Reusable Components for Implementing Agent Interactions*

Juan M. Serrano, Sascha Ossowski, and Sergio Saugar

University Rey Juan Carlos, Department of Computing
{JuanManuel.Serrano, Sascha.Ossowski, Sergio.Saugar}@urjc.es

Abstract. Engineering component interactions is a major challenge in the development of large-scale, open systems. In the realm of multiagent system research, organizational abstractions have been proposed to overcome the complexity of this task. However, the gap between these modeling abstractions, and the constructs provided by todays agent-oriented software frameworks is still rather big. This paper reports on the $\mathcal{RICA}$–$\mathcal{J}$ multiagent programming framework, which provides executable constructs for each of the organizational, ACL-based modeling abstractions of the $\mathcal{RICA}$ theory. Setting out from a components and connectors perspective on the elements of the $\mathcal{RICA}$ metamodel, their executions semantics is defined and instrumented on top of the JADE platform. Moreover, a systematic reuse approach to the engineering of interactions is put forward.

1 Introduction

In the past few years, multi-agent systems (MAS) have been proposed as a suitable software engineering paradigm to face the challenges posed by the development of large-scale, open systems [1, 2]. Two major characteristics of MAS are commonly put forward to justify this claim. Firstly, agents are excellent candidates to occupy the place of autonomous, heterogeneous and dynamic components that open systems require [2]. Secondly, the organizational stance advocated in various degrees by most MAS methodologies, provides an excellent basis to deal with the complexity and dynamism of the interactions among system components [1]. In particular, organization-oriented abstractions such as roles, social interactions, groups, organizations, institutions, etc., have proved to be an effective means to model the interaction space of complex MAS [1, 3–5].

However, the gap between these modeling abstractions, and the constructs provided by todays agent-oriented software frameworks is still huge. A way to bridge this gap is to include organization-oriented abstractions as first-class constructs into a multiagent programming language. For this purpose, it is essential to define the execution semantics of the new programming constructs, in a way that is independent from any technological basis [6]. In addition, from a mainstream software

* Research sponsored by the Spanish Ministry of Science and Education (MEC), project TIC2003-08763-C02-02.

R.H. Bordini et al. (Eds.): ProMAS 2005, LNAI 3862, pp. 101–119, 2006.
© Springer-Verlag Berlin Heidelberg 2006

engineering perspective, it is of foremost importance that the new abstractions foster a systematic reuse approach, specially in the context of large-scale software systems [7]. A similar concern has been put forward in the field of MAS engineering regarding the reuse of organizational structures [8] and agents [9].

This paper reports on the $\mathcal{RICA}$–$\mathcal{J}$ multiagent programming framework, which provides executable constructs for each of the modeling abstractions of the *Role/Interaction/Communicative Action* ($\mathcal{RICA}$) theory [4]. For this purpose, the execution semantics of the different elements of the $\mathcal{RICA}$ metamodel is specified, drawing inspiration from component and connector (C&C) architectures [10]. With regard to reusability matters, the $\mathcal{RICA}$–$\mathcal{J}$ software framework exploits the organizational stance on ACLs postulated by the $\mathcal{RICA}$ theory, which results in the identification of generic application-independent social interactions.

The remainder of the paper is organized as follows. Section 2 shows how a C&C perspective on the $\mathcal{RICA}$ theory can be used to define an execution semantics for its elements. Section 3 analyzes communicative roles and interactions from the point of view of generic software components [7], so as to identify policies for their reuse. Section 4 provides a survey of the $\mathcal{RICA}$–$\mathcal{J}$ framework, emphasizing the mapping of the C&C-based execution semantics to the JADE platform, and the architecture of a $\mathcal{RICA}$–$\mathcal{J}$ application. The paper is concluded with a discussion on the lessons learned as well as pointers to current and future work.

2 MAS as a C&C Style

From a software architecture point of view an application structure is described as a collection of interacting components [10]. Components represent the computational elements or processing units of the system (the locus of control and computation), while connectors represent interactions among components. Different types of connectors represent different forms of interaction: pipes, procedure call, SQL link, event-broadcast, etc. This section will first show how this general view of software architecture fits the multiagent social architecture endorsed by the $\mathcal{RICA}$ theory. Then, based upon this view, and as a previous step to the instrumentation of the theory, the execution semantics of the $\mathcal{RICA}$ theory will be outlined.

2.1 Components and Connectors in the $\mathcal{RICA}$ Theory

The metamodel of the $\mathcal{RICA}$ theory provides a modeling language of the *organizational* and *communicative* features of MAS [4]. In this section we will focus on the key organizational abstraction, namely social interactions. Moreover, we introduce *group meetings* as the context in which social interactions take place, thus extending the basic set of abstractions of the theory[1]. The next section deals with the communicative layer of the metamodel.

[1] The metamodel may be further extended with other kinds of abstractions such as *Organization Types* and *Norms*. However, given the purpose of this paper, they are not needed.

Here, and throughout the rest of this paper, we will refer for illustration purposes to a common application domain in the organization literature: the management of scientific conferences [1][11]. *Authors*, *Reviewers*, *Program Committee Members* and *Program Committee Chairs* (PC chairs) would be common roles played by agents within an organization designed to support this process. Interactions among these agents will take place in the context of several group meetings, such as the *submission* and the *reviewing* group types. Figure 1 shows a partial $\mathcal{RICA}$ model of the submission group expressed in terms of a UML class diagram, which makes use of several stereotypes that refer to the kind of meta-entities and -relationships of the $\mathcal{RICA}$ metamodel (the `<<C>>` stereotype stands for the *c*apabilities of interactive role types).

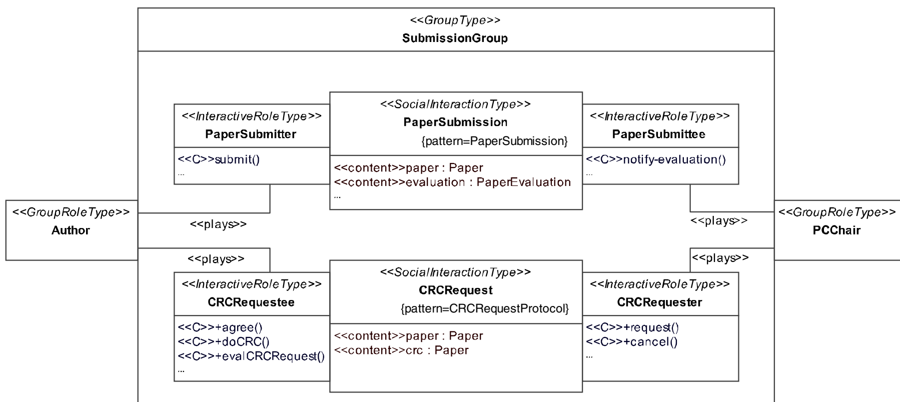

Fig. 1. $\mathcal{RICA}$ model of the *Submission* group

The specification of a group type establishes the kinds of agents which may participate in groups of that kind by identifying several *Group Role* types. Authors and PC Chairs, for example, represent the agents that may participate in a submission group. Moreover, the different kinds of interactions held in some group meeting must be specified as well. For instance, in the context of the submission group, *Paper Submission* and *Camera Ready Copy (CRC) Request* interactions will happen. An agent participates in a social interaction by taking on certain *Interactive Role* types. For instance, an author will participate in a paper submission interaction as the *Paper Submitter*, whereas the PC chair will take part in these interactions as a *Paper Submittee*. The *plays* association in the $\mathcal{RICA}$ metamodel among a group role and an interactive role establishes that agents occupying the former position may take part in some kind of interaction by playing the latter interactive role. Interactive role types characterize the behaviours that agents may show when they engage in the interaction, in terms of the communicative actions (CA) (e.g. *submit*, *agree*) and other types of social actions (e.g. *doCRC*, performed by authors in order to generate the CRC of an

accepted paper) that they *can* perform. Besides the participant roles, other components may form part of the definition of a kind of social interaction. Firstly, a collection of parameter types which specify the *content* of the interaction, such as the *paper* to be submitted and its *evaluation*. Input/output parameters of communicative and social actions must refer to the parameters declared for the interaction type. Secondly, a collection of interaction protocol types representing different *patterns* along which the interaction is supposed to develop (e.g. the *submissionProcedure*).

From an organizational point of view on MAS [1, 11, 12], and without discarding the autonomy of agents, it is possible to abstract from the agents' internal architecture, and focus on the roles that an agent may play within the organization. So, agents can be conceived as a particular type of *software component*. Furthermore, social interactions can be considered as different kinds of *connectors*[2], since they establish the interaction rules among agents, and thus mediate their communication and coordination activities: paper submission, CRC requests, and so on, specify the particular manner in which authors and PC Chairs interact in the context of the submission group. Moreover, pushing the analogy even further, connector types declare a number of roles and protocols [13] which directly map onto the interactive roles and interaction protocols that the $\mathcal{RICA}$ theory associates to social interactions. For instance, *caller* and *callee* in a RPC connector interaction, *reader* and *writer* in a pipe interaction, and so forth, are analogues of *PaperSubmitter* and *PaperSubmittee* as declared by the paper submission interaction.

Social interactions mainly differ from pipes, SQL links, and other types of connectors in their characteristic interaction mechanism: the Agent Communication Language (ACL). Firstly, ACLs allow for a more anthropomorphic description of the interactive roles and protocols than connectors. Secondly, they allow for more flexibility, as the equivalent to connector protocols *may* be derived from the communicative action semantics [14].

2.2 An Execution Semantics for $\mathcal{RICA}$ Models

As a first step towards defining programming abstractions that correspond to the $\mathcal{RICA}$ modeling entities, the execution semantics of the later needs to be defined: we have to show how social social interactions are enacted by agents at run-time in the context of group meetings[3]. In this section we will sketch this execution semantics based on a run-time instance of the previous conference management model (see Figure 2). As we focus on the dynamics of social interactions, we will assume that an instance of the submission group has already been set up by some PC Chair. Moreover, we will consider that three authors, a_3, a_4 and a_5, and two PC chairs, a_1 and a_2, has joined this group meeting.

The behaviour of agents within a group meeting is given by the group role instances and the interactive role instances that they play. Hence, an agent

[2] Group types may also be conceptualised as a special kind of connector.

[3] Here, agents, social roles, interactions, and so forth, denote *instances* of the corresponding $\mathcal{RICA}$ metamodel abstractions.

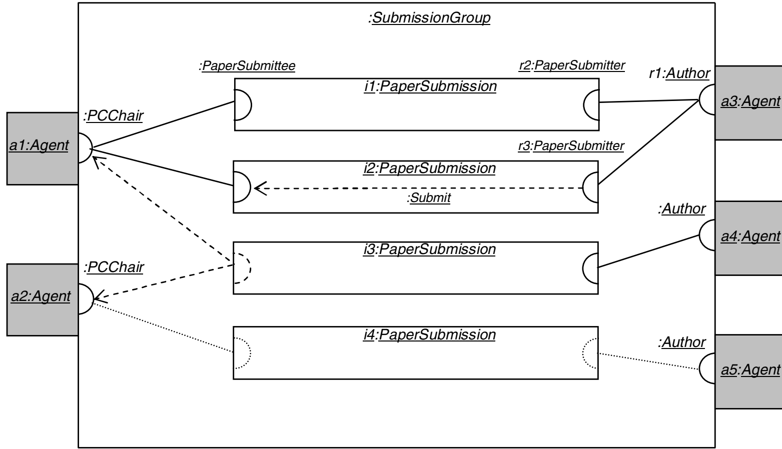

Fig. 2. Run-time instance of the e-commerce model

comprises a set of role instances: each of them encapsulates an action selection strategy that is compliant with the restrictions put forward by the role type. Role instances are run in parallel by the agent and may be active or suspended.

If some agent has activated a given group role instance, it may *engage* other agent(s) of the same group in social interaction. Submission interactions, for example, will be started by authors. The execution of an *engage* action results in the creation of a new interaction. Once the interaction has been created, the agent will instantiate the corresponding interactive role type to carry out its activity within that particular interaction. Thus, a given agent has as many interactive role instances as interactions in which it is participating. For instance, in figure 2, the participation of author agent a_3 in paper submission interactions i_1 and i_2 is carried out by its role instances r_2 and r_3, respectively.

The *engage* action specifies a collection of addressees which will be notified of the new interaction. In our example, authors will normally engage *any* PC Chair in paper submission. Interaction i_3, shown in the example, represents the result of the engagement performed by agent a_4. Once some agent is notified of a new interaction addressed to it, it may *join* that interaction and instantiate its corresponding interactive role type. Only the addressees of some interaction may join it. However, it may happen that some agent who received the engagement notification can not join the interaction because the role's position is already occupied by other agent who was faster in joining it[4]. Note that the *engage* action automatically involves a *join* action performed by the initiator agent.

The interactive role behaviour determines which types of actions will be *executed* by the agent in the course of the interaction. It also makes sure that the

[4] This essentially depends on the cardinality feature declared by the interactive role type. For instance, in multi-party conversations such as auctions multiple bidders may join the interaction.

agent *observes* the types of actions executed by other participant in the interaction, and that it *aborts* action execution if some condition is fulfilled. Abortions will normally affect non-communicative actions, whereas observation will result in the suspension of the role behaviour until the action (normally, a CA) is performed.

For instance, as soon as a PC Chair joins a submission interaction initiated by some author it will wait for the author's submission (i.e. for the performance of a *submit* CA, as exemplified in interaction i_2). Once the author submits its paper, it will be vigilant for the agreement or refusal of the PC Chair. In case of agreement, the PC Chair will eventually *notify* the *evaluation* of the paper. The PC Chair may then *leave* the interaction and deallocate its interactive role, thus finishing its participation in that conversation. Once the author receives the notification of the paper's evaluation it will *close* (and, hence, also *leave*) the interaction. Some interaction may only be closed by its initiator (i.e. the one who initially engage other agents in the conversation). If some interaction is closed the pending agent(s) will be only able to leave it (i.e. no action performing will be allowed). Interaction i_4 represents an interaction between agents a_5 and a_2, closed by the initiator agent a_5.

Interaction behaviour need not be explicitly declared by role types, but can also be inferred from the protocol which regulate the social interaction. Thus, protocol *instances* in $\mathcal{RICA}$ provide all the aforementioned services: action execution, observation, abortion, etc. Still, the particular way in which these services are provided depends on the execution semantics of the technique in which the interaction protocol is specified (FSMs, Petri nets, etc.).

3 ACLs and Component-Based Development

This section will evaluate the potential for reusability of the $\mathcal{RICA}$ theory in the context of the above execution semantics. Specifically, we look for reusable software artifacts (i.e. components[5] [7]), which may support a systematic reuse approach. First, we will show how ACL dialects serve to identify these reusable components. Then, we will discuss their associated customization mechanisms.

3.1 Reusable Components in the $\mathcal{RICA}$ Theory

As the *CRCRequest* interaction model shows (see figure 1), a social interaction type must specify the communicative actions that participating agents can perform. Protocols that regulate their behaviours may be defined as well. Still, PC Chairs are not the only kind of agents that will issue requests or cancellations. Similarly, a CRC request protocol may essentially be modeled after a generic "request protocol" (e.g. as the FIPA Request Protocol [15]). The $\mathcal{RICA}$ theory

[5] This sense of the word "component" should not be confused with the meaning of the term in the context of C&C architectures. The context will provide the right interpretation.

abstracts these pragmatic features (CAs and protocols) away from the definition of interactive roles and social interactions, and encapsulates them in their characteristic *communicative roles* and *communicative interactions*, particular types of interactive roles and social interactions defined from a set of performatives and protocols, i.e. an ACL dialect. For instance, the *requester* role type is the unique type of role characterized by *request* and *cancel* CAs. Communicative roles may also include non-communicative actions. For instance, *requestee* agents will need to determine if the requested action can be performed or not, which may be accomplished by the *evalRequest* action. We call the characteristic interaction in which requester and requestee agents participate *action performing.* This communicative interaction type, shown in figure 3, encapsulates the request protocol, besides generic content parameters such as the action to be requested, the agreement conditions, etc.

Thanks to the cross-domain features of CAs and protocols, ACL-based interactions are generic first-order reusable components[6]. Indeed, communicative interactions provide the pragmatic features of application-dependent social interactions, which basically differ at the semantic level. However, dynamic features of interactive role types are not completely defined at this level of abstraction. Particularly, a communicative interaction type says nothing about the rules to be followed in order to set up or close an interaction of that kind: the *engagement* and *closing* rules. Similarly, the *joining* and *leaving* rules for each kind of communicative role type are not declared either, so that these kinds of roles will be *abstract.* Note that these rules complements the static features of the $\mathcal{RICA}$ metamodel described in the last section.

3.2 Customization Mechanism

There are two major customization mechanisms to reuse communicative interaction components in a given application domain: *delegation* and *specialization.* Due to lack of space, the discussion is constrained to the latter mechanism, which relies in a recursive definition model by which some type (the derived type) is defined in terms of a super-type (or base type), by *extending*, *overriding* and/or *inheriting* some of its definition components[7].

For instance, figure 3 shows the recursive specification of the CRC Request social interaction, by customizing the action performing communicative interaction. Different characteristics are *inherited*: the content parameters and the request protocol. Similarly, CRC Requester and Requestees inherit the CAs. Some features of the action performing interaction are not inherited, but *overridden* by specializing components. For instance, the *evalRequest* action that CRC Requestees perform as a special kind of requestee, is overriden by the *evalCRCRequest* action, which represents the particular way in which a request will

[6] Note that these reusable components are actually *connectors*, in the C&C perspective.

[7] This recursive model applies not only to interactive roles and social interactions, but also to any kind of meta-entity of the $\mathcal{RICA}$ metamodel: group types, protocol types, and so forth.

be evaluated in this context. In general, social actions may either declare a default *execution* method, or be *abstract*, so that particular agents must provide the actual implementation. In the case of the *evalCRCRequest* action, a default implementation may be provided whereby the request will be accepted if the requestee is actually the author of the paper. Overriding declarations of role actions introduces a *dynamic binding* feature in the execution semantics of $\mathcal{RICA}$ models. For instance, when CRC Requestee agents are required to perform the *evalRequest* action (which is legal, since they are requestee agents), the action that is actually executed at run-time is *evalCRCRequest*[8].

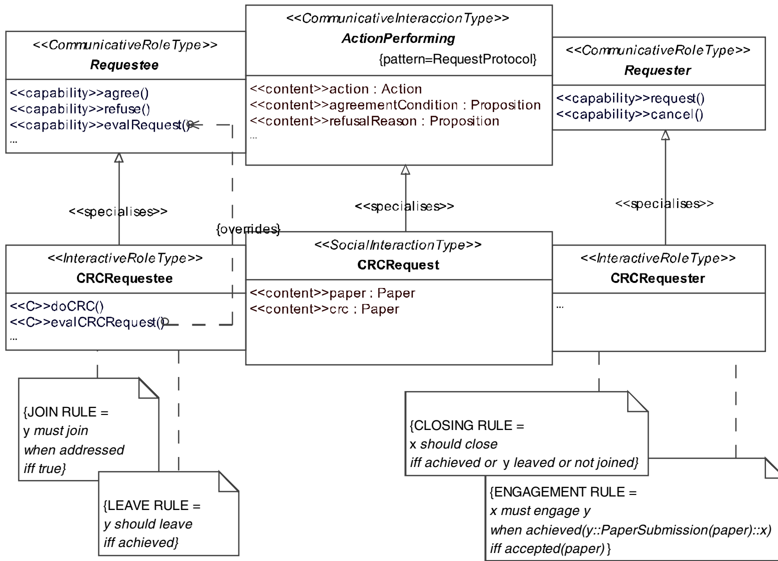

Fig. 3. Recursive definition of *CRCRequest* interactions

The model of figure 3 also illustrates the case of *extensions*: e.g. the *doCRC* action extends the inherited and overridden set of actions that CRC Requestee agents can perform. As another example of extension (involving abstract features of the parent type), the social interaction type also declares the engagement, closing, joining and leaving rules. These rules are represented in figure 3 as tagged values inside notes. Engagement and closing rules are defined for CRC Requesters, while joining and leaving are defined for CRC Requestees:

– *Engagement rule*: The set up may be established when a paper submission interaction finishes successfully and the paper was accepted. If so happens,

[8] This mechanism allows for a direct reuse of interaction protocols, fully specified in the scope of communicative interactions. For instance, the *Interaction State Machine* (ISM) specification of the request protocol [16] may be reused "as-is" by the CRC request interaction.

the paper submittee (a PC Chair, as declared in figure 1), must engage the paper submitter (an author) to play the requester role and obtain the CRC of the accepted paper[9].

- *Joining rule*: An author must join the interaction as a requestee when it is addressed by the PC Chair.
- *Leaving rule*: An author should leave the interaction when the CRC is provided (i.e. the interaction's purpose is achieved).
- *Closing rule*: A PC Chair should close the interaction if and only if the CRC has been provided, the author leaves the interaction before finishing its job, or the addressed author does not join before the established deadline for the delivery of the CRC.

Finally, it should be noted that communicative interactions may be reused to specify other communicative interactions as well. For example, the *Submission* communicative interaction may be defined as an specialisation of *Action Performing* interactions. Moreover, taking into account that a *submit* CA is a special kind of *request*, the generic *RequestProtocol* may be used in place of particular submission protocols.

4 The $\mathcal{RICA}$–$\mathcal{J}$ Framework

The $\mathcal{RICA}$ theory, given its metamodel and execution semantics, can be conceived as a programming language with close links to Architectural Description Languages (ADLs) [13]. For instance, the CRC Requestee role type may be declared as shown in figure 4. Unlike its UML representation (see figure 3), the grammatical form also conveys the full declaration of social actions (parameters and execution blocks). Dotted lines should be replaced by actual code, possibly object-oriented (e.g. Java). The reserved words `player` and `interaction` play a similar role to the reserved word `this` in Java: the first one denotes the agent which is playing that role instance (an *Author* agent, in the example); the second one, the interaction to which the agent is connected as a player of the interactive role type (an instance of *CRCRequest*, in the example). Finally, the functions `achieved` and `addressedFor` denote predefined operations of social interaction and agent instances, respectively.

As a more pragmatic alternative to the direct instrumentation of the "$\mathcal{RICA}$ programming language", the $\mathcal{RICA}$–$\mathcal{J}$ ($\mathcal{RICA}$-JADE [17]) framework instruments the $\mathcal{RICA}$ theory on top of the FIPA-compliant JADE platform [18], which is used as the underlying middleware and programming environment. This section will first describe the $\mathcal{RICA}$–$\mathcal{J}$ architecture and general features based on the execution semantics described in section 2. Then, we outline how the the reusability concern described in section 3 is captured in the framework.

[9] In general, the engagement rule may not specify a particular agent as the addressee but a definite description representing a collection of agents. The engagement rules for paper submission interactions, as suggested previously, may establish that the author must engage *any* agent playing the *PC Chair* role in the submission group.

```
interactive role type CRCRequestee specialises Requestee
must be joined by Author when player.addressedFor(CRCRequest)
should be leaved iff interaction.achieved()
performs{
    social action type evalCRCRequest(in paper)
    overrides evalRequest
    executes{ ... }

    social action type doCRC(in paper, out crc)
    executes{ ... }
}
```

Fig. 4. Grammatical declaration of the *Seller* role

4.1 $\mathcal{RICA}$–$\mathcal{J}$ Architecture

The $\mathcal{RICA}$–$\mathcal{J}$ framework extends the JADE platform with a layer that provides a virtual machine based on the $\mathcal{RICA}$ abstractions. This layer is decomposed into two major modules, implemented by the `rica.reflect` and `rica.core` Java packages, which instrument the $\mathcal{RICA}$ metamodel and execution semantics, respectively. Thus, the former package includes the classes `InteractiveRoleType`, `SocialInteractionType`, etc., while the latter contains abstract types, such as `Agent`, `InteractiveRole` and `SocialInteraction`. The `rica.reflect` classes are functional analogues of the reflective classes of the standard `java.lang.reflect` package. Moreover, they ensure the consistency of the programmed $\mathcal{RICA}$ model (e.g. that any social role specializes a communicative role). On the other hand, `rica.core` classes may be seen as analogues of the `java.lang.Object` class since, for instance, all particular agent types must be programmed by extending the `rica.core.Agent` class.

The `rica.core` classes map the common behaviour and structure of agents, roles, etc., as defined by the $\mathcal{RICA}$ execution semantics, to the supported abstractions of the JADE framework: basically, agents, behaviours and ACL messages. The resulting architecture of a $\mathcal{RICA}$–$\mathcal{J}$ agent is exemplified in figure 5. This figure's object model depicts the run-time structure of the a_3 author agent previously shown in figure 2.

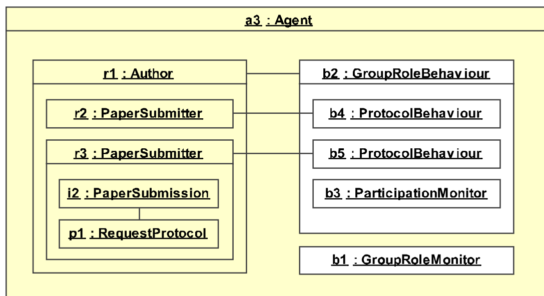

Fig. 5. $\mathcal{RICA}$–$\mathcal{J}$ agent architecture

Any `rica.core.Agent` (a kind of JADE agent) schedules an `GroupRoleMonitor` (a JADE cyclic behaviour), in charge of creating and deallocating the group role instances in which its functionality is decomposed. The activation of a given group role has two major consequences: firstly, the agent makes public in a role-based Directory Facilitator (actually, a wrapper of the `jade.domain.DFService`) that it currently plays that role; secondly, the `GroupRoleBehaviour` (a JADE parallel behaviour) managed by the `rica.core.GroupRole` instance is scheduled. This parallel behaviour contains a `ParticipationMonitor` behaviour[10], in charge of enacting new interactions (through engagement) or joining the agent to interactions initiated by other agents. When some agent initiates its participation in a new interaction, it will instantiate the corresponding `rica.core.InteractiveRole` subclass, together with the JADE behaviour which manages its participation in the interaction. This behaviour will be part of the group role's parallel behaviour.

According to the execution semantics, the interactive role behaviour determines the actions that will be executed, observed or interrupted by the agent in the course of the interaction. As far as CAs are concerned, their execution will result in the automatic creation and sending of the corresponding `jade.lang.acl.ACLMessage`. Conversely, the observation of CAs results in receiving an `ACLMessage` and automatically translating it back to a `rica.core.CommAction` object. These conversions can be performed automatically by relying on the content parameters, the JADE ontology, and the addressee language that the interaction instance holds. The `ProtocolBehaviour` class models a generic interactive role behaviour which determines the agent's activity according to the rules established by a protocol which regulates the interaction. This behaviour is decoupled from the particular formalism used to specify the given protocol, since it only depends on the `rica.core.Protocol` interface (which declares the protocol services specified by the execution semantics). Particular formalisms may be integrated into the framework by instrumenting their execution semantics in a `Protocol` subclass.

4.2 Programming in $\mathcal{RICA}$–$\mathcal{J}$

The architecture of a MAS in the $\mathcal{RICA}$–$\mathcal{J}$ framework, shown in figure 6, is structured around two major types of modules: the first one refers to the implementation of the different agents participating in the MAS; the second one, to the implementation of the MAS organization. This composite module closely follows the structure of its $\mathcal{RICA}$ model: in essence, we may identify an optional *protocol formalism* module, one mandatory *communication* module, and the implementation of the *application-dependent* social interactions and non-interactive roles.

As we argued in section 3, communicative components are highly reusable components, so that they will be likely reused from an application-independent

[10] This cyclic behaviour acts as an *interaction factory*: since $\mathcal{RICA}$–$\mathcal{J}$ interactions are instrumented *subjectively*, each participant (initiator or not) holds a `rica.core.SocialInteraction` instance representing the interaction from its own perspective.

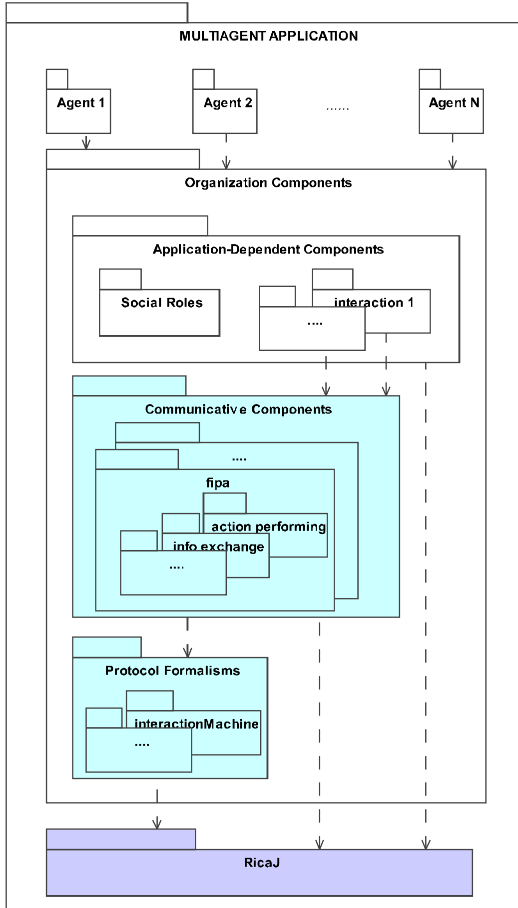

Fig. 6. Architecture of a $\mathcal{RICA}$–$\mathcal{J}$ MAS application

library of communicative interactions. This library, implemented under the `acl` package, currently contains some of the FIPA ACL underlying interactions [4], and other "non-standard" interactions such as submission and advisement interactions. On the other hand, protocol formalisms are highly reusable as well. The `protocol.ism` package instruments the *Interaction State Machine* specification technique [16]. Therefore, the protocol and communication module will be likely implemented by *component developers*, whereas the application-dependent social module would be in charge of *organization developers*. Finally, *independent users* would be in charged of implementing their agents. These programmers rely on the components available in the organizational library, possibly customizing the roles types to be played by overriding their default functionality or implementing their abstract actions. The following paragraphs will briefly describe some general guidelines in the implementation of communicative and social interaction components.

Communicative Components. Above all, communicative roles and interactions encapsulate the CAs and protocols of their characteristic ACL dialect. However, the Java classes which instrument these components also provide basic support functionality for the execution semantics of these components. Firstly, the constructors of the social interaction classes may provide the required initialization of the content parameters and participant addresses. Secondly, the generic vocabulary of the communicative interaction (including the performatives), will be implemented as a JADE ontology. Finally, generic social actions defined by communicative roles may be provided with default implementations.

For instance, figure 7 partially shows the implementation of the *Requestee* communicative role type. The $\mathcal{RICA}$ type information is embedded in the Java class by means of `public static final` fields, following the established implementation scheme for communicative roles types: a field of type `CommInteractionType` declares the type of communicative interaction to which the role type belongs, whereas role capabilities are declared by fields whose types are assignable from the `SocialActionType` class (so, `CommActType` fields will contribute to the role capabilities, since this class extends the former one). Furthermore, the class constructor allows the player agent to register the JADE ontology defined for action performing interactions. This ontology declares the performatives (e.g. *agree*, *refuse*, etc.) and other generic concepts and predicates (e.g. the predicate `CanNotPerform`, useful when the kind of requested action is not among the capabilities of the requestee agent). Note that the java class is declared `abstract`, since the rules for joining and leaving this kind of roles are not declared.

Figure 7 also shows a partial implementation of the *EvalRequest* social action. Similarly to the *Requestee* role type, the static features of the action type are declared by `static` fields: e.g. the `input` and `output` fields declares the input and output parameters of the action, which are initialised with the corresponding parameter types declared by the *ActionPerforming* communicative interaction. For each kind of *parameter*, a private field `parameterValue` of the corresponding type is declared, together with a pair of `get/set` methods to access and set the value of the parameter. The implementation of accessor methods for output parameters provide default values which may be overriden by specialisations of this action type. Thus, according to the `getRefusalReason` method, the request will be refused if the requested action can not be performed by the requestee agent (the method `getPerformer()` will return the role instance which is actually executing the action). A default value is also provided for the *notificationRequired* parameter, indicating that the agreement to perform the action should be notified (in particular requests, this default value might be overriden to *false* if the performance of the requested action is imminent). Finally, a default implementation of the `Action`'s `execute()` method is also provided: the action will be considered *successfully executed* if a refusal reason or the agreement condition has been set; otherwise, the method returns in a *suspended* state.

Social Components. Interactive roles and social interactions extend communicative roles and interaction classes, thus inheriting the general interaction management mechanism. Typically, they will provide the engagement, joining,

```
public abstract class Requestee extends InteractiveRole{

    /** Type info. */
    public static final CommRoleType type = new CommRoleType(Requestee.class);
    public static final CommInteractionType interaction = ActionPerforming.type;
    public static final SocialActionType evalRequest = EvalRequest.type;
    public static final CommActType agree = Agree.type;
    public static final CommActType refuse = Refuse.type;
    ...

    /** Creates a requestee role for the specified agent. */
    public Requestee(Agent agent){
        super(agent);
        updateDomainOntology(ActionPerformingOntology.getInstance());
        ...
    }
}

public class EvalRequest extends SingleSocialAction{

    /** Type info. */
    public static final SingleSocialActionType type =
        new SingleSocialActionType(EvalRequest.class);
    public static final ParameterType[] input =
        new ParameterType[]{ActionPerforming.action};
    public static final ParameterType[] output =
        new ParameterType[]{ActionPerforming.notificationRequired,
                            ActionPerforming.agreementCondition,
                            ActionPerforming.refusalReason};

    /** Auxiliary methods for input/output parameters */
    private SocialAction actionValue;
    public void setAction(SocialAction newValue){...}
    public SocialAction getAction(){return actionValue;}

    private Predicate refusalReasonValue;
    public void setRefusalReason(Predicate newValue){...}
    public Predicate getRefusalReason(){
        if (refusalReasonValue == null &&
            !getAction().getType().canBePerformedBy(getPerformer().getType())){
            setRefusalReason(new CanNotPerform());
        }
        return refusalReasonValue;
    }

    private Predicate agreementConditionValue;
    public void setAgreementCondition(Predicate newValue){...}
    public Predicate getAgreementCondition(){return agreementConditionValue;}

    private Boolean notificationRequiredValue;
    public void setNotificationRequired(Boolean newValue) {...}
    public Predicate getNotificationRequired(){
        if (notificationRequired==null){
            setNotificationRequired(new Boolean.TRUE)
        }
        return notificationRequired;
    }

    /** Overriden SocialAction interface */
    public ExecutionState execute(){
        if (getRefusalReason()!=null || getAgreementCondition()!=null){
            return ExecutionState.SUCCESSFUL;
        }else{
            return ExecutionState.SUSPENDED;
        }
    }
}
```

Fig. 7. Implementation of the *Requestee* communicative role type

leaving and closing rules by overriding/declaring the corresponding methods: `mustBeJoinedBy`, `shouldBeLeft`, etc. Moreover, the Java classes will also provide the social interaction ontology, application-specific implementations of general

```
public class CRCRequestee extends Requestee{

    /** Type info. */
    public static final InteractiveRoleType type =
       new InteractiveRoleType(CRCRequestee.class);
    public static final SocialInteractionType interaction = CRCRequest.type;
    public static final SocialActionType evalCRCRequest = EvalCRCRequest.type;
    public static final SocialActionType doCRC = DoCRC.type;
    ...

    /** Overriden Interactive Role interface. */
    public static SocialInteraction mustBeJoinedBy(GroupRole role){
        if (role.getType()!=Author.type){
            return null;
        }else{
            return role.addressedFor(CRCRequest.type);
        }
    }

    public boolean shouldBeLeft(){
        return getInteraction().hasBeenSuccessful();
    }

    /** Creates a CRCRequestee role for the specified agent. */
    public CRCRequestee(Agent agent){
        super(agent);
        updateDomainOntology(CRCRequestOntology.getInstance());
    }

}

public class EvalCRCRequest extends EvalRequest{

    /** Type info */
    public static final SingleSocialActionType type =
        new SingleSocialActionType(EvalCRCRequest.class);
    public static final ParameterType[] input =
        new ParameterType[]{CRCRequest.paper};

    /** Auxiliary methods for input/output parameters */
    private Paper paperValue;
    public void setPaper(Paper newValue){...};
    public Paper getPaper(){ return paperValue; }

    /** Overriden Requestee interface */
    public Predicate getRefusalReason(){
        if (refusalReasonValue == null &&
            !getPaper().isAuthor(getPlayer().getAID())){
            setRefusalReason(new NotAuthor());
        }
        return refusalReasonValue;
    }

    public Predicate getAgreementCondition(){
        if (agreementConditionValue == null && getRefusalReason()==null){
            setAgreementCondition(new TrueProposition());
        }
        return agreementConditionValue;
    }

}
```

Fig. 8. Implementation of the *CRCRequestee* interactive role type

communicative role methods, and all other code concerning the actual environment in which the application is deployed (database connection, web servers, etc.).

As figure 8 shows, the implementation of the *CRCRequestee* role type firstly includes the declaration of the static features of the $\mathcal{RICA}$ type: essentially, the set of capabilities, which is extended with the *DoCRC* action; moreover, the

EvalRequest action is overriden by its specialisation *EvalCRCRequest*[11]. The method `mustBeJoined` declares the joining rule for *CRCRequestee* roles according to the specification discussed in section 3.2. This method is invoked by the group role playing the interactive role, i.e. an *Author* instance, in this particular case. If the return value is not `null`, the author agent will join the specified interaction and will instantiate the `CRCRequestee` class to carry out its activity within that interaction. The leaving rule is implemented by overriding the `shouldBeLeft` method, declared by the `rica.core.InteractiveRole` class. It returns `true` if, and only if, the interaction has finished successfully. Finally, the `CRCRequestee` constructor ensures that the specific ontology for this kind of interactions is registered in the agent's content manager. This ontology declares, for example, the proposition `NotAuthor`, which stands for the fact that the requestee is not the author of the paper specified in the interaction.

Figure 8 also shows a partial implementation of the *EvalCRCRequest* social action. The `EvalCRCRequest` class overrides some of the default methods specified by the `EvalRequest` class. Specifically, the request will be refused if the player agent is not the author of the input paper. Moreover, the agreement condition is automatically set to *true* if the condition for refusal is not satisfied. Thus, according to the generic implementation of the inherited `execute` method, the action will succeed the first time is performed.

5 Conclusion

This paper has shown how agent interactions can be modeled and instrumented by customizing generic communicative interactions identified from ACL-dialects. Communicative interactions serve as micro-organizational modeling patterns that structure the interaction space of specific MAS, complementing similar reuse-approaches based on macro-organizational structures [8] or agent components [9]. Communicative interactions are also the key computational abstraction in the $\mathcal{RICA}$–$\mathcal{J}$ programming framework, as their execution semantics determines a substantial part of the logic required to manage agent interactions. The encapsulation of these interactions around software component libraries significantly simplifies the implementation of the multi-agent organization. Moreover, the $\mathcal{RICA}$–$\mathcal{J}$ framework also relieves agent programmers from the implementation of low-level issues concerning the dynamics of agents within the organization. On the other hand, it should be stressed that the proposed approach does not endanger the autonomy of agents, since the social roles available in the organization library may be fully customized to account for the particular requirements of each agent.

The currently implemented library of communicative interactions may be extended to cover dialects proposed for other specific domains (e.g. negotiation [19, 20]), or the dialogue types put forward by argumentation theorists [21–24].

[11] If some action of the super-role is specialised by a new action of the derived role type, the super-action is implicitly overriden. This is a limitation of the current implementation.

Note that the implementation of communicative interactions by means of the $\mathcal{RICA}$–$\mathcal{J}$ framework is independent of any semantic paradigm, be it intentional [25], social [26], or protocol-based [27]. In fact, BDI or commitment-based agent architectures may be instrumented as refinements of the general C&C agent architecture, thus complementing the protocol-based semantics that the $\mathcal{RICA}$–$\mathcal{J}$ framework currently instruments.

Another contribution of this paper refers to the C&C perspective on MAS by defining the execution semantics of the $\mathcal{RICA}$ theory. This specification, albeit informal, shares the motivations of the formal operational semantics of groups and role dynamics established by Ferber & Gutknecht [6], and Dastani et al. [28]. Furthermore, the C&C-based perspective that we have put forward may well be extended to specify the execution semantics of this and related larger-grained organizational abstractions, such as scenes [5]. Since these abstractions can be ultimately reduced to different types of social interactions, they may be conceived as composite connectors [29]. Moreover, modeling social interactions in terms of software connectors has as a major consequence the identification of the characteristic roles that their participant agents may play within it. This feature of the $\mathcal{RICA}$ metamodel allows to distinguish it from other organizational approaches, and makes possible the reuse approach to social interactions put forward by this conceptual framework.

We have shown how the $\mathcal{RICA}$–$\mathcal{J}$ framework instruments the $\mathcal{RICA}$ execution semantics on top of the JADE platform, but other agent infrastructures (e.g. tuple-based [9]), or technologies (e.g. web services), may provide the required underlying middleware services and basic abstractions as well. On the other hand, the programming language perspective on the $\mathcal{RICA}$ theory complements the results on the field of agent-oriented programming languages, currently geared towards deliberative or cognitive capabilities of agents [30, 31]. Moreover, by placing MAS in the broader spectrum of software architectures, this paper motivates the transfer of research from this field (e.g. on ADLs [13]).

Future work will concentrate on further validation of the $\mathcal{RICA}$–$\mathcal{J}$ framework with the final intention to get a JADE add-on release. The extension of the underlying metamodel with coarse-grained organizational abstractions, and the instrumentation of the interaction monitoring and compliance capabilities that any open-driven framework must offer [2], will be considered as well.

References

1. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering and Methodology **12** (2003) 317–370
2. Singh, M.P.: Agent-based abstractions for software development. In Bergenti, F., Gleizes, M.P., Zambonelli, F., eds.: Methodologies and Software Engineering for Agent Systems. Kluwer (2004) 5–18
3. Ferber, J., Gutknecht, O.: A meta-model for the analysis of organizations in multi-agent systems. In Demazeau, Y., ed.: Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98), Paris, France, IEEE Press (1998) 128–135

4. Serrano, J.M., Ossowski, S., Fernández, A.: The pragmatics of software agents - analysis and design of agent communication languages. Intelligent Information Agents - An AgentLink Perspective (Klusch, Bergamaschi, Edwards & Petta, ed.), Lecture Notes in Computer Science **2586** (2003) 234–274
5. Esteva, M., Rodriguez, J.A., Sierra, C., Garcia, P., Arcos, J.L.: On the formal specifications of electronic institutions. In Dignum, F., Sierra, C., eds.: Agent-mediated Electronic Commerce (The European AgentLink Perspective). Volume 1191 of LNAI., Berlin, Springer (2001) 126–147
6. Ferber, J., Gutknecht, O.: Operational semantics of a role-based agent architecture. In Jennings, N.R., Lesperance, Y., eds.: Intelligent Agents VI. Proceedings of the 6th Int. Workshop on Agent Theories, Architectures and Languages. Volume 1757 of LNAI., Springer (1999)
7. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley (1997)
8. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Organizational abstractions for the analysis and design of multi-agent systems. In Ciancarini, P., Wooldridge, M.J., eds.: AOSE. Volume 1957 of LNCS. Springer (2000) 235–252
9. Bergenti, F., Huhns, M.N.: On the use of agents as components of software systems. In Bergenti, F., Gleizes, M.P., Zambonelli, F., eds.: Methodologies and Software Engineering for Agent Systems. Kluwer (2004) 19–31
10. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology **6** (1997) 213–249
11. Dignum, V., Vázquez-Salceda, J., Dignum, F.: Omni: Introducing social structure, norms and ontologies into agent organizations. In Bordini, R., Dastani, M., Dix, J., Seghrouchni, A., eds.: Programming Multi-Agent Systems Second International Workshop ProMAS 2004. Volume 3346 of LNAI., Springer (2005) 181–198
12. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: An organizational view of multi-agent systems. In: AOSE. (2003)
13. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In Leavens, G.T., Sitaraman, M., eds.: Foundations of Component-Based Systems. Cambridge University Press (2000) 47–68
14. Bretier, P., Sadek, D.: A rational agent as the Kernel of a cooperative spoken dialogue system: Implementing a logical theory of interaction. In Müller, J.P., Wooldridge, M.J., Jennings, N.R., eds.: Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III. Volume 1193 of LNAI., Berlin, Springer (1997) 189–204
15. Foundation for Intelligent Physical Agents: FIPA Interaction Protocol Library Specification. http://www.fipa.org/repository/ips.html (2003)
16. Serrano, J.M., Ossowski, S.: A semantic framework for the recursive specification of interaction protocols. In: Coordination Models, Languages and Applications. Special Track of the 19th ACM Symposium on Applied Computing (SAC 2004). (2005)
17. Serrano, J.M.: The RICAJ framework. `http://platon.escet.urjc.es/~jserrano` (2005)
18. JADE: The JADE project home page. `http://jade.cselt.it` (2005)
19. Wooldridge, M., Parsons, S.: Languages for negotiation. In Horn, W., ed.: Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000), Berlin, IOS Press (2000) 393–397

20. Sierra, C., Jennings, N.R., Noriega, P., Parsons, S.: A framework for argumentation-based negotiation. In Singh, M.P., Rao, A., Wooldridge, M.J., eds.: Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97. Volume 1365 of LNAI., Berlin, Springer (1998) 177–192
21. McBurney, P., Parsons, S.: A formal framework for inter-agent dialogues. In Müller, J.P., Andre, E., Sen, S., Frasson, C., eds.: Proceedings of the Fifth International Conference on Autonomous Agents, Montreal, Canada, ACM Press (2001) 178–179
22. Lebbink, H., Witteman, C., Meyer, J.J.: A dialogue game to offer an agreement to disagree. In Bordini, R., Dastani, M., Dix, J., Seghrouchni, A., eds.: Programming Multi-Agent Systems Second International Workshop ProMAS 2004. Volume 3346 of LNAI., Springer (2005)
23. Amgoud, L., Maudet, N., Parsons, S.: Modelling dialogues using argumentation. In: E. Durfee, editor, Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS-2000), Boston, MA, USA, IEEE Press (2000) 31–38
24. Walton, D.N., Krabbe, E.C.W.: Commitment in Dialogue. State University of New York Press (1995)
25. Cohen, P.R., Levesque, H.J.: Communicative actions for artificial agents. In Lesser, V., ed.: Proceedings of the First International Conference on Multi–Agent Systems, San Francisco, CA, MIT Press (1995) 65–72
26. Singh, M.P.: A social semantics for agent communication languages. In Dignum, F., Greaves, M., eds.: Issues in Agent Communication. LNAI, vol. 1916. Springer (2000) 31–45
27. Pitt, J., Mamdani, A.: A protocol-based semantics for an agent communication language. In Thomas, D., ed.: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol1), S.F., Morgan Kaufmann Publishers (1999) 486–491
28. Dastani, M., van Riemsdijk, B., Hulstijn, J., Dignum, F., Meyer, J.J.: Enacting and deacting roles in agent programming. In Odell, J., Giorgini, P., Müller, J.P., eds.: Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004,. Volume 3382 of Lecture Notes in Computer Science., Springer (2004)
29. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of the 22nd International Conference on Software Engineering, ACM Press (2000) 178–187
30. Hindriks, K.V., Boer, F.S.D., der Hoek, W.V., Meyer, J.J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems **2** (1999) 357–401
31. Shoham, Y.: Agent-oriented programming. Artificial Intelligence **60** (1993) 51–92