

# Extended Modeling Languages for Interaction Protocol Design

Jean-Luc Koning  
Leibniz-Esisar  
50, rue Laffemas - BP 54  
26902 Valence, France  
koning@esisar.inpg.fr

Marc-Philippe Huget  
Magma-Leibniz  
46, avenue Félix Viallet  
38031 Grenoble, France  
Marc-Philippe.Huget@imag.fr

Jun Wei      Xu Wang  
Institute of Software  
Chinese Academy of Sciences  
Beijing, China  
wj@otcaix.iscas.ac.cn  
xuwang@cs.ust.hk

## ABSTRACT

Successful development of agent interaction protocols requires modeling methods and tools that support a relatively complete development lifecycle. Agent-based systems are inherently complex but exhibit many similarities to object-oriented systems. For these reasons not only current modeling languages need to be extended, but also related tools should be provided for agent interaction protocol design to be supported. In this paper, we focus on the design stage of an agent interaction protocol development cycle. We start by giving general criteria for comparing agent modeling languages. The ones we take into consideration in this paper are extensions of Agent-UML and FIPA-UAML languages. We describe these languages and discuss some extensions on a simplified application of the Netbill electronic commerce protocol. We then briefly introduce a component-based formal specification language in order to support the protocol's design stage and present a tool built upon the FIPA norm (making use of the PDN or UAML notation) which supports the analysis and design of interaction protocols.

## 1. INTRODUCTION

The development cycle of agent interaction protocols (AIP) for multiagent systems does not account for as large a literature as the one dedicated to communication protocols in distributed systems. Let us quote Singh's precursory work on interaction oriented programming [11] where protocol engineering comprises three main stages.

**Design and validation.** A dedicated way to tackle stage 1 is through the use of colored Petri nets since such a formalism supports concurrent processing. Besides a whole set of validation tools is available.

**Observation of the protocols' execution.** Stage 2 deals with a post-mortem analysis of the message scheduling.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSE 2001 Montréal, Canada

Copyright 2001 ACM 0-12345-67-8/90/01 ..\$5.00

## Recognition and explanation of conversations.

Stage 3 checks whether the interactions unfolded according to the protocol and that the overall behavior corresponds to what the designer expected. It also highlights the agents' behavior during those interactions and helps pinpoint possible causes of failure.

Because of the quite distinctive nature of the two sets of protocols found in distributed and multiagent systems, it is not possible to fully apply results from works in communication protocols to interaction protocols. Therefore, it is necessary to define a suited development cycle that, when possible, makes use of existing techniques from distributed systems, or otherwise derives new ones.

In this paper, we will focus on the design stage of an agent interaction protocol development cycle. Section 2 starts by giving general criteria for comparing agent modeling languages. The ones taken into consideration in this paper are extensions of Agent-UML and FIPA-UAML languages. We describe these languages and discuss some extensions on a simplified application of the Netbill electronic commerce protocol (section 3). Section 4 then briefly introduces a component-based formal specification language in order to support the protocol's design stage and presents a tool built upon the FIPA norm (making use of the PDN or UAML notation) which supports the analysis and design of interaction protocols.

## 2. AGENT MODELING LANGUAGES

Essentially two families of agent modeling languages have been used for representing AIPs<sup>1</sup>: one is Agent-UML [10] and the other is FIPA-UAML [5]. In this section, we briefly compare these two families as well as their respective extension against a set of general criteria.

### 2.1 Some General Criteria on Agent Modeling Languages

In order to compare agent modeling languages, let us first list a series of nine general criteria one may want to see supported by AIPs one way or another.

<sup>1</sup>Both of these forms were combined in 2000 and are now referred to simply as AUML.

**Roles:** Agents are not only represented by their name but also according to their role within the interaction protocol. Such an approach enables to easily take into account a variable number of agents. Once those roles are identified there is no need to modify the design of the interaction protocol when a new agent is brought into place.

**Synchronous/asynchronous communication:** When agents send messages to one another they wait (resp. do not wait) till those messages are read prior to keep on running.

**Concurrency:** A number of messages can be sent or received at the same time.

**Loop:** A set of messages is sent a number of times. Either this number is explicitly known or the loop is based on a condition that must be true for the loop to keep on being activated.

**Temporal constraints:** An agent specifies a deadline that corresponds to a point in time before which some messages are expected.

**Exception:** A way to handle unexpected events that could either stop the course of an interaction or lead to a failure.

**Design:** Connected to the visual modeling language a set of algorithms and/or tools to go to a formal corresponding definition is provided.

**Validation:** Connected to the visual modeling language a set of algorithms and/or tools for validating properties on interaction protocols is provided. It may either be a structural or a functional validation.

**Implementation:** Some algorithms and/or tools can lead to some code generation to make a protocol executable by the agents.

Table 1 gives a synthesized view on the following four graphical languages AUML, EAUML, UAML, UAMLe against these nine criteria.

The first six criteria deal with the direct characteristics of the visual language, and as a matter of fact, all four languages provide them. Sharper differences between these agent modeling languages appear among the last three criteria, i.e., when one considers them as a stage of an overall AIP life-cycle.

Each of the four agent modeling languages will now be discussed regarding the first six characteristics: roles, synchronous/asynchronous semantics, concurrency, loops, time constraints and exception handling.

## 2.2 Agent-UML and EAUML

AUML [10] [9] is a proposal for the specification of agent based systems. It has been mainly applied to model protocols for multiagent interactions. AUML provides a set of extensions based on UML sequence diagrams, packages,

	AUML	EAUML	UAML	UAMLe
Roles	✓			
Sync./Async.	Both	Asynchronous	Both	
Concurrency	Specific connector		Separation of the various messages using boxes	
Loop	At the level of a message or a group of message			
Time	Through deadlines			
Exception		By means of a special connector for triggering actions	Not directly	Upon a set of messages
Design	Possible augmented UML tools	Algorithms for translation into FSM	No graphical tools	Graphical tool DIP
Validation	No direct bridge to validators	Algorithms for translation into Promela for use with Spin	No direct bridge to validators	Translation to FSM for accessibility analysis and model-checking
Implementation	No known algorithm for protocol synthesis	Code generation	No known algorithm for protocol synthesis	Code generation

**Table 1: Some criteria for comparing agent modeling languages.**

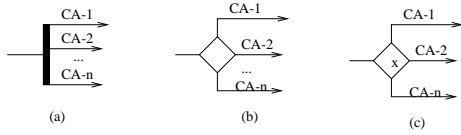


Figure 1: Connectors for message sending in AUML.

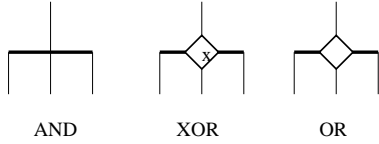


Figure 2: Connectors for lifeline in AUML.

templates: protocol diagrams, agent roles, extended message semantics, multi-threaded lifelines, nested and interleaved protocols, and protocol templates. The core of these extensions for AIPs is found in what could be called *protocol diagrams* that combine sequence diagrams with the notation of state diagrams for the specification of AIPs.

In AUML messages are considered to be exchanged between agents that are denoted by their role. Loops are taken care both at the level of single messages and sets of messages. Conditional branching can be represented by means of standard *if-then-else* choice and also through some dedicated connectors representing concurrent threads. Figure 1 respectively shows *and*, *xor* and *or* connectors.

On one hand the simplicity of UML sequence diagrams makes them suitable for expressing requirement, but lack of semantics makes them sometimes ambiguous and therefore difficult to be interpreted. AUML’s proposal has improved this situation.

EAUML [13] is essentially based on AUML protocol diagrams. It brings forth some simplifications and modifications in that it adopts a somewhat different view from AUML as far as control threads for single agent and message characterization. EAUML should not be seen as a competing alternative to AUML but rather as a way of viewing AIP visual modeling from another angle.

The extended notations for agent lifelines (see figure 2) and message sending (see figure 1) in AUML are kept but with a different semantics as far as the inclusive-or lifeline. Also only an asynchronous semantics is kept for messages. Besides, these messages are abstract symbolic messages rather than mere speech act messages.

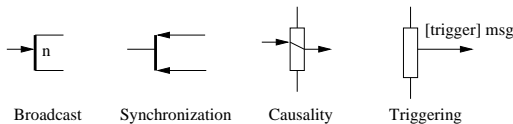


Figure 3: Connectors in EAUML.

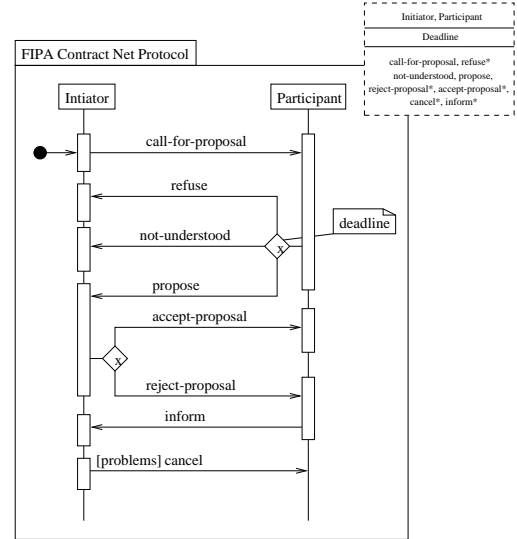


Figure 4: Contract Net in EAUML.

The extensions EAUML provides essentially deal with message passing within sequence diagrams. Figure 3 shows four new connectors. *Message broadcast* corresponds to passing a message  $n$  times. *Message triggering* corresponds to the guarded messaging in UML and AUML but in EAUML this also implies some internal events trigger the message sending. *Message synchronization* and the keyword “*silent*” have no direct counterpart in AUML. The former means an agent has to wait for several messages to arrive. The latter is a constraint that can be placed on messages to denote that the sending or receiving of a message has no effect on the current state. *Message causality* is introduced to indicate a causal relationship between two messages. The purpose of this construct is to simplify the dynamic model.

Figure 4 depicts the contract net protocol expressed as an EAUML sequence diagram. This example very much looks like the one given by Odell et al. [10]. The major difference with the AUML sequence diagram deals with the handling of message *Cancel*. This is not a regular message since it appears only in case of trouble we can make use of the fourth connector of figure 3.

### 2.3 Unified Agent Modeling Language and UAMLe

UAML [5] is probably the first graphical language proposed (by FIPA) for representing AIPs. Most of the characteristics seen with AUML also appear in UAML: agents are denoted via their role, several types of message sendings along with possible added constraints are allowed (synchronous, asynchronous, broadcast, repeated sendings, temporal constraints, etc.). As shown in figure 5, concurrent messages are allowed. Sub-protocols are an interesting notion introduced in UAML that denotes a sequence of messages inside one protocol.

AUML represents alternatives in interaction states by means of activation bars whereas UAML makes use of boxes with

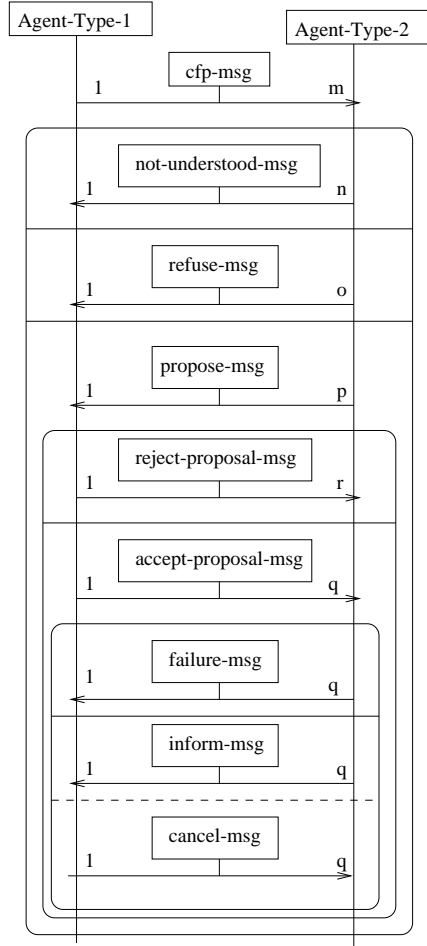


Figure 5: Contract Net in UAML.

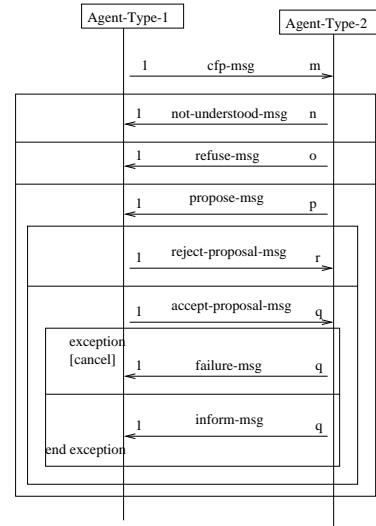


Figure 6: Contract Net in UAMLe.

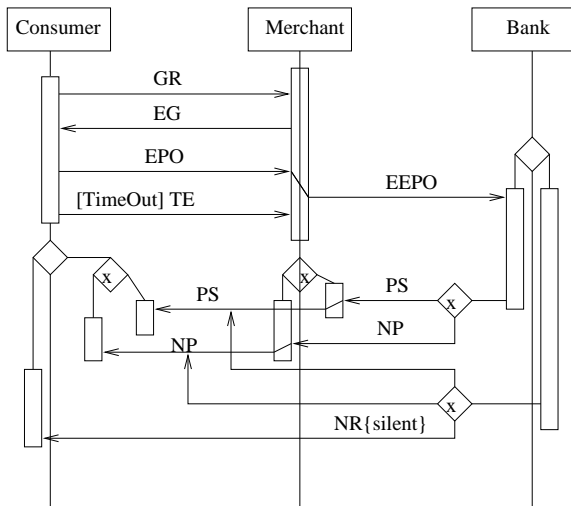
separations between the possible cases. Each message is defined via a box containing a message (i.e., with an arrow). A sub-protocol (e.g., see the box starting with the message *not-understood-msg*) may contain other sub-protocols (as shown by two other nested boxes in figure 5). Possible choices are separated by lines and messages to be handled concurrently are separated by dotted lines (like between *inform-msg* and *cancel-msg*).

Compared with UAML, UAMLe essentially enables to synchronize one agent on several messages of different types and also introduces the notion of exception at the level of a single message as well as a set of messages. In the classical Contract Net protocol (with AUML [10] and with UAML [5]) the final message is a *cancel* message returned by the initiator after receiving the last inform message. Actually, it would be better to send a *cancel* message only if an exception arises. In UAMLe (figure 6) this exception handling is denoted by an *exception [cancel] ... end exception* statement. Therefore the exception applies to the whole time interval that corresponds to the waiting for an answer by the agent in charge of the task. When the exception is caught the *cancel* message is sent to that agent.

### 3. DESIGNING AGENT INTERACTION PROTOCOL

#### 3.1 Designing the Netbill Protocol Using EAUML

In order to clarify the extension in EAUML sequence diagrams, let us look at the agent-based Netbill [3] purchase protocol. Although we give here a simplified version of the Netbill protocol (see figure 7) it embodies the primary characteristics of agent interaction protocols in electronic commerce, including asynchronous messaging, distributed processing, concurrency, communication uncertainties, etc. The agent-based modeling of this protocol can be abstracted to involve only three agents, one consumer, one merchant and a commonly trusted bank. Consumers buy e-goods through



**Figure 7: Transaction Protocol of NetBill in EAUML.**

a web-browser from the merchant. Payment between them is settled by the bank.

Figure 7 illustrates the interaction pattern among the three parties. Message passing is asynchronous. Causal messaging relates in/out messaging into one action such as on the Merchant lifeline where in-message “endorsed electronic payment order” (EEPO) and “electronic payment order” (EPO) are causally related. The triggered messaging implies that some internal event happened so that one message sending is triggered. The triggered messaging on the lifeline of Consumer expresses that the timeout event occurred and then caused message “transaction enquiry” (TE) to be sent out. XOR and OR message sendings are selected depending on some state condition. The XOR message sending between “payment slip” (PS) and “no payment” (NP) on the bank lifeline is chosen based on a state condition about payment transaction status. The *silent* message “no record” (NR) does not affect the state of the recipient (Consumer) nor the sender (Bank).

### 3.2 Designing the Netbill Protocol Using UAMLe

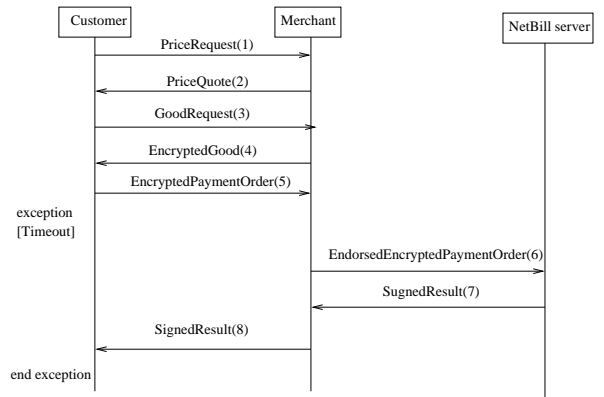
The Netbill protocol is represented with UAMLe in figure 8. It shows the various steps by means of sub-protocols.

In conjunction with the UAMLe modeling language a formal description is provided. We give such a detailed description of the Netbill protocol in section 4.2.

## 4. TOWARDS A COMPONENT-BASED SPECIFICATION

### 4.1 Protocols and Micro-Protocols

It is important for an interaction protocol to be reusable, i.e., a piece of a protocol could be replaced by another without having to start a whole new development cycle and to globally think out the protocol but to be able to reuse parts of a protocol.



**Figure 8: Complete Transaction Protocol of NetBill in UAMLe.**

One could define interaction protocols as sets of components, called micro-protocols (i.e., they represent interaction units that themselves contain a set of performatives and whose contents is the piece of information to be passed on), that can be assembled in a protocol via a dedicated composition language called CPDL. See [6] for an extended article on issues related to the modeling of component-based interaction protocols.

Like components in software engineering, a micro-protocol is defined by an *executable* part which is a set of performatives and an *interface* part for connecting micro-protocols together. Such a micro-protocol is composed of four attributes:

- Its *name* identifies a unique micro-protocol.
- Its *semantics* is used to help designers know its meaning without having to analyze its definition. These two fields make up the micro-protocol’s signature. The other two attributes refer to its implementation.
- Its *parameters’ semantics*. When making use of a micro-protocol it is necessary to know all the parameters’ semantics since they are used for building messages.
- Its *definition* corresponds to the ordered set of performatives constituting the micro-protocol. Each performative is described along with its parameters like the sender, the receiver and the message’s content.

Combining micro-protocols into a general interaction protocol can be done with some logic-based formulae encompassing a sequence of micro-protocols. The relation between the micro-protocols’ parameters should also be specified by telling which are the ones matching. Suppose two parameters  $u$  and  $v$  are used in a same protocol, if they handle an identical parameter, this parameter should have a unique name. This facilitates the agents’ work in allowing them to reuse preceding values instead of having to look for their real meaning. This approach is very much oriented towards data reuse.

CPDL is a description language for interaction protocols based on a finite state automaton paradigm which we have endowed with a set of features coming from other formalisms such as:

- *tokens* in order to synchronize several processes as this can be done with high-level Petri nets.
- *timeouts* for the handling of time in the other agents' answers. This notion stems from temporal Petri nets.
- *beliefs* that must be taken into account prior to firing a transition. This notion is present in predicate/transition Petri nets as well as in temporal logic. Beliefs within the protocol's components as it is the case in AgenTalk [8].

Compared with a finite state automaton a CPDL formula includes the following extra characteristics:

1. a conjunction of predicates in first order logic that sets the conditions for the formula to be executed.
2. the synchronization of processes through the handling of tokens. Such behavior is given through the **token** predicate.
3. the management of time and time stamps in the reception of messages with the **time** predicate.
4. the management of loops that enable a logic formula to stay true as long as the premise is true, with a **loop** predicate.

A CPDL well-formed formula looks like:

$$\alpha, \{b \in \mathcal{B}\}^*, \text{loop}(\bigwedge p_i) \mapsto \text{micro-protocol}^*, \beta$$

A CPDL formula corresponds to an edge going from an initial vertex to a final one in a state transition graph. Such an arc is labeled with the micro-protocols, the beliefs and the loop conditions.  $\alpha$  denotes the state the agent is in prior to firing the formula and  $\beta$  denotes the state it will arrive in once the formula has been fired.

$\{b \in \mathcal{B}\}^*$  represents the guard of a formula. Such a guard is a conjunction of first-order predicates that needs to be evaluated to true in order for the formula to be used.  $\mathcal{B}$  is the set of the agent's beliefs. This guard is useful when the set of formulae contains more than one formula with a same initial state. Only one formula can have a guard evaluated to true, and therefore it is fired. This requires that no formula be nondeterministic and that two formulae cannot be fired at the same time. In the current version of CPDL, predicates used for beliefs are defined within the language, and agents have to follow them.

As indicated earlier the **loop** predicate aims at handling loops within a formula. Its argument is a conjunction of predicates. It loops on the set of micro-protocols involved in the formula while it evaluates to true.

## 4.2 Description of Netbill in CPDL

Given that there are three different roles in the Netbill protocol (see figure 8) it is divided into three interaction parts: one between the consumer and the merchant, one between the bank and the merchant, and one between the merchant and the other two roles. The CPDL expression of the latter protocol is given as

```
init  $\mapsto$  PriceRequest(C,M,G), A1
A1, exception{timeout = exit}  $\mapsto$  GoodsDelivery(C,M,G), A2
A2  $\mapsto$  Payment(C,M,G), end
```

Variables  $C$ ,  $M$  and  $N$  correspond to the consumer, the merchant and the bank. The definition of micro-protocol *PriceRequest* is:

```
request-price(C,M,G).inform(M,C,P)
```

The one for *GoodsDelivery* is:

```
request(C,M,G).send(M,C,G)
```

Micro-protocol *Payment* is defined as:

```
pay(C,M,EPO).pay(M,N,EEPO).inform(N,M,R).inform(M,C,R)
```

Variable  $G$  corresponds to the requested good and  $P$  stands for its price. *request-price* is a performative. Performatives *inform* and *request* have the same semantics as the one in FIPA-ACL [5]. Performative *send* corresponds to the sending of good  $G$ .

Variable  $EPO$  corresponds to the Electronic Payment Order sent by the consumer. When it is passed from the merchant agent to the bank agent, the former agent is adding a key that is necessary for decyphering the good  $G$ , thus leading to the variable  $EEPO$  (Endorsed EPO). Variable  $R$  is the result of the financial transaction as well as the key need for decyphering. The semantics of performative *pay* is the payment of good  $G$ .

The exception inside the second formula corresponds to the case where the consumer agent is refusing the price given by the merchant agent. In Netbill, nothing is said concerning whether the consumer agent has to let know the merchant agent that the interaction is over. Therefore an exception allows to take into consideration the fact that a client agent presumably closed the interaction whenever the duration between the price offer and the request for the good is too long.

## 4.3 A Tool for Supporting AIP Design

We have developed a platform with a tool dedicated to helping design interaction protocols (DIP). This tool follows the

component-based approach presented here above. As shown on figure 9, such a tool enables to design and bring into play a protocol in a graphical manner by relying on micro-protocols and on the compositional language CPDL.

Like the SDL communication protocol description language [12], we advocate two methods for representing protocols: a textual one by means of logical formulas (CPDL) and a graphic one (UAMLe). UAMLe extends UAML in a way to take micro-protocols into account [6].

The protocol described on figure 9 shows four edges labeled with micro-protocols along with their parameters (micro-protocols are written inside boxes). Two of these CPDL formulas have beliefs attached to them (between brackets).

Our platform is endowed with a true graphic editor that enables to define interaction protocols in the graphic language UAMLe. Such a tool allows to (1) build and (2) modify protocols. For this, DIP maintains some information about a protocol: its name, its set of micro-protocols, its semantics and its set of CPDL formulas

Another feature is (3) the automatic translation into CPDL of a protocol represented by a high-level Petri net. (4) DIP allows to display a protocol in the alternate FIPA's notation called the *Protocol Description Notation* (PDN). Unlike UAML (and UAMLe), PDN is a tree-like description of a protocol where each node represents a protocol state and the transitions going out of a node correspond to the various types of message that can be received or sent at the time the interaction takes place. Since DIP is also used in analysis and implementation phases, it is possible to store a description of the protocol in natural language and the designer can generate a skeleton of the protocol in a programming language.

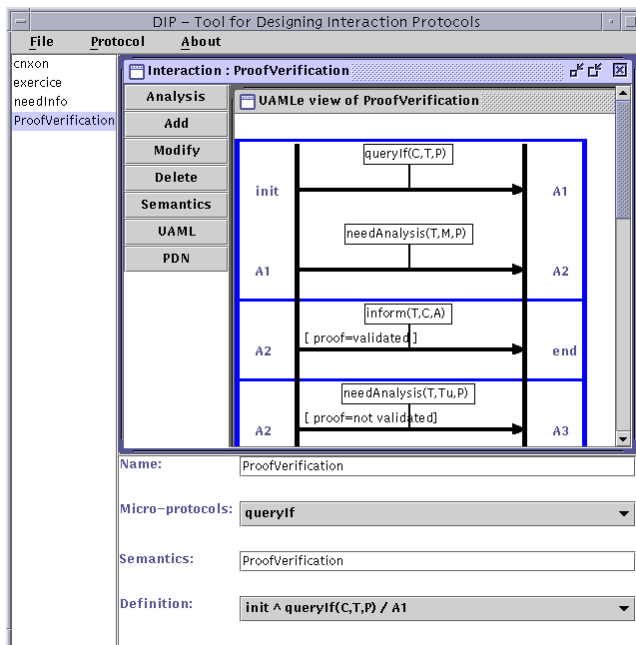


Figure 9: Tool for designing interaction protocols.

## 5. DISCUSSION

In the field of multiagent systems, there are very few tools supporting the design of interaction protocols to date. Let us mention AgentBuilder [1] and AgentTool [2].

As opposed to the approach advocated in this paper, Agent-Builder does not allow for an easy reuse of existing protocols in order to build new ones as with DIP. Protocols in Agent-Builder are defined by means of finite state automata which unfortunately do not efficiently handle synchronization nor meeting points among agents. Furthermore, AgentBuilder leans on a proprietary protocol's structure which makes it very difficult to utilize any external tool to perform validation tests.

It is also impossible to import protocols expressed in some other formalism which limits reusing. In the approach we have presented, interaction protocols are given in an open formalism which makes it possible to import foreign protocols expressed by means of Petri nets for instance.

In the domain of distributed systems protocol engineering has been tackled for a long time. This has led to numerous effective tools. Let us mention Design/CPN [4] which allows to manage protocols by means of colored Petri nets.

Design/CPN enables to graphically design and test Petri nets. Such a software tool is capable of simulating the execution of a protocol but is not open to other tools. Its proprietary formalism for representing protocols forces designers to address the protocol implementation issue with still another formalism which is not adequate as far as validation is concerned. On the other hand DIP aims at keeping a same formalism up to the point of validation [7].

## Acknowledgments

This work was partially supported by the Sino-French Advanced Research Program (PRA SI 00-06). CIPE project: *Component-based Interaction Protocol Engineering*. J.-L. KONING & WEI Jun).

## 6. REFERENCES

- [1] <http://www.agentbuilder.com>.
- [2] <http://en.afi.af.mil/ai/agentool.htm>.
- [3] B. Cox, J. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *Proceedings of the First USENIX Workshop in Electronic Commerce*, July 1995.
- [4] DesignCPN. <http://www.daimi.au.dk/designCPN>.
- [5] FIPA. *Specification: Agent Communication Language*. Foundation for Intelligent Physical Agents, <http://www.fipa.org/spec/fipa99spec.htm>, September 1999. Draft-2.
- [6] J.-L. Koning and M.-P. Huget. A component-based approach for modeling interaction protocols. In *10th European-Japanese Conference on Information Modelling and Knowledge Bases*, Finland, May 2000.
- [7] J.-L. Koning and M.-P. Huget. A semi-formal specification language dedicated to interaction

protocols. In H. Kangassalo, H. Jaakkola, and E. Kawaguchi, editors, *Information Modelling and Knowledge Bases XII*, Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, 2001.

- [8] K. Kuwabara, T. Ishida, and N. Osato. AgenTalk: Describing multiagent coordination protocols with inheritance. In *Seventh IEEE International Conference on Tools with Artificial Intelligence*, pages 460–465, Herndon, Virginia, November 1995.
- [9] J. Odell, H. V. D. Parunak, and B. Bauer. Extending uml for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, Austin, Texas, july, 30 2000. ICue Publishing.
- [10] J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in uml. In P. Ciancarini and M. Wooldridge, editors, *Proceedings of First International Workshop on Agent-Oriented Software Engineering*, Limerick, Ireland, june, 10 2000. Springer-Verlag.
- [11] M. P. Singh. Toward interaction oriented programming. Technical Report TR-96-15, North Carolina State University, May 1996.
- [12] K. J. Turner. *Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL*. John Wiley and Sons, Ltd, 1993.
- [13] J. Wei, S.-C. Cheung, and X. Wang. Towards a methodology for formal design and analysis of agent interaction protocols : An investigation in electronic commerce. In *International Software Engineering Symposium*, Wuhan, Hubei, China., March 2001.