# Social Relationships for Designing Agent Interaction in JADE

Matteo Baldoni, Cristina Baroglio and Federico Capuzzimati
Dipartimento di Informatica
Università degli Studi di Torino
c.so Svizzera 185, I-10149 Torino (Italy)
Email: {matteo.baldoni,cristina.baroglio,federico.capuzzimati}@unito.it

***Abstract*—Current agent platforms do not provide agents the means for reasoning about expected behaviours during interactions. This lack is due to the absence of design primitives to explicitly shape interaction patterns as first-class resources. This work presents 2COMM4JADE, a framework based on JADE and CArtAgO platforms that allows definition of social relationships among parties, represented by social commitments, decoupled from the agent design itself.**

## I. Introduction and Motivation

*Multi-Agent Systems* (MAS) represent a preferred choice for building complex systems, where the autonomy of each component represent a major requirement. Agent-oriented software engineers can choose from a substantial number of agent platforms [19], [8], [14], [5]. Platforms and frameworks like JADE [6], TuCSoN [22], DESIRE [9], JIAC [27], all provide *coordination mechanisms* and *communication infrastructures* [8]. We claim, however, that none of the current infrastructures for MAS is adequate to really preserving autonomy, and do not fit the high degree of decoupling often required by such systems. This can be obtained only via a clear separation of the agent specification from the coordination specification, that avoids any "hard-coding" of the regulations of the system inside the activities of the participants. To this aim, we propose to explicitly represent interaction patterns among agents in terms of normatively defined *social relationships*; the normative characterization is grounded on *commitments*, which feature a social and observational semantics [25]; in order to represent the coordination requirements we propose to rely on commitment-based protocols [29].

The framework we propose, 2COMM4JADE, uses JADE as basic agent platform, which provides a FIPA compliant communication framework and an agent-developing middleware. JADE does not offer an agent programming language, but it is a sound, well-established and industry-adopted agent framework. Although the first release was presented in 2007, a dynamic community helps developers in updating and evolving the platform, providing new functionalities and capabilities. Thus, it supplies a valuable starting ground.

In order to reify the social relationships we rely on the Agents & Artifacts meta-model (A&A) [28], [21], which provides abstractions for environments and artifacts, that can be acted upon, observed, perceived, notified, and so on. When embodied inside artifacts, social relationships can be examined by the agents (to take decisions about their behavior), as advised in [11], used (which entails that agents accept the corresponding regulations), constructed, e.g., by negotiation, specialized, composed, and so forth. Last but not least, the use of artifacts enables the implementation of monitoring functionalities for verifying that the on-going interactions respect the commitments and for detecting violations and violators.

Summarizing, this work proposes to introduce in MAS an explicit notion of social relationships, captured as commitments (Section II). Social relationships are actual resources, implemented through artifacts, that are made available to the agents, and are first-class entities of the model, as well as agents. The framework 2COMM4JADE (Section III) realizes the proposal based on an extension of JADE and of CArtAgO. We show the impact of the proposal on programming by means of an example (Section IV).

## II. Modeling Social Relationships

We propose to explicitly represent social relationships among the agents. By social relationships we mean normatively defined relationships, and the expected patterns of interaction, between two or more agents, resulting from the enactment of *roles*, and subject to *social control*. We envisage both agents and social relationships as first-class entities that interact in a bi-directional manner. Social relationships are created by the execution of *interaction protocols* and provide expectations on the agents' behaviour. It is, therefore, necessary to provide the agents the means to create and to manipulate, and to observe, to monitor, to reason and to deliberate on social relationships so to take proper decisions about their behaviour. We do so by exploiting *properly defined artifacts*, that reify both *interaction protocols*, defined in terms of social relationships, and the *sets of social relationships*, created during protocol execution. Such artifacts are made available to agents as resources.

The aim of our proposal is to clearly *decouple* the interaction design from the agent design, using artifacts to encode the coordination logic. Commitment-based protocols provide a means of coordination, based on the *notification* of social events, e.g. the creation of a commitment. Interacting agents use artifacts to coordinate and interact, depending on the roles they play and on their objectives. Such a decoupling avoids interaction logics hard–coded in agent programs, that can lead to an increasing developing effort and a difficult maintenance of the system, especially when it constitutes a society of autonomous entities belonging to different organizations. Instead, relying on commitment-based protocols allows a modular definition of components of the system and of how

they interact, using the notion of commitment to define the shape and the evolution of interaction patterns.

A commitment [24] is represented with the notation $\mathsf{C}(x, y, r, p)$, capturing that the agent $x$ commits to the agent $y$ to bring about the consequent condition $p$ when the antecedent condition $r$ holds. Antecedent and consequent conditions generally are conjunctions or disjunctions of events and commitments. When $r$ equals $\top$, we use the short notation $\mathsf{C}(x, y, p)$ and the commitment is said to be *active*. Commitments have a *regulative* nature, in that debtors are expected to behave so as to satisfy the engagements they have taken. This practically means that an agent is expected to behave so as to achieve the consequent conditions of the active commitments of which it is the debtor.

Agents share a *social state* that contains commitments, and update it as a result of performing their activities. Specifically, the manipulation of commitments is done by means of the standard operations *create*, *cancel*, *release*, *discharge*, *assign*, *delegate*. As in [24], we postulate that discharge is performed concurrently with the actions that lead to the given condition being satisfied and causes the commitment to not hold. Delegate and assign transfer commitments respectively to a different debtor and to a different creditor. For details see [24], [29], [10].

The next question to answer is how to integrate, inside the outlined agent and artifact setting, the rules that regulate the process by which social relationships are created and manipulated. In our proposal, this is done by relying on the notion of *commitment-based interaction protocol*. This kind of protocol consists of a set of actions (process activities), whose semantics is shared, and agreed upon, by all of the participants to the interaction [29], [10]. The semantics of the social actions is given in terms of the operations which allow the modification of the social state and that have already been described.

From an organizational perspective, a protocol is structured into a set of *roles*. Roles and agents are different entities, and we assume that roles cannot live autonomously: they exist in the system in view of the interaction, because agents, for interacting, use artifacts and execute actions on them. We follow the ontological model for roles proposed in [7], and brought inside the object-oriented paradigm in [4], which is characterized by three aspects: (1) *Foundation*: a role must always be associated with the institution it belongs to and with its player; (2) *Definitional dependence*: the definition of the role must be given inside the definition of the institution it belongs to; (3) *Institutional empowerment*: the actions defined for the role in the definition of the institution have access to the state of the institution and of the other roles, thus, they are called powers; instead, the actions that a player must offer for playing a role are called requirements. The agents that will be the *role players* become able to perform protocol actions, that are *powers* offered by a specific role and whose execution affect the social state. On the other hand, they need to satisfy the related *requirements*: specifically, in order to play a role an agent needs to have the capabilities of satisfying the related commitments – capabilities which can be internal of the agent or supplied as powers as well.

Following [12], we need social relationships to *be recognized* as having a normative force (so that they will provide social expectations on the stakeholders' behaviour), to *be accepted* explicitly by the participants to the interaction, to *be inspected* so as to allow stakeholders to decide whether conforming to them. To this aim, we propose to explicitly model social relationships as *resources*, that are made available to the interacting peers, in the very same way as other kinds of resources; a resource that has to be inspectable, and therefore can not be modeled as an agent. To solve this issue, we adopt the *Agents and Artifacts* (A&A) meta-model [28], [21], that extends the agent paradigm with another primitive abstraction, the *artifact*. An artifact is a computational, programmable system resource, that can be manipulated by agents, residing at the same abstraction level of the agent abstraction class. For their very nature, artifacts can encode a mediated, programmable and observable means of communication and coordination between agents.

We interpret the fact that an agent uses an artifact as the explicit acceptance, by the agent, of the regulation encoded by that artifact, and modeled by the interaction protocol that the artifact reifies. This is an important aspect because it allows the interacting parties to perform *practical reasoning*, based on expectations: participants expect that the debtors of commitments behave so as to satisfy the corresponding consequent conditions; when this does not happen, a violation is raised. Relying on artifacts allows also other kinds of manipulation, including (but not limited to) an agent playing a role in the interaction, and an agent observing the interaction that is being carried on.

Summarizing, we claim that an agent-based framework should satisfy the following requirements: 1) Explicit representation of the social relationships, created and evolving along the interaction. 2) Social relationships should be first-class objects, which can be used for programming the agent behavior. 2COMM4JADE fulfills both.

## III. 2COMM4JADE: A Commitment-based Infrastructure for Social Relationships

After providing the background, we describe the developed implementation framework, that we named 2COMM4JADE. The focus is to provide adequate support for programming social relationships by exploiting a declarative, interaction-centric approach and by relying on existing technologies as far as possible. Specifically, the implementation framework combines two well-known platforms: JADE [6] and CArtAgO [23]. JADE agents represent stakeholders, commitment-based interaction protocols are realized as CArtAgO artifacts.

JADE is a popular and industry-adopted agent framework. It offers to developers a Java middleware that is FIPA-compliant [17]. Its robustness and well-proven reliability makes JADE a preferred choice in developing MAS. A JADE-based system is composed of one or more *containers*, each grouping a set of agents in a logical *node* and representing a single JADE runtime. The overall set of containers is called a *platform*, and can spread across various physical hosts. The resulting architecture hides the underlying layer, allowing support for different low-level frameworks (JEE, JSE, JME, etc.). JADE provides communication and infrastructure services, allowing agents, that have been deployed in different containers, to discover and interact with each other.

CArtAgO is a framework based on the A&A meta-model [28], [21]. It extends the agent programming paradigm with the first-class entity of *artifact*: a resource that an agent can use, and that models working environments. It provides a way to define and organize *workspaces*, that are logical groups of artifacts, that can be joined by agents at runtime. The environment is itself programmable and encapsulates services and functionalities. CArtAgO provides an API to program *artifacts* that agents can use, regardless of the agent programming language or the agent framework used. This is possible by means of the *agent body* metaphor: CArtAgO provides a native agent entity, which allows using the framework as a complete MAS platform as well as it allows mapping the agents of some platform onto the CArtAgO agents, which, in this way, becomes a kind of "proxy" in the artifacts workspace. The former agent is the mind, that uses the CArtAgO agent as a body, interacting with artifacts and sensing the environment. An agent interacts with an artifact by means of public *operations*, which can be equipped with *guards*: conditions that must hold in order for operations to produce their effects. When guards do not hold, actions can still be performed but without consequences.

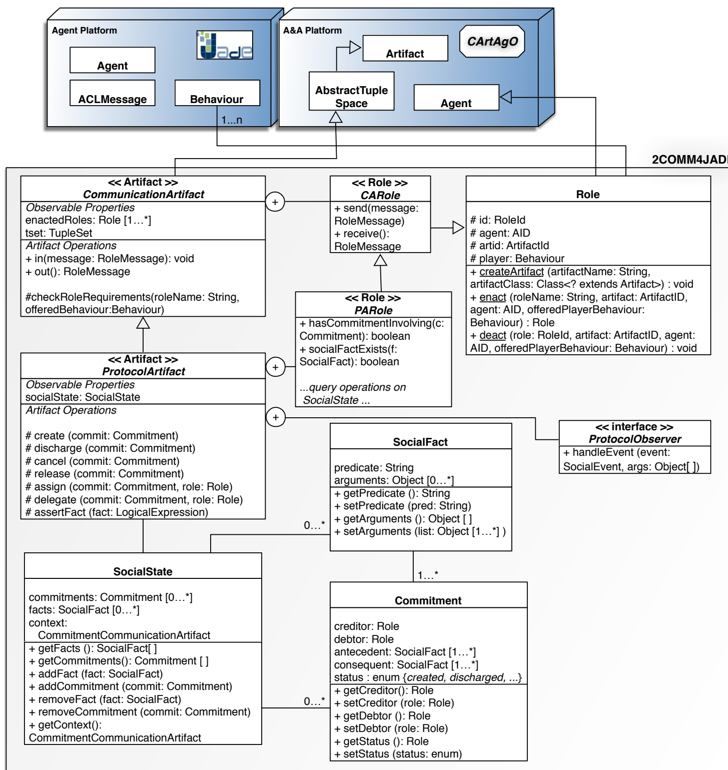


Fig. 1. Excerpt of the UML class diagram of 2COMM4JADE.

2COMM4JADE is organized as follows. JADE supplies standard agent services, i.e. message passing, distributed containers, naming and yellow pages services, agent mobility. When needed, an agent can enact a protocol role, which provides a set of operations by means of which agents participate in a mediated interaction session. Protocol roles are provided by *communication artifacts*, that are implemented by means of CArtAgO. Each communication artifact corresponds to a specific protocol enactment and maintains an own social state and an own communication state.

Figure 1 reports an excerpt of the 2COMM4JADE UML class diagram[1]. Let us get into the depths of the implementation:

- *CommunicationArtifact* (CA for short) provides the basic communication operations *in* and *out* for allowing mediated communication. CA extends an abstract version of the *TupleSpace* CArtAgO artifact: briefly, a blackboard that agents use as a tuple-based coordination means. In and out are, then, operations on the tuple space. CA also traces who is playing which role by using the property *enactedRoles*.
- Class *Role* extends the CArtAgO class *Agent*, and contains the basic manipulation logic of CArtAgO artifacts. Thus, any specific role, extending this supertype, will be able to perform operations on artifacts, whenever its player will decide to do so. Role provides static methods for creating artifacts and for *enacting/deacting* roles. This is done by passing a reference to the JADE agent behaviour that will actually play the role.
- The class *CARole* is an inner class of CA and extends the Role class. It provides the *send* and *receive* primitives, by which agents can exchange messages. Send and receive are implemented based on the *in* and *out* primitives provided by CA.
- *ProtocolArtifact* (PA for short) extends CA and allows modeling the social layer with the help of commitments. It maintains the state of the on-going protocol interaction, via the property *socialState*, a store of social facts and commitments, that is managed only by its container artifact. This artifact implements the operations needed to manage commitments (create, discharge, cancel, release, assign, delegate). PA realizes the commitment life-cycle and for the assertion/retraction of facts. Operations on commitments are realized as *internal operations*, that is, they are not invokable directly: the protocol social actions will use them as primitives to modify the social state. We refer to modifications occurred to the social state as *social events*. Being an extension of CA, PA maintains two levels of interaction: the social one (based on commitments), and the communication one (based on message exchange).
- The class *PARole* is an inner class of PA and extends the CARole class. It provides the primitives for *querying the social state*, e.g. for asking the commitments in which a certain agent is involved, and the primitives that allow an agent to become, through its role, an *observer of the events* occurring in the social state. For example, an agent can query the social state to verify if it contains a commitment with a specific condition as consequent, via the method `existsCommitmentWithConsequent (InteractionStateElement el)`.

[1]The source files of the system and examples are available at the URL http://di.unito.it/2COMM.

Alternatively, an agent can be notified about the occurrence of a social event, provided that it implements the inner interface *ProtocolObserver*. Afterwards, it can start observing the social state. PARole also inherits the communication primitives defined in CARole.

- The class *SocialFact* represents a fact of some relevance for the ongoing interaction, that holds in the current state of interaction. A social fact is asserted for tracking the execution of a protocol action. Actions can have additional effects on the social state; in this case, corresponding social facts are added to it.

In order to specify a *commitment-based interaction protocol*, it is necessary to extend PA by defining the proper social and communicative actions as operations on the artifact itself. Actions can have guards that correspond to *context preconditions*: each such condition specifies the context in which the respective action produces the described social effect. Since we want agents to act on artifacts only through their respective roles, when defining a protocol it is also necessary to create the roles. We do so by creating as many extensions of PARole as protocol roles. These extensions are realized as inner classes of the protocol: each such class will specify, as methods, the powers of a role. Powers allow agents who play roles to actually execute artifact operations. The typical schema will be:

```
public class MyProtocolArtifact
  extends ProtocolArtifact {
  // ...
  static {
    addEnabledRole("Role1", Role1.class);
    addEnabledRole("Role2", Role2.class);
    // ...
  }
  // MY PROTOCOL ARTIFACT OPERATIONS
  @OPERATION
  public void op1(...) {
    // prepare a message, if needed; in that case,
    send(message);
    // modify the social state,
    // e.g. create commitment, update commitment
  }
  // ...
  // INNER CLASSES for ROLES
  public class Role1 extends PARole {
    public Role1(Behaviour player, AID agent) {
      super("Role1", player, agent);
    }
    // define social actions for Role1
    public void action1(...) {
      doAction(this.getArtifactId(),
        new Op("op1", ..., getRoleId()));
    }
    // ...
  }
  public class Role2 extends PARole {
    // ...
  }
  // ...
}
```

Let us, now, consider Figure 2, which sketches how the agent model reacts to a new social event occurrence. For a JADE agent to play a role, one of its behaviours must implement the method `handleEvent`, which receives the event just occurred in the social state. The agent programmer

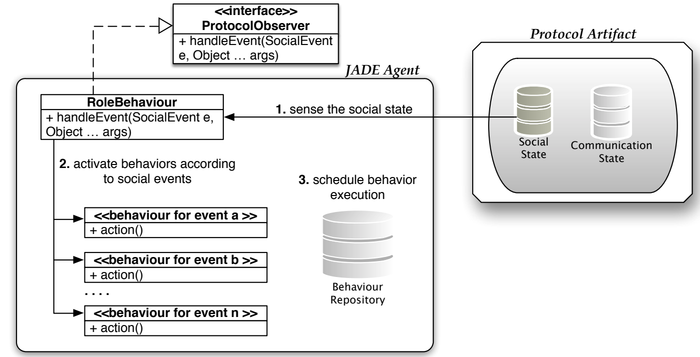


Fig. 2. 2COMM4JADE event handling schema.

will simply implement the logic for handling that event, adding proper behaviour(s) to the agent's behaviour repository. When scheduled, the behaviour will be executed, and the event handled. A *methodological schema* for handling a social event follows.

**Algorithm 1:** Methodological Schema for Handling Social Events.

**Data**: Social event $se$ notified to agent $a$
**Data**: Set $S$ of social events of interest
**Result**: Add a behaviour to the behaviour set of $a$
Check whether $se$ involves a commitment $c$ which has $a$ as debtor or creditor;
**if** $se \in S$ **then**
  $b$ = behaviour that handles the occurred modification of $c$;
  add $b$ to the set of behaviours of $a$;
**end**

The proposed algorithm explains how to use the notifications performed by the protocol artifact to the agent. Depending on which social event occurs, i.e. on which commitment is modified and how (e.g. added, discharged, detached), a specific behaviour is added and scheduled for execution, competing with others already present in the behaviour set. The agent designer can choose which social events are to be tackled by the agent. The following is the pseudo-code of an example implementation that agrees with the schema:

```
public class MyBehaviour extends
   SomeJadeBehaviour implements ProtocolObserver {
  [ ... ]
  public void action() {
    ArtifactId art = Role.createArtifact
         (myArtifactName, MyArtifact.class);
    myRole = (SomeRole) (Role.enact
       (MyArtifact.ROLE_NAME, art, this,
        myAgent.getAID()));
    myRole.startObserving(this);
    // add the initial behaviour of the agent
  }
  public void handleEvent(SocialEvent e,
              Object... args) {
    SETemplate t = new SETemplate(
        myRole.getRoleId());
    SETemplate t2 = new SETemplate(
        myRole.getRoleId());
    t.iAmDebtor().commitIsDetached()
```

```
20          .consequentMatch(...);
21      t2.iAmCreditor().commitIsConditional()
22          .antecedentMatch(...);
23      if (t.match(e) {
24        myAgent.addBehaviour(...);
  // a behaviour to handle the case
25      } else if (t2.match(e)) {
26          myAgent.addBehaviour(...);
  // a behaviour to handle another case
27      } else
28          // ...
  // behaviours for different cases
29      }
30
31    }
32  }
```

The agent behaviour implements *ProtocolObserver*, thus allowing the agent to be notified about social events. This will be done after the agent executes the method *startObserving*, specifying which artifact it requires to observe. The implementation of *ProtocolObserver* requires, in turn, the implementation of the method *handleEvent*. Here, it is possible to test the kind of event that was notified. If it is a commitment, it is possible to further check specific conditions of interest on it, including its state, the identity of its debtor and/or creditor, the antecedent or consequent condition (lines 19-22). The agent will, then, add appropriate behaviours to handle the detected situation. A template-based matching mechanism for social events is provided (class `SETemplate`, lines 15–18) used by programmer in order to specify matching conditions. Each template class method returns `this`, thus compacting the code for construction of complex conditions simply using the standard method dot notation.

The handler represents the classical agent *sense-plan-act cycle*, rephrased into "sense the social state", "activate behaviors according to social events", "schedule behavior execution". Notice that this mechanism represents an agent-oriented declination of callbacks. The agent paradigm forbids to make use of pure method invocation on the agent, that is autonomous by definition. Instead, the agent designer provides a collection of behaviours in charge for handling the different, possible evolutions of the social state, that are scheduled for execution when the corresponding condition happens. For example, a specific behaviour can be added when a new commitment is added, and the creditor of that commitment is the agent; or when a social event is added to the social state. This way, an intuitive and social-based programming schema is provided to agent developers.

## IV. Programming Social Relationships: an Example

*FinancialMAS* is an example application of 2COMM4JADE. Inspired by [13], it allows financial interactions between three categories of stakeholders: (1) investors, (2) financial promoters and (3) banks. Investors have the goal to place investments; promoters to propose and finalize investment contracts, on behalf of a bank and on the basis of current investor's profile; banks to finalize contracts placed by promoters.

Let us focus on the Contract Net Protocol (CNP for short) [17], which is used inside FinancialMAS to manage the interaction between the Investor and the Financial Promoter when the former looks for some investment. We adopt the following CNP formulation:

> *cfp* **causes** create(*C(i, p, propose, accept* $\vee$ *reject)*)
> *accept* **causes** *none*
> *reject* **causes** release(*C(p, i, accept, done* $\vee$ *failure)*)
> *propose* **causes** create(*C(p, i, accept, done* $\vee$ *failure)*)
> *refuse* **causes** release(*C(i, p, propose, accept* $\vee$ *reject)*)
> *done* **causes** *none*
> *failure* **causes** *none*

where $i$ stands for the role *Initiator* and $p$ for *Participant*. The instance used by FinancialMAS will tie the Investor to the $i$ and the Financial Promoter to $p$. Initiator supplies its player the powers *cfp* (call for proposal), *accept*, and *reject*. The first allows the initiator to ask participants for proposals for solving a task of interest. If a proposal is chosen, action *accept* notifies the winner and all other proposals are rejected. The role participant supplies its player the powers *propose*, *refuse*, *done*, and *failure*. Action *propose* allows a participant to supply a solution for a task, action *refuse* allows declining the invitation to send a proposal. If a proposal is accepted, the winning participant is expected to execute the task and either provide the result by means of the action *done* or communicate its failure. Powers allow agents to affect the social state. For instance, when an agent playing the role Initiator executes *cfp*, the social state is modified by creating the commitment *C(i, p, propose, accept* $\vee$ *reject)*. This addition binds $i$ to either accept or reject a proposal, if one is received. The agent is free to decide not only which course of action to take but also how to realize acceptance or rejection.

Let us see how CNP is realized in 2COMM4JADE. The class CNP extends ProtocolArtifact and implements as artifact operations all the protocol actions. In the sketch below we detail, for the sake of brevity, only the action *cfp*. Lines 6–10 represent the construction and sending of a call for proposals, which will be inserted in the blackboard and made available to participants. The recipient of the message is left generic because the Initiator may not know its peers, who may join even later in the future. Lines 15–19 modify the social state by adding commitments and by stating that the event *cfp* has occurred. Lines 27–42 defines the role in terms of its powers. Here (lines 30–33), in fact, the power *cfp* is defined in terms of the homonym artifact operation.

```
1  public class CNP extends ProtocolArtifact {
2    // ...
3    @OPERATION
4    public void cfp(Task task, RoleId initiator) {
5      // send message with the task
6      RoleMessage cfp = new RoleMessage();
7      RoleId dest = new RoleId(PARTICIPANT_ROLE,
8            RoleId.GENERIC_ROLE);
9      cfp.setContent(task);
10     cfp.setRoleSender(initiator);
11     cfp.setRoleReceiver(dest);
12     cfp.setPerformative(ACLMessage.CFP);
13     send(cfp);
14     // update the social state
15     createAllCommitments(new Commitment
16       (initiator, dest, "propose",
17       new CompositeExpression(LogicalOperatorType.OR,
18         new Fact("accept"), new Fact("reject"))));
19     assertFact(new Fact("cfp", initiator, task));
20   }
```

```
21   @OPERATION
22   public void propose(Proposal prop,
23       RoleId participant, RoleId initiator) {
24     // ...
25   }
26   // ...
27   // Role classes
28   public class Initiator extends PARole {
29     // ...
30     public void cfp(Task task) {
31       doAction(this.getArtifactId(),
32         new Op("cfp", task, getRoleId()));
33     }
34     // ...
35   }
36   public class Participant extends PARole {
37     public void propose(Proposal proposal,
38         RoleId proposalSender) {
39       // ...
40     }
41     // ...
42   }
43 }
```

The following, instead, is a sketch of the definition of a JADE agent, playing the role Initiator by using 2COMM4JADE. In this example, the artifact is created by the agent which, then, enacts the role of interest. At line 12, the agent uses the power *cfp* through the role it plays. Finally, at line 14, it inspects if someone committed to accomplish the task, and reads the proposals in the tuple space in order to make a choice.

```
1  public class InitiatorAgent extends Agent {
2    // ...
3    public class InitiatorBehaviour
4        extends OneShotBehaviour {
5      // ...
6      public void action() {
7        ArtifactId art = Role.createArtifact
8          (ARTIFACT_NAME, CNPArtifact.class);
9        initiator = (Initiator)(Role.enact
10         (CNPArtifact.INITIATOR_ROLE, art, this,
11          myAgent.getAID()));
12       initiator.cfp(t);
13       // ...
14       boolean proposalPerformed =
15         initiator.existsCommitmentWithConsequent(
16         new CompositeExpression(
17           LogicalOperatorType.OR,
18           new Fact("done"), new Fact("failure"));
19       // ...
20     }
21   }
22 }
```

More interesting is to exploit the feature of 2COMM4JADE, which allows roles to register for the notification of social events. Thanks to the event handler, modifications to the social state will add specific behaviours to the agents, aimed at tackling the case. The code is organized by following the schema described in Algorithm 1. Figure 3 contains the UML diagram for this solution.

```
1  public abstract class InitiatorBehaviour
2         extends OneShotBehaviour
3         implements ProtocolObserver {
4    public String artifactName;
5    protected Initiator initiator;
6    public abstract Behaviour
7        commitToAcceptOrRejectIfPropose();
8    public abstract Behaviour
9        satisfyCommitToAcceptOrReject();
```

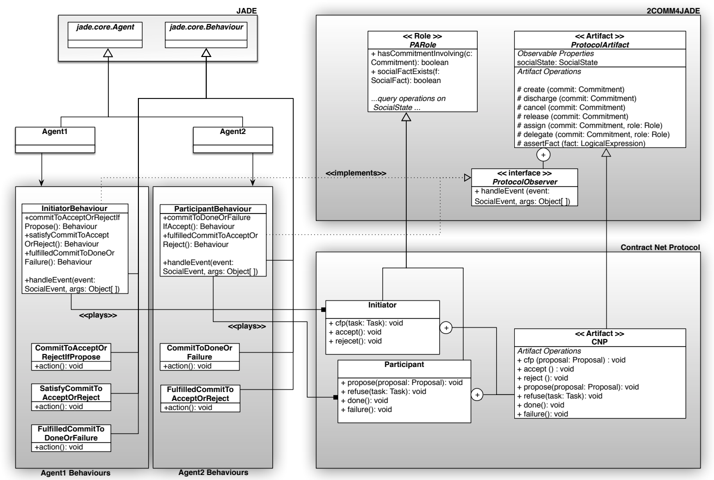


Fig. 3. CNP with 2COMM4JADE and support for event handling.

```
10   public abstract Behaviour
11       fulfilledCommitToDoneOrFailure();
12   public InitiatorBehaviour(String artifactName){
13     this.artifactName = artifactName;
14   }
15   public void action() {
16     ArtifactId art = Role.createArtifact(artifactName,
17       CNPArtifact.class);
18     initiator = (Initiator) (Role.enact(
19       CNPArtifact.INITIATOR_ROLE, art, this,
20       myAgent.getAID()));
21     initiator.startObserving(this);
22     myAgent.addBehaviour(
23       this.commitToAcceptOrRejectIfPropose());
24   }
25   public void handleEvent(SocialEvent e,
26       Object... args) {
27     SETemplate t = new SETemplate(initiator.getRoleId());
28     t.iAmDebtor().commitIsDetached();
29     t.matchCreditor(CNPArtifact.PARTICIPANT_ROLE);
30     t.matchConsequent("accept OR reject");
31     if (t.match(e)) {
32       myAgent.addBehaviour(
33         satisfyCommitToAcceptOrReject());
34     } else {
35       t.matchConsequent("done OR failure");
36       if (t.match(e))
37         myAgent.addBehaviour(
38           fulfilledCommitToDoneOrFailure());
39     }
40   }
41 }
```

After line 21, all events occurring in the social state are notified to the role Initiator, which will handle them by executing *handleEvent* after a callback. The above abstract behaviour is extended by the concrete behaviour of the agent that plays the role Initiator. In particular, here we find the methods that create the actual behaviours for managing the social events.

```
1 public class InitiatorAgent extends Agent {
2   // ...
3   public class InitiatorBehaviourImpl
4       extends InitiatorBehaviour {
5    public final String ARTIFACT_NAME = "CNP-1";
6    public InitiatorBehaviourImpl() {
7      super(ARTIFACT_NAME);
8    }
```

```
9     public Behaviour commitToAcceptOrRejectIfPropose(){
10      return new CommitToAcceptOrRejectIfPropose(
11        initiator);
12    }
13    public Behaviour satisfyCommitToAcceptOrReject(){
14      return new SatisfyCommitToAcceptOrReject(
15        initiator);
16    }
17    public Behaviour fulfilledCommitToDoneOrFailure(){
18      return new FulfilledCommitToDoneOrFailure(
19        initiator);
20    }
21  }
22 }
```

As an example, we describe the behaviour *SatisfyCommitToAcceptOrReject*, which gathers proposals and selects the one to accept.

```
1  public class SatisfyCommitToAcceptOrReject
2      extends OneShotBehaviour {
3    Initiator initiator = null;
4    ArrayList<Proposal> proposals =
5      new ArrayList<Proposal>();
6    public SatisfyCommitToAcceptOrReject(
7        Initiator initiator) {
8      super();
9      this.initiator = initiator;
10   }
11   public void action() {
12     ArrayList<RoleMessage> propos =
13         initiator.receiveAll(ACLMessage.PROPOSE);
14     for (RoleMessage p : propos) {
15       proposals.add((Proposal) (p.getContents()));
16     }
17     initiator.accept(proposals.get(0));
18     for (int i = 1; i < proposals.size(); i++) {
19       initiator.reject(proposals.get(i));
20     }
21   }
22 }
```

In 2COMM4JADE, an agent consists of a set of behaviours aimed at accomplishing given social relationships: such behaviours depend neither on when nor on how the social relationships of interest are created inside the social state. These aspects are, in fact, encoded in the protocol artifact that creates them based on the actions the agents perform. As a consequence, modifying how or when a social relationship is created does not have any impact on the agent implementation.

## V. Conclusions and Discussion

In this work, we have proposed 2COMM4JADE, an infrastructure for allowing actors to behave following an accepted set of regulations, in a self-governance context. 2COMM4JADE is based on JADE, for what concerns the support to the realization of interacting agents, and integrates self-governance mechanisms by relying on the reification of commitments and of commitment-based protocols. These are, at all respects, resources that are made available to stakeholders and that are realized by means of artifacts. We are working on an extension for Jason too, called 2COMM4Jason; details in [1]. Recently, we developed on top of 2COMM4JADE a commitment-based typing system [2]. Such typing includes a notion of compatibility, based on subtyping, which allows for the safe substitution of agents to roles along an interaction that is ruled by a commitment-based protocol. Type checking can be done dynamically when an agent enacts a role.

The proposal is characterized, on the one hand, by the flexibility and the openness that are typical of MAS, and, on the other, by the modularity and the compositionality that are typical requirements of the methodologies for design and development. One of the strong points of the proposal is the decoupling between the design of the agents and the design of the interaction, that builds on the decoupling between computation and coordination done by coordination models, like tuple spaces. This is a difference with respect to JADE where no decoupling occurs: a pattern of interaction is projected into a set of JADE behaviours, one for each role. Binding the interaction to ad-hoc behaviours does not allow having a global view of the protocol and complicates its maintenance.

Decoupling is an effect of explicitly representing social relationships as resources: agent behaviour is, thus, defined based on the existing social relationships and not on the process by which they are created. For instance, in CNP the initiator becomes active when the commitments that involve it as a debtor, and which bind it to accept or reject the proposals, are detached. It is not necessary to specify nor to manage, inside the agent, such things as deadlines or counting the received proposals: the artifact is in charge of these aspects.

The difference with tuple spaces and CArtAgO itself is that 2COMM4JADE artifacts reify social relationships as commitments, giving them that normative value that, jointly with the fact that by using an artifact an agent explicitly accepts the regulations encoded by it, enables the generation of expectations about the agents behaviour (as advised in [11]). As such, it allows agents to reason and take decision also based on the others' expected behaviour. For instance, the presence of a commitment *C(i, p, propose, accept* $\vee$ *reject)* grants the participant that its proposal will receive a feedback (it will either be accepted or rejected); if not, the initiator will be liable of a violation. In a tuple space, instead, agents can just read or write the shared blackboard but cannot have expectations on the behaviour of their parties.

This work has also relationships with works concerning e-institutions, like [15], [16], [20]. E-institutions introduce a normative aspect that lies on a different level with respect to the one we capture with commitments. Indeed, in our proposal the normative layer stands in the observational semantics of the social actions. We mean to extend the regulative aspects, as a first step, by enriching commitment protocols as proposed in [18], [3]. More in general, we mean to move towards a representation of business processes as suggested in the Comma methodology [26], where commitment patterns regulate activities instead of being used for giving the action semantics. Another advantage is that artifacts are an actual system for implementing mediated interaction, they supply efficient, simple, and direct means for checking the powers for performing institutional actions, and they offer runtime mechanisms for letting the agents perceive the state of the interaction.

## References


[1] Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. Social computing in JaCaMo. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263

of *Frontiers in Artificial Intelligence and Applications*, pages 959–960. IOS Press, 2014.

[2] Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. Typing Multi-Agent Systems via Commitments. In M. B. van Riemsdijk, F. Dalpiaz, and J. Dix, editors, *Proc. of the 2nd International Workshop on Engineering Multi-Agent Systems, EMAS 2014, held in conjuction with AAMAS 2014*, pages 341–359, Paris, France, May 2014. IFAAMAS.

[3] Matteo Baldoni, Cristina Baroglio, Elisa Marengo, and Viviana Patti. Constitutive and Regulative Specifications of Commitment Protocols: a Decoupled Approach. *ACM Transactions on Intelligent Systems and Technology*, 4(2):22:1–22:25, March 2013.

[4] Matteo Baldoni, Guido Boella, and Leendert van der Torre. Interaction between Objects in powerjava. *Journal of Object Technology, Special Issue OOPS Track at SAC 2006*, 6(2):5–30, 2007.

[5] Matteo Baldoni, Andrea Omicini, Cristina Baroglio, Viviana Mascardi, and Paolo Torroni. Agents, Multi-Agent Systems and Declarative Programming: What, When, Where, Why, Who, How? In A. Dovier and E. Pontelli, editors, *Twenty-five Years of Logic Programming in Italy*, volume 6125 of *Lecture Notes in Computer Science*, pages 204–230. Springer, Berlin Heidelberg, 2010.

[6] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. JADE - A Java Agent Development Framework. In R. H. Bordini, M. Dastani, J. JDix, and A. El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer, Berlin Heidelberg, 2005.

[7] Guido Boella and Leendert W. N. van der Torre. The ontological properties of social roles in multi-agent systems: definitional dependence, powers and roles playing roles. *Artificial Intelligence and Law*, 15(3):201–221, 2007.

[8] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J. Gómez-Sanz, João Leite, Gregory M. P. O'Hare, Alexander Pokahr, and Alessandro Ricci. A Survey of Programming Languages and Platforms for Multi-Agent Systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[9] Frances M. T. Brazier, Barbara Dunin-Keplicz, Nicholas R. Jennings, and Jan Treur. DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. *Int. J. Cooperative Inf. Syst.*, 6(1):67–94, 1997.

[10] Amit K. Chopra. *Commitment Alignment: Semantics, Patterns, and Decision Procedures for Distributed Computing*. PhD thesis, North Carolina State University, Raleigh, NC, 2009.

[11] Amit K. Chopra and Munindar P. Singh. An Architecture for Multiagent Systems: An Approach Based on Commitments. In *Proc. of the AAMAS Workshop on Programming Multiagent Systems (ProMAS)*, pages 184–202, Budapest, Hungary, May 2009. IFAAMAS.

[12] Rosaria Conte, Cristiano Castelfranchi, and Frank Dignum. Autonomous Norm Acceptance. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V, Agent Theories, Architectures, and Languages, 5th Int. Workshop, ATAL '98*, volume 1555 of *Lecture Notes in Computer Science*, pages 99–112, Paris, France, 1999. Springer.

[13] European Parliament. Directive 2004/39/EC of the European Parliament and of the Council of 21 April 2004 on markets in financial instruments. *Official Journal of the European Union*, L145:1–44, 2004.

[14] Michael Fisher, Rafael H. Bordini, Benjamin Hirsch, and Paolo Torroni. Computational Logics and Agents: A Road Map of Current Technologies and Future Trends. *Computational Intelligence*, 23(1):61–91, 2007.

[15] Nicoletta Fornara, Francesco Viganò, and Marco Colombetti. Agent communication and artificial institutions. *Autonomous Agents and Multi-Agent Systems*, 14(2):121–142, 2007.

[16] Nicoletta Fornara, Francesco Viganò, Mario Verdicchio, and Marco Colombetti. Artificial institutions: a model of institutional reality for open multiagent systems. *Artificial Intelligence and Law*, 16(1):89–105, 2008.

[17] Foundation for Intelligent Physical Agents. FIPA Specifications, 2002. http://www.fipa.org.

[18] Elisa Marengo, Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Viviana Patti, and Munindar P. Singh. Commitments with Regulations: Reasoning about Safety and Control in REGULA. In K. Tumer, P. Yolum, L. Sonenberg, and P. Stone, editors, *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2011*, volume 2, pages 467–474, Taipei, Taiwan, May 2011. IFAAMAS.

[19] Viviana Mascardi, Maurizio Martelli, and Leon Sterling. Logic-Based Specification Languages for Intelligent Software Agents. *Theory and Practice of Logic Programming*, 4(4):429–494, 2004.

[20] Daniel Okouya, Nicoletta Fornara, and Marco Colombetti. An Infrastructure for the Design and Development of Open Interaction Systems. In M. Cossentino, A. El Fallah-Seghrouchni, and M. Winikoff, editors, *Engineering Multi-Agent Systems - 1st Int. Workshop, EMAS 2013, Revised Selected Papers*, volume 8245 of *Lecture Notes in Computer Science*, pages 215–234, St. Paul, Minnesota, USA, 2013. Springer.

[21] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.

[22] Andrea Omicini and Franco Zambonelli. TuCSoN: a coordination model for mobile information agents. In D. G. Schwartz, M. Divitini, and T. Brasethvik, editors, *Proc. of the 1st Int. Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9 June 1998. IDI – NTNU, Trondheim (Norway).

[23] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.

[24] Munindar P. Singh. An Ontology for Commitments in Multiagent Systems. *Artificial Intelligence and Law*, 7(1):97–113, 1999.

[25] Munindar P. Singh. A Social Semantics for Agent Communication Languages. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Computer Science*, pages 31–45, Berlin Heidelberg, 2000. Springer.

[26] Pankaj R. Telang and Munindar P. Singh. Comma: a commitment-based business modeling methodology and its empirical evaluation. In W. van der Hoek, L. Padgham, V. Conitzer, and M. Winikoff, editors, *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012*, pages 1073–1080, Valencia, Spain, 2012. IFAAMAS.

[27] Alexander Thiele, Thomas Konnerth, Silvan Kaiser, Jan Keiser, and Benjamin Hirsch. Applying JIAC V to Real World Problems: The MAMS Case. In L. Braubach, W. van der Hoek, P. Petta, and A. Pokahr, editors, *Multiagent System Technologies, 7th German Conference, MATES 2009*, volume 5774 of *Lecture Notes in Computer Science*, pages 268–277, Hamburg, Germany, 2009. Springer.

[28] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.

[29] Pinar Yolum and Munindar P. Singh. Commitment Machines. In J.-J. Ch. Meyer and M. Tambe, editors, *Intelligent Agents VIII, 8th International Workshop, ATAL 2001, Revised Papers*, volume 2333 of *Lecture Notes in Computer Science*, pages 235–247, Seattle, WA, USA, 2002. Springer.