Computational Intelligence, Volume 18, Number 2,

TO COMMIT OR NOT TO COMMIT: MODELLING AGENT CONVERSATIONS FOR ACTION

Roberto A. Flores and Robert C. Kremer
Department of Computer Science, University of Calgary, AB, T2N 1N4, CANADA

Conversations are sequences of messages exchanged among interacting agents. For conversations to be meaningful, agents ought to follow commonly known specifications limiting the types of messages that can be exchanged at any point in the conversation. These specifications are usually implemented using conversation policies (which are rules of inference) or conversation protocols (which are predefined conversation templates). In this article we present a semantic model for specifying conversations using conversation policies. This model is based on the principles that the negotiation and uptake of shared social commitments entail the adoption of obligations to action, which indicate the actions that agents have agreed to perform. In the same way, obligations are retracted based on the negotiation to discharge their corresponding shared social commitments. Based on these principles, conversations are specified as interaction specifications that model the ideal sequencing of agent participations negotiating the execution of actions in a joint activity. These specifications not only specify the adoption and discharge of shared commitments and obligations during an activity, but also indicate the commitments and obligations that are required (as preconditions) or that outlive a joint activity (as post-conditions). We model the Contract Net Protocol as an example of the specification of conversations in a joint activity.

Key words: autonomous agents, conversation policies, social commitments, speech acts.

1. INTRODUCTION

Software agents are autonomous, collaborative problem solving entities which are increasingly being applied as a key abstraction for developing software applications (Jennings and Wooldridge 1998). One of the main characteristics of agent-based systems is that agents seek to interact among themselves to perform tasks that help them to meet their (individual or collective) objectives. For example, agents request and offer services, schedule the delivery of parts for manufacturing processes, and negotiate the best possible deal when shopping for books.

Although agents could interact through any action that affects their common environment, we are particularly interested in agents that interact through their communications. In this view, software agents are conceptualized as purely communicational entities (Ferber 1999) that only interact by exchanging messages through a communications channel.

Conversations are the meaningful exchange of messages between interacting agents. In agent-based systems, conversations are traditionally specified using conversation protocols (which are static structures specifying the sequences of messages making a conversation) and conversation policies (which are rules of inference specifying the principles governing the connectedness of message sequences during conversations). Although policies and protocols are normally seen as competing techniques, we subscribe to the view that a protocol is a particular conversation structure that should be constructed following the principles specified by conversation policies. However, it still remains a challenge for the agent communication language community to define the properties and principles that conversation policies should represent (Greaves et al. 1999).

In this paper, we describe a model of conversations in which conversations are guided by policies based on the principle that agents requesting the performance of actions must negotiate the uptake of shared social commitments. As described in Section 2, our model defines a small set of fundamental policies to negotiate the shared state of social commitments, whose uptake entails the obligation to perform the actions specified in the negotiated commitments. We see this negotiation as instrumental for autonomous agents to advance the

state of their joint activities. In Section 3 we illustrate our model with an example involving a conversation using the Contract Net Protocol (Smith 1980). Section 4 discusses our model and related work and finally, Section 5 presents future work and our conclusions.

2. MODELLING AGENT CONVERSATIONS FOR ACTION

This section describes our model for conversations. This model is specified using the Object-Z formal language notation (Smith 2000), which has the advantages of allowing type checking of specifications and a reasonably straightforward translation to computer implementations.[1]

In the subsections below, we first present the definition of elements in our model, such as actions and events (which are the occurrence of actions), speech acts (which are specified as actions) and utterances (which are modelled as occurrences of speech acts). Subsequent subsections define social commitments, shared social commitments, and the operations that can be proposed to affect the state of shared social commitments of agents, namely their addition or discard. This is followed by the definition of agents, which are modelled as entities with a record of utterances, shared social commitments, and the obligations resulting from the adoption of these social commitments. The negotiation of shared social commitments is supported by a simple protocol implemented using illocutionary points and conversation policies. Lastly, this section presents our definition of normative societies, i.e., societies where norms define the expected behaviour of agents when engaged in joint activities.

2.1. Actions

Actions are defined as either *individual actions* (which are atomic actions performed by one agent) or *composite actions* (which are collections of actions performed by one or more agents). We model the composition of actions following the Composite design pattern (Gamma, et al., 1995). To that end, we define the classes *Action* (as the superclass component node), *IndividualAction* (as the leaf node, a subclass of Action), and *CompositeAction* (as the composite node, another subclass of Action).

The class *Action* (shown below) contains the state variable *performers* (to reference a set of one or more objects of type—or subtype of—*Agent*), and the empty operation *Perform* (which is overwritten by subclasses to specify what it means to do the action).

Action

$$performers : \mathbb{P}_1 \downarrow Agent$$

$$Perform \mathrel{\widehat=} [\,]$$

The class *IndividualAction* (shown below) inherits from *Action* and defines the variable *performer* (to reference an instance of type—or subtype of—*Agent*), which is specified as the only agent in the set of performers (which effectively makes it the only performer of the action).

[1] An additional advantage is that it allows formal proving of certain properties of the specified system. In the case of our model, an example proof can be found in (Flores, 2002).

IndividualAction

Action

$$performer : \downarrow Agent$$

$$performers = \{performer\}$$

Lastly, the class *CompositeAction* (shown below) is a class inheriting from *Action* that defines the variable *actions* (to reference a set of one or more instances of type—or subtype of—*Action*), and the operation *Perform*, which specifies that all actions in the actions set are performed concurrently. This class also specifies that the set of performers of this action is equal to the set of all performers of actions in the set actions.

CompositeAction

Action

$$actions : \mathbb{P}_1 \downarrow Action$$

$$\begin{aligned} performers = \{ & performer : \downarrow Agent \mid \\ & \forall\, action : actions \\ & \bullet \forall\, agent : action.performers \\ & \quad \bullet\ agent = performer\} \end{aligned}$$

$$Perform \mathrel{\widehat=} \bigwedge action : actions \bullet action.Perform$$

Basic Individual Actions. Our model defines three basic individual actions from which we derive more meaningful actions:

- *ToInput*: actions that receive an input,
- *ToOutput*: actions that generate an output, and
- *ToProcess*: actions that receive an input and generate an output.

All inputs and outputs in these classes are sets of objects of type—or subtype of—*Data* (which is an empty class that acts as the superclass for all data classes).

ToInput

IndividualAction

$$data? : \mathbb{P} \downarrow Data$$

ToOutput

IndividualAction

$$data! : \mathbb{P} \downarrow Data$$

ToProcess

ToInput

ToOutput

Joint Actions. Lastly, a *joint action* is a type of composite action in which there are two or more performers.[2] All joint actions in our model are derived from this class.

JointAction

CompositeAction

$$\#performers \geq 2$$

Communicational Actions. A *communicational action* is a joint action that involves communication between two agents. In our model the basic communicational action (from which other communicational actions are derived) is *ToCommunicate*. As shown below, this action specifies the state variables *sender* and *receiver* (both of type *Agent*), a variable *data* (to reference a non-empty set of objects of type *Data*), and two individual actions named *send* and *receive* (of type *ToOutput* and *ToInput*, respectively). This class also specifies that the *sender* is the performer of the *send* action; that the *receiver* is the performer of the *receive* action; that the value of the variable *data* equals both the output of the *send* action and the input of the *receive* action; and that the actions *send* and *receive* are the only actions in this joint action. Lastly, the operation *Perform* indicates that the performance of the *send* action precedes the performance of the *receive* action (which effectively makes the output of one action the input for the next).

ToCommunicate

JointAction

$$sender, receiver : \downarrow Agent$$
$$data : \mathbb{P}_1 \downarrow Data$$
$$send : \downarrow ToOutput$$
$$receive : \downarrow ToInput$$

$$sender = send.performer \land$$
$$receiver = receive.performer \land$$
$$data = send.data! = receive.data?$$
$$actions = \{send, receive\}$$

$$Perform \mathrel{\widehat=} send.Perform \ ⨟ \ receive.Perform$$

Speech Acts. According to the speech-as-action tradition (Austin 1962; Searle 1969), speech acts are composed of an *illocutionary point* (conveying the intent of a speech act), an *illocutionary force* (expressing the strength) and a *propositional content* (carrying the information or data).

In our model we see the illocutionary point as the main carrier of meaning (rather than the illocutionary force, which is not modelled).[3] As such, speaking is a joint action in which a

[2]This definition is an oversimplification of joint actions as described by H.H. Clark (Clark, 1996). Clark specifies that there exists two types of individual actions: *autonomous actions* (actions that are performed by one agent in isolation) and *participatory actions* (actions performed by one agent in coordination with the performances of other agents in a joint action). As such, joint actions have participatory actions as their base actions. In the case of this model, however, individual actions are treated as atomic and no differentiation is made between the two types.

[3]To illustrate the difference between the illocutionary point and illocutionary force in a speech act, imagine the utterances "Could you please do your homework now?" and "You must do your home work right

speaker communicates illocutionary points to an addressee (where an illocutionary point is a subtype of *Data*). As shown below, the class of *ToSpeak* is a subclass of *ToCommunicate* that defines the state variables *speaker* and *addressee* (to indicate the sender and receiver agents), *points* (to indicate a non-empty set of illocutionary points that is communicated), and the variables *voice* and *hear* (to indicate the actions for sending and receiving the communicated illocutionary points, respectively). The types *ToVoice* and *ToHear* (from which the variables *voice* and *hear* are instantiated) are defined as subclasses of *ToSend* and *ToReceive*, respectively.

ToSpeak

ToCommunicate

$speaker, addressee : \downarrow Agent$
$points : \mathbb{P}_1 \downarrow IllocutionaryPoint$
$voice : \downarrow ToVoice$
$hear : \downarrow ToHear$

$speaker = sender \land$
$addressee = receiver \land$
$points = data$

$voice = send \land$
$hear = receive$

Lastly, we define an illocutionary point that conveys information (i.e., propositional content) between a speaker and addressee. As shown below, the *Inform* illocutionary point defines a variable *informing* that references a set of instances of type (or subtype of) *Data*. In section 2.5.1 we define other illocutionary points that are used for the negotiation of shared social commitments.

Other Basic Joint Actions. We define two basic joint action classes from which other application-dependent actions are derived. These joint actions are *ToGenerate* and *ToProduce*. As shown below, the class *ToGenerate* specifies that a *generator* agent first performs a *ToOutput* action that produces some output data (*out*). Then it performs a *ToSpeak* action to inform a *receiver* agent of the output data.

now!" These utterances have the same illocutionary point (that of the addressee doing his/her homework now) but different illocutionary force (the former being a polite request and the latter a forceful order).

ToGenerate

JointAction

$$
\begin{array}{l}
generator, receiver : \downarrow Agent \\
out : \mathbb{P} \downarrow Data \\
output : \downarrow ToOutput \\
speak : \downarrow ToSpeak \\
\hline
generator = output.performer = speak.speaker \land \\
receiver = speak.addressee \land \\
out = output.data! \land \\
actions = \{output, speak\} \\
\exists\, i : Inform \mid \\
\quad i \in speak.points \\
\bullet\ i.informing = out
\end{array}
$$

$$Perform \mathrel{\widehat=} output.Perform \fatsemi speak.Perform$$

The class *ToProduce* is a subclass of *ToGenerate* in which the output action is an instance of *ToProcess*, i.e., an individual action that generates an output given an input.[4] Based on the behaviour inherited from *ToGenerate*, the output of the action is communicated to the receiver through an *Inform* illocutionary point.

ToProduce

ToGenerate

$$
\begin{array}{l}
producer : \downarrow Agent \\
in : \mathbb{P} \downarrow Data \\
process : \downarrow ToProcess \\
\hline
producer = generator \land \\
process = output \land \\
in = process.data?
\end{array}
$$

Events & Utterances. Actions in our model are abstract concepts that have not occurred in the environment. When they do happen they are considered events. Therefore, an event is the occurrence of an action at a certain moment in time. As shown below, *Event* is a class specifying the state variables *time* (to indicate the time of occurrence of the event) and *action* (to indicate the action that occurred).[5]

Event

$$
\begin{array}{l}
time : Time \\
action : \downarrow Action
\end{array}
$$

Just as an event is the record of the occurrence of any type of action, an utterance is an event specifically involving the illocution of a speech act. Therefore, *Utterance* is a class

[4]This class also defines the variable *producer* to reference the inherited *generator* agent. The only reason for this redundancy is to facilitate readability, e.g., a producer performs a producing action (c.f., generator).

[5]In our model, the type *Time* is simply defined as a natural number, i.e., Time == $\mathbb{N}$.

inheriting from *Event* whose variable *action* is restricted to an instance of type *ToSpeak*.

Utterance
Event

$$speechAct : ToSpeak$$
$$speechAct = action$$

Scheduling Actions. When actions are negotiated they ought to refer to a time interval indicating when their performance is expected to occur. We define the class *Interval* (shown below) to indicate a time period. This class defines the state variables *from* and *until* to denote the lower and upper bound of the interval.

Interval

$$from, until : Time$$
$$from \leq until$$

Using this definition, the class *Acting* (shown below) is specified as a subclass of *Action* that defines a variable *time* of type *Interval*. *Interval* denotes the time period within which the action is to be performed.[6] This class specifies an empty operation *ProcessNorm* that is overwritten by subclasses and which is used at the time that the action is evaluated by norms in a society (as explained in section 2.5:Negotiating Shared Social Commitments).

Acting
Action

$$time : Interval$$
$$ProcessNorm \mathrel{\widehat=} [\,]$$

The class *IndividualActing* is a subclass of *IndividualAction* and *Acting* that overrides the operation *ProcessNorm* to invoke the operation *ProcessAction* of a given norm, which is an instance of type *ActionNorm* (this class is also explained in section 2.5).

IndividualActing
IndividualAction
Acting

$$ProcessNorm \mathrel{\widehat=} [\, norm? : \downarrow ActionNorm \,] \bullet norm?.ProcessAction$$

Likewise, the class *CompositeActing* is a subclass of *IndividualAction* and *Acting* that overrides the operation *ProcessNorm* not only to invoke the operation *ProcessAction* but also to invoke the operation *ProcessNorm* for each enclosed action. Also, this class constraints all enclosed actions to be of type (or subtype of) *Acting*, and that the time period in which these

[6] We follow the convention of naming actions either on their infinitive form (e.g., *ToSpeak*) for abstract actions, or on their present participle form (e.g., *Speaking*) for scheduled actions (i.e., actions that have an expected interval time of occurrence).

actions are expected to occur is within the time of occurrence specified for the composite action.

CompositeActing

$$
\begin{array}{l}
CompositeAction \\
Acting \\
\hline
\forall\, action : actions \\
\bullet\ \exists\, a : \downarrow Acting \mid \\
\quad a = action \\
\quad \bullet\ a.time.from \geq time.from \land \\
\quad\ a.time.until \leq time.until \\
\hline
ProcessNorm \mathrel{\widehat=} [\, norm? : \downarrow ActionNorm \,] \bullet \\
\qquad\qquad \bigwedge action : actions \bullet norm?.ProcessAction \land action.ProcessNorm
\end{array}
$$

2.2. Social Commitments

Social commitments are directed obligations in which one agent (called the *debtor* of the commitment) has a responsibility relative to another agent (the *creditor* of the commitment) for the performance of an action (Singh, et al. 1999).[7] Accordingly, the class *SocialCommitment* specifies the state variables *debtor*, *creditor* and *action*.

SocialCommitment

$$
\begin{array}{l}
debtor, creditor : \downarrow Agent \\
action : \downarrow Acting
\end{array}
$$

Shared Social Commitments. Once a social commitment has been proposed and agreed upon by negotiating agents it acquires the status of being shared. The class *SharedSocialCommitment* represents this type of social commitment. This class inherits from *SocialCommitment* and defines the variable *among* to indicate the set of agents among which this commitment is shared.[8]

SharedSocialCommitment

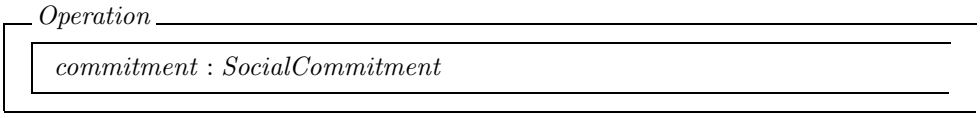
$$
\begin{array}{l}
SocialCommitment \\
\hline
among : \mathbb{P} \downarrow Agent \\
\hline
\#among \geq 2
\end{array}
$$

Operations. The proposal and later acceptance of a social commitment can have two possible outcomes: either the commitment is adopted as shared, or the commitment is discarded as not shared any longer. Therefore, two operations are defined that agents can use when putting forth commitments for negotiation: adding and deleting. First, we define the

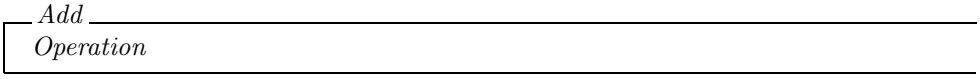
[7] This type of commitment is also known as a *relational commitment* (Ferber 1999).

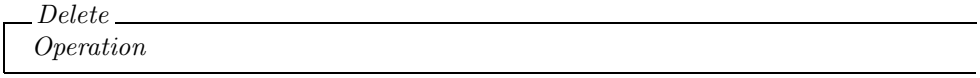
[8] Although strictly speaking a social commitment could be shared by any number of agents (as long as it is more than one) our model currently deals only with commitments shared between a speaker and an addressee.

class *Operation* as the superclass for these two operations. This class defines a single state variable *commitment* to reference an instance of type *SocialCommitment*.



The classes *Add* and *Delete* are then defined as inheriting from the class *Operation* (the usage of these operations is later shown in section 2.5).





2.3. Obligations

Obligations are engagements that bind an agent to a course of action.[9] In our model, obligations are created (and discarded) by the adoption (and discharge) of shared social commitments. For example, that Bob has adopted a commitment in which he is to tell Alice the time creates the obligations in Bob that he has to find out the time and that he has to tell the time to Alice. Obligations are defined as actions that are expected to occur (i.e., as any action of type *Acting*).

$$Obligation == \downarrow Acting$$

2.4. Agents

Agents are conceptualized as entities that keep a collection of shared social commitments and obligations, and a history of their utterances.

As shown in Figure 1, the class *Agent* defines the variables *commitments* (which holds the shared social commitments of the agent), *obligations* (which holds the social obligations of the agent),[10] *inbox* and *history* (which are the sequence of utterances that the agent has

[9] This definition of obligation is akin to the notion of *simple action-commitment statements* specified by D.N. Walton and E.C.W. Krabbe (1995).

[10] There is a compelling reason to define the collections of shared commitments and obligations as bags (which allow duplicate elements) rather than sets (which do not allow duplicates). In our model, that an agent holds identical obligations only means that she has recorded those entries given independent interactions, not necessarily that the agent will perform the involved actions as may times as recorded. That is, the possible optimization of performances (i.e., whether one performance satisfies all obligations or if independent performances are required) is bound by the nature of the involved actions. For example, that Alice has committed to Bob to brush her teeth after dinner and that she has also committed to Charles to brush her teeth after dinner would generate two identical entries in Alice's record of obligations. No other details withstanding, Alice should be able to perform the action once to satisfy her commitments with both Bob and Charles. On the other hand, there are actions that would not allow such optimizations. That would be the case of Alice and Bob adopting a shared commitment in which Bob bakes a cake for Alice tomorrow. If later that day Alice and Bob adopt another commitment in which Bob bakes a cake (one with identical characteristics as that of the previously requested cake) for Alice by the same time tomorrow, then Alice and Bob will hold a pair of identical commitments and obligations that should lead Bob to prepare two identical cakes for Alice.

Agent

$$commitments, com\_add, com\_delete : \text{bag } SharedSocialCommitment$$
$$obligations, obl\_add, obl\_delete : \text{bag } Obligation$$
$$inbox, history : \text{seq } Utterance$$
$$current : Utterance$$

$$\forall c : \text{dom } commitments \bullet self \in c.among$$

$$\forall o : \text{dom } obligations \bullet self \in o.performers$$

$$\forall u : \text{ran } (inbox \frown \langle current \rangle \frown history) \bullet u.speechAct.speaker = self \lor u.speechAct.addressee = self$$

$$\forall u_1, u_2 : Utterance \mid \langle u_1, u_2 \rangle \text{ in } inbox \bullet u_1.time > u_2.time$$

INIT

$$commitments = \varnothing$$
$$obligations = \varnothing$$
$$inbox = \langle\,\rangle$$
$$history = \langle\,\rangle$$

AddUtterance

$$\Delta(inbox)$$
$$u? : Utterance$$

$$inbox' = inbox \frown \langle u? \rangle$$

SetCurrent

$$\Delta(inbox, current, com\_add, com\_delete, obl\_add, obl\_delete)$$

$$current' = head\ inbox$$
$$inbox' = tail\ inbox$$
$$com\_add' = \varnothing$$
$$com\_delete' = \varnothing$$
$$obl\_add' = \varnothing$$
$$obl\_delete' = \varnothing$$

$$ProcessCurrent \mathrel{\widehat=} [\, norms? : \mathbb{P} \downarrow Norm \,] \bullet \bigwedge n : norms? \bullet n.ProcessUtterance$$

AdvanceCurrent

$$\Delta(current, history)$$

$$history' = \langle current \rangle \frown history$$
$$commitments' = (commitments \setminus com\_delete) \uplus com\_add$$
$$obligations' = (obligations \setminus obl\_delete) \uplus obl\_add$$

$$Process \mathrel{\widehat=} SetCurrent \fatsemi ProcessCurrent \fatsemi AdvanceCurrent$$

FIGURE 1. Definition of the class *Agent* (part 1 of 2)

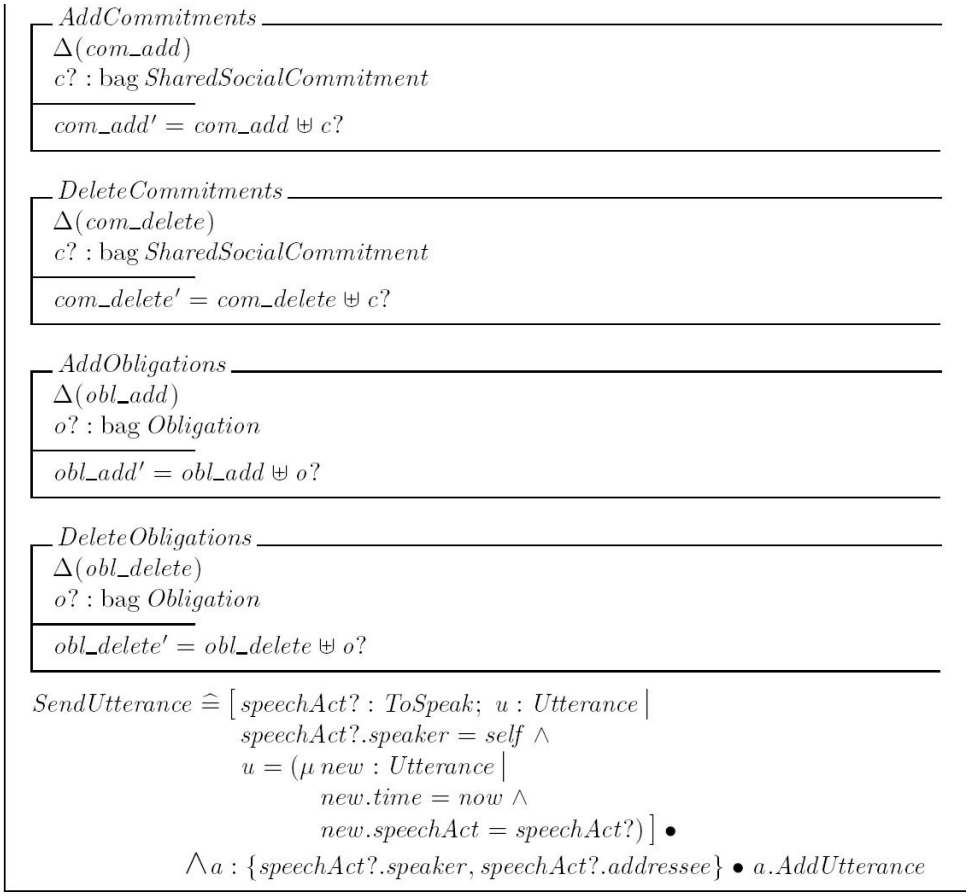


Figure 2. Definition of the class *Agent* (part 2 of 2)

just received but has not processed yet, and the sequence of utterances that have already been processed, respectively), and the variable *current* (which specifies the utterance that is currently being processed).[11]

The remaining variables *com_add*, *com_delete*, *obl_add* and *obl_delete* are auxiliary variables that function as temporary containers to hold the commitments and obligations that are accumulated by the various norms when processing the utterance in current. As it will be seen shortly, norms do not modify the state of commitments and obligations of an agent while each of the norms is being evaluated but rather all modifications are pooled in the above auxiliary variables for their later application once the utterance has been processed by all norms. The benefit of this technique is that it maintains the consistency of an agent state regardless of the order in which norms process an utterance, and the order in which the illocutionary points within the utterance are processed by a norm.

[11]The meaning of "processing an utterance" will become clearer in later sections on norms and societies. In brief, when an agent receives an utterance this utterance is stored in the *inbox* sequence until its turn comes to be processed by the norms that govern the behaviour of the agent (at which point the utterance becomes *current*, i.e., the utterance currently being processed). Once this utterance has been processed, it becomes part of the history of utterances of the agent (and thus it is appended to the *history* sequence).

In addition, several restrictions are set on the following variables:

- *commitments* only holds shared social commitments in which the agent is one of the agents among which this commitment is shared,
- *obligations* only holds obligations in which the agent is one of the performers of the action, and
- *inbox* (whose utterances are ordered by time of occurrence), *current* and *history* only hold utterances in which the agent is either the speaker or the addressee.[12]

The interface of this class is defined by the operations shown in Figure 1 and Figure 2. Figure 1 shows the following operation schemas:

- *INIT*, which initializes the variables *commitments*, *obligations*, *inbox* and *history* to empty,
- *AddUtterance*, which appends an utterance to the *inbox* sequence,
- *SetCurrent*, which initializes as empty all auxiliary variables, and assigns the next utterance out of *inbox* as the next *current*,
- *ProcessCurrent* which processes the utterance in *current* through all supplied norms,
- *AdvanceCurrent*, which adds *current* to the history of utterances, and updates the commitments and obligations of the agent with those commitments and obligations from the auxiliary variables, and
- *Process*, which sequentially invokes *SetCurrent*, *ProcessCurrent* and *AdvanceCurrent*.

Figure 2 shows the operation schemas *AddCommitments* and *DeleteCommitments*, which add shared social commitments to the *com_add* and *com_delete* variables, respectively; and *AddObligations* and *DeleteObligations*, which add obligations to the variables *obl_add* and *obl_delete*. Lastly, this class defines the operation *SendUtterance*, which specifies that the agent utters a given speech act to an addressee if and only if the agent is set as the speaker of the speech act.

2.5. Negotiating Shared Social Commitments

One way to support the autonomy of agents is to allow them to decide whether or not other agents can commit them to execute actions. In the case of our model, this means that shared social commitments are not imposed but rather are negotiated between interacting agents. To that end, we define a negotiation protocol that we call the *Protocol for Proposals* (PFP). Briefly explained, an instance of the PFP starts with a proposal from a speaker to a hearer to adopt or discard a shared social commitment. Either the hearer replies with an acceptance, rejection or counterproposal, or the speaker issues a withdrawal (i.e., a rejection

[12]This last restriction reflects the fact that the *Agent* class is designed to model the behavioural representation that an observer agent holds of other agents in the environment. In practice this means that only when an agent (Alice) has perceived that an utterance has occurred between two agents (Bob and Charles), then she is justified in updating the representation she maintains of these two agents. Now imagine that there was another agent (Dave) in the proximity, who may have also perceived the utterance between Bob and Charles. Would Alice be justified in updating her representation of Dave to reflect that he has witnessed the communication? From a behavioural perspective, the answer is no. Basically, Alice can only know that Dave has also witnessed the utterance if Dave makes his awareness public (e.g. by uttering that he knows about it), in which case Alice would be justified in updating her and Dave's representations because there was an utterance involved. Otherwise Alice would need to reason whether there are compelling reasons that justify concluding that Dave knows about the utterance—which clearly is a rational process that lies outside of the scope of our model.

to one's own proposal) or a counterproposal.[13] All replies except a counterproposal terminate an instance of the protocol. A counterproposal is deemed as a proposal in the sense that it can be followed by the same messages that can reply to a proposal (but with speaker-hearer roles inverted if the previously proposed hearer is the speaker of the counterproposal).[14] Lastly, when an acceptance is issued, both speaker and hearer simultaneously apply the proposed (and now accepted) operation to their record of shared social commitments and obligations. As described below, our model implements the PFP using illocutionary points and conversation policies.

Illocutionary points. This section defines the illocutionary points that are used to implement the PFP, namely *Propose*, *Accept*, *Reject* and *Counter* (for counter-proposals).

Propose. The class *Propose* defines the state variables *proposing* (which indicates the operation on a social commitment being proposed) and *reply* (which indicates the time interval when a reply is expected).

Propose

IllocutionaryPoint

$proposing : \downarrow Operation$
$reply : Interval$

$proposing \in Add \wedge proposing \in Delete$

Accept. The class *Accept* defines the state variable *accepting* (which indicates the operation on a social commitment being accepted).

Accept

IllocutionaryPoint

$accepting : \downarrow Operation$

$accepting \in Add \wedge accepting \in Delete$

Reject. The class *Reject* defines the state variable *rejecting* (which indicates the operation on a social commitment being rejected).

Reject

IllocutionaryPoint

$rejecting : \downarrow Operation$

$rejecting \in Add \wedge rejecting \in Delete$

[13] It is also possible that the hearer goes silent. In such cases, the elapsing of the expected reply time indicates to the speaker (or any observer) that the hearer either intentionally forfeited his obligation to reply or was unable to communicate as expected. In such matters, we are assuming that communication between agents is reliable, i.e., the transmission of utterances is always achieved (c.f. the *coordinated attack* problem (Fagin, et al. 1995)).

[14] In theory, a counterproposal can follow another counterproposal *ad infinitum*; in practice, however, successive counterproposals are limited by the reasoning, competence or endurance of interacting agents.

Counter. The class *Counter* is a class inheriting from *Reject* and *Propose*, where the former indicates a commitment previously proposed and now being rejected, and the latter indicates a newly proposed commitment (and corresponding expected time of reply).

Counter
Reject
Propose

Conversation Policies. In our model, we define conversation policies as norms that agents are expected to follow during their conversations. As shown below, the class *Norm* (which is the superclass of all norms in our model) defines a sole abstract operation *ProcessUtterance*. This operation, which is invoked when evaluating the current utterance of an agent, is overridden by subclasses defining concrete normative behaviour.

Norm
$ProcessUtterance \mathrel{\widehat=} [\,]$

The class *ActionNorm* is a subclass of *Norm* that defines an abstract operation *ProcessAction*. This operation is invoked by norms processing an utterance. It is overridden by policies that generate obligations based on the actions of negotiated social commitments.

ActionNorm
Norm
$ProcessAction \mathrel{\widehat=} [\,]$

The first policy in our model specifies that agents proposed or counterproposed have to reply.[15] As shown in Figure 3, this policy is specified as the class *PFPpolicy1* that inherits from *Norm* and defines the operations *ProcessUtterance* (which overrides the operation inherited from *Norm*) and *ProcessProposal*.

The operation *ProcessUtterance* specifies that for each proposal in a given utterance it successively invokes the operations *ProcessProposal* and *AddObligations*, which result in the generation and addition of obligations to a given agent instance.

The operation *ProcessProposal* is specified as receiving a proposal and returning a set of obligations to reply. In the case that the agent given as input is the speaker of the proposal, this operation returns obligations to a speaking action in which the agent is to hear the reply. In the case that the agent is the addressee, the operation returns obligations to the same speaking action but where the agent is to voice the reply. In addition, the reply's illocutionary points are specified by the axiom *isReplyTo* (which is defined below).

The axiom *isReplyTo* is a function that defines the set of illocutionary points that qualify as a reply to a given proposal. As shown below, this operation specifies that a proposal is replied to by a set of illocutionary points where there is either one illocutionary point accepting (and not one rejecting), or one illocutionary point rejecting (and not one accepting) the operation specified in the proposal.

[15] Although this is a strong assumption for autonomous agents, we see it as a rule of politeness: you answer if you are proposed. In any event, agents are still free to disregards this (or any) policy when they see it fit (e.g., if an insidious or defective agent sends inappropriate or hostile messages).

$$
\begin{array}{|l}
\hline
PFPpolicy1 \\
\lceil(ProcessUtterance) \\
Norm \\
\quad \begin{array}{|l}
\hline
ProcessProposal \\
agent?, speaker?, addressee? : \downarrow Agent \\
propose? : \downarrow Propose \\
result! : \mathbb{P}\, Obligation \\
\hline
\exists\, sa : Speaking \mid \\
\quad sa.time = propose?.reply \land \\
\quad sa.speaker = addressee? \land \\
\quad sa.addressee = speaker? \land \\
\quad sa.points = isReplyTo(propose?) \\
\bullet\ (agent? = speaker? \land result! = \{sa, sa.hear\}) \lor \\
\quad (agent? = addressee? \land result! = \{sa, sa.voice\}) \\
\hline
\end{array} \\
ProcessUtterance \mathrel{\widehat=} [\, agent? : \downarrow Agent;\ u? : Utterance \,] \bullet \\
\quad \bigwedge propose : \downarrow Propose \mid \\
\qquad propose \in u?.speechAct.points \\
\quad \bullet\ ProcessProposal\ {}^{\circ}_{9}\ agent?.AddObligations \\
\hline
\end{array}
$$

FIGURE 3. *Policy 1*: A proposal commits agents to reply.

$$
\begin{array}{|l}
\hline
isReplyTo : \downarrow Propose \to \mathbb{P}_1 \downarrow IllocutionaryPoint \\
\hline
\forall\, propose : \downarrow Propose \\
\bullet\ \exists\, points : \mathbb{P} \downarrow IllocutionaryPoint \mid \\
\quad \exists\, a : Accept;\ r : \downarrow Reject \mid \\
\qquad a.accepting = propose.proposing \land \\
\qquad r.rejecting = propose.proposing \\
\quad \bullet\ (a \in points \land r \notin points) \lor \\
\qquad (r \in points \land a \notin points) \\
\quad \bullet\ points = isReplyTo(propose) \\
\hline
\end{array}
$$

Policy 2: Discharging Obligations to Reply. Once an agent has adopted obligations to reply, she can expect these obligations to be discarded if any of the following two conditions occur:

1 The agent that was proposed to (or counterproposed to) utters a speech act containing an *Accept* illocutionary point with the same operation on commitment as that of the previously uttered *Propose* (or *Counter*).
2 The proposing or proposed to (or the counterproposing or counterproposed to) agent utters a speech act containing a *Reject* illocutionary point with the same operation on commitment as that of the previously uttered *Propose* (or *Counter*).

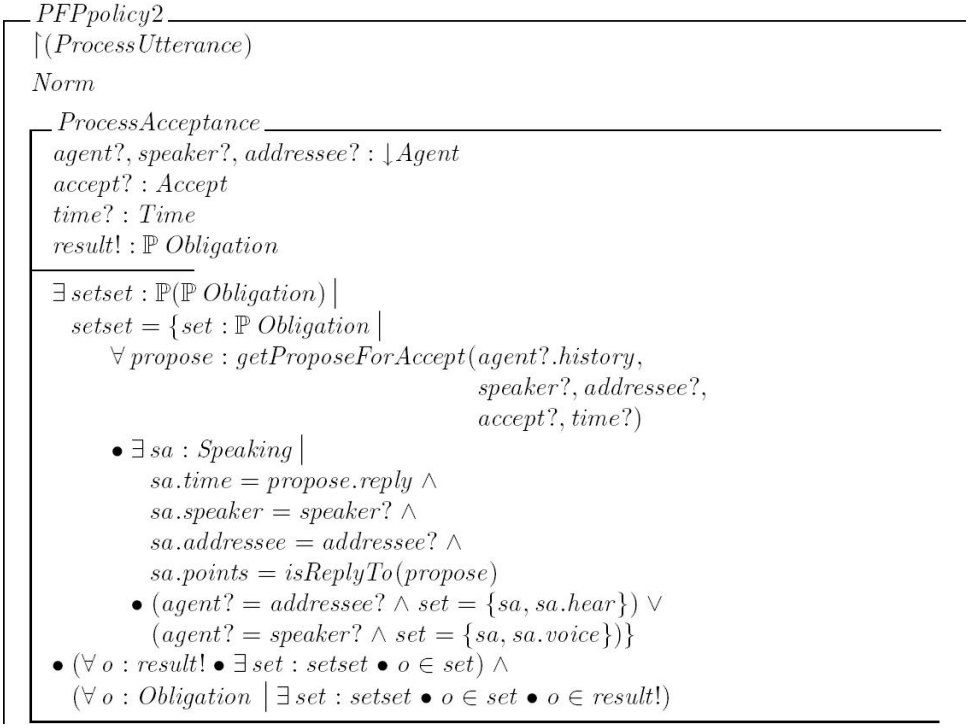


FIGURE 4. *Policy 2*: A reply releases agents of the obligation to reply (part 1 of 2).

To support these conditions we define the class *PFPpolicy2* (shown in Figure 4 and Figure refpolicy2b), which inherits from *Norm* and declares the operations *ProcessAcceptance*, *ProcessRejection* and *ProcessUtterance* (which overrides the abstract operation inherited from *Norm*).

The operation *ProcessUtterance* is specified as invoking the operations *ProcessAcceptance* and *ProcessRejection* for each acceptance and rejection (respectively) in a given utterance, and then invoking the operation *DeleteObligations*, which results in the generation and deletion of obligations from a given agent instance.

The operation *ProcessAcceptance* is specified as receiving an acceptance and checking whether this acceptance has matching proposals in the history of past utterances of the agent. If such proposals exist then the history of utterances is analysed once more to find out whether these proposals have been replied to. If at least one of these proposals has not been replied to, then the acceptance is a legitimate reply, and thus the corresponding obligations are selected for discharge. The process of finding matching proposals to a given acceptance is specified by the axiom *getProposeForAccept*, which is defined shortly.

Similarly to the operation *ProcessAcceptance*, the operation *ProcessRejection* specifies that for each given valid rejection that matches one or more past, non-replied to proposals there exists obligations to reply that will be later discarded from the state of the agent. As before, this operation takes a rejection and checks whether there are matching proposals in the history of past utterances of the agent. If such proposals exist then the history of utterances is analysed once more to detect whether each of these proposals has been replied to. If at least one proposal has not been replied to, then the rejection is a legitimate reply to the proposal, and thus the corresponding obligations to reply are selected for discharge.

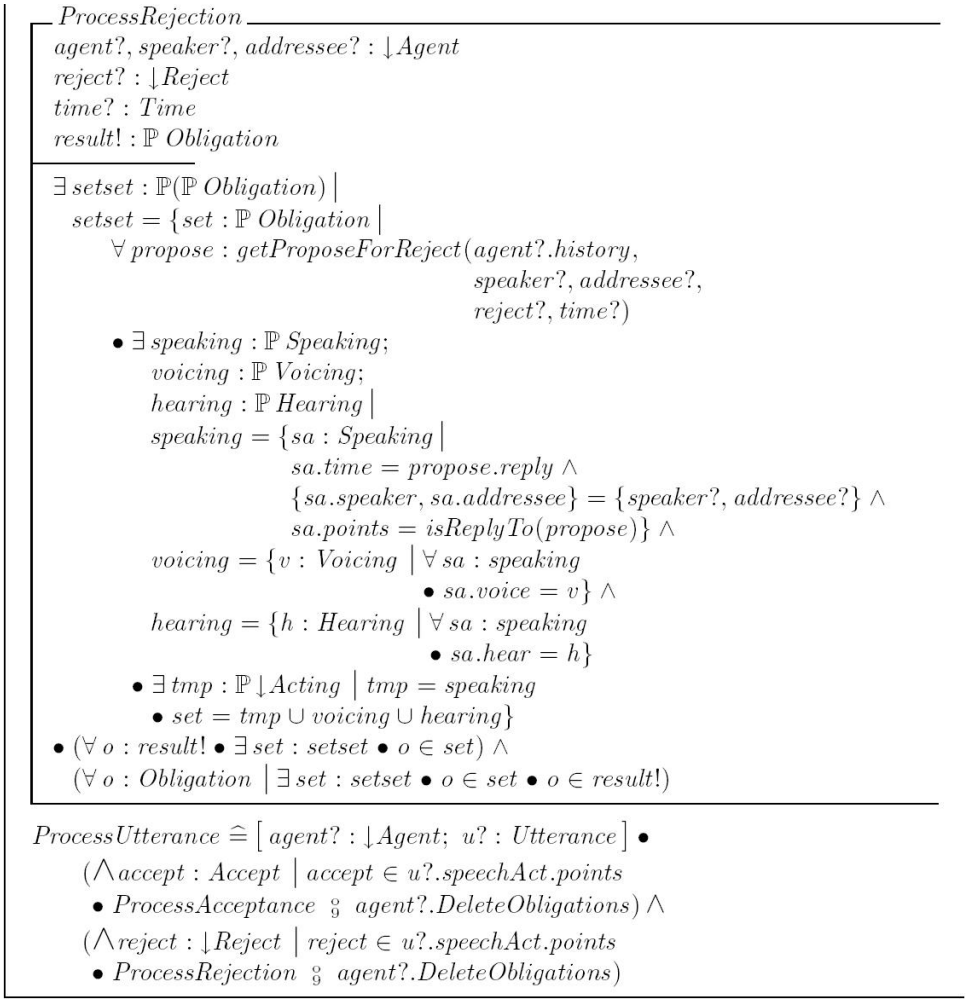


FIGURE 5. *Policy 2*: A reply releases agents of the obligation to reply (part 2 of 2).

The process of finding matching proposals to a given rejection is specified by the axiom *getProposeForReject*, which is defined next.

Querying the History of Utterances. The PFP specifies that conversations negotiating shared social commitments are patterns of proposals followed by an acceptance, a rejection or a counterproposal. To keep track of these patterns, agents maintain a history of the utterances in which they have participated. When an agent receives an acceptance or a rejection, the history of utterances is analysed to determine whether there is a matching past proposal (i.e., a not-yet-replied proposal with an identical operation on commitment as that of the acceptance or rejection) to derive that this acceptance or rejection is part of an ongoing conversation.

These queries are supported by the axioms *getProposeForAccept* and *getProposeForReject*.

The axiom *getProposeForAccept* (shown in Figure 6) specifies that if there exists a past

$$
\begin{array}{l}
getProposeForAccept : \operatorname{seq} Utterance \times Agent \times Agent \times \\
\qquad\qquad Accept \times Time \to \mathbb{P} \downarrow Propose \\
\hline
\forall\, utterances : \operatorname{seq} Utterance;\ speaker, addressee : Agent; \\
\qquad\qquad accept : Accept;\ time : Time \\
\bullet\ \exists\, result : \mathbb{P} \downarrow Propose \mid \\
\quad result = \{p : \downarrow Propose \mid \\
\qquad\qquad \forall\, u_1 : Utterance \mid \\
\qquad\qquad\quad u_1 \in \operatorname{ran} utterances \land \\
\qquad\qquad\quad u_1.time < time \land \\
\qquad\qquad\quad u_1.speechAct.speaker = addressee \land \\
\qquad\qquad\quad u_1.speechAct.addressee = speaker \\
\qquad\qquad \bullet\ \exists\, p_1 : \downarrow Propose \mid \\
\qquad\qquad\quad p_1 \in u_1.speechAct.points \land \\
\qquad\qquad\quad p_1.reply.from \leq time \leq p_1.reply.until \land \\
\qquad\qquad\quad p_1.proposing = accept.accepting \land \\
\qquad\qquad\quad \neg\, (\exists\, u_2 : Utterance \mid \\
\qquad\qquad\qquad\quad u_2 \in \operatorname{ran} utterances \land \\
\qquad\qquad\qquad\quad u_1.time < u_2.time < time \land \\
\qquad\qquad\qquad\quad p_1.reply.from \leq u_2.time \leq p_1.reply.until \\
\qquad\qquad\qquad\quad \bullet\ (\exists\, a_1 : Accept \mid \\
\qquad\qquad\qquad\qquad a_1 \in u_2.speechAct.points \land \\
\qquad\qquad\qquad\qquad a_1.accepting = p_1.proposing \land \\
\qquad\qquad\qquad\qquad u_2.speechAct.speaker = u_1.speechAct.addressee \land \\
\qquad\qquad\qquad\qquad u_2.speechAct.addressee = u_1.speechAct.speaker \\
\qquad\qquad\qquad\qquad \bullet\ \neg\, (\exists\, r_1 : \downarrow Reject \mid \\
\qquad\qquad\qquad\qquad\qquad r_1 \in u_2.speechAct.points \\
\qquad\qquad\qquad\qquad\quad \bullet\ r_1.rejecting = a_1.accepting)) \lor \\
\qquad\qquad\qquad\qquad (\exists\, r_2 : \downarrow Reject \mid \\
\qquad\qquad\qquad\qquad\quad r_2 \in u_2.speechAct.points \land \\
\qquad\qquad\qquad\qquad\quad r_2.rejecting = p_1.proposing \land \\
\qquad\qquad\qquad\qquad\quad \{u_1.speechAct.speaker, u_1.speechAct.addressee\} = \\
\qquad\qquad\qquad\qquad\quad \{u_2.speechAct.speaker, u_2.speechAct.addressee\} \\
\qquad\qquad\qquad\qquad\quad \bullet\ \neg\, (\exists\, a_2 : Accept \mid \\
\qquad\qquad\qquad\qquad\qquad\quad a_2 \in u_2.speechAct.points \land \\
\qquad\qquad\qquad\qquad\qquad\quad u_2.speechAct.speaker = u_1.speechAct.addressee \land \\
\qquad\qquad\qquad\qquad\qquad\quad u_2.speechAct.addressee = u_1.speechAct.speaker \\
\qquad\qquad\qquad\qquad\qquad\quad \bullet\ a_2.accepting = r_2.rejecting))) \\
\qquad\qquad \bullet\ p_1 = p\} \\
\quad \bullet\ result = getProposeForAccept(utterances, speaker, addressee, accept, time)
\end{array}
$$

FIGURE 6. Axiom *getProposeForAccept*: Retrieving a past proposal matching an acceptance.

proposal from addressee to speaker that:

a is still answerable at the time that the acceptance occurred,
b proposes the same operation on commitment as that of the acceptance, and
c for which a later acceptance or rejection replying to this proposal does not exist;

$$
\begin{array}{l}
getProposeForReject : \mathrm{seq}\ Utterance \times Agent \times Agent \times \\
\qquad\qquad \downarrow Reject \times Time \rightarrow \mathbb{P} \downarrow Propose \\
\hline
\forall\, utterances : \mathrm{seq}\ Utterance;\ speaker, addressee : Agent; \\
\qquad\qquad reject : \downarrow Reject;\ time : Time \\
\bullet\ \exists\, result : \mathbb{P} \downarrow Propose \mid \\
\quad result = \{p : \downarrow Propose \mid \\
\qquad\qquad \forall\, u_1 : Utterance \mid \\
\qquad\qquad\quad u_1 \in \mathrm{ran}\ utterances \wedge \\
\qquad\qquad\quad u_1.time < time \wedge \\
\qquad\qquad\quad \{u_1.speechAct.speaker, u_1.speechAct.addressee\} = \\
\qquad\qquad\quad \{speaker, addressee\} \\
\qquad\qquad \bullet\ \exists\, p_1 : \downarrow Propose \mid \\
\qquad\qquad\quad p_1 \in u_1.speechAct.points \wedge \\
\qquad\qquad\quad p_1.reply.from \leq time \leq p_1.reply.until \wedge \\
\qquad\qquad\quad p_1.proposing = reject.rejecting \wedge \\
\qquad\qquad\quad \neg\, (\exists\, u_2 : Utterance \mid \\
\qquad\qquad\qquad\quad u_2 \in \mathrm{ran}\ utterances \wedge \\
\qquad\qquad\qquad\quad u_1.time < u_2.time < time \wedge \\
\qquad\qquad\qquad\quad u_2.time \leq p_1.reply.until \\
\qquad\qquad\qquad\quad \bullet\ (\exists\, a_1 : Accept \mid \\
\qquad\qquad\qquad\qquad a_1 \in u_2.speechAct.points \wedge \\
\qquad\qquad\qquad\qquad a_1.accepting = p_1.proposing \wedge \\
\qquad\qquad\qquad\qquad u_2.speechAct.speaker = u_1.speechAct.addressee \wedge \\
\qquad\qquad\qquad\qquad u_2.speechAct.addressee = u_1.speechAct.speaker \\
\qquad\qquad\qquad\qquad \bullet\ \neg\, (\exists\, r_1 : \downarrow Reject \mid \\
\qquad\qquad\qquad\qquad\qquad r_1 \in u_2.speechAct.points \\
\qquad\qquad\qquad\qquad\quad \bullet\ r_1.rejecting = a_1.accepting)) \vee \\
\qquad\qquad\qquad\quad (\exists\, r_2 : \downarrow Reject \mid \\
\qquad\qquad\qquad\qquad r_2 \in u_2.speechAct.points \wedge \\
\qquad\qquad\qquad\qquad r_2.rejecting = p_1.proposing \wedge \\
\qquad\qquad\qquad\qquad \{u_1.speechAct.speaker, u_1.speechAct.addressee\} = \\
\qquad\qquad\qquad\qquad \{u_2.speechAct.speaker, u_2.speechAct.addressee\} \\
\qquad\qquad\qquad\qquad \bullet\ \neg\, (\exists\, a_2 : Accept \mid \\
\qquad\qquad\qquad\qquad\qquad a_2 \in u_2.speechAct.points \wedge \\
\qquad\qquad\qquad\qquad\qquad u_2.speechAct.speaker = u_1.speechAct.addressee \wedge \\
\qquad\qquad\qquad\qquad\qquad u_2.speechAct.addressee = u_1.speechAct.speaker \\
\qquad\qquad\qquad\qquad\quad \bullet\ a_2.accepting = r_2.rejecting))) \\
\qquad\qquad \bullet\ p_1 = p\} \\
\quad \bullet\ result = getProposeForReject(utterances, speaker, addressee, reject, time)
\end{array}
$$

Figure 7. Axiom *getProposeForReject*: Retrieving a past proposal matching a rejection.

then this proposal is a member of the set returned by the function.

The axiom *getProposeForReject* (shown in Figure 7) is specified similarly.

Our third policy specifies the consequences of participating in PFP conversations: that of adopting and discarding shared social commitments and the obligations that these com-

PFPpolicy3

$\lceil(ProcessUtterance, ProcessAction)$

$ActionNorm$

ProcessAction

$agent? : \downarrow Agent$
$acting? : \downarrow Acting$
$result! : Obligation$

$agent? \in acting?.performers$
$result! = acting?$

ProcessAcceptance

$speaker?, addressee? : \downarrow Agent$
$accept? : Accept$
$result! : SharedSocialCommitment$

$\exists\, sc : SocialCommitment \mid$
$sc = accept?.accepting.commitment$
$\bullet\ result! = (\mu\, ssc : SharedSocialCommitment \mid$
$ssc.among = \{speaker?, addressee?\} \land$
$ssc.debtor = sc.debtor \land$
$ssc.creditor = sc.creditor \land$
$ssc.action = sc.action)$

$ProcessUtterance \mathrel{\widehat=} [\, agent? : \downarrow Agent;\ u? : Utterance \,] \bullet$
$(\bigwedge accept : Accept;\ action : \downarrow Acting \mid$
$accept \in u?.speechAct.points \land$
$accept.accepting \in Add \land$
$accept.accepting.commitment.action = action \land$
$\#(getProposeForAccept(agent?.history,$
$u?.speechAct.speaker,$
$u?.speechAct.addressee,$
$accept, u?.time)) \geq 1$
$\bullet\ (ProcessAcceptance \fatsemi agent?.AddCommitments) \land$
$(action.ProcessNorm \fatsemi agent?.AddObligations)) \land$
$(\bigwedge accept : Accept;\ action : \downarrow Acting \mid$
$accept \in u?.speechAct.points \land$
$accept.accepting \in Delete \land$
$accept.accepting.commitment.action = action \land$
$\#(getProposeForAccept(agent?.history,$
$u?.speechAct.speaker,$
$u?.speechAct.addressee,$
$accept, u?.time)) \geq 1$
$\bullet\ (ProcessAcceptance \fatsemi agent?.DeleteCommitments) \land$
$(action.ProcessNorm \fatsemi agent?.DeleteObligations))$

FIGURE 8. *Policy 3*: Accepting a proposal leads to uptake shared commitments and obligations.

mitments entail. As shown in Figure 8, this policy is defined as a class inheriting from *ActionNorm* that defines the operations *ProcessAction* (which overrides the empty operation inherited from *ActionNorm*), *ProcessAcceptance* and *ProcessUtterance* (which overrides the abstract operation inherited from *Norm*).

The operation *ProcessUtterance* selects all acceptances in a given utterance and checks whether each of these is a valid reply to a past proposal (as indicated by the previously defined axiom *getProposeForAccept*). For each acceptance that satisfies this criterion, this operation invokes the operation *ProcessAcceptance*, which is followed by either the operation *AddCommitments* or *DeleteCommitments* (depending on whether the acceptance is adopting or discarding a commitment, respectively). In addition, these operations are performed concurrently with the operation *ProcessNorm* followed by either *AddObligations* or *DeleteObligations* (also depending on the type of operation in the acceptance).

Lastly, the operation *ProcessAcceptance* defines that for a given acceptance there exists a shared social commitment that will be later added to (or discarded from) the state of the agent. In the same manner, the operation *ProcessAction* defines that for any given action in which the agent is one of the performers then there is the obligation that the agent performs the action (an obligation that will be later added or discarded from the state of the agent).

Policy 4: Adopting and Discharging Obligations to Propose. One additional policy defines responsibilities about the adoption and discharge of obligations to action. This policy states that once a commitment to action is adopted then there is an agent that will propose its discharge. For example, if Alice has requested the time to Bob, and Bob has accepted the request, then Bob is responsible for proposing the discharge of the action. The class *ToProposeToDischarge* is the superclass of all actions that once accepted are to be proposed for discharge. As shown below, this class inherits from the class *ToSpeak* and specifies the agent variables discharger and discharged as the speaker and addressee of the speech act. In addition, this class specifies that the speech act communicates an illocutionary point to propose the discharge of the action.

ToProposeToDischarge

ToSpeak

$$discharger, discharged : \downarrow Agent$$

$$discharger = speaker \land$$
$$discharged = addressee$$
$$\exists\, p : \downarrow Propose \mid$$
$$p \in points$$
$$\bullet\ p.proposing \in Delete \land$$
$$p.proposing.commitment.action = self$$

Our fourth policy is specified by the class *ProposingToDischargePolicy* (shown in Figure 9), which inherits from *ActionNorm* and defines the operations *ProcessAction* (which overrides the operation inherited from *ActionNorm*) and *ProcessUtterance* (which overrides the operation inherited from *Norm*). In brief, the operation *ProcessUtterance* specifies that for each acceptance found in the utterance, such that there is a past propose matching the acceptance, then the action operation *ProcessAction* is performed followed by one of the agent operations *AddObligations* or *DeleteObligations* (depending on whether the acceptance is accepting to add or to delete a social commitment). Lastly, the operation *ProcessAction*

$$
\begin{array}{l}
\textbf{ProposingToDischargePolicy} \\
\restriction(\mathit{ProcessUtterance}, \mathit{ProcessAction}) \\
\mathit{ActionNorm} \\
\quad \textbf{ProcessAction} \\
\quad \mathit{agent?} : \downarrow \mathit{Agent} \\
\quad \mathit{sc?} : \mathit{SocialCommitment} \\
\quad \mathit{deleting?} : \downarrow \mathit{ProposingToDischarge} \\
\quad \mathit{result!} : \mathbb{P}\ \mathit{Obligation} \\
\quad \rule{3cm}{0.4pt} \\
\quad \exists\, \mathit{speak} : \mathit{Speaking} \mid \\
\quad\quad \mathit{speak} = (\mu\, \mathit{sa} : \mathit{Speaking} \mid \\
\quad\quad\quad\quad\quad\quad \mathit{sa.time} = \mathit{deleting?.time} \land \\
\quad\quad\quad\quad\quad\quad \mathit{sa.speaker} = \mathit{deleting?.discharger} \land \\
\quad\quad\quad\quad\quad\quad \mathit{sa.addressee} = \mathit{deleting?.discharged} \land \\
\quad\quad\quad\quad\quad\quad (\exists\, \mathit{propose} : \downarrow \mathit{Propose} \mid \\
\quad\quad\quad\quad\quad\quad\quad \mathit{propose} \in \mathit{sa.points} \land \\
\quad\quad\quad\quad\quad\quad\quad \mathit{propose.proposing} \in \mathit{Delete} \\
\quad\quad\quad\quad\quad\quad \bullet \exists\, \mathit{sc} : \mathit{SocialCommitment} \mid \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{sc.debtor} = \mathit{sc?.debtor} \land \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{sc.creditor} = \mathit{sc?.creditor} \land \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{sc.action} = \mathit{deleting?} \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ \mathit{sc} = \mathit{propose.proposing.commitment})) \\
\quad \bullet\ (\mathit{agent?} = \mathit{deleting?.discharger} \land \mathit{result!} = \{\mathit{speak}, \mathit{speak.voice}\}) \lor \\
\quad\quad (\mathit{agent?} = \mathit{deleting?.discharged} \land \mathit{result!} = \{\mathit{speak}, \mathit{speak.hear}\}) \\
\mathit{ProcessUtterance} \mathrel{\widehat{=}} [\, \mathit{agent?} : \downarrow \mathit{Agent};\ u? : \mathit{Utterance} \,] \bullet \\
\quad\quad (\bigwedge \mathit{action} : \downarrow \mathit{Acting} \mid \\
\quad\quad\quad \exists\, \mathit{accept} : \mathit{Accept} \mid \\
\quad\quad\quad\quad \mathit{accept} \in u?.\mathit{speechAct.points} \land \\
\quad\quad\quad\quad \mathit{accept.accepting} \in \mathit{Add} \land \\
\quad\quad\quad\quad \mathit{accept.accepting.commitment.action} = \mathit{action} \\
\quad\quad\quad \bullet\ \#(\mathit{getProposeForAccept}(\mathit{agent?.history}, u?.\mathit{speechAct.speaker}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad u?.\mathit{speechAct.addressee}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{accept}, u?.\mathit{time})) \geq 1 \\
\quad\quad \bullet\ \mathit{action.ProcessNorm} \mathbin{⨾} \mathit{agent?.AddObligations}) \land \\
\quad\quad (\bigwedge \mathit{action} : \downarrow \mathit{Acting} \mid \\
\quad\quad\quad \exists\, \mathit{accept} : \mathit{Accept} \mid \\
\quad\quad\quad\quad \mathit{accept} \in u?.\mathit{speechAct.points} \land \\
\quad\quad\quad\quad \mathit{accept.accepting} \in \mathit{Delete} \land \\
\quad\quad\quad\quad \mathit{accept.accepting.commitment.action} = \mathit{action} \\
\quad\quad\quad \bullet\ \#(\mathit{getProposeForAccept}(\mathit{agent?.history}, u?.\mathit{speechAct.speaker}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad u?.\mathit{speechAct.addressee}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{accept}, u?.\mathit{time})) \geq 1 \\
\quad\quad \bullet\ \mathit{action.ProcessNorm} \mathbin{⨾} \mathit{agent?.DeleteObligations})
\end{array}
$$

FIGURE 9. *Policy 4*: Committing to propose discharging a shared commitment to action.

specifies that given a *ProposingToDischarge* action[16] there exists an obligation in which the *discharger* agent is to propose to the *discharged* agent the discharge of a social commitment involving the action.

Societies. A basic assumption in our model is that collaborative agents voluntarily participate in normative societies, i.e., societies that specify the norms of behaviour that agents in the society are expected to follow (Conte and Castelfranchi 1995; Conte and Dellarocas 2001).[17] We conceptualize societies as normative entities indicating the joint activities in which member agents can participate. Societies unify all elements in our model.

Joint activities are sets of actions that are carried out by an ensemble of agents acting in coordination with each other toward achieving certain dominant goals (Clark 1996). The superclass of all joint activities in our model is the class *JointActivity*, which defines a sole variable *actions* (to refer to a set of joint actions) and an abstract operation *Interaction* (to define the ideal sequences of communications in the activity).

JointActivity

$$actions : \mathbb{P} \downarrow JointActing$$

$$Interaction \mathrel{\widehat=} [\,]$$

The class *Society* is the superclass of all societies in our model. This class defines variables referencing a set of norms, a set of joint activities and a set of the agents associated to the society (where all agents participating in any joint activity are part of the society). This class also specifies the operation *ExecuteNorms*, which invokes the operation *Process* for all agents with incoming utterances.

Society

$$agents : \mathbb{P} \downarrow Agent$$
$$norms : \mathbb{P} \downarrow Norm$$
$$activities : \mathbb{P} \downarrow JointActivity$$

$$\forall\, activity : activities \bullet \forall\, action : activity.actions \bullet \forall\, performer : action.performers \bullet performer \in agents$$

$$ExecuteNorms \mathrel{\widehat=} \bigwedge a : agents \mid \#a.inbox \geq 1 \bullet a.Process$$

Lastly, the class *PFPsociety* (below) is defined to denote those societies that have as their norms the conversation policies described so far.

[16]The class *ProposingToDischarge* is a subclass of *ToProposeToDischarge* and *CompositeActing*.

[17]In this view, agents not only have the autonomy to adopt norms but also the autonomy to abide or disregard them according to their assumed costs of obedience and transgression. Although our model does not explore that area, other more complex societies may define norms that restore equity and avoid potential injury by making ill-behaved agents liable for their actions.

PFPsociety

Society

INIT

$(\exists_1\ p : PFPpolicy1 \bullet p \in norms)$
$(\exists_1\ p : PFPpolicy2 \bullet p \in norms)$
$(\exists_1\ p : PFPpolicy3 \bullet p \in norms)$
$(\exists_1\ p : ProposingToDischargePolicy \bullet p \in norms)$

3. EXAMPLE

This section illustrates how our model can be used to model the interaction of agents in a joint activity, specifically, the Contract Net Protocol (Smith 1980), which is a task allocation mechanism often used in multi-agent systems.

An instance of the Contract Net Protocol (CNP) begins when a manager attempts to delegate actions by sending a request for bids to agents that could potentially perform these actions. Agents who are willing submit a bid showing their abilities and willingness to perform these actions. The manager then evaluates submitted bids and selects the most suitable bidder and awards the contract to that bidder (i.e., offers the execution of the actions). Once the contract is awarded, the acceptance of this awarding makes the awarded agent the contractor (i.e. the agent executing the actions). Finally, the protocol terminates when the contractor submits the results of the actions to the manager.[18]

To specify the CNP in terms of our model of conversations, it is necessary to identify the information, actions, social commitments and illocutionary points that are exchanged on interactions between managers and contractors. These are described below.

3.1. Information

There are five types of information that can be communicated in the CNP:

- the requirements (for producing a bid),
- the bids themselves,
- the contracts,
- the notification of the awarding or rejection of bids, and
- the results of executing a contract.

The class *Requirement* is a subclass of *Data* that specifies an action and its constraints (where the constraints could be action dependencies, maximum costs afforded, times of expected execution, and so on). The requirements for a bid are a set of instances of this class.

[18] Although this description is the more representative of the CNP minor variants exist. As described by R.G. Smith (1980), variations include, for example, those where the contractor transmits preliminary results while executing an action, and those with various contractors performing actions simultaneously.

Requirement
Data

$$action : \downarrow Action$$
$$constraints : \mathbb{P} \downarrow Data$$

The type definitions *BidItem* and *ContractItem* specify the types of the items that compose bids and contracts, respectively. These items are defined to have the same type as a requirement.

$$BidItem == Requirement$$
$$ContractItem == Requirement$$

The class *EvaluationResult* is used for notifying whether or not a bid has been awarded for execution. This class (which inherits from *Data*) solely defines a Boolean value to indicate the awarding (or not) of a contract.

EvaluationResult
Data

$$true : \mathbb{B}$$

Lastly, the class *ResultsItem* defines the type of the instances returned after executing a contract. This class inherits from *Requirement* and defines the variable *outcome* to hold the results of executing the inherited action.

ResultsItem
Requirement

$$outcome : \downarrow Data$$

3.2. Actions

There are three actions involved in the CNP:

- *To submit a bid*: in which a bidder creates and submits a bid to a manager.
- *To evaluate a bid*: in which a manager evaluates a bid and informs a bidder of the outcome of this evaluation, and
- *To execute a contract*: in which a bidder executes an awarded contract and submits its results to a manager.

Bidding. Bidding is an action in which an agent produces and announces to another agent a bid. This action is modelled as a class named *ToBid* that inherits from *ToProduce* and *ToProposeToDischarge*, and which declares the variables *bidder* (as the producer and discharger), *manager* (as the receiver and discharged), and *requirements* (as the abiding criteria for the production of the bid). It also defines the data produced and communicated as a bid (i.e., a set of objects of type *BidItem*).

ToBid

ToProduce
ToProposeToDischarge

$$manager : \downarrow Manager$$
$$bidder : \downarrow Contractor$$
$$requirements : \mathbb{P}_1\ Requirement$$

$$bidder = producer = discharger \land$$
$$manager = receiver = discharged \land$$
$$requirements = in$$
$$\exists\ bid : \mathbb{P}\ BidItem$$
$$\bullet\ bid = out$$

Based on this definition, the type *Bidding* is defined as the union of (i.e., as the polymorphic type of) the classes *ToBid*, *CompositeActing* and *ProposingToDischarge*.[19]

$$Bidding == ToBid \bigcup CompositeActing \bigcup ProposingToDischarge$$

Evaluating a Bid. Evaluating a bid is an action in which a manager announces to a bidder whether or not a submitted bid is awarded for execution. This action is defined as a class named *ToEvaluateBid* that inherits from *ToEvaluateBid* and *ToProposeToDischarge*, and which declares the variables *manager* (as the producer and discharger), *bidder* (as the receiver and discharged) and *bid* (as the bid submitted for evaluation). This definition also specifies that the awarding (or not) of a bid is communicated through an object of type *EvaluationResult*.

ToEvaluateBid

ToProduce
ToProposeToDischarge

$$manager : \downarrow Manager$$
$$bidder : \downarrow Contractor$$
$$bid : \mathbb{P}\ BidItem$$

$$manager = producer = discharger \land$$
$$bidder = receiver = discharged \land$$
$$bid = in$$
$$\exists\ awarded : EvaluationResult$$
$$\bullet\ \{awarded\} = out$$

Based on this definition, the action *EvaluatingBid* is defined as the union of the classes *ToEvaluateBid*, *CompositeActing* and *ProposingToDischarge*.

$$EvaluatingBid == ToEvaluateBid \bigcup CompositeActing \bigcup ProposingToDischarge$$

[19]Note that the time interval specified in the classes *CompositeActing* and *ProposingToDischarge* gets unified in this definition. This unification specifies that a proposal to discharge the action will occur within the interval when the action is performed.

Executing a Contract. Performing a contract is an action in which a contractor executes a contract and communicates the results to the manager. This action is defined as a class named *ToExecuteContract* that inherits from *ToProduce* and *ToProposeToDischarge* and which declares the variables *contractor* (as the producer and discharger), *manager* (as the receiver and discharged) and *contract* (as the actions to execute). This definition also specifies that this action results in a non-empty set of instances of type *ResultsItem*.

ToExecuteContract

ToProduce
ToProposeToDischarge

$$
\begin{array}{l}
manager : \downarrow Manager \\
contractor : \downarrow Contractor \\
contract : \mathbb{P}_1\ ContractItem
\end{array}
$$

$$
\begin{array}{l}
contractor = producer = discharger \land \\
manager = receiver = discharged \land \\
contract = in \\
\exists\, results : \mathbb{P}_1\ ResultsItem \\
\bullet\ results = out
\end{array}
$$

Based on this definition, the action *ExecutingContract* is defined as the union of the classes *ToExecuteContract*, *CompositeActing* and *ProposingToDischarge*.

$$ExecutingContract == ToExecuteContract \bigcup CompositeActing \bigcup ProposingToDischarge$$

3.3. Social Commitments

Three axioms identify the social commitments involving the aforementioned actions. As such, these axioms specify *commitments to bid*, *commitments to evaluate a bid*, and *commitments to execute a contract*.

Commitment to Bid. The axiom *isCommitmentToBid* is a function that receives a *Bidding* action and returns a social commitment that has as its creditor and debtor the manager and bidder of the action, and where the action of the commitment is the given *Bidding* action.

$$isCommitmentToBid : Bidding \rightarrow SocialCommitment$$

$$
\begin{array}{l}
\forall\, bidding : Bidding \\
\bullet\ \exists\, sc : SocialCommitment \mid \\
\quad sc.creditor = bidding.manager \land \\
\quad sc.debtor = bidding.bidder \land \\
\quad sc.action = bidding \\
\quad \bullet\ sc = isCommitmentToBid(bidding)
\end{array}
$$

Commitment to Evaluate a Bid. Along the same lines, the axiom *isCommitmentToEvaluateBid* is a function that receives an *EvaluatingBid* action and returns a social commitment that has as its creditor and debtor the bidder and manager of the action, and where the action of the commitment is the given *EvaluatingBid* action.

$$isCommitmentToEvaluateBid : EvaluatingBid \rightarrow SocialCommitment$$

$$\begin{aligned}
&\forall\, evaluating : EvaluatingBid \\
&\bullet\ \exists\, sc : SocialCommitment \mid \\
&\quad sc.creditor = evaluating.bidder \land \\
&\quad sc.debtor = evaluating.manager \land \\
&\quad sc.action = evaluating \\
&\quad \bullet\ sc = isCommitmentToEvaluateBid(evaluating)
\end{aligned}$$

Commitment to Execute a Contract. Lastly, the axiom *isCommitmentToExecuteContract* is a function that receives an *ExecutingContract* action and returns a social commitment that has as its creditor and debtor the manager and contractor of the action, where the action of the commitment is the given *EvaluatingBid* action.

$$isCommitmentToExecuteContract : ExecutingContract \rightarrow SocialCommitment$$

$$\begin{aligned}
&\forall\, executing : ExecutingContract \\
&\bullet\ \exists\, sc : SocialCommitment \mid \\
&\quad sc.creditor = executing.manager \land \\
&\quad sc.debtor = executing.contractor \land \\
&\quad sc.action = executing \\
&\quad \bullet\ sc = isCommitmentToExecuteContract(executing)
\end{aligned}$$

Illocutionary Points. This section specifies the composition of the illocutionary points used by agents to negotiate commitments in the CNP.

Proposing to Bid. CNP interactions begin when a manager requests that a prospective bidder produce and submit a bid that adheres to certain required criteria. This communication is supported by the axioms *isProposeToAdoptBidding* and *isInformRequirements*.

The axiom *isProposeToAdoptBidding* defines a function that receives a *Bidding* action and an interval time within which a reply is expected, and returns a proposal to adopt a social commitment to the bidding action.

$$isProposeToAdoptBidding : Bidding \times Interval \rightarrow \downarrow Propose$$

$$\begin{aligned}
&\forall\, bidding : Bidding;\ reply : Interval \\
&\bullet\ \exists\, propose : \downarrow Propose \mid \\
&\quad propose.proposing \in Add \land \\
&\quad propose.reply = reply \land \\
&\quad propose.proposing.commitment = isCommitmentToBid(bidding) \\
&\quad \bullet\ propose = isProposeToAdoptBidding(bidding, reply)
\end{aligned}$$

In addition, the axiom *isInformRequirements* is a function that receives a set of requirements as input and returns an inform containing such requirements.

$$isInformRequirements : \mathbb{P}_1\, Requirement \rightarrow Inform$$

$$\begin{aligned}
&\forall\, requirements : \mathbb{P}_1\, Requirement \\
&\bullet\ \exists\, inform : Inform \mid \\
&\quad inform.informing = requirements \\
&\quad \bullet\ inform = isInformRequirements(requirements)
\end{aligned}$$

Accepting or Rejecting to Bid. Once a request for bids has been issued, it is expected that it will be replied to with an acceptance or a rejection (as specified by the *Protocol for Proposals*). To support acceptances to such requests, the axiom *isAcceptToAdoptBidding* defines a function that receives a *Bidding* action and returns an acceptance to adopt committing to this action.

$$
\begin{array}{|l}
isAcceptToAdoptBidding : Bidding \rightarrow Accept \\
\hline
\forall\, bidding : Bidding \\
\bullet\ \exists\, accept : Accept \mid \\
\quad accept.accepting \in Add \land \\
\quad accept.accepting.commitment = isCommitmentToBid(bidding) \\
\quad \bullet\ accept = isAcceptToAdoptBidding(bidding)
\end{array}
$$

Agents receiving a request for bids can decline to commit to such a request (for example, if they are not capable of performing the requested actions, or if they cannot accommodate the execution of the actions under current state constraints). As such, the axiom *isRejectToAdoptBidding* defines a function that receives a *Bidding* action as input and returns a rejection to commit to this action.

$$
\begin{array}{|l}
isRejectToAdoptBidding : Bidding \rightarrow \downarrow Reject \\
\hline
\forall\, bidding : Bidding \\
\bullet\ \exists\, reply : \downarrow Reject \mid \\
\quad reply.rejecting \in Add \land \\
\quad reply.rejecting.commitment = isCommitmentToBid(bidding) \\
\quad \bullet\ reply = isRejectToAdoptBidding(bidding)
\end{array}
$$

Submitting a Bid for Evaluation. In the event that a request for bid is accepted, the bidder is now responsible for producing and submitting a bid to the manager.

The axiom *isProposeToDischargeBidding* specifies a proposal to discharge a commitment to bid. This axiom is a function that receives a *Bidding* action and an interval (within which a reply is expected), and returns a proposal to discharge a social commitment to do the given bidding action.

$$
\begin{array}{|l}
isProposeToDischargeBidding : Bidding \times Interval \rightarrow \downarrow Propose \\
\hline
\forall\, bidding : Bidding;\ reply : Interval \\
\bullet\ \exists\, propose : \downarrow Propose \mid \\
\quad propose.proposing \in Delete \land \\
\quad propose.reply = reply \land \\
\quad propose.proposing.commitment = isCommitmentToBid(bidding) \\
\quad \bullet\ propose = isProposeToDischargeBidding(bidding, reply)
\end{array}
$$

The axiom *isInformBid* (which informs a bid) is a function that receives a bid and returns an inform containing such a bid.

$$
\begin{array}{|l}
isInformBid : \mathbb{P}\, BidItem \rightarrow Inform \\
\hline
\forall\, bid : \mathbb{P}\, BidItem \\
\bullet\ \exists\, inform : Inform \mid \\
\quad inform.informing = bid \\
\quad \bullet\ inform = isInformBid(bid)
\end{array}
$$

Lastly, the axiom *isProposeToAdoptEvaluating* specifies a proposal to adopt a commitment to evaluate a bid. This axiom is a function that receives an *EvaluatingBid* action and an interval specifying the expected reply time, and returns a proposal to adopt a social commitment to perform this evaluating action.

$$
\begin{array}{|l}
isProposeToAdoptEvaluating : EvaluatingBid \times Interval \rightarrow \downarrow Propose \\
\hline
\forall\, evaluating : EvaluatingBid;\ reply : Interval \\
\bullet\ \exists\, propose : \downarrow Propose \mid \\
\quad propose.proposing \in Add \wedge \\
\quad propose.reply = reply \wedge \\
\quad propose.proposing.commitment = isCommitmentToEvaluateBid(evaluating) \\
\quad \bullet\ propose = isProposeToAdoptEvaluating(evaluating, reply)
\end{array}
$$

Accepting to Evaluate a Bid. Once a bid has been submitted for evaluation, it is expected that the manager receiving the bid will reply both to the proposal that the bidder is no longer committed to submit a bid, and to the proposal that he evaluates the just submitted bid.

The axiom *isAcceptToDischargeBidding* specifies an acceptance to discharge a commitment to bid. This axiom is a function that receives a *Bidding* action as input and returns an acceptance to discharge a social commitment to this bidding action.

$$
\begin{array}{|l}
isAcceptToDischargeBidding : Bidding \rightarrow Accept \\
\hline
\forall\, bidding : Bidding \\
\bullet\ \exists\, accept : Accept \mid \\
\quad accept.accepting \in Delete \wedge \\
\quad accept.accepting.commitment = isCommitmentToBid(bidding) \\
\quad \bullet\ accept = isAcceptToDischargeBidding(bidding)
\end{array}
$$

Lastly, the axiom *isAcceptToAdoptEvaluating* specifies an acceptance to adopt a commitment to evaluate a bid. This axiom is a function that receives an *EvaluatingBid* action as input and returns an acceptance to adopt a social commitment to perform this evaluating action.

$$
\begin{array}{|l}
isAcceptToAdoptEvaluating : EvaluatingBid \rightarrow Accept \\
\hline
\forall\, evaluating : EvaluatingBid \\
\bullet\ \exists\, accept : Accept \mid \\
\quad accept.accepting \in Add \wedge \\
\quad accept.accepting.commitment = isCommitmentToEvaluateBid(evaluating) \\
\quad \bullet\ accept = isAcceptToAdoptEvaluating(evaluating)
\end{array}
$$

Awarding a Contract. After evaluating the merits of a bid, a manager communicates to the bidder whether or not she is awarded the execution of the contract. To communicate this awarding, the manager sends:

- a proposal to discharge the commitment that he evaluates the bid,
- the affirmative result of the evaluation,
- the awarded contract, and
- a proposal to adopt a commitment in which the bidder does this contract.

The axiom *isProposeToDischargeEvaluating* specifies a proposal to discharge the commitment to evaluate a bid. This axiom is a function that receives an *EvaluatingBid* action

and an interval specifying an expected reply time, and returns a proposal to discharge a social commitment to do the evaluation.

$$
\begin{array}{|l}
isProposeToDischargeEvaluating : EvaluatingBid \times Interval \rightarrow \downarrow Propose \\
\hline
\forall\, evaluating : EvaluatingBid;\ reply : Interval \\
\bullet\ \exists\, propose : \downarrow Propose \mid \\
\quad propose.proposing \in Delete \land \\
\quad propose.reply = reply \land \\
\quad propose.proposing.commitment = isCommitmentToEvaluateBid(evaluating) \\
\quad \bullet\ propose = isProposeToDischargeEvaluating(evaluating, reply)
\end{array}
$$

The axiom *isInformEvaluation* (which informs the result of an evaluation) is a function that receives an instance of type *EvaluationResult* and returns an inform for this result.

$$
\begin{array}{|l}
isInformEvaluation : EvaluationResult \rightarrow Inform \\
\hline
\forall\, decision : EvaluationResult \\
\bullet\ \exists\, inform : Inform \mid \\
\quad inform.informing = \{decision\} \\
\quad \bullet\ inform = isInformEvaluation(decision)
\end{array}
$$

Likewise, the axiom *isInformContract* (which informs a contract) is a function that receives a contract and returns an inform.

$$
\begin{array}{|l}
isInformContract : \mathbb{P}\, ContractItem \rightarrow Inform \\
\hline
\forall\, contract : \mathbb{P}\, ContractItem \\
\bullet\ \exists\, inform : Inform \mid \\
\quad inform.informing = contract \\
\quad \bullet\ inform = isInformContract(contract)
\end{array}
$$

Lastly, the axiom *isProposeToAdoptExecuting* specifies a proposal to adopt a commitment to execute a contract. This axiom is a function that receives an instance of type *ExecutingContract* and an interval indicating an expected time of reply, and returns a proposal to adopt this executing action.

$$
\begin{array}{|l}
isProposeToAdoptExecuting : ExecutingContract \times Interval \rightarrow \downarrow Propose \\
\hline
\forall\, executing : ExecutingContract;\ reply : Interval \\
\bullet\ \exists\, propose : \downarrow Propose \mid \\
\quad propose.proposing \in Add \land \\
\quad propose.reply = reply \land \\
\quad propose.proposing.commitment = isCommitmentToExecuteContract(executing) \\
\quad \bullet\ propose = isProposeToAdoptExecuting(executing, reply)
\end{array}
$$

Accepting the Evaluation of a Bid. After a manager announces the outcome of an evaluation, the (failed or awarded) bidder will acknowledge this outcome by accepting that the manager is no longer committed to evaluate the bid. To that end, the axiom *isAcceptToDischargeEvaluating* is a function that returns an illocutionary point accepting to discharge a commitment to evaluate a bid.

$$
\begin{array}{|l}
\hline
\mathit{isAcceptToDischargeEvaluating} : \mathit{EvaluatingBid} \rightarrow \mathit{Accept} \\
\hline
\forall\, \mathit{evaluating} : \mathit{EvaluatingBid} \\
\bullet\ \exists\, \mathit{accept} : \mathit{Accept} \mid \\
\quad \mathit{accept.accepting} \in \mathit{Delete} \wedge \\
\quad \mathit{accept.accepting.commitment} = \mathit{isCommitmentToEvaluateBid}(\mathit{evaluating}) \\
\quad \bullet\ \mathit{accept} = \mathit{isAcceptToDischargeEvaluating}(\mathit{evaluating}) \\
\end{array}
$$

Accepting or Rejecting the Awarding of a Contract. If the contract is awarded, the bidder must confirm whether she will execute the contract by accepting or rejecting the proposal for its execution.[20] As such, the axioms *isAcceptToAdoptExecuting* and *isRejectToAdoptExecuting* define the illocutionary points for accepting and rejecting the adoption of a commitment to execute a contract, respectively.

$$
\begin{array}{|l}
\hline
\mathit{isAcceptToAdoptExecuting} : \mathit{ExecutingContract} \rightarrow \mathit{Accept} \\
\hline
\forall\, \mathit{executing} : \mathit{ExecutingContract} \\
\bullet\ \exists\, \mathit{accept} : \mathit{Accept} \mid \\
\quad \mathit{accept.accepting} \in \mathit{Add} \wedge \\
\quad \mathit{accept.accepting.commitment} = \mathit{isCommitmentToExecuteContract}(\mathit{executing}) \\
\quad \bullet\ \mathit{accept} = \mathit{isAcceptToAdoptExecuting}(\mathit{executing}) \\
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{isRejectToAdoptExecuting} : \mathit{ExecutingContract} \rightarrow \downarrow \mathit{Reject} \\
\hline
\forall\, \mathit{executing} : \mathit{ExecutingContract} \\
\bullet\ \exists\, \mathit{reject} : \downarrow \mathit{Reject} \mid \\
\quad \mathit{reject.rejecting} \in \mathit{Add} \wedge \\
\quad \mathit{reject.rejecting.commitment} = \mathit{isCommitmentToExecuteContract}(\mathit{executing}) \\
\quad \bullet\ \mathit{reject} = \mathit{isRejectToAdoptExecuting}(\mathit{executing}) \\
\end{array}
$$

Executing a Contract and Submitting Results. In the event that a contractor accepts to execute the awarded action, it is expected that the results of the action will be communicated to the manager. To that end, the axioms *isProposeToDischargeExecuting* and *isInformResults* define the illocutionary points for proposing to discharge executing the contract, and for submitting the results of its execution, respectively.

$$
\begin{array}{|l}
\hline
\mathit{isProposeToDischargeExecuting} : \mathit{ExecutingContract} \times \mathit{Interval} \rightarrow \downarrow \mathit{Propose} \\
\hline
\forall\, \mathit{executing} : \mathit{ExecutingContract};\ \mathit{reply} : \mathit{Interval} \\
\bullet\ \exists\, \mathit{propose} : \downarrow \mathit{Propose} \mid \\
\quad \mathit{propose.proposing} \in \mathit{Delete} \wedge \\
\quad \mathit{propose.reply} = \mathit{reply} \wedge \\
\quad \mathit{propose.proposing.commitment} = \mathit{isCommitmentToExecuteContract}(\mathit{executing}) \\
\quad \bullet\ \mathit{propose} = \mathit{isProposeToDischargeExecuting}(\mathit{executing}, \mathit{reply}) \\
\end{array}
$$

[20] One reason to reject the awarding of a contract is if agents lack the necessary resources for its execution. As explained by J. Ferber (1999), one strategy to allocate resources for bids is to secure these resources at the time of bidding. This strategy, which he called *early commitment*, allows the straightforward execution of a contract (since resources are allocated beforehand), but at the cost of their sub-optimal use if the awarding does not happen. A second strategy consists of submitting a bid without securing the resources needed for execution. The disadvantage of this strategy, which he called *late commitment*, is that agents may be unable to allocate the resources for executing a contract if these resources are scarce at the time of the awarding. In such cases, agents may have no other option than to reject the awarding of the contract.

$$isInformResults : \mathbb{P}\, ResultsItem \rightarrow Inform$$

$$\forall\, results : \mathbb{P}\, ResultsItem \bullet \exists\, inform : Inform \mid inform.informing = results \bullet inform = isInformResults(results)$$

Accepting the Results of a Contract. Lastly, once the contract has been executed and the results submitted to the manager, the manager (if he finds these results satisfactory) will acknowledge their acceptance. As such, the axiom *isAcceptToDischargeExecuting* defines the illocutionary point that accepts the discharge of executing a contract.

$$isAcceptToDischargeExecuting : ExecutingContract \rightarrow Accept$$

$$\forall\, executing : ExecutingContract \bullet \exists\, accept : Accept \mid accept.accepting \in Delete \land accept.accepting.commitment = isCommitmentToExecuteContract(executing) \bullet accept = isAcceptToDischargeExecuting(executing)$$

3.4. Participants

The CNP involves two types of participants: a manager and a contractor. Our model specifies the interactions of these participants as utterances constrained by committal preconditions. This means that (for example) for a contractor to submit a bid, she is required to have an obligation to submit a bid.

Manager. The class *Manager* (which is shown in Figure 10 and Figure 11) is a subclass of *Agent* that defines operations for requesting a bid, for accepting a bid for evaluation, for notifying whether a bid is awarded as a contract, and for receiving the results of executing a contract.

Requesting a Bid. The operation *RequestingBid* defines the behaviour for requesting a bid. This operation is defined as the sequential composition of the operations *ProposeToAdoptBidding* and *SendUtterance*.

The operation *ProposeToAdoptBidding* returns a speech act where the speaker and the addressee are the manager and bidder of the provided *Bidding* action (and where the speaker is also the current manager instance executing the operation). This definition also specifies that the resulting speech act contains a proposal to adopt a commitment to bid and an inform indicating the requirements for the bid.[21] Lastly, the operation *SendUtterance*, which is inherited from the class *Agent*, communicates a speech act (the one resulting from *ProposeToAdoptBidding*) between the speaker and the addressee of the speech act.

Evaluating a Bid. The operation *EvaluatingBid* defines the behaviour for accepting a bid for evaluation. In brief, this operation receives a *Bidding* action for discharge and an *EvaluatingBid* action for adoption, and evaluates whether these actions specify the same agent as their bidder, and whether the manager (i.e., the current instance) holds obligations

[21] In this example the *Inform* illocutionary point informs the same requirements as those listed in the *Bidding* action being proposed. We modelled this redundancy for clarity of the example. Other more optimal definitions may not include it. From the point of view of the specification, this redundancy does not create a significant overhead given the referential nature of Object-Z variables.

FIGURE 10. Definition of the class *Manager* (part 1 of 2)

to reply to a proposal to discharge the *Bidding* action and to a proposal to adopt the *EvaluatingBid* action (as specified by the axioms *existsReplyToProposeToDischargeBidding* and *existsReplyToProposeToAdoptEvaluating*, which are defined next). The fulfilment of these conditions leads to the operations *AcceptToDischargeBidding* and *AcceptToAdoptEvaluating* (which define speech acts for accepting to discharge bidding, and for accepting to adopt the evaluation of a bid, respectively) followed by the operation *SendUtterance*.

The axiom *existsReplyToProposeToDischargeBidding* is a function that assesses whether a provided set of obligations contains a *Speaking* action in which the manager is able to reply to a proposal to discharge a given *Bidding* action.

Figure 11. Definition of the class *Manager* (part 2 of 2)

$$existsReplyToProposeToDischargeBidding : \mathbb{P}\, Obligation \times Bidding \rightarrow \mathbb{B}$$

$$\begin{array}{l}
\forall\, obligations : \mathbb{P}\, Obligation;\ bidding : Bidding \\
\bullet\ existsReplyToProposeToDischargeBidding(obligations, bidding) \Leftrightarrow \\
\quad (\exists\, spoken, replied : Interval \mid \\
\qquad spoken.from \leq now \leq spoken.until \wedge \\
\qquad replied.from \leq now \leq replied.until \\
\quad\ \bullet \exists\, propose : \downarrow Propose \mid \\
\qquad\ propose = isProposeToDischargeBidding(bidding, replied) \\
\qquad\ \bullet \exists\, o : obligations \\
\qquad\quad \bullet \exists\, speaking : Speaking \mid \\
\qquad\qquad speaking.speaker = bidding.manager \wedge \\
\qquad\qquad speaking.addressee = bidding.bidder \wedge \\
\qquad\qquad speaking.time = spoken \wedge \\
\qquad\qquad speaking.points = isReplyTo(propose) \\
\qquad\qquad \bullet\ speaking = o)
\end{array}$$

Likewise, the axiom *existsReplyToProposeToAdoptEvaluating* assesses whether a provided set of obligations contains a *Speaking* action in which the manager is able to reply to a proposal to adopt the given *EvaluatingBid* action.

$$\begin{array}{l}
existsReplyToProposeToAdoptEvaluating : \\
\qquad\qquad \mathbb{P}\, Obligation \times EvaluatingBid \rightarrow \mathbb{B}
\end{array}$$

$$\begin{array}{l}
\forall\, obligations : \mathbb{P}\, Obligation;\ evaluating : EvaluatingBid \\
\bullet\ existsReplyToProposeToAdoptEvaluating(obligations, evaluating) \Leftrightarrow \\
\quad (\exists\, spoken, replied : Interval \mid \\
\qquad spoken.from \leq now \leq spoken.until \wedge \\
\qquad replied.from \leq now \leq replied.until \\
\quad\ \bullet \exists\, propose : \downarrow Propose \mid \\
\qquad\ propose = isProposeToAdoptEvaluating(evaluating, replied) \\
\qquad\ \bullet \exists\, o : obligations \\
\qquad\quad \bullet \exists\, speaking : Speaking \mid \\
\qquad\qquad speaking.speaker = evaluating.manager \wedge \\
\qquad\qquad speaking.addressee = evaluating.bidder \wedge \\
\qquad\qquad speaking.time = spoken \wedge \\
\qquad\qquad speaking.points = isReplyTo(propose) \\
\qquad\qquad \bullet\ speaking = o)
\end{array}$$

Awarding a Contract. The operation *AwardingContract* defines the behaviour for awarding the execution of a contract. Besides defining an instance of type *EvaluationResult* that holds a true value, this operation receives an *EvaluatingBid* action for discharge and an *ExecutingContract* action for adoption, and evaluates whether these actions specify the same agent as their bidder and contractor (respectively), and whether the manager holds an obligation to propose discharging the *EvaluatingBid* action (as specified below by the axiom *existsSpeakToProposeToDischargeEvaluating*). The fulfilment of these conditions leads to the operations *ProposeToDischargeEvaluating* and *ProposeToAdoptExecuting* (which define speech acts for proposing to discharge the evaluation of a bid, and for proposing to adopt the execution of a contract, respectively) followed by the operation *SendUtterance*.

The axiom *existsSpeakToProposeToDischargeEvaluating* is a function that assesses whether a provided set of obligations contains a *Speaking* action in which the manager is able to reply

to a proposal to discharge the given *EvaluatingBid* action.

Rejecting a Bid. The operation *RejectingBid* specifies the behaviour for rejecting a bid (which is the alternative outcome to awarding the execution of a contract). Besides defining an instance of type *EvaluationResult* that holds a false value, this operation receives an *EvaluatingBid* action for discharge, and evaluates whether the manager holds an obligation to propose discharging the action (as specified by the previously defined axiom *existsSpeakToProposeToDischargeEvaluating*). The fulfilment of these conditions leads to the sequential composition of the operations *ProposeToDischargeEvaluating* (which was also described earlier) and *SendUtterance*.

Accepting Results of a Contract. Lastly, the operation *AcceptingResults* specifies the behaviour for accepting a proposal to discharge the execution of a contract. This operation evaluates whether the manager holds an obligation in which he replies to a proposal to discharge the execution of a contract (as specified below by the axiom *existsReplyToProposeToDischargeExecuting*). If true, this condition leads to the operations *AcceptToDischargeExecuting* (which defines a speech act for accepting to discharge the execution of a contract) and *SendUtterance*.

The axiom *existsReplyToProposeToDischargeExecuting* is a function that assesses whether a provided set of obligations contains a *Speaking* action in which the manager is able to reply to a proposal to discharge the given *ExecutingContract* action.

$$
\begin{array}{|l}
existsReplyToProposeToDischargeExecuting : \\
\qquad \mathbb{P}\, Obligation \times ExecutingContract \to \mathbb{B} \\
\hline
\forall\, obligations : \mathbb{P}\, Obligation;\ executing : ExecutingContract \\
\bullet\ existsReplyToProposeToDischargeExecuting(obligations, executing) \Leftrightarrow \\
\quad (\exists\, spoken, replied : Interval \mid \\
\qquad spoken.from \leq now \leq spoken.until \land \\
\qquad replied.from \leq now \leq replied.until \\
\quad \bullet \exists\, propose : \downarrow Propose \mid \\
\qquad propose = isProposeToDischargeExecuting(executing, replied) \\
\qquad \bullet \exists\, o : obligations \\
\qquad\quad \bullet \exists\, speaking : Speaking \mid \\
\qquad\qquad speaking.speaker = executing.manager \land \\
\qquad\qquad speaking.addressee = executing.contractor \land \\
\qquad\qquad speaking.time = spoken \land \\
\qquad\qquad speaking.points = isReplyTo(propose) \\
\qquad\qquad \bullet\ speaking = o)
\end{array}
$$

Contractor. The class Contractor (which is shown in Figure 12, Figure 12 and Figure 12) is a subclass of *Agent* that defines operations for

- accepting and rejecting requests for bids,
- submitting bids for evaluation,
- accepting and rejecting the awarding of contracts, and
- submitting the result of executing contracts.

Accepting a Request for Bid. The operation *AcceptingToBid* specifies the behaviour for committing to submit a bid. This operation evaluates whether the bidder holds an obligation

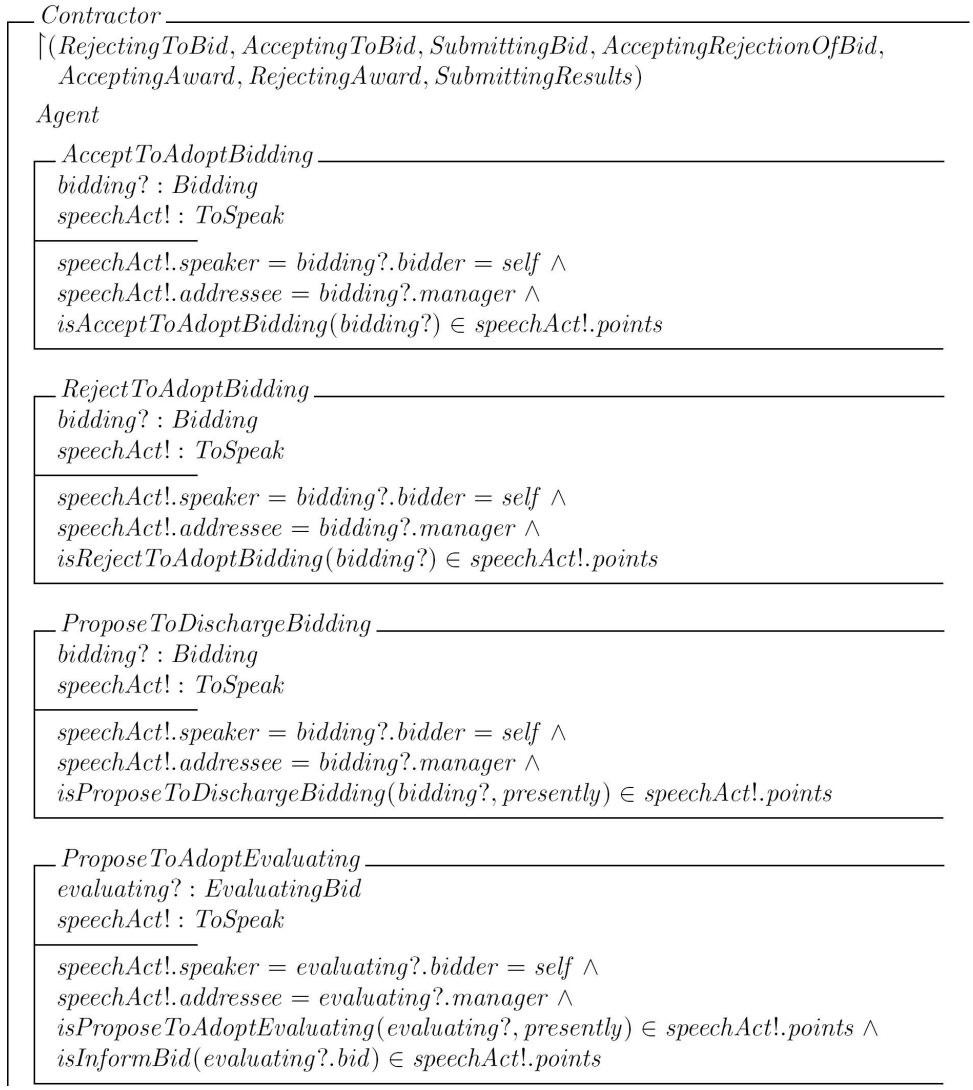


FIGURE 12. Definition of the class *Contractor* (part 1 of 3)

to reply to a proposal to submit a bid (as specified below by the axiom *existsReplyToProposeToAdoptBidding*). If true this condition leads to the operations *AcceptToAdoptBidding* (which defines a speech act for accepting to submit a bid) and *SendUtterance*.

The axiom *existsReplyToProposeToAdoptBidding* is a function that assesses whether a provided set of obligations contains a *Speaking* action in which the bidder is able to reply to a proposal to adopt the given *Bidding* action.

AcceptToDischargeEvaluating

$evaluating? : EvaluatingBid$
$speechAct! : ToSpeak$

$speechAct!.speaker = evaluating?.bidder = self \land$
$speechAct!.addressee = evaluating?.manager \land$
$isAcceptToDischargeEvaluating(evaluating?) \in speechAct!.points$

AcceptToAdoptExecuting

$executing? : ExecutingContract$
$speechAct! : ToSpeak$

$speechAct!.speaker = executing?.contractor = self \land$
$speechAct!.addressee = executing?.manager \land$
$isAcceptToAdoptExecuting(executing?) \in speechAct!.points$

RejectToAdoptExecuting

$executing? : ExecutingContract$
$speechAct! : ToSpeak$

$speechAct!.speaker = executing?.contractor = self \land$
$speechAct!.addressee = executing?.manager \land$
$isRejectToAdoptExecuting(executing?) \in speechAct!.points$

ProposeToDischargeExecuting

$executing? : ExecutingContract$
$speechAct! : ToSpeak$
$results : \mathbb{P}\, ResultsItem$

$\forall\, c : executing?.contract$
$\bullet\ \exists\, r : results \bullet r.action = c.action \land r.constraints = c.constraints$
$\#results = \#executing?.contract$
$speechAct!.speaker = executing?.contractor = self \land$
$speechAct!.addressee = executing?.manager \land$
$isProposeToDischargeExecuting(executing?, presently) \in speechAct!.points \land$
$isInformResults(results) \in speechAct!.points$

Figure 13. Definition of the class *Contractor* (part 2 of 3)

$existsReplyToProposeToAdoptBidding : \mathbb{P}\, Obligation \times Bidding \to \mathbb{B}$

$\forall\, obligations : \mathbb{P}\, Obligation;\ bidding : Bidding$
$\bullet\ existsReplyToProposeToAdoptBidding(obligations, bidding) \Leftrightarrow$
$(\exists\, spoken, replied : Interval \mid$
$spoken.from \leq now \leq spoken.until \land$
$replied.from \leq now \leq replied.until$
$\bullet\ \exists\, propose : \downarrow Propose \mid$
$propose = isProposeToAdoptBidding(bidding, replied)$
$\bullet\ \exists\, o : obligations$
$\bullet\ \exists\, speaking : Speaking \mid$
$speaking.speaker = bidding.bidder \land$
$speaking.addressee = bidding.manager \land$
$speaking.time = spoken \land$
$speaking.points = isReplyTo(propose)$
$\bullet\ speaking = o)$

$$
\begin{array}{l}
\mathit{AcceptingToBid} \mathrel{\widehat=} \lfloor\, \mathit{bidding?} : \mathit{Bidding} \mid \\
\quad \mathit{existsReplyToProposeToAdoptBidding}(\mathrm{dom}\ \mathit{obligations}, \mathit{bidding?}) \,\rfloor \bullet \\
\quad \mathit{AcceptToAdoptBidding} \fatsemi \mathit{SendUtterance} \\
\mathit{RejectingToBid} \mathrel{\widehat=} [\, \mathit{bidding?} : \mathit{Bidding} \mid \\
\quad \mathit{existsReplyToProposeToAdoptBidding}(\mathrm{dom}\ \mathit{obligations}, \mathit{bidding?}) \,] \bullet \\
\quad \mathit{RejectToAdoptBidding} \fatsemi \mathit{SendUtterance} \\
\mathit{SubmittingBid} \mathrel{\widehat=} [\, \mathit{bidding?} : \mathit{Bidding};\ \mathit{evaluating?} : \mathit{EvaluatingBid} \mid \\
\quad \mathit{bidding?.manager} = \mathit{evaluating?.manager} \land \\
\quad \mathit{existsSpeakToProposeToDischargeBidding}(\mathrm{dom}\ \mathit{obligations}, \mathit{bidding?}) \,] \bullet \\
\quad (\mathit{ProposeToDischargeBidding} \land \\
\quad\ \mathit{ProposeToAdoptEvaluating}) \fatsemi \mathit{SendUtterance} \\
\mathit{AcceptingRejectionOfBid} \mathrel{\widehat=} [\, \mathit{evaluating?} : \mathit{EvaluatingBid}; \\
\qquad\qquad\qquad\qquad \mathit{executing} : \mathit{ExecutingContract} \mid \\
\quad \mathit{executing.manager} = \mathit{evaluating?.manager} \land \\
\quad \mathit{executing.contract} \subseteq \mathit{evaluating?.bid} \land \\
\quad \mathit{existsReplyToProposeToDischargeEvaluating}(\mathrm{dom}\ \mathit{obligations}, \mathit{evaluating?}) \land \\
\quad \neg\ \mathit{existsReplyToProposeToAdoptExecuting}(\mathrm{dom}\ \mathit{obligations}, \mathit{executing}) \,] \bullet \\
\quad \mathit{AcceptToDischargeEvaluating} \fatsemi \mathit{SendUtterance} \\
\mathit{AcceptingAward} \mathrel{\widehat=} [\, \mathit{evaluating?} : \mathit{EvaluatingBid};\ \mathit{executing?} : \mathit{ExecutingContract} \mid \\
\quad \mathit{evaluating?.manager} = \mathit{executing?.manager} \land \\
\quad \mathit{existsReplyToProposeToDischargeEvaluating}(\mathrm{dom}\ \mathit{obligations}, \mathit{evaluating?}) \land \\
\quad \mathit{existsReplyToProposeToAdoptExecuting}(\mathrm{dom}\ \mathit{obligations}, \mathit{executing?}) \,] \bullet \\
\quad (\mathit{AcceptToDischargeEvaluating} \land \\
\quad\ \mathit{AcceptToAdoptExecuting}) \fatsemi \mathit{SendUtterance} \\
\mathit{RejectingAward} \mathrel{\widehat=} [\, \mathit{evaluating?} : \mathit{EvaluatingBid};\ \mathit{executing?} : \mathit{ExecutingContract} \mid \\
\quad \mathit{evaluating?.manager} = \mathit{executing?.manager} \land \\
\quad \mathit{existsReplyToProposeToDischargeEvaluating}(\mathrm{dom}\ \mathit{obligations}, \mathit{evaluating?}) \land \\
\quad \mathit{existsReplyToProposeToAdoptExecuting}(\mathrm{dom}\ \mathit{obligations}, \mathit{executing?}) \,] \bullet \\
\quad (\mathit{AcceptToDischargeEvaluating} \land \mathit{RejectToAdoptExecuting}) \fatsemi \mathit{SendUtterance} \\
\mathit{SubmittingResults} \mathrel{\widehat=} [\, \mathit{executing?} : \mathit{ExecutingContract} \mid \\
\quad \mathit{existsSpeakToProposeToDischargeExecuting}(\mathrm{dom}\ \mathit{obligations}, \mathit{executing?}) \,] \bullet \\
\quad \mathit{ProposeToDischargeExecuting} \fatsemi \mathit{SendUtterance}
\end{array}
$$

FIGURE 14. Definition of the class *Contractor* (part 3 of 3)

Rejecting a Request for Bid. Along the same lines, the operation *RejectingToBid* specifies the behaviour of a bidder that rejects committing to submit a bid. This operation checks whether the bidder holds an obligation to reply to a proposal to submit a bid (as defined by the previously defined axiom *existsReplyToProposeToAdoptBidding*), which leads to the operations *RejectToAdoptBidding* (which defines a speech act rejecting to submit a bid) and *SendUtterance*.

Submitting a Bid for Evaluation. The operation *SubmittingBid* specifies the behaviour for submitting a bid for evaluation. This operation receives a *Bidding* action for discharge and an *EvaluatingBid* action for adoption, and evaluates whether these actions specify the same agent as their manager, and whether the bidder (i.e., the current bidder instance) holds an obligation to propose discharging the *Bidding* action (as specified below by the

axiom *existsSpeakToProposeToDischargeBidding*). The fulfilment of these conditions leads to the conjunctive composition of the operations *ProposeToDischargeBidding* and *ProposeToAdoptEvaluating* (which define speech acts for proposing to discharge the submission of a bid, and for proposing to adopt the execution of a contract, respectively) followed by the operation *SendUtterance*.

The axiom *existsSpeakToProposeToDischargeBidding* is a function that assesses whether a set of obligations contains a *Speaking* action in which the bidder is able to propose to discharge a given *Bidding* action.

$$
\begin{array}{|l}
existsSpeakToProposeToDischargeBidding : \mathbb{P}\, Obligation \times Bidding \rightarrow \mathbb{B} \\
\hline
\forall\, obligations : \mathbb{P}\, Obligation;\ bidding : Bidding \\
\bullet\ existsSpeakToProposeToDischargeBidding(obligations, bidding) \Leftrightarrow \\
\quad (\exists\, spoken, replied : Interval \mid \\
\qquad spoken.from \leq now \leq spoken.until \land \\
\qquad replied.from \leq now \leq replied.until \\
\quad \bullet\ \exists\, o : obligations \\
\qquad \bullet\ \exists\, speaking : Speaking \mid \\
\qquad\quad speaking.speaker = bidding.bidder \land \\
\qquad\quad speaking.addressee = bidding.manager \land \\
\qquad\quad speaking.time = spoken \land \\
\qquad\quad isProposeToDischargeBidding(bidding, replied) \in speaking.points \\
\qquad\quad \bullet\ speaking = o)
\end{array}
$$

Accepting the Rejection of a Bid. The operation *AcceptingRejectionOfBid* specifies the behaviour for accepting that a bid was not awarded for execution. This operation receives an *EvaluatingBid* action and evaluates whether the bidder holds an obligation to reply to a proposal to discharge this action. In addition, this operation defines an *ExecutingContract* action that is compatible to the *EvaluatingBid* action (i.e., the executing action covers all possible contracts that could result from the evaluating action), and evaluates that the bidder does not hold a proposal to adopt the *ExecutingContract* action (as specified below by the axioms *existsReplyToProposeToDischargeEvaluating* and *existsReplyToProposeToAdoptExecuting*). The satisfaction of these conditions leads to the composition of the operations *AcceptToDischargeEvaluating* (which defines a speech act accepting to discharge the evaluation of a bid) and *SendUtterance*.

The axiom *existsReplyToProposeToDischargeEvaluating* is a function that assesses whether a provided set of obligations contains a *Speaking* action in which the bidder is able to reply to a proposal to discharge a given *EvaluatingBid* action.

$$existsReplyToProposeToDischargeEvaluating : \mathbb{P}\, Obligation \times EvaluatingBid \rightarrow \mathbb{B}$$

$$\begin{array}{l}
\forall\, obligations : \mathbb{P}\, Obligation;\ evaluating : EvaluatingBid \\
\bullet\ existsReplyToProposeToDischargeEvaluating(obligations, evaluating) \Leftrightarrow \\
\quad (\exists\, spoken, replied : Interval \mid \\
\qquad spoken.from \leq now \leq spoken.until \wedge \\
\qquad replied.from \leq now \leq replied.until \\
\quad\ \bullet \exists\, propose : \downarrow Propose \mid \\
\qquad\ propose = isProposeToDischargeEvaluating(evaluating, replied) \\
\quad\ \bullet \exists\, o : obligations \\
\quad\quad \bullet \exists\, speaking : Speaking \mid \\
\qquad\quad\ speaking.speaker = evaluating.bidder \wedge \\
\qquad\quad\ speaking.addressee = evaluating.manager \wedge \\
\qquad\quad\ speaking.time = spoken \wedge \\
\qquad\quad\ speaking.points = isReplyTo(propose) \\
\quad\quad\ \bullet\ speaking = o)
\end{array}$$

The axiom *existsReplyToProposeToAdoptExecuting* assesses whether a provided set of obligations contains a *Speaking* action in which the bidder is able to reply to a proposal to adopt a given *ExecutingBid* action.

$$existsReplyToProposeToAdoptExecuting : \mathbb{P}\, Obligation \times ExecutingContract \rightarrow \mathbb{B}$$

$$\begin{array}{l}
\forall\, obligations : \mathbb{P}\, Obligation;\ executing : ExecutingContract \\
\bullet\ existsReplyToProposeToAdoptExecuting(obligations, executing) \Leftrightarrow \\
\quad (\exists\, spoken, replied : Interval \mid \\
\qquad spoken.from \leq now \leq spoken.until \wedge \\
\qquad replied.from \leq now \leq replied.until \\
\quad\ \bullet \exists\, propose : \downarrow Propose \mid \\
\qquad\ propose = isProposeToAdoptExecuting(executing, replied) \\
\quad\ \bullet \exists\, o : obligations \\
\quad\quad \bullet \exists\, speaking : Speaking \mid \\
\qquad\quad\ speaking.speaker = executing.contractor \wedge \\
\qquad\quad\ speaking.addressee = executing.manager \wedge \\
\qquad\quad\ speaking.time = spoken \wedge \\
\qquad\quad\ speaking.points = isReplyTo(propose) \\
\quad\quad\ \bullet\ speaking = o)
\end{array}$$

Accepting an Awarded Contract. The operation *AcceptingAward* specifies the behaviour for accepting to execute a contract. This operation receives an *EvaluatingBid* action for discharge and an *ExecutingContract* action for adoption, and evaluates whether these actions specify the same agent as their manager, and whether the bidder holds obligations to reply to a proposal to discharge the *EvaluatingBid* action, and to reply to a proposal to adopt the *ExecutingContract* action (as indicated by the previously defined axioms *existsReplyToProposeToDischargeEvaluating* and *existsReplyToProposeToAdoptExecuting*). The fulfilment of these conditions leads to the operations *AcceptToDischargeEvaluating* (which was also described earlier) and *AcceptToAdoptExecuting* (which defines a speech act accepting to adopt the execution of a bid) followed by the operation *SendUtterance*.

Rejecting an Awarded Contract. Along the same lines, the operation *RejectingAward* specifies the behaviour of a bidder that rejects to adopt committing to execute a contract. This operation receives an *EvaluatingBid* action for discharge and an *ExecutingContract* action for adoption, and evaluates whether these actions specify the same agent as their manager, and whether the bidder holds obligations to reply to a proposal to discharge the *EvaluatingBid* action, and to reply to a proposal to adopt the *ExecutingContract* action (as specified by the previously defined axioms *existsReplyToProposeToDischargeEvaluating* and *existsReplyToProposeToAdoptExecuting*). If true, these conditions lead to the composition of the operations *AcceptToDischargeEvaluating* (which was also described above) and *RejectToAdoptExecuting* (which defines a speech act rejecting to adopt executing a contract) followed by the operation *SendUtterance*.

Submitting Results of Executing a Contract. The operation *SubmittingResults* specifies the behaviour for proposing to discharge being committed to the execution of a contract and for sending the results of its execution. This operation receives an *ExecutingContract* action for discharge, and evaluates whether the bidder holds an obligation to propose to discharge this action (as specified below by the axiom *existsSpeakToProposeToDischargeExecuting*). If true, this condition leads to the operations *ProposeToDischargeExecuting* (which defines a speech act that proposes to discharge the execution of the contract and communicates the results of its execution) and *SendUtterance*.

The axiom *existsSpeakToProposeToDischargeExecuting* is a function that assesses whether a provided set of obligations contains a *Speaking* action in which the bidder can propose to discharge the *ExecutingBid* action.

$$
\begin{array}{|l}
existsSpeakToProposeToDischargeExecuting : \\
\qquad \mathbb{P}\, Obligation \times ExecutingContract \to \mathbb{B} \\
\hline
\forall\, obligations : \mathbb{P}\, Obligation;\ executing : ExecutingContract \\
\bullet\ existsSpeakToProposeToDischargeExecuting(obligations, executing) \Leftrightarrow \\
\quad (\exists\, spoken, replied : Interval \mid \\
\qquad spoken.from \leq now \leq spoken.until \land \\
\qquad replied.from \leq now \leq replied.until \\
\quad \bullet\ \exists\, o : obligations \\
\qquad \bullet\ \exists\, speaking : Speaking \mid \\
\qquad\quad speaking.speaker = executing.contractor \land \\
\qquad\quad speaking.addressee = executing.manager \land \\
\qquad\quad speaking.time = spoken \land \\
\qquad\quad isProposeToDischargeExecuting(executing, replied) \in speaking.points \\
\qquad\quad \bullet\ speaking = o)
\end{array}
$$

This section presented the communicational operations that agents in the roles of managers and contractors can perform (as long as their committal preconditions are met). However, these operations are disembodied of any concrete interaction. The next section will show how these operations are assembled into structured contract net interactions.

3.5. The Contract Net Protocol as a Joint Activity

The class *ContractNet* (which is shown in Figure 15) is a subclass of *JointActivity* that specifies the interactions that can occur in the CNP. This class defines two participants (a *manager* and a *contractor*) and three actions in which they participate (*bidding*, *evaluating* and *executing*). This class also defines that the actions specified in the *bid* are a subset of

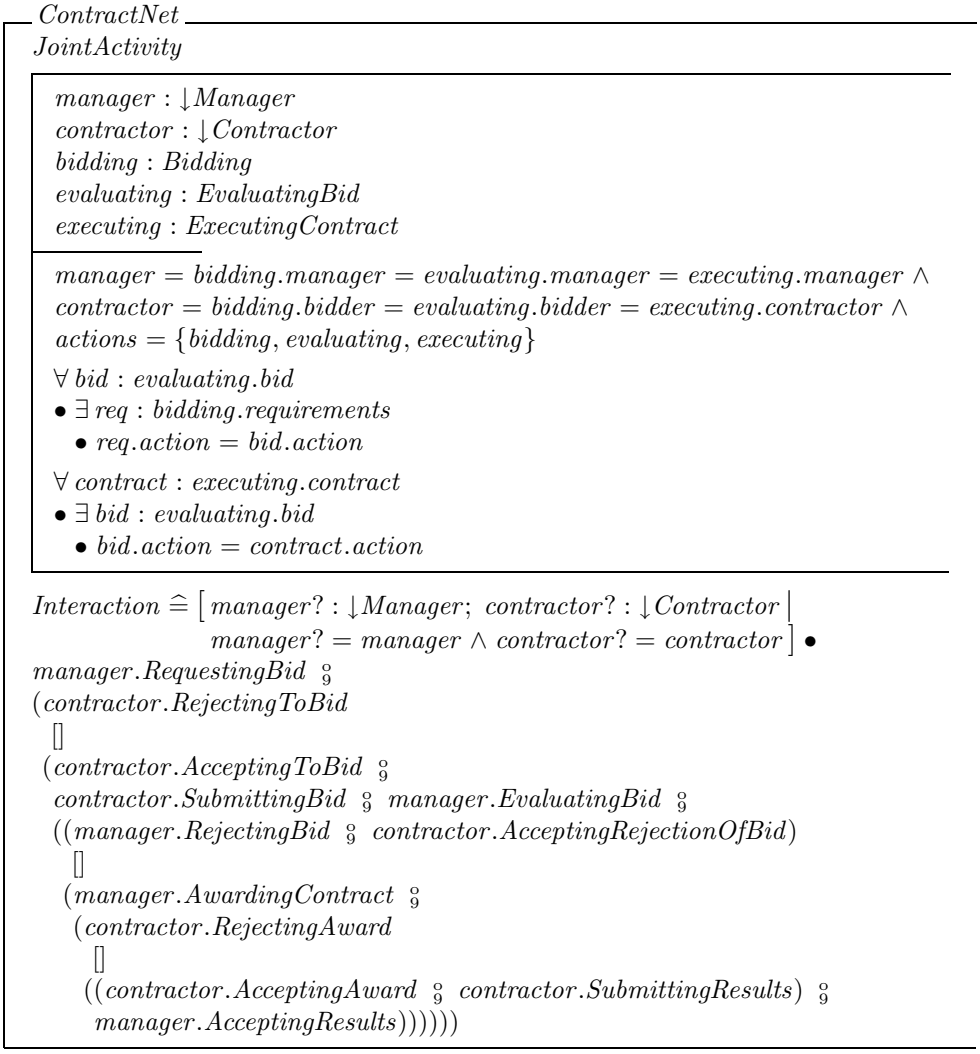


FIGURE 15. Definition of the Contract Net Protocol as a joint activity.

those in the *requirements*, and that the actions in the *contract* are a subset of those in the *bid*.

The operation *Interaction* defines the sequences of interdependent agent operations making the allowed interactions for the activity. This operation (which is illustrated as a conversation protocol in Figure 16) specifies that a request for bid from a manager is followed either by a rejection or an acceptance from the bidder.[22] In the case of a rejection, no other participation follows, thus signalling the end of the interaction. On the other hand, the bidder's acceptance to bid is followed by a submission of a bid, and the manager's acceptance to evaluate it. At this point, the manager either rejects the bid (which if accepted by the con-

[22]Although it is not explicitly modelled, we assume that counterproposals are followed by a rejection.

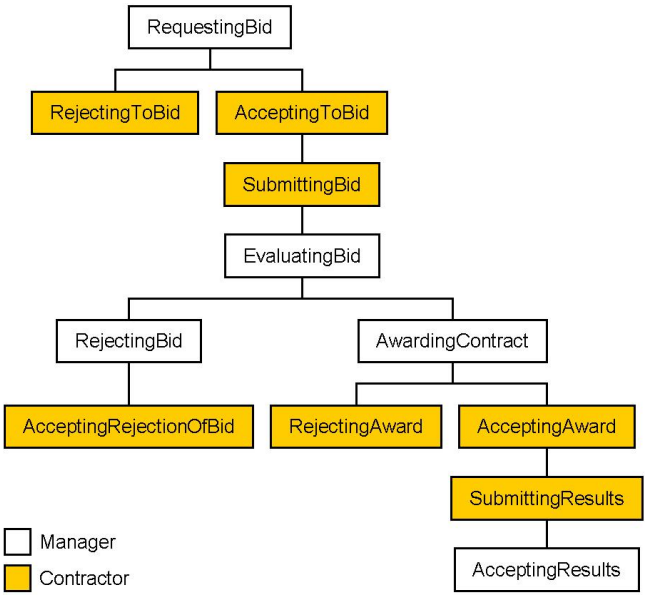


Figure 16. Conversation protocol of interactions in the *Contract Net* joint activity.

tractor leads to the end of the interaction) or awards it as the contract. In the case of being awarded the contract, the contractor either rejects the awarding (ending the interaction) or accepts the awarding. Lastly, an acceptance is followed by the submission of results and the manager's acceptance of those results.

3.6. Contract Net Society

The class *ContractNetSociety* is specified as a subclass of *PFPsociety* that defines *ContractNet* as a joint activity where a manager requests to one or more bidders to submit a bid, and also that the bids abide by the same requirements.[23]

ContractNetSociety

PFPsociety

$$
\begin{aligned}
ContractNet \mathrel{\widehat=} [\, & contractNet : \mathbb{P}_1\ ContractNet \mid \\
& contractNet \subseteq activities \land \\
& (\exists\, manager : \downarrow Manager;\ requirements : \mathbb{P}_1\ Requirement \\
& \quad \bullet \forall\, cn : contractNet \\
& \quad\quad \bullet\ cn.manager = manager \land \\
& \quad\quad\quad cn.bidding.requirements = requirements) \,] \bullet \\
& \bigwedge cn : contractNet \bullet cn.Interaction
\end{aligned}
$$

3.7. Example Conversation: Executing a Contract

Figure 17 shows a UML interaction diagram for an interaction in the contract net activity. Specifically, it shows a conversation that begins with a request for bid and advances until

[23]This definition was simplified to allow for variations of the CNP, such as those in which various contractors are awarded the execution of actions (Smith 1980).

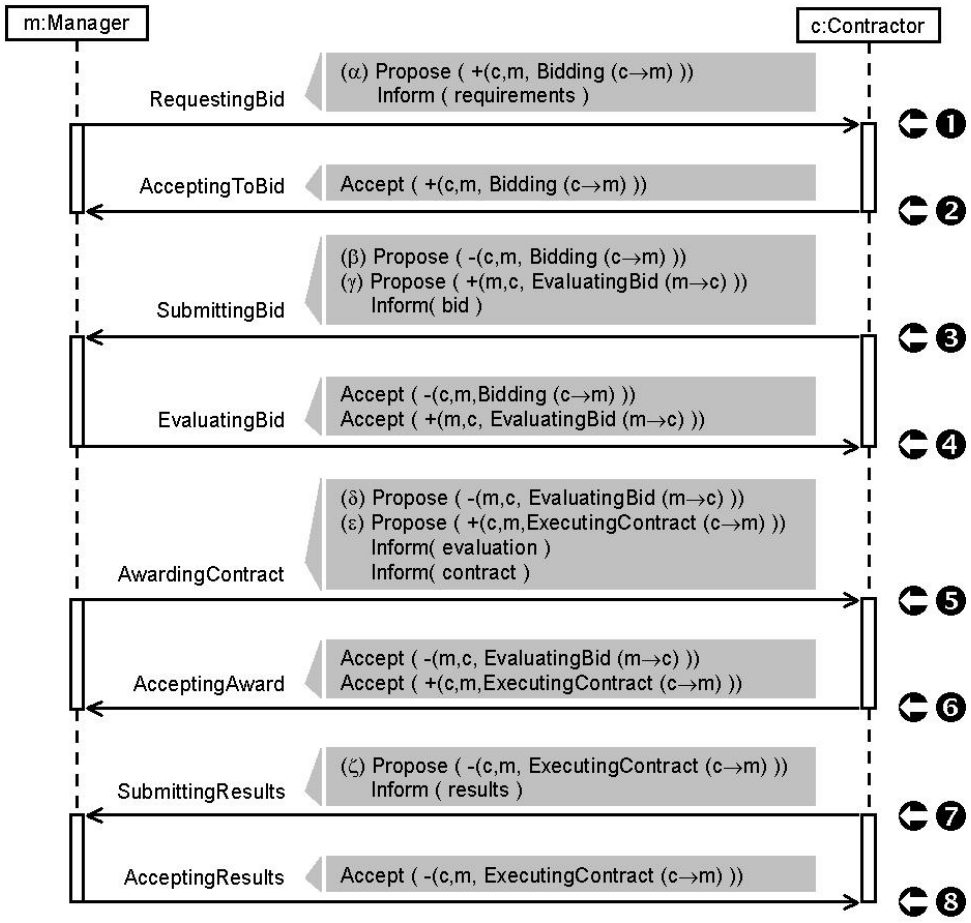


FIGURE 17. UML interaction diagram for a Contract Net Protocol conversation.

the contract is executed and its results submitted. This conversation is specified by the sequence of the operations *RequestingBid*, *AcceptingToBid*, *SubmittingBid*, *EvaluatingBid*, *AwardingContract*, *AcceptingAward*, *SubmittingResults* and *AcceptingResults*. Figures 18 and 19 show the state of shared social commitments and obligations on the manager and the contractor as this conversation evolves.

Requesting a Bid. As shown in Figure 17, the interaction begins with an utterance from the manager (identified as m) to a contractor (identified as c) in which he requests that she submit a bid based on the given requirements.

As specified in m's operation *RequestingBid*, this speech act contains a *Propose* illocutionary point (labelled α), proposing the adoption of a shared social commitment in which c is responsible to m for an action *Bidding* in which c performs and informs the results of the action to m (as before, the representation used in this figure has been simplified for clarity). As shown in Figure 18, the uttering of this proposal triggers the following conversational policy:

- Policy 1 (the uttering of a proposal commits the addressee to reply to the proposal):

FIGURE 18. State of shared social commitments and obligations of the manager and contractor in the Contract Net conversation example (part 1 of 2).

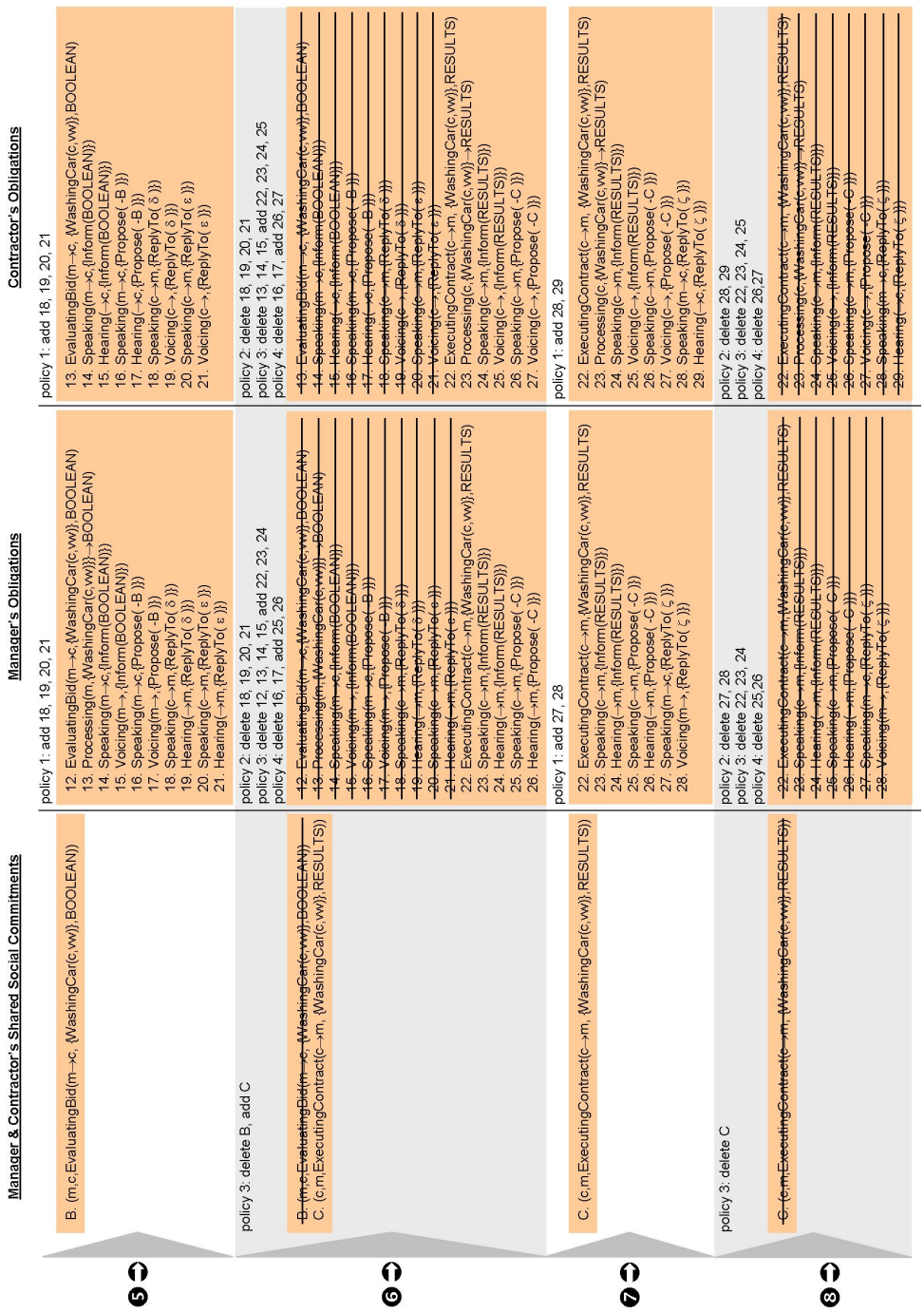


FIGURE 19. State of shared social commitments and obligations of the manager and contractor in the Contract Net conversation example (part 2 of 2).

the uttering of proposal α results in the adoption of obligations in which c replies to m's proposal α (added as obligations 1 and 2 in Figure 18 on both the manager and the contractor).[24]

Accepting to Bid. The next interaction (labelled as interaction 2 in Figure 17) specifies the execution of c's operation *AcceptingToBid*, in which she accepts committing to submit a bid (if there is an obligation to reply to a request for bid—which exists as obligations 1 and 2). Uttering this acceptance results in the application of the following policies:

- Policy 2 (replying to a proposal discharges the obligation to reply): the acceptance to uptake the operation proposed in a discharges the obligation to reply to α (which deletes obligations 1 and 2 in Figure 18 on both the manager and the contractor).
- Policy 3 (accepting a proposal causes the uptake of the proposed operation): the acceptance to uptake the operation proposed in α causes the adoption of the proposed commitment, in this case to submit a bid (added as commitment A in Figure 18). In addition, this acceptance results in the adoption of obligations to perform the joint action. As such, the contractor adopts obligations to produce and communicate a bid (obligations 3 to 6), and the manager adopts obligations to receive it (obligations 3 to 5).[25]
- Policy 4 (accepting a *ProposingToDischarge* action obligates the discharger to propose its discharge): the acceptance to adopt the action *Bidding* (which is a subtype of *ProposingToDischarge*) results in the adoption of obligations in which c (the *discharger*) is to propose to m (the *discharged*) discharging the action. These obligations are added as obligations 6 and 7 on the manager, and 7 and 8 on the contractor.

Submitting a Bid. The next interaction (labelled as interaction 3 in Figure 17) specifies the execution of c's operation *SubmittingBid*, in which a) she proposes to discharge the commitment to submit a bid (if there is an obligation in which c proposes to discharge the commitment that she submit a bid—which exists as obligations 7 and 8); b) she informs a bid; and c) she proposes to adopt a commitment in which m evaluates this bid. The uttering of these proposals (which are labelled β and γ) triggers the following conversational policies:

- Policy 1 (the uttering of a proposal commits the addressee to reply to the proposal): the uttering of proposal β results in the adoption of obligations in which m replies to β (added as obligations 8 and 9 on the manager, and 9 and 10 on the contractor), and
- Policy 1 (*ditto*): the uttering of proposal γ results in the adoption of obligations in which m replies to γ (added as obligations 10 and 11 on the manager, and 11 and 12 on the contractor).

Accepting a Bid for Evaluation. The next interaction (labelled as interaction 4 in Figure 17) specifies the execution of m's operation *EvaluatingBid*, in which m accepts to discharge the commitment that c submit a bid, and accepts to evaluate the submitted bid. These acceptances are uttered if obligations exist in which m replies both to a proposal to discharge submitting a bid (which exist as obligations 8 and 9) and to a proposal to adopt evaluating a bid (which exist as obligations 10 and 11). The uttering of these acceptances triggers the following conversational policies:

[24] In the case of the contractor the acquired obligation is for voicing a reply, and in the case of the manager for hearing it.

[25] In this example, the action requested is that of the contractor washing a VW car.

- Policy 2 (replying to a proposal discharges the obligations to reply): the acceptance to uptake the operation proposed in β discharges the obligations to reply to β (thus deleting obligations 8 and 9 on the manager, and 9 and 10 on the contractor).
- Policy 2 (*ditto*): the acceptance to uptake the operation proposed in γ discharges the obligations to reply to γ (thus deleting obligations 10 and 11 on the manager, and 11 and 12 on the contractor).
- Policy 3 (accepting a proposal causes the uptake of the proposed operation): the acceptance to uptake the operation proposed in β causes the discharge of the commitment to submit a bid (labelled as commitment A in Figure 18) and all corresponding obligations (i.e., obligations 3, 4 and 5 on the manager, and 3 to 6 on the contractor).
- Policy 3 (*ditto*): the acceptance to uptake the operation proposed in γ causes the adoption of a commitment in which m evaluates a bid for c (added as commitment B), and its corresponding obligations (where the manager is obligated to evaluate a bid and communicate the result of this evaluation—which are obligations 12 to 15—and the contractor is obligated to hear this evaluation—as described by obligations 13 to 15).
- Policy 4 (accepting to discharge a *ProposingToDischarge* action discards the obligations in which the discharger of the action is to propose its discharge): the acceptance to discharge the commitment to submit a bid results in the discharge of the obligations in which c is to propose discharging the commitment that she submits a bid (which deletes obligations 6 and 7 on the manager, and 7 and 8 on the contractor), and lastly
- Policy 4 (accepting to adopt a *ProposingToDischarge* action obligates the discharger of the action to propose its discharge): the acceptance to adopt the action to evaluate a bid results in the adoption of obligations in which m (the *discharger*) is to propose c (the *discharged*) to discharge this action (which adds obligations 16 and 17 on both the manager and the contractor).

Awarding a Contract. The next interaction (labelled as interaction 5 in Figure 17) specifies the execution of m's operation *AwardingContract*, in which m proposes to discharge that he evaluates a bid, informs the result of the evaluation, informs a contract, and proposes that the contractor adopt executing the given contract. The committal precondition for this utterance is that there exist obligations in which m is to propose discharging the commitment that he evaluates a bid for c (which are obligations 16 and 17). The uttering of these proposals (which are labelled δ and ϵ) triggers the following conversational policies:

- Policy 1 (the uttering of a proposal obligates the addressee to reply to the proposal): the uttering of proposal δ results in the adoption of obligations in which c replies to δ (added as obligations 18 and 19 in Figure 19 on both the manager and the contractor), and
- Policy 1 (*ditto*): the uttering of proposal ϵ results in the adoption of obligations in which c replies to ϵ (added as obligations 20 and 21 on both the manager and the contractor).

Accepting the Awarding of a Contract. The next interaction (labelled as interaction 6 in Figure 17) specifies the execution of c's operation *AcceptingAward*, which specifies that c accepts the discharge of the commitment that m evaluates a bid for c (if there are obligations in which c replies to a proposal to discharge this commitment—which exist as obligations 18 and 19), and that c accepts to adopt executing a contract (if there are obligations in which c replies to a proposal to adopt executing a contract—which exists as obligations 20 and 21). The uttering of these acceptances results in the application of the following policies:

- Policy 2 (replying to a proposal discharges the obligation to reply): the acceptance to

uptake the operation proposed in δ discharges the obligations to reply to δ (which deletes obligations 18 and 19 on both the manager and the contractor).

- Policy 2 (*ditto*): the acceptance to uptake the operation proposed in ϵ discharges the obligations to reply to ϵ (which deletes obligations 20 and 21 in both the manager an the contractor).
- Policy 3 (accepting a proposal causes the uptake of the proposed operation): the acceptance to uptake the operation proposed in δ causes the discharge of the commitment to evaluate a bid (labelled as commitment B) as well as its corresponding obligations (which are obligations 12 to 15 on the manager, and 13 to 15 on the contractor).
- Policy 3 (*ditto*): the acceptance to uptake the operation proposed in ϵ causes the adoption of a commitment in which c executes a contract for m (added as shared commitment C). The adoption of this commitment also results in the adoption of obligations in which the contractor executes the contract and informs its results (added as obligations 22 to 25) and the manager receives such results (added as obligations 22 to 24).
- Policy 4 (accepting to discharge a *ProposingToDischarge* action discharges the obligations in which the discharger proposes the discharge of the action): the acceptance to discharge the commitment to evaluate a bid results in the discharge of the obligations in which m is to propose discharging the commitment that he evaluates a bid (which deletes obligations 16 and 17 on both the manager and the contractor), and lastly,
- Policy 4 (accepting to adopt a *ProposingToDischarge* action obligates the discharger to propose the discharge of the action): the acceptance to adopt the action to execute a contract results in the adoption of obligations in which emphc (the discharger) is to propose to m (the discharged) discharging this action (which adds obligations 25 and 26 on the manager, and 26 and 27 on the contractor).

Submitting Results of Executing a Contract. The next interaction (labelled as interaction 7 in Figure 17) specifies the execution of c's operation *SubmittingResults*, which specifies that c proposes to m the discharge of the commitment that she executes the contract (if there are obligations in which she proposes to discharge the action—which exist as obligations 25 and 26). The uttering of this proposal (labelled ζ) triggers the following conversational policy:

- Policy 1 (the uttering of a proposal obligates the addressee to reply to the proposal): the uttering of proposal ζ results in the obligations in which m replies to ζ (which adds obligations 27 and 28 on the manager, and 28 and 29 on the contractor).

Accepting the Results of a Contract. The last interaction in this conversation (labelled as interaction 8 in Figure 17) indicates the execution of m's operation *AcceptingResults*, which specifies that m accepts to discharge the execution of a contract. This acceptance is uttered if obligations exist in which m replies to a proposal to discharge the execution of the contract (which exist as obligations 27 and 28). The uttering of this acceptance results in the application of the following policies:

- Policy 2 (replying to a proposal discharges the obligations to reply): the acceptance to uptake the operation proposed in ζ discharges the obligations to reply to ζ (thus deleting obligations 27 and 28 on the manager, and 28 and 29 on the contractor).
- Policy 3 (accepting a proposal causes the uptake of the proposed operation): the acceptance to uptake the operation proposed in ζ causes the discharge of the commitment to execute a contract (labelled as commitment C) as well as its corresponding obligations (which are obligations 22 to 24 on the manager, and 22 to 25 on the contractor), and lastly,

- Policy 4 (accepting to discharge a *ProposingToDischarge* action discards the obligations in which the discharger is to propose discharging the action): the acceptance to discharge the commitment to execute a contract results in the discharge of the obligations in which m is to propose discarding the commitment to execute the contract (therefore deleting obligations 25 and 26 on the manager, and 26 and 27 on the contractor).

At this point, the interaction ends leaving none of the shared social commitments and obligations adopted during the interaction, thus indicating that this conversation does not result on commitments and obligations that outlive the activity.

4. DISCUSSION AND RELATED WORK

The major contribution of our work is to have specified a model for structuring conversations using conversation policies whose principles are the negotiation of shared social commitments to action. It is important to point out that the conversations defined in our model would be no different than *ad hoc* conversation protocols if they were not supported by a strict semantics. To support this claim, we presented an example conversation showing how our model provides for structured CNP conversations based on the negotiation of commitments.[26]

To recap, joint activities specify the constraints that conversations should abide by. In particular they specify the sequencing of agent participations (i.e., the conversational moves), defining the order in which commitments to action are negotiated. The preconditions of a conversation are the commitments and obligations that are not created during the interaction but are required by the agent participations throughout the conversation; likewise, post-conditions are commitments and obligations that are created during the interaction but are not discharged when the conversation ends. Interactions do not define a unique set of pre- and post-conditions, as it is possible that different message sequences in the interaction may require/yield different pre/post-conditions. In the case of the CNP interaction, there were no preconditions or post-conditions.[27] Also, the interaction in this example was modelled with a minimum of flexibility. Other more versatile (but elaborated) specifications may define counterproposals (which could likely be defined recursively), or may optimize consecutive utterances made by one agent (for example, the bidder's *AcceptingToBid* and *SubmitBid*) into one speech act. This flexibility makes it apparent that conversations could either evolve in a straightforward manner (as in the current example) or could digress to refine their subject of negotiation (for example, by issuing counterproposals). In addition, agents could be programmed at different levels of sophistication to handle one, a few, or all of these sequences (as long as these sequences contain complete PFP instances, i.e., pairs of proposals and a matching rejection or acceptance). Therefore, agent implementations could stretch from those agents that follow the most straightforward conversation (and which may not even have any internal representation of the commitments negotiated) to those with rational engines, allowing them to extend and refine their conversations to account for the context of occurrence. In any event, these agents should be able to seamlessly interact as long as the conversation abides by the interaction specification.

Although our model could be seen as an improvement over traditional approaches to rep-

[26]We describe in (Flores, 2002) a formal proof supporting this conversation.

[27]In contrast, a "getting married" activity may establish post-conditions (e.g., commitments to love and be faithful to one's spouse), while a "getting divorced" activity may likely require as preconditions commitments produced by the "getting married" activity.

resenting conversations protocols, which do not specify any formal semantics beyond those of their corresponding graphical representations[28], ours is not the only framework to model conversations. For example, P. McBurney and S. Parsons (2002) propose a high-level formal framework for conversations based on the typology of human dialogues for argumentation described by D.N. Walton and E.C.W. Krabbe (Walton and Krabbe 1995). Although our approach to conversations may be seen as more applied, there are conceptual parallels between this framework and our model, such as the notions of dialogic and semantic commitments (which are the notions portrayed by our policies 1, 2 and 4, and policy 3, respectively). Another approach to conversations is the *commitment machines* model (Yolum and Singh 2001), which is designed as a semantic framework for conversation policies. Our model is similar to *commitment machines* in that the uttering of speech acts declares social commitments, and that these social commitments are adopted and discharged as the conversation evolves; but we differ in that our model explicitly advances conversations through the negotiation of fine-grained dialogic commitments (for example, to speak, voice and hear). We believe that our model's advantage is that it provides more for autonomy of agents by implementing a negotiation mechanism for the uptake of commitments, which is especially important when utterances are intended to commit agents other than the speaker.

5. CONCLUSION AND FUTURE WORK

At present, our model shows a mechanism for mutual agreement to establish responsibilities toward the execution of actions. However, this model is based exclusively on observable behaviour, and it is unrelated to the cognition guiding the actions of goal-directed agents. The importance of social commitments is that they provide for a principled way to connect the external world of interactions with the internal world of individual rational action. As such, one main avenue for future research is to investigate the value of social commitments to bridge the concepts of rationality (which are inherently private) and conversations (which are public social phenomena). Our model could be enhanced by exploring theories of individual social action (Castelfranchi 1998), specifically those dealing with the modelling of deliberate normative agents, that is, agents that guide their behaviour by reasoning about the norms in their society (Castelfranchi, et al. 2000). Moreover, by being able to reason about norms, agents could infer the normative reasoning of other agents based on their observable communications, which allows predicting and influencing future behaviour. This type of agent could then be applied to open environments where the coordination of action is regulated and globally optimized by monitoring agents (e.g., Pechoucek and Norrie 2000).

ACKNOWLEDGEMENTS

We are grateful to Douglas Norrie, Michal Pechoucek, Martyn Fletcher, Frank Maurer, Jörg Denzinger, Brian Gaines and the researchers at the Intelligent Systems Group at the University of Calgary for their comments on our model. We also thank the anonymous reviewers for their thoughtful suggestions. This work was supported by the Natural Sciences and Engineering Council of Canada (*NSERC*), Smart Technologies, Inc., the Alberta Software Engineering Research Consortium (*ASERC*) and the Alberta Informatics Circle of Research Excellence (*iCORE*).

[28]Conversation protocols are usually represented using state-transition diagrams (e.g., Winograd and Flores 1986; Bradshaw et al. 1997) and Petri Nets (e.g., Ferber 1999; Cost et al. 1999).

REFERENCES

AUSTIN, J.L. 1962. How to Do Things with Words. Harvard University Press.

BARBUCEANU, M. and FOX, M.S. 1995. COOL: A Language for Describing Coordination in Multi-Agent Systems. Proceedings of the First International Conference on Multi-Agent Systems, V. Lesser (Ed.), AAAI Press, June 1995, pp. 17-25.

BRADSHAW, J.M., DUTFIELD, S., BENOIT, P. and WOOLLEY, J.D. 1997. KAoS: Toward An Industrial-Strength Open Agent Architecture. Software Agents, J.M. Bradshaw (Ed.), AAAI Press, pp. 375-418.

CASTELFRANCHI, C. 1998. Modelling Social Action for AI Agents. Artificial Intelligence, 103, pp. 157-182.

CASTELFRANCHI, C. DIGNUM, F., JONKER, C.M. and TREUR, J. 2000. Deliberate Normative Agents: Principles and Architecture. Intelligent Agents VI - Proceedings of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL 1999). Lecture Notes in AI 1757, Springer Verlag, Berlin, pp. 206-220.

CLARK, H.H. 1996. Using language. Cambridge University Press.

CONTE, R. and CASTELFRANCHI, C. 1995. Cognitive and Social Action. University College London Press.

CONTE R. and DELLAROCAS, C. 2001. Social Order in Multiagent Systems. Kluwer Academic Publishers.

COST, R.S., CHEN, Y., FININ, T., LABROU, Y. and PENG, Y. 1999. Modeling Agent Conversations with Colored Petri Nets. Proceedings of the Third International Conference in Autonomous Agents (Agents '99), Workshop on Specifying and Implementing Conversation Policies, M. Greaves and J. Bradshaw (Eds.), Seattle, Washington, pp. 59-66.

FAGIN, R., HALPERN, J.Y., MOSES Y. and VARDI, M.Y. 1995. Reasoning About Knowledge. The MIT Press.

FERBER, J. 1999. Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Publishing Co.

FLORES, R.A. 2002. A Theory of Software Agent Conversations. Ph.D. dissertation, Department of Computer Science, University of Calgary, Canada.

GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts, Addison Wesley Publishing Co.

GREAVES, M., HOLMBACK, H. and BRADSHAW, J. 1999. What is a Conversation Policy? Third International Conference in Autonomous Agents, Workshop on Specifying and Implementing Conversation Policies, M. Greaves and J. Bradshaw (Eds.), Seattle, Washington, pp. 1-9.

JENNINGS, N.R. and WOOLDRIDGE, M.J. 1998. Agent Technology: Foundations, Applications and Markets. Springer-Verlag.

MCBURNEY, P. and PARSONS, S. 2002. Games that agents play: A formal framework for dialogues between autonomous agents. Journal of Logic, Language and Information. Special Issue on Logic and Games.

PECHOUCEK, M., and NORRIE, D.H. 2000. Knowledge Structures for Reflective Multi-Agent Systems: On Reasoning about other Agents. Report 538. University of Calgary, Department of Mechanical and Manufacturing Engineering.

SEARLE, J. 1969. Speech Acts. Cambridge University Press.

SINGH, M.P., RAO, A.S. and GEORGEFF, M.P. 1999. Formal Methods in DAI: Logic-Based Representation and Reasoning. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, G. Weiss (Ed.), The MIT Press, pp. 331-376.

SMITH, G. 2000. The Object-Z Specification Language. Kluwer Academic Publishers.

SMITH, R.G. 1980. The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver. IEEE Transactions in Computers, Volume 29, Number 12, pp. 1104-1113.

Walton, D.N. and Krabbe, E.C.W. 1995. Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning. State University of New York Press.

Winograd, T. and Flores, F. 1986. Understanding Computers and Cognition: A New Foundation for Design. Ablex Publishing.

Yolum, P. and Singh, M.P. 2001. Synthesizing Finite State Machines for Communication Protocols. Technical Report TR-2001-06. North Carolina State University, Department of Computer Science.