A Component-Based Approach for Modeling Interaction Protocols

Jean-Luc Koning[*] **Marc-Philippe Huget**[*,**]
[*]MAGMA-LEIBNIZ [**]LAMSADE, *Univ. Paris Dauphine*
46, Avenue Félix Viallet *Place Maréchal de Lattre de Tassigny*
38031, Grenoble, France *75775 Paris cedex 16, France*
{Jean-Luc.Koning, Marc-Philippe.Huget}@imag.fr

Abstract. This paper deals with the modeling of interaction protocols for multiagent systems. The methods currently used rely on protocol description formalisms such as finite state automata, logic or Petri nets which are not adequate for reusing pieces of protocols. We outline here a component-based approach which reveals a number of advantages such as flexibility, abstraction and reuse. This approach brings into play components corresponding to a set of performatives. In order to aggregate those components, we introduce the CPDL language which we have derived from linear logic. We demonstrate the ease of use of this approach on the well-known Contract Net protocol example, and compare our model to similar approaches drawn from the fields of distributed systems and multiagent systems. Prior to concluding on the limits and possible extensions of this approach, we detail the software tool which supports the design and implementation of interaction protocols according to our formalism.

1 A Monolithic Approach

1.1 Introduction

Initially the multiagent community essentially focused on the development of systems. Conference proceedings reveal numerous applications. Up to now, rather few works had dealt with methodological aspects of multiagent systems. However, it seems this field matured lately and can now rely on its design and applications experiences in order to derive some methods. As an example, [2] deals with the development of multiagent systems and touches on the issue of genericity and reuse. Also [28] puts forward a method relying on a decomposition of a multiagent system into four domains such as the agent models, the world model, the world-agent interoperability, and the world-agent coordination model. One starts to witness a beginning in the formalizing of analysis and design approaches in the domain of multiagent systems.

With this in mind one deals here with the interactional aspects of multiagent systems. Usually, a multiagent system designer first defines the agents' architecture as well as the world (environment) they will live in. Then the designer applies to the agents an interaction mode they will use in order to communicate among them and achieve global tasks. This article should be seen in the context of an interaction oriented programming approach [32]. Such mode thinks out the interaction first and then focuses on the intentional structure of the agents.

The monolithic approach currently followed for modeling interaction protocols forces multiagent system designers to think in terms of the underlying architecture's details, the types of agents involved, and the application domain. However, a designer should normally be able to design acceptable interaction modes as independently as possible from the software entities that will use them [31]. Not only is this a tedious task, but the current process must be repeated from scratch whenever the system is to be adapted for any other application. Furthermore, no help is provided when systems implemented in different ways must be integrated.

From a pragmatic perspective, multiagent systems essentially rely upon three or four formalisms in order to describe interaction protocols: Petri nets [27], finite state automata [29], logic [14] or the Z specification language [11]. Not one allows for an easy identification and reuse of protocols or parts of them. Most of the time they force a designer to take up their work from the beginning.

Petri Nets

With Petri nets it is difficult to identify specific portions [5]. For instance, on figure 1 how would you isolate the portion that deals with the request of information? Designers must first look for the first performative that deals with a request, and then must identify the rest up to the achievement of the information (see doted area on figure 1). Moreover, with marked Petri nets [6] which are suited for managing process synchronization one may wonder how to reuse parts of a protocol since the handling of tokens leads to analyzing preceding parts.

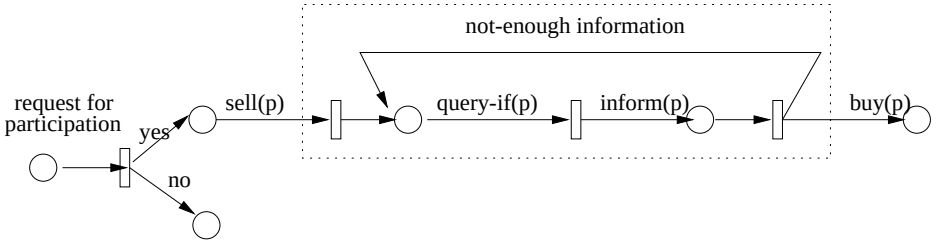

Figure 1: Petri net description of a protocol.

Finite State Automata

The problem is similar with automata. It is difficult to identify the various parts of the automaton shown on figure 2 at first glance. Let us remark it encompasses three parts a, b and c as delimited by doted lines.

Z Specification Language

The formal language Z [34] could also be considered because of its use for modeling multiagent systems and interaction protocols [11] but it does not allow for the synchronization of processes and above all, no algorithm is available for validating and verifying protocols. Furthermore, as underlined in [8], Z is not aimed at representing protocols: *"A criticism that has been leveled at this use of Z is that it is inappropriate for modeling interactions between agents."*.

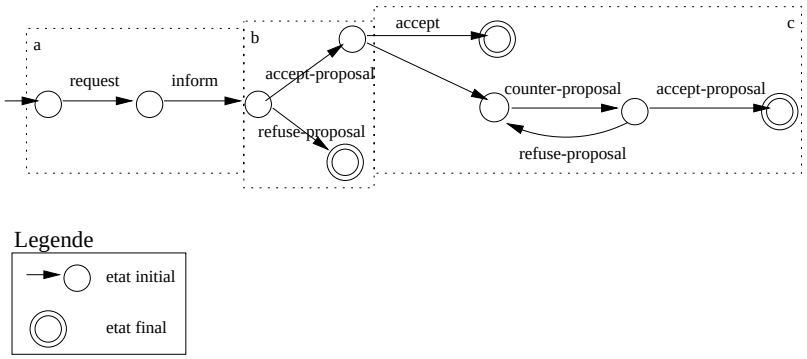

Figure 2: Automaton description of a protocol.

Logic

As for the two preceding formalisms, the logic formalism does not allow for an easy identification of the various parts of a protocol. For example, using Concurrent METATEM [13] to convey the Contract Net protocol [33] does not lead to a clear separation of the four stages that compose the protocol: call for proposals, proposal, contract, results (see table 1). Besides, since the logic formalism is closely related to the agents' cognitive architecture (of *Belief-Desire-Intention* type [23]) it is necessary to adapt the protocol according to the logic used by the agents.

start $\Rightarrow$ announce(task(Name, Desc, Req, self))
($\neg$ award(T,A)) $\mathcal{S}$ announce(T) $\wedge$ competent(A,T) $\Leftrightarrow$ possible(A,T)
possible(self,T) $\Rightarrow$ bid(T,self)
bidded(T,X) $\wedge$ most-preferable(T,Y) $\Rightarrow$ award(T,Y)
($\neg$ completed(T,self,X) $\wedge$ $\neg$ split(T,T1,T2)) $\mathcal{S}$ award(T,self) $\wedge$ ◉ result(T,R) $\Rightarrow$
completed(T,self,R)

Table 1: Temporal logic description of the *Contract Net* protocol [14].

As one can see, these formalisms do not allow for an easy reuse of protocols. Designers have to build interaction protocols from scratch.

1.3 Outline of the Article

This article presents a formalism based on components where a protocol is not an atomic element anymore but a combination of components that achieve a particular task. In such a framework of interaction protocol engineering based upon components, this paper describes how these components (called *micro-protocols*) are represented, as well as a language allowing their combination in order to make up of a whole protocol. Such a language is put forward since a component does not intrinsically embody any link to other components. A component does not hold any other component's name nor does it know how to call it. Such a language enables to handle generic and reusable components since it is not necessary to alter them prior to their using. The language proposed for the combination of micro-protocols has been derived from linear logic [18].

This paper is divided as such. Section 2 describes the motivations and advantages of a component-based approach for the design of interaction protocols in the domain of multia-

gent systems. Section 3 details the elements of this approach: the micro-protocols and the aggregation language used to combine them. In section 4 this model is applied to the well-known Contract Net protocol. Section 5 compares such an approach to similar ones taken from the distributed systems and multiagent systems domains. Section 6 presents a software tool which supports the design and implementation of interaction protocols according to our formalism. Finally, section 7 discusses the limits and possible extensions of this approach.

2 Contributions of a Component-Based Approach

The approach presented here is based on components[1]. A protocol is thus an aggregation of components, each of which having a particular role.

This research is motivated by a twofold acknowledgment:

- The design of interaction protocols moves more and more 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.
- Needs in interaction protocols match up. For example, only one part of the FIPA-Contract-net and the FIPA-iterated-Contract-net [12] 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 [31] 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:

1. Interactions between agents must be design from scratch every time.
2. 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.
3. Systems are designed independently from each other and cannot easily be integrated.
4. Updating or modifying a system in an elegant manner is impossible: an agent cannot be replaced by a new one.

We have identified five major characteristics a component-based approach might bring to the modeling of interaction protocols are: reuse, ease of construction, flexibility, abstraction, and facilitated verification.

1. **Reuse.** As far as the current protocol description methods, it is not possible to easily extract pieces from a protocol. This necessitates to look for the beginning and end of such pieces. Furthermore, it appears it is difficult to identify the exact semantics of these protocol pieces. By representing a protocol as a combination of components and providing a semantics for each of them, they can be reused in other protocols.

[1] A component is an identifiable part of a larger program or a construction. Usually, a component provides a particular function or set of functions. A component is in a certain state and its behavior is represented through attributes and methods [15].

2. **Ease of construction.** A protocol modeling process may largely be facilitated if one considers its components have inputs and outputs for data coming from or going to other components. Combining protocols may thus happen by linking the output of one component to the input of the next one. Concatenating a high-level Petri net to another one imposes to take into account and probably modify the marking (the way tokens are placed) in order to prevent some transitions from becoming not fireable. For example, if an initial Petri net only bears m tokens whereas n are awaited ($n > m$), it becomes impossible to cross such transition. The problem is identical with colored Petri nets where variable names must be altered.

3. **Flexibility.** With the current formalisms one cannot consider replacing a part of a protocol by another. For instance, in a high-level Petri net it is necessary to search for the place to be modified and to take the token into account since one cannot readily replace a portion of it by another. This poses some problems for firing transitions. The same is true with logic where it is necessary to modify the added piece in order for this piece to contain the appropriate variables and predicates. In a component-based approach, a protocol is more flexible since one just has to remove the component that does not suit and one replaces it by another one. The only constraint is to link the new component to the others already in place.

4. **Abstraction.** Following a component-based approach introduces a meta-level. A basic level corresponds to elements pertaining to a component (see section 3.1). A higher level conveys the global protocol in a more abstract fashion, i.e., a plan. The use of components hides implementation details which are useless for a protocol's understanding. With the current approaches, a protocol's global view cannot easily be perceived.

5. **Facilitating the verification.** We agree with the following idea [22]: *"A well, structured protocol can be built from a small number of well-designed and well-understood pieces. Each piece performs one function and performs it well. To understand the working of the pieces from which it is constructed and the way in which they interact. Protocols that are designed in this way are easier to understand and easier to implement efficiently, and they are more likely to be verifiable and maintainable"*. With components, it becomes easier to verify a protocol because (1) each component is less complex than a whole protocol and can be separately verified, (2) the global protocol is then verified by checking that the components are correctly linked together.

Let us show on an intuitive example how a component-based approach for designing interaction protocols facilitates their reuse. For the Petri net representation of a protocol given on figure 3, replacing the doted area by the portion given on figure 4 requires to modify the variables' name and change the tokens in order to produce the protocol shown on figure 5. On the other hand, for a same interaction protocol defined by means of components, one just have to replace component *bcde* by one component *bc* and two other components *de* (2) and *lm* (3), see figure 6.

3 Designing Interaction Protocols by Combining Components

As seen in section 2 a component-based approach shows a number of advantages especially as far as reuse.

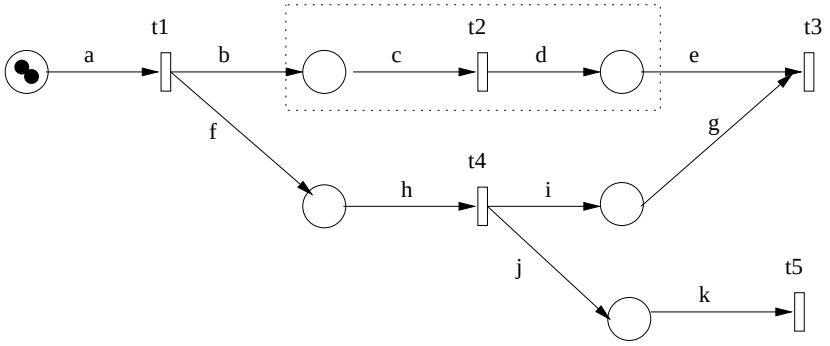

Figure 3: Example of an initial Petri net.

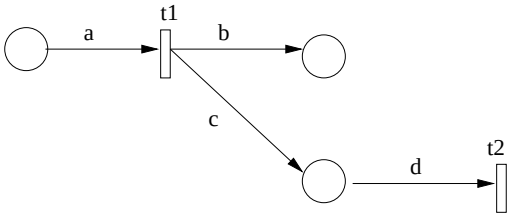

Figure 4: Piece of a Petri net to add.

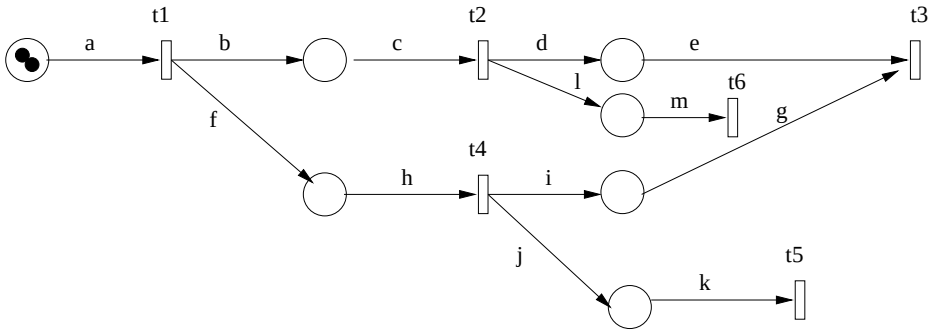

Figure 5: Resulting Petri net.

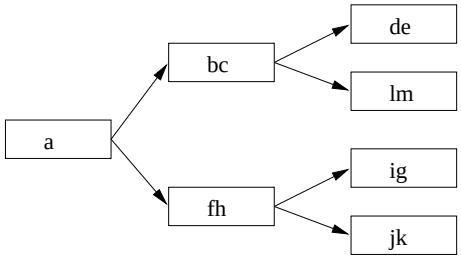

Figure 6: Equivalent alternative with a component-based approach.

In software engineering and more specifically in the domain of component engineering, one may distinguish the application level and the components level. Classically, components are aggregated in order to provide an application's features. Such decomposition is also present in our approach. At a global level, general interaction stages are defined. At a lower level the exhaustive set of components is given. Once combined those components convey the protocol's semantics. Our components are called *micro-protocols* since they represent an interaction sample, i.e., a protocol sample.

3.1 Definition of a Micro-Protocol

Micro-protocols are the basic components for designing protocols. They represent actions or particular tasks. They are composed of performatives which are the atomic elements of speech act theory [1]. The various categories of performatives may be found in [30]. In this paper, the performatives' semantics come from FIPA's proposals [12] for their inter-agent communication language ACL.

One may formally define micro-protocols and protocols by:

micro-protocol : : = performative | performative * micro-protocol protocol : : = micro-protocol | micro-protocol + protocol

where + here denotes the operator for aggregating micro-protocols and * denotes the operator for aggregating performatives.

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.

A micro-protocol's attributes are: its name, its semantics, its parameters' semantics and its definition. The *name* of a micro-protocol is unique. Since a micro-protocol's implementation is hidden, 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.

When making use of a micro-protocol it is necessary to know the semantics of all parameters since they are used in the constitution of messages.

The *definition* corresponds to the ordered set of performatives constituting a micro-protocol. Each performative is described along with its parameters like the sender, the receiver and the message's content.

As an example, a transmission micro-protocol could be defined by:

Name: transmission *Semantics of the parameters*: A: sender B: receiver C: content D: result E: receiver *Definition*: query-if(A,B,C).inform(B,A,D).inform(A,E,D) *Semantics*: transmission

Such micro-protocol aims at sending information from agent B to agent E via agent A. Agent A starts by asking for the value of a variable (parameter C) through the query-if performative. The result stored in variable D is sent back by agent B by means of an inform performative. Then agent A sends this result to agent E by means of an inform performative.

The semantics of sender and receiver indicates which is the parameter denoting the sender (agent A) or receivers of a message. Two receivers are defined here: B and E. In fact, the agents receiving performative query-if and inform are not the same. Agent A requests some piece of information from the first one in order to forward it to the second one.

Parameters C and D have two different semantics: content and result. In both cases, it is the content of the message being sent. The parameter with the result semantics is needed by the agent in case it wants to retrieve the message's content afterwards.

This micro-protocol is generic enough to be reusable. For example, this could be the case with a data server sending data from one agent to a group of agents via a broker, or with an intermediate agent A helping two agents B and E negotiate.

3.2 A Language for Aggregating Micro-Protocols

Combining micro-protocols into a general interaction protocol can be done with logic formulae encompassing the sequence of micro-protocols used. The relation between the micro-protocols' parameters should also be specified by telling 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.

In order to aggregate micro-protocols, we call into play the CPDL language (*Communication Protocol Description Language*) [24] that we derived from linear logic [16]. Linear logic can be justified this way [17]: "*Classical and intuitionistic logics deal with stable truths:*

if A and A $\Rightarrow$ *B, then B, but A still holds*

This is perfect in mathematics but wrong in real life, since real implication is causal. A causal implication cannot be iterated since the conditions are modified after its use; this process of modification of the premises is known in physics as reaction. For instance, if A is to spend 1 on a pack of cigarettes and B is to get them, you lose 1 in this process, and you cannot do it a second time."

This logic is based on sequents made of two members that use meta-variables, i.e., variables of propositional formulae that implicitly accept any substitution. In propositional logic sequents are composed of two members linked by a $\rightarrow$ connector. In linear logic the $\multimap$ connector separates those two members, such as the &, $\oplus$ and $\otimes$ connectors. For a thorough explanation, see [16].

CPDL keeps from linear logic the notion of resource and sequent. The reason we relied on linear logic to define this language is that (1) there is a direct equivalence between linear logic and Petri nets—we have shown how to go from Petri nets to CPDL and vice versa—and (2) the problem of accessibility in Petri nets can be readily checked with linear logic—a set of CPDL formula can straightforwardly be translated into linear logic formulae for accessibility tests. Proving that a Petri net is accessible amounts to proving that a premice M leads to a conclusion M' by applying some sequents. A designer may thus reuse previously built Petri nets by translating them into CPDL. Likewise, testing a protocol's accessibility can be accomplished by transforming a Petri net into a linear logic representation.

A protocol is a set of CPDL formulae and each formula contains zero (for τ-transitions) or more micro-protocols. A CPDL well-formed formula looks like:

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

A CPDL formula correspond 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

token. 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.

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

Tokens come from high-level Petri nets and from the notion of resources in linear logic. They are equivalent to post-conditions that need to be evaluated to true before going to the formula's final state. Let us suppose, an agent has to wait for n answers but only receives m messages ($n > m$), it has to wait until the $n - m$ more messages arrive before continuing the interaction. Obviously, the notion of token only makes sense when receiving (not for sending) messages.

States *init* and *end*$_i$ 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 [33] 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*).

There are two kinds of formulae in CPDL, the one just described and the *end* formula. This is a proper CPDL formula since linear logic neither deals with produced nor consumed resources. In CPDL states are resources. Therefore, an *init* resource is produced at the beginning. Then, it is consumed when firing the first formula thus producing a new resource corresponding to the formula's final state, and so on. The interaction ends whenever resources are lacking. This is the meaning of the *end* formula.

Here is the example of a simple protocol:

init $\mapsto$ query_if(A,B,C), A_1 $A_1 \mapsto$ inform(B,A,D), end end

This protocol makes use of FIPA performatives [12]. It shows that agent A asks one or several agents (B) concerning C, and agent B sends back that piece of information in parameter D. Due to lack of space, the protocol's execution algorithm is not described here; it is quite similar to the one provided in [3].

In our approach, we chose to develop and use a language derived from linear logic. [9] relies on the formal description language Z as in [10]. [14] relies on temporal logic. The formulae in these last two formalisms handle performatives but could also easily be adapted to handle micro-protocols. However one of the main advantages of linear logic is its direct equivalence with Petri nets. As opposed to linear logic, temporal logic does not embody the notion of resource. Therefore a premice which has been true in the past keeps on being true unless another formula invalidates it. There's no easy way to represent resource production and consumption with temporal logic. This notion of resource consumption is important with Petri nets as tokens can be consumed.

4 The *Contract Net* Protocol Example

In this section we apply our approach to model the Contract Net protocol [7]. Two kinds of agents are brought into play in this protocol: a manager and contractors. This protocol

encompasses four phases: task announcement, bidding, bid processing, and reporting results. A definition of the Contract Net according to our approach is:

init $\mapsto$ cfp(A,B,C), A_1
A_1, token(n) $\mapsto$ propose(B,A,D), A_2
$A_2 \mapsto$ perform(A,B,D), end
$A_2 \mapsto$ reject-proposal(A,B,D), end
end

Let us suppose Agent A has a manager role, i.e., it announces the task to be processed. It is first in the *init* state. The only formula containing this state is fired. This calls the cfp micro-protocol where cfp(A,B,T) means that a message with performative cfp is sent from agent A to (any) agent B concerning task T. In other words, agent A makes a task announcement and reaches state A_1.

On one hand, the agents denoted by B enter state A_1 when getting the message from A. As they are in a message sending stage they do not take into account the marking given by *token*. The only micro-protocol available is then propose, which refers to performative propose(B,A,P) meaning that the (B) agents bid, i.e., they submit proposal P to agent A. The meaning of these variables are kept throughout the micro-protocols by means of a shared space holding these data.

On the other hand, in state A_1 agent A is in a message receiving stage. It has to wait for n messages (indicated by *token(n)*) in order to fire the second formula. Unlike the Contract Net implementation given in [33], this example does not take any *timeout* into account. Agent A thus waits for all of the expected answers. Of course a deadlock may arise if at least one agent does not answer.

After sending their proposal, the (B) agents enter state A_2 and wait for A's answer. When agent A has received the n expected proposals, it chooses one among them and sends back one award using the third formula (the first formula starting at state A_2). The concerned micro-protocol (perform) is composed of performatives accept-proposal(A,B,D) and inform(B,A,R) where D is the proposal and R the result of this task. Agent A then waits for the task result from the contractee B.

Whether agents B have received an accept-proposal or a reject-proposal message, they either fire the first or second formula beginning with state A_2. In the first case, agent B performs the task and sends back a message with an inform performative to agent A and ends its interaction. In the second case, B stops its interaction.

The first formula corresponds to the first stage of the Contract Net, the second formula to the second stage and the next two formulae to the last two stages of the Contract Net.

With this example, one may notice how implementation details can be hidden. Let us suppose that one wants to extend this protocol and have agent A send its result to other agents. This only requires an extra micro-protocol *send* (containing performative inform). One just have to replace the *end* state by A_3 in the first formula beginning with state A_2 , and add the following formula: $A_3 \mapsto$ send(A,E,R), end.
The complete protocol then becomes:

init $\mapsto$ cfp(A,B,C), A_1
A_1, token(n) $\mapsto$ propose(B,A,D), A_2
$A_2 \mapsto$ perform(A,B1,D), A_3
$A_2 \mapsto$ reject-proposal(A,B,D), end
$A_3 \mapsto$ send(A,E,R), end
end

Adding a formula to a protocol and changing a final state as shown above highlights how easy it is to build or extend protocols. This usually requires to add some formulae and give the right variable names when necessary.

5 Related works

Component-based approaches can also be found in other works both in distributed and multiagent systems.

FT-Linda

FT-Linda [19] is a specialized version of the Linda coordination language [4] where fault-tolerant application features have been added (Fault-Tolerant Linda). Linda is a language intended for ensuring the coordination of parallel applications where each process communicates with the others via some shared memory called a *tuple space*. It is a communication abstraction defined as a set that can hold data elements called tuples. FT-Linda is used to design communication protocols in distributed systems. Components in FT-Linda are also micro-protocols which are objects representing behaviors to be triggered in case of events occurring during the application.

As opposed to our approach, FT-Linda does not provide any real implementation of protocols since those micro-protocols run concurrently. Moreover, they do not encompass any semantics. Thus, designers have to analyze them prior to reusing them. Finally, designing protocols with FT-Linda necessitates the taking into account of variable names used in the other micro-protocols. This limits any possible reuse and do not ease the design process.

COSY

Works on communication and cooperation in COSY [21] have led to view a protocol as an aggregation of primitive protocols. Each primitive protocol can be represented by a tree where each node corresponds to a particular situation and transitions correspond to possible messages an agent can either receive or send, i.e., the various interaction alternatives.

Our micro-protocols are similar to these primitive protocols but they include decision making at the formulae level. Let us note that our protocol execution algorithm is very close to COSY's [3]. For example, on figure 7, a primitive protocol may choose two alternatives once a command performative has been used, i.e., either informing(Reject) or informing(Report).

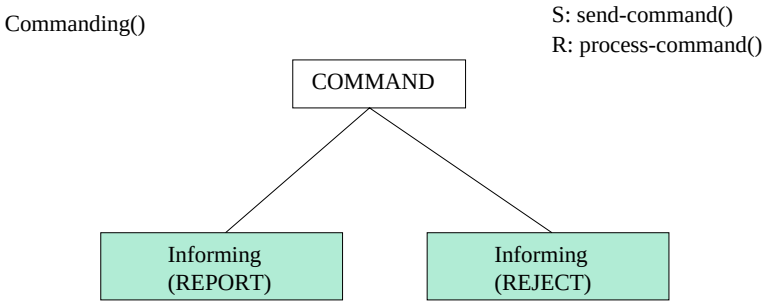

Figure 7: Primitive Protocol Commanding.

As far as the semantics, COSY takes a different stand. A primitive protocol's semantics is derived from its first message. It is thus impossible to differentiate between two primitive protocols containing a same first performative. Provided you have the primitive example shown on figure 7, COSY does not allow you to also have a primitive protocol starting with Command and offering three choices like informing(Reject), informing(Report) and

informing(Not-Understood). Both these primitive protocols cannot coexist since according to COSY they each have a commanding semantics.

AgenTalk

In AgenTalk [26], protocols inherit from one another. They are described as scripts containing the various steps of a possible sequence of interactions. Beliefs are embedded into scripts too.

Compared to our approach, AgenTalk does not allow for an efficient handling of abstraction (see section 2) since a script can be linked to pretty much any state of any other script. There is no unique state dealing with the input or output of data. Therefore abstraction is limited because scripts have to carefully be analyzed prior to being aggregated. Moreover, flexibility is also limited because for any particular script's state one has to find whether it is possible to hook it up to the next script's state. Finally, AgenTalk's scripts do not handle any semantics.

General Remarks

On one hand, COSY's [20] or AgenTalk's [25] modeling of the Contract net (see figures 8 and 9 respectively) resemble the one presented in this paper (see section 4). On the other hand, the first two approaches appear not to facilitate possible modifications. These approaches are implemented in LISP, thus leading to a great amount of code. For example, AgenTalk's script for the Contract Net contractor part is much more complex than the five formulae based on the four micro-protocols[2] given in section 4. With our model, provided designers have access to all the micro-protocols definition, they are capable of understanding the unfolding of a whole protocol.

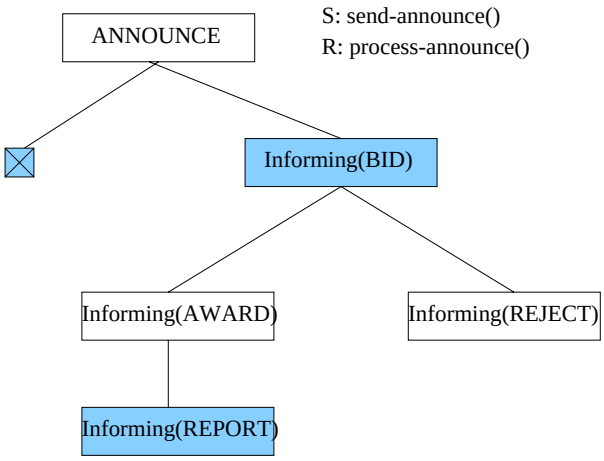

Figure 8: Contract Net Protocol in COSY.

Among the three approaches just discussed, COSY is the closest to ours even though it handles the protocols' semantics in a rather restricting way (i.e., it is defined according to the first performative of a primitive protocol). Furthermore, COSY does not handle beliefs at the protocol level.

[2] One of them only amounts to an *end* formula.

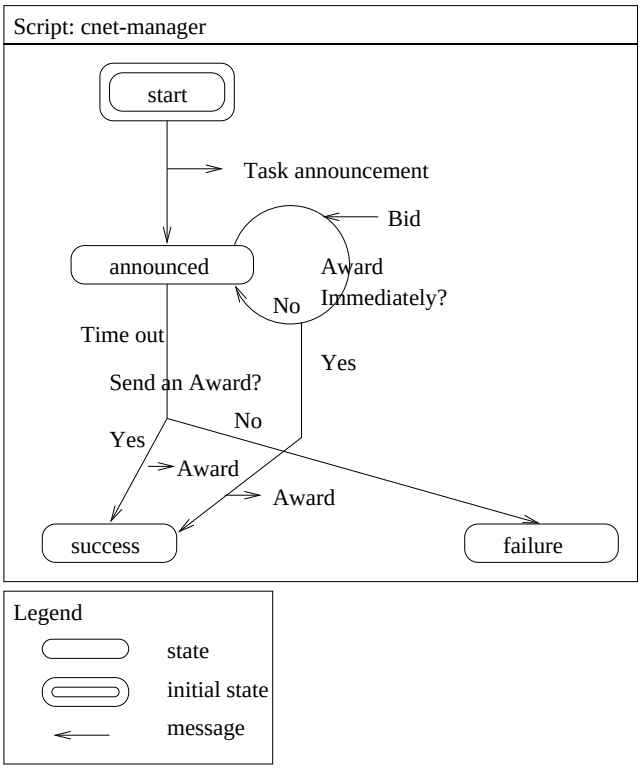

Figure 9: Contract Net Protocol in AgenTalk.

6 A Tool for Designing Interaction Protocol

This section outlines a software tool called DIP (*Designing Interaction Protocols*) we have developed to help designers model interaction protocols according to our component-based approach. This tool enables to graphically design and bring into play a protocol using micro-protocols as well as the aggregation language CPDL (see figure 10).

The underlying graphical description method is UAML$_e$, an extension of UAML (*Unified Agent Modeling Language*) [12], which we developed in order to be able to handle the concept of micro-protocols [24]. UAML enables to represent the various kinds of agents and the different message exchanges occurring during an interactional action. Each interaction state is represented by a box containing all the possible alternatives for the interactions to follow, i.e., the possible messages that could be sent from the current state. As exemplified on figure 11, once the request-msg message has been sent, the three possible messages could be agree-msg (which in turns leads to two new possible choices), refuse-msg or not-understood-msg.

The protocol described on figure 10 shows four arrows labeled with two micro-protocols along with their parameters (micro-protocols are written inside boxes). Two of these CPDL formulae have beliefs attached to them (between square brackets). All of them have a marking whose number of expected tokens is given in a circle on the right hand side. The boxes found in a UAML notation are shown here as well. All CPDL features are present.

With DIP a designer may choose to create a protocol from scratch or start from a pre-existing one. The protocol's graphical representation is given in the UAML$_e$ notation. Five functions may be activated (left hand side of figure 10). The `Add` function allows for the adding of a formula to the current protocol. The designer has thus to fill out fields such as the initial and final states, and may provide an ordered set of micro-protocols along with their pa-

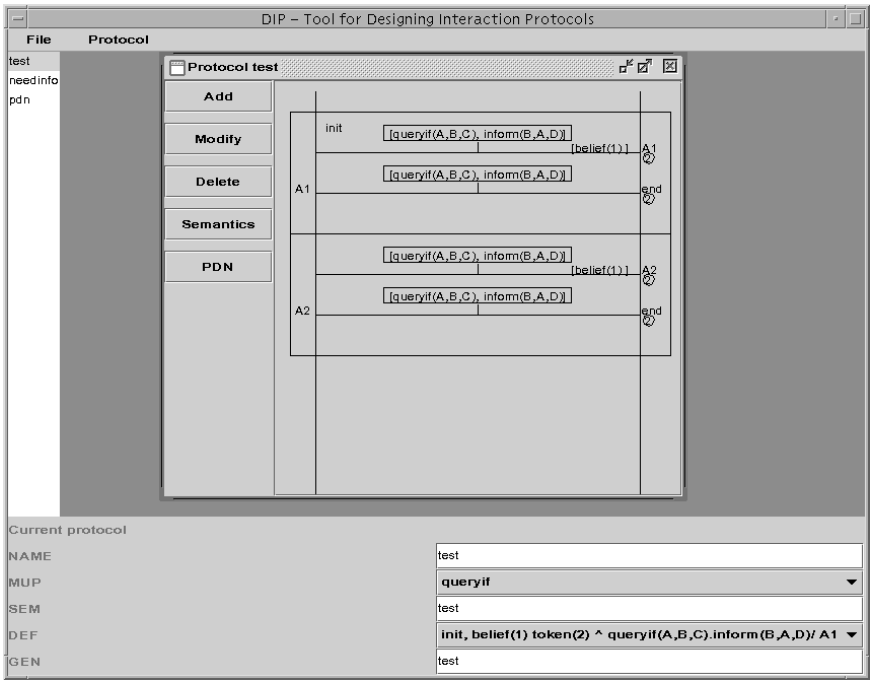

Figure 10: Tool for Designing Interaction Protocols.

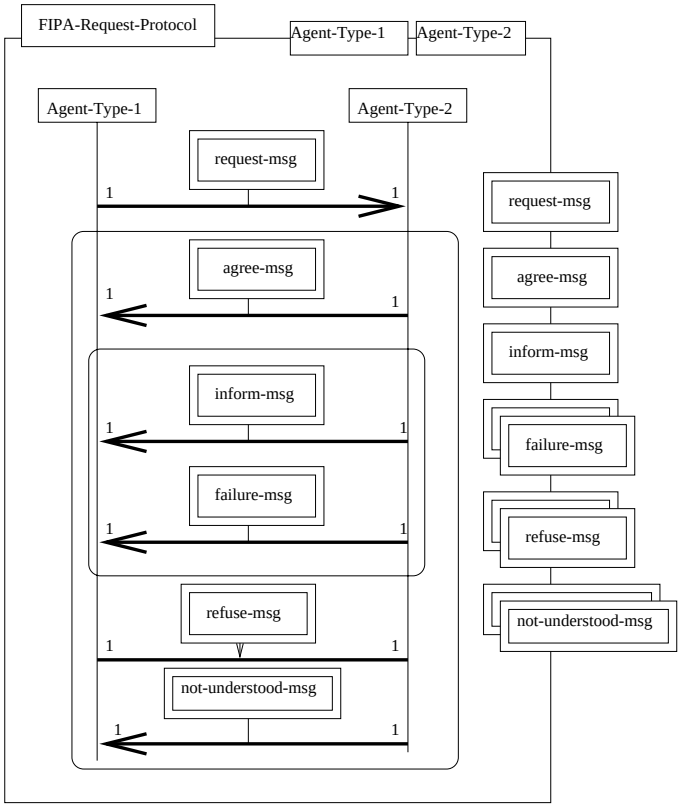

Figure 11: Request protocol in UAML notation.

rameters, beliefs and tokens. The `Modify` function allows for the modification of a formula. The designer chooses a formula and modifies its fields. The `Delete` function allows for the deletion of a given formula from the list of formulae. The `Semantics` function allows for the modification of the protocol's semantics. Finally, the `PDN` function allows the protocol to be displayed using the alternate FIPA's notation called the *Protocol Description Notation* (PDN) [12]. Unlike UAML and UAML$_e$, 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 of the interaction (see figure 12).

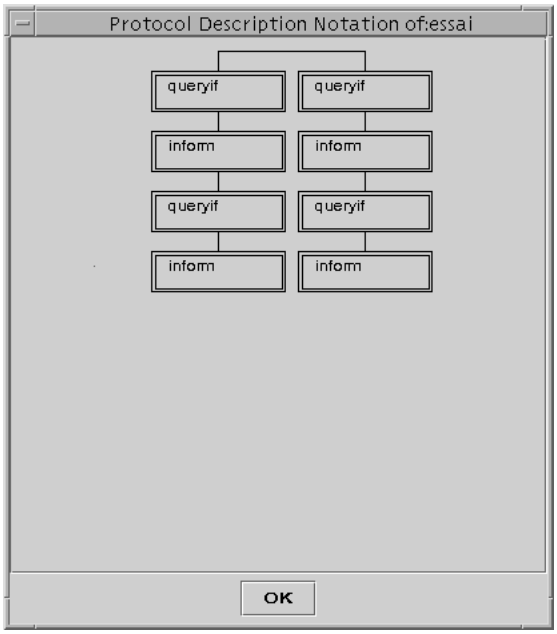

Figure 12: An example of a protocol description in PDN.

The DIP software also provides a translation of a high-level Petri net representation of a protocol to a set of CPDL formulae as there is a direct equivalence between linear logic (which CPDL inherits from) and high-level Petri nets [24]. Furthermore, this tool supplies information about the protocol (see figure 10) such as: (1) its name, (2) its set of micro-protocols, (3) its semantics, (4) its set of CPDL formulae, and (5) the generic protocol used for creating this protocol. Indeed, in a reuse setting, it is possible to model a new protocol starting from a generic one.

Eventually, all micro-protocols and protocols are stored as text files. The data corresponding to the protocol presented on figure 10 is the following:

```
NAME: test
MUP: queryIf, inform.
SEM: test
DEF:
init, belief(1) token(2) ^ queryif(A,B,C).inform(B,A,D)/ A1
init, token(2) ^ queryif(A,B,C).inform(B,A,D)/ A2
A1, token(2) ^ queryif(A,B,C).inform(B,A,D)/ end
A2, token(2) ^ queryif(A,B,C).inform(B,A,D)/ end
end
GEN: test
```

The query-if micro-protocol is encoded as:

```
NAME: queryIf
SEMPARAM:
A:sender,
B:receiver,
C:content.
FUNC: query-if(A,B,C)
SEM: query-if
```

Field names correspond to the ones used for micro-protocols as presented in section 3.

7 Concluding Remarks

7.1 Summary

Current approaches for modeling interaction protocols rarely take reuse into account. Used formalisms are based upon Petri nets, finite state automata among others. These techniques do not enable to identify and easily handle pieces of protocols.

The approach presented in this paper is inspired by software engineering where applications are built in a modular fashion. A multiagent system designer may then model interaction protocols by combining components. Such an approach bears some advantages such as reuse as well as ease of construction.

In the framework of interaction protocol engineering a component is a micro-protocol that contains a set of performatives and that is described as semantics doted entities. Such micro-protocols are formally linked together by means of the CPDL language which stems from linear logic. This model has then been applied to the well-known Contract Net protocol example. A comparison of our approach against other existing ones taken from the fields of distributed and multiagent systems has also been conducted. Finally, the tool we have developed for facilitating the design of interaction protocols according to this component-based approach has been described.

7.2 Future Work

This research work may be extended in several directions. First of all, micro-protocols could be executed in a less linear fashion. Up to now, micro-protocols represent a sequence of performatives, which imposes a strong constraint to their unfolding. We work on endowing micro-protocols with decision making and try to endow them with beliefs. For example, in the Contract Net protocol version presented in section 4, two formulae have been created from state A_2 in order to handle the performatives accept-proposals and reject-proposal. Had one embedded decision making and beliefs capabilities within micro-protocols, these two formulae could have been merged into one with two execution branches: accept-proposal.inform and reject-proposal.

A second direction deals with the implementation of time management in the micro-protocol aggregation language. The Contract Net protocol version of our example is limited. Actually, the agent in charge of getting the bids is waiting for all answers since no timeout is given. We are in the process of inserting predicates in the logic formulae in order to get rid of this type of difficulties.

A third research direction has to do with the validation of protocols defined by components. On one hand, we are proposing new algorithms capable of taking into account this component-based architecture. One of the advantages of the component-based approach is the greater ease in protocol validation (see section 2). This advantage can be found again

in the verification algorithm since in order to validate a protocol the designer comes first to validate the micro-protocols and second the connection between these micro-protocols. This reduces the complexity of the protocol validation phase. On the other hand, we are currently hooking up our software to verification and simulation tools in order to possess a complete platform for developing interaction protocols when designing multiagent systems.

References

[1] J. L. Austin. *How to Do Things with Words*. Clarendon Press, 1962.

[2] F. M. T. Brazier, C. M. Jonker, and Jan Treur. Principles of compositional multiagent system development. In José Cuena, editor, *15th IFIP World Computer Congress, IT&KNOWS Conference*, Vienna (Austria), Budapest (Hungaria), September 1998. Austrian Computer Society, Chapman and Hall.

[3] Birgit Burmeister, Afsaneh Haddadi, and Kurt Sundermeyer. Generic, configurable, cooperation protocols for multi-agent systems. In C. Castelfranchi and Jean-Pierre Muller, editors, *From Reaction to Cognition*, volume 957 of *Lecture notes in AI*, pages 157–171, Berlin, Germany, 1995. Springer Verlag. Appeared also in MAAMAW-93, Neuchatel.

[4] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communication of the ACM*, 35(2):97–107, February 1992.

[5] R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. Modeling agent conversation with colored petri nets. In Jeff Bradshaw, editor, *Autonomous Agents'99, Special Workshop on Conversation Policies*, May 1999.

[6] René David and Hassane Alla. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice Hall Int., Hertfordshire, UK, 1992.

[7] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem-solving. *Artificial Intelligence*, 20:63–109, 1983.

[8] Mark d'Inverno, Michael Fisher, Alessio Lomuscio, Michael Luck, Maarten de Rijke, Mark Ryan, and Michael Wooldridge. Formalisms for multi-agent systems. *The Knowledge Engineering Review*, 12(3), 1997.

[9] Mark d'Inverno, David Kinny, and Michael Luck. Interaction protocols in agentis. In Yves Demazeau, editor, *Third International Conference on MultiAgent Systems (ICMAS-98)*, pages 112–119, Paris, France, July 1998. IEEE.

[10] Mark d'Inverno, David Kinny, Michael Luck, and Mike Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV: Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, volume 1365 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 1997.

[11] Mark d'Inverno and Michael Luck. Formalising the contract net as a goal-directed system. In Walter Van de Velde and John Perram, editors, *Agents Breaking Away, MAAMAW 96*, number 1038 in Lecture Notes in Artificial Intelligence, pages 72–85. Springer-Verlag, 1996.

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

[13] M. Fisher. A survey of concurrent metatem — the language and its applications. In *First International Conference on Temporal Logic (ICTL)*, Bonn, Germany, July 1994.

[14] M. Fisher and M. Wooldridge. Specifying and executing protocols for cooperative action. In *International Working Conference on Cooperating Knowledge-Based Systems (CKBS-94)*, Keele, 1994.

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

[16] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987. North-Holland.

[17] J.-Y. Girard. Linear logic: Its syntax and semantics. *Advances in Linear Logic*, pages 1–42, 1995.

[18] 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.

[19] O. Guedes, D. Bakken, N. Bhatti, M. Hiltunen, and R. Schlichting. A customized communication subsystem for ft-linda. In *Brazilian Symposium on Computer Networks*, 1995.

[20] Afsaneh Haddadi. Towards a pragmatic theory of interaction. In *First International Conference on MultiAgent Systems (ICMAS-95)*, pages 133–139, San Francisco, June 1995. AAAI Press.

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

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

[23] D. Kinny, A. Rao, and M. Georgeff. A methodology and modelling technique for systems of bdi agents. In W. Van de Velde and J. Perram, editors, *Agents Breaking Away: 7th Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)*, volume LNAI 1038, pages 56–71, Eindhoven, The Netherlands, 1996. Springer-Verlag.

[24] Jean-Luc Koning and Marc-Philippe Huget. Validating reusable interaction protocols. In Hamid Arabnia, editor, *The 2000 International Conference on Artificial Intelligence (ICAI-00)*, Las Vegas, NV, June 2000. CSREA Press.

[25] Kazubiro Kuwabara, Toru Ishida, and Nobuyasu Osato. AgenTalk: Coordination protocol description for multiagent systems. In *First International Conference on MultiAgent Systems (ICMAS-95)*, San Francisco, June 1995. AAAI Press. Poster.

[26] Kazubiro Kuwabara, Toru Ishida, and Nobuyasu Osato. AgenTalk: Describing multiagent coordination protocols with inheritance. In *Seventh International Conference on Tools with Artificial Intelligence*, pages 460–465, Herndon, Virginia, November 1995.

[27] R. Lee. Distributed electronic trade scenarios: representation, design, prototyping. *International Journal of Electronic Commerce*, 3(2):105–136, 1999.

[28] H. J. Müller. Towards agent systems engineering. *Data and Knowledge Engineering*, 23:217–245, 1997. Elsevier, North-Holland.

[29] Tuomas Sandholm and Victor Lesser. Issues in automated negotiation and electronic commerce: Extanding the contract net framework. In *First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 328–335, San Francisco, USA, June 1995. AAAI Press.

[30] J. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, 1969.

[31] Munindar P. Singh. On the semantics of protocols among distributed intelligent agents. In *IEEE Conference on Computers and Communication*, pages 1–14, Phoenix, USA, 1992.

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

[33] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.

[34] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.