A Modelling Framework for Generic Agent Interaction Protocols

José Ghislain QUENUM[2], Samir AKNINE[1], Jean-Pierre BRIOT[1], and Shinichi HONIDEN[2]

[1] Laboratoire d'Informatique de Paris 6,
8 rue du Capitaine Scott, 75015 Paris, France
[2] National Institute of Informatics
2-1-2 Hitotsubashi, Tokyo 101-8430, Japan

Abstract. This paper presents a framework to represent generic protocols. We call generic protocols, agent interaction protocols where only a general behaviour of the interacting entities can be provided. Our framework is grounded on the AUML graphical formalism. From this formalism, we identified five fundamental concepts on top of which we defined the formal specifications for the framework. We address a lack in protocol representation by emphasising the description of actions performed in the course of interactions based on generic protocols. The framework is formal, expressive and of practical use. It helps decouple interaction concerns from the rest of agent architecture. Several application levels exist for our framework. First, we used it to address two issues faced in the design of agent interactions based on generic protocols. At a more concrete level, this framework can be used to publish the protocols agent interactions are based on in a multi-agent system.

1 INTRODUCTION

Interaction is one of the key aspects in agent-oriented design. It allows agents to put together the necessary actions in order to perform complex tasks collaboratively. The coordination mechanism needed for a safe execution of these actions is often governed by a sequence of message exchanges: interaction protocols. Usually, only a general description of how agents should behave during the interactions is provided. Such protocols are called generic protocols. An issue in open and heterogeneous multi-agent systems (MAS) is concerned with the description of generic protocols, especially with respect to their correct interpretation. A subsequent issue is the need to decouple interaction concerns from the rest of agent architecture.

To date, there has been some endeavour to develop new protocol representation formalisms. The formalisms developed thus far have several drawbacks. They usually focus on data exchange through a communication channel (Promela/SPIN [7]). Some others are either informal (or semi-formal) (e.g., AUML [1]) or demand advanced knowledge in logics (e.g., the formal framework in [10]). Therefore, there is an obvious need for a formal, yet practical and expressive generic protocol representation framework. Additionally, such a framework should provide the building blocks to help decouple the

interaction concerns from the rest of agent architecture. We address this need in this paper.

The solution we arrived at is a framework for the description of generic protocols. It complies with most of the criteria required of a conversation policy in [6]. The philosophy of our framework is to start from AUML, which is a well established agent interaction representation formalism. But we depart from AUML by addressing the lacks and incompleteness which limit it. A common trend in protocol representation consists of describing only the sequence of message exchanges. However, some actions are needed to produce these messages and handle them when received. As we will see later, some actions might be executed beyond the communication level during an interaction. Thus, in addition to the description of message exchange, our framework introduces the description of actions needed in the course of an interaction. This provides us with the ability of describing the behaviour agents will exhibit while playing a role in a protocol. A particular aspect in our framework is our focus on generic protocols. This keeps us from providing a complete representation for actions. We introduced action categories to fix this weakness.

Our framework offers several advantages. It builds on the graphical representation in AUML, which eases the message exchange perception for human designers. In addition, it offers the means to depict what happens beyond the message exchange level. The framework is expressive, formal and of practical use for protocol representation. Particularly, we offer at least the same expressiveness as in AUML (and its extensions) without introducing new control flows. Rather, we only use event description and (if necessary) three connectors: *and*, *or* and *xor*. We also ease the implementation of protocols in our framework by providing a XML representation. As an application, we used our framework to address two issues in agent interaction design of open and heterogeneous MAS: (1) an automatic derivation of agent interaction model from generic protocol specifications, in order to address the issue of consistency during interactions based on generic protocols in an heterogeneous MAS; and (2) an analysis of generic protocol specifications in order to enable agents to dynamically select protocols when they have to perform a task in collaboration. A more practical usage of our framework is the possibility to publish protocol specifications for agent interactions in a MAS.

The remainder of this paper is organised as follows. Section 2 discusses some related work. Section 3 introduces the fundamental concepts we use in the framework and presents both the specifications and their semantics. Section 4 discusses some properties one can check for a protocol represented following this framework. Finally, section 5 concludes the paper.

2 RELATED WORK

Several formalisms have been developed to represent interaction protocols. In this section, we discuss the most suitable ones for agent interaction.

AUML [1] and its extensions are graphical frameworks for protocol diagram representation. These frameworks, though practical and easy to use, are informal (or semi formal). It is then hard to check properties or even define the semantics of a protocol represented in these formalisms. [15] and [2] automate the translation process from

AUML to a textual description, which is more machine readable. The advantage of this automatic translation, though undebatable, is weakened by many other AUML original limitations, f.i., the lack of emphasis on action representation in protocol representation.

Some formal frameworks have been proposed for protocol representation. [14] defined a framework using concepts similar to ours. However, in our case agents are not represented in protocol specifications. They are expected to play (in a protocol configuration and instantiation standpoint) roles at runtime. [10] made significant advances in the area of protocol representation for agent interaction. This work developed a formal framework which combines Propositional Dynamic Logic and belief and intention modalities (PDL-BI). The framework covers a broad spectrum of issues related to agent interactions. However, it requires advanced knowledge in logics. In our opinion, logics is useful to define the semantics and check some properties for protocols. But due to the complexity it may introduce, we believe that it should be hidden at the specifications stage, as usually done in programming languages. Additionally, PDL-BI focuses on the messages exchange. But, as we showed above, agent interaction protocols demand more than message exchange.

IOM/T [3] is a more recent language for interaction representation. Our work, though sharing some similarities with IOM/T, departs from it on the following points. Firstly, we focus on generic protocols, where we consider generic actions. Secondly, the behaviour of the agents in IOMT/T (the actions they perform) is not associated with the events which occur in the MAS. Thirdly, the language is Java-like. However, we believe that a protocol description language is supposedly a declarative one. Especially for open and heterogeneous MAS. We address this need in this paper by developing a formal framework for generic protocols representation. Our framework is a declarative language and offers expressiveness as well as ease of use.

3 THE FRAMEWORK

We introduce the fundamental concepts our framework is based on. Then, we present the specifications and the semantics of these concepts.

3.1 FUNDAMENTAL CONCEPTS

Our framework is based on the AUML protocol diagram. Thereof, we identified five fundamental concepts: *protocol*, *role*, *event*, *action* and *phase*. A graphical illustration of these concepts is given in Fig. 1.

Definition 1. Protocol

A protocol is a sequence of message exchanges between at least two roles. The exchanged messages are described following an Agent Communication Language (ACL) e.g., FIPA ACL [5], KQML [9].

More formally, a protocol consists of a collection of roles $\mathbf{R}$, which interact with one another through message exchange. The messages belong to a collection $\mathbf{M}$ and the exchange takes place following a sequence, $\mathbf{\Omega}$. A protocol also has some intrinsic properties $\mathbf{\Theta}$ (attributes and keywords) which are propositional contents that provide

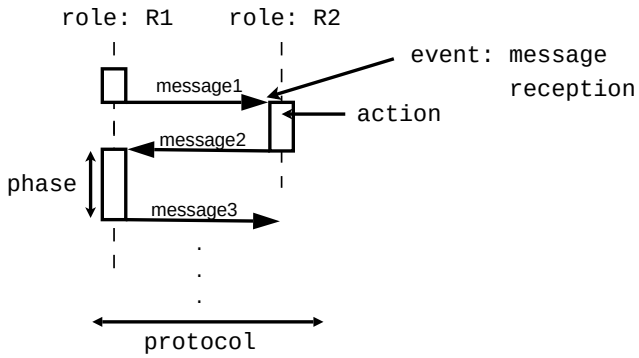

Fig. 1. Graphical illustration of concepts in generic protocols.

a context for further interpretation of the protocol. We note $\mathbf{p} \stackrel{\text{def}}{=} < \mathbf{\Theta}, \mathbf{R}, \mathbf{M}, \mathbf{\Omega} >$. In $\mathbf{\Omega}$, the message exchange sequence, each element is denoted by $r_i \xrightarrow[a_\alpha, m_{k-1}]{m_k} r_j$, to be interpreted as "the role r_i sends the message m_k to r_j, and that m_k is generated after action a_α's execution and the prior exchange of m_{k-1}". Additional elements may be introduced in this representation, but we do not discuss them in this paper.

Definition 2. Generic Protocol

A generic protocol is a protocol wherein the actions which are taken, to handle, produce the contents of exchanged messages, etc. cannot be thoroughly described. A complete description of these actions depends on the architecture of each agent playing a role in the protocol.

Note that the attributes and keywords we use in the current version have been identified from our experimentations. Currently, we only use three attributes: *class*, *return_value* and *participants_count*. *class* is the type of processing performed through the execution of an interaction based on this protocol (e.g., we use *request* to denote that the participant performs some task on behalf of the initiator). *return_value*, when any, depicts how the final result is represented. *participants_count* is the number of distinct participant roles in the protocol. As for keywords, several ones can be used in the current version. For example, *IncrementalProcess* means that some partial results may be considered for the ongoing process. Some more experiments are needed to extend these attributes and keywords.

Each of the communicating entities is called a role. Roles are understood as standardised patterns of behaviour required of all agents playing a part in a given functional relationship in the context of an organisation [4].

Definition 3. Role *In our framework, a role consists of a collection of phases. As we will see later in Section 3.2, a role may also have global actions (which are executed outside all phases) and some data other than message content,* variables*.*

$\forall \mathbf{r} \in \mathbf{R}, \mathbf{r} \stackrel{\text{def}}{=} < \mathbf{\Theta_r}, \mathbf{P_h}, \mathbf{A_g}, \mathbf{V} >$, where $\mathbf{\Theta_r}$ corresponds to the role's intrinsic properties (e.g., cardinality) which are propositional contents that help further interpret the

role, $\mathbf{P_h}$ the set of phases, $\mathbf{A_g}$ the set of global actions and $\mathbf{V}$ the set of variables. The roles can be of two types: (1) *initiator*, the unique role of the protocol in charge of starting its execution; (2) *participant*, any role partaking in an interaction based on the protocol.

The behaviour of a role is governed by events. An event is an atomic change which occurs in the environment of the MAS. An informal description of the types of events we consider in our framework is given in Table 1. A more formal interpretation of these events is discussed in Section 3.3. The behaviour a role adopts once an event occurs is described through actions.

Event Type	**Description**
Change	The content of a variable has been changed.
Endphase	The current phase has completed.
Endprotocol	The end of the protocol is reached.
Messagecontent	The content of a message has been constructed.
Reception	A new message has been received.
Variablecontent	The content of a variable has been constructed.
Custom	Particular event (error control or causality).

Table 1. Event Types.

Definition 4. Action

An action is an operation a role performs while executing. This operation transforms the whole environment or the internal state of the agent currently playing this role. An action has a category ν, a signature Σ and a set of events it reacts to or produces. We note $a \stackrel{def}{=} < \nu, \Sigma, E >$.

Since our framework focuses on generic protocols, we can only provide a general description for the actions which are executed in these protocols. Hence, we introduced action categories to ease the definition of a semantics for these actions. Table 2 contains an informal description of these categories. We discuss their semantics in Section 3.3.

Definition 5. Phase *Some successive actions sharing direct links can be grouped together. Each group is called a phase. Two actions share a direct link if the (or only a part of them) input arguments of one are generated by the other (f.i. when a* send *action sends the message generated in a prior action).*

3.2 FORMAL SPECIFICATIONS

The formal specifications are defined through a EBNF grammar. Only essential parts of this grammar are discussed in this section. A thorough description of this grammar is given in Appendix A. In sake of easy implementation of generic protocols, we represent

Action Type	Description
Append	Adds a value to a collection.
Remove	Removes a value from a collection.
Send	Sends a newly generated message.
Set	Sets a value to a variable.
Update	Updates the value of a variable.
Compute	Computes a new information.

Table 2. Action categories.

them in XML in our framework. However, as XML is too verbose, a simpler (bracket-based) representation will be used for illustration in this paper.

RUNNING EXAMPLE We will use the Contract Net Protocol (CNP) [13] to illustrate the specifications we present. The sequence diagram (protocol diagram in AUML) of this protocol is given in Fig. 2. The labels placed on message exchange arrows in the figure are not performatives, but message identifiers.

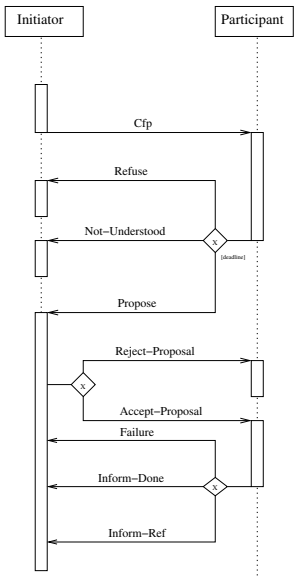

Fig. 2. The Contract Net Protocol.

The rationale of the CNP consists of an initiator having some participants perform some processing on its behalf. But beforehand, the participants which will perform the

processing are selected on the basis of the bids they proposed, in-reply to the initiator's call for proposals. When the selected participants are done with their processing, each of them notifies the initiator agent of the correct execution (or error occurrence) of the part it committed in performing.

PROTOCOL The following production rules define a protocol:
<protocol>:=<protproperties><roles><messagepatterns>
<protproperties>:=<protdescriptors><protattributes><protkeywords?>
<protdescriptors>:=<identifier><title><location>
<protattributes>:=<class><participantcount><return>
<protkeywords>:=<protkeyword+> <protkeyword>:="IncrementalProcess"|...
An illustration of these rules is given as follows.

```
(protocol
 (protocolproperties
  (protocoldesc ident='cnpprot' title='ContractNet' location='KqmlCnp.xml')
  (protocolattr class='Request' participantcount='1' return='operationresult)
  (protkeyws '...'))
 (roles ...)
 (messagepatterns ...))
```

As one can see from these rules, the exchange sequence Ω is not explicitly specified. Actually, it is described through *send* actions in each role. When several messages can be sent, we use connectors *and*, *or* and *xor* to compose them.

ROLE Protocol diagrams only show the communication flow between roles. However, there may be some information beyond the communication level. For example, in the CNP, the action an initiator executes in order to make a decision upon the participants' bids is hidden behind the communication flow. Actually, this action exploits information from different participants of the protocol. Moreover, information like the deadline for bidding, cannot be extracted from any message content. Then, we introduced a global area for each role where we describe actions which are beyond the communication flow, as well as data which cannot be extracted from any message content. Note that actions relevant to the global area are no more associated with any phase. The production rules hereafter define a role.
<roles>:=<role><role>|<roles><role>
<role>:=<roleproperties><variables?><actions?><phases>
<roleproperties>:=<roledescriptors><roleattributes><rolekeywords?>
<roledescriptors>:=<identifier><name>
<roleattributes>:=<cardinality>
<variables>:=<variable+> <variable>:=<ident><type>
<actions>:=<action+>
Each role is described through its intrinsic properties (f.i., name and cardinality), its variables (pieces of information the role handles which are not extracted from any message content), its global actions and the phases the non global actions are grouped in.

In the example below, the *initiator* role of the CNP has three variables: `deadline`, `bidsCol` and `deliberations`. `deadline` contains the time when bidding should

stop. `bidsCol` is a collection where participants' bids are stored. `deliberations` contains the decision (accept or reject) the initiator made upon each bid. Each variable has an identifier and the type of the data it contains. The content of a variable is characterised using some abstract data types. We also use these data types to represent message content and action signature. String, Number and Char are some examples of the data types we use in our framework. The description of these types is out of the scope of this paper. The only global action in this role is named `Deliberate`. Through this action, the initiator makes a decision upon the participants' bids. Global actions are described in the same way like local (located in a phase) ones: category, signature and events. The description of `Deliberate` explains itself from the example. The special word *eventref* is used here to refer to an event defined elsewhere (*change* event which occurred against the `bidsCol` variable). As we will see later, this word sometimes introduces causality between actions.

```
(role ident='initiator'
 (roleproperties (roledescriptors ident='initiator' name='Initiator')
  (roleattributes cardinality='1'))
 (variables (variable ident='bidsCol' type='collection')
  (variable ident='deliberations' type='map')
  (variable ident='deadline' type='date'))
 (actions(action category='compute' description='Deliberate'
   (signature (arg type='date' dir='in')
    (arg type='collection' dir='in')(arg type='map' dir='out'))
   (events (event type='change' dir='in' object='deadline' ident='evt0')
    (eventref dir='in' ident='evt5')
    (event type='change' dir='out' object='deliberations' ident='evt1'))))
  (phases ...))
```

PHASE As stated above, each phase is a sequence of actions that share some direct links. We use the following rules to define a phase:
<phases>:=<phase+>
<phase>:=<actions>
<action>:=<category><description?><signature><events>
For example, in the initiator role of the CNP, the first phase consists of producing and sending the `cfp` message. This phase contains two actions: `prepareCFP` and `sendCFP`. `prepareCFP` produces the `cfp` message. It is followed by `sendCFP` which sends the message to each identified participant.

```
(phase ident='phs1'
 (actions (action category='compute' description='prepareCFP'
   (signature(arg type='date' dir='in')(arg type='any' dir='out'))
   (events (event type='variablecontent' dir='in' object='deadline')
    (event type='messagecontent' dir='out' object='cfp' ident='evt2')))
  (action category='send' description='sendCFP'
   (signature (message ident='cfp'))
   (events(eventref dir='in' ident='evt2')
   (eventref type='custom' dir='out' ident='cus01')
    (event type='endphase' dir='out' ident='evt3')))))
```

MESSAGE Though we did not define messages as a concept, we use them in the formal specifications because they contain part of the information manipulated during

interactions. The concept of message is well known in ACL, and their semantics is defined accordingly.

We propose an abstract representation of messages, which we call *message patterns*. A message pattern is composed of the performative and the content type of the message. We also offer the possibility to define the content pattern, a UNIX-like regular expression which depicts the shape of the content. Note that at runtime, these messages will be represented with all the fields as required by the adopted ACL. In our framework, we represent all the message patterns once in a block and refer to them in the course of the interaction when needed. In our opinion, it sounds to constrain to the use of only one ACL all along a single protocol description. The following rules define message patterns.

<messagepatterns>:=<acl><messagepattern+>
<acl>:='fipa'|'kqml'
<messagepattern>:=<performative><identifier><content>
<content>:=<type><pattern?>
The example below describes the message patterns used in the CNP.

```
(messagepatterns acl='Kqml'
 (messagepattern performative='achieve' ident='achmsg'
  (content type='any' pattern='...'))
 (messagepattern performative='sorry' ident='refuse'
  (content type='null' pattern='...'))
 (messagepattern performative='tell' ident='propose'
  (content type='any' pattern='...'))
 (messagepattern performative='deny' ident='reject'
  (content type='null' pattern='...'))
 (messagepattern performative='tell' ident='accept'
  (content type='string' pattern='...')) ...)
```

DESIGN GUIDELINE As a guideline for protocol design and description in our framework, we recommend several design rules. They guarantee the correctness of a protocol represented in our framework. We introduce some of them here.

Proposition 1. *For each role of a protocol, there should be at least one action which drives into the terminal state. Every such action should be reachable from the role's initial state.*

Corollary 1. *From their semantics, roles can be represented as graphs. And for every path in this graph, there should be an action which drives to a terminal state.*

Proposition 2. *When two distinct transitions can be fired from a state, the set of events which fire each one of the transitions, though intersect-able, should be distinguishable.*

Proposition 3. *When an action produces a message, it should be immediately followed by a* send *action, which will be responsible for sending the message.*

3.3 SEMANTICS OF THE CONCEPTS

EVENT As we saw, an event informs of an atomic change. It may have to do with the notified role's internal state. But usually, the notification is about other roles' internal

state. Therefore, events are the grounds for roles coordination. Due to space constraints, we only discuss the semantics of two types of events in this section.

change: this event type notifies of a change of the value of a variable. Let v be this variable, *change(v)* denotes this event. In order to define the semantics of our concepts, we introduce some expressions in a meta-language, which we call primitives. These primitives are functions and predicates. *value* is one of these primitives (actually a function). It returns the value of a data at a given time. Let $\mathcal{T}$ be the time space, and d and t a data and a time respectively ($t \in \mathcal{T}$), $Value(d, t)$ denotes this function. Value$(d, t) = \emptyset$ means that the data d does not exist yet at time t. We interpret the *change* event as follows: $\exists\ (t_1, t_2) \in \mathcal{T} \times \mathcal{T}$:

$$(t_1 \neq t_2) \wedge (\text{Value}(v, t_1) \neq \emptyset) \wedge (\text{Value}(v, t_1) \neq \text{Value}(v, t_2))$$

endprotocol: this event type notifies of the end of the current interaction. For each role, all the phases have either completed or are unreachable. Also any global action of each role is either already executed or unreachable. A phase is unreachable if none of its actions is reachable. Actually, if the initial action is unreachable, the phase it belongs to will also be unreachable. Again, we introduced three new primitives: *Follow*, *Executed* and *Unreachable*. *Follow* is a function which returns all the immediate successors of a phase. Let p_1 and p_2 be two phases, p_1 immediately follows p_2, if any of the input events of the initial action of p_1 refers to a prior event generated by one of the actions (usually the last one) of p_2. *Unreachable* is a predicate which means that the required conditions for the execution of an action do not hold. Therefore, this action cannot be executed. Finally, *Executed* is a predicate which means that an action has already been executed. Let P_r be the set of phases and $A_{p_{kr}}$ the set of executable actions for phase p_{kr} in a role r. Let also A_{G_r} be the set of global actions for role r. We interpret the *endprotocol* event as follows:$\forall r \in \mathcal{R}, \forall a_\alpha \in A_{G_r}$,

$$(\text{Unreachable}(a_\alpha) \vee \text{Executed}(a_\alpha)) \wedge (\forall p_{kr} \in P_r,\ (\text{Follow}(p_{kr}) = \emptyset) \vee (\forall a_i \in A_{p_{kr}}, \text{Unreachable}(a_i)))$$

ACTION Actions are executed when events occur. And once executed, they may produce some new change in the MAS. Events are therefore considered as *Pre* and *Post* conditions for actions' execution. Here again, we only discuss the semantics of the *append* and *send* action categories.

Let $\mathcal{E}$ be the set of all the event types we consider in our framework. We define $\mathcal{E}'$ as a subset of $\mathcal{E}$: $\mathcal{E}' = \mathcal{E} - \{endphase, endprotocol\}$.

append: this action adds a data to a collection. Let a_i be such an action,

Pre $= \bigvee_j e_j$, where $e_j \in \mathcal{E}'$
Post $= \bigvee_j e_j, \exists k\ e_k =' change' \wedge (\exists (t_1, t_2) \in \mathcal{T} \times \mathcal{T}, \exists d, v \in args(a_i),$
$(t_1 < t_2) \wedge (\text{isElement}(v, d, t_1) = false) \wedge (\text{isElement}(v, d, t_2)))$

isElement() is a predicate which returns true when a data belongs to a collection at a given time. *args()* returns the arguments of an action.

send: this action sends a message. It is effective both at the sender and the receiver sides. Let a_i be such an action. We interpret it as follows: at the sender side:

Pre $= \bigvee_j e_j$, where $\forall m_j \in arguments(a_i), \exists\, k,\, e_k =$ messagecontent(m_j)
Post $= (Trans(m_j) = true)$

at the receiver side:

Pre $= \emptyset$
Post $= \bigvee_j e'_j$, where $\forall m_j \in arguments(a_i), \exists!\, k,\, e'_k =$ reception(m_j)

ACL usually define the semantics of their performatives by considering the belief and intention of the agents exchanging (sender and receiver) these performatives. This approach is useful to show the effect of a message exchange both at the sender and the receiver sides. In our framework, we adopt a similar approach when an action produces or handles a message. We use the knowledge the agent performing this action has with respect to the message. Hence, we introduce a new predicate, $Know(\phi, a_g)$, which we set to true when the agent a_g has the knowledge ϕ. $Know$ is added to the post conditions of the action when the latter produces a message. It is rather added to the pre conditions of the action when it handles a message. Note that ϕ is the (propositional) content of the message. Moreover, when an action ends up a phase or the whole protocol, its Post condition is extended with the *endphase* and *endprotocol* events respectively.

PHASE The semantics of a phase is that of a collection of actions sharing some causality relation. The direct links between actions of a phase are augmented with a causality relation introduced by events. We note $\mathbf{p_h} \stackrel{\text{def}}{=} < \mathbf{A}, \prec >$, where $\mathbf{A}$ is a set of actions and $\prec$ a causality relation which we define as follows ($|\mathbf{A}|$ is the cardinality of $\mathbf{A}$):
$\forall \mathbf{a}_\imath, \mathbf{a}_\jmath \in \mathbf{A}, \mathbf{a}_\imath \neq \mathbf{a}_\jmath, \mathbf{a}_\imath \prec \mathbf{a}_\jmath \Longleftrightarrow |\mathbf{A}| > 1 \ \wedge\ \exists e \in Post(\mathbf{a}_\imath), e \in Pre(\mathbf{a}_\jmath)$.

Proposition 4. *Let* $\mathbf{a}_\imath$ *and* $\mathbf{a}_\jmath$ *be elements of* $\mathbf{P_h}$*, such that* $\mathbf{a}_\imath$ *always precedes* $\mathbf{a}_\jmath$*,*

$$(\mathbf{a}_\imath \prec \mathbf{a}_\jmath) \vee (\exists \mathbf{a_p}, \ldots, \mathbf{a_k}, \mathbf{a}_\imath \prec \mathbf{a_p} \ldots \prec \mathbf{a_k} \prec \mathbf{a}_\jmath)$$

ROLE An event generated at the end of a phase can be referred to in other phases. Thus, the causality relation between actions of phases can be extended to interpret roles. We consider a role as a labelled transition system having some intrinsic properties. $\mathbf{r} \stackrel{\text{def}}{=} < \mathbf{\Theta_r}, \mathbf{S}, \mathbf{\Lambda}, \longrightarrow >$ where:

- $\mathbf{\Theta_r}$ are the intrinsic properties of the role;
- $\mathbf{S}$ is a finite set of states;
- $\mathbf{\Lambda}$ contains transitions labels. These are the actions the role performs while running;
- $\longrightarrow \subseteq \mathbf{S} \times \mathbf{\Lambda} \times \mathbf{S}$ is a transition function.

As an illustration, we give part of the semantics of the initiator role of the CNP, which we call $\mathbf{r_0}$. $\mathbf{r_0} = < \mathbf{\Theta_{r_0}}, \mathbf{S}, \mathbf{\Lambda}, \longrightarrow >$, with:

- $\mathbf{\Theta_{r_0}} =' cardinality = 1 \ \wedge\ isInitiator = true \ \ldots'$;

– $\mathbf{S} = \{\mathbf{S_0}, \mathbf{S_1}, \mathbf{S_2}, \mathbf{S_3}, \mathbf{S_4}, \mathbf{S_5}, \mathbf{S_6}, \mathbf{S_7}, \mathbf{S_8}, \mathbf{S_9}, \mathbf{S_{10}}, \mathbf{S_{11}}\}$;
– $\mathbf{\Lambda} = \{\mathbf{a_0}, \mathbf{a_1}, \mathbf{a_2}, \mathbf{a_3}, \mathbf{a_4}, \mathbf{a_5}, \mathbf{a_6}, \mathbf{a_7}\}$;
– $\longrightarrow = \{(\mathbf{S_0}, \mathbf{a_0}, \mathbf{S_1}), (\mathbf{S_1}, \mathbf{send}_{[\mathbf{m_0}]}, \mathbf{S_2}), (\mathbf{S_2}, \mathbf{a_1}, \mathbf{S_7}), (\mathbf{S_2}, \mathbf{a_2}, \mathbf{S_3}), (\mathbf{S_3}, \mathbf{a_4}, \mathbf{S_4}), (\mathbf{S_4}, \mathbf{a_3}, \mathbf{S_5}), (\mathbf{S_5}, \mathbf{send}_{[\mathbf{m_3}]}, \mathbf{S_{11}}), (\mathbf{S_5}, \mathbf{send}_{[\mathbf{m_4}]}, \mathbf{S_6}), (\mathbf{S_6}, \mathbf{a_5}, \mathbf{S_8}), (\mathbf{S_6}, \mathbf{a_6}, \mathbf{S_9}), (\mathbf{S_6}, \mathbf{a_7}, \mathbf{S_{10}})\}$

m_0, m_3 and m_4 belong to the set of messages exchanged during the protocol.

PROTOCOL The semantics of a protocol is a combination of the semantics of its intrinsic properties, that of each role and finally the semantics of the coordination mechanism. Recall that the coordination mechanism, in our case, is the sequence of message exchanges. The sequence of messages actually exchanged during the interaction is known only at runtime. This raises up one of the limitations of the work concerned with agent interaction protocols semantics. They usually proposed *a priori* semantics for protocols. However, as protocols generally offer several possible exchange sequences, several possible semantics may coexist for an interaction based on a protocol. [8] proposed *a posteriori* semantics through a platform called *Protocol Operational Semantics (POS)*. We build on this platform and adopt *a posteriori* semantics for protocols in our framework. Two main reasons account for such an approach. Firstly, the messages exchange sequence can be mapped to a graph of possibilities for exchanged messages. Therefore, the semantics of an interaction based on this protocol consists of a path in this graph. Secondly, the semantics of communicative acts defined in ACL is not enough to define the semantics of a protocol. The executed actions' semantics should also be included. However, apart from send actions, all the other actions can only have general characterisation before the execution of the interaction. A more precise semantics of these actions can only be known at runtime.

As an illustration, let us assume that the semantics of each role of the CNP is known, we define that of the whole protocol as follows. $\mathbf{p} =< \mathbf{\Theta}, \mathbf{R}, \mathbf{M}, \mathbf{\Omega} >$, where $\mathbf{R} = \{\mathbf{r_0}, \mathbf{r_1}\}$ and $\mathbf{M} = \{\mathbf{m_0}, \mathbf{m_1}, \ldots \mathbf{m_7}\}$. $\mathbf{m_0}$ corresponds to `cfp`, $\mathbf{m_1}$ corresponds to `refuse`, etc. (see Fig. 2). $\mathbf{\Omega} = \mathbf{\Omega_1}|\mathbf{\Omega_2}|\mathbf{\Omega_3}|\mathbf{\Omega_4}|\mathbf{\Omega_5}|\mathbf{\Omega_5}|\mathbf{\Omega_6}$. $\mathbf{\Omega_6}$ is the case where everything went correctly and the participant notifies the initiator of the correct performance of the task. $\mathbf{\Omega_6} =< r_0 \xrightarrow[a_0]{m_0} r_1, r_1 \xrightarrow[a_8, m_0]{m_2} r_0, r_0 \xrightarrow[a_3, m_2]{m_3} r_1, r_1 \xrightarrow[a_{10}, m_4]{m_7} r_0 >$

4 PROPERTIES

4.1 LIVENESS

Proposition 5. *For every role of a protocol, events will always occur and fire some transition until the concerned role enters a terminal state.*

Proof. Each role is a transition system. And from the description of transition systems, unless an error occurs, an event will always occur and require to fire a transition until the role enters a terminal state, where the execution stops.

4.2 SAFETY

We consider two safety properties: *consistent messages exchange* and *Unambiguous protocol execution*.

Proposition 6. Consistent messages exchange *Messages exchange is consistent in our framework. Precisely, any message a role sends is received and handled at least by one role. By the same token, any message a role receives has a sender (generally another role).*

Proof. We prove both parts of the proposition.

1. The sequence of message exchanges $\mathbf{\Omega}$ is described as follows: $\mathbf{\Omega} =< r_0 \xrightarrow[a_0]{m_0} r_1, r_1 \xrightarrow[a_8,m_0]{m_2} r_0, \ldots >$. This representation shows that any message sent is received and handled by at least one role.
2. Any received message has been generated elsewhere, since its identifier exists. Additionally, from proposition 3, any message generated is automatically sent. Hence, any message received is sent by a role.

Proposition 7. Unambiguous protocol execution *For each action a role can take, there is an unambiguous set of events which fire its execution.*

Proof. The proof follows from the direct application of proposition 2 and is omitted here because of space constraints.

4.3 TERMINATION

Proposition 8. *Each role of a protocol represented in our framework always terminates.*

Proof. From Proposition 1, each role has at least an action which brings that role to a terminal state. Once this terminal state is reached, the interaction stops for the concerned role. When all the roles enter a terminal state, the whole interaction definitely stops.

This proof is insufficient when there are several alternatives or loops in the protocol. Corollary 1 addresses this case. Actually, only one path of the graph (with respect to the transition system) corresponding to the current role will be explored. And as this path ends up with an action driving to a terminal state, the role will terminate.

5 CONCLUSION

We believe that a special care is needed in representing generic protocols, since only partial information can be provided for them. Therefore, We developed a framework to represent generic protocols for agent interactions. Our framework puts forth the description of the actions performed by the agents during interactions, and hence highlights their behaviour during protocols execution. In this, we depart from the usual protocol representation formalisms which only focus on exchanged messages descriptions. Our framework is based on a graphical formalism, AUML. It is formal, at least as expressive

as AUML (and its extensions) and of practical use. As we discussed in the paper, this framework has been used to address issues in agent interaction design.

Since actions in generic protocols can be described only in a general way, a more precise description of these actions is dependent on the architecture of the agent about to perform them in the context of an interaction. This is usually done by hand by agent designers when they have to configure agent interaction models. Doing such a configuration by hand may lead to inconsistent message exchange in an heterogeneous MAS. We address this issue by developing some mechanisms to automatically carry this configuration out. These mechanisms consist of looking for similarities between the functionalities from agent architecture and actions of protocols. These mechanisms are presented in [12].

Protocol selection is another issue we faced while designing agent interactions based on generic protocols. Usually, agent designers select the protocols their agents will use to interact during the performance of collaborative tasks. But this static protocol selection severely limits interaction in open and heterogeneous MAS. Thus, we developed some mechanisms to enable agents to dynamically select the protocol they will use to interact. These mechanisms require some reasoning about the specifications of the protocols. Again, we used this framework, since it enables us to reason about the mandatory coordination mechanisms for the performance of collaborative tasks. We described part of the mechanisms we proposed in [11].

References

1. B. Bauer and J. Odell. UML 2.0 and Agents: How to Build Agent-based Systems with the new UML Standard. *Journal of Engineering Applications of Artificial Intelligence*, 18:141–157, 2005.
2. G. Casella and V. Mascardi. From AUML to WS-BPEL. Technical report, Computer Science Department, University of Genova, Italy, 2001.
3. T. Doi, Y. Tahara, and S. Honiden. IOM/T: an Interaction Description Language for Multi-agent Systems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 778–785, 2005.
4. M. Esteva, J. A. Rodriguez, C. Sierra, P. Garcia, and J. L. Arcos. On the Formal Specification of Electronic Institutions. In *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*. 2001.
5. FIPA. FIPA Communicative Act Library Specification. Technical report, Foundation for Intelligent Physical Agents, 2001.
6. M. Greaves, H. Holmback, and J. Bradshaw. Waht is a Conversation Policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents 1999*, 1999.
7. G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.
8. J-L Koning and P-Y Oudeyer. Introduction to POS: A Protocol Operational Semantics. *International Journal on Cooperative Information Systems*, 10(1 2):101–123, 2001. Special Double Issue on Intelligent Information Agents: Theory and Applications.
9. Y. Labrou and T. Finin. A proposal for a new KQML Specification. Technical report, University of Maryland Baltimore County (UMBC), 1997.

10. S. Paurobally, J. Cunningham, and N. R. Jennings. A Formal Framework for Agent Interaction Semantics. In *Proceedings. 4th International Joint Conference on autonomous Agents and Multi-Agent Systems*, pages 91–98, Utrecht, The Netherlands, 2005.
11. J. G. Quenum and S. Aknine. A Dynamic Joint Protocols Selection Method to Perform Collaborative Tasks. In P. Petta M. Pechoucek and L.Z. Varga, editors, *4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2005)*, LNAI 3690, pages 11–20, Budapest, Hungary, September 2005. Springer Verlag.
12. J. G. Quenum, A. Slodzian, and S. Aknine. Automatic Derivation of Agent Interaction Model from Generic Interaction Protocols. In P. Giorgini, J. P. Muller, and J. Odell, editors, *Proceedings of the Fourth International Workshop on Agent-Oriented Software Engineering*. Springer Verlag, 2003.
13. G. Smith. The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. *IEEE Trans. on Computers*, 29(12):1104–1113, 1980.
14. C. Walton. Multi-agent Dialogue Protocols. In *Proceedings of the Eight Int. Sympossium on Artificial Intelligence and Mathematics*, 2004.
15. M. Winikoff. Towards Making Agent UML Practical: A Textual Notation and Tool. In *Proc. of the First Int. Workshop on Integration of Software Engineering and Agent Technology (ISEAT)*, 2005.

A EBNF Grammar

```
  <protocol>:=<protproperties><roles><messagepatterns>
<protdescriptors>:=<protdescriptors><protattributes><protkeywords?>
<protdescriptors>:=<identifier><title><location>
<protattributes>:=<class><participantcount><return>
<protocolkeywords>:=<protocolkeyword+>
<protocolkeyword>:="containsconcurrentroles"|"iterativeprocess"|
                   "incrementalprocess"|"subscriptionrequired"|
                   "alterableservicecontent"|"alterableproposalcontent"
 |"dividableservice"
<title>:=<word>
<class>:=<word>
<location>:=<locationheader><path>
<locationhearder>:="http://www."|"http://"|"file://"|"ftp://ftp."
<path>:=<directory+><word>''.''<word>
<directory>:=<word>
<participantcount>:=<digit+>|"n"
<return>:="operationresult"|"information"|"agentaddress"
<roles>:=<role><role>|<roles><role>
<messagepatterns>:=<acl><messagepattern+>
<role>:=<roleproperties><variables?><actions?><phases>
<roledescriptors>:=<roledescriptors><roleattributes><rolekeywords?>
<roledescriptors>:=<identifier><name>
<roleattributes>:=<cardinality><concurrentparticipants?>
<concurrentparticipantset>:=<identifier+>
<rolekeywords>:=<rolekeyword+>
<rolekeyword>:=<word>
```

```
<name>:=<word>
<cardinality>:=<digit+>|"n"
<variables>:=<variable+>
<variable>:=<identifier><type>
<type>:="number"|"string"|"char"|"boolean"|"date"|
               "collection"|"null"|"any"|"map"
<identifier>:=<letter+> "id" <digit+>
<letter>:="a"|"b"|"c"|..."z"
<digit>:="0"|"1"|"2"|...|"9"
<word>:=<letter+>
<space>:=""
<actions>:=<action+>
<phases>:=<phase+>
<phase>:=<identifier><actions>
<action>:=<category><description?><signature?><events>
<description>:=(<word><space?>)*
<category>:="append"|"custom"|"remove"|"send"|"set"|"update"
<signature>:=<arguments>|<messages>
<arguments>:=(<argset>|<argdesc>)+
<argset>:=<settype>(<argset>|<argdesc>)+
<argdesc>:=<identifier><type><direction>
<direction>:="in"|"out"|"inout"
<messages>(<message>|<messageset>)+
<message>:=<identifier>
<messageset>:=<settype>(<messageset>|<message>)+
<settype>:="and"|"or"|"xor"
<events>:=(<event>|<eventref>|<eventset>)+
<eventset>:=<settype>(<event>|<eventref>|<eventset>)+
<event>:=<identifier?><eventtype><object>
<eventtype>:="change"|"custom"|"emission"|"endphase"|
           "endprotocol"|"messagecontent"|"reception"|"variablecontent"
<object>:=<message>|<variableid>
<variableid>:=<identifier>
<eventref>:=<identifier>
<messagepattern>:=<identifier><performative><content>
<performative>:=<fipaperformative>|<kqmlperformative>
<kqmlperformative>:="ask-all"|"ask-one"|"ask-if"|"stream-all"|...
<acl>:="fipa"|"kqml"
<content>:=<type><pattern?>
<pattern>:=<word*><space>
```