RPI.Social: Simple Enactment and Execution of First-Class Agent Interaction Protocols

Atef Nouri and Wided Lejouad Chaari
Laboratoire COSMOS, ENSI, University of La Manouba, 2010 La Manouba, Tunisia

Keywords: First-Class Agent Interaction Protocols, Multiagent Systems, Executable Interaction Protocols, Role Enactment, Agent-Oriented Software Engineering.

Abstract: In Multiagent systems, first-class interaction protocols are those whose implementations are decoupled from the agents. A previous work has introduced the RPI framework (Role Playing Interactions) and established the contribution of RPI.Idiom which is a high-level language for the definition of such protocols. In this paper, we present RPI.Social which is a social engine associated with the agent to help it use interaction protocols written in RPI.Idiom. RPI.Social has two components: RPI.Social.IoP (for Initiate or Play) and RPI.Social.Exec. The first component deals with discovering and initiating interaction protocols for an agent which has a goal that could only be pursued through interacting. The same component is also used by agents invited for interaction to decide whether to participate or to decline the invitation. The second component serves as an interpreter of interaction protocols with several mechanisms and rules to coordinate and share results amongst the interacting agents. The main contribution of this paper is a solution for agents to automatically identify and execute first-class interaction protocols.

1 INTRODUCTION

Multiagent systems (MAS) have two defining core concepts: agents and interactions. Despite the equal importance they both share in the definition of MAS, the focus has always been put on agents. Indeed, research on interactions has previously dealt with communication languages, documentation (e.g. negotiation protocols) (Miller and McBurney, 2007) and the game theoretical aspect of interaction protocols (Shoham and Leyton-Brown, 2008), etc. In contrast, agents have been studied as single autonomous objects (mobility, BDI, etc.) and as interacting entities (institutions (Morales et al., 2017), organizations (Jensen, 2015), roles, etc.). As a result, over the years, agents have become more capable at performing in their environments while interaction protocols grew more diverse. Nevertheless, as part of the effort to efficiently document interaction protocols, there have been several works that have dealt with the abstraction of interactions. These efforts started with early MAS methodologies and design formalisms, notably AUML (Bauer et al., 2000). The aim of these works was to decouple interactions and agents at design time. Some works in the last decade took the decoupling to the implementation phase so that the interaction's code is no longer intertwined with the interacting agents'. Such interactions are interchangeably called reified (Khalfaoui and Chaari, 2013) and first-class (Miller and McBurney, 2007).

As easy as specifying an interaction protocol amongst abstract roles using some notation could be, the actual interpretation of such specification hides some tedious tasks that have to be performed by the interacting agents. In fact, in the case of second-class interactions, agents know by design the course of action that has to be taken and the execution is automatically based on little to no deliberation on the actual content of the interaction i.e. the agent trusts that its programmer:

- has taken care of checking the preconditions and the postconditions of the interaction; and
- made sure that the goal for which the interaction is executed in the first place would be possibly fulfilled by the end of the exchange.

On the other hand, agents interacting using first-class interaction protocols need to have the capabilities to:

- match their goals for interacting with the end result of the interaction protocol;
- initiate an interaction protocol or take a role in one;

Nouri, A. and Chaari, W.
RPI.Social: Simple Enactment and Execution of First-Class Agent Interaction Protocols.
DOI: 10.5220/0006599400910102
In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART 2018) - Volume 1*, pages 91-102
ISBN: 978-989-758-275-2

- enact roles in protocols;
- execute the operations assigned to their roles.

In order to provide the aforementioned capabilities to agents, we propose RPI.Social, a social engine associated with the agent to endow it with the ability to take roles in the execution of interaction protocols. A previous work (Nouri et al., 2015) has introduced RPI.Idiom which is a high-level language for implementing first-class interaction protocols within a framework baptized Role Playing Interactions (or RPI). Along with the descriptive language, RPI provides through RPI.Social mechanisms for the agents to be able to reason on interaction protocols and execute them.

The purpose of this paper is to introduce RPI.Social and give an overview on its inner workings. The present paper does not attempt to demonstrate the properties of RPI.Idiom interaction protocols that are enumerated in Section 2.2 since the language was fairly presented in (Nouri et al., 2015). Likewise, RPI.Inference is not covered by this paper.

The rest of this paper is organized as follows. Section 2 is a brief introduction of RPI framework. RPI.Social.IoP, the unit used for enacting interactions and roles is detailed in Section 3. RPI.Social.Exec is presented in Section 4. Section 5 gives an overview on RPI.Inference. Section 6 is an exposition of some related works with comparisons that aim to position RPI.Social in the literature and highlight its contribution. Section 7 is the conclusion.

2 RPI FRAMEWORK

This Section gives an overview of the features and aims of the RPI framework and its components.

2.1 RPI Framework Overview

The Role Playing Interactions framework proposes an approach to separate interaction protocols from the agents. The goal of RPI is to provide a common environment for implementing and executing interaction protocols that have these properties:

- To be role-based i.e. to be defined between a set of abstracting roles instead of the actual interacting agents.
- To be automatically enactable and executable i.e. agents should be self-sufficient in enacting roles in the interaction protocol and in executing their parts in it.
- To have a generic structure i.e. the unit of construction of the interaction is a behaviour which covers the actions of transmission and reception of messages, as well as any action or service that the underlying agent could perform or provide.
- To have a meaning and to be socially generic i.e. each unit of construction has its own explicit meaning. The meanings attributed to the construction units are independent from the social prescriptions (e.g. commitments) which defines the property of social genericity.
- To be modular in order to support the reuse and the potential composition of protocols.

Similarly, an agent within the RPI environment has a set of properties:

- Its implementation excludes interactions and explicit references to external implementations of interaction protocols.
- It has a set of state variables and another one for social behaviours that both are exposed to other agents to the roles played in interactions.
- At a given moment during runtime, it may have a goal which is expressed in the same language as the constraints in RPI.Idiom (see 2.2). To fulfill its goal, the agent either takes individual actions or it starts interaction.
- It has a social engine that manages the whole process of interacting with other agents. The social engine which is specified by RPI is a defining part of an agent within the framework.

RPI is divided into several component subsystems that achieve a part of the functionalities provided by the framework (Fig. 1).

The central sub-system of the framework is RPI.Idiom which is the language for defining interaction protocols. Section 2.2 gives a quick overview of the language and its main features.

RPI.Idiom interaction protocols are managed by RPI.Library. This sub-system provides the developer with a tool for managing the repository of protocols. It also plays the role of interaction protocols broker to the agents within the framework.

The RPI.Social sub-system is the social engine associated with an agent. This sub-system has two component units:

- RPI.Social.IoP for initiating interactions and/or playing roles in them; and
- RPI.Social.Exec for performing behaviours associated with the agent's role within an interaction protocol.

The features of RPI.Social are detailed in Section 3 and 4.

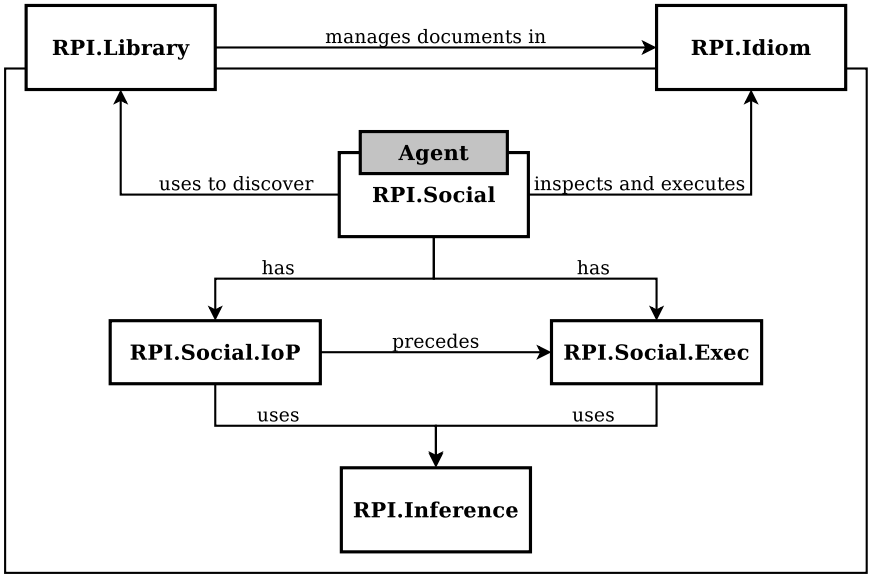

Figure 1: Interdependence within RPI.

RPI.Inference is the sub-system responsible for providing the tools to RPI.Social to reason on interaction protocols to decide whether to pick the protocol to achieve a given goal, or if a behaviour is well-executed during an interaction. An overview of the tools RPI.Inference provides is given in Section 5.

2.2 RPI.Idiom Overview

RPI.Idiom is a high-level language for implementing first-class interaction protocols. Protocols specified in RPI.Idiom verify the properties of the RPI framework mentioned in the previous Section (2.1). An RPI.Idiom interaction protocol has two parts:

- a data template in which the data and the roles relevant to the interaction are declared; and
- a process template that defines the actual interaction in terms of behaviours.

As Fig. 3 shows, the process template which is the part that specifies the interaction protocol is defined by a premises system and a move. The former is basically a declaration of the predicates and the axioms using them that would be used in the latter. The move is the actual process that specifies how the interaction would be executed. A move has two constraints: one to denote its precondition and the second for its postcondition. These constraints are expressed using the predicates declared in the premises system. A constraint is a ground formula that could be atomic (i.e. a literal) or composite (i.e. a conjunction or a disjunction of ground formulas). The move could be atomic or composite, where the latter is defined as a composition of moves.

There are four composition operators to combine moves:

- **and-seq**(M): a sequence of sub-moves M;
- **and-con**(M): a set of concurrent sub-moves M;
- **xor**(M): a choice of exactly one sub-move from M; and
- **or**(M): a subset of concurrent sub-moves from M, or -alternatively- a subset choice.

The constraints of the combined moves are also composed using the following rules:

- for **xor** and **or** operators, the constraints of the move are the disjunction of the respective constraints of the underlying sub-moves;
- for **and-con** operator, the constraints of the composite move are the conjunction of the respective constraints of the underlying sub-moves; and
- for **and-seq** operator, the constraints of the composite move are the conjunction of the respective constraints of the sub-moves after eliminating the inconsistent constraints, i.e. only the initial preconditions and the final postconditions are kept.

An atomic move, on the other hand, is defined by an instance of a behaviour belonging to a role, the sets of inputs and outcomes, and the usual constraints for the move. The precondition of the atomic move acts as a filter on the inputs while its postcondition specifies the effect of performing the behaviour on the outcomes.

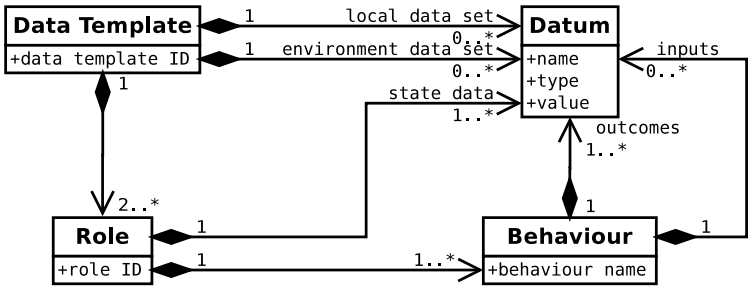

Figure 2: Structure of the data template in RPI.Idiom.

A role is defined by a data set that tracks its state during the interaction and a set of behaviours that the role player should be able to perform. The declaration of the set of roles that would be placeholders for the interacting agents within an RPI.Idiom protocol is done in the data template. As shown in Fig. 2, this part of an RPI.Idiom interaction protocol specification also contains the declaration of two data sets. The first is the environment data set which contains the bare necessary knowledge about the environment for the interaction protocol while the second is the local data set which contains temporary knowledge for the span of the interaction. The elements of three data sets (roles' states, environment's, local's) are used as inputs and outcomes for the behaviour instances that represent the atomic moves in the protocol.

3 RPI.Social.IoP: ENACTMENT OF INTERACTION PROTOCOLS AND ROLES

In the previous Section, we gave a glimpse of RPI.Idiom, the language to use for the specification of interaction protocols in RPI. RPI.Idiom is considered to be the static dimension of interactions in the framework. The dynamic dimension is ensured by both units of RPI.Social. This Section puts the focus on RPI.Social.IoP (Initiate or Play) that is the first of these two units. RPI.Social.IoP is basically an interaction that transcends RPI which purpose is to bootstrap RPI interaction protocols.

RPI.Social.IoP has two complementary protocols: an active one for initiating interactions and a passive one for receiving and responding to invitations to play roles in interactions. In each of the protocols, the social engine of the agent is in one of a given set of states at all times. Transitions between states are defined by events such as receiving a notification or data, or reaching a milestone in the enactment of the interaction protocol. A transition usually triggers an algorithm executed by the social engine of the agent to advance the enactment of the interaction protocol.

The intuition behind RPI.Social.IoP is the need for establishing an ad-hoc agent organization for the interaction protocol since an RPI interaction requires permanent agents for its span. This way, the members of the organization should play the roles in the interaction. This is similar to the TCP connection establishment through the three-way handshake process. Due to that similarity, we represent the state-transition aspect of RPI.Social.IoP as a finite state machine (FSM). Fig. 4 depicts the state machine for an initiator agent. Likewise, Fig. 5 is the state machine for a non-initiator agent. A second aspect of the unit is algorithmic and it defines the behaviour of the agent in a given state (e.g. in the *Idle* state, the agent awaits for a triggering event to happen as seen in the state machines).

3.1 Social Engine States in RPI.Social.IoP

The social engine of an agent could be in one of eight states. One of those states is the initial state called *Invalid* (denoted by the solid filled circle in Fig. 4 and Fig. 5). It is a temporary state in which the agent is being initialized and is consequently not ready for interacting. As soon as the initialization is properly done, the social engine of the agent transits automatically and irreversibly from *Invalid* to *Idle*. The other six states are *Fetching*, *Waiting Response*, *Waiting Approval*, *Starting Interaction*, *Idle Interaction* and *Busy Interaction*.

In the *Idle* state the social engine is neither attempting to enact a role nor executing an interaction protocol. In the case when the agent comes across a goal that could not be attained but through interacting, the social engine turns to the *Fetching* state and starts searching for an interaction protocol that helps it fulfill that goal. If no sufficient interaction protocol is found, the social engine rolls back to the *Idle* state. However, if a candidate interaction protocol is found,

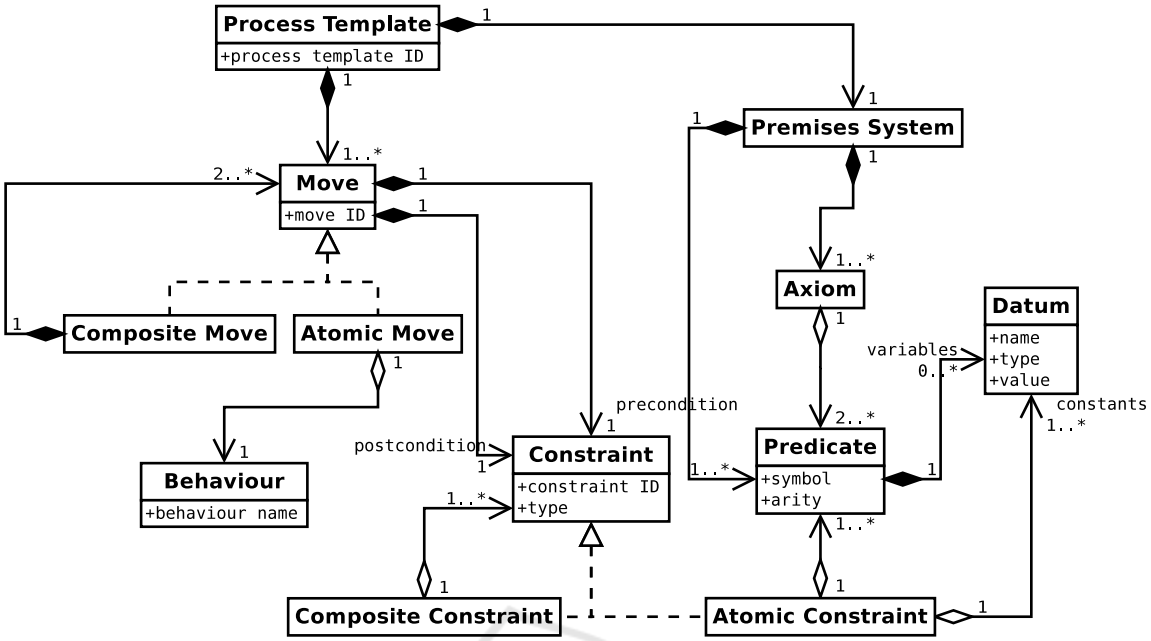

Figure 3: Structure of the process template in RPI.Idiom.

an invitation to play a role in that particular protocol is sent to a set of agents that could be selected based on one or more criteria like proximity, etc. The invitation takes the form of a PROPOSE message embedding the identifier of the interaction protocol. The message is broadcast to the chosen agents. Upon sending the invitation, the social engine of the initiator agent switches to the *Waiting Response*. In that state, the social engine is passively waiting for potential candidates to enact roles in the proposed interaction protocol. The candidatures are ranked by default from the first to be received which holds the highest priority, to the last positive reply to the invitation which has the least priority. If the initiator fails to establish an ad-hoc agent organization for the interaction, it rejects all the candidatures and reverts to the *Fetching* state.

From an invited agent's perspective, its social engine does not accept invitations unless it is in the *Idle* state. If the social agent does not accept to play a role, it sends a REJECT_PROPOSAL to the initiator. Otherwise, it sends an ACCEPT_PROPOSAL message to the initiator agent and embed its candidature in the content of the message, then it switches to the *Waiting Approval* state in which it awaits for two events: the acceptation of the candidature coming from the initiator and the complete foundation of an ad-hoc agent organization for the purpose of instantiating the interaction protocol. In the case of a rejection from the initiator due to failure to form the ad-hoc organization or any other reason, the invited agent turns back to the *Idle* state.

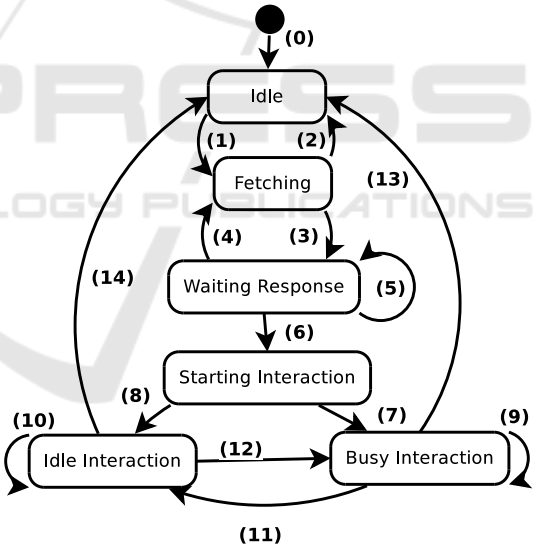

Figure 4: FSM for initiating interactions in RPI.

When all the role players are gathered together to interact, the interaction starts and the social agent of every agent taking part in it goes immediately to the *Starting Interaction* state.

The work of RPI.Social.IoP ends when an agent goes to either the *Idle Interaction* or the *Busy Interaction* states. Both of the latter states and the transitions related to them (transitions (7) to (12) shown in Fig. 4 and Fig. 5) are part of RPI.Social.Exec as explained in Section 4.

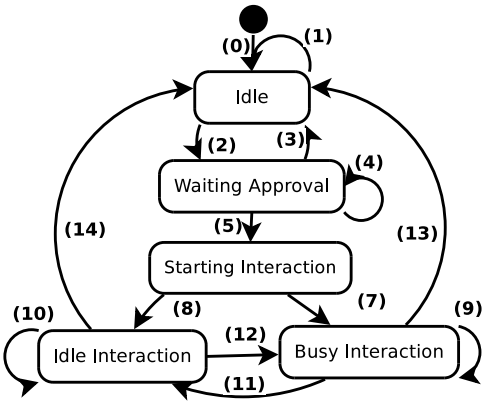

Figure 5: FSM for responding to invitations to interacting in RPI.

The tables 1 and 2 complement respectively figures 4 and 5 with the exhaustive state transitions in the realm of RPI.Social.IoP as well as their triggering events and the actions that are executed right before the transition.

3.2 RPI.Social.IoP Mechanics

While being in a given state or in the process of transiting from one state to another, the social engine may execute algorithms that attempt to match data between the agent and the role, and to evaluate preconditions and postconditions. These algorithms define the inner workings of RPI.Social.IoP.

3.2.1 Initiation of an Interaction

As soon as the social engine enters the *Fetching* state, it starts executing the algorithm for fetching protocols. Before starting the algorithm, the social engine queries the repository of interaction protocols for an enumeration of all its available items and sets. The search for an adequate protocol could be done on multiple rounds until either an ad-hoc organization is formed and the interaction is started, or the protocols enumeration is entirely exhausted and no match is found.

On each fetching round, the social engine picks the first protocol that satisfies a following set of the agent's requirements:

- The agent is able to play a role in the interaction.
- The agent is able to acquire and lock environment resources needed for the interaction (i.e. environment data set as defined in the RPI.Idiom specification of the protocol).
- The precondition of the interaction is contingent (i.e. its truth value is *true* or *unknown*) for the agent's current state (i.e. the state of the role that the agent relates to should allow it to be potentially able to start the interaction).
- The postcondition of the interaction protocol should potentially satisfy the agent's goal (i.e. it should not be in contradiction with the goal).

The social engine acquires the environment data prescribed in the RPI.Idiom interaction protocol using the same method of matching for role data sets as described later in the **identification to a role in an interaction protocol** process. Each of these resources is locked to be exclusively used by the role players during the interaction.

The whole purpose of initiating an interaction for an agent is to achieve a goal which is unreachable otherwise. For that reason, verifying the alignment of the interaction protocol's postcondition with the agent's goal is part of the selection of the protocol. The first part of the verification is the unification between the expressions of both the goal and the postcondition. The second and last part is to establish whether the postcondition entails the goal. The entailment is verified using RPI.Inference seen later in Section 5.

3.2.2 Responding to an Invitation to Interact

Upon the reception of a PROPOSE message to play a role in an interaction while in the *Idle* state, a positive response to the invitation depends on two requirements on the invited agent's side:

- Like the initiator, the agent should be able to play a role in the interaction.
- The precondition of the interaction is contingent for the agent's current state in the same manner as the initiator.

If the two requirements are met, the agent responds with an ACCEPT_PROPOSAL message otherwise it sends back a REJECT_PROPOSAL.

3.2.3 Identification to a Role in an Interaction Protocol

In order to identify with a role in an interaction protocol, the agent runs the algorithm for picking a role to play in the interaction protocol. Along with the interaction protocol as an input to the algorithm, there is a flag that indicates if the agent is the initiator of the interaction. For each role defined in the protocol, the agent attempts to generate a *profile* for that particular role. If the *profile* is generated, the agent uses it to prepare a *candidature* object that will be sent to the other potential role players.

Table 1: State transitions for an initiator agent.

Transition	Trigger	Action
Invalid → *Idle* (0)	The social engine is properly initialized	None
Idle → *Fetching* (1)	The agent decides that interacting is needed to attain a goal	None
Fetching → *Idle* (2)	No sufficient interaction protocol is found	None
Fetching → *Waiting Response* (3)	The agent finds a candidate interaction protocol	Send PROPOSE [broadcast/multicast] {interaction protocol identifier}
Waiting Response → *Fetching* (4)	Waiting for candidatures timeout is up	Send CANCEL [broadcast/multicast]
Waiting Response (5)	Receive ACCEPT_PROPOSAL [candidate] {candidature of the sender}	Send FAILURE [candidate] if the candidature is unintelligible
Waiting Response (5)	Receive REJECT_PROPOSAL [invited agent]	None
Waiting Response (5)	Receive ACCEPT_PROPOSAL [candidate] {candidature of the sender} and all the roles could be filled	Form an ad-hoc organization of role players and send CONFIRM to everyone of them
Waiting Response (5)	Receive FAILURE [role player] after the CONFIRM messages are sent to the role players	Re-send CONFIRM [role player] {role players list}
Waiting Response (5)	Receive INFORM [role player] {sender's candidature}	Send INFORM [role player] {own candidature}
Waiting Response → *Starting Interaction* (6)	Receive CONFIRM [role player] from every role player	Go to RPI.Social.Exec
Waiting Response → *Fetching* (4)	Receive CANCEL [role player]	Send CANCEL [broadcast/multicast]
Waiting Response → *Fetching* (4)	Receive ACCEPT_PROPOSAL [candidate] {candidature of the sender} and all the roles could be filled but the agent organization could not be formed	Send CANCEL to every role player
Waiting Response → *Fetching* (4)	An unrecoverable failure	Send CANCEL [broadcast/multicast]

The generation of a role *profile* is simply made by matching the state data set of the role and its behaviours with the internal state and the capabilities of the agent. The matching of data is done on both types and labels. Matching labels could be either word by word or semantic in accordance to an external configuration set by the designer of the multiagent application. Similarly, the matching of the behaviours is performed on the label of the behaviour as well as on its input and output data.

3.2.4 Evaluation of the Precondition

At the fetching stage of initiating an interaction in RPI, the precondition of the protocol is evaluated. Its truth value could be *true*, *false* or *unknown*. At this stage, the agent is interested in the possibility of running the interaction protocol; accordingly, it checks the precondition for contingency i.e. the social engine plugs in the precondition the values of the environment data set and the state data set items that belong to the role the agent identifies to, then checks if the truth value of the expression is not *false*. Both *true* and *unknown* truth values are acceptable to pick the interaction protocol since the values attributed to the state data sets of the other roles are unknown at the time of the evaluation.

3.2.5 Establishing an Ad-Hoc Organization for the Interaction

At some point of the interaction instantiation, the social engine of the initiator agent gathers a set of candidatures for every role defined in the RPI.Idiom protocol. Only one candidature is picked for a given role to be part of an ad-hoc organization. The algorithm that

Table 2: State transitions for an invited agent.

Transition	Trigger	Action
Invalid → *Idle* (0)	The social engine is properly initialized	None
Idle → *Waiting Approval* (2)	Receive PROPOSE [initiator] {interaction protocol identifier} and a role could be played in the interaction	Send ACCEPT_PROPOSAL [initiator] {own candidature}
Waiting Approval → *Idle* (3)	Waiting for CONFIRM from the initiator timeout is up	Send CANCEL [initiator]
Waiting Approval (4)	Receive CONFIRM [initiator] {role players list}	Send FAILURE [initiator] if the list is unintelligible otherwise send INFORM [role players] {own candidature} and become a confirmed role player
Waiting Approval (4)	Receive INFORM [role player] {sender's candidature}	Send FAILURE [role player] if the candidature is unintelligible
Waiting Approval (4)	Receive FAILURE [role player]	Send INFORM [role player] {own candidature}
Waiting Approval → *Starting Interaction* (5)	Receive CONFIRM [initiator] role players list and all the role players are known to the agent	Go to RPI.Social.Exec
Waiting Approval → *Idle* (3)	Receive CONFIRM [initiator] role players list and all the role players are known to the agent but the interaction could not be started	Send CANCEL [initiator]
Waiting Approval → *Starting Interaction* (5)	Receive INFORM [role player] {sender's candidature}, the agent is a confirmed role player and all the other role players are known to it	Go to RPI.Social.Exec
Waiting Approval → *Idle* (3)	Receive INFORM [role player] {sender's candidature}, the agent is a confirmed role player and all the other role players are known to it but the interaction could not be started	Send CANCEL [initiator]
Waiting Approval → *Idle* (3)	Receive CANCEL [initiator]	None
Waiting Approval → *Idle* (3)	An unrecoverable failure	Send CANCEL [initiator]
Idle → *Idle* (1)	Receive PROPOSE [initiator] {interaction protocol identifier} and no role could be played in the interaction	Send REJECT_PROPOSAL [initiator]

achieves this attempts to find the first organization of role players for which the precondition of interaction protocol is *true*. In contrast to the first evaluation of the precondition performed by the initiator, this one does not accept *unknown* as a valid truth value since all the role players are present.

4 RPI.Social.Exec: EXECUTION OF INTERACTION PROTOCOLS

Once the interaction is launched, RPI.Social.Exec comes into action. This framework sub-system is responsible of the execution of the move specified in the process template of the interaction protocol. Depending on the type of the move, its execution is treated as follows:

- If the move is atomic, the role to which it is associated executes it and passes the results to the rest of the role players.

- If the move is a composition (**and-con**), its component moves are executed recursively and concurrently to the best of the agent's ability.
- If the move is a composition (**or**), its selected component moves (i.e. those which are possible) are executed recursively and concurrently to the best of the agent's ability.
- If the move is a composition (**xor**), the unique selected component move is executed recursively.
- If the move is a composition (**and-seq**), the component moves are executed recursively one after the other.

Upon the formation of the ad-hoc organization for the interaction, the social engine of every role player enters *Starting Interaction* transitional state. In that state, the social engine pushes in the task queue of its underlying agent a *ProcessLauncherTask* task which navigates the interaction's main process's tree to reach the first atomic move. If the reached move is defined by one of the role player's behaviour, the latter pushes in its task queue an item to execute the behaviour mapped to the move and immediately switches to the *Busy Interaction* state (transition (7) in Fig. 4 and Fig. 5), otherwise it switches to the *Idle Interaction* state (transition (8) in Fig. 4 and Fig. 5). From that point on, the social engine alternates between *Busy Interaction* and *Idle Interaction* states which are essentially the RPI.Social.Exec states (transitions (9) to (12) in Fig. 4 and Fig. 5).

In the *Busy Interaction* state, when the agent runs the move's task, first, it evaluates the precondition of the move; then, it executes it; after that it evaluates the postcondition to detect if the execution was successful; and eventually, it pushes in its queue a task performs two operations:

- It sends to the rest of the organization an INFORM message that holds the state of the run (i.e. if it was done and whether it was successful or not) and the values of the output.
- It emulates the execution of the move on its own internal representation of the interaction.

In the *Idle Interaction* state, the social engine expects INFORM messages from busy role players. When one of those message is received, if the move is done, the execution is emulated by plugging in the output values and the state of the run is marked too.

At the end of each of the RPI.Social.Exec states, the social engine executes *ProcessLauncherTask* task again unless the root process that defines the whole interaction protocol is marked as done. Fig. 6 depicts the routine of task processing by RPI.Social.Exec.

5 RPI.Inference OVERVIEW

In RPI, agent goals, interaction protocols constraints and axioms are represented using predicates. Goals and constraints are logic expressions that contain only atomic formulas or their negations applied only to constant symbols, i.e. ground expressions. An axiom is a rule with a head and a potential body, similar to a Horn clause (Van Emden and Kowalski, 1976). However, axioms in RPI differ from Horn clauses with these properties:

- the head of the rule could be a negative, on the contrary, the head of the rule form of a Horn clause is always the positive unless it is a goal clause; and
- unlike a Horn clause, an axiom in RPI may contain more than one positive atomic formula.

Based on these two properties, in contrast to Horn clauses, there is more than one way to express an axiom as a rule. In fact, each of the literals in the axiom could be placed as head to the rule.

5.1 Main Rule and Interpretations of an Axiom

The main rule of an axiom is the one that explicitly defines it in RPI.Idiom. An interpretation of an axiom, on the other hand, is any rule derived from its main one.

For example, if the main rule is:

$$p(X,Y) \text{ :- } \neg r(Y), s(X,Y).$$

where p is the head of the rule; then, the interpretations will be the following:

$$r(X) \text{ :- } \neg p(X,Y), s(X,Y).$$

$$\neg s(X,Y) \text{ :- } \neg r(Y), \neg p(X,Y).$$

In RPI, there are two distinct operations that are performed on logic expressions:

- evaluation of ground expressions; and
- proving entailment relations among ground expressions.

The unit of RPI responsible for providing these two operations is RPI.Inference.

In some cases, the information needed for the evaluation of a precondition is incomplete. There are two approaches to deal with this case: default to either truth values *false* or *unknown*. In the former approach the world is assumed to be closed while in the latter the world is assumed to be open. In a closed world, e.g. in a Prolog program, anything that can't

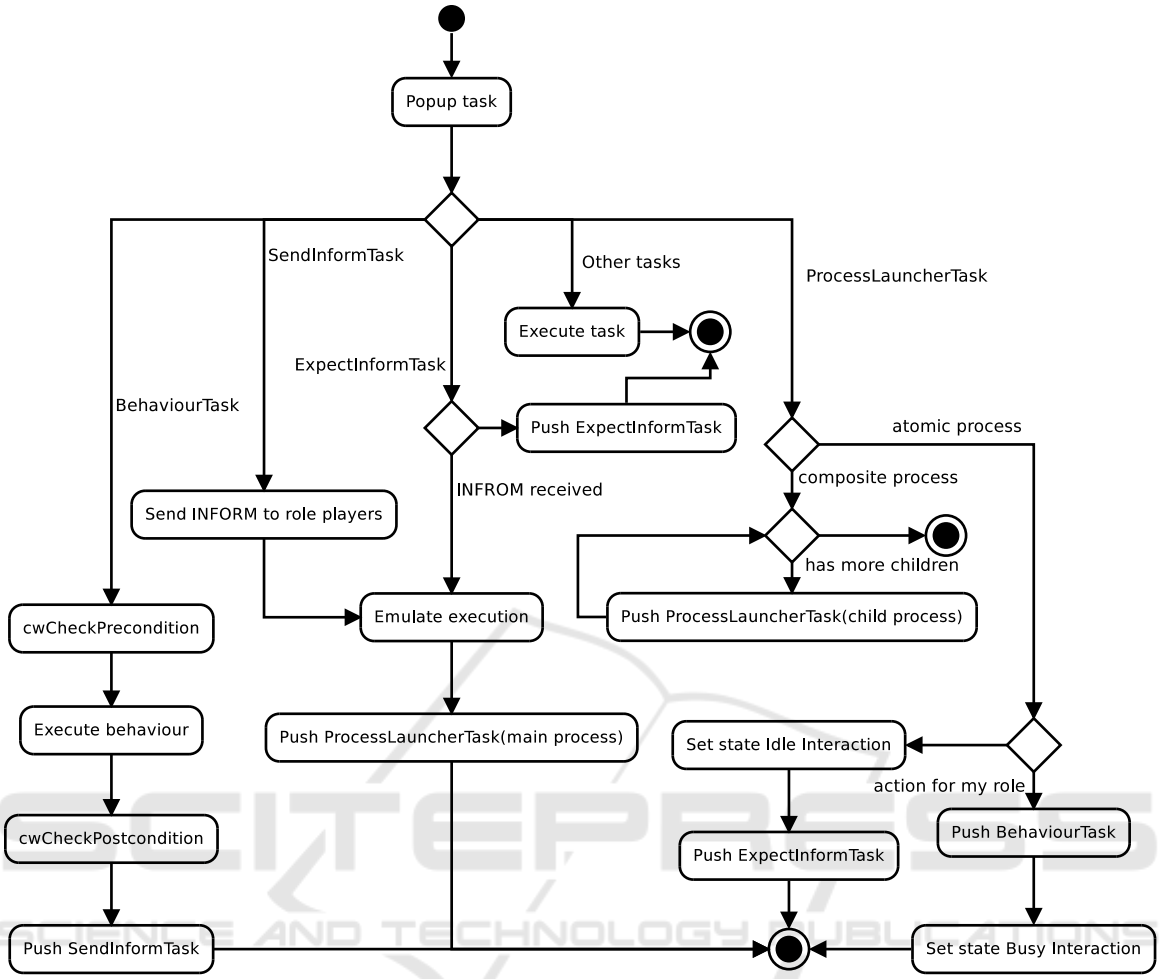

Figure 6: Activity diagram for processing tasks in RPI.Social.Exec.

be inferred from the system is automatically *false*. By contrast, in an open world, the system acknowledges the lack of the necessary information to know the veracity of a queried truth value and attributes to it the *unknown* value. In order to operate in an open world, RPI adopts Kleene's logic (Kleene, 1938) which is a ternary logic that recognizes a third truth value for the indeterminate.

The evaluation of expressions is procedural like a function call, i.e. the values of the arguments are passed to the predicates and the whole expression is evaluated for a truth value. If, for a given predicate, any of the arguments is unknown at the moment of the evaluation, that predicate's truth value is *unknown*.

The system for proving entailment is built on top of Prolog. The left side of the entailment is the first ground expression plus the set of axioms defined in the interaction protocol's RPI.Idiom specification, while its right side is the second ground expression. In RPI.Inference, we label the first and second expressions facts and crossfacts, respectively. The basic idea is to generate a Prolog program with the axioms and the facts. Then, the generated program is queried for individual literals in the crossfacts. If a query does not return *true*, a second one is launched with the negation of the same literal. If that second query does not return *true* as well, the truth value of the literal will be *unknown*, otherwise it will be *false*.

6 RELATED WORKS

The previous work (Nouri et al., 2015) argues that the works on first-class interaction protocols that are close to RPI.Idiom are the ARIP model (Khalfaoui and Chaari, 2013), the RASA language (Miller and McBurney, 2007) and the commitment machines (Yolum and Singh, 2002). We extend the comparison between these approaches to implementing first-class agent protocols and RPI to cover RPI.Social as well.

While RASA protocols are claimed to be executable, there is no explicit mechanism that specifies how to achieve that, thus this approach is eliminated from the comparison. ARIP model on the other hand relies on a third party language for implementing component interaction to specify actual agent protocol. While ARIP interaction protocols are executable as component interactions, the model is limited to the discovery and the instantiation of the protocols and has not any form of control over the flow of their execution. This is equivalent to having only the same functionalities provided by RPI.Social.IoP.

Considering the OWL-P ontology language (Desai et al., 2006) as a representative work for the commitment machines, its interaction protocols are compiled into Jess rules which then could be executed on demand. The execution of the interaction protocols in OWL-P seems to be comparable with RPI.Social.Exec. However, the initiation of a protocol is slightly different from RPI.Social.IoP. In fact, agents that wish to interact will have to pick their protocols from a repository with the possibility to compose them if needed. Then, the agent enacts one of the roles in interaction protocols and register itself as a service provider for the interaction. An initiator agent would seek a service provider and ask for the description of its role which is subsequently enacted for the span of the interaction.

XRole (Cabri et al., 2002) is an XML notation for roles in the BRAIN framework (Cabri et al., 2003). The interaction protocols in BRAIN are supported by XRole where each role has a set of actions that trigger events and a set of events that it listens to. The mediator, called the interaction infrastructure, is the part of the environment that generates events from the actions performed. RoleX (Cabri et al., 2004) is the part of BRAIN that dynamically enacts roles by agents. To this extent, RoleX covers a subset of RPI.Social.IoP functionalities since agents in RPI assume roles only for an imminent interaction. The mediator in BRAIN plays the role of an event bus that dispatches events to roles listening to them. Therefore, in BRAIN a third party entity coordinates between the interacting agents. The same functionality is covered by RPI.Social.Exec on an agent level.

In (Dastani et al., 2005), the authors introduce the concept of role enactment. The agent has four operators for assuming the role, playing it and for the opposite of these two operations, which makes them respectively *enact*, *activate*, *deact* and *deactivate*. While this proposition shares a common ground with RPI.Social.IoP, it has a strong organizational component that implies norms and restrictions on the model for implementing first-class interaction.

7 CONCLUSION

The RPI framework provides the tools necessary for the multiagent system developer to design and implement first-class interaction protocols that could be autonomously instantiated by the agents in runtime. The framework has a modular design in which loosely-coupled responsibilities within its ecosystem are assigned to different sub-systems.

Our contribution in this paper, is the introduction of RPI.Social which is the sub-system of the RPI framework that performs all the social behaviours of the agent on its behalf. It consists of two units. The first, RPI.Social.IoP, takes care of initiating interaction protocols and responding to invitations to play roles in others. The second unit which RPI.Social.Exec is responsible for the execution of the interaction and the coordination between the role players. The paper gives an overview on RPI.Idiom and RPI.Inference on which RPI.Social depends.

The literature on the execution of first-class agent protocols is very rare due to the relative scarcity of works on the implementation of reified protocols. Accordingly, assessing the relevance of the contribution of RPI.Social was made in comparison to some of the few works on the implementation of first-class interactions that explained their mechanisms for running their interaction protocols.

RPI.Social has the limitation of not being fully fault-tolerant. The bootstrap protocol defined by RPI.Social.IoP is made to be tolerant to the unavailability of agents during the negotiations for establishing an ad-hoc organization for the interaction. However, RPI.Social.Exec would not recover from events such as the sudden absence of an interacting agent.

In our future works we intend to add the support for fault tolerance to RPI.Social to overcome its limitation. Besides, we will focus on detailing the contribution of RPI.Inference and RPI.Library.

REFERENCES

Bauer, B., Muller, J. P., and Odell, J. (2000). An extension of UML by protocols for multi-agent interaction. In *Proceedings of the Fourth International Conference on MultiAgent Systems*, pages 207–214. IEEE.

Cabri, G., Ferrari, L., and Leonardi, L. (2004). The RoleX environment for multi-agent cooperation. In *Cooperative Information Agents VIII*, pages 257–270. Springer.

Cabri, G., Leonardi, L., and Zambonelli, F. (2002). XRole: XML roles for agent interaction. In *Proceedings of the 3rd International Symposium "From Agent Theory to Agent Implementation", at the 16th European*

Meeting on Cybernetics and Systems Research (EMCSR 2002), Wien.

Cabri, G., Leonardi, L., and Zambonelli, F. (2003). BRAIN: a framework for flexible role-based interactions in multiagent systems. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 145–161. Springer.

Dastani, M., van Riemsdijk, M. B., Hulstijn, J., Dignum, F., and Meyer, J.-J. C. (2005). Enacting and Deacting Roles in Agent Programming. In *Proceedings of the 5th International Conference on Agent-Oriented Software Engineering*, AOSE'04, pages 189–204. Springer.

Desai, N., Mallya, A. U., Chopra, A. K., and Singh, M. P. (2006). OWL-P: a methodology for business process development. In *Agent-Oriented Information Systems III*, pages 79–94. Springer.

Jensen, A. S. (2015). *The AORTA Reasoning Framework-Adding Organizational Reasoning to Agents*. PhD thesis, Department of Applied Mathematics and Computer Science, Algorithms and Logic, Department of Applied Mathematics and Computer Science, Department of Informatics and Mathematical Modeling, Technical University of Denmark.

Khalfaoui, S. and Chaari, W. L. (2013). Automatic Reuse of Interaction Protocols in MAS: ARIP Model. In *Advanced Methods and Technologies for Agent and Multi-Agent Systems, Proceedings of the 7th KES Conference on Agent and Multi-Agent Systems - Technologies and Applications (KES-AMSTA 2013)*, pages 315–324. IOS Press.

Kleene, S. C. (1938). On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(04):150–155.

Miller, T. and McBurney, P. (2007). Using constraints and process algebra for specification of first-class agent interaction protocols. In *Engineering Societies in the Agents World VII*, pages 245–264. Springer.

Morales, J., Wooldridge, M., Rodríguez-Aguilar, J. A., and López-Sánchez, M. (2017). Evolutionary Synthesis of Stable Normative Systems. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, pages 1646–1648, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Nouri, A., Chaari, W. L., and Ghedira, K. (2015). RPI.Idiom: A high-level language for first-class agent interaction protocols. In *Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of*, pages 1–8. IEEE.

Shoham, Y. and Leyton-Brown, K. (2008). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, New York, NY, USA.

Van Emden, M. H. and Kowalski, R. A. (1976). The Semantics of Predicate Logic As a Programming Language. *Journal of the ACM*, 23(4):733–742.

Yolum, P. and Singh, M. P. (2002). Commitment machines. In *Intelligent Agents VIII*, pages 235–247. Springer.