On the Impact of Agent Communication Languages on the Implementation of Agent Systems*

Juan Manuel Serrano and Sascha Ossowski

Artificial Intelligence Group
Department of Computing, ESCET
University Rey Juan Carlos
{jserrano,sossowski}@escet.urjc.es
http://www.ia.escet.urjc.es

Abstract. Today's software platforms that support the construction of agent systems in accordance with the FIPA specifications essentially provide enabling infrastructure services, and are still far away from adequately rendering support to current methodologies and theories for building agent systems, especially when social concepts play a significant role. Elsewhere, we have presented the $\mathcal{RICA}$ theory, that smoothly integrates relevant aspects of Agent Communication Languages and Organisational Models, so as to provide guidelines for agent system design. This paper explores the impact of this theory on the actual development and implementation of agent-based applications. For this purpose, the $\mathcal{RICA}$ metamodel is considered as a programming language, in which roles, interactions, communicative actions, etc., are first-class language entities. We show how this language can be effectively implemented as a software framework that extends the JADE platform, and provide an example that illustrates its potential.

1 Introduction

In the last decade, the development of software agent applications has largely benefited from the standardisation efforts made by the Foundation for Intelligent Physical Agents (FIPA) [1], as well as from the many software platforms that implement the FIPA specifications (e.g. [2, 3]). These platforms, however, provide rather basic infrastructure services, and are still far away from adequately rendering support to current methodologies and theories for building agent systems [4]. This is particularly true for approaches that set out from a social-level analysis [5], such as the Role/Interaction/Communicative Action ($\mathcal{RICA}$) theory [6, 7], which smoothly integrates relevant aspects of Agent Communication Languages (ACL) [8, 9] and Organisational Models (OM) [10, 11] by introducing the concepts of Communicative Roles and Interactions, so as to provide guidelines for agent system specification.

This paper explores the impact that the RICA metamodel has on the design and, in particular, on the implementation of agent-based applications: it shows how agent system programming can be leveraged by relying on roles and interactions as first-class programming constructs, and by structuring these abstractions in terms of ACL

* Research sponsored by the Spanish Ministry of Science and Technology (MCyT), project TIC2003-08763-C02-02

components. The paper is organised as follows: Section 2 outlines the fundamentals of the RICA theory in terms of programming language metaphor. In Section 3, we present the architecture and design of the RICA-J prototype, a software framework implemented on top of the JADE platform [2]. Section 4 gives an example for the use of this framework, which illustrates how an important part of agent system programming gives rise to, and may rely on, highly-reusable libraries of communicative roles and interactions. Finally, we present the lessons learnt from this enterprise.

2 The $\mathcal{RICA}$ theory

The $\mathcal{RICA}$ theory provides a conceptual framework which establishes a systematic link between the specifications of the organisational model and the ACL of a multiagent application. According to the theory, the specification of the organisational model comprises the definition of entities such as agents, roles and interactions, while the specification of the ACL deals with the definition of communicative actions and protocols. The $\mathcal{RICA}$ theory is made up of two major components: a metamodel, which defines the language used to specify the different *types* of organisational and communicative entities; and a specification of the structure and behaviour of these entities.

2.1 The $\mathcal{RICA}$ metamodel

The $\mathcal{RICA}$ metamodel is outlined by the UML class diagram of figure 1. The relations between its entities are subject to a number of constraints which will be discussed in the sequel.

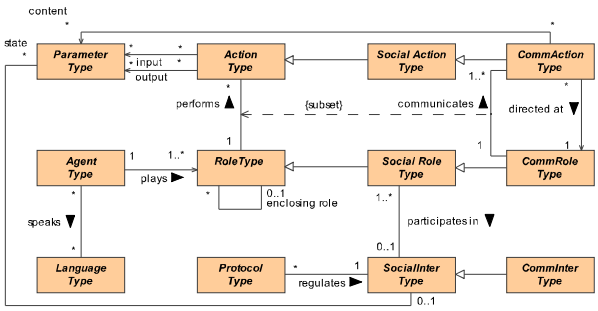

Fig. 1. The $\mathcal{RICA}$ metamodel

The behaviour of *agents* manifests itself in the form of different *roles*. So, the definition of a (type of) agent basically requires the specification of a collection of (types of) roles[1]. Role definitions required the specification of a collection of *actions*, which represent the specific tasks to be performed as part of the role's functionality. Moreover, roles

[1] In the sequel, the qualification "type of" will be omitted whenever the context makes clear that we are talking about types and not instances.

which delegate part or all of their functionality to other roles are called *enclosing roles*. The definition of an action requires, in turn, the specification of its types of input and output *parameters*. The $\mathcal{RICA}$ theory establishes a distinction between generic roles and so-called *social roles*, a *kind* of role which represents the functionality of agents participating in *social interactions*. The state of a social interaction is represented by a collection of parameters.

The distinction between generic actions and, so-called, *social actions*, parallels the one established for roles. Input/output parameters defined for a social action must have been declared as state parameters of its corresponding type of interaction. In this way, the state of social interactions will be updated as a result of the execution of social actions. *Communicative actions* are conceived as special kinds of social action performed with the intention to be observed by agents playing a specific role [12]. The propositional content parameters of a communicative action simply consists of its input parameters. There are actually two parallel hierarchies of actions, corresponding to disjunctive and single actions, respectively. The FIPA *inform-if* and *request* communicative actions are examples of disjunctive and single communicative actions, respectively. In the context of an interaction, *protocols* are optional artefacts that prescribe the (social) actions that agents must perform. According to the $\mathcal{RICA}$ metamodel, a protocol must identify the type of regulated interaction (which, in turn, constrains the roles, actions, etc. that the protocol can refer to), and may be augmented by whatever protocol specification technique (AUML sequence diagrams, Petri-nets, interaction machines, etc.).

The following example, which closely follows a similar one found in the JADE distrubtion, illustrates the use of the $\mathcal{RICA}$ concepts as a conceptual framework for modelling multiagent applications. Consider a simple e-commerce setting. *Purchase* interactions are established between pair of agents, playing respectively the roles of *buyer* and *seller*. The parameters declared for this type of interaction include the item to be purchased, the credit card used in the payment, its successful status, etc. Buyers may perform different FIPA communicative actions. For instance, they may *request* the seller to *sell* some item (a type of social action), or *cancel* the selling action once it has been initiated. Furthermore, sellers may *evaluate* (a type of social action) whether the sell action can be performed, and under which conditions the item should be sold. Sellers may also take communicative actions to *agree* or *refuse* to sell an item, etc. The *FIPA Request* protocol is used as a purchase protocol. *client* agents playing the role of buyers, and *clerk* agents act as sellers, both using *FIPA ACL/FIPA SL* as their communication language.

The remaining entities of the $\mathcal{RICA}$ metamodel in 1, *communicative roles* and *communicative interactions*, establish the link between communicative and organisational concepts. The rationale behind these particular kinds of social roles and interactions relates to reusability matters: given that requests and cancellations, for instance, might be declared for other roles different from the buyer role, it makes sense to define a generic role, named *requester*, encapsulating the declaration of these actions. Then, the seller role, as well as any other role requiring these CAs as part of its characteristic behaviour, may simply reuse their declaration from the requester role. Similarly, the FIPA Request protocol may be attached to a generic social interaction which declares the interaction parameters and communicative roles required by the actions included in the protocol.

Social roles that are defined in this manner (in terms of communicative actions) are called communicative roles. In a similar way, social interactions, defined by protocols and communicative roles, are called communicative interactions.

Communicative roles and interactions encapsulate the whole pragmatic competence of agents. So, non-communicative social roles must directly or transitively specialise from, or delegate their communicative competence to, some communicative role. Other communicative entities in the e-commerce example may include the *requestee* communicative role, encapsulating the declaration of the *agree* and *refuse* CAs, and *action performing* communicative interactions, which can be regulated, among others, by the FIPA Request protocol. The parameters declared for this type of interaction, include the action to be performed, the successful status of the interaction, etc. Note that communicative roles may also define non-communicative social actions. For instance, the evaluation action identified for the seller role could actually be a social action declared for the generic requestee role.

Figure 2 shows an instantiation of the $\mathcal{RICA}$ metamodel for the e-commerce application, using an specialisation pattern. The model is depicted as a UML class diagram, using stereotypes to indicate the type of each declared element. Parameters and the language spoken by agents are represented as attributes, while actions and social roles are modelled after UML operations and the interface classifier, respectively. Concerning the instances of the $\mathcal{RICA}$ relationships, they are represented using UML associations, except to model specialisations, where the UML generalisation is used instead. Note that the declaration of the action parameters, which would be represented by operation attributes, have been omitted from the diagram.

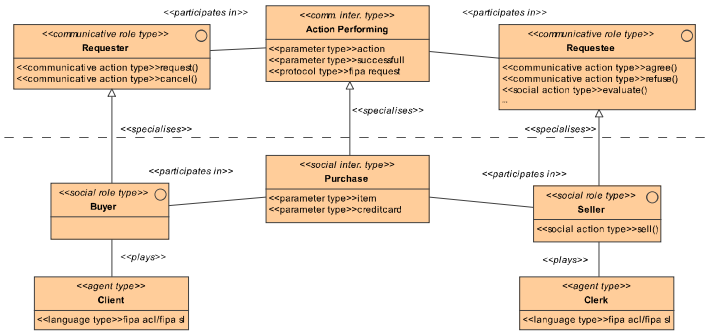

Fig. 2. The organisational model of a simple e-commerce application

2.2 Structure and behaviour of $\mathcal{RICA}$ entities

Figure 3 shows a static view of the relationships which hold between the different instances of the $\mathcal{RICA}$ entities at a given run-time slice. *Agents*, *roles*, *social interactions*, etc. are considered as the most generic types of entities, specifying the common

structure and behaviour of its instances. Thus, they play the same role as the java class `Object`, which any programmed class has as super-class.

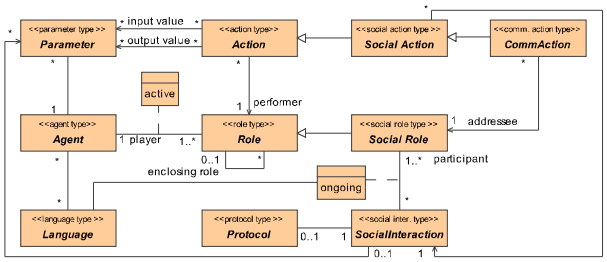

Fig. 3. Structure of $\mathcal{RICA}$ entities

As shown in figure 3, at a given time, agents may be concurrently *playing* several roles[2]. A number of other roles, qualified as *abandoned*, may have previously been played. The predefined behaviour common to all agents is characterised by examining, whenever an agent is created or some event occurs, its collection of roles in order to identify abandoned roles that must be wake-up, and active roles that must be abandoned.

Roles whose type declarations include the definition of some enclosing role set up a link to the latter at the time of their instantiation. When a role is played or abandoned, its characteristic behaviour is activated or deactivated, respectively. In general, the behaviour of some role, which specifies the conditions that must be met in order to perform its declared actions, is defined by its type. However, pure enclosing roles (whose funcionality is completely split into other roles) have a predefined behaviour which defaults, similarly to agents, to periodically examining its enclosee roles in order to update the collections of active and abandoned roles of the player agent. Social roles for which some interaction type is defined, have a predefined behaviour as well. Besides examining their enclosee roles, they monitor the conditions defined for initiating interactions of the declared type. In addition, they go through the ongoing interactions to see if some of them must be finished. At any time, an agent playing a social role, may be participating in multiple on-going interactions in accordance with the role's type. Interactions can finish prematurely, because of some abnormal execution or by an explicit abortion, or upon completion (either successfully or unsuccessfully).

The initiating conditions as well as the general agent behaviour within some interaction is specified by its type declaration. However, those interactions regulated by a collection of protocols will delegate by default these responsibilities to them. Hence, protocol types are required to define their particular initiating conditions, as well as the logic required to identify which activity a given role must perform at any given time. The behaviour of a role in the context of an interaction regulated by some protocol defaults to performing the required activities until the protocol itself signals the completion of the interaction. The activities prescribed by protocols comprise the (concurrent

[2] The types of these roles must obviously conform to the declaration of the agent type. This kind of type conformance applies throughout the following discussion.

or sequential) performance, observation, or abortion of social actions. The *performance* of any kind of action (social or non-social) requires a collection of consistently instantiated input parameters, while its result is conveyed through the output parameters. Social actions are executed in the context of a particular interaction, in such a way that its input and output parameters values refer to the parameters representing the actual state of the interaction. In general, the particular activity performed by some agent when executing some action must be defined by its corresponding type. Communicative actions, though, have a predefined activity, which consists of sending a message to some agent's role in any language understood by the receiver agent (e.g. FIPA ACL/FIPA SL). With regard to the *observation* of other agent's actions, the particular way to achieve this activity must be defined by the corresponding type of social action. The same applies to the *abortion* of the execution of some action.

Note that the $\mathcal{RICA}$ metamodel, together with the execution-based semantics just sketched, can be considered as a programming language offering different types of constructs of organisational and communicative nature. In fact, the declarations of the different types of entities identified by the $\mathcal{RICA}$ metamodel can be conveyed by programming language constructs, as an alternative representation mechanism to the UML modelling notation. Just as Java programs are composed of declarations of classes, methods, attributes, etc., a $\mathcal{RICA}$ program comprises declarations of the different agents, roles, interactions, etc., required by the application. As an example, the following piece of code conveys the structure of the declaration of some role.

```
social role type SomeRole [specialises BaseRole]
activation{...}{...}
plays Role1, Role2, ...
performs{

      action type action1(in param1,..., out paramN,...)
      on execution{...}
      on abortion{...}

      ...
}
```

In the example, reserved words are capitalised and relate to the different types of entities and relationships of the $\mathcal{RICA}$ theory. The syntax and indentation features attempt to resemble those of the Java language. Any conventional programming language (e.g., Java) may complete this organisational-based agent programming language, by providing the specification of the activation conditions and the execution code of their actions.

3 $\mathcal{RICA}$–$\mathcal{J}$: a $\mathcal{RICA}$-based Software Framework

This section describes the $\mathcal{RICA}$–$\mathcal{J}$ software framework (an acronym for $\mathcal{RICA}$-Jade), which allows to program multiagent systems based on the $\mathcal{RICA}$ theory and its operational semantics that we have sketched in the previous section. We have chosen to develop the software framework on top of an existing tool, in order to reuse those middleware aspects required for supporting agent interactions, as well as the different agent

abstractions provided. In particular, we use *Jade* [2], a FIPA-compliant platform implemented in *Java*, so our framework is also based that language. The implementation of the $\mathcal{RICA}$ layer (implemented in a java package named `rica`) was designed with the goal of relieving, as much as possible, the application developers from those programming tasks that could be automated. These tasks include, for instance, the programming of the agent's and generic role's predefined behaviours, dealing with the instantiation, playing and abandonment of roles; the programming of the default behaviours of agents participating in some interaction under some protocol; the updates on the interaction state resulting from the execution of actions, etc.

In order to achieve this goal, the *types* of the application agents, roles and interactions, etc., must be available at runtime. Hence, the $\mathcal{RICA}$ layer has been subdivided into two modules: A *reflective* module, implemented by the package `rica.reflect` (an analogue of the `java.reflect` standard package), containing those classes which implement the metamodel entities of the $\mathcal{RICA}$ theory: `AgentType`, `RoleType`, etc.; and a *core* module, implemented by the package `rica.core` (an analogue of the `jade.core` package), providing classes implementing the structure and common behaviour of the different $\mathcal{RICA}$ abstractions: `Agent`, `Role`, `SocialInteraction`, `Protocol`, etc. The next subsections will describe the structure and design guidelines followed in the implementation of the reflective and core modules.

3.1 The *reflective* module

The `rica.reflect` package provides an implementation of the metamodel entities of the $\mathcal{RICA}$ theory, in such a way that the class diagram of figure 1, conveys the general structure of this package as well. `AgentType`, `RoleType`, etc., classes are analogues to the java class `Class`, which represent the type of some declared object. The `Object` class, in turn, is an analogue of the classes implemented in the `rica.core` package, which are objects of a more specific nature. The basic problems faced in the design of the reflective capabilities of the framework are related to these questions:

- Firstly, how can the declaration of the types of roles, interactions, and so on, be embedded in the declaration of java classes?
- Secondly, how can these declarations be accessed?

The devised solution to answer the second question, which sets up the premises to solve the first one, follows the pattern used by the Java language itself. In java, type information is accessible in two ways: statically, using the static `class` field, automatically declared for any class, and, dynamically (at run-time), by the `Object` instance method, `getClass()`. Similarly to the `class` field, any class specifying some $\mathcal{RICA}$ type will necessarily declare a static field, named `type`, containing the declaration of its type. Concerning the dynamic access to the type of some $\mathcal{RICA}$ object, `getType()` instance methods will be defined for each of the $\mathcal{RICA}$ top-level classes: `Agent`, `Role`, `SocialInteraction`, etc. This methods will access the corresponding `type` field declared for the dynamic class of the object. Figure 4 illustrates this design pattern using some role entity as example.

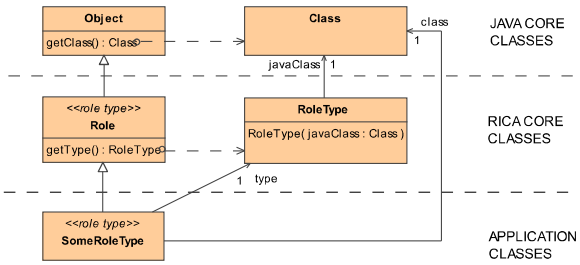

Fig. 4. Reflective design pattern

With regard to the first question, the implementation of an entity type will be commonly carried out by means of the implementation of a java class [3]. In those cases, the declaration of the type will be embedded in the class by means of `public static final` fields, which will be, in general, initialised by accessing the `type` fields of the declaration members[4]. Then, the declaration will be created automatically by accessing these static fields. As shown in figure 4, the classes implementing the $\mathcal{RICA}$ metamodel entities, must declare a constructor which takes as parameter the java class corresponding to the static class of the $\mathcal{RICA}$ entity.

All the java classes implementing some entity type will follow the same implementation schema corresponding to that type. For instance, the schema for embedding the declaration of a role type in a java class is as follows:

```
public [abstract] class SomeRole extends BaseRole|Role
{
    /** Static type. */
    public static final RoleType type = new RoleType(SomeRole.class);

    /** Played roles. */
    public static final RoleType role1 = SomeRole1.type;
    public static final RoleType role2 = SomeRole2.type;
    ...

    /** Performed actions. */
    public static final ActionType action1 = SomeAction1.type;
    public static final ActionType action2 = SomeAction2.type;
    ...
}
```

3.2 The *core* module

The `rica.core` package is designed around the collection of classes shown in figure 5. These classes implement the major types of $\mathcal{RICA}$ entities whose structure and dynamic behaviour was described in section 2.2. As can be observed, the abstract classes

[3] The exception are parameter types which are implemented as fields of the declaring classes of interactions.

[4] Thus, the static accessor will be used for declaration purposes, while the `getType()` instance methods will be used at run-time.

(interfaces) of the diagram represent the base action types. Single and disjunctive action hierarchies provide actual implementations of these interfaces (not shown in the figure). On the other hand, the only $\mathcal{RICA}$ entities which are left from the diagram of figure 5 are: *parameters*, which, as was previously explained, are embedded in java as fields of the corresponding interaction and action classes, and *languages*, whose implementation will be described bellow. Also, note that social interactions objects in the $\mathcal{RICA}$–$\mathcal{J}$ framework are not shared by the different participating agents, but are *owned* by a single participant's role. In this way, social interaction objects represent the view of the interaction from one participant's perspective, called its (characteristic) *performer*. Later, we will deal with the issue of participants representation.

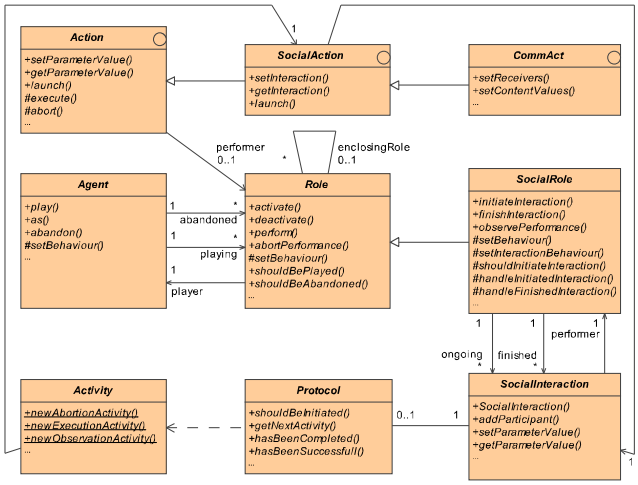

Fig. 5. Major structure of the `rica.core` package

Figure 5 also shows the main methods declared by these classes, which manifest a close correspondence with the computational semantics prescribed for each type of entity (methods parameter types are omitted from the class diagram to improve its readability). We can distinguish between two different kind of methods: those corresponding to the dynamic characteristic of the declared types, and those related to the state of its instances:

– *Type related methods.* As it was previously described, agent, role, social role, and action types, require certain characteristics concerning its functionality, which complement the static characteristics identified by the $\mathcal{RICA}$ metamodel. For instance, role types must specify its characteristic behaviour and activation conditions. These components of their declaration can be specified by overriding the `setBehaviour`, `shouldBeAbandoned` and `shouldBePlayed` methods. Similarly, the `setInteractionBehaviour` and `shouldInitiateInteraction` and methods allow particular social roles to specify their conditions to initiate its participation in some interaction, as well as its characteristic behaviour within it. Moreover, programmers

can implement the `handleInitiatedInteraction` and `handleFinishedInteraction` abstract methods, which are *hooks* invoked when the specified interaction is initiated or finished, respectively. Some of these type of methods have predefined implementations, as those previously mentioned for social roles, which resort to the protocols declared for its social interaction type, or those related to the behaviours of agents and pure enclosing roles. Other ones, such as those corresponding to the activation conditions of roles, or the execution functionality of actions are declared as `abstract`. Finally, some of them are declared as `final`, as those fixing the general behaviour of social roles, or the execution functionality of communicative actions.

– *Instance related methods.* These methods do not carry type information, but simply allow to change or observe the state of the different instances. In this way, the `play(RoleType r)` and `abandon(RoleType r)` methods, cause some agent to play and abandon the role instance of the specified type (these methods will be called from the predefined behaviour of the agent), and the `as(RoleType r):Role` method returns the role instance of the specified type currently being played by the receiving agent. Similarly, the `launch()` method starts the execution of some action, and the `initiate(SocialInteraction i)` method cause the specified interaction in which some social role is participating to be initiated. Last, the `getNextActivity()` method of the `Protocol` class, allow the social roles to get the activities that must be performed in each step of some interaction according to the state of the protocol.

The behaviours of agents and roles are represented by means of the Jade abstraction, `jade.core.Behaviour`. The predefined behaviour for agent types is implemented as a cyclic behaviour, which checks the activation conditions of the agent roles, through the `shouldBeActivated` method. Once this method returns true, the role's characteristic behaviour will be scheduled for execution. In the case of social roles, its predefined behaviour is a parallel behaviour which necessarily includes a cyclic behaviour which invokes the `shouldInitiateInteraction` method, to check if some interaction must be initiated. The default implementation of this method creates an instance of the social role's interaction type, and checks if any of the protocols that regulate its type of interaction (which will be accessed using the reflective capabilities, i.e. the `getType()` method) signals the beginning of the interaction. As soon as the `shouldBeInitiated` protocol's method returns true, the interaction will be initiated, and the social role's interaction behaviour (the one returns by the `getInteractionBehaviour` method), will be scheduled as part of the social role's parallel behaviour. Moreover, the `handleInitiatedInteraction` method would be invoked. The social role's predefined interaction behaviour performs the activities returned by the `getNextActivity` method of the protocol (execute, observe or interrupt some action), until the end of the interaction, which will be signalled by its `hasBeenCompleted` method. In this way, the whole dynamics of some agent concerning those aspects related to the management of interactions (its initiation, development and finalisation) are automated through the predefined behaviours of agents and social roles.

The `rica.core` package also includes other classes providing the *glue* linking the $\mathcal{RICA}$ abstractions with those provided by the Jade platform. Among these classes, the `ACLMessageTranslator` class helps to translate some communicative action to and from an object of the `jade.lang.acl.ACLMessage` class. To achieve this task, the language spoken by the agent performing the communicative action (represented by the Jade abstraction, `jade.content.lang.Codec`), and the ontology of the social interaction in which the action is performed (represented by an instance of the `jade.content.onto.Ontology` class), will be used. The `rica.core.RolePlayingDFService` bridge class provides convenient ways to published the `types` of social roles that some agent is currently playing, as well as its spoken languages, through the `jade.domain.DFService` class. In this way, when some agent is committed to initiate an interaction and must look for other participants, agents playing the types of roles declared for the interaction's type, and speaking some of the agent's languages, can be searched for in the directory service.

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

The design of a multiagent application in the $\mathcal{RICA}$–$\mathcal{J}$ framework departs from its organisational analysis in terms of the $\mathcal{RICA}$ metamodel, possibly elaborated on the basis of the method suggested in [6]. This analysis will uncover those communicative interactions that application-dependent agents and interactions rely on. In case that a need for the use of protocols is identified, the $\mathcal{RICA}$ model of the application is complemented with their proper specification, using for that purpose whatever protocol specification technique be most adequate.

The architecture of application programs in the $\mathcal{RICA}$–$\mathcal{J}$ framework is shown in figure 6. Applications rely on the reuse of two kinds of component libraries: those implementing the communicative aspects of the application-dependent interactions, and those allowing the implementation of protocol specifications on the basis of different kinds of formalisms. These libraries can be based upon FIPA specifications or other contributions made available by independent application developers. Currently, most of the FIPA ACL components are implemented in a library of communicative interactions under the `fipa` java root package: `fipa.closedActionPerforming`, `fipa.closedInfoExchange`, etc. Protocols are implemented based on so-called interaction machines [7], available through the `protocol.interactionMachine` package.

Given that the necessary protocol specification and instrumentation techniques are available, the implementation of a multiagent application based on $\mathcal{RICA}$–$\mathcal{J}$ requires to design and implement (1) the communicative interactions needed that are not present in the available communicative libraries, and (2) the application-dependent interactions and agents. In the following we will illustrate both steps on the basis of the e-commerce model of figure 2, and using interaction machines as protocol specification formalism.

On the one hand, the implementation of *communicative interaction* types and their corresponding communicative roles, actions, protocols, etc., components, requires embedding their $\mathcal{RICA}$ definitions in proper subclasses of the $\mathcal{RICA}$–$\mathcal{J}$ core superclasses, as explained in section 3.1. On the other hand, for fully implementing the different types of components, additional aspects concerning the behaviour of these entities

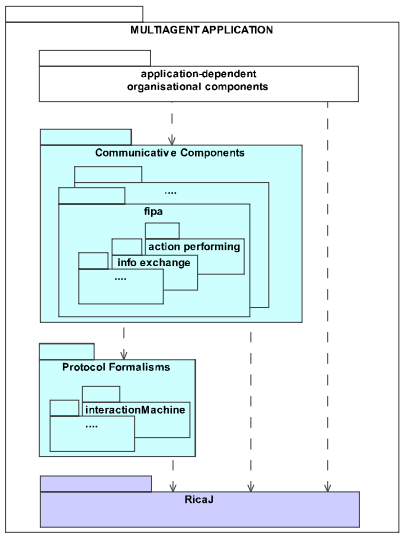

Fig. 6. Architecture of a $\mathcal{RICA}$–$\mathcal{J}$ application

must be addressed, following the general requirements described in section 3.2. This section describes the particular way in which communicative and social components will generally addressed these implementation features. For instance, the playing and abandonment conditions for communicative roles, which are implemented by overriding the corresponding `shouldBePlayed()` and `shouldBeAbandon()` `Role` methods, should be left unspecified, as these conditions closely depends on the specific type of social role specialising the communicative role. Similarly, the `handleInitiatedInteraction()` and `handleFinishedInteraction()`, should not be implemented. On the contrary, those aspects concerning the conditions that should be met in order to initiate and interaction of the kind declared for the communicative role, will be usually defined at this level.

For instance, an *action performing* interaction is initiated when the requester generates the intention that some action be performed, and some requestee partner has been found. Following this scheme, the implementation of the `ActionPerforming` class, which is partially shown in table 1, provides a constructor for the requester role which updates its *action* parameter, taking into account the intentional state of the specified role. If some action shall be performed, a partner agent playing the requestee role is located through the role-based directory service[5].

The action performing interaction is instantiated by the requester agent once the monitoring behaviour of its social role's behaviour is restarted, and the `shouldInitiateInteraction` method is subsequently invoked. Then, the fipa request protocol is instantiated and asked whether the requester must initiate its participation in the inter-

[5] The `setRestOfParticipant` method,declared by the `SocialRole` class, achieve this task, thanks to the reflective capabilities of the framework.

```
public class ActionPerforming extends Communication{

    /** Type info. */
    public static final ParameterType action = new ParameterType("action",SocialAction.class);
    public static final CommRoleType requester = Requester.type;
    public static final CommRoleType requestee = Requestee.type;
    public static final ProtocolType submissionProcedure = FIPARequestProtocol.type;
    ...

    /** Constructor for requesters. */
    public ActionPerforming(Requester requester){
        super(requester);
        setAction(requester.getIntendedAction());
        if (getAction()!=null){ setRestOfParticipants(); ... }
    }

    ...
}
```

```
public class FIPARequestProtocol extends InteractionMachineProtocol{

    /** Type info. */
    public static final CommInteractionType interaction = ActionPerforming.type;

    /** Interaction Machine */
    public static class FIPARequestMachine extends InteractionMachine{

        /** States. */
        public static final InitialState initial = new InitialState("initial");
        public static final State evaluating = new SimpleState("evaluating",Evaluate.type);
        public static final State performing = new SimpleState("performing", action);
        ...

        /** Transitions. */
        public static final Transition initial_evaluating_request =
            new Transition("initial_evaluating_request",initial,evaluating,Request.type);
        public static final Transition evaluating_peforming_refuse =
            new Transition("evaluating_performing_refuse",evaluating,done_unsuccessfull,Refuse.type,
                           new Not(new Successfull(Evaluate.type)));
        ...
    }
}
```

```
public abstract class Requester extends Communicator{

    /** Type info. */
    public static final CommActType request = Request.type;
    public static final CommActType cancel = Cancel.type;
    ...

    /** Intention management. */
    private SocialAction intendedAction;
    public void setIntendedAction(SocialAction action){ ...; getRoleBehaviour().restart();}
    public SocialAction getIntendedAction(){ ...; return action};
    public void dropIntentedAction(SocialAction action){...; getRoleBehaviour().restart()};
}
```

```
public abstract class Requestee extends Communicator{

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

    /** Called from Evaluate */
    public abstract boolean willPerform(SocialAction action);
    public boolean willInmediately(SocialAction action){ return true;}
    ...

}
```

```
public class Seller extends Requestee{

    /** Type info. */
    public static final ActionType sell = Sell.type;
    ...

    /** Overriden Requestee interface. */
    public boolean willPerform(SocialAction action){
        return (action instanceof Sell) && isValidCreditCard(((Sell)action).getCreditCard());
    }

    /** Overriden role object interface. */
    public boolean shouldBePlayed(){...}
    public boolean shouldBeAbandoned(){...}

}
```

Table 1. Implementation of the action performing interaction type

action. This protocol, implemented by means of an interaction state machine, is initiated if the initial *request* transition can be activated, which in turn, requires the action parameter (its propositional content) to be initialised. Hence, the interaction will be initiated if this parameter was set in its instantiation.

The implementation of the `Requester` role provides the `getIntendedAction` method, as well as the `setIntendedAction` and `dropIntention`, to update its intentional state. These methods may only be invoked from the user interface of the agent, or from other roles which the agent is currently playing. Moreover, they will cause the role's behaviour to be restarted, in order to initiate a new interaction, or finish an ongoing one through the cancel communicative action.

Besides communicative actions, just as the `Request` and `Cancel` actions declared by the requester agent, non-communicative social actions can be defined for communicative role types as well, as for the requestee role. However, the full specification of these types of actions will normally be available at the level of application-dependent roles. Thus, the particular implementation of the *evaluate* action performed by this type of roles, will depend on the kind of social role reusing its declaration. The evaluate action can be implemented by the `Evaluate` class, a subclass the `SingleSocialAction` core class. Its `execute` method could be left unspecified, but an alternative implementation schema, resorting to abstract methods of the `Requestee` class is more convenient. This class provides implementations for identifying as to whether the action must be performed, and if it will be performed immediately:

Finally, *application-dependent* components typically consist of *social* roles and interactions. As the pragmatic competence of social roles is implemented by the communicative roles that they specialise, the only thing left to be determined is the moment in which this competence will be performed, i.e. its activation conditions, and the possible actions performed in the beginning and ending of the interactions. For instance, the buyer role, as a kind of requester, can be implemented, as shown below, by extending the `Requester` class described above. Its activation is set by the interval of time during which its human counterpart is connected to the e-commerce portal or some purchase is pending to be completed. The activities performed in the initiation and finishing of some purchase will most probably be related to updates in its user interface.

5 Conclusions

In this paper we have shown how multiagent applications can be developed based on the reuse of communicative components, structured along organisational concepts, which encapsulate the ACL of the MAS. In particular, we have presented the $\mathcal{RICA}$–$\mathcal{J}$ software framework that proves that communicative roles and interactions cannot only be used as *conceptual* tools for the analysis and design of ACLs (see [6] for an analysis of FIPA ACL in these terms), but also as *software* components providing much of the functionality that application-dependent social roles and interactions require.

The $\mathcal{RICA}$–$\mathcal{J}$ framework allows to build up highly-reusable libraries of communicative roles, interactions and protocols, and thus relieves MAS programmers of most tasks associated with the management of interactions, a key issue in the implementation of any MAS. Moreover, it is based on a separation of concerns, as the pragmatic com-

petence of agents is encapsulated in special software modules, and is thus kept away from non-communicative aspects of the application.

The framework also stresses the use of interactions as first-class programming abstractions, something which is absent from JADE and other MAS frameworks. A notable exception to this rule are approaches based on electronic institutions [13]. Still, the $\mathcal{RICA}$–$\mathcal{J}$ framework relies on a programming language metaphor and a specification of the execution-based semantics of the different types of organisational meta-entities. This bias towards organisational modelling also differentiates our approach from other proposals in the field of agent programming languages, which tend to focus on intra-agent programming matters, rather than on the social structure of the application.

Future work comprises the (re-)implementation of several MAS applications so as to fully explore the potential and limitationsuse of the $\mathcal{RICA}$–$\mathcal{J}$ framework. Furthermore, we are interested in relating our work to current FIPA efforts in the field.

References

1. FIPA: The foundation for intelligent physical agents. (http://www.fipa.org/)
2. Bellifemine, F., Poggi, A., Rimassa, G.: Jade: a fipa2000 compliant agent development environment. In: Proceedings of the fifth international conference on Autonomous agents, ACM Press (2001) 216–217
3. Nwana, H.S., Ndumu, D.T., Lee, L.C., Collis, J.C.: Zeus: a toolkit and approach for building distributed multi-agent systems. In: Proceedings of the third annual conference on Autonomous Agents, ACM Press (1999) 360–361
4. Omicini, A., Ossowski, S., Ricci, A.: Coordination infrastructures in the engineering of multiagent systems. In Bergenti, F., Gleizes, M., Zambonelli, F., eds.: Methodologies and software engineering for agent systems – An AgentLink Perspective. Kluwer (2004)
5. Ossowski, S., Omicini, A.: Coordination knowledge engineering. Knowledge Engineering Review **4** (2003) 309–316
6. Serrano, J., Ossowski, S., Fernández, A.: The pragmatics of software agents – analysis and design of agent communication languages. In et al., K., ed.: Intelligent Information Agents – The AgentLink Perspective. Springer-Verlag (2003)
7. Serrano, J., Ossowski, S.: An organisational approach to the design of interaction protocols. In Huget, M.P., ed.: Communication in Multiagent Systems. Volume 2650 of Lecture Notes in Computer Science. Springer (2003) 194–208
8. Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer **31** (1998) 40–47
9. Cohen, P.R., Levesque, H.J.: Communicative actions for artificial agents. In Lesser, V., ed.: ICMAS'95, MIT Press (1995) 65–72
10. Ferber, J., Gutknetch, O.: A meta-model for the analysis of organizations in multi-agent systems. In Demazeau, Y., ed.: ICMAS'98, IEEE Press (1998) 128–135
11. 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
12. Sadek, M.D.: Dialogue acts are rational plans. In: Proceedings of the ESCA/ETRW Workshop on the structure of multimodal dialogue, Maratea, Italie (1991) 1–29
13. 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