Programming JADE and Jason agents based on social relationships using a uniform approach

Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati

Università degli Studi di Torino — Dipartimento di Informatica
c.so Svizzera 185, I-10149 Torino (Italy)
{matteo.baldoni,cristina.baroglio,federico.capuzzimati}@unito.it

Abstract. Interaction is an essential feature in multiagent systems. Design primitives are needed to explicitly model desired patterns. This work presents 2COMM as a framework for defining social relations among parties, represented by social commitments. Starting from the definition of interaction protocols, 2COMM allows to decouple interaction design from agent design, an advantage that simplifies agent programming, independently of the chosen agent platform. A comparison between real implementations using 2COMM is provided for JADE and Jason agents.

Keywords: Social Commitments · Agents & Artifacts · Agent-Oriented Software Engineering

1 Introduction and Motivation

Multi-Agent Systems (MAS) represent a preferred choice for building complex systems, where the autonomy of each component represents a major requirement. Agent-oriented software engineers can choose from a substantial number of agent platforms [16,9,14,3]. Tools like JADE [6], TuCSoN [18], DESIRE [10], JaCaMo [8], all provide *coordination mechanisms* and *communication infrastructures* [9]; in our opinion, however, they lack of abstractions that allow a clear and explicit modeling of interaction: the way in which agents interact is spread across and "hard-coded" into agent implementations. A clear separation of the agent specification from the coordination specification would have the advantage of increasing agent decoupling as well as the decoupling of the agents from the logic the rules their interaction.

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 [21]; in order to represent the coordination requirements we propose to rely on commitment-based protocols [23]. We also claim that the social relationships and commitment-based protocols should be integrated in the system in the form of *resources* because this allows participants to dynamically recognize, accept, manipulate, reason on them, and decide whether to conform to them. This supplies a basis for coordination [13]. Furthermore, this work shows how starting from interactions in building a system can be useful in programming socially-responsive agents.

We rely on 2COMM [1] for building commitment-based protocols, in order to decouple interaction from agent logic, in a way that is not bounded to an agent platform. 2COMM is presented along with two connectors, one for JADE and the other for JaCaMo. Connectors enable agents from different platforms to use commitment protocols. When developing an agent that would play a role in a protocol, social commitments that the agent has to handle constitute an outline developers will follow; the result is a programming schema that make easier the actual implementation of agents.

In order to reify the social relationships we rely on the Agents & Artifacts meta-model (A&A) [22,17], which provides abstractions for environments and artifacts, that can be acted upon, observed, perceived, notified, and so on. 2COMM adopts the abstraction of artifact to construct communication protocols that realize a form of mediated, programmable communication, and in particular commitment protocols to establish an interaction social state agents can use to take decisions about their behaviour. Through 2COMM protocol artifacts, social relationships can be examined by the agents, as advised in [12], used (which entails that agents accept the corresponding regulations), constructed, e.g., by negotiation, specialized, composed, and so forth. Finally, 2COMM artifacts enable 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 2). 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 2COMM (Section 3) realizes the proposal based on an extension of JaCaMo; we propose programming schemas for JADE and Jason agents. We show the impact of the proposal on programming by means of an example (Section 4) based on Contract Net Protocol (CNP), a FIPA standard protocol. The example shows 1) practical advantages in terms of better code organization and easier coding of agents interaction, and 2) how agent implementation is lead by the interaction pattern, providing a cross-platform programming pattern.

2 Modeling Social Relationships

We propose to explicitly represent social relationships among the agents. By social relationships we mean normatively defined relationships, between two or more agents, resulting from the enactment of *roles*, and subject to *social control*. Thus, we encode social relationships as commitments. A commitment [20] 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.

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, to manipulate, to observe, to monitor, to reason, and to deliberate on social relationships. 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*, that are created during the protocols execution, available to agents as resources.

An artifact (A&A meta-model [22,17]) 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 its explicit acceptance, of the implications of the interaction protocol that the artifact reifies. This allows the interacting parties to perform *practical reasoning*, based on expectations: a debtors of a commitment is expected to behave so as to satisfy the commitment consequent conditions; otherwise, a violation is raised.

A commitment-based protocol consists of a set of actions, whose semantics is shared, and agreed upon, by all of the participants to the interaction [23,11]. The semantics of the social actions is given in terms of commitment operations (as usual for commitments, *create*, *cancel*, *release*, *discharge*, *assign*, and *delegate*). The execution of commitment operations modifies the *social state* of the system, which is shared by the interacting agents. As in [20], 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 [20,23,11]. Commitment-based protocols provide a means of coordination, based on the *notification* of social events, e.g. the creation of a commitment. Agents use artifacts to coordinate and interact in a way that depends on the roles they play and on their objectives. Such a decoupling avoids interaction logics to be hard-coded in agent programs, a thing that would lead to an increasing development efforts and a difficult maintenance of the system, especially in cross-firm settings. 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.

From an organizational perspective, a protocol is structured into a set of *roles*. We assume that roles cannot live autonomously: they exist in the system in view of the interaction. We follow the ontological model for roles proposed in [7], and brought inside the object-oriented paradigm in [4,5], 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.

3 2COMM: a Commitment-based infrastructure for social relationships

We have claimed that an agent-based framework should satisfy two requirements: 1) Explicit representation of the social relationship; 2) Social relationships should be first-class objects, which can be used for programming the agent behavior. 2COMM fulfills both requirements. Thanks to the *social relationship* abstraction, 2COMM enables an approach to agent programming that is not coupled to the chosen agent platform.

Currently, 2COMM supports social relationship-based agent programming for *JADE* and *JaCaMo* (i.e. Jason) agents. 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. JaCaMo [8] is a platform integrating Jason (as an agent programming language), CArtAgO and Moise (as a support to the realization of organizations). Normative/organizational specification is expressed as a Moise organization and translated into artifacts, that agents can decide to use.

We realize commitment-based interaction protocols by means of CArtAgO [19] artifacts. The core of 2COMM is in charge for management, maintenance and update of the social interaction state associated to each instance of a protocol artifact. CArtAgO 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. An API allows programming *artifacts*, 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. 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.

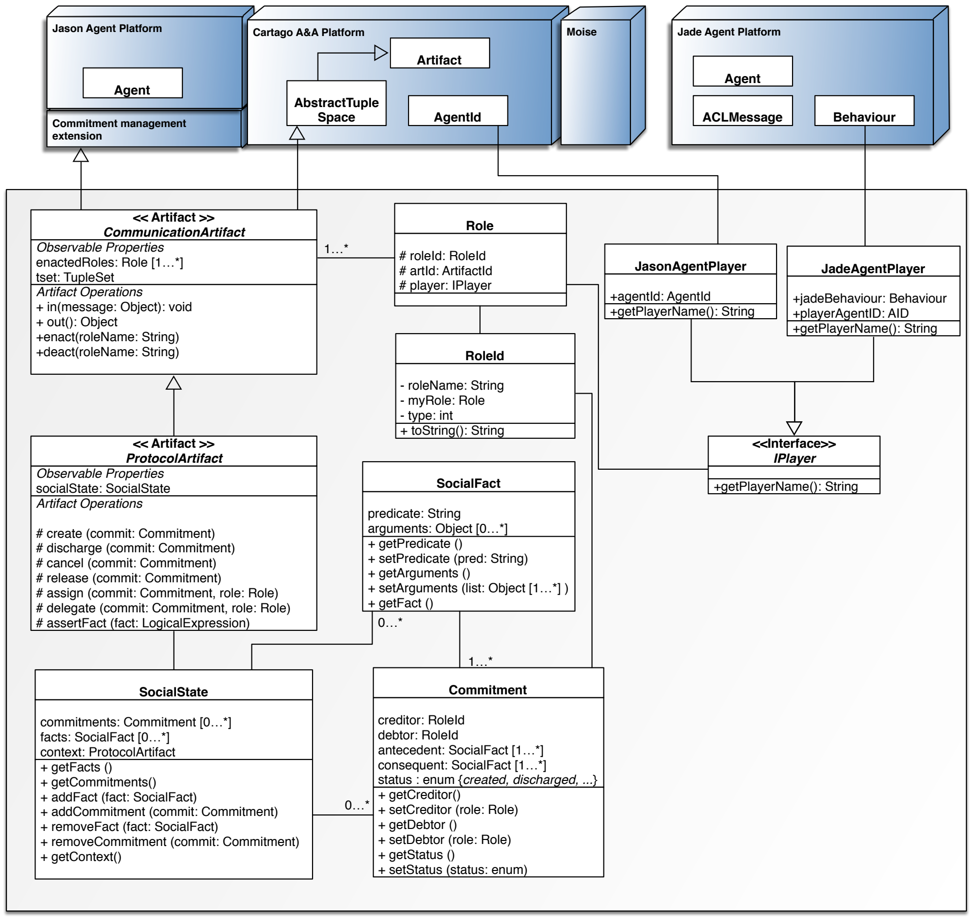


Fig. 1: Excerpt of the UML class diagram of 2COMM and connectors for JADE and Jason.

2COMM is organized as follows. 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. Roles are linked to agents of the specific platform, via connector classes that implements the `IPlayer` interface. Figure 1 reports an excerpt of the 2COMM UML class diagram[1]. Let us get into the depths of the implementation:

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

– *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 super-type, 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; the connector is in charge for linking agent and protocol through an instance of requested role.
– The class *CARole* is an inner class of CA and extends the Role class. It provides the *send* and *receive* primitives, implemented based on the *in* and *out* primitives provided by CA, by which agents can exchange messages.
– *ProtocolArtifact* (PA for short) extends CA and allows modeling the social layer with the help of commitments. It maintains the state of the ongoing 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 lifecycle 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 (by commitments), and the communication one (by 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)`. 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.
– The class *IPlayer* is the interface between roles and players adopting them. Currently 2COMM provides implementations for Jade (*JadeBehaviourPlayer*) and Jason (*JasonAgentPlayer*).

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. 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:

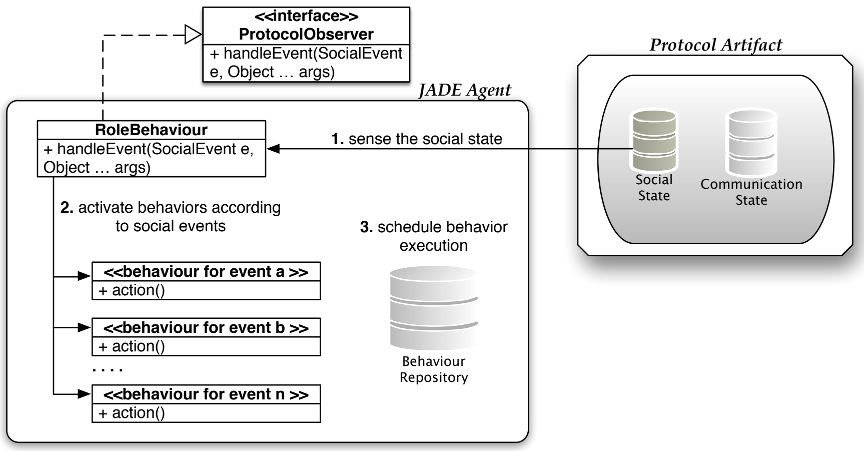
```java
public class MyProtocolArtifact
  extends ProtocolArtifact {
  // ...
  static {
    addEnabledRole("Role1", Role1.class);
    addEnabledRole("Role2", Role2.class);
    // ...
  }
  // MY PROTOCOL ARTIFACT OPERATIONS
  @OPERATION
  @ROLE(name="roleName")
  public void op1(...) {
    // prepare 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 {
    // ...
  }
  // ...
}
```

Protocol designers program the interaction protocol once. The resulting artifact can, then, be used in a JADE or in a JaCaMo context. Let us now briefly illustrate general schemas for programming JADE and Jason agents.

3.1 JADE schema

Figure 2 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 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. The following is the pseudo-code of an example implementation that agrees with the schema:

```java
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,
        new JadeBehaviourPlayer(this, myAgent.getAID())));
    myRole.startObserving(this);
    // add the initial behaviour of the agent
```



<<interface>>
ProtocolObserver
+ handleEvent(SocialEvent e, Object … args)
JADE Agent
Protocol Artifact
RoleBehaviour
+ handleEvent(SocialEvent e, Object … args)
1. sense the social state
Social State
Communication State
2. activate behaviors according to social events
3. schedule behavior execution
<<behaviour for event a >>
+ action()
<<behaviour for event b >>
+ action()
....
<<behaviour for event n >>
+ action()
Behaviour Repository

Fig. 2: 2COMM event handling schema for JADE agents.

```
12     }
13     public void handleEvent(SocialEvent e,
14                 Object... args) {
15       SETemplate t = new SETemplate(
16           myRole.getRoleId());
17       SETemplate t2 = new SETemplate(
18           myRole.getRoleId());
19       t.iAmDebtor().commitIsDetached()
20           .consequentMatch(...);
21       t2.iAmCreditor().commitIsConditional()
22           .antecedentMatch(...);
23       if (t.match(e)) {
24         myAgent.addBehaviour(...);   // behaviour to handle the case
25       } else if (t2.match(e)) {
26           myAgent.addBehaviour(...);// behaviour to handle another case
27       } else
28           // ...                  // behaviours for different cases
29       }
30     }
31   }
```

The basic schema, proposed for implementing a JADE behaviour, tracks how to handle social events that a protocol artifact signals to an agent. Signaling is performed through the *handleEvent* method, whose parameter contains the social effects of the event (e.g. if a commitment is added or satisfied, if a social fact is asserted, and such like). The implementation of *handleEvent* should contain conditions related to the occurred event. In JADE, event-related behaviours are added to the agent's behaviour library when a certain condition holds (in Jason, a plan will be triggered when a certain condition holds). If the social event to be notified 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 use 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.

3.2 Jason schema

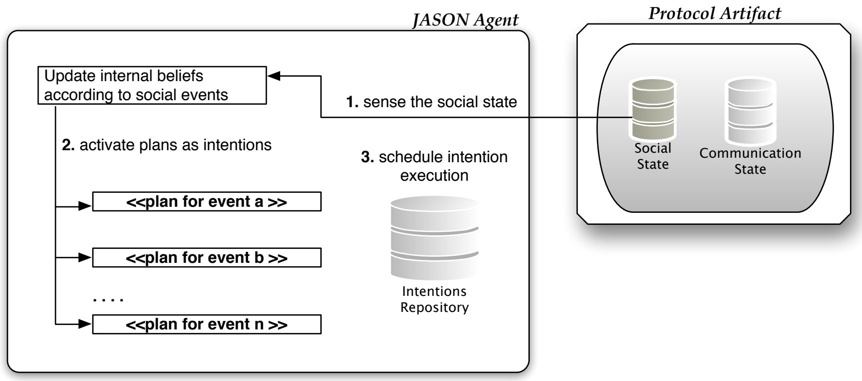


Fig. 3: 2COMM event handling schema for Jason agents.

A Jason agent has the capability of performing reasoning cycles, that is, the agent architecture performs a cycle of sense-plan-act, that allows agents to evaluate which plans are triggered for execution each time an event occurs.This is a difference with JADE, which, instead, provides the abstraction of agent only as a set of behaviours with communication capabilities. For this reason it is not required for Jason agents to foresee a specific processing of social events; in facts, they can be modeled as regular Jason events, fired by the protocol artifact. The adoption of artifacts that signal the occurrence of social events to focusing agents allows plan specifications whose triggering events involve commitments, as depicted in figure 3. Commitments can also be used inside a plan context or body. As a difference with beliefs, commitment assertion/deletion can only occur through the artifact, as a consequence of a modification of the social state. For example, this is the case that deals with commitment addition:

$$+cc(debtor, creditor, antecedent, consequent, status) :$$
$$\langle context \rangle \leftarrow \langle body \rangle.$$

The plan is triggered when a commitment that unifies with the one on the left hand side appears in the social state with the specified status. The syntax is the standard for Jason plans. *Debtor* and *Creditor* are to be substituted by the proper role names. The plan may be devised so as to achieve a change of the status of the commitment (e.g.: the debtor will satisfy the consequent, the creditor will satisfy the antecedent and so detach the commitment) or it may be devised to allow the agent to do something as a reaction (e.g. collecting information). A similar schema can be used in the case of commitment deletion and in the case of addition (deletion) of social facts. Commitments can also be used in contexts and in plans as test goals ($?cc(\dots)$) or achievement goals ($!cc(\dots)$). Addition or deletion of such goals can, as well, be managed by plans. For example:

$$+!cc(debtor, creditor, antecedent, consequent, status) :$$
$$\langle context \rangle \leftarrow \langle body \rangle.$$

The plan is triggered when the agent creates an achievement goal concerning a commitment. Consequently, the agent will act upon the artifact so as to create the desired social relationship. After the execution of the plan the commitment $cc(debtor, creditor, antecedent, consequent, status)$ will hold in the social state and will have been projected onto the belief bases of each of the agents which focused on the artifact.

4 Programming Agents on Social Relationships: an example

2COMM protocols constitute an outline for building agents that entails a uniform implementation of agent social abilities among different platforms: how to react to social events. We present a real implementation of the Contract Net Protocol (CNP) [15], and how JADE and Jason agents can be implemented. We adopt the following commitment-based CNP formulation:

cfp **causes** create(*C(i, p, propose, accept* ∨ *reject)*)
accept **causes** *none*
reject **causes** release(*C(p, i, accept, done* ∨ *failure)*)
propose **causes** create(*C(p, i, accept, done* ∨ *failure)*)
refuse **causes** release(*C(i, p, propose, accept* ∨ *reject)*)
done **causes** *none*
failure **causes** *none*

where i stands for the role *Initiator*, p for *Participant*, and *none* means that the action execution just makes existing commitments progress (e.g. *propose* detaches *C(i, p, propose, accept* ∨ *reject)*), if any involves it. Initiator supplies its player the actions *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 actions *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. Actions affect the social state, e.g., when an Initiator executes *cfp*, the commitment $C(i,\ p,\ propose,\ accept \vee reject)$ is added to the social state. This binds i to either accept or reject a proposal, if one is received.

```
 1 public class Cnp extends ProtocolArtifact {
 2   private int numberMaxProposals = 10;
 3   private int actualProposals = 0;
 4    // ... other protocol operations ...
 5   @OPERATION
 6   @ROLE(name="participant")
 7   public void propose(String prop, int cost,
 8                       String init) {
 9     Proposal p = new Proposal(prop, cost);
10     RoleId participant =
11        getRoleIdByPlayerName(getOpUserName());
12     RoleId initiator =
13        getRoleIdByRoleCanonicalName(init);
14     p.setRoleId(participant);
15      RoleMessage propMessage = new RoleMessage(
16         participant,initiator,ACLMessage.PROPOSE,proposal);
17     send(propMessage);
18     defineObsProperty("proposal",
19        p.getProposalContent(),p.getCost(),
20        participant.getCanonicalName());
21     createCommitment(new Commitment(participant,
22        initiator,"accept", "done OR failure"));
23     assertFact(new Fact("propose", participant,
24                         prop));
25     actualProposals++;
26     if (actualProposals == numberMaxProposals) {
27       RoleId groupParticipant =
28          new RoleId("participant");
29       createCommitment(new Commitment(initiator,
30                groupParticipant,
31                "true", "accept OR reject"));
32     }
33   // ... other protocol operations ...
34    // Role classes
35   public class Initiator extends PARole {
36     // ...
37     public void cfp(Task task) {
38       doAction(this.getArtifactId(),
39         new Op("cfp", task, getRoleId()));
40     }
41     // ...
42   }
43   public class Participant extends PARole {
44     public void propose(Proposal proposal,
45        RoleId proposalSender) {
46       // ...
47     }
48     // ...
49   }
50 }
```

propose (line 7) is a social action. It is realized as a CArtAgO operation, in fact it is decorated by the CArtAgO Java annotation @OPERATION, line 5. It can be executed only by an agent playing the role *participant*. This is specified by the 2COMM Java annotation @ROLE(name="participant"), line 6. It asserts social fact (line 23), that traces the proposal made by the participant; then, it counts the received proposals and, when their number is sufficient, signals this fact to the initiator by the creation of a commitment (line 21) towards the group of participants. A message of performative PROPOSE (line 15) containing the participant's proposal is sent to the initiator.

The proposed CNP implementation remains the same independently from the fact that it is used by a JADE or a Jason agents. 2COMM current version uses role internal classes for JADE agents, while these are ignored if the enacting agents are written in Jason. Let us now compare agents implementations to highlight similarities and analogies. We will focus on the code for the Initiator role, starting from a JADE agent.

Protocol designers can provide full support to JADE developers by implementing behaviours that act as *social event adapters*, so an agent developer only needs to provide behaviours for the events that are signaled by the artifact. A clear advantage is an improved code reuse and modularization: the agent needs to be able to react to social events, adopting corresponding behaviours, and, therefore, the agent's autonomy is not jeopardized by extending the adapter. Here is a possible implementation for the Initiator adapter behaviour.

```
 1 public abstract class InitiatorAdapterBehaviour
 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();
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 }}
```

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.

```
public class InitiatorAgent extends Agent {
  // ...
  public class InitiatorBehaviourImpl
      extends InitiatorBehaviour {
   public final String ARTIFACT_NAME = "CNP-1";
   public InitiatorBehaviourImpl() {
     super(ARTIFACT_NAME);
   }
   public Behaviour commitToAcceptOrRejectIfPropose(){
    return new CommitToAcceptOrRejectIfPropose(
      initiator);
   }
```

```
    public Behaviour satisfyCommitToAcceptOrReject(){
     return new SatisfyCommitToAcceptOrReject(
       initiator);
    }
    public Behaviour fulfilledCommitToDoneOrFailure(){
     return new FulfilledCommitToDoneOrFailure(
       initiator);
    }
 }
}
```

The agent logic is structured as a number of behaviours that are in charge for handling the social events. When a social event is received, the adapter loads the corresponding behaviour, that is scheduled for the execution. This is similar to how a Jason agent is programmed, that is, a collection of plans that become active when a trigger is satisfied. We describe the behaviour *SatisfyCommit-ToAcceptOrReject*, which gathers proposals and selects the one to accept.

```
public class SatisfyCommitToAcceptOrReject
    extends OneShotBehaviour {
  Initiator initiator = null;
  ArrayList<Proposal> proposals =
    new ArrayList<Proposal >();
  public SatisfyCommitToAcceptOrReject(
       Initiator initiator) {
    super();
    this.initiator = initiator;
  }
  public void action() {
    ArrayList<RoleMessage> propos =
        initiator.receiveAll(ACLMessage.PROPOSE);
    for (RoleMessage p : propos) {
      proposals.add((Proposal) (p.getContents()));
    }
    initiator.accept(proposals.get(0));
    for (int i = 1; i < proposals.size(); i++) {
      initiator.reject(proposals.get(i));
    }
  }
}
```

This implementation is analogous to how a Jason agent can be programmed to react to the same commitment:

```

+cc(My_Role_Id, "participant", "true",
              "(accept OR reject)", "DETACHED")
  : enactment_id(My_Role_Id) & not evaluated
  <- +evaluated;
      .wait(2000);
      .findall(proposal(Content,Cost,Id),
              proposal(Content,Cost,Id),Proposals);
      .min(Proposals,proposal(Proposal,Cost,Winner_Role_Id));
      accept(Winner_Role_Id).
      ... action 'reject' for all other proposals ...
```

We now report and comment excerpts of Jason agent code for the *Initiator*.

```
/* Initial goals */
!startCNP.
/* Plans */
+!startCNP : true
  <- makeArtifact("cnp","cnp.Cnp",[],C);
      focus(C);
      enact("initiator").
+enacted(Id,"initiator",Role_Id)
  <- +enactment_id(Role_Id);
      !cc(Role_Id, "participant", "propose",
              "(accept OR reject)","CONDITIONAL").
+!cc(My_Role_Id, "participant", "propose",
              "(accept OR reject)","CONDITIONAL")
  <- .print("sending cfp");
      .wait(2000);
      cfp("task-one").
+cc(My_Role_Id, "participant", "true",
              "(accept OR reject)", "DETACHED")
  : enactment_id(My_Role_Id) & not evaluated
  <- +evaluated;
      .wait(2000);
      .findall(proposal(Content,Cost,Id),
              proposal(Content,Cost,Id),Proposals);
```

```
24        .min(Proposals,proposal(Proposal,Cost,Winner_Role_Id));
25        accept(Winner_Role_Id).
26        ... action 'reject' for all other proposals ...
27 +cc(Participant_Role_Id, My_Role_Id, "true",
28               "(done OR failure)", "DISCHARGED")
29   :  done(Result)
30   <-  .print("Task resolved: ",Result).
31 +cc(Participant_Role_Id, My_Role_Id, "true",
32               "(done OR failure)", "DISCHARGED")
33   :  failure(Participant_Role_Id)
34   <-  .print("Task failed by ",Participant_role_id).
```

!startCNP, line 2, is an initial goal, that is provided for beginning the interaction. In this implementation, the agent which plays the initiator role is in charge for creating the artifact (*makeArtifact("cnp","cnp.Cnp",[],C)*) that will be used for the interaction. The agent will, then, enact the role "initiator" (*enact("initiator")*); the artifact will notify the success of the operation by asserting an *enacted* belief. Since the program contains the plan triggered by the *enacted* belief, the initiator agent can, then, execute *cfp*. When enough participants will have committed to perform the task, in case their proposal is accepted (*cc(My_Role_Id, "participant", "true", "(accept OR reject","DETACHED")*, the initiator agent evaluates the proposals and decides which to *accept* (we omit the reject case for sake of brevity).

Summarizing, a JADE agent leveraging 2COMM artifacts 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. Analogously for Jason agents, plans are not affected by modifications made on protocol: it is possible to adapt the interaction logic to different contexts without any impact on agents. Each plan is defined as reaction to a social event, whose evolution is stated by the artifact.

The following table synthesizes a comparison among JADE and JaCaMo, highlighting aspects that are improved or added by 2COMM.

	JADE	**+ 2COMM**	**JaCaMo**	**+ 2COMM**
Programmable communication channels	X	✓	✓	✓
Notification of social relationships of interaction	X	✓	X	✓
Interaction/agent logic decoupling	X	✓	X	✓
Expected behaviours reasoning	X	✓	✓	✓
Library of reusable patterns of interaction	✓	✓	X	✓
Runtime interaction monitoring	X	✓	✓	✓
Social-based Agent Programming Pattern	X	✓	X	✓
Norms and Obligations modeling	X	X	✓	✓

Table 1: Comparison among JADE, JaCaMo and 2COMM improvements.

5 Conclusions and Discussion

In this work, we have proposed 2COMM, an infrastructure for allowing actors to behave following an accepted set of regulations, in a self-governance context. 2COMM 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. 2COMM supports programming JADE and Jason agents, by following a uniform approach. Recently, we developed on top of 2COMM a commitment-based typing system [2] for JADE agents. 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 or JaCaMo where no decoupling occurs: a pattern of interaction is projected into a set of JADE behaviours or Jason plans, one for each role. Binding the interaction to ad-hoc behaviours/plans 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. Testing 2COMM with Jason and JADE proved that programming agents starting from their desired interaction can be a valuable starting point, that can be extended towards a methodology useful for open and heterogeneous scenarios. We intend to explore this direction by adding connectors for different agent platforms.

References

1. Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. A commitment-based infrastructure for programming socio-technical systems. *ACM Transactions on Internet Technology (TOIT)*, 14(4):23, 2014.
2. Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. Typing multi-agent systems via commitments. In Fabiano Dalpiaz, Jürgen Dix, and M.Birna van Riemsdijk, editors, *Engineering Multi-Agent Systems*, volume 8758 of *Lecture Notes in Computer Science*, pages 388–405. Springer International Publishing, 2014.

3. Matteo Baldoni, Cristina Baroglio, Viviana Mascardi, Andrea Omicini, and Paolo Torroni. Agents, multi-agent systems and declarative programming: What, when, where, why, who, how? In *25 Years GULP*, volume 6125 of *Lecture Notes in Computer Science*, pages 204–230. Springer, 2010.
4. Matteo Baldoni, Guido Boella, and Leendert van der Torre. Modelling the Interaction between Objects: Roles as Affordances. In J. Lang, F. Lin, and J. Wang, editors, *Knowledge Science, Engineering and Management: First International Conference, KSEM*, volume 4092 of *LNCS*, pages 42–54, Guilin City, China, August 5-8 2006. Springer.
5. Matteo Baldoni, Guido Boella, and Leendert van der Torre. Interaction between objects in powerjava. *Journal of Object Technology*, 6(2), 2007.
6. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE - A Java Agent Development Framework. In *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer, 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. Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747 – 761, 2013.
9. 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.
10. Frances M. T. Brazier, Barbara M. Dunin-Keplicz, Nick R. Jennings, and Jan Treur. Desire: Modelling Multi-Agent Systems in a Compositional Formal Framework. *International Journal of Cooperative Information Systems*, 06(01):67–94, March 1997.
11. Amit K. Chopra. *Commitment Alignment: Semantics, Patterns, and Decision Procedures for Distributed Computing.* PhD thesis, North Carolina State University, Raleigh, NC, 2009.
12. Amit K. Chopra and Munindar P. Singh. An Architecture for Multiagent Systems: An Approach Based on Commitments. In *Proc. of ProMAS*, 2009.
13. Rosaria Conte, Cristiano Castelfranchi, and Frank Dignum. Autonomous norm acceptance. In *Intelligent Agents V, Agent Theories, Architectures, and Languages, ATAL '98*, volume 1555 of *Lecture Notes in Computer Science*, pages 99–112. Springer, 1998.
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. Foundation for Intelligent Physical Agents. FIPA Specifications, 2002. http://www.fipa.org.
16. Viviana Mascardi, Maurizio Martelli, and Leon Sterling. Logic-based specification languages for intelligent software agents. *TPLP*, 4(4):429–494, 2004.
17. 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.
18. Andrea Omicini and Franco Zambonelli. TuCSoN: a coordination model for mobile information agents. In *1st International Workshop on Innovative Internet Infor-*

mation Systems (IIIS'98), pages 177–187. IDI – NTNU, Trondheim (Norway), 8–9 June 1998.

19. 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.
20. Munindar P. Singh. An ontology for commitments in multiagent systems. *Artif. Intell. Law*, 7(1):97–113, 1999.
21. Munindar P. Singh. A social semantics for agent communication languages. In *Issues in Agent Communication*, volume 1916 of *LNCS*, pages 31–45. Springer, 2000.
22. 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.
23. P. Yolum and M. P. Singh. Commitment Machines. In *Intelligent Agents VIII, 8th International Workshop, ATAL 2001*, volume 2333 of *LNCS*, pages 235–247. Springer, 2002.