Chapter 1

ENGINEERING INTERACTION PROTOCOLS FOR MULTIAGENT SYSTEMS

A Full Development Cycle

Marc-Philippe Huget*
University of Liverpool
Department of Computer Science
Liverpool L69 7ZF, United Kingdom
mph@csc.liv.ac.uk

Jean-Luc Koning
Inpg-CoSy
50 rue Laffemas, BP 54
26902 Valence cedex 9, France
Jean-Luc.Koning@inpg.fr

Abstract This chapter advocates a full development cycle for the engineering of interaction protocols for multiagent systems. Such an incremental cycle is composed of five main stages, which are presented in detail. After a state-of-the-art introduction on the interaction issue in multiagent systems, we discuss the analysis stage. This section first introduces a prototypical analysis document that helps designers gather a protocol's informal specification. It then applies this procedure to a protocol dedicated to a distributed learning environment. The following section tackles the formal description issue and introduces the related component-based formal specification language called CPDL. It also discusses the use of visual modeling languages and applies it to the former example. Then we explain how to perform the validation stage and how to validate properties. We also focus on the building of an operational version

*This work has been achieved while the first author was PhD student at the MAGMA Team, Leibniz, in Grenoble, France.

of a protocol. The last stage deals with conformance testing. Finally, this chapter ends with some discussion.

Keywords: Multiagent system, interaction protocol, life cycle.

1. Introduction

1.1 Interaction Protocols in Multiagent Systems

Multiagent systems are becoming increasingly popular as a new programming paradigm that provides the right abstraction level and the right model to build a lot of distributed applications. Its basic components are agents which are *encapsulated computer systems that are situated in some environment and that are capable of flexible, autonomous action in that environment in order to meet their design objectives* [1].

In order to tackle the decentralized nature of multiagent problems, a new approach consists in adopting an agent-oriented view of the world. Furthermore, the agents are to interact with one another, mostly to achieve their individual objectives. In order to reach any cooperation/coordination/negotiation goal agents rely on interaction protocols to exchange information.

An agent interaction protocol can be seen as a set of rules that guide the interaction among several agents. For a given state of the protocol only a finite set of messages may be sent or received. If one agent is to use a given protocol, it must agree to conform to such a protocol and obey the various rules. Moreover, it must comply with the semantics of the protocol.

A rule may be either of a syntactic or semantic nature. A syntactic rule has to do with the protocol's architecture, i.e., the transitions between the protocol's states. Semantic rules define actions that the agents are to perform when sending or receiving messages. For example, receiving a message with an *inform* performative may modify an agent's behavior since the related piece of information is added to the agent's knowledge base. From then on, such a piece of information is part of the agent's belief.

Making use of interaction protocols enables agents to reach a solution in a quicker way. Indeed, the agents know the messages they can receive in a given state, the messages they can send and the rules that guide their choice in case of non-determinism when several messages are possible. The agents thus go faster towards a solution.

For a thorough definition of protocols in multiagent systems, one may mention the works from Pitt et al. [2] or Greaves et al. [3].

1.2 Communication Protocols

1.2.1 Communication Protocols in Distributed Systems. Interaction protocols for multiagent systems stem from the distributed and telecommunication systems domain. They are specific though in that multiagent systems are general open systems (such as agent-based systems for electronic commerce for instance). Therefore, such protocols must be able to handle a variable number of agents. Another characteristic is that protocols for distributed systems only deal with data chunks whereas agents have to do with a performative (i.e., verb) plus some contents. Besides, in distributed systems the various processes are embodied in a protocol in order for a task to be entirely performed. On the other hand, agents are capable to decide if and when they will start interacting, or as Odell et al. put it "an agent is an object that can say 'go' (dynamic autonomy) and 'no' (deterministic autonomy)" [4].

Holzmann [5] and Lai and Jirachiefpattana [6] have given extensive definitions of protocols in the field of telecommunication.

1.2.2 Communication Protocol Engineering. The development cycle of communication protocols usually embodies seven stages: identification of needs, design, formal description, validation, protocol synthesis, conformance testing, and interoperability testing. Let us give a brief definition for each of these stages.

- The first stage consists in identifying the needs in terms of available and required communication services. The heart of designing protocols deals with the building of protocols whose available services should be as close to the required ones as possible. Therefore, such an informal description of a protocol in natural language should be both comprehensible and complete. A thorough example of some assessment of needs for a data transfer protocol such as the *Sliding Window* can be found in [7].

- The design stage aims at obtaining a more formal description of the protocol. Designers usually rely on use cases, chronograms (an example of the TCP protocol may be found in [8]) or algorithms.

  A main difference with the preceding stage is the introduction of messages and types of messages. The first stage only provides actions whereas this one also defines the message related to these actions as well as their type.

  Both these stages are essential and are prerequisites for a sound protocol design.

- Holzmann [5] points out that the use of natural language may lead to ambiguities or wrong interpretations. It is therefore necessary to make use of a formalism to be as precise and rigorous as possible. This stage is called the formal description and consists in formally defining a protocol. Many formalisms can be used for representing communication protocols. Among the main ones, let us mention finite state automata [5][9], Petri nets [10], and languages such as Lotos [11], Estelle [6] and SDL [6].

  Going from an informal to a formal description is not an automated process and therefore heavily depends on the designer's skill. Having a formal description enables use of protocol handling and validation tools.

- The validation stage ensures that properties defined during the first stage are present in the formal representation of the protocol. Two different paths for the validation can be followed.

  On the one hand, the *reachability analysis* deals with the validation of general properties that do not depend on the protocol's objectives. On the other hand, model checking has to do with properties related to the protocol's objectives. In both cases, one has to build the graph of the protocol's reachable states, i.e., the set of interaction states accessible from the initial state.

  The reachability analysis checks the presence or absence of leaves representing the protocol's terminal states in the graph. As far as model checking, properties are translated into a temporal logic formula and then validated on the graph via a model checker. Given that protocols may be complex and large, the generation of the graph of accessible states may be time consuming; the generation of such a graph may be reduced by a number of algorithms [5] [6].

- The protocol synthesis stage consists in obtaining an operational version of the formalized and validated protocol. This is a two-step stage. First, an algorithm automatically generates the skeleton of a program whose code corresponds to the protocol's transitions. Second, the designer inserts the code that corresponds to the actions to be performed following a transition between two protocol states. This stage adds some semantics to the protocol.

- As just seen the protocol synthesis stage involves some code that is manually added by the designer. This may have introduced some flaws or omissions. Conformance testing is thus a stage that ensures the obtained protocol still follows the assessment of needs.

It therefore checks whether the behavior of the operational protocol matches the formal one.

- The last stage is the interoperability testing stage which checks whether two processes using the protocol are capable of interoperating, or whether two versions of the protocol are able to communicate.

1.3 Engineering Interaction Protocols

1.3.1 One Approach. Figure 1 proposes an incremental development life cycle for agent interaction protocols. It relies on the traditional communication protocol life cycle as much as possible and introduces specificities whenever interaction protocols cannot be likened to communication protocols.

The full cycle encompasses the following main stages. The *analysis* stage consists in a natural language description of the protocol to be designed. It identifies the various agents' roles, the message exchanges, and their contents as well. This stage embodies the first two stages given in the communication development cycle.

The next stage consists in deriving the protocol's *formal specification.* This is achieved via a graphical modeling of the protocol by a designer, which makes the modeling results amenable to algorithm processing. This stage aims at getting rid of ambiguities that might have been introduced earlier due to the natural language description. It incorporates some static verification of the visual specifications. When errors are found, the process traces back to visual modeling for their modifications.

Then comes a *validation* stage, which checks the protocol's behavioral properties. An algorithm automatically generates finite state machines (FSM) from the textual format of the modeling specification. A semi-automatic algorithm and some hand-coding rules are provided to translate the protocol's FSMs to Promela [5] specifications. A model-checker Spin [12] is used to simulate agentsÅf interactions and debug some design flaws and errors, and to verify the satisfaction of some property specification expressed in linear temporal logic [13].

The *protocol synthesis* stage consists in automatically generating the executable version of a protocol that will be integrated in the agents. A *conformance testing* stage then checks whether the protocol's code conforms to the properties defined earlier in the analysis stage. As for communication protocols, it turns out that the two crucial stages are specification and validation [14].

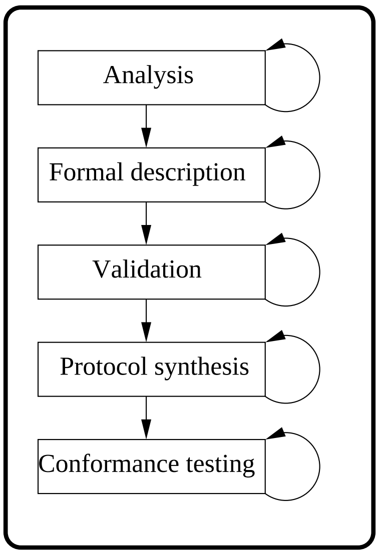


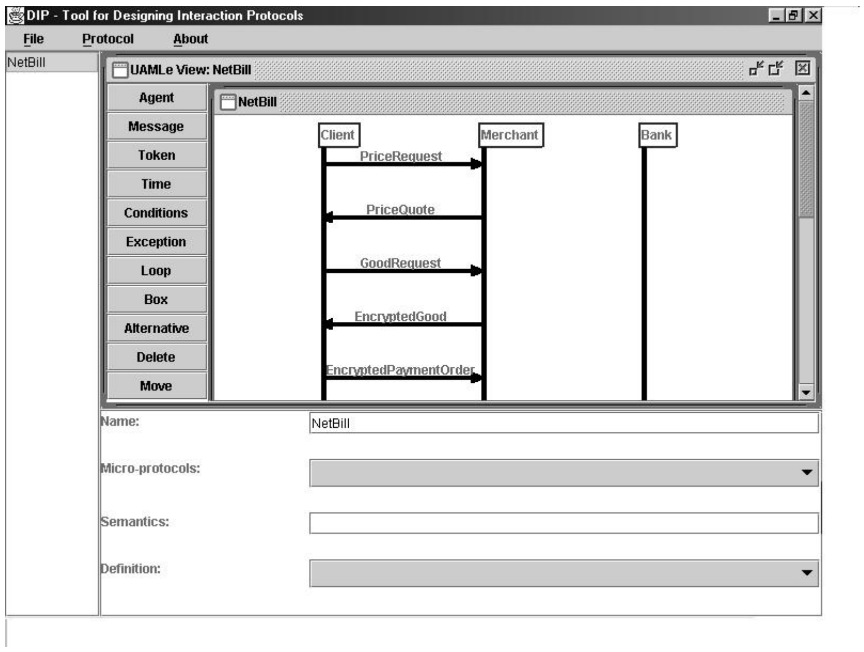
Figure 1.1. Interaction protocol development cycle.

As illustrated in the previous figure, there is no interoperability stage since it intrinsically poses hard problems due to the nature of the (autonomous) agents.

1.3.2 Some tools. Not only should a development cycle provide a procedure to go from a set of requirements to a fully operational protocol, but also it is of crucial importance for it to rely on tools, which can help designers in their task. This chapter details three of them.

A Modeling Tool to Support the Design Stage: DIP. We have developed a platform with a tool dedicated to helping design interaction protocols (DIP). This tool follows a component-based approach. As shown on Figure 2, it is endowed with a true graphic editor that enables definition of interaction protocols in the graphic language UAMLe by relying on micro-protocols and on the compositional language CPDL.

Another feature is the automatic translation into CPDL of a protocol represented by a high-level Petri net. DIP allows the display of 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 messages that can be received or sent at the time the interaction takes place. Since DIP is also used in analysis and implementation stages, 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.



Name: NetBill

Micro-protocols:

Semantics:

Definition:

Figure 1.2. Tool for designing agent interaction protocols.

The protocol described in Figure 2 shows the three agents' roles. Each interaction between the agents is represented by a directed edge labeled with the speech act used in the corresponding message. The first two messages deal with the handling of an exception.

A Tool for Testing Protocols: TAP. TAP (Testing Agents and Protocols) is a tool dedicated to the validation of interaction protocols. It performs the accessibility analysis of CPDL described protocols and translates their CPDL representation into their PROMELA counterpart [5] in order to be able to make use of the well known SPIN model checker [12].

A Tool Dedicated to Conformance Testing: CTP. The CTP (Conformance Testing of Protocols) tool is capable of generating a graph of accessible states for a formal and valid protocol, and checks whether the protocol's behavior is correct.

1.4 Overview of the Baghera Application

Prior to dealing in detail with each of the stages of the interaction protocol engineering process let us introduce one application that will help exemplify the approach advocated all through this chapter.

The aim of the BAGHERA project [15] is to build a decentralized system, which supports distance learning. Such an application is originally dedicated to the teaching of eighth-grade geometry problems when a student has an enforced long stay at the hospital. In order to avoid wasting a whole school year those students are given the opportunity to log into a local workstation. From there they are able to reach a set of new or partially known geometry problems as well as pieces of knowledge related to currently or already studied topics.

The multiagent system Baghera gathers a diversity of actors (professors of various competencies and students of various levels) and types of actors (artificial and human) spread on a network. Compared to current tele-teaching platforms Baghera's multiagent approach introduces new features. First, it affords a spatial distribution and mobility to the students and professors. They do not have to be in a given location. Second, the frequency, duration and time of login are left open for the students as well as the professors. There is no requirement to be connected at a given time. Third, it ensures a personalized learning scheme follow-up which allows each student to receive help and suggestions that take into account his/her own learning trajectory, and to support a professor's help to a student by providing him/her with a piece of information that is relevant for that student as far as the current reasoning situation s/he is in, or, more globally, his/her learning progress. As opposed to an interaction via a video channel for instance where a professor and a student need to rebuild a common setting in order to be able to efficiently interact (student level, special difficulties related to a knowledge domain and the current problem, etc.). Such a multiagent system contributes to this construction by providing to professors pieces of a student's history or by helping to match in a relevant way a student to a professor, or even a student to another student when this pairing is enough to make the student's learning progress.

By means of such a computer system (see Figure 3), students are in direct relation with several artificial agents.

- A *companion* agent assists a student by providing him/her a graphical user interface that will help interaction with the other agents and mostly with the tutor agent. The *companion* agent is also in charge of managing the student's school bag which encompasses both the already and yet to be solved geometry problems as well as some didactic information such as his/her knowledge level and his/her knowledge background (the theorems, axioms, proofs s/he knows).

- A *tutor* agent has clearly a didactic purpose. It coordinates the rest of the student's agents and guides his/her learning scheme via the *companion* agent by providing assignments that comply with his/her level. It is endowed with dialog capabilities in order to manage the interaction with professors or possibly some other students. One of its main goals is to rebuild the learning setting of a student (what s/he is knowing or lacking) whenever a professor needs it. It also possesses negotiation capabilities with the *verificator* agent in order to shape the derived proofs given the student's knowledge profile. It steps into the students' reasoning process if they take a wrong path in their problem solving or if they have a hard time understanding a proof. Therefore it keeps a close look at what a student is doing and what is going on in the system. It takes into account the *verificator*'s viewpoint in order to guide the students and lead them in the right procedure towards a solution.
- A *verificator* agent analyzes the validity and coherence of a proof given by a student. It allows the building of analogies between what a student is currently doing and the proofs s/he may have obtained in the past. It encapsulates a resolution module that helps in building mathematical proofs by suggesting whole or partial demonstrations. Such an agent is a formal reasoning tool capable of refutation, belief revision, and the proposing of counter-examples.

In a similar fashion, a *companion* agent assists professor, which help her/him interact with her/his *assistant* agent via a graphical user interface. This latter agent's purpose is to guide a student's learning progress through his/her *tutor* agent. Each professor owns a set of compartments that hold geometry problems arranged according to their type and difficulty level. Baghera's approach consists in allowing the *tutor* agent to call into play the *verificator*'s services. The scenario between a student and a computer is thus rather general compared to the one where a student receives an evaluation from a professor several days after turning in an assignment.

1.5 Chapter's Outline

This chapter advocates a full development cycle for the engineering of interaction protocols for multiagent systems. Such an incremental cycle

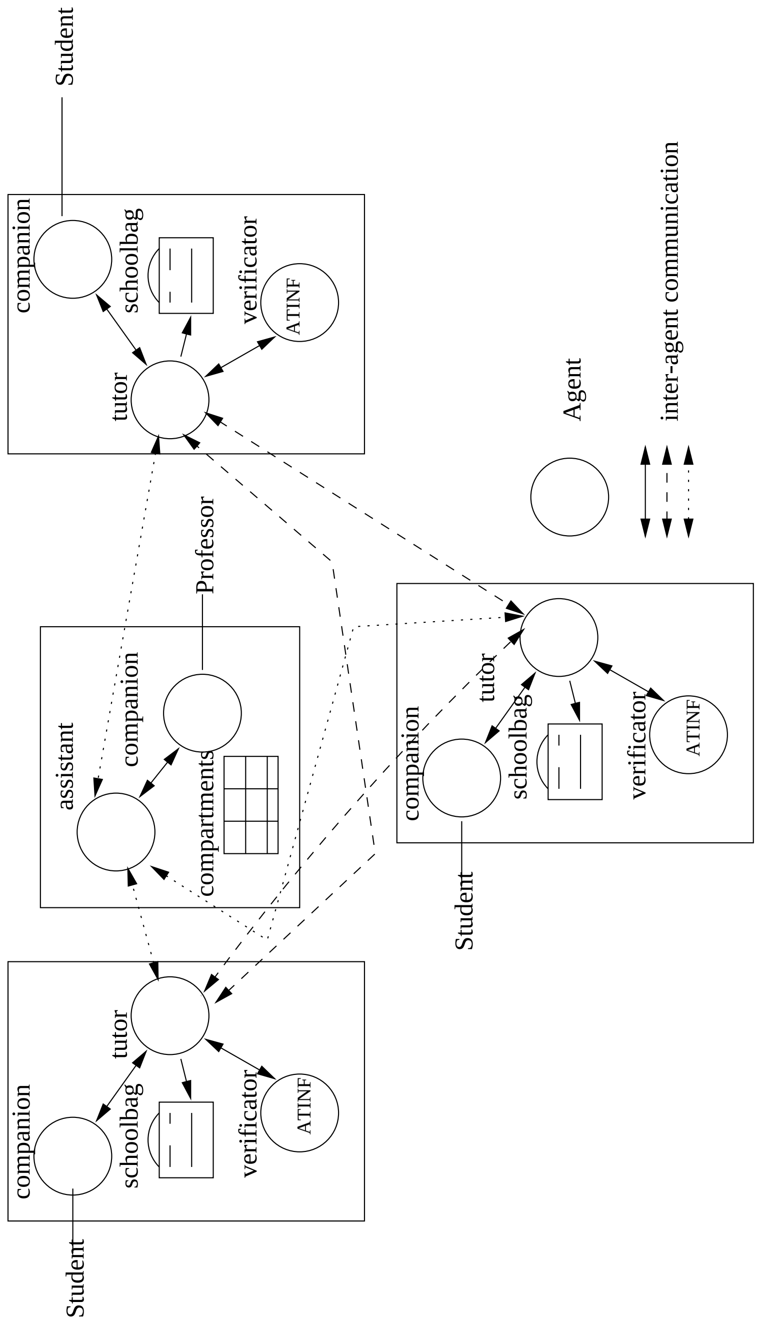


Figure 1.3. The Baghera system

is composed of 5 main stages, which are presented in detail. Section B discusses the analysis stage. It first introduces a prototypical analysis document that helps designers gather a protocol's informal specification. It then applies this procedure to Baghera's *ProofAnalysis* protocol. Section C tackles the formal description issue and introduces the related component-based formal specification language called CPDL. It also discusses the use of visual modeling languages and applies it to the Baghera example. Section D explains how to perform the validation stage and how to validate properties. Section E focuses on the building of an operational version of a protocol. The last stage deals with conformance testing and is explained in section F. Finally, this chapter ends with some discussion.

2. The Analysis stage

The first stage in all design cycles is the analysis stage or the specification of requirements. This one aims at defining the goal of the product to design. In our case, this goal is the protocol's unfolding.

The analysis document is a natural language description that encompasses the plain specification of needs and the general design step as exhibited in distributed systems engineering [5]. On the one hand, this analysis stage defines the protocol's unfolding. On the other hand, it describes its data types and messages. We suggest that some analysis document gathers the following fields: the protocol's name, its keywords, the agent roles involved in the interaction protocol, the agent which initiates the interaction, any prerequisites, the protocol's unfolding, the constraints on the execution, and the postconditions.

It is composed of a set of fields:

Name: This field corresponds to the name of the protocol. It is used in order to distinguish one specification of requirements from others.

Keywords: A protocol designer often reuses previous protocols or parts from them. It is necessary that designers can easily look for a protocol, which fits their needs. Using keywords is an interesting solution since they allow us to distinguish easily one protocol from the others, and particularly, when one has a set of protocols which all have the same domain of action but some different features. For instance, the Contract Net has several different protocols available: n agent negotiation, iterated negotiation, etc.

Agents: In fact, this field gives the role of each agent in the protocol. For instance, in the electronic commerce protocol NetBill [16], one does not mention the agents Smith#1 and Jones#1 but their role: client. This notion of role is a convenient way to tackle the notion of dynamicity in the protocol and the openness of multiagent systems. So, when new agents appear during the interaction, one does not change the protocol.

Initiator: This is the beginner of the interaction.

Prerequisite, constraints and termination: Some constraints must be checked during the protocol use cycle. Before the interaction, the prerequisites have to be true in order to begin the interaction. During the interaction where constraints must be fulfilled and at the end of the interaction when the termination predicates must be checked.

These all three fields are not used at the same time. The prerequisite field is used during the agent design and particularly, during the design of the agent part managing protocol execution. The two remaining fields are considered during protocol design.

Summary: A protocol description might represent a voluminous document so, it is worthwhile if designers can read a summary of this protocol provided by this field.

Protocol's unfolding: This is the main field in the specification of requirements. It contains a set of information about the goal of the protocol and its unfolding. Moreover, messages and message types are described for each step of the interaction. Its unfolding is presented either in natural language or with chronograms or use cases [17]. The design of this specification of requirements is essential and must be **accurate**. Actually, its document is present in all stages of the design cycle. The protocol's unfolding field is used during the formal description stage. Both constraint field and termination field are used during the validation stage. Finally, the prerequisite field is considered during the protocol synthesis stage.

The writing of this document might be realised with our tool DIP described before (see Figure 2). It is thus possible to integrate the specification of requirements and the formal description in one tool.

We illustrate this stage with one example coming from the Baghera project. This example is the verification of proofs realised by students. Students write proofs for their geometrical exercises and want to know if their proofs are correct or not. They ask their *companion* agent to check their proofs. Then this agent asks the *tutor* agent which asks the *verificator* agent. This agent checks if the proof is correct or not. It can also reply that it can not answer on the validity of the proof. After that, the *verificator* agent answers the *tutor* agent, which forwards this answer to the *companion* agent.

Name of the protocol: ProofAnalysis

Keywords: Baghera, proof validation

Agents: Student companion, tutor, verificator, teacher companion and assistant

Initiator : Student companion

Prerequisite: The student must be connected to the system

Summary: The student informs his/her *companion* agent that he/she wants to know if his/her proof is valid or not. The *companion* agent asks the *tutor* agent about it. Then, the *tutor* agent asks the *verificator* agent about it. Three answers are coming from the *verificator* agent: either the proof is valid or it is invalid or the *verificator* can not answer. In the two first cases, the answer is forwarded to the *companion* agent. In the latter case, the *tutor* agent then asks the other *tutor* agents if they have an answer about this analysis. If they have an answer, this one is forwarded to the *companion* agent and finally, if they do not have answers, the *tutor* agent asks the teacher's *assistant* about the answer and forwards its answer to the *tutor* agent.

Constraints: Nothing

Protocol's unfolding: The *companion* agent asks the *tutor* agent about the proof analysis with the *query-if* performative and the content of this message is the proof. The *tutor* agent forwards this proof to the *verificator* agent with the same performative. The *verificator* agent answers with the performative *inform.* The content of the message is either *true* if the proof is valid, or *false* if it is invalid or *noidea* if the *verificator* can decide if the proof is valid or not. If the content is *true* or *false*, the *tutor* agent sends the information to the *companion* agent with the *inform* performative. Finally, if the content is *noidea*, the *tutor* agent asks the other *tutor* agents about the proof with the *query-if* performative. These *tutor* agents answer with the *inform* performative. If the content is *true* or *false*, it is sent to the *companion* agent. If the content is *noidea*, the *tutor* agent asks the teacher's *assistant* about the proof. Whatever is the answer, the *tutor* agent answers to the *companion* agent with the *inform* performative.

Termination: An answer is sent to the student, either his/her proof is valid or invalid or it is impossible to decide if this proof is correct or not.

3. The formal description stage

A natural language specification of requirements is described in the previous stage. As Holzmann notes, using natural language can carry a misunderstanding of the protocol's properties or its unfolding [5]. Actually, if the specification of requirements is described by one designer and the design stage by another one, the document has to be without misunderstandings on the protocol's unfolding. Moreover, natural language does not make the reuse of such parts of the protocol easier. Finally, this approach does not allow checking if some properties are present in the protocol. A convenient solution is to use an accurate and rigorous tool in order to represent a protocol. This tool is called a formalism. Therefore, the formal description stage consists in translating an informal description of a protocol into a formal one. This stage aims at getting rid of ambiguities that might have been introduced due to the natural language description.

This stage is essential either in communication protocol engineering or in interaction protocol engineering.

Here is a list of formalisms used in order to represent interaction protocols:

- Finite State Automata [18], AgenTalk [19-21], COSY [22, 23]
- Petri Nets [24, 25]
- Temporal logic [26-28]
- Z [29-31]
- LOTOS [32]
- SDL [33]

3.1 Towards a New Interaction Modeling Language

We have identified five major characteristics an interaction modeling language might have: reuse, synchronization, validation, ease of design and tools.

C 1. **Reuse.** Designers have to reuse previous protocols or such parts of them as Singh quoted it [34]. The reuse notion is linked with the notion of modularity since it is easier to reuse protocols defined modularly.

C 2. **Synchronization.** Agents act concurrently in multiagent systems. The parallelism of action is also present in interaction. So, the formalism has to implement the notion of meeting points and the ability to synchronize agents.

C 3. **Validation.** The formalism has to be provided with algorithms and tools in order to check properties. At least, some translations into another formalisms, where algorithms and tools exist, must exist.

C 4. **Ease of design.** The design of protocol in this formalism has to be easy and, if possible, it has to contain a graphical modeling language.

C 5. **Tools.** Designers must have tools in order to design and to check properties.

The comparison between these criteria and these formalisms is summed up in table 1.

Formalism	C_1	C_2	C_3	C_4	C_5
Finite state automata	-	-	+	+	+
Petri nets	+	+	+	+	+
Temporal logic	-	-	+	-	-
Z	+	-	-	-	-
LOTOS	+	+	+	-	+
SDL	+	-	+	+	+

Table 1.1. Comparison between formalims and criteria.

The plus sign denotes that the formalism fulfills the criterion. The minus sign does not mean the formalism does not take into account the criterion but this is not fully present.

This article does not describe all the explanations of the results. One can find these ones in [35]. One would rather focus on the conclusions coming from the comparison between criteria and formalisms.

As far as one reads the table, some formalism would not be chosen: finite state automata, temporal logic and the languages Z and SDL. Actually, these ones do not take into account the notion of synchronization, which is essential in multiagent systems.

Only two formalisms can be considered: Petri nets and LOTOS. Petri nets are reusable if one takes hierarchical ones or recursive ones. However, there are few tools allowing their aggregation. LOTOS is difficult to understand and is not provided with graphical description. One can also mention that these formalisms do not take into account the notion of openness, the modification of the number of agents during the interaction.

The solution we consider is to define a new modeling language called CPDL, which fulfills all the criteria. The criterion reuse appears thanks to a component-based approach. Moreover, the synchronization between agents is possible. The verification of protocols is realized by algorithms using CPDL and by translations into finite state automata and Petri nets. CPDL is a textual language and we add two graphical languages: GrCPDL (Graphical CPDL) and UAMLe (Unified Agent Modelling Language, an extended version). Finally, we propose a tool to design interaction protocols in CPDL, GrCPDL, UAML and UAMLe: DIP.

3.2 A Component-Based Approach

The approach presented here is based on components. A protocol is thus a combination of components, each of which having a particular role. This research is motivated by a twofold acknowledgment:

1 The design of interaction protocols moves increasingly towards a design based on the reuse of existing pieces but the current description methods do not allow for such reuse yet. Indeed, these description methods force a designer to carry out a tedious analysis in order to extract the desired protocol pieces.

2 Needs in interaction protocols match up. For example, only one part of the FIPA-Contract-net and the FIPA-iterated-Contract-net [37] is specific. Therefore it seems natural to define two parts within each of these protocols: a common part implemented only once and used in both protocols, and a specific part implemented in each of them. Since components encapsulate a particular behavior, it is possible to define components with some well-defined properties. This way, it would be possible to create components for specific purposes such as negotiation, cooperation, information request, etc.

Singh [38] looked into the design of protocols and noticed some problems raised by current technologies while dealing with the formal semantics of protocols. He enumerates four issues that are closely akin to our conclusions:

- Interactions between agents must be designed from scratch every time.
- These interactions' semantics is included in the procedures, some of which embody code related to the network or to the operating system. This makes the system's validation and modification not commonplace and sometimes rather difficult.
- Systems are designed independently from each other and cannot easily be integrated.
- Updating or modifying a system in an elegant manner is impossible: a new one cannot replace an agent.

A description of the advantages of this approach is presented in [5,35,39].

3.3 Definition of Micro-Protocols

In the previous section, we said that a component-based approach was advocated. A component is called a micro-protocol since it refers to a part of a protocol.

A micro-protocol is composed of four attributes:

1 Its **name** identifies a unique micro-protocol.

2 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.

3 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.

4 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.

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.

The parameter's semantics field contains the semantics of each parameter used in the protocol. This information is interesting for two reasons: designers can understand what are the requested data and the interaction module can ask unknown data to the reasoning module.

Designers are not constrained to follow a specific semantics. Actually, the definition of semantics and ontologies goes past our work in protocol engineering. The FIPA association proposes a set of semantics: SL0, SL1 and CL for example (see `http://www.fipa.org`).

The definition field is the heart of the micro-protocol. It contains the set of performatives used during the interaction. Each performative has parameters: usually the sender, the receiver and the content. This definition is not restricted to an ordered set of performatives but it is possible to add loops, choices, deadlines, synchronization points and exceptions.

Here is the grammar for the definition field:

```
<definition> ::= <exception> <definition> |
                 <time> <performatif> <definition> |
                 <time> <token> <performatif> <definition> |
                 <time> <exception> <performatif> <definition> |
                 <token> <performatif> <definition> |
                 <loop> <definition> |
                 <alternative> <definition> | ϵ
<exception> ::= 'exception(' <corps-exception> ')'
<corps-exception> ::= <condition> ',' <corps-exception> |
                      <condition>
<condition> ::= <identificateur> '=' <identificateur>
<time> ::= 'time(' <nombre> ')'
<token> ::= 'token(' <nombre> ')'
<loop> ::= '{'<condition>'}' '['<definition>']'
<alternative> ::= '('<corps-alternative>')'
<corps-alternative> ::= <definition> '|' <definition> |
                        <definition>
<performatif> ::= <id>'('<id>','<id>','<id>')' | <exit>
<id> ::= <lettre> | <chiffre> | <id> <lettre> | <id> <chiffre> |
         <perf> | <exp> | <timeout>
<lettre> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|
             X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|
             u|v|w|x|y|z
<chiffre> ::= 0|1|2|3|4|5|6|7|8|9
<perf> ::= perf
<exp> ::= exp
<timeout> ::= timeout
<exit> ::= exit
```

- The notion of choices reminds us of the fact that it is possible to have several messages for one step of the interaction. It is possible to choose between several messages for the sending or the receiving action. One speaks about messages and not performatives since it is allowed to have the same sender and receiver and a different content. A vertical bar separates each alternative from each other, and all the alternatives are enclosed in parentheses. For instance, (agree(A,B,C) | disagree(A,B,C)) proposes two alternatives, either agree or disagree.
- It is noteworthy that designers can have the notion of loops without making some artificial methods like linking the initial state of the loop to the final state of the loop and forcing the exit of the loop by the reasoning part of the agent. We provide a convenient method where predicates are enclosed in brackets and the body of the loop in square brackets. The loop follows until the conditions become false.

Here is an example:

```
{hasMoreElements}[request(A,B,"one element").answer(B,A,E)]
```

This is an iterative data request. The request is made by the request performative and this request follows until the knowledge *hasMoreElements* becomes false. This value is modified by the content of the answer performative.

- The time predicate deals with deadlines, i.e., a time for receiving a message. This predicate has one parameter, which is the number of time unit. An example of deadline is an information server, which sends information each time the deadline is passed.
- The token predicate, similar to the token in Petri nets, allows for synchronizing several agents on one point of the interaction. If the designer inserts token(3) on a performative. He/she expects to receive three answers coming from other agents and the interaction follows only if these three messages are received.
- There is an intelligent management of tokens since the three waited messages are not necessarily the same but might be different. For example, some agents can answer with the *agree* performative, some others with the *disagree* performative and some others with the *not-understood* performative. If the number in parentheses is a star, the number of agents for the previous sent message is taken into account in order to define the number of waited messages. The

time and token predicates are placed before the performative and are used once.

- The exception predicate is global in comparison with the last two predicates. If designers want to change the exception, they have to insert again an exception. The exception of interest is to manage some unusual or unexpected cases, for instance, an unknown performative. Each exception is separated by others by a comma. An example is when a client wants to connect to a server and the latter informs the client that the connection is refused. The micro-protocol will be:

  ```
  exception(C="refused").request(A,B,"connection").
  answer(B,A,C).inform(A,B,identity)
  ```

  When the exception is triggered, the action is not inserted in the interaction module but in the reasoning part of the agent. Actually, an action could use some knowledge stored in the knowledge base of the agent. The interaction module only informs the reasoning part of the agent that an exception is triggered.

All the combinations between the time, token and exception predicates, the loops and the choices are not allowed. Moreover, it is possible to insert all these features inside a loop or a choice. Usually, the exception predicate is inserted at the beginning of the micro-protocol definition. If one finds an exception predicate during the definition, it replaces the previous one.

The only interesting combinations are:

- time-token: it deals with the ability for the agent to wait several messages and if all the messages are not received before the deadline is passed, the agent can follow the interaction.
- time-exception: if the deadline is passed and the exception predicate contains the keyword timeout then it is possible to trigger an action corresponding to the time passing.

3.4 The CPDL Language

Combining micro-protocols into a general interaction protocol can be done with some logic formulae encompassing a sequence of micro-protocols. The relation between the micro-protocols' parameters should also be specified by saying which are the ones matching.

Suppose two micro-protocols α and β 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 automata 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 marked 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.

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 timestamps 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:

$$\alpha, b \in B^*, loop(\wedge p_i) \longmapsto micro-protocol^*, \beta$$

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.

α denotes the state the agent is in prior to firing the formula and β denotes the state it will arrive in once the formula has been fired.

{b ∈ 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. 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 non-deterministic 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.

States **init** and **end** are reserved keywords. They correspond to the initial and final states of a protocol. Several final states may exist because several sequences of messages may take place from a single protocol. For example, with the Contract Net protocol [40] whether a bid is accepted or not leads to two different situations; one providing an actual result (*accepted-proposal*) and one stopping the negotiation process (*rejected-proposal*).

We illustrate CPDL language with one example coming from Baghera. It is the protocol for proof analysis described in the analysis step:

```
init → queryif(Companion,Tutor,Proof), A1
A1 → needanalysis(Tutor,Verificator,Proof), A2
A2, proof=validated → inform(Tutor,Companion,Analysis), end
A2, proof=invalidated → needanalysis(Tutor,{Tutor},Proof), A3
A3, proof=validated → inform(Tutor,Companion,Analysis), end
A3, proof=invalidated → needanalysis(Tutor,{Assistant},Proof), A4
A4, proof=validated → inform(Tutor,Companion,Analysis), end
A4, proof=invalidated → inform(Tutor,Companion,Fail), end
```

The request (the performative *query-if*) and the answer (the performative *inform*) are merged in one micro-protocol called *needAnalysis*.

3.5 Graphical Modeling Languages for Protocol's Representation

Like the SDL communication protocol description language [11], it is interesting interaction protocol designers have both a textual language in order to represent protocols and to make easier the use by tools and a graphical one which is more accessible for designers.

In our proposal of interaction protocol engineering, the textual language is CPDL (Communication Protocol Description Language) and the graphical one is GrCPDL (Graphical CPDL) and UAMLe. The GrCPDL language allows designers to graphically represent formulae in CPDL. The second language is more interesting. UAMLe is an extension of the FIPA graphical language UAML (Unified Agent Modeling

Language) [37]. We propose an extension in order to add, among others, the notion of exception.

Essentially two families of agent modeling languages have been used for representing agent interaction protocols[1]: one is Agent-UML [4] and the other is FIPA-UAML [37] (see Figure 4).

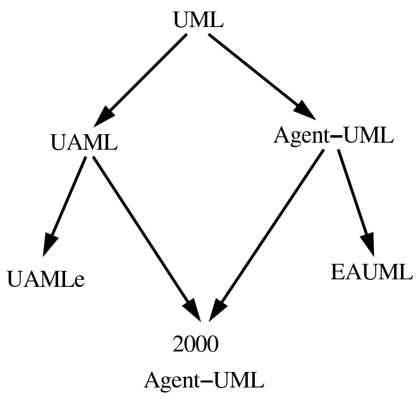


Figure 1.4. Graphical modeling languages for interaction protocols.

Both UAML and Agent-UML give birth to one extension: EAUML, which is an extension of Agent-UML and our proposal UAMLe [41]. All these proposals consider agents by their role in protocols and each message is sent from one role to one another. Moreover, they are all based on UML and UML sequence diagrams.

Our article [41] present a comparison between these four languages. Here are the criteria used for the comparison and the results of the comparison:

Roles: Agents are not represented by their name but according to their role within the interaction protocol. Such an approach enables one 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 further run-operation.

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 corresponding formal definition is provided. This may involve a translation of the description into finite state machines (FSM).

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. For this purpose one may rely on the SPIN, PROMELA model-checker [12].

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

Table 2 gives a synthetized 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 interaction protocol life cycle.

As the main point of this table, one can see that criteria about the features of languages are present but criteria on the design cycle are more or less present.

3.6 UAML and UAMLe Languages

UAML [37] is probably the first graphical language proposed (by FIPA) for representing agent interaction protocols. Most of the characteristics in UAML also appear in AUML: agents are denoted via their role, several types of message sending along with possible added constraints are allowed (synchronous, asynchronous, broadcast, repeated sending, 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.

The letters attached to the edges represent a cardinal value, e.g., the first edge indicates that m copies of the message are to be sent and n (or o or p) answers (n $\leq$ m) can be sent back and so on.

	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 reachability analysis and translation to PROMELA for model-checking
Protocol synthesis	No known algorithm for protocol synthesis	Code generation	No known algorithm for protocol synthesis	Code generation

Table 1.2. Some criteria for comparing interaction protocol modeling languages.

UAML represents alternatives in interaction states by means of boxes with 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*).

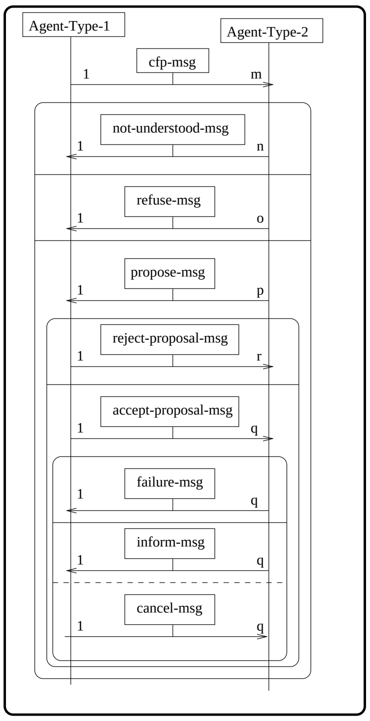


Figure 1.5. Contract Net protocol in UAML

Compared with UAML, UAMLe essentially enables synchronization of 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 [4] and with UAML [37]) 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 (see 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.

3.7 A Tool for Supporting Agent Interaction Protocol Design

We have developed a platform with a tool dedicated to helping design interaction protocols (DIP). This platform also contains a tool for validation (TAP) and a tool for conformance testing (CTP). DIP follows the component-based approach presented here above. As shown on Figure 2, such a tool enables to design and implement a protocol in a graphical manner by relying on micro-protocols and on the compositional language CPDL.

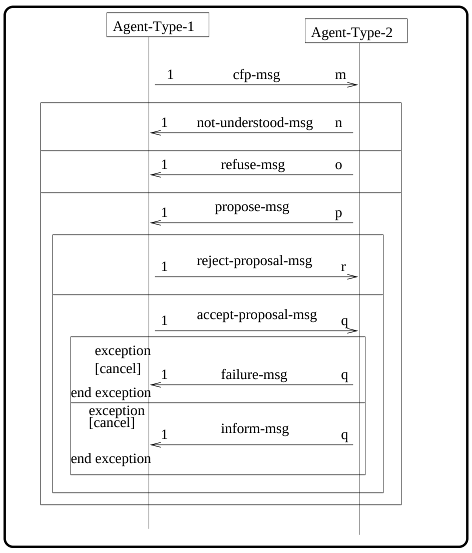


Figure 1.6. Contract net protocol in UAMLe.

Our platform is endowed with a true graphic editor that enables the definition of interaction protocols in the graphic language UAMLe (see Figure 2). Such a tool allows one (1) to build and (2) to 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 one to display a protocol in the alternate FIPA 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 protocol synthesis 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.

4. The Validation Step

Formal description of protocols does not avoid errors and some requested behaviors can be absent. Interaction protocol designers have two verification technique in order to check if good properties are present, if wrong properties have disappeared and if all the properties defined in the document of requirements are described: the first technique uses the notion of graph and defines the checked property as a property on a graph; it is the reachability analysis. The second technique defines a model of the protocol and a property defined by a temporal logic formula; it is the model-checking [5].

4.1 The Reachability Analysis

The validation stage checks whether a formal description of a protocol satisfies the protocol's requirements. Today the validation method mostly used both in academic and industrial settings is based on the accessibility test. Checking whether a protocol is accessible consists in checking that all of the protocol's states can be reached from the initial state. This comes down to generating a graph of all the states accessible from the initial state. One can thus verify a number of properties such as accessibility, deadlocks, livelocks, completion of a protocol, etc.

The downside of such a technique is that a state-space explosion may quickly be reached even with rather simple protocols if they make use of a significant number of variables with wide domains. For our platform, we have built a validation tool called TAP that is capable of verifying some independent properties on the protocol using CPDL. TAP also implements a translator that turns a CPDL specification into a PROMELA [5] specification in order to make use of a model checker [43].

The algorithm for the reachability analysis is the following:

1 Generate the accessibility graph

2 Define the property as a graph property

3 Check whether this is a property of the graph or not

The first step is automatic. An algorithm for the generation of the accessibility graph can be found in [5]. This algorithm is similar to a depth-first search in the protocol. As we use CPDL as formal language, we only provide a translation from CPDL into a graph. This algorithm is described in [44]. It is quite similar with the one proposed by Holzmann but, here, we have to consider the protocol's unfolding.

The second step requires one to translate the property into a graph property. For instance, one can translate the termination property as follows: if a node has no outgoing vertices, this node must be quoted as a final state. Then, one has just to check the property on the graph.

4.2 One Example

The example of agent introduction in a society is a protocol example where the new agent and the agent responsible for the introduction do not have the same protocol.

The initiator is the agent, which wants to enter the society. Here is its protocol:

```
init →connection(A, B, "connection"), end
```

and the micro-protocol *connection* is:

```
query-if(A,B,C).(inform(B,A,"accepted").inform(A,B,id) |
                                    inform(B,A,"refused"))
```

The initiator asks if it is possible to enter the society with the *query-if* performative. Then, the agent responsible for the introduction answers either accepted if the introduction is allowed or refused if the introduction is not possible. If the introduction is possible, the new agent sends its identity to the agent.

The agent responsible for the introduction has the following protocol:

```
init → connection(A, B, "connection").publish(B,C,"new agent"), end
```

The micro-protocol *connection* is the same as before. The micro-protocol *publish* only contains one performative *inform* in order to inform all the agents within the society of the appearance of a new agent.

Figure 7 gives the graph G1 for the protocol P1 which is the protocol for the initiator. The plus sign corresponds to a message sending and the minus sign to a message receiving.

Figure 8 gives the graph for the protocol P2 which is the protocol for the agent responsible for the introduction.

Then, one has to merge these two graphs into one graph. The resulting graph is the graph G2 since all the differences are in the protocol P2 and in the micro-protocol *publish*.

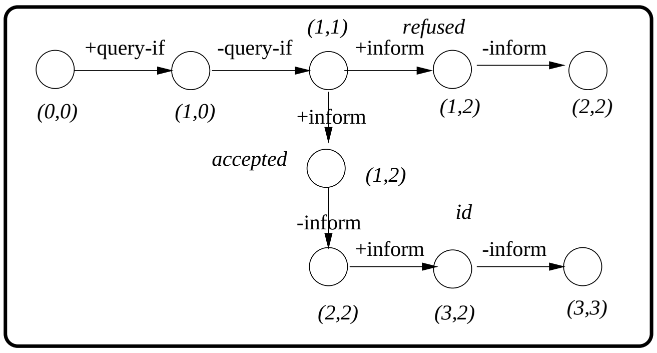


Figure 1.7. The reachability graph for the protocol P1.

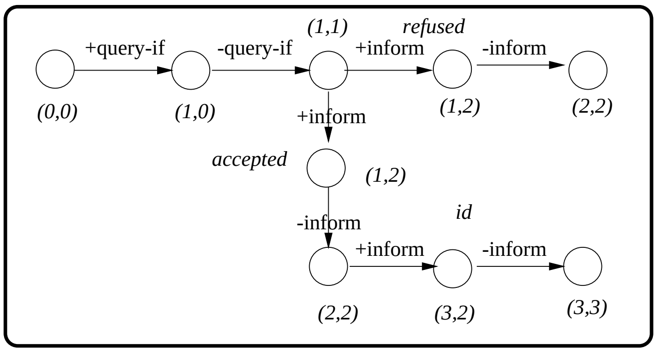


Figure 1.8. The reachability graph for the protocol P2.

From now on, one has to check the property. Some properties can be checked with our tool TAP if protocols are described in CPDL.

4.3 The Model-Checking Approach

The model-checking approach needs to build a model of the protocol. This model is the reachability graph of the protocol described during in the previous section. Model-checking is more complex than reachability analysis because in reachability analysis, one checks if the property is present or not and this property is described as a graph property whereas, here, the property is a temporal logic formula. Two temporal logics are available: linear temporal logic and branching temporal logic. Linear temporal logics (Propositional Temporal Logic PTL [45]) are used in order to check some structural properties like mutual exclusion and deadlocks where branching temporal logics (FML in Concurrent METATEM and CTL* [45]) check some functional properties which are dependent from the goal of the protocol.

For instance, D'Argenio et al. [46] present an example for a bounded retransmission protocol. This property is:

$$\forall \neg K.BAD \quad \text{and} \quad \forall \neg L.BAD$$

K and L are two bounded channel used between two processes in order to exchange data. This property corresponds to the validation that for no paths (for all squares); it is possible to have simultaneously the property BAD on both K and L. The property BAD corresponds to the capacity overhead of a channel.

There exists some works in interaction protocol validation in multiagent systems. Wen and Mizoguchi [47] use the model-checker SMV of Carnegie-Mellon. Lacey and DeLoach [49] [50] and Wei et al. [38] use the model-checker SPIN designed by Holzmann [12].

Our solution in our interaction protocol engineering is to translate the protocol in CPDL into a program in PROMELA and then use the model-checker SPIN [5]. The algorithm first translates the protocol in CPDL into a finite state automaton and then this automaton into a program in PROMELA. The tool TAP realizes these translations.

5. The Protocol Synthesis Stage

5.1 Phase Role

By now, a designer will have a formal description of a protocol and a validation of it. It is time to realize an operational version of such formal protocols. Describing an operational version of a protocol is called protocol synthesis or more traditionally in software engineering, implementation. This stage derives a program in a given programming language from a formal version.

5.2 Two Methods for the Protocol Synthesis

Two methods are proposed in our interaction protocol engineering. The first one (see section 2.1) is the classical approach used in communication protocol engineering: a program is defined from the formal version of the protocol. The second one (see section 2.2) is our proposal and the protocol is directly executed in CPDL language. This execution is realized by a mechanism called interaction module.

5.2.1 Protocol Synthesis Approach. The protocol synthesis approach is the classical one used in communication protocol engineering. Chu in his thesis [50] presents several algorithms in order to synthetize a program corresponding to the protocol. These algorithms are more or less automated and need more or less human actions.

The main problem with this approach is that it is not really obvious how to define a program having the behavior of the protocol. As Löffler and Serhrouchni noted in [51], interaction protocol designers can make some mistakes during the synthesis. Moreover, there is a big gap between the formal language and the programming language. Actually, if the formal language carries some notions, which can be added -with difficulty- in the programming language, it is easy to implement some wrong behaviors. For instance, Petri nets carry deadlines but a language like LISP does not consider deadlines. So, if designers want to translate a Petri net into a LISP program, there is more chance some errors will be inserted. Figure 9 presents the fact that it is obvious to have several versions of a protocol: a first version for the validation, a second one for the protocol synthesis and the last one for the conformance testing.

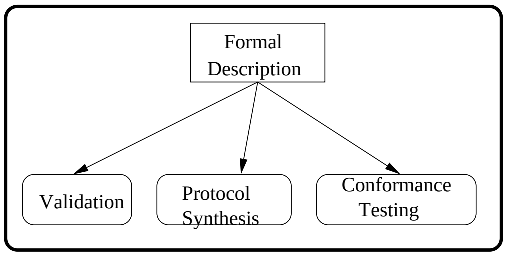


Figure 1.9. Classical approach for protocol synthesis.

The first step of this synthesis is the generation of a program skeleton from the formal version of the protocol. This skeleton corresponds to the transitions of the protocol. These transitions can be stored in an adjacency matrix. A second solution is to realize a program where each state of the protocol is represented by one condition statement *if* and for all transitions going out this state, a condition statement *if* is described.

For instance:

```
If (state == N1)
   If (message == a)
      perform something knowing a
   If (message == b)
      perform something knowing b
   If(message == c)
      perform something knowing c
```

The protocol synthesis algorithm recalls the depth-first search in graphs. Actually, the algorithm considers all the CPDL formulae in the protocol and then each micro-protocol for these formulae. From these micro-protocols, the algorithm takes all the performatives and builds the skeleton.

The skeleton can be divided into three parts: a first part for the initiator of the interaction, a second part for the agents which attend the interaction and a third part in order to build the medium in order to link agents.

The design tool DIP realizes the protocol synthesis and the skeleton is described in Java. DIP splits the code into four classes. A class for the initiator also called Server; a class for the attendees called Client. The class Protocol makes the management of the interaction and each transition from the graph is stored in a class Transition.

Some examples of protocol synthesis can be found in [44].

In order to manage the gap between the formal version and the programming language, we propose a second approach where a mechanism directly runs the CPDL code. This approach has some similarity with interpreted languages like LISP [52]. This second approach is described in the next section.

5.2.2 Direct Execution of Protocols in CPDL Language.

The main problem with the previous approach is that errors can occur during the translation from a formal version into a programming language or some features will be absent. A solution would be to delete the translation and to directly execute the protocol in CPDL language. So, if designers can directly run the protocol in CPDL, errors and lack of features disappear. This proposal is summarized in Figure 10, where one only finds one version for the validation and the protocol synthesis.

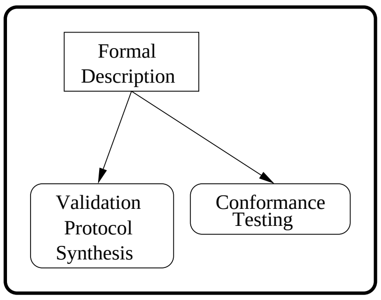


Figure 1.10. Direct run of protocols in CPDL.

Direct run of protocols in CPDL language recalls interpreted languages like LISP [52]. These languages need some compiler. Here, our compiler is called the interaction mechanism and it is stored in the interaction module. The interaction module is the architecture part of the agent, which manages interaction and protocols. The interaction module is composed by the interaction mechanism which unfolds protocols and two libraries for protocols and micro-protocols (see Figure 11).

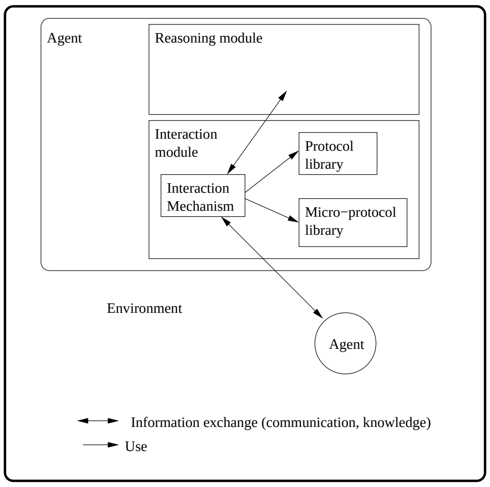


Figure 1.11. Interaction module for the management of protocols.

A protocol is composed of two parts: a first part, which is a finite state automaton, and a second part, which is the semantics, corresponding to the transitions. Since the interaction mechanism can handle all protocols in CPDL language, designers have just to define semantics for all these transitions. Actually, transition semantics depends on the beliefs of the agent so a "universal" interaction mechanism can have all the possible semantics.

A direct run of protocols can be found in [44].

5.2.3 Comparison of These Two Approaches. The protocol synthesis approach is automatic and designers only have to define the semantics for all the transitions. However, this synthesis can be expensive in lines of code as the number of states and transitions tends to increase. Direct run of protocols in CPDL decreases designer's effort if protocols use FIPA communicative acts. Actually, the interaction module manages yet the semantics of these communicative acts.

The protocol synthesis is a convenient approach since it is possible to reuse the program for simulation and for the conformance testing. However, the synthesis has to be re-made whenever the protocol is modified and designers have to insert the semantics again. Protocols in direct run approach can be easily modified since it is not necessary to re-do the synthesis of the protocol again.

In conclusion, protocol synthesis is an interesting approach once there are just few protocols and they do not have a lot of states and transitions. As protocols tend to increase, direct run of protocols may be better.

6. The Conformance Testing Stage

Previous stages in this proposal of interaction protocol engineering offer the possibility to formally design, to check some properties and to synthetize an operational version of the protocol. As noted by Löffler and Serhrouchni [51] and Holzmann [5], the translation from a formal description of a protocol into an operational one can insert errors and misconceptions and some features described in the specification of requirements can be absent. The conformance testing stage checks if the operational version of the protocol always has the properties defined in the specification of requirements and, of course, checks if wrong properties are absent.

The conformance testing stage differs from the validation one since the conformance testing checks properties on an operational version whereas the validation stage checks properties on a formal version of the protocol. The conformance testing algorithm is given in Figure 12. Designers have a formal description of the protocol (specification s in the figure), then they synthetize an operational version (implementation i) from this formal version which will be checked. A set of tests (Ts in the figure) is then defined from the specification s. The conformance testing corresponds to the application of these tests to the implementation. The protocol is proved if all the tests manage on the protocol.

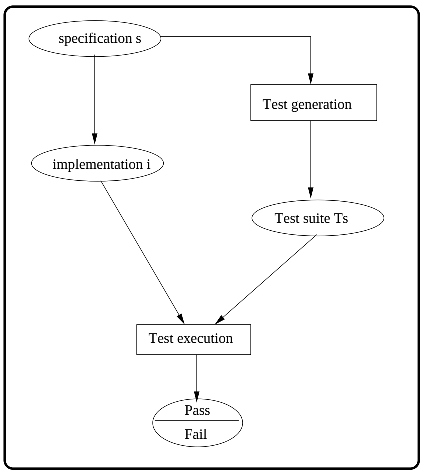


Figure 1.12. Conformance testing.

If one test fails, the designer has to find the problem and return to the design cycle in order to correct it. There exist two methods in order to define the tests: either designers have the formal version of the protocol and they define a set of tests that strictly match the protocol; or they only have the features of the protocol and, they only define a set of tests which partially fit the protocol. This latter case is frequent in the communication protocol domain where a protocol of Nth level can use protocols coming from *N-1*th level and these protocols are not provided with their formal definition.

We only consider designers having the formal version of the protocol in our explanations. The conformance testing is similar to the test stage in software engineering. Actually, in software engineering, one checks if when one gives data to a software, the behavior of the software is that expected. This is the same principle for interaction protocols. The algorithm looks for the checked state of the interaction (the state e1 in Figure 13, then applies some input data and checks if the output state of the operational version of the protocol corresponds to the one of the formal version.

Conformance testing needs to generate the accessibility graph. The algorithm is defined previously in the validation stage section (see section

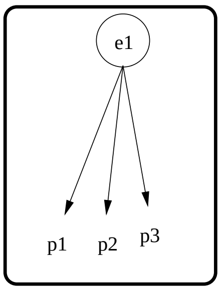


Figure 1.13. Conformance testing example.

D). When the graph is created, one has to apply the following algorithm proposed by [5]:

For all vertices j going out the state i, execute the three following steps:

1 Define the set of needed messages in order to reach the state i

2 Apply this set of messages to the operational version of the protocol and then apply the vertice j.

3 Check if the out state of the operational version of the protocol matches with the formal one.

7. Conclusion and Perspectives

Even though a number of suggestions concerning the engineering of interaction protocols have been proposed through this chapter, some work is still needed.

- The analysis stage leads to a document called the requirements that encompasses a set of different fields. It would be interesting now to provide designers with a set of guidelines that would help them fill out those fields.
- The formal description stage has been thoroughly detailed here. For now, there are three graphical languages; namely Agent-UML which is the merging of UAML and AUML, EAUML which extends Agent-UML, and our language called UAMLe which extends UAML. Both these extensions should be merged into Agent-UML, which should incorporate such notions as exceptions. Besides, one still needs to intrinsically incorporate Agent-UML within our approach and adapt the DIP tool by offering users the possibility to visually define their protocols via Agent-UML.

- The literature in communication protocol engineering attests that protocols are getting more and more complex, and therefore validating them has become a rather difficult task especially when dealing with the generation of a complete graph of accessible states. The current state of the art shows a number of solutions/algorithms to remedy this problem. It seems promising to apply such algorithms to interaction protocols. For instance, on-the-fly validation algorithms allow validation of a graph as it is being generated as well as the identification of cases where a functionality is absent.

  The validation stage detailed in this chapter emphasized a modular validation of protocols described in CPDL. The main example dealt with the protocol termination. One should investigate other properties and consider whether a micro-protocol's environment may have an influence on another therefore leading to the handling of contexts. Such an approach may be extended to model checking so as to obtain a modular model checking.

  The validation stage presented so far does not take into account the real function of an interaction protocol. It could be extended in this way. Besides the agents are not part of the validation process. In a more thorough approach they are.

- The interoperability test does not belong to our protocol engineering development but one should investigate what could be its meaning for interaction protocols and which tests must be really conducted. This could be performed at the level of ontologies or at the level of communication protocols between agents in order to obtain a common ground between all the agents.

- The last stage in the life cycle in software engineering has to do with maintenance. Such a stage does not belong to the interaction protocol engineering we are suggesting, neither does it belong to the communication protocol engineering. It would be appropriate to provide utility and quality measures upon the designed protocols and check afterwards how adequate is the obtained protocol.

  In a second step, one could have the agents modify their protocol themselves depending on their own needs. Since a protocol is a composition of micro-protocols, an agent could replace any of its micro-protocols by a more appropriate one and check whether the results are any better. One step even further, agents could build up their own micro-protocols, either according to the works on dialogism [53], or by making use of patterns in order to make new micro-protocols.

References

[1] Wooldridge, M., Agent-Based Software Engineering, *IEE Proc. Software Engineering*, 144, 26, 1997.

[2] Pitt, J., Anderton, M. and Cunningham, J., Normalized Interactions Between Autonomous Agents, in *Proc. International Workshop of the Design of Cooperative Systems*, Juan les Pins, 1995.

[3] Greaves, M.T., Holmack, H. and Bradshaw, J., CDT- A tool for agent conversation design, in *Proc. of 1998 National Conference on Artificial Intelligence (AAAI-98) Workshop on Software Tools for Developing Agents*, Madison, AAAI Press, 1998, 83-88.

[4] Odell, J., Van Dyke Parunak, H. and Bauer, B., Extending UML for Agents, in *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, Austin, Texas, Wagner, G., Lesperance, Y. and Yu, E., Eds., Icue Publishing, 2000.

[5] Holzmann, G., Design and Validation of Computer Protocols, Prentice-Hall, 1991.

[6] Lai, R. and Jirachiefpattana, A., *Communication Protocol Specification and Verification*, Kluwer Academic Publisher, 1998.

[7] Facchi, C., Haubner, M. and Hinkel, U., The SDL Specification of the Sliding Window Protocol Revisited, in *Technical ReportTUM-19614*, Technische Universität München, 1996.

[8] Schieferdecker, I., Abruptly Terminated Connections in TCP – A Verification Example, in *COST 247 International Workshop on Applied Formal Method in System Design*, Brezocnik, Z. and Kapus, T., 1996.

[9] Salomma, A., *Theory of Automata*, Pergamon Press, 1969.

[10] Jensen, K., *High-Level Petri Nets, Theory and Application*, Springer-Verlag, 1991.

[11] Turner, K.J., *Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL*, John Wiley and Sons, Ltd., 1993.

[12] Holzmann, G., The Model-Checker SPIN, *IEEE Transactions on Software Engineering*, 5, 23, 1997.

[13] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer, 1992.

[14] Koning, J.-L. and Huget, M.-P., Validating Reusable Interaction Protocols, in *Proc. The 2000 International Conference on Artificial Intelligence (IC-AI 00)*, Las Vegas, Nevada, Arabnia, H., Ed., CSREA Press, 2000.

[15] Huget, M.-P., Koning, J.-L. and Bergia, L., Une plate-forme pour l'ingénierie des protocoles et son application au projet de télé-enseignement Baghera, in *Proc. Systèmes multiagents : méthodologie, technologie et expérience, 8emes Journées Francophones Intelligence Ar-*

tificielle Distribuée et Systèmes Multi-Agents, JFIADSMA'00, Pesty, S. and Sayettat-Fau, C., Eds., 2000, 297-301. (In French)

[16] Cox, B., Tygar, J.D. and Sirbu, M., NetBill Security and Transaction Protocol, in *Proc. Of the First USENIX Workshop in Electronic Commerce*, 1995.

[17] Jacobson, I., Christerson, M., Jonsson, P. And Övergaard, G., *Object-Oriented Software Engineering: a Use Case Driven Approach*, Addison-Wesley, 1992.

[18] Barbuceanu, M. and Fox, M.S., COOL: A Language for Describing Coordination in MultiAgent System, in *Proc. First International Conference on Multi-Agent Systems (ICMAS95)*, San Francisco, AAAI Press, 1995, 17-24.

[19] Kuwabara, K., Ishida, T. and Osato, N., AgenTalk: Coordination Protocol Description for Multiagent Systems, in *Proc. First International Conference on Multi-Agent Systems (ICMAS95)*, San Francisco, AAAI Press, 1995.

[20] Kuwabara, K., Ishida, T. and Osato, N., AgenTalk: Describing Multiagent Coordination Protocols with Inheritance, in *Proc. Seventh IEEE International Conference on Tools with Artificial Intelligence,* Herndon, IEEE Press, 1995, 460-465.

[21] Ueno, I., Yoshida, S. and Kuwabara, K., A Multi-Agent Approach to Service Integration, in *Proc. Practical Applications of Intelligent Agents and Multi-Agent Systems (PAAM)*, London, 1997.

[22] Haddadi, A., *Communication and Cooperation in Agent Systems: A Pragmatic Theory*, Lecture Notes in Computer Science, 1056, Springer-Verlag, 1996.

[23] Burmeister, B., Haddadi, A. and Sundermeyer, K., Generic, Configurable, Cooperation Protocols for Multi-Agent Systems, in *Proc. Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, Neuchatel, Castelfranchi, C. and Müller, J.-P., Eds., Springer-Verlag, 1995, 157-171.

[24] Cost, R.S., Chen, Y., Finin, T. and Labrou, Y., Modeling Agent Conversation with Colored Petri Nets, in *Proc. Autonomous Agents'99 Special Workshop on Conversation Policies*, Bradshaw, J., Ed., 1999.

[25] Koning, J.-L., François, G. and Demazeau, Y., An Approach for Designing Negotiation Protocols in a Multiagent Systems, in *Proc. 15th IFIP World Computer Congress, IT&KNOWS Conference*, Vienna, Cuena, J., Ed., Chapman and Hall, 1998.

[26] Fisher, M., A Survey of Concurrent METATEM – The Language and its Applications, in *Proc. First International Conference on Temporal Logic (ICTL)*, Bonn, 1994.

[27] Fisher, M. and Wooldridge, M., Specifying and Executing Protocols for Cooperative Action, in *Proc. International Working Conference on Cooperating Knowledge-Based Systems (CKBS-94)*, Keele, 1994.

[28] Wooldridge, M., *Rational Agents*, MIT Press, 2000.

[29] Luck, M. and d'Inverno, M., Structuring a Z Specification to Provide a Formal Framework for Autonomous Agent Systems, in *Proc. ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users*, Bowen, J. and Hinchey, M., Eds., Lecture Notes in Computer Science, 967, Springer-Verlag, 1995, 47-62.

[30] d'Inverno, M. and Luck, M., Understanding Autonomous Interaction, in *Proc. 12th European Cinference on Artificial Intelligence (ECAI)*, 1996.

[31] d'Inverno, M., Kinny, D. and Luck, M., Interaction Protocol in Agentis, in *Proc. Third International Conference on Multi-Agent Systems (ICMAS98)*, Paris, Demazeau, Y., Ed., IEEE Press, 1998, 112-119.

[32] Koning, J.-L., AGIP: A Tool for Automating the Generation of Conversation Policies, in *Proc. 16th IFIP World Computer Congress, Intelligent Information Processing (IFIP-00)*, Beijing, Shi. Z., Ed., 2000.

[33] Iglesias, C., Garrijo, M., Gonzales, J. and Velasco, J., Design of Multi-Agent System Using MAS-Common KADS, in *Proc. of ATAL 98, Workshop on Agent Theories, Architectures and Languages,*Paris, Lecture Notes in Artificial Intelligence, LNAI 1555, 1998, 163-176.

[34] Singh, M., Towards Interaction Oriented Programming, in *Technical Report TR 96-15*, North Carolina University, 1996.

[35] Koning, J.-L. and Huget, M.-P., A Semi-Formal Specification Language Dedicated to Interaction Protocols, in *Frontiers in Artificial Intelligence and Applications*, Kangassalo, H., Jaakkola, H. and Kawaguchi, E., Eds., IOS Press, 2001.

[36] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[37] FIPA, Specification: Agent Communication Language, in *Manual*, 1999.

[38] Singh, M., On the Semantics of Protocols Among Distributed Intelligent Agents, in *Proc. IEEE Conference on Computers and Communication*, Phoenix, 1992, 1-14.

[39] Chow, C., Gouda, M.G. and Lam, S.S., A Discipline for Constructing Multiphase Communication Protocols, *ACM Transactions on Computer Systems*, 3, 4, 315, 1985.

[40] Smith, R.G., The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, *IEEE Transactions on Computer*, C-29-12, 1104, 1980.

[41] Koning, J.-L., Huget, M.-P., Wei, J. and Wang, X., Extended Modeling Languages for Interaction Protocol Design, in *Proc. of Agent-Oriented Software Engineering (AOSE-01)*, Montreal, 2001.

[42] Wei, J., Cheung, S.-C. and Wang, X., Towards a methodology for formal design and analysis of agent interaction protocols : An investigation in electronic commerce, in *Proc. International Software Engineering Symposium,* Wuhan, 2001.

[43] Clarke, E.M., Emerson, E.A. and Sistla, A.P., Automatic verification of finite-state concurrent systems using temporal logic specifications, in *ACM Transactions on Programming Languages and Systems*, 8-2, 244, 1986.

[44] Huget, M.-P., Une ingénierie des protocoles d'interaction dans les systèmes multi-agents, PhD Thesis, University of Paris Dauphine, 2001. (In French)

[45] Clarke, E.M., Grumberg, O. and Peled, D.A., *Model-Checking*, MIT Press, 1999.

[46] d'Argenio, P.R., Katoen, J.-P., Ruys, T. and Tretmans, J., The Bounded Retransmission Protocol Must Be On Time!, in *Proc. of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1217, 1997, 416-431.

[47] Wen W. and Mizoguchi, F., A Case Study on Model-Checking Multi-Agent Systems Using SMV, in *Technical Report,* Science University of Tokyo, 1998.

[48] Lacey, T. and DeLoach, S.A., Verification of Agent Behavioral Models, in *Proc. The 2000 International Conference on Artificial Intelligence (IC-AI 2000)*, Las Vegas, Arabnia, H., Ed., CSREA Press, 1998.

[49] Lacey, T. and DeLoach, S.A., Automatic Verification of Multiagent Conversations, in *Proc. Eleventh Annual Midwest Artificial Intelligence and Cognitive Science Conference*, University of Arkansas, 2000.

[50] Chu, P.-Y. M., Towards Automating Protocol Synthesis and Analysis, PhD Thesis, Ohio State University, 1989.

[51] Löffler, S. and Serhrouchni, A., Protocol Design: from Specification to Implementation, in *Proc. 5th Open Workshop for High-Speed Networks*, 1996.

[52] Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga, 1980.

[53] Chicoisne, G., Ricordel, P.-M., Pesty, S. and Demazeau, Y., Outils et Pistes pour la Pratique du Dialogisme entre Agents, in *Proc. 6èmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA-98)*, Pont à Mousson, Barthès, J.-P., Chevrier, V. and Brassac, C., AFCET AFIA, 1998, 163-176. (In French)