A Generic Formal Framework for Multi-agent Interaction Protocols

Bo Chen and **Samira Sadaoui**

Department of Computer Science, University of Regina

3737 Waskana Parkway, Regina SK, Canada, S4S 0A2

{sadaouis,chen112b}@cs.uregina.ca

Contents

1 Introduction

Interaction is the main and inherent characteristic of multi-agent systems (MAS) [37]. The flexible and organizational interactions among agents make MAS different from other kind of systems, even single agent systems. Indeed, the overall capacity of a MAS can exceed the sum of individual agent capacities [28]. This is why multi-agent technologies are widely expected to cope with difficulties in developing heterogeneous and distributed open systems [20]. However, A MAS is complex to design and construct, due to the undeterministic and autonomous behaviors of agents. Especially, agents can interact in flexible and sophisticated ways to achieve system goals. To let agents successfully work together, there should be some constraints and rules to guide the cooperative, coordinative or even competitive behaviors of agents. In correspondence, agent interaction protocols (AIP) are used to define these constraints and rules [25, 23].

AIP impose particular constraints on agent messages to manage agent communication and negotiation. The constraints specify the set of allowed message types, message contents and the correct order of messages during the conversations between agents. AIP can enforce agents to act correctly in predictable ways. Thus, the properties of a MAS can be preserved. For example, an auction protocol ensures that auctioneers are fair, i.e. give equal opportunities for all buyers to win an item. In addition, AIP can be viewed as reusable software components to design agent interaction in applications [10], because AIP represent abstract and formal patterns of agent interaction. Since AIP play an important and essential role in MAS development [37], formal specification and verification of AIP are needed [24].

The protocol engineering issue remains a challenge for MAS research [24]. On the first hand, interaction systems are complex to verify, validate and reuse [17]. Especially, the concurrency, reactivity, autonomy, openness, and extensibility of MAS bring new challenges. On the other hand, there is a wide gap between theory and practice. Indeed, the advances in theories and software architecture must be complemented by the advances in engineering techniques and methodologies [29]. In the field of multi-agent systems, there are few tools supporting the design of interaction protocols [22]. In fact, even though several theories and methodologies have already been proposed for developing AIP. However, existing methodologies of AIP suffer from some limitations, such as poor capacity for designing complex protocols, being limited when facing the concurrency of interactions [24], lack of suitable tools. One solution is to resort to the well-established and well-proved formal technologies for concurrent and distributed systems [24, 38].

This paper aims to provide a generic formal computational framework to develop AIP, and to verify their "correct behaviors" using model checking. This formal framework is based on the formal specification language Lotos [5], which has been widely applied in distributed and concurrent systems. In addition, several rigorous tools and sound methodologies have already been developed for Lotos. Based on Lotos technologies, the proposed reusable framework brings several advantages, including:

- Taking charge of concurrency and synchronization naturally because Lotos was originally designed for concurrent systems.
- Defining the generic architecture of AIP, i.e. the basic components and their connectors in interactions. This architecture, which expresses many fundamental and essential characteristics of agent interaction, can be reused to develop different protocols.
- Providing a suitable structure for formalizing agent interaction, which is important for building correct IP specifications [31, 17].
- Providing the important correctness properties(safety, liveness and fairness) of AIP, that any AIP application should satisfy.
- Using several tools to formally analyze, animate and verify IP specifications expressed in Lotos. Therefore, the correctness of AIP can be proved.

item Improving the understanding of dynamic behaviors of AIP through simulation.

An stepwise approach is also proposed to incrementally develop interaction protocols using the methodology developed in [6]. Following this methodology, the overall complexities are broken into serial sub-steps. Each step evaluates and takes a small amount of decisions in isolation, and a more refined version of the design is elaborated [27]. The specifications can be refined stepwise toward implementation in this way.

This paper is organized as follows: In Section 2, we analyze the architecture and characteristics of multi-agent interaction. In Section 3, we provide a generic framework for specifying agent protocols. In Section 5, we study what are the most important properties to be verified on AIP. The generic framework is also proved correct. In Section 5, we demonstrate how to develop an English auction protocol hat is correct from the generic framework.

1.1 Agent Interaction Protocols

A multi-agent system can be viewed as an artificial virtual society, in which agents can cooperate or coordinate to perform tasks. The agent interaction requires a set of agreed messages, rules for actions based upon reception of various messages, and assumptions of the communication channels [35]. These constraints ,rules and patterns can be abstracted and formalized as AIP, which are basis for agent negotiation and cooperation. Using the protocols, the autonomous behaviors of agents can be somehow predictable, even though agents are anonymous, because agents are obligated to obey some interactive rules. The interaction protocols range from negotiation schemas to a simple request for a task.

AIP are patterns representing both message communication and the corresponding constraints on the content of such messages. They describe an allowed sequence of messages and message content among agents [26]. AIP can also be considered as reusable software design patterns describing problems frequently occurring in multi-agent systems [3]. The roles and message details in an AIP can be modified to adapt to different scenarios to solve the problems [10]. Protocol definition consists of a set of attributes given below [36, 10]:

- Purpose. It is the goal that an interaction is supposed to achieve.
- Messages. A message consists of a sender, a set of receivers, type of messages and message content. The involved agents are expected to understand the message elements defined in the protocol.
- Ontology. It gives meaning to the symbols in the content expression.
- Rules. Those rules define the dependencies and relationships between messages as well as the constraints on the message structure, such as the types of messages agents can receive and send in a particular situation.
- Input and output. They define the message types that can be sent and received by a participant.
- States and transitions. An AIP is usually modelled as a Finite Transition Systems. The state of the system changes according to the actions performed by participants.
- Participants. They include the initiator and responder roles in the conversation.
- Reasons. For example, a participant should give reasons to other agents why it accepts or rejects a request.

- Views. AIP can be examined by different views from different participants, i.e. from a local role's view or from an external observer's view.

The messages are expressed in agent communication languages, such as FIPA ACL [13] and KQML [1]. Messages express the intention of an agent expecting what other agents are expected to perform. Agent languages are based on the speech acts theory, which is a model for human communication [28]. Performative act is a key concept in the theory. It is an action that a speaker performs (speaks) in order to convey part of its mental state to the hearer that the act is directed to [19].

Role is a set of agents satisfying distinguished properties, interfaces, service description and behaviors. An agent role is an abstract description of an entity with the specified functions [8]. When we compare the role of agents to the interface of objects, we can find many similarities. An interface can be implemented by different classes and a class can provide multiple interfaces while a role can be played by different agents and an agent can act in several roles. For example, an agent can be a seller and buyer in two different auction conversations undertaking concurrently.

For example, the FIPA Request Interaction Protocol [13] has a purpose that allows one agent to request another to perform some actions. There are only one initiator and one responder. The output of the initiator contains two kinds of messages: *Request and Cancel*, the output of responder has six types of messages: *Not-Understood, Refuse, Agree, Failure, Inform-Done and Inform-Result*. The behaviors of the protocol can be described as follows: when an agent receives a *Request*, it may agree or refuse the request, and it can also issue a *Not-Understood*. If it agrees, it will tell the initiator the result of its actions according to the request. The result may be *failure, Inform-Done, and Inform-Result*. During the interaction, the initiator can cancel the interaction anytime after the interaction starts.

Agent interaction is more complex than object interaction in distributed systems because agents are autonomous and interactive [37]. Unlike objects need outside control to execute their methods, agents have control over whether and how they process external requests, using their knowledge of the environment and the effect of their actions. The challenge grows when the agent has to interact with different parties, using different protocols.

Many methodologies have been developed to help capture, represent, specify, validate and verify AIP, such as Extended UML [23], COOL [2], Extended Finite State Machines

[23] and Petri Nets [8]. Protocols are modelled using several theories, such as *expection* and *commitment* [14]. However, these theories and methdologies bear some limitations summarized in [24], as mentioned in introduction part.

1.2 Multi-agent System Architectures

The knowledge of MAS architecture is necessary to develop a suitable AIP specification structure. MAS are usually constructed upon agent platforms. Therefore, MAS are similar in many aspects. A typical abstract MAS architecture is shown in figure 1, and it is based on the FIPA specification [1].

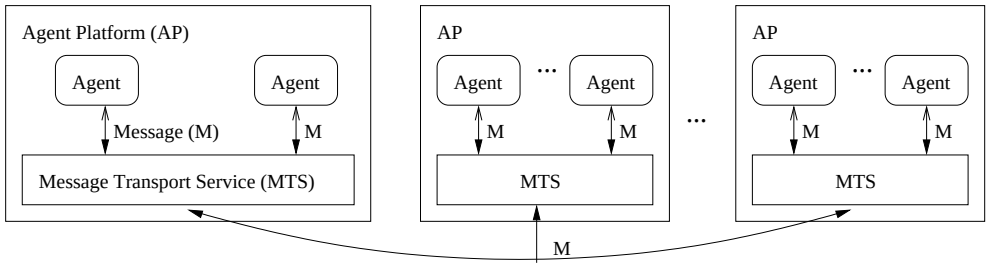

Figure 1: Agent Abstract Architecture

Each agent features an inner control responsible for changing its status and sending messages [32]. Agents reside in agent platforms which provide fundamental functionalities such as:

- Lifecycle management. It is responsible for *creating, activating, suspending, invoking, moving* and *destroying* agents. In its lifecycle, an agent may stay in different states, such as *unknown, active, waiting* and *suspending*.

- Message transport service (MTS). It plays a vital role in agent communication. MTS provides message addressing and transferring services. MTS also enforces the rules of message buffering with respect to different states of agents. When an agent is in a waiting state, MTS stores the messages in its message buffer for further processing when it becomes active. When an agent is in unknown state, MTS may discard the messages.

- Directory service. It is used to find agents or services provided by them. It is similar to the familiar white and yellow pages provided by tele-companies.

[1] http://www.fipa.org/repository/index.html

2 AIP Specification

The specification language Lotos is an ideal choice for the description, validation and verification of agent interaction protocols. For instance, we have applied Lotos to describe, simulate and verify a dynamic agent-based online auction protocol [7]. First, Lotos has been extensively studied and applied on concurrent and distributed systems, such as ISO communication protocols. Second, Lotos is executable, modular and capable of synchronization between processes [9]. Third, many tools have been developed for the validation and verification of Lotos specifications. In addition, many aspects of system functionality can be effectively represented in Lotos [12], such as the external and internal descriptions, representing respectively what the system does and how the functionalities are implemented. Various specification styles and development methodologies have also been proposed to make the language useful in different development stages from requirements capture to detailed design [33].

Our model of the interactive behaviors of an agent is based on the semantics of Lotos, defined by Labelled Transition System (LTS). A protocol is modelled as a set of communicating processes executing concurrently. They express the constraints on the relationship between *Send* and Recv gates, representing sending and receiving messages. These constraints represent the protocol mechanism. This model emphasizes the agent collaborative aspects, namely, its interactive behaviors. It follows the social approach to describe AIP [14, 32]. The advantage of this model is that agent interaction can be described even internal and mental structure of agents are unclear.

Lotos is suitable to design architecture [18]. We will use it to design a flexible framework for specifying agent interaction protocols. This framework will capture as many essential properties of MAS as possible while it tries to avoid the details irrelevant to AIP.

2.1 Architecture Design

The essential techniques for effectively handling complexity of any software systems are decomposition, abstraction and organization. Object-oriented methods decompose a system into objects, while agent-oriented methods abstract a system as a number of agents. The appropriate separation of concerns is vital to modularity and reuse. Interaction protocol specifications are decomposed into role modules and message transportation modules, which are modelled as Lotos processes. A grey-box approach [32] is applied to design

these modules. This means that we specify both the external and internal behaviors of agents. However, the internal aspects of an agent are only described as far as it affects the interaction. Internal deliberations are represented as undeterministic choices. This architecture results in a good separation of concerns.

The framework is designed to be flexible and scalable, separate the standard functionality form application-specific behaviors. We try to capture the essential structure of agent interactions and only necessary agent internal design. We keep the specifications in appropriate abstraction level that make verification viable. Therefore, we do not mix protocol specification with life-cycle management specification, which are internal elements, viewed from the observers.

The overall communication pattern description is accomplished by the parallel composition of the role modules and message transportation modules. This convenience is due to the synchronization mechanisms of Lotos. These modules synchronize on the message exchange gates to form the whole constraints on the message sequence. All the interaction scenarios can be produced automatically by simulating the specification. These scenarios can be represented in different formats, such as text, tree and MSC. Unlike UML that can only model a small number of possible scenarios, our approach can simulate dynamically almost all the possible scenarios allowed by a protocol.

Styles denote the way the designer expresses the functionality of system, making use of language elements. The individual modules are defined first using the state-oriented style. This style provides insight in the amount of state information to be maintained by a resource, and the complexity of manipulating this information. The internal insight helps transform the formal specification into a final implementation of the resource, because the style gives much hints to develop data and program structures of final implementation. Finally, all the individual modules are composed to obtain the overall behaviour of the system using resource-oriented style [34, 30, 11].

We also consider the limitations of available Lotos tools. That means, we do not use some features of Lotos language, such as recursive processes on parallel operators.

2.2 Message Structure

Messages are specified using Lotos data part, which can abstractly represent message content. The symbolic messages are independent of any agent language. We focus on

message semantics, without caring about its physical implementation details.

A message has three parts: a sender, a set of receivers, and a content. This means that one message can be sent to multiple agents at once. For instance, in the message *msg(sid, add(rid, nil), cnt(not-understood))*, the sender is *sid*, the receiver is *rid*, and the message content is *not-understood*. Because Lotos data type definition allows polymorphism, the operation *cnt* can encapsulate any numbers and kinds of information, such as *cnt(request)* and *cnt(inform-done, result)*.

2.3 Agent Role Modules

Each role in an interaction is modelled by a process, which encapsulates a set of conversation rules related to a role in an interaction protocol. An interaction protocol can be viewed as a set of constraints on the temporal order of messages and the messages' content. The overall constraints consist of local constraints through individual roles. A role module only focuses on how an agent in a given state receives a message of specified type, performs local actions, sends out messages, and switches to another state.

An agent interacts with its environment through sending and receiving messages. Correspondingly, a role process has two gates: *Send* and *Recv*. An action *Send* has a common format *Send !m*, where *m* is a expression of message type. Through this gate, a role process synchronizes with the process *MessageTransportation* to offer a message. A *Recv* action has a common format *Recv !id ?m:Message*. Through this gate, a role process synchronizes with *AgentMessageBuffer* to read a message.

Agents are both reactive and proactive. Agents are proactive because they not only react, but also exhibit goal-oriented behaviors. Agents are reactive because they must take timely actions in response to changes in the environment. In correspondence, the role process consists of a proactive and a reactive part. The active part runs actively depending on its inner state and events. The reactive part executes activities only when it is triggered by incoming messages. The role's structure is illustrated in figure 2.

In correspondence, the Lotos specification of proactive and reactive part has the following structure:

- The proactive part:

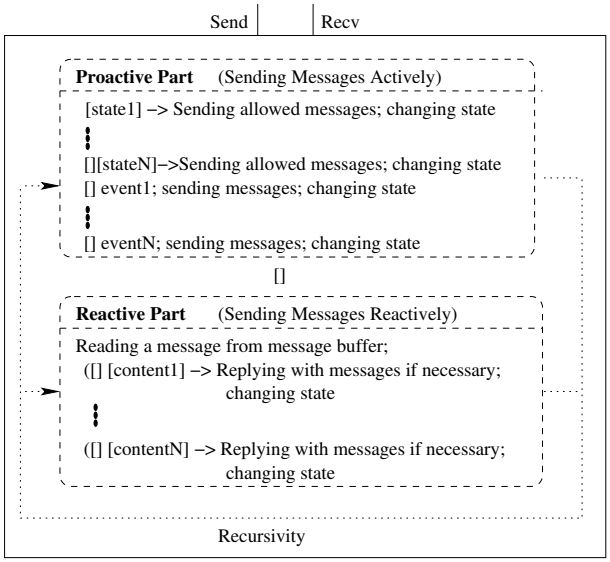

Figure 2: Role Process

[state1] → perform some actions, change its state ...
[]
...
[]
[stateN] → ...
[]
[event1] → perform some actions depending on inner event; change its state ...
[]
...
[]
[eventN] → ...

- The reactive part:

pick up a message from message buffer (
    [content1] → perform some actions depending on state and message content,
        change its state...
    []
    ...
    []
    [contentN] → ...
)

Every role process has a *state* data to control its message output. This *state* reflects the interaction stages from an agent's local viewpoint. A completed interaction can be divided into several stages. In each stage, a role can only send out some specified types of messages. From the beginning to the ending of an interaction, every role should at

least has an initial and a terminal state. Entering into a new state is triggered by what messages a role has received and sent, and sometimes, the inner events, such as time-out. Each state are simply specified in Lotos as distinct values in a sort. Those values should be comparable, hence operations equality and non-equality should be defined.

In each stage of an interaction, a role should record its state, and other information related to a specified application. These information is combined together as a *Session* data structure. The *Session* has following structure: *Session(AgentId, State, Info1, ... InfoN)*. For example, an auctioneer agent session has the following fields: conversation state, administrator, a set of buyers, auction item, seller, starting price, bid increment, reserved price, current bid, current winner. We define the operations to extract each field from a variable of session sort, such as getState, getSeller, and getBuyers.

2.4 Message Transportation Module

The protocols are layered upon an underlying asynchronous point-to-point messaging infrastructure [10]. We use gates *Send and Recv* as well as *buffer* data structure to describe asynchronous message exchanging. The message transportation module is the core functionality that all protocols needs. It can be reused in all interaction protocol specifications without any modification. It simulates the transferring of a message from a sender to several receivers concurrently.

Message transportation module is modelled by two Lotos processes: *MessTransportation [Send, BuffIn]* and *AgentMessageBuffer [BuffIn, Recv]*, as shown in figure 3. Messages transportation delivers one agent's intention and information to other agents. Messages can be sent in a synchronous or asynchronous way. When using synchronous communication, the sender waits (blocked) until it makes sure that the message has been taken. When using asynchronous way, the sender does not wait, instead it continues processing immediately after outputting a message. Message transportation allows both one-to-one and one-to-many messages exchanging.

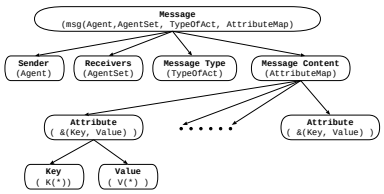

Figure 3: Message Transportation

Asynchronous message exchanging is realized through three synchronization steps.

1. First, the synchronization between a sender and *MessTransportation*.
2. Then, the synchronization between *MessTransportation* and *AgentMessageBuffer*. This step adds a message into a receiver's buffer.
3. Finally, the synchronization between the receiver and *AgentMessageBuffer*. In this step, an agent extracts a message and removes it from the buffer, which buffer is vital for asynchronous message transportation.

3 Incremental Development of AIP

FIPA Interaction Protocols [1] are used as examples to demonstrate AIP development using the proposed framework and methodologies. .

The design process of complex distributed and concurrent systems is a complicated task. Hence it is better to construct AIP in an incremental approach as defined in [31]. This approach is also called step-wise refinement [4]. The design process consists of the steps to identify the message types, states and design the Lotos guard expressions. These expressions specify the conditions and constraints of message pattern in an interaction protocol. The steps are illustrated in figure 4.

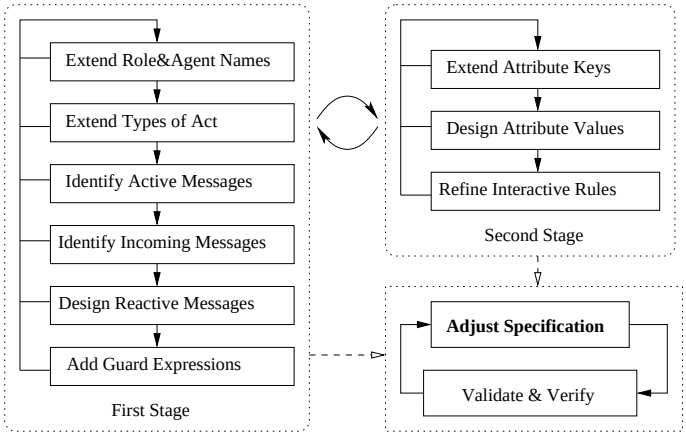

Figure 4: Step-Wise Refinement

The first step is to identify how many types of messages exist in an interaction protocol. Message types are specified as constructors. Because the representation is symbolic, it is

[1] http://www.fipa.org/repository/index.html

easy and convenient to add any new message types. For example, below is the message types in the request interaction protocol.

```
request, refuse, agree, failure, inform-done, inform-result, cancel, not-understood
: → MsgType
```

Next, decide what messages a role can send and receive. Initiator can only send out *request* and *cancel* messages. And a responder can send out *refuse, agree* and so on. Below, we defines both the initiator and responder's possible output of messages.

```
Process Initiator :=
    Send !msg(id, add(p, nil), cnt(request))
  []Send !msg(id, add(p, nil), cnt(cancel))
endproc

process Responder :=
    Send !msg(id, add(ir, nil), cnt(not-understood));
  []Send !msg(id, add(ir, nil), cnt(refuse));
  []Send !msg(id, add(ir, nil), cnt(agree));
  []Send !msg(id, add(ir, nil), cnt(failure));
  []Send !msg(id, add(ir, nil), cnt(inform-done));
  []Send !msg(id, add(ir, nil), cnt(inform-result));
endproc
```

Next, we have to figure out the required interactive rules. The consideration includes the relationships between actions. For example, some requests are allowed to delay, while some others have to be done immediately. To express that two actions occur one after another, we should combine these two actions using the action prefix operator (;). Otherwise, we should assemble them in parallel to allow them and other actions to occur alternately.

Example 3.1 This is an example rule in the *Request* protocol. When a responder receives a *request*, it may respond with *not-understood*, *refuse* or *agree*.

```
[messageType eq request] → (
    Send !msg(...not-understood...);
  []
  Send !msg(...refuse...);
  []
  Send !msg(...agree...);
)
```

Then, identify the agent *states* from an agent's standpoint. *States* identification can be done incrementally. We can add a new *state* whenever we find it necessary during

the development. For example, below is the initiator role's state. In its *initial* state, initiator can send out a *request*; in its *started* state, it can send out a *cancel* message; in its terminal state, it can do nothing related to this interaction.

```
initial, started, terminal
→ InitiatorSessionState
```

Based on the above information, we can design the guard expressions to describe the constraints of message pattern from an agent's local viewpoint. Guards are boolean expressions of states, the conditions that some action can be performed or not. After this step, we can simulate the protocol. If we find something unpredicted, we can go back to above steps to correct and refine the protocol.

Example 3.2 After a responder accept a request, it can send out the following messages at any time.

```
[st eq started] rightarrow (
  Send !msg(...not-understood...);
[]Send !msg(...failure...);
[]Send !msg(...inform-done...);
[]Send !msg(...inform-result...);
```

This approach can help us find inconsistences and then deal with subtle situations in the protocol. First, we can execute the protocol specification via simulation, to observe the possible behaviors. Simulation not only helps users understand interaction pattern between agents, but also helps users find the subtle design errors. Second, verification tools examine the specification against the temporal logics used to describe the properties of a protocol. This can produces all the unwanted behaviors if they exists in a specification.

4 Validation and Verification

There are several kinds of verification in MAS [15, 16], including:

- Verify protocol properties.
- Verify their implementation is equivalent to the protocol.
- Verify that an agent will always comply to an AIP.

AIP properties can be expressed in different temporal logics, such as CTL and ACTL. CADP offers various tools for validation and verification of Lotos Specifications [7]. Verification can increase our confidence that a specification satisfies its requirements and meets some important properties, such as liveness or safety. The general properties of AIP can be categorized as below:

- Safety. It requires that a property be held in any sequence of messages. For example, there is not deadlock in the specification before the protocol is ending. And some actions should always occur before another actions. Deadlock freedom can be expressed by the formula *[true] ¡true¿ true*.

  To facilitate verification, we can define some temporal logic macro expressions. For example, the order of actions can be expressed using a predefined macro *before(action1, action2)*. The macro is defined as below:

  Example 4.1 macro before(A, B) = not([(not (B))* . (A)] false) end macro

- Liveness. It requires that a property should be satisfied by at least one sequence of messages in the protocol. For example, there is not livelock in the specification. LiveLock can be expressed as *¡ true ¿ ("i")*.

- Fairness and Justice. There are different kinds of fairness properties. For example, one requires that each action has the infinite opportunity to be performed. An action ACT will be fairly reached initially can be expressed as: [(not ÄCT¨)] ¡ true . ÄCT¨¿ true.

AIP properties can also be directly modelled in Lotos. The observational equivalence between property specifications and the protocol specification is calculated automatically. The observational equivalence is also useful to check the compliance between agents and protocols.

4.1 Explanation

In order to design agents compliant to a protocol, we have to understand the protocol thoroughly. Given a Lotos AIP specification, the recognition and explanation of a protocol can be accomplished by automatic or manual simulation of the specification. Simulation is symbolically executing a Lotos specification. It can help us understand subtle aspects

in AIP. By simulating a protocol expressed in Lotos, we can observe and recognize its behaviors, and also produce different scenarios of a protocol under different situations. Thus, we obtain an unambiguous understanding of the protocol.

Another way to examine the protocol is to generate the *Labelled Transition System* (LTS) of the protocol. The edges (transitions) represent sending messages. In a LTS, there is a distinguished initial state, in which any conversation starts, and several terminal states that signal the ending of a conversation. The path from the initial state to a final state represents a complete interaction between agents.

We can also obtain different views of an interaction protocol by hiding some gates. For example, in a local interpretation, the edges either model the sending of a message, or its reception. In contrast, from an external observer's view (global interpretation of the interactions among the agents), edges exclusively represent sending messages [21]. For the second viewpoint, we can hide gates *Recv* and *BuffIn* in the specification.

5 Specialization to the Online Auction

The generic framework provides generic data types for reuse, and templates for building correct protocols. Specialization of the framework consists of the following steps:

- Extending, renaming and overloading existing data types
- Deciding the types of messages
- Deciding the states reflecting the evolution of a protocol
- Adding constraints to remove undeterminable choices due to incompleteness
- Refining the protocol to a suitable level by repeating the above steps

The agent-based auction protocol in [7] is used as an example to demonstrate the specialization procedure.

5.1 Data part

A generic data type can be extended using operator "IS". New agent identifiers (roles) and new message types can be easily added as below:

```
GenericAgent
→
TYPE AuctionAgent IS GenericAgent
        administrator,
        auctioneer,
        seller,
        buyer1,
        buyer2 → Agent
ENDTYPE
```

New session states can be defined by first renaming, and then extending.

```
GenericAgentState
→
TYPE GeneralAuctioneerAgentState IS
        GenericAgentState renamedby
SORTNAMES
        AuctioneerAgentState FOR State
ENDTYPE
→
TYPE AuctioneerAgentState IS
        GeneralAuctioneerAgentState
        auctionStarted,
        auctionTimeOut,
        auctionDone → AuctioneerAgentState
ENDTYPE
```

A message content consists of a message type and other data. We can define different message content by overloading "content".

```
AuctionMessageContent
→
TYPE AuctionMessageContent is GenericMessageContent
        content:
                MsgType, Item, Money, Agent
                (* a proposal *)
                → MsgContent
        content:
                MsgType, Agent, Item, Agent, Money, Money
                (* call for proposal *)
                → MsgContent
ENDTYPE
```

5.2 Behaviour part

The *GenericAgent* process can be used as a template to define new process representing a role in a protocol. For example, to design an Auctioneer process, *GenericAgent* will be actualized as *Auctioneer*, *AgentSession* will be actualized as *AuctioneerSession*, and so on.

```
PROCESS GenericAgent[Send, Recv](AgentSession as) : NOEXIT :=
        LET
        id : Agent = getID(as),
        cst : State = getState(as),
        info : SessionInfo = getSessionInfo(as)
        IN (
                ...(* Active part *)
                Recv !id ?c : Message; (
                        Let sender : Agent = getSender(m),
                                cnt : Content = getContent(m),
                                t : MessageType = getType(m)
                        IN(
                                ... (* Reactive part *)
                        )
                )
ENDPROC
```

→

```
PROCESS Auctioneer[Send, Recv](AuctioneerSession as) : NOEXIT :=
        ...
ENDPROC
```

Behavior specialization consists of the following steps:

1. Decide all the message types a role can send out actively. These actions are not triggered immediately by received messages.

```
Deciding the actively outgoing messages
        Send !msg(id, ..., cnt(callProposal,...))
         []
        Send !msg(id, ..., cnt(AuctionSuccess,...))
         []
        Send !msg(id, ..., cnt(AuctionFailure,...))
```

2. Decide all the message types a role will receive.

```
Deciding the incoming messages
        [mt eq subscribe]
         []
        ...
        []
        [mt eq propose]
```

3. Decide the actions a role will perform after it receives a specified type of message.

```
Deciding the reactively outgoing messages
[mt eq cancelSubscription] → (
        Send !msg(id, ..., cnt(acceptCancelSub, ...)
        []
        Send !msg(id, ..., cnt(rejectCancelSub, ...)
)
```

4. Decide how a role will change its state and session information after receiving and sending a message.

```
Deciding the transformation of state and session
        Send !msg(id, ..., cnt(acceptCancelSub, ...));
                AuctioneerAgent[Send, Recv](id, removeBuyer(sender, as))
         []
        Send !msg(id, ..., cnt(rejectCancelSub, ...));
                AuctioneerAgent[Send, Recv](id, as)
```

5. Add more constraints to remove undeterministic choices.

```
[mt eq cancelSub] → (
        [sender ne winner] →
                Send !msg(id, ..., cnt(acceptCancelSub, ...));
                ...
        []
        [sender eq winner] →
                Send !msg(id, ..., cnt(rejectCancelSub, ...));
                ...
)
```

The above steps will be repeated until all the constraints have been added and the protocol is complete. During the procedure, the protocol can be simulated to examine their behaviors even if it is not completed.

6 Conclusion

Lotos language and its related technologies have shown their strength in specifying and analyzing agent interaction protocols. First, they can ambiguously describe the requirements of agent interaction protocols. Second, the specification can be executed symbolically to show the dynamic behaviors of agent communication. Third, it is possible to validate and verify the specification. The success of Lotos in specifying concurrent systems will contribute to the agent fields.

However, Lotos has some limitations, such as that it cannot specify quantitative time and exception handling. ELotos removes those limitations and provides better structuring mechanisms, such as module, interface and other user-friendly features. ELotos is more powerful for specifying distributed systems. We would like to apply ELotos in such area but no supporting tools.

References

[1] Specification of the kqml agent-communication language. http://www.cs.umbc.edu/kqml/kqmlspec/spec.html, 2001.

[2] M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.

[3] B. Bauer, J. P. Muller, and J. Odell. An extension of UML by protocols for multiagent interaction. In *International Conference on MultiAgent Systems (ICMAS'00)*, pages 207–214, Boston, Massachussetts, july 2000.

[4] Kees Bogaards. LOTOS supported system development. In Kenneth J. Turner, editor, *Proc. Formal Description Techniques I*, pages 279–294. North-Holland, Amsterdam, Netherlands, 1989.

[5] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14(1):25–59, January 1988.

[6] Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris A. Vissers, editors. *The LOTOSPHERE Project*. Kluwer Academic Publishers, London, UK, 1995.

[7] Bo Chen and Samira Sadaoui. Simulation and validation of a dynamic online auction. In *7th IASTED International Conference on SOFTWARE ENGINEERING AND APPLICATIONS*, 2003.

[8] R. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng. Modeling agent conversations with colored petri nets. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 59–66, Seattle, Washington, May 1999.

[9] Rossana Maria de Castro Andrade. *Capture, Reuse, and Validation of Requirements and Analysis Patterns for Mobile Systems*. PhD thesis, University of Ottawa, 2001.

[10] Mark d'Inverno, David Kinny, and Michael Luck. Interaction protocols in agentis. In *Third International Conference on Multi-Agent Systems (ICMAS98)*, pages 261–268, 1998.

[11] Mohammed Faci, Luigi M. S. Logrippo, and Bernard Stépien. Formal specification of telephone systems in LOTOS: The constraint-oriented style approach. *Computer Networks and ISDN Systems*, 21(1):53–67, mar 1991.

[12] Luis Ferreira Pires and W. Lopes de Souza. Stepwise refinement design example using LOTOS. In Juan Quemada, Jose A. Mañas, and Enrique Vázquez, editors, *Proc. Formal Description Techniques III*. North-Holland, Amsterdam, Netherlands, November 1990.

[13] FIPA. Acl message structure specification, http://www.fipa.org/specs/fipa00061/sc00061g.html, 2001.

[14] Nicoletta Fornara and Marco Colombetti. Defining interaction protocols using a commitment-based agent communication language. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 520–527. ACM Press, 2003.

[15] F. Guerin and J. Pitt. Guaranteeing properties for e-commerce systems. In *AAMAS 2002 Workshop on Agent-Mediated Electronic Commerce IV*, Bologna, 2002.

[16] F. Guerin and J. Pitt. Proving properties of open agent systems. In *First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 557–558, Bologna, 2002. ACM Press.

[17] Nabil Hameurlain. A formal framework for behavioural reuse of agent components: Application to interaction protocols. In *10th European Workshop on Multi- Agent Systems. Modelling Autonomous Agents in a Multi-Agents Wold, MAAMAW'01*, 2001.

[18] M. Heisel and N. Levy. Using LOTOS Patterns to Characterize Architectural Styles. In M. Bidoit and M. Dauchet, editors, *Proc. Conf. on Theory and Practice of Sw Development (TAPSOFT 97)*, volume 1214, pages 818–832, Lille, France, 1997. Springer-Verlag, Berlin.

[19] Haw Siang Hon. Comparision of kqml and fipa acl. In *SURPRISE 2001*, 180 Queen's Gate, LondonSW7 2BZ, UK, May - June 2001.

[20] Thomas Juan, Adrian Pearce, and Leon Sterling. Roadmap: extending the gaia methodology for complex open systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 3–10. ACM Press, 2002.

[21] Jean-Luc Koning. Designing and testing negotiation protocols for electronic commerce applications. In Frank Dignum and Carles Sierra, editors, *Agent Mediated*

Electronic Commerce, The European AgentLink Perspective, volume 1991 of *Lecture Notes in Computer Science*, pages 34–60. Springer, 2001.

[22] Jean-Luc Koning, Marc-Philippe Huget, Jun Wei, and Xu Wang. Extended modeling languages for interaction protocol design. In *Revised Papers and Invited Contributions from the Second International Workshop on Agent-Oriented Software Engineering II*, pages 68–83. Springer-Verlag, 2002.

[23] Jurgen Lind. Specifying agent interaction protocols with standard UML. In *AOSE*, pages 136–147, 2001.

[24] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 517–526. ACM Press, 2002.

[25] J. Odell, H. Parunak, and B. Bauer. Extending uml for agents. In *Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, 2000.

[26] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In *AOSE*, pages 121–140, 2000.

[27] Luis Ferreira Pires and Wanderley Lopes de Souza. Step-wise refinement design example using LOTOS. In *FORTE*, pages 255–262, 1990.

[28] A. Ribeiro and Y. Demazeau. A dynamic interaction model for multi-agent systems. In *II Iberoamerican Workshop on Distributed Artificial Intelligence and Multi-Agent Systems*, pages 27–36, Toledo, oct 1998.

[29] Munindar P. Singh. Synthesizing coordination requirements for heterogeneous autonomous agents. *Autonomous Agents and Multi-Agent Systems*, 3(2):107–132, 2000.

[30] K. Turner. Constraint-oriented style in lotos. In *British Computer Society Workshop on Formal Methods in Standards*, Didcot, 1988.

[31] Kenneth J. Turner. Template-based specification in LOTOS. Technical report, Department of Computing Science and Mathematics, University of Stirling, UK, May 1990.

[32] Mirko Viroli and Andrea Omicini. Specifying agent observable behaviour. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 712–720. ACM Press, 2002.

[33] Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen, and Ed Brinksma. Specification styles in distributed systems design and verification. In *Selected papers of the 6th International conference on Logic programming*, pages 179–206. Elsevier Science Publishers B. V., 1991.

[34] Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen, and Ed Brinksma. Specification styles in distributed systems design and verification. In *Selected papers of the 6th International conference on Logic programming*, pages 179–206. Elsevier Science Publishers B. V., 1991.

[35] W. Wen and F. Mizoguchi. Analysis and verification of multi-agent interaction protocols. In *EEE APSEC'99*, pages 252–259, Takamatsu, Japan, 1999.

[36] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. A methodology for agent-oriented analysis and design. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 69–76, Seattle, WA, USA, 1999. ACM Press.

[37] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[38] Jie Xing and Munindar P. Singh. Formalization of commitment-based agent interaction. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 115–120. ACM Press, 2001.