Validating Reusable Interaction Protocols

Jean-Luc KONING*
*MAGMA-LEIBNIZ
46, avenue Félix Viallet
38031 Grenoble cedex, France

Marc-Philippe HUGET*,**
**LAMSADE, Univ. Paris Dauphine
Place Maréchal de Lattre de Tassigny
75775 Paris cedex 16, France

Abstract *Protocol engineering in the domain of multiagent systems is still in its infancy. Some rare articles tackle this issue. In this paper we aim at drawing on protocol engineering from the domain of distributed systems to tackle the validation of interaction protocols. This paper puts forward the CPDL language as a means to formally represent protocols in a way that takes into account both their flexibility and reusability. It introduces two bridges, one between CPDL and linear logic in order to perform some structural tests, and one between CPDL and Petri nets in order to make use of already existing tools from this domain to run the simulations of protocols. We of course detail the CPDL formal description on a protocol example and run the validation on it.*

Keywords: Multiagent systems, interaction protocols, validation, formal specification.

1 Introduction

1.1 Flexible and Reusable Interaction Protocols in Multiagent Systems

In order to coordinate their actions agents in a multiagent system are to exchange information. Such interactions are usually guided by sets of rules called protocols. Not only do these interaction protocols constrain the possible messages they can either send or receive, but they also allow for the conveying of actions other agents have to follow.

Interaction protocols in multiagent systems make use of agent communication communication languages like KQML [7] and ACL [2] which are based on speech acts. Sending a message from one agent to another comes down to sending a performative (corresponding to a verb) whose contents represents the piece of information to communicate. For example, the performative *answer(NoGoods)* indicates an agent's reply (verb *answer*) telling there is no more goods (*contents*).

In multiagent systems interaction protocol designers need to adopt a protocol engineering approach even though there is no standard protocol engineering technique as for distributed systems yet. Nevertheless, such designers must be able to formally represent protocols in order to uncover any ambiguity that might lurk in the description due to its natural language nature [4].

The two main interaction protocol features a designer is interested in are *flexibility* and *reusability*, i.e., a protocol should easily be modified and its pieces should be seen as somewhat autonomous entities that could help build new ones.

As far as specifying interaction protocols several formalisms have been used such as finite state automata, Petri nets and temporal logic but as seen in [6] they hardly account for flexibility and reusability. Such formalisms do not enable to easily identify pieces of a protocol or modify it without undergoing a heavy protocol analysis.

Formal languages like LOTOS, Estelle or SDL which stem from distributed systems are not really used in multiagent systems. They do not provide any clear semantics for the objects that represent a piece of a protocol nor do they help derive a protocol's role. Furthermore, those formalisms lack *readability* thus limiting a protocol's reuse potential.

For all these reasons, we have put forward a formalism which takes into account the notions of flexibility and reusability, and defines a clear semantics. To tackle these issues we consider a component-based approach [6] and therefore do not see a protocol as a monolithic entity anymore

but as an aggregation of components, thus facilitating the substitution of a component for another. We call such components *micro-protocols*. They are formed by a set of performatives.

The formal language we defined to aggregate micro-protocols is called CPDL (*Communication Protocol Description Language*). It stems from linear logic and makes use of the notion of resource in order to take advantage of the temporary nature of knowledge as it is used in protocols. CPDL logical formulas may recall of transitions from finite state automata. CPDL also accounts for process synchronization by means of a *token* predicate, which handles a state marking in a similar way as in high-level Petri nets.

As there are predicates in predicate/transition Petri nets, CPDL is capable of conveying conditions on the use of a logical formula. Such a condition is made of a conjunction of first order predicates.

A CPDL well-formed formula looks like:

$$\alpha, \{b \in \mathcal{B}\}^\star, token(i)_{1 \leq i \leq n \in N} \mapsto \text{micro-protocol}^*, \beta$$

Where states α and β correspond to states prior and after the triggering of the transition.

1.2 Validating Interaction Protocols

In distributed systems, protocol engineering is composed of several ordered stages [4]: (1) the service specification stage defines the needs and properties a protocol should satisfy, (2) the design stage corresponds to an informal natural language description of a protocol, (3) the formal description gives a full description using a formalism, (4) the validation makes some structural tests, (5) the implementation is the coding of an executable description starting from a formal description, (6) the conformance test checks whether the protocol actually implements the properties previously stated, and (7) the interoperability test ensures that two processes using the same protocol are capable of communicating.

Along with the formal description the validation stage is the most important stage. It makes sure a protocol does not have any structural flaws such as the non-accessibility for some automata's states, deadlocks, livelocks, lack of termination, etc.

Protocol engineering in the domain of multiagent systems is still in its infancy. Some rare articles tackle this issue at large [12] [11], some only touch on the description stage [8], [10], or the validation stage [5]. We aim here at drawing on protocol engineering from the domain of distributed systems to tackle the validation of interaction protocols.

This paper puts forward the CPDL language as a means to formally represent protocols in a way that takes into account both their flexibility and reusability. Section 2 presents a protocol we are going to run the validation on and details its CPDL formal description. Section 3 introduces two bridges, one between CPDL and linear logic in order to perform the accessibility test, and one between CPDL and Petri nets in order to make use of already existing tools from this domain to run the simulations of protocols. Section 4 presents a reverse translation from Petri nets back to CPDL. This allows for the reuse of a huge Petri net library of previously defined protocols.

2 Example of a Protocol

Designing an interaction protocol comes down to first tackle its description, i.e., the definition of its goal, its needs and the properties it should satisfy.

2.1 Informal Description of a Protocol

2.1.1 Identification of a Protocol's Phases

Let us see an example based on a client/server type of dialog where the server and clients are agents. Such a protocol aims at allowing client agents to connect a server agent in order to collect some information. The server agent possibly asks several questions until it is up to answering the client agents' initial request. Then the client agents disconnect.

This brief description shows four phases in this protocol: (1) the logging in, (2) the request from a client agent and questions from the server agent, (3) the answer from the server agent, and (4) the logging out of a client agent. Figure 1 gives a Petri net representation of this protocol.

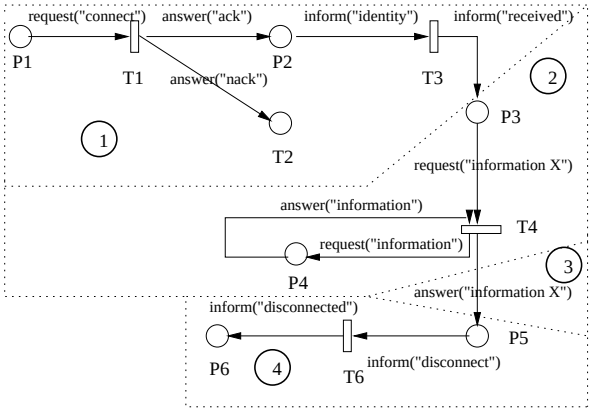

Figure 1: Petri net protocol of a client-server dialog.

The way these phases are delimited is arbitrary and may vary depending on the designers. Indeed, a designer may consider there are two sub-phases included in phase (2): the request made by a client agent and the questions asked by the server agent.

Phase 1 This phase deals with the logging in on the server agent. Client agents ask to log in on the server agent by means of a *connect* message. In this case, the server agent may either accept (and thus sends a *nack* message back to the client agent), or refuse (and thus sends an *ack* message back). When a client agent receives an *ack* message it sends its identity back to the server agent which answers with a *received* message.

Phase 2 This phase corresponds to a client agent's query and the server agent's request for additional information. When the server agent judges it does not have enough information to answer a client agent's question, they both enter into a question/answer loop in order to clarify the initial question. This stage comes to a close when the server agent considers it gained enough information to build its answer.

Phase 3 This is the phase where the server agent sends its answer to a client agent.

Phase 4 The final phase is the logging out. Client agents inform the server agent they want to log out by means of a *disconnect* message. The server agent sends back a *disconnected* message.

2.1.2 Remarks

This example is quite mundane, therefore the induced description of the four phases is rather short. For more complex protocols, the use of message sequence charts [9] may become necessary in order to specify the various message exchanges or the possible use cases [1].

In the domain of protocol engineering for distributed systems, the design stage deals with the various possible messages and introduces their semantics. We did it too here but we could also have gone a step farther in describing the used performatives.

2.2 Formal Description of a Protocol

As far as communication protocols, the description stage's purpose consists in formally describing protocols in order to uncover any ambiguity that might have been introduced by a natural language description. Let us now formally describe the protocol just seen:

$$\begin{array}{l} \text{init} \mapsto \text{login(A,B,C)}, A_1 \\ A_1 \mapsto \text{request(A,B,X)}, A_2 \\ A_2 \mapsto \text{sendinfo(B,A,X)}, A_3 \\ A_3 \mapsto \text{logout(A,B,C)}, \text{end}_2 \\ \text{end}_2 \end{array}$$

CPDL is a formal language that can readily be used by agents for their interactions. Even though such a textual language is suited for formally conveying protocols, a graphical language seems more appropriate for designers provided that the graphical representation could automatically be translated into CPDL. For this purpose we advocate the use of the graphical language UAMLe.

UAMLe [6] is an extension of UAML (Unified Agent Modeling Language) which is a graphical description method proposed by the international Foundation for Intelligent Physical Agents (FIPA) [2]. UAML enables to represent the different kinds of agents and the different message exchanges. Each interaction state is represented by a box containing all the possible choices for the next interaction, i.e., the possible messages that could be sent

from a current state. UAML allows for the representation of message sendings and broadcasting, messages with a predicate-based condition, deadlines, alternatives and loops.

UAMLe accounts for the micro-protocol concept which is absent from UAML. We have improved the notion of marks found in UAML in a similar way as that handled in high-level Petri nets where for some tokens the possibly received messages can be different. This means it is possible for messages received while in a particular state not to be of identical types. We also suggested that a loop be not restricted to one performative but deal with a set of micro-protocols all belonging to a same logical formula.

The CPDL specification of the protocol introduced in section 2.1 is a set of formulas whose initial state is denoted by *init* and the final states are denoted by end_i. Several final states may exist because several sequences of messages may take place within a single interaction.

As one can see, our protocol example encompasses four micro-protocols: *login*, *request*, *sendinfo*, and *logout*, that correspond to the protocol's four phases.

The login micro-protocol deals with (1) the server agent's announcement that a client agent is logged in—whose corresponding speech act is *answer("ack")*—, (2) a client agent's identification—whose corresponding speech act is *inform("identity")*—, and (3) the answer from the server agent telling it received the piece of information—whose corresponding speech act is *inform("received")*. This micro-protocol contains a decisional choice since it is possible for the server agent to refuse the login and stop the interaction by an *answer("nack")* message.

The request micro-protocol encompasses a *request* performative firstly for the *request("informationX")* speech act aimed at querying some information about *X*, and secondly for the *request("information")* speech act aimed at querying some additional information by the server agent and whose answer by a client agent is *answer("information")*.

The sendinfo micro-protocol only encompasses the *answer* performative that indicates the server agent's answer when querying a client agent for information.

The logout micro-protocol encompasses performatives *inform* telling that a client agent wants to log out and *inform* again which is the server agent's answer to let a client agent know it has been disconnected.

This protocol example highlights the relevance of a component-based approach for managing flexibility and reuse. Indeed, the login micro-protocol for instance tells that a client agent's identity is to be sent to the server agent which answers back that it received the piece of information. The identification process might turn out to be more heavy if for instance a client agent must provide several pieces of information corresponding to its identity and password. Modifying a protocol this way does not pose any problem with components since this means replacing an existing micro-protocol by a new one without having to change the general shape of the protocol. This shows the flexible nature of our approach.

The login process is present in numerous protocols. Since our interaction pieces are represented by micro-protocols, it is thus possible to readily reuse such micro-protocols in other protocols without any additional cost as far as code porting.

3 Interaction Protocol Validation

Validating a protocol has to do with the verification of a number of structural properties such as accessibility, deadlocks, livelocks, completion of a protocol, etc., and the overall validation of a protocol's model according to a particular formalism. A second aspect of validation has to do with a protocol's simulation in order to validate its behavior.

According to [3] it is easier to perform an accessibility test on a protocol when it is represented by a set of linear logic formulas. We therefore suggest to translate a CPDL specification of a protocol into linear logic in order to perform such a test (see section 3.1).

On the other hand, Petri nets have at their disposal a lot of well-proven tools for simulation and validation. It therefore turns out more promising to use such tools rather than provide new ones dedicated to our formalism. This means one ought to be able to translate a CPDL specification to an equivalent Petri net (see section 3.2).

We exemplify the validation stage on the protocol just described (see section 3.3)

3.1 Protocol 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. With finite state automata or Petri nets, testing a protocol's accessibility comes down to generating a graph of all the states accessible from the initial state. In linear logic, this test amounts to proving that all states are valid starting from an initial state.

In order to do this, let us provide an algorithm for translating a CPDL specification into its linear logic representation.

A micro-protocol is an aggregation of performatives. Therefore, a well-formed CPDL formula F does not lead to a unique transition between α and β but to as many transitions T_i as there are performatives within all of F's micro-protocols. For a formula F with a single micro-protocol that contains n performatives one obtains n developed formulas where states e_i are temporary states created by the development of the n formulas:

$$\begin{array}{|l} \alpha \mapsto \mathrm{perf}_1 e_1 \\ e_1 \mapsto \mathrm{perf}_2 e_2 \\ \vdots \\ e_{n-1} \mapsto \mathrm{perf}_n \beta \end{array}$$

Starting from these n formulas, one needs now to build the linear logic formulas. Initial state α and final state β are kept in the linear logic formulas; the first one being in a left member of the sequent, and the second one in a right member of the sequent. Performative p_i is inserted in the right member of the sequent. The guard in the left member of a CPDL logic formulas also appears in the left member of the corresponding linear logic formula under the form of a conjunction of resources.

In linear logic, tokens are handled through a multiplicative conjunction connector $\otimes$. If a protocol is in state α and expects n tokens one mentions this as $\alpha^{\otimes n}$ in the linear logic formula. Since CPDL and linear logic are two close formalisms, the conversion stage is quite simple. It is a rewriting of a base formula. p_i corresponds to a performative obtained after developing the micro-protocols.

An additional stage in the translation algorithm would be to apply the cut rule [3] in order to reduce the number of formulas. We will not discuss this here.

3.2 Protocol Simulation

Simulating protocols specified with CPDL can be accomplished via existing tools from the Petri nets domain. This therefore necessitates a translation stage from CPDL specifications into the Petri nets formalism.

3.2.1 Translation Algorithm

In an abstract fashion, for each of the protocol's formulas this translation algorithm creates a transition from the formula's initial state (i.e., α) to the final state (i.e., β) through a place T_i. This transition's label Pre(α, T_i) refers to the performative to be used whereas the label Post(β, T_i) is empty since there cannot be any double sending or receiving of a message.

If a logical formula contains several tokens then the resulting high-level Petri net must account for it. Therefore a marking is placed between α and T_i.

Of course the development of micro-protocols into CPDL is identical to what has just been seen in section 3.1.

3.2.2 Remarks

This algorithm does not pose any problem since it only consists in decomposing micro-protocols in order to form arcs in the Petri net. An additional stage could be to minimize the obtained Petri net in the end. Indeed, arcs with an ε label are factorized. Based on the transitions, it is possible to factorize the arcs.

This algorithm's complexity is $O(n)$ where n is the number of performatives used in all the protocol's micro-protocols.

3.3 Example of a Protocol Validation

The preceding sections showed different possible algorithms for translating protocols from a CPDL specification to a Petri net or linear logic set of formulas. Let us now validate the accessibility of the protocol presented in section 2. In order to do this, one first needs to translate a protocol into its linear logic equivalent and therefore develop all the micro-protocols prior to translating the obtained formulas into linear logic :

$$
\begin{array}{|l|}
\hline
\text{init} \multimap \text{request}(\text{"connect"}) \otimes A_1 \\
A_1 \multimap \text{answer}(\text{"ack"}) \otimes, A_2 \\
A_2 \multimap \text{inform}(\text{"identity"}) \otimes A_3 \\
A_3 \multimap \text{inform}(\text{"received"}) \otimes A_4 \\
A_1 \multimap \text{answer}(\text{"nack"}) \otimes end_1 \\
A_4 \multimap \text{request}(\text{"informationX"}) \otimes A_5 \\
A_5 \multimap \text{answer}(\text{"informationX"}) \otimes A_6 \\
A_5 \multimap \text{request}(\text{"information"}) \otimes A_7 \\
A_7 \multimap \text{answer}(\text{"information"}) \otimes A_5 \\
A_6 \multimap \text{inform}(\text{"disconnect"}) \otimes A_8 \\
A_8 \multimap \text{inform}(\text{"disconnected"}) \otimes end_2 \\
\hline
\end{array}
$$

Now that we have obtained the protocol in a linear logic formalism, let us show how to achieve the accessibility test on state *end*$_2$ starting from the initial state . Verifying the protocol's accessibility with linear logic amounts to proving state *end*$_2$ is valid when starting from state *init*. The proof goes as follows:

$\text{init}, \text{init} \multimap \text{request}(\text{"connect"}) \otimes A_1 \vdash$
$\text{request}(\text{"connect"}), A_1$
$A_1, A_1 \multimap \text{answer}(\text{"ack"}) \otimes, A_2 \vdash$
$\text{answer}(\text{"ack"}), A_2$
$A_2, P_2 \multimap \text{inform}(\text{"identity"}) \otimes A_3 \vdash$
$\text{inform}(\text{"identity"}), A_3$
$A_3, A_3 \multimap \text{inform}(\text{"received"}) \otimes A_4 \vdash$
$\text{inform}(\text{"received"}) A_4$
$A_4, A_4 \multimap \text{request}(\text{"informationX"}) \otimes A_5 \vdash$
$\text{request}(\text{"informationX"}), A_5$
$A_5, A_5 \multimap \text{answer}(\text{"informationX"}) \otimes A_6 \vdash$
$\text{answer}(\text{"informationX"}), A_6$
$A_6, A_6 \multimap \text{inform}(\text{"disconnect"}) \otimes A_8 \vdash$
$\text{inform}(\text{"disconnect"}), A_8$
$A_8, A_8 \multimap \text{inform}(\text{"disconnected"}) \otimes end_2 \vdash$
$\text{inform}(\text{"disconnected"}), end_2 \square$

Thus state end_2 is accessible from state *init*.

4 Protocol Reuse

Within our reuse approach we provide an algorithm for translating a Petri net representation of a protocol into a CPDL specification. Such an algorithm allows for the reuse of already existing protocols described by means of Petri nets and thus to reuse them by augmenting our CPDL protocols' library.

This algorithm is the reverse of the one presented in section 3.2. Instead of developing a protocol in CPDL, one factorizes it into a CPDL specification starting from its Petri net representation. Indeed, a set of performatives may correspond to a single micro-protocol, therefore all the formulas will be replaced by a single one.

The algorithm starts by transforming each of the arcs into logical CPDL formulas. In case of a high-level Petri net, each of the marks is inserted within the formulas by means of a *token* predicate and a number of token indicated by the Petri net's mark. The last stage consists in factorizing the whole protocol. One takes the various paths and verifies whether it is possible to replace a sub-path by an equivalent micro-protocol. The paths are made of the performatives used to go from the initial state to a current interaction state.

Later the algorithm inserts parameters and replaces the initial and final states of the Petri net by the CPDL *init* et end_i states.

5 Perspectives

On most interaction protocols such as the one presented in section 2.1 deadline and loop issues are not vital, but it may be important to allow for the specification of these notions in CPDL. Timed Petri nets deal with deadlines. Such a feature is helpful for stopping an agent from waiting indefinitely (e.g., when an agent is expecting an event from another agent does not want to send it or that does not exist). In a similar way CPDL makes use of a *time* predicate.

A complete CPDL well-formed formula, thus would be:

$\alpha, \{b \in \mathcal{B}\}^\star, loop(\bigwedge p_i, \gamma), time(i)_{1 \leq i \leq n \in N},$
$token(i)_{1 \leq i \leq n \in N} \mapsto$ micro-protocol$^*, \beta$

We have introduced here the *time* and *loop* predicates. The conjunction of predicates present in the loop parameters correspond to necessary conditions for carrying on. If this condition evaluates to false the protocol then proceeds to the state indicated in the second argument. In our example, the request micro-protocol encompasses a loop since the server agent's request can be repeated several times. When translating a CPDL specification into a Petri net in order to run the protocol's simulation, loops can easily be introduced by imposing their final state to be identical to their initial one. We are currently endowing the algorithms seen in this paper with both these new features loops and deadlines.

Even though the algorithms we have presented fulfill their purpose they do not fully take advantage of the modular nature of the protocols specified with CPDL. We are currently working on this issue in order to alleviate the overall validation stage once some of the micro-protocols have previously been validated even on different occasions, i.e., inside different protocols.

References

[1] D. Amyot, L. Logrippo, and R. J. A. Buhr. Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage. In G. Leduc, editor, *Conférence Francophone pour l'Ingénierie des Protocoles (CFIP-97)*. Hermes, 1997.

[2] FIPA. *Specification: Agent Communication Language*. Foundation for Intelligent Physical Agents, http://www.fipa.org/spec/fipa99spec.htm, September 1999. Draft-2.

[3] F. Girault, B. Pradin-Chézalviel, L. A. Künzle, and R. Valette. Linear logic as a tool for reasoning on a petri net model. In *Conference on Emerging Technologies and Factory Automation (ETFA-95)*, volume 1, pages 49–57, Paris, France, October 1995. INRIA/IEEE.

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

[5] J.-L. Koning, G. François, and Y. Demazeau. Formalization and pre-validation for interaction protocols in multiagent systems. In Henri Prade, editor, *13th European Conference on Artificial Intelligence (ECAI-98)*, pages 298–302, Brighton, UK, August 1998. John Wiley & Sons.

[6] J.-L. Koning and M.-P. Huget. A component-based approach for modeling interaction protocols. In H. Kangassalo and E. Kawaguchi, editors, *10th European-Japanese Conference on Information Modelling and Knowledge Bases*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2000.

[7] Y. Labrou and T. Finin. A proposal for a new kqml specification. Technical Report TR-CS-97-03, Computer Science and Electrical Engineering Dept, UMBC, 1997.

[8] F. J. Martin, E. Plaza, and J. A. Rodriguez-Aguilar. Conversation protocols: Modeling and implementing conversations in agent-based systems. In *Autonomous Agents'99 Special Workshop on Conversation Policies*, 1999.

[9] Ina Schieferdecker. Abruptly terminated connections in tcp - a verification example. In Z. Brezocnik and T. Kapus, editors, *COST 247 International Workshop on Applied Formal Method in System Design*, 1996.

[10] A. El-Fallah Seghrouchni and S. Haddad. A cooperation algorithm for multi-agent planning. In *European Workshop on Modelling Autonomous Agents and Multi-Agent World (MAAMAW-99)*, pages 86–, 1999.

[11] A. El-Fallah Seghrouchni and H. Mazouzi. Une démarche méthodologique pour l'ingénierie des protocoles d'interaction. In *Journées Francophones Intelligence Artificielle Distribuée et Systèmes Multi-Agents (JFIADSMA99)*. Hermes Sciences, 1999.

[12] M. P. Singh. Toward interaction oriented programming. In *Second International Conference on Multi-Agent Systems (ICMAS-96)*, Tokyo, Japan, December 1996.