Environment as dimensions to engineer MAS in conjunction with the agent dimension. Our work is clearly related to such approaches in terms of modelling. Much contribution has been given in that direction; interested readers can find comprehensive accounts in Weyns and Parunak (2007), Sterling and Taveter (2009), Stratulat *et al.* (2009). These include in particular work that aims at defining a

[18] For a overview of the relevant literature, see Bordini *et al.* (2005), Bordini *et al.* (2006), Dastani (2015).

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

unifying meta-model for developing MAS, integrating concepts belonging to different dimensions, for example, the FAML meta-model (Beydoun *et al*., 2009). In AOSE, agent, organization, and environment concepts are used as concepts to drive the analysis and high-level design of MAS. However, they do not explore their value from a *programming* perspective. So in this paper, while recognizing the importance of having conceptual frameworks and methodologies that make it possible to exploit agent, organization, and environment concepts to model MAS, we argue that such concepts can have a key role also at the programming level, in particular to be then exploited as *first-class abstractions* in agent-oriented programming languages and frameworks.

In this way, we aim at contributing to fill an evident gap that exists between the modelling level and the implementation level. Typically, MAS meta-models like FAML or the one described in Sterling and Taveter (2009) are rather comprehensive, including concepts that are similar or analogous to the ones that take part in the JaCaMo dimensions. For instance, the concept of *service* that appears in Sterling and Taveter (2009) or the concept of *facet* in FAML are strongly related to the notion of *artefact* which is part of the JaCaMo meta-model. The same applies for the notion of *role*. However, in Sterling and Taveter (2009), for instance, when agent programming platforms and languages are considered for building a MAS concretely, there is an apparent gap, since the considered platforms (Jason (Bordini *et al*., 2007), 2APL/2OPL (Dastani, 2008), GOAL (Hindriks, 2009), JACK (Winikoff, 2005), and JADE (Bellifemine *et al*., 2007)) are able to deal only (or almost only) with the agent level, so high-level organization and environment concepts cannot be directly mapped[19]. Also, typically low-level workarounds are used (such as modelling everything as an agent, even environment or organizational abstractions). We argue that the investigation of platforms like JaCaMo—providing an explicit programming support for the essential, orthogonal dimensions of MAS—is important also in a model-driven engineering perspective, so as to have platform-specific models that are rich enough to allow a consistent mapping of high-level concepts defined at a platform-independent level.

7 Concluding remarks

This paper presented, from a practical point of view, the MAOP approach that we developed over many years. With explanations and examples using the JaCaMo platform that implements this approach, we have highlighted the main contributions of such an approach through a coherent set of concepts and first-class programming abstractions that span over the different key dimensions of MAS: agents (and their interaction), environment, and organization. This set of abstractions can be used to assist the development of a MAS. The suggested programming approach exemplified in this paper aims at providing a level of flexibility with respect to the complexity of the systems to be developed, so that, for example, more complex concepts and abstractions such as the ones related to organizations can be introduced and used incrementally, as needed.

On the one hand, the choice of building the JaCaMo platform on top of specific languages and frameworks (Jason (Bordini *et al*., 2007), CArtAgO (Ricci *et al*., 2007), and Moise (Hubner *et al*., 2007)) provides some constraints on the level of interoperability and openness with respect to the integration with other existing agent-based models and technologies. In fact, agents implemented in different languages and technologies could be in principle integrated into a JaCaMo system by means of shared CArtAgO environments, or by means of FIPA-based communication. On one hand, from the environment point of view the EIS framework and technology (Behrens *et al*., 2012) provides a more effective support to integrate agents written for different technologies to work together within the same environment. On the other hand, our choices of specific languages made it possible to create the links among the concepts—both from the syntactic and semantic points of view—that eventually enable the synergy among the dimensions in our MAOP approach. Despite being based on specific agent-oriented programming models and technologies, the platform and the MAS developed on top of it can be adapted to support, for example,

[19] Some of the concepts used in the modelling are available in some platforms (e.g. teams in JACK and norms in 2OPL). However, the set of concepts in these platforms is limited when compared to those used in modelling (e.g. role and protocols are not available in most of those platforms).

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

extensions of the basic BDI agent architecture—by exploiting the customization and extensibility of the Jason agent platform—and the integration of existing Java-based libraries, suitably wrapped as environment artefacts.

This approach brings improvements to the programming of MAS, considering the action and percept model provided in general by agent programming languages on one hand, and the organization on the other hand. Referring to the first set of improvements, we can mention first *dynamic action repertoire*: the repertoire of actions available to agents is dynamic and can be extended/reshaped dynamically by agents themselves by creating/removing artefacts dynamically at runtime. This is an improvement with respect to existing agent programming languages, where the set of (external) actions available to an agent is given by the set of actuators that are statically defined for the agent, typically implemented in an *ad hoc* way for a particular environment. By inheriting the semantics of the operation model defined for artefacts, the expressivity of the agent action model is increased in various ways (Ricci *et al*., 2012). Typically, actions in agent programming languages are modelled as *atomic events* and this tampers with the possibility of modelling and implementing concurrent actions (overlapping in time), which is an important feature especially in MAS (Ferber & Müller, 1996), especially for coordination purposes. By mapping actions into operations, actions inherit a *process-based* semantics (Ricci *et al*., 2012), which makes it possible then to model long-term actions, overlapping in time, and then to easily design coordinating actions, providing some synchronization capability. In some BDI agent programming languages the burden of understanding if an action done by an agent succeeded or not is upon the agent itself (and the agent programmer), by reasoning about the beliefs (coming from percepts). By mapping actions into operations, the action model is extended with an explicit and well-defined notion of success/failure for actions: an action succeeds if the corresponding operation execution on the artefact side completes with success. This in general simplifies agent programming and reduces agent program size, although agents might still need to reason about beliefs to ensure successful action execution in non-deterministic environments.

Referring to the second set of improvements, the mapping of atomic actions of organization into operations on artefacts provides some important features that are important from the design and programming perspectives. The first one is *uniformity*—the same action and perception model is used to enable the interaction between agents and the environment as well as between agents and the organization, without the need for introducing specific primitives and mechanisms for interaction with the organization. The organization management infrastructure is *distributed*, in terms of collections of (interconnected) artefacts possibly belonging to different workspaces running on distinct network nodes. Furthermore, agents can change *dynamically* the shape of an organization by acting on the set of organizational artefacts available to the agents. Finally, agents may be equipped with *high-level reorganizing capabilities*, and the specifications of the organization themselves are part of the information made observable to agents by the organizational artefacts. This means that, for agents that understand the organization specification format, there is the potential for them to reason about the organizations in which they partake and therefore to change them at runtime. This allows for complex on-the-fly restructuring of computational systems to be done with high-level abstractions.

Acknowledgements

J. F. H. and R. H. B. would like to thank CNPq and CAPES for partially funding their research.

References

Baldoni, M., Baroglio, C., Calvanese, D., Micalizio, R. & Montali, M. 2016a. Towards data- and norm-aware multiagent systems. In *Engineering Multi-Agent Systems: 4th International Workshop, EMAS 2016, Singapore, Singapore, May 9–10, 2016, Revised, Selected, and Invited Papers*, Baldoni, M., Müller, J. P., Nunes, I., & Zalila-Wenkstern, R. (eds). Springer International Publishing, 22–38.

Baldoni, M., Baroglio, C., Capuzzimati, F. & Micalizio, R. 2016b. Commitment-based agent interaction in JaCaMo+. *Fundamenta Informaticae* **21**, 1001–1030.

Behrens, T., Hindriks, K., Bordini, R., Braubach, L., Dastani, M., Dix, J., Hübner, J. & Pokahr, A. 2012. An interface for agent-environment interaction. In *Proceedings of Eighth International Workshop on Programming Multi-Agent Systems (ProMAS@AAMAS 2010)*, Collier, R., Dix, J., & Novák, P. (eds). LNCS **6599**, 139–158. Springer.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

Bellifemine, F. L., Caire, G. & Greenwood, D. 2007. *Developing Multi-Agent Systems With JADE*. Wiley Series in Agent Technology. John Wiley & Sons.

Beydoun, G., Low, G., Henderson-Sellers, B., Mouratidis, H., Gomez-Sanz, J. J., Pavon, J. & Gonzalez-Perez, C. 2009. Faml: A generic metamodel for MAS development. *IEEE Transactions on Software Engineering* **35**, 841–863.

Boissier, O., Bordini, R., Hübner, J. F., Ricci, A. & Santi, A. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* **78**(6), 747–761.

Boissier, O., Hübner, J. F. & Sichman, J. S. 2007. Organization oriented programming from closed to open organizations. In *Engineering Societies in the Agents World VII (ESAW 06)*, O'Hare, G., O'Grady, M., Dikenelli, O. & Ricci, A. (eds). LNCS **4457**, 86–105. Springer-Verlag.

Bordini, R. H., Braubach, L., Dastani, M., Seghrouchni, A. E. F., Gomez-Sanz, J. J., Leite, J., O'Hare, G., Pokahr, A. & Ricci, A. 2006. A survey of programming languages and platforms for multi-agent systems. *Informatica* **30**(1), 33–44.

Bordini, R. H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds) 2005. *Multi-Agent Programming: Languages, Platforms, and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer.

Bordini, R. H., Hübner, J. F. & Wooldrige, M. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. John Wiley & Sons.

Bratman, M. E. 1987. *Intention, Plans, and Practical Reason*. Harvard University Press.

Bromuri, S. & Stathis, K. 2008. Situating cognitive agents in GOLEM. In *Engineering Environment-Mediated Multi-Agent Systems*, Weyns, D., Brueckner, S. & Demazeau, Y. (eds). LNCS **5049**, 115–134. Springer Berlin/ Heidelberg.

Dastani, M. 2008. 2APL: a practical agent programming language. *Autonomous Agent and Multi-Agent Systems* **16**, 241–248.

Dastani, M. 2015. Programming multi-agent systems. *Knowledge Engineering Review* **30**(4), 394–418.

Dastani, M., Grossi, D., Meyer, J.-J. & Tinnemeier, N. 2008. Normative multi-agent programs and their logics. In *KRAMAS-08, Proceedings*.

Demazeau, Y. 1995. From interactions to collective behaviour in agent-based systems. In *Proceedings of the 1st European Conference on Cognitive Science*, 117–132.

Demazeau, Y. & Rocha Costa, A. C. d. 1996. Populations and organizations in open multi-agent systems. In *PDAI 96 – 1st National Symposium on Parallel and Distributed AI*.

Ferber, J. & Müller, J.-P. 1996. Influences and reaction: a model of situated multiagent systems. In *2nd International Conference on Multi-Agent Systems (ICMAS'96)*.

Foundation for Intelligent Physical Agents (FIPA) 2002. Contract net interaction protocol specification. Technical report.

Hindriks, K. V. 2009. Programming rational agents in GOAL. In *Multi-Agent Programming*, Bordini, R. H., Dastani, M., Dix, J. & Seghrouchni, A. E. F. (eds), 119–157. Springer.

Hindriks, K. V., van Riemsdijk, B., Behrens, T., Korstanje, R., Kraayenbrink, N., Pasman, W. & de Rijk, L. 2011. Unreal goal bots. In *Agents for Games and Simulations II: Trends in Techniques, Concepts and Design*, Dignum, F. (ed.), 1–18. Springer Berlin Heidelberg.

Hübner, J. F., Sichman, J. S. & Boissier, O. 2002. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In *Proceedings of the 16th Brazilian Symposium on Artificial Intelligence (SBIA'02)*, 118–128.

Hübner, J. F., Sichman, J. S. & Boissier, O. 2007. Developing organised multi-agent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering* **1**(3/4), 370–395.

Labrou, Y., Finin, T. & Peng, Y. 1999. Agent communication languages: the current landscape. *IEEE Intelligent Systems* **14**(2), 45–52.

Lemaître, C. & Excelente, C. B. 1998. Multi-agent organization approach. In *Proceedings of II Iberoamerican Workshop on DAI and MAS,* Garijo, F. J. & Lemaître, C. (eds), 7–16.

Occello, M., Baeijs, C., Demazeau, Y. & Koning, J.-L. 2002. Mask: an AEIO toolbox to develop multi-agent systems. *Knowledge Engineering and Agent Technology, IOS Series on Frontiers in AI and Applications* **63**, 64.

Omicini, A., Ricci, A. & Viroli, M. 2008. Artifacts in the A&A meta-model for multi-agent systems. *Journal of Autonomous Agents and Multi-Agent Systems* **17**(3), 432–456.

Rao, A. S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, Van de Velde, W. & Perram, J. (eds), Lecture Notes in Artificial Intelligence **1038**, 42–55.Springer-Verlag.

Ricci, A., Piunti, M., Acay, L. D., Bordini, R. H., Hübner, J. F. & Dastani, M. 2009. Integrating heterogeneous agent programming platforms within artifact-based environments. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10–15*, Sierra, C., Castelfranchi, C., Sichman, J. S., & Decker, K. S. (eds), 225–232.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press

Ricci, A., Santi, A. & Piunti, M. 2012. Action and perception in agent programming languages: from exogenous to endogenous environments. In *Programming Multi-Agent Systems. ProMAS 2010. Lecture Notes in Computer Science*, Collier R., Dix J. & Novák P. (eds). Springer.

Ricci, A., Viroli, M. & Omicini, A. 2006. Programming mas with artifacts. In *Programming Multi-Agent Systems: Third International Workshop, ProMAS 2005, Utrecht, The Netherlands, July 26, 2005, Revised and Invited Papers*, Bordini, R. H., Dastani, M. M., Dix, J., & El Fallah Seghrouchni, A. (eds), 206–221. Springer Berlin Heidelberg.

Ricci, A., Viroli, M. & Omicini, A. 2007. CArtAgO: a framework for prototyping artifact-based environments in MAS. In *Environments for MultiAgent Systems III*, *3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers*, Weyns, D., Parunak, H. V. D., & Michel, F. (eds). LNAI **4389**, 67–86. Springer.

Ricordel, P. & Demazeau, Y. 2002. VOLCANO: a vowels-oriented multi-agent platform. In *Proceedings of the International Conference of Central Eastern Europe on Multi-Agent Systems (CEEMAS'01)*, Dunin-Keplicz, B. & Nawarecki, E. (eds). LNAI **2296**, 253–262. Springer Verlag.

Rocha Costa, A. C. d. & Dimuro, G. 2009. A minimal dynamical organization model. In *Multi-Agent Systems: Semantics and Dynamics of Organizational Models, chapter XVII*, Dignum, V. (ed.). IGI Global, 419–445.

Sterling, L. & Taveter, K. 2009. *The Art of Agent-Oriented Modeling*. The MIT Press.

Stratulat, T., Ferber, J. & Tranier, J. 2009. MASQ: towards an integral approach to interaction. In *AAMAS (2009)*, 813–820.

Urovi, V., Bromuri, S., Stathis, K. & Artikis, A. 2010. Initial steps towards run-time support for norm-governed systems. In *Coordination, Organizations, Institutions, and Norms in Agent Systems VI*, Lecture Notes in Computer Science **6541**, 268–284. Springer.

Weyns, D., Omicini, A. & Odell, J. J. 2007. Environment as a first-class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems* **14**(1), 5–30Special Issue on Environments for Multi-Agent Systems.

Weyns, D. & Parunak, H. V. D. (eds) 2007. *Special Issue on Environments for Multi-Agent Systems, volume 14 (1) of Autonomous Agents and Multi-Agent Systems*. Springer.

Winikoff, M. 2005. Jack intelligent agents: An industrial strength platform. In *Multi-Agent Programming: Languages, Platforms, and Applications. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations*, Bordini, R. H., Dastani, M., Dix, J. & El Fallah Seghrouchni, A. (eds). Springer, 175–193.

Zatelli, M. R., Ricci, A. & Hübner, J. F. 2016. Integrating interaction with agents, environment, and organisation in JaCaMo. *International Journal of Agent-Oriented Software Engineering (IJAOSE)* **5**(2, 3), 266–302.

https://doi.org/10.1017/S026988891800005X Published online by Cambridge University Press