A Modeling 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
{Samir.Aknine,Jean-Pierre.Briot}@lip6.fr
[2] National Institute of Informatics
2-1-2 Hitotsubashi, Tokyo 101-8430, Japan
{joque,honiden}@nii.ac.jp

Abstract. Agent-UML (AUML) extended UML in order to facilitate the modeling process for agent based systems. It offers several graphical notations, including protocol diagrams which represent agent interaction protocols. In this paper, we describe an AUML-based framework to specify generic protocols. We call generic protocols, agent interaction protocols where only a general behavior of the interacting entities can be described. From AUML protocol diagrams, we identified five fundamental concepts on top of which we defined formal specifications of generic protocols. Through our specifications, we addressed a lack in generic protocol representation by emphasizing the description of actions performed in the course of interactions based on such protocols. The framework we developed is formal, expressive and of practical use. It helps decouple interaction concerns from the rest of an agent's architecture. As an application, we used this framework to publish the specifications of generic protocols for agent interactions in several multi-agent system applications we developed. Additionally, the framework helped us address two issues faced in the design of agent interactions based on generic protocols, protocol configuration and their dynamic selection.

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 mechanisms needed for a safe performance of these actions are often represented as a sequence of message exchanges, called interaction protocols. Usually, only a general description of the behavior required of agents partaking in these interactions is provided. Such protocols are called generic protocols. The description of generic protocols, especially with respect to their correct interpretation is a critical issue in open and heterogeneous multi-agent systems (MAS). A subsequent issue is the need to decouple interaction concerns from the other components of an agent, whatever architecture is adopted for that agent.

To date, there has been some endeavor to develop new protocol specification formalisms. The formalisms developed thus far have several drawbacks. They usually focus on data exchange through a communication channel (Promela/SPIN [10]). Some

M. Baldoni and U. Endriss (Eds.): DALT 2006, LNAI 4327, pp. 207–224, 2006.

others are either informal (or semi-formal) (e.g., AUML [2]) or demand advanced knowledge in logics (e.g., the formal notations defined by Paurobally et al [13], Alberti et al [1] and Giordano et al [8]). 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 fix the separation issue between interaction aspects and the other elements of the architecture adopted for an agent. We address this need in this paper.

The solution we developed is a framework to specify generic protocols. It conforms to the principles established for conversation policies by Greaves et al [9]. Our protocol notation is based on Agent-UML (AUML), a popular agent interaction representation formalism. But, we address (in our framework) the lacks and incompleteness which limit AUML. As commonly witnessed in several protocol representation formalisms, AUML only stresses the sequence of message exchanges. However, some actions are needed to produce these messages and handle them when received. Even, as we will see later, some actions which neither send messages nor handle them, might be executed 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 to describe the behavior agents will exhibit while playing a role in a protocol. A particular aspect in our framework is our focus on generic protocols, which keeps us from providing a complete representation for actions. Hence, we introduced action categories to fix this weakness.

Our framework offers several advantages. It builds on the graphical representation of protocol diagrams in AUML, which offers the (human) designers a better message exchange perception. In addition, it offers the means to depict what happens beyond the message exchange layer, in the course of an interaction. 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 constructs (sequence, loop and other control flows). Rather, we efficiently exploit event description to cover all these possibilities. Also, protocols in our framework are easily implemented following a XML format. As a concrete application, we used our framework to publish the specifications of generic protocols agent interactions are based on in several MAS applications we developed. Moreover, we used this framework to address two issues in agent interaction design for 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 tasks in collaboration.

The remainder of this paper is organized 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. We discuss some of them in this section.

AUML [2] and its extensions are graphical frameworks for protocol diagram representation. These frameworks, though practical and easy to use, do not emphasize the representation of actions performed in the context of an interaction. It is then hard to reason about the behavior agents, playing a role in a protocol, should be required of beyond the message exchange layer. As well, the graphical representation is useful only for human designers; it remains unreadable for computers. Casella and Mascardi [3] addressed this limitation by automating the translation process from AUML to a textual description, which is more machine readable. Winikoff [18] puts this textual representation of AUML protocol diagrams a step further. The work proposed a textual notation which defines a syntax for AUML protocol diagram specifications. The notation is accompanied by a tool that helps view the graphical representation corresponding to a textual specification. The advantage of relating a textual notation to AUML (whether automatically or not), though undebatable, is weakened by many other AUML's original limitations, f.i., the lack of emphasis on the description of (generic) actions in protocol representation, and the ambiguity about the formal semantics for protocols as well.

Some formal frameworks have been proposed for protocol representation. For example, Walton [17] defined a framework using concepts similar to ours. However, this framework directly introduces the notion of agent in protocol representation. This does not help separate the interaction concerns from the other parts of the architecture of an agent. In our opinion, this association between agents and roles should result from a configuration and instantiation process of protocols. Paurobally et al [13] 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 strongly believe that it should be hidden at the specification stage, as usually done in programming languages. Additionally, PDL-BI focuses on message exchanges. But, as we showed above, agent interaction protocols demand more than message exchange. Alberti et al [1] and Giordano et al [8] also developed formal protocol representation notations based on temporal logic. These formalisms are too theoretical, and thus cannot gain wide adoption in the area of protocol representation. Also, they suit commitment protocols, which aim at describing the social states the agents share during an interaction, instead of their mental states. The main difference between these two formalisms and ours is the different (representation and) interpretation of actions and messages.

IOM/T [5] is another recent language for agent interaction representation. Our work, though sharing some similarities with IOM/T, departs from it in the following points. Firstly, we focus on generic protocols, where we consider generic actions. Secondly, the behavior of agents in IOM/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 protocol representation. Our framework proposes an expressive declarative language which offers 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. From AUML, we identified five fundamental concepts: *protocol*, *role*, *event*, *action* and *phase*. A graphical illustration of these concepts is given in Fig. 1.

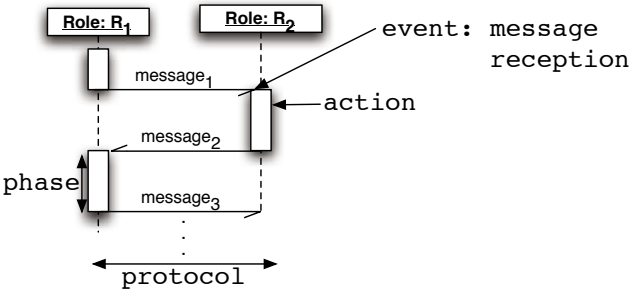

Fig. 1. Graphical illustration of concepts in generic protocols

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 [7], KQML [12], the commitment-based ACL introduced in [4].*

More formally, a protocol consists of a collection of roles $\mathbf{R}$*, which interact with one another through message exchanges. The messages belong to a collection* $\mathbf{M}$ *and the exchange takes place following a sequence,* $\mathbf{\Omega}$*. A protocol can also have some intrinsic properties* $\mathbf{\Theta}$ *(attributes and keywords) which are propositional contents (actually predicates) that provide a context for a further interpretation of the protocol. We note* $\mathbf{p} \stackrel{def}{=} < \mathbf{\Theta}, \mathbf{R}, \mathbf{M}, \mathbf{\Omega} >$.

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}".

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. are not*

thoroughly specified. A complete description of these actions depends on the architecture of each agent playing a role in the protocol.

Each of the communicating entities is called a role. Roles are understood as standardized patterns of behavior required of all agents playing a part in a given functional relationship in the context of an organization [6].

Definition 3. (Role) *In our framework, a role consists of a collection of phases. As we will see later (Section 3.2), a role may also have global actions (which are not bound to any phase) and some data other than message content,* variables.

$\forall \mathbf{r} \in \mathbf{R}, \mathbf{r} \stackrel{\text{def}}{=} < \mathbf{\Theta_r}, \mathbf{\Pi}, \mathbf{A_g}, \mathbf{V} >$, where $\mathbf{\Theta_r}$ corresponds to the role's intrinsic properties (e.g., cardinality) which are propositional contents that help further interpret the role, $\mathbf{\Pi}$ the set of phases, $\mathbf{A_g}$ the set of global actions and $\mathbf{V}$ the set of variables. In our framework, roles can be of two types: (1) *initiator*, the unique role of the protocol in charge of starting[1] its execution; (2) *participant*, any role partaking in an interaction based on the protocol.

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

Table 1. Event Types

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).

Definition 4. (Action) *An action is an operation a role performs during its execution. 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[2] description for the actions which are executed in these protocols. Hence, we introduced action categories to define the semantics of these actions. Table 2 contains an informal description of these categories. We discuss their semantics in Section 3.3.

[1] Starting a protocol demands more than sending its initial message.

[2] The term general here is used in the sense of describing the skeleton of these actions.

Table 2. Action Categories

Action Category	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.

Definition 5. (Phase) *Successive actions sharing direct links can be grouped together. Each group is called a phase. Two actions a_i and a_j share a direct link, if the input arguments (or only a part of the input) of a_j are generated by (the output result of) a_i.*

3.2 Formal Specifications

The formal specifications are defined through an 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 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) [16] to illustrate our specification formalism. The sequence diagram (protocol diagram in AUML) of this protocol is given in Fig. 2. Note that the labels placed on the message exchange arrows in the figure are not performatives, but message identifiers.

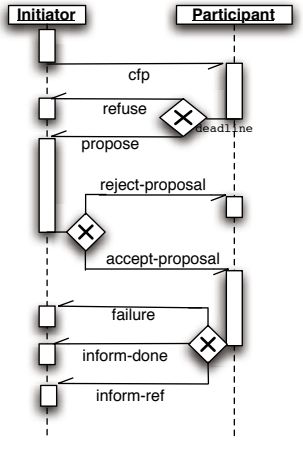

Fig. 2. The Contract Net Protocol

The rationale of CNP consists in 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 to performing.

Protocol. The following production rules define a protocol. In As one can see from these rules, the exchange sequence $\mathbf{\Omega}$ contrary to Definition 1, is not explicitly specified. Actually, it is located in the definition of roles, and precisely in the *send* actions of these roles.

$$
\begin{aligned}
< protocol > &:= < protproperties >< roles >< messagepatterns > \\
< protproperties > &:= < protdesc >< protattributes >< protkeywords? > \\
< protdesc > &:= < identifier >< title >< location > \\
< protattributes > &:= < class >< participantcount > \\
< protkeywords > &:= < protkeyword+ > \\
< protkeyword > &:= \text{“}IncrementalResolution''| \ldots
\end{aligned}
$$

Example 1, we exemplified the use of these rules to specify CNP.

```
(protocol
 (protproperties
  (protdesc :ident cnpprot :title ContractNet :location Cnp.xml)
  (protocolattributes :class Request :participantcount 1)
  (protkeywords ''containsMultipleInstanceRole''))
 (roles ...)
 (messagepatterns ...))
```

Example 1. Specifying CNP

The properties of a protocol consist of descriptors (identifier, title and location), keywords and attributes, which we identified from the experiments we carried out with our framework. The keywords are propositional contents which help further characterize the protocols. Currently, we consider the following keywords: (1) *ContainsMultipleInstanceRole* which means that there can be several instances of a participant role in this protocol; (2) *ContainsIterativeProcess*, which means that a sequence of actions can be repeatedly executed in the protocol; (3) *IncrementalResolution*, which means that an anytime algorithm can lie behind the execution of the protocol; (4) *ContainsDividableProcessing* which means that the processing associated with this protocol can be divided for several participants; (5) *SubscriptionRequired*, which means that the processing associated with this protocol requires a prior subscription; (6) *AlterableCommitment* which means that the commitments are not indefeasible.

Concerning the attributes, they are functions which we use to refine the description of a protocol. The current version of our framework allows two attributes to be set: (1) *class*, which indicates what kind of processing is implied by the protocol, (2) *participantcount*, which indicates how many types of participant roles does the protocol contain.

Role. Protocol diagrams only show the communication flow between roles. However, there may be some information beyond the communication layer. For example, in CNP, the action an initiator executes, in order to make a decision upon the bids the participants issued, 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. 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 not tied up with any phase. The production rules hereafter define a role.

$$
\begin{aligned}
< roles > &:= < role >< role > \mid < roles >< role > \\
< role > &:= < roleprop >< variables? >< actions? >< phases > \\
< roleprop > &:= < roledesc >< roleattributes >< rolekeywords? > \\
< roledesc > &:= < identifier >< name > \\
< roleattributes > &:= < cardinality > \\
< variables > &:= < variable+ > \\
< variable > &:= < ident >< type > \\
< actions > &:= < action+ >
\end{aligned}
$$

Each role is described through its intrinsic properties (f.i., cardinality), its variables, its global actions and phases. From Example 2, the *initiator* role of CNP has three variables: `deadline`, `bidsCol` and `deliberations`. `deadline` informs of the moment when bidding should stop. `bidsCol` is a collection where bids issued by participants are stored. `deliberations` contains the decision (accept or reject) the initiator made upon each bid. Each variable has an identifier and the type of data it contains. The content of a variable is characterized 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 as local (located in a phase) ones: category (see Table 3.1), signature (input and output data types) and events (input and output). Note that each part (input and output) of the signature as well as the events is composite. We introduce three types of connector (*and*, *or*, *xor*, with their usual meanings) to assemble the elements of these parts. `Deliberate` is a *compute* action. It takes a *date* and a *collection* as input arguments (*:dir in*) and a *Map* as output result. `Deliberate` is executed when the value of `deadline` changes (*change* event) and

that at least one bid has been stored in `bidsCol`. Once executed it changes the value of `deliberations`. The reserved 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 special word sometimes helps define causality between actions.

```
(role :ident initiator
 (roleprop (roledesc :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 ...))
```

Example 2. Specifying the initiator role of CNP

Phase. As stated above, each phase is a group of actions that share 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 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. The description of this phase is given in Example 3.

```
(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)))))
```

Example 3. Specifying the first phase of the initiator role of CNP

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 that only one ACL be used all along a single protocol description. The following rules define message patterns. Example 4 describes the message patterns used in CNP.

$$
\begin{aligned}
< messagepatterns > &:= < acl >< messagepattern+ > \\
< acl > &:= 'fipa' | 'kqml' \\
< messagepattern > &:= < performative >< identifier >< content > \\
< content > &:= < type >< pattern? >
\end{aligned}
$$

```
(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 ...)) ...)
```

Example 4. Specifying message patterns in CNP

Design Guideline. As a guideline for protocol design and specifications in our framework, we recommend several design rules. Following these rules ensures that the resulting protocol specifications are wellformed and correct (ambiguity and inconsistency-proof). In the future, we envision to devise some algorithms (and a tool) which automate the process of checking whether a protocol specification complies with our guidelines. We introduce these guidelines 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. *For every message m_i of the message set* $\mathbf{M}$ *of a protocol, there is at least one send action of a role in the protocol, which sends m_i.*

Proposition 3. *When two distinct actions can be executed at a point in a role definition, the set of events which fire each action, though intersect-able, should be distinguishable.*

Proposition 4. *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. This change 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 role coordination. In this section, we briefly discuss the semantics of some events used in our framework. When needed in the definition of the semantics of our concepts, we introduce some expressions in a meta-language, which we call primitives.

change: this event type notifies of a change of the variable's value. Let v be a variable, *change(v)* denotes the event. We introduce the *value* primitive, which returns the value of a data at a given time point. Let d and t be a data and a time point respectively, $Value(d, t)$ denotes this function. Value$(d, t) = \emptyset$ means that the data d does not exist yet at time point t. We interpret the *change* event as follows:

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

endprotocol: this event type notifies of the end of the current interaction. The phases in each role, 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. We introduced three new primitives: *Follow*, *Executed* and *Unreachable*. *Follow* is a function which returns all the immediate successors of a phase. Let π_1 and π_2 be two phases, π_2 immediately follows π_1, if any of the input events of the initial action of π_2 refers to a prior event generated by one of the actions (usually the last one) of π_1. *Unreachable* is a predicate which means that the required conditions for the execution of an action do not hold, therefore preventing this action from being executed. Finally, *Executed* is a predicate which means that an action has already been executed. Let Π be the set of phases for a role r and A_π the set of executable actions for a phase π in r. Let also A_{G_r} be the set of global actions of r. We interpret the *endprotocol* event as follows:

$$\forall r \in \mathcal{R}, \forall a_\alpha \in A_{G_r}, (\text{Unreachable}(a_\alpha) \vee \text{Executed}(a_\alpha)) \wedge (\forall \pi \in \Pi, (\text{Follow}(\pi) = \emptyset) \vee (\forall a_i \in A_\pi, \text{Unreachable}(a_i)))$$

reception: this event type notifies of the reception of a new message. Let m' denote the received message, we interpret this event as follows (notation being *reception(m')*):

$$\exists\, t_1, t_2 (\mathrm{t_1} < \mathrm{t_2}) \wedge (\mathrm{m'} \notin_{\mathrm{t1}} \mathfrak{M}') \wedge (\mathrm{m'} \in_{\mathrm{t2}} \mathfrak{M}')$$

The symbol $\in_t$ (resp. $\notin_t$) means *belongs* (resp. *does not belong*) at time point t. $\mathfrak{M}'$ is the set of messages an agent received during an interaction.

variablecontent: this event notifies of the (fresh) construction of the content of a variable. Let v be a variable, *variablecontent(v)* denotes the event, which we interpret as follows:

$$\exists\, t_1, t_2(\mathrm{t}_1 < \mathrm{t}_2) \wedge (\mathrm{Value}(\mathrm{v}, \mathrm{t}_1) = \emptyset) \wedge (\mathrm{Value}(\mathrm{v}, \mathrm{t}_2) \neq \emptyset)$$

Action. Actions are executed when events occur. And once executed, they may generate new events. Events are therefore considered as *Pre* and *Post* conditions for actions' execution. Here again, we only discuss the semantics of some action categories: *append*, *set*, *compute* and *send*. Let $\mathbf{E}$ be the set of all the event types we consider in our framework and $\mathbf{E}' = \mathbf{E} - \{endphase, endprotocol\}$.

append: this action adds a data to a collection. Let a_i be such an action. In the following we introduce two primitives: *isElement()* and *Arguments()*. *isElement()* is a predicate which returns true when a data belongs to a collection at a given time point. *Arguments()* returns the input arguments of an action.

$$\begin{aligned}&\mathrm{Pre} = \{e_j, e_j \in \mathbf{E}'\}\\ &\mathrm{Post} = \{e_j, \exists k\ e_k = change\ \wedge (\exists t_1, t_2, d, v \in \mathrm{Arguments}(a_i), (t_1 < t_2) \wedge\\ &(\mathrm{isElement}(v, d, t_1) = false) \wedge (\mathrm{isElement}(v, d, t_2) = true))\}\end{aligned}$$

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:

$$\begin{aligned}&\mathrm{Pre} = \{e_j, \forall m_j \in \mathrm{Arguments}(a_i), \exists\, k, e_k = \mathrm{messagecontent}(m_j)\}\\ &\mathrm{Post} = \{Trans(m_j) = true\}\end{aligned}$$

at the receiver side:

$$\begin{aligned}&\mathrm{Pre} = \emptyset\\ &\mathrm{Post} = \{e_j, \forall m_j \in \mathrm{Arguments}(a_i), \exists!\, k, e_k = \mathrm{reception}(m_j)\}\end{aligned}$$

set: this action sets the value of a data. Let a_i be such an action,

$$\begin{aligned}&\mathrm{Pre} = \{e_j, e_j \in \mathbf{E}'\}\\ &\mathrm{Post} = \{e_j, \forall v_j \in \mathrm{Arguments}(a_i), \exists!\, e_j, e_j = \mathrm{variablecontent}(v_j)\}\end{aligned}$$

compute: this action computes some information. Let a_i be such an action,

$$\begin{aligned}&\mathrm{Pre} = \{e_j, e_j \in \mathbf{E}'\}\\ &\mathrm{Post} = \{e_j, e_j \in \mathbf{E}' - \{reception\}\}\end{aligned}$$

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 content of the message. In the future, we wish to extend the interpretation of this predicate and enable agents to share some social states. Thus, our specification formalism could cover the commitment protocols.

Moreover, when an action ends up a phase or the whole protocol, its Post condition is extended with *endphase* and *endprotocol*, 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 $\pi \stackrel{\text{def}}{=} < \mathbf{A}_\pi, \prec >$, where $\mathbf{A}_\pi$ is a set of actions and $\prec$ a causality relation which we define as follows:

$$\forall \mathbf{a}_i, \mathbf{a}_j \in \mathbf{A}_\pi, \mathbf{a}_i \prec \mathbf{a}_j \Longleftrightarrow \exists e \in Post(\mathbf{a}_i), e \in Pre(\mathbf{a}_j).$$

Proposition 5. *Let* $\mathbf{a}_i$ *and* $\mathbf{a}_j$ *be actions of a phase* π*, such that* $\mathbf{a}_i$ *always precedes* $\mathbf{a}_j$*,*

$$(\mathbf{a}_i \prec \mathbf{a}_j) \vee (\exists \mathbf{a_p}, \ldots, \mathbf{a_k}, \mathbf{a}_i \prec \mathbf{a_p} \ldots \prec \mathbf{a_k} \prec \mathbf{a}_j)$$

Role. The causality relation between actions of phases can be extended to interpret roles. Indeed, an event generated at the end of a phase can be referred to in other phases. On this basis, we defined an operational semantics for roles. The inference rules behind this semantics cover sequences, loops, alternatives, etc. In these inference rules, defined as usually, the statements are replaced by actions. We do not discuss these rules in this paper due to space constraint. Thanks to the operational semantics, we interpret a role is a labeled transition system with some intrinsic properties. $\mathbf{r} =< \mathbf{\Theta_r}, \mathbf{S}, \mathbf{\Lambda}, \longrightarrow >$ 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), and $\longrightarrow \subseteq \mathbf{S} \times \mathbf{\Lambda} \times \mathbf{S}$ is a transition function. As an illustration, we give the semantics of the initiator role of CNP, which we call $\mathbf{r_0}$. Note that the messages associated with the send actions are numbered following their position in Fig. 2. $\mathbf{r_0} =< \mathbf{\Theta_{r_0}}, \mathbf{S}, \mathbf{\Lambda}, \longrightarrow > \mathbf{\Theta_{r_0}} = \{"cardinality = 1" \wedge "isInitiator = true"\}$ $\mathbf{S} = \{\mathbf{S_0}, \mathbf{S_1}, \mathbf{S_2}, \mathbf{S_3}, \mathbf{S_4}, \mathbf{S_5}, \mathbf{S_6}, \mathbf{S_7}, \mathbf{S_8}, \mathbf{S_9}, \mathbf{S_{10}}, \mathbf{S_{11}}, \mathbf{S_{12}}\}$ $\mathbf{\Lambda} = \{\mathbf{a_0}, \mathbf{a_1}, \mathbf{a_2}, \mathbf{a_3}, \mathbf{a_4}, \mathbf{a_5}, \mathbf{a_6}, \mathbf{a_7}, \mathbf{a_{11}}\}$ $\longrightarrow = \{(\mathbf{S_0}, \mathbf{a_0}, \mathbf{S_1}), (\mathbf{S_1}, \mathbf{send}_{[\mathbf{m_0}]}, \mathbf{S_2}), (\mathbf{S_2}, \mathbf{a_1}, \mathbf{S_7}), (\mathbf{S_2}, \mathbf{a_2}, \mathbf{S_3}), (\mathbf{S_3}, \mathbf{a_4}, \mathbf{S_4}), (\mathbf{S_4}, \mathbf{a_3}, \mathbf{S_5}), (\mathbf{S_5}, \mathbf{send}_{[\mathbf{m_3}]}, \mathbf{S_{11}}), (\mathbf{S_5}, \mathbf{send}_{[\mathbf{m_4}]}, \mathbf{S_6}), (\mathbf{S_6}, \mathbf{a_5}, \mathbf{S_8}), (\mathbf{S_6}, \mathbf{a_6}, \mathbf{S_9}), (\mathbf{S_6}, \mathbf{a_7}, \mathbf{S_{10}}), (\mathbf{S_2}, \mathbf{a_{11}}, \mathbf{S_{12}})\}$.

Protocol. The semantics of a protocol is that of a collection of interacting graphs (the roles) which coordinate their execution following the message exchange sequence Ω. This collection also has some intrinsic properties, some propositional contents which are true in the environment of the MAS. Nevertheless, some limitations subsist in this

way of interpreting generic protocols. Usually *a priori* semantics is proposed for protocols. However, *a priori* semantics is not sufficient to interpret a generic protocol. Two main reasons account for such an insufficiency. Firstly, the message exchange sequence can be mapped to a graph of possibilities in the regard of exchanged messages. Therefore, the semantics of an interaction based on this protocol corresponds to 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 semantics of the executed actions should be included too. However, except send actions, all the other actions can only have general interpretation before the execution of the interaction, or its configuration for an agent. A more precise semantics of these actions can only be known at runtime (or sometimes the design time for agents). To this end, we introduce *a posteriori* semantics for protocols in our framework. Particularly, we draw on *Protocol Operational Semantics (POS)* developed by Koning and Oudeyer [11]. In our framework, this additional interpretation feature consists in refining the path followed in each graph corresponding to the roles involved in the interaction. Also, the semantics of the actions is enriched by that of the methods executed in place.

As an illustration, let us assume that the semantics of each role of 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 $\mathbf{M} = \{\mathbf{m_0}, \mathbf{m_1}, \ldots \mathbf{m_8}\}$. $\mathbf{\Omega} =< r_0 \xrightarrow[a_0]{m_0} r_1, r_1 \xrightarrow[a_7, m_0]{m_1|m_2|m_8} r_0, r_0 \xrightarrow[a_3, m_2]{m_3|m_4} r_1, r_1 \xrightarrow[a_{10}, m_4]{m_5|m_6|m_7} r_0 >$.

4 Properties

When one designs a generic protocol, it is mandatory to formally prove its properties, in order to ease a wide adoption of this protocol. In this section we discuss some general properties for protocols designed in our framework. Here, we focus on two properties, liveness and safety, related to the correctness of protocols specified following our framework. Finally, we discuss the termination property, which one of the critical ones, for generic protocols. Note that we assume that the design guidelines discussed in Section 3.2 are respected. There are other properties specific to generic protocols, *equivalence*, *compliance*, *similarity*, which we do not discuss in this paper.

4.1 Liveness

Definition 6. *Liveness A role of a protocol is alive when it still has a sequence of actions to perform before reaching its terminal state. As a consequence, a protocol is alive when at least one of its roles is still alive.*

Proposition 6. *For every role of a protocol, events will always occur and fire some transition until the concerned role enters a terminal state.*

Proof. We prove this property only on a design standpoint, i.e. we do not assume anything about what actually happens in the MAS at runtime. Each role is considered a transition system; and from the description of transition systems, unless a faulty situation is encountered, 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

Definition 7. (Safety) *A safe protocol is one where nothing inconsistent happens during its execution. Particularly, we focus on two aspects of safety: (1)* consistent message exchange*, which means that each sent message is received and handled by at least one role; (2)* Unambiguous execution*, which explicitly requires some clear conditions to hold every time a role has to take an action.*

Proposition 7. (Consistent message exchange) *The message exchange sequence of a protocol, designed in our framework and which respects our design guidelines, is consistent. 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. From Proposition 2, each message in $\mathbf{M}$ is sent at least by one *send* action. On the other hand, every *reception* event in any role is related to a received message which belongs to $\mathbf{M}$ too. Thus, every received message has been generated and automatically sent (see Proposition 4) by a *send* action, supposedly of a different role.

Proposition 8. (Unambiguous protocol execution) *For each action a role can take, there is an unambiguous set of events which fire its execution.*

Proof. Let a_i be an action of a role r and E_{a_i} be the set of events which fire the execution of a_i. If a_i is the unique action that can be performed at the current execution point of r, the proposition is straightforward. Let's now assume that there exists another action a_j which can be executed at the same point as a_i. If $E_{a_i} \cap E_{a_j} = \emptyset$, then the proposition is also straightforward. In the case where $E_{a_i} \cap E_{a_j} \neq \emptyset$, from Proposition 3 we know that $E_{a_i} \neq E_{a_j}$. Thus, for a_i to be performed events in $E_{a_i} - E_{a_j}$ should occur. And $E_{a_i} - E_{a_j}$ is a unambiguous subset of E_{a_i}.

4.3 Termination

Proposition 9. (Termination) *Each role of a protocol represented in our framework always terminates.*

Proof. From Proposition 1, each role has a sequence of actions which bring 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. However, 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 for the specifications of 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 the behavior required of them during the execution of protocols. In this, we depart from the usual protocol representation formalisms which only focus on the description of exchanged messages. 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 various 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 that will perform them in the context of an interaction. This is usually done by hand by agent designers when they have to set up 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 an automatic generic protocol configuration mechanism (see [15]). This mechanism consist in looking for similarities between the functionalities in the architecture of an agent and actions of generic protocols.

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. However, this static protocol selection severely limits interaction execution in open and heterogeneous MAS. Thus, we developed a dynamic protocol selection mechanism (see [14]) to address these limitations. During the dynamic protocol selection, agents reason about the specifications of the protocols known to them and the specification of the task to perform. Again, we used this framework, since it enables us to accomplish the reasoning about the mandatory coordination mechanisms for the performance of collaborative tasks.

References

1. M. Alberti, D. Daolio, and P. Torroni. Specification and Verification of Agent Interaction Protocols in a Logic-based System. In *ACM Symposium on Applied Computing (SAC)*, pages 72–78. ACM Press, 2004.
2. 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.
3. G. Casella and V. Mascardi. From AUML to WS-BPEL. Technical report, Computer Science Department, University of Genova, Italy, 2001.
4. M. Colombetti, N. Fornara, and M. Verdicchio. A Social Approach to Communication in Multiagent Systems. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies (DALT)*, number LNCS 2990, pages 191–220, Australia, Melbourne, 2003. Springer.
5. T. Doi, Y. Tahara, and S. Honiden. IOM/T: An Interaction Description Language for Multiagent Systems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 778–785, 2005.
6. 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.

7. FIPA. FIPA Communicative Act Library Specification. Technical report, Foundation for Intelligent Physical Agents, 2001.
8. L. Giordano, A. Martelli, and C. Schwind. Specifications and Verification of Interaction Protocols in a Temporal Action Logic. In *Journal of Applied Logic (Special Issue on Logic-based Agent Verification)*, 2005.
9. M. Greaves, H. Holmback, and J. Bradshaw. What is a Conversation Policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents 1999*, 1999.
10. G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.
11. 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.
12. Y. Labrou and T. Finin. A proposal for a new KQML specification. Technical report, University of Maryland Baltimore County (UMBC), 1997.
13. 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.
14. 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.
15. 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.
16. 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.
17. C. Walton. Multi-agent Dialogue Protocols. In *Proceedings of the Eight Int. Symposium on Artificial Intelligence and Mathematics*, 2004.
18. 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

$$
\begin{aligned}
< protocol > &:= < protproperties >< roles >< messagepatterns > \\
< protdescriptors > &:= < protdescriptors >< protattributes >< protkeywords? > \\
< protattributes > &:= < class >< participantcount > \\
< protocolkeyword > &:= "containsconcurrentroles" | "iterativeprocess" | \ldots \\
< roles > &:= < role >< role > | < roles >< role > \\
< messagepatterns > &:= < acl >< messagepattern+ > \\
< role > &:= < roleproperties >< variables? >< actions? >< phases > \\
< roledescriptors > &:= < roledescriptors >< roleattributes >< rolekeywords? > \\
< roleattributes > &:= < cardinality >< concurrentparticipants? > \\
< rolekeywords > &:= < rolekeyword+ > \\
< cardinality > &:= < digit+ > | "n" \\
< variables > &:= < variable+ > \\
< variable > &:= < identifier >< type > \\
< type > &:= "number" | "string" | "char" | "boolean" | \ldots \\
< actions > &:= < action+ > \\
< phases > &:= < phase+ > \\
< phase > &:= < identifier >< actions > \\
< action > &:= < category >< description? >< signature? >< events > \\
< category > &:= "append" | "custom" | "remove" | "send" | "set" | "update" \\
< signature > &:= < arguments > | < messages > \\
< arguments > &:= (< argset > | < argdesc >) + \\
< argset > &:= < settype > (< argset > | < argdesc >) + \\
< argdesc > &:= < identifier >< type >< direction > \\
< 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" | "endphase" | \ldots \\
< object > &:= < message > | < variableid > \\
< eventref > &:= < identifier > \\
< messagepattern > &:= < identifier >< performative >< content > \\
< performative > &:= < fipaperformative > | < kqmlperformative > \\
< content > &:= < type >< pattern? >
\end{aligned}
$$