Extending a logic based one-to-one negotiation framework to one-to-many negotiation

Paolo Torroni[1] and Francesca Toni[2]

[1] DEIS, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
ptorroni@deis.unibo.it
[2] Department of Computing, Imperial College, 180 Queens Gate, SW7 London, UK
ft@doc.ic.ac.uk

Abstract. [13] presents a logic-based approach to multi-agent negotiation. The advantages of such approach stem from the declarativeness of the model, which allows to formulate and prove some interesting properties (such as termination and convergence of a protocol), to the possibility of identifying and combining varieties of agents, implementing different negotiation policies, and of forecasting the behaviour of a system with no need for simulation. The work introduces a language for negotiation that allows to cater for two agent dialogues, in a one-to-one negotiation setting. Auctions are an example of one-to-many negotiation mechanisms, where agents try to maximize their profit by buying items in competition with other parties, or selling them to crowds of bidders. In this paper, we show how the negotiation framework of [13] can be extended to accommodate a suitable negotiation language and coordination mechanism (in the form of a shared blackboard) to tackle one-to-many negotiation.

1 Introduction

Autonomous Agents and Multi-Agent Systems (MAS) have represented a hot topic of Computer Science disciplines for the past ten and more years. They have often been adopted as a metaphor to model autonomous entities capable of interacting and being part of organizations or societies. The idea of making them intelligent asked for the contribution of disciplines such as Artificial Intelligence (AI) [16]. Logic-based AI has been playing a foreground role in the MAS community since the very beginning, e.g. with the advent of the BDI model for agent beliefs, desires, and intentions [9].

Recently, computational logic has started to make a relevant contribution to the development of MAS [10, 14]. Indeed, computational logic-based formalisms are a powerful way to model and implement the agents' knowledge and the reasoning of the agent, that uses and possibly updates such knowledge. All these ingredients suit well a MAS context, which is normally open, dynamic, and unpredictably evolving. Computational logic can contribute to modeling both individual agents and societies of agents, and to providing operational semantics that can be straightforwardly used as a basis to implement a system.

In [13, 12], Sadri, Toni and Torroni introduced a computational logic-based approach to agent dialogue for negotiation of resources. Negotiation is one of the main research streams in MAS, due both to the wide range of possible applications, spanning electronic markets, task reallocation, and distributed resource management, just to cite some, and to its strong commercial interest. In a logic setting, agents are given a logic-based knowledge representation, including goals, intentions, and beliefs. When an agent is missing a resource, it negotiates with another agent, in a dialogue-based framework, requesting the missing resource and carrying out the dialogue by means of utterances, or *dialogue primitives*. In the course of a dialogue, both agents could possibly modify their own intentions, for instance as a consequence of a 'better' (i.e. less expensive) plan that could arise during the dialogue. Agents decide which primitive to utter based upon a computational logic-based proof procedure, that treats dialogue primitives as 'hypotheses' (or 'abducibles'). Such hypotheses are singled out so that some given 'negotiation policies' are enforced. The policies are represented as 'integrity constraints' that need to be satisfied, as in databases and abductive reasoning.

In realistic applications, negotiation mechanisms need to cater for and exploit the plurality of agents in societies. The agents looking for resources might want to choose among several marketplaces, the agents providing resources might want to choose among different possible buyers, and all agents, in general, might want to play as buyers and sellers, in different contexts, according to their subjective needs and to the objective situation. In this respect, *auctions* can be seen as negotiation patterns where agents do not deal with each other in pairs, but rather compete in crowds for the achievement of their individual goals. In auctions, agents try to maximize their profit by buying items in competition with other parties, or selling them to selected agents in crowds of bidders.

In this paper, we show how to tackle one-to-many negotiation within the one-to-one negotiation framework of [13]. The contribution of the paper is in the introduction of a logic approach to automated auctions, and in particular in the extension of an existing framework and in the proposed implementation of the English auction protocol by means of a program written in the extended language. One of the most innovative aspects of such approach is that it is operational, i.e., the logic that describes the agent knowledge is a program that is implementable to build agent applications, whose properties can be studied and proven. We extend the negotiation language, and we introduce a suitable coordination mechanism (via a shared blackboard). Although this is ongoing work, we believe that such approach is a promising starting point for building up an integrated logic-based negotiation framework, within which we aim to define and prove properties that rule the marketplace, without resorting to simulation.

The paper is organized as follows: in Sections 2 and 3 we briefly discuss the one-to-one negotiation language and framework, respectively, introduced in [13]. In Section 4 we introduce the concept of auctions, referring to some classical auction protocols in the literature. Starting from the individual protocols, we draw a common schema driving our choices in the design of the extended language. In Section 5 and 6 we show how to extend the framework to tackle auctions, and we illustrate the extension by realizing a particular protocol, the English auction. In Section 7 we draw some conclusions.

2 A Language For Negotiation

In this section we will briefly sketch the language for negotiation adopted in [13] and give an example of a two-agent negotiation dialogue. In the next section, we will give a flavor of how such dialogue can be generated within a computational logic setting.

In the following dialogue, inspired by [8], agent a asks agent b for a resource (a $nail$), needed to carry out a task (to hang a picture). Being refused the requested resource, a asks b the reason why, aiming at acquiring additional information that could help a to bargain for the resource or find an alternative resource to carry out the task.

$tell(a, b, request(give(nail)), 1)$
$tell(b, a, refuse(request(give(nail))), 2)$
$tell(a, b, challenge(refuse(request(give(nail)))), 3)$
$tell(b, a, justify(refuse(request(give(nail))), \sim have(nail)\}), 4)$

In general, a dialogue is a sequence of *dialogue moves* or *primitives*, where a primitive is represented in the form of an atom in the predicate $tell$. Such predicate has four arguments, respectively: the sender, the recipient, the subject, and the time of the primitive. Time is understood as a *transaction* time, rather than actual time.

Notice that such primitives are tailored to the needs of a two-agent dialogue setting, where, in particular, the recipient is one specific agent. No support for broadcast or multi-cast is provided. However, the framework presented in [13], as described in the next section, is independent of the concrete negotiation language. In Section 5 we will introduce a language supporting multi-casting (via a blackboard) and allowing one-to-many communication primitives. The language that we will propose for auctions is again a set of primitives expressed in the form of *tell* predicates, with a fundamental difference: the second argument can either be a single agent, or a *group* of agents. We assume that agents can join groups, uniquely identified in the system. In particular, such identification can be either universal, including *all* agents in the system, or can be used to specify the group composed by the subscribers of a specific auction, as we will see in the following. All of this is in line with the current research on agent systems and interactions, e.g., with KQML. In fact, KQML provides support for multi-casting in a very similar way. A difference with KQML is that in our setting we do not need to introduce predicates other than $tell$. In fact, as we will see later, the (past) dialogue is used by the abductive proof-procedure that generates the dialogue itself as a monotonically growing knowledge base (see for instance the definition of predicate *on-sale* in Section 6). In particular, we do not make use of non-monotonic primitives such as the KQML *untell*, for all utterances keep holding in the utterer's knowledge base.

3 A Negotiation Framework

In addition to the negotiation language, the ingredients needed in building a dialogue framework are: a knowledge representation formalism, a proof procedure for reasoning automatically with the knowledge, and a communication layer. We will not get into details in the description of the knowledge representation and proof procedure on which the agents base their reasoning, and we will mostly refer to [13]. In brief, as far as the knowledge representation is concerned, we will assume that agents have a (declarative) representation of goals, beliefs, and intentions, namely plans to achieve goals. As an example, the knowledge of some agent a can be the following $\mathcal{K}_a$:

$\mathcal{B}_a$ domain-specific beliefs, e.g., $is\_agent(b), i\_am(a)$, as well as domain-independent beliefs, e.g., those implementing negotiation protocols and policies;
$\mathcal{R}_a$: $\{$ $have(picture)$, $have(hammer)$, $have(screwdriver,)$ $\}$;
$\mathcal{I}_a$: $\{$ $available(\{hammer, picture\}), plan(\{obtain(nail), hit(nail), hang(picture)\}, 0), goal(\{hung(picture)\}), missing(\{nail\}, 0)$ $\}$;
$\mathcal{D}_a$: $\emptyset$.
$\mathcal{G}_a$: $hung(picture)$;

where $\mathcal{B}_a$ stands for agent a's *beliefs*, $\mathcal{R}_a$ stands for *resources*, $\mathcal{I}_a$ for *intentions*, $\mathcal{D}_a$ for (past) *dialogue*, and $\mathcal{G}_a$ for *goals*.

The beliefs can include knowledge that can be used to generate plans. Here we do not make any assumptions on how plans are generated, whether by a planner or from existing libraries.

The beliefs also include *dialogue constraints* that express how agents should react to dialogue moves of other agents. A very simple example of dialogue constraints is the following:

$$\begin{aligned} & tell(X, a, \mathbf{request}(\mathbf{give}(R)), T) \wedge have(R, T) \\ \Rightarrow \quad & tell(a, X, \mathbf{accept}(\mathbf{request}(\mathbf{give}(R))), T+1) \end{aligned}$$

This constraints reads as follows: 'if agent a receives a request from another agent, X, about a resource R that it has, then a tells X that it will accept the request'[1] A formal specification for *have* and *need* is given in [13].

Sets of dialogue constraints express shared protocols and/or individual policies of agents. Dialogues such as the one illustrated in the previous section can be generated automatically within a computational logic setting, e.g. by means of an abductive proof procedure executed within an agent cycle [5]. In computational logic [4], abduction is a reasoning mechanism that allows to find suitable explanations to given observations or goals, based on an abductive logic program. In general, an abductive logic program is a triple $\langle P, \mathcal{A}, IC \rangle$, where P is a logic program, $\mathcal{A}$ is a set of so-called *abducibles* or *hypotheses*, i.e., *open* atoms which can be used to form explanations, and IC is a set of *integrity constraints*, i.e. sentences that need to be satisfied by all explanations. Given a goal g, abduction aims at finding a set of abducibles that, if used to enlarge P, allow to *entail* g while satisfying IC.

The adoption of automatic proof procedures such as that of [1] or [3], supported by a suitable agent cycle such for instance the *observe-think-act* of [5], implements a concrete concept of entailment with respect to knowledge bases expressed in abductive logic programming terms. The execution of the proof procedure within the agent

[1] In our setting, a saying that it accepts X's request about R is equivalent to saying that a *gives* X R: not being concerned with execution, we assume that once an agent tells that it will give a resource, it will actually do it at some point in the future.

cycle allows to produce hypotheses (explanations) that are consistent with the agent integrity constraint, IC. Constraints play a major role in abduction, since they are used to drive the formulation of hypotheses and prevent the procedure from generating wrong explanations to goals.

In [13], abduction has been used to model agent dialogue, with dialogue constraints being represented as integrity constraints, and the beliefs being represented as abductive logic programs. Dialogue constraints are fired each time the agent is expected to produce a dialogue move, e.g., each time another agent sends a request for a resource. Such move is then produced as an hypothesis that must be assumed true in order to keep the knowledge base satisfying the ICs. The use of abduction in the agent dialogue context, as opposed to other (less formal) approaches, has several advantages, among which the possibility to determine properties of the dialogue itself, and the one-to-one relationship holding between specification and implementation, due to the operational semantics of the adopted abductive proof procedure.

4 Auctions

Auctions could be interpreted as an alternative way to negotiate and retrieve resources. They provide an alternative to one-to-one negotiation, e.g. the negotiation carried out by means of dialogues as introduced in the previous section. Indeed, in a framework that includes auctions as a negotiation mechanism, if an agent x needs a resource, it can either ask it to another agent in the system, or find out an auction where the resource is sold and try to obtain it.

Differently from the general dialogue framework, in auctions the *price* of items plays a major role. In particular, the main resource limit is on the initial budget of the single bidders (otherwise agents would buy *at any price*). The participants of an auction are called *auctioneer*, if they sell resources, and *bidders*, if they compete against one another to buy resources. Reasonably, we will assume that there are at least two bidders (otherwise we simply have one-to-one negotiation). In the following, we will call a the auctioneer, b_1, b_2, . . ., b_n the bidders.

In general, an auction consists of at least three steps, as shown in Figure 1.

Fig. 1. Basic steps of an auction

The *announcement* step is an utterance made by the auctioneer a, where it declares that an item is on sale at a certain price, for a certain timeout. The *bid* phase is an offer made by a bidder, who is willing to buy the item on sale for a certain price. The *award* phase is a statement made by a to notify everybody that the item has gone[2].

[2] In general, items could also be declared unsold. For the sake of simplicity, we will consider this as a sub-case of this last award phase.

There are various kinds of auctions in literature. We will cite here only four of them, that we consider representative, and in particular: the *English* auction, the *Dutch* auction, the *First Price Sealed Bid (FPSB)* auction, and the *Vickrey* auction.

English auction In the case of the English auction, the price of the item that is currently on sale is increased until there is only one bid. The highest bid wins. This behaviour can be represented as in Figure 2, where there can be any number of bids before moving on to the award.

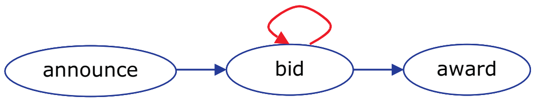

Fig. 2. English auction: price goes up, 1 bid at a time

Dutch auction In the case of the Dutch auction, the price of the item that is currently on sale goes down until there is one bid. The first bid wins. This behaviour can be represented as in Figure 3, where the announcement can be repeated several times before a bid is made.

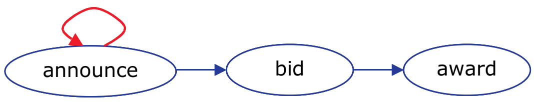

Fig. 3. Dutch auction: price goes down, 1 bid at a time

FPSB auction In the case of the First Price Sealed Bid auction, k bids are simultaneously made by k bidders, $k \leq n$. Bids are sealed, i.e., bidders do not know each other's bids. The highest bid wins. This behaviour can be represented as in Figure 4, where the announcement is made once and k simultaneous bid are made. This protocol is in general faster that the two previous ones, since it is guaranteed to terminate in three steps.

Fig. 4. FPSB auction: k bids are collected, $k < n + 1$, highest wins

Vickrey auction In the case of the Vickrey auction, again, k bids are simultaneously made by k bidders, $k \leq n$. Bids are sealed, i.e., bidders do not know each other's bids. The difference with the FPSB case is that the second highest bid wins. This behaviour is not different from that represented in Figure 4,

It is worth to notice that, by a protocol point of view, Vikrey and FPSB auctions are the same. In fact, the only difference is the final price that the winner pays, which can play a role in terms of strategies, but not in terms of protocols.

Before the auction takes place, the auctioneer must declare *which objects* are about to be sold, *when* the auction is starting, *which kind* of auction it is going to be (English, Dutch, etc.), how long the *timeout* is going to be, and so on. We could call the period of such declarations the *publishing* phase. Then, auctions could be assigned unique *identifiers* (IDs), that will be used by all those who need to refer to the auction.

In general, the auctioneer must know who is participating at the auction. We will therefore assume that bidders *subscribe* to the auction before it starts[3] and form a group. The group is named after the auction ID. After all the objects have been either awarded or declared unsold, the auction is *closed*.

In general, we could represent the four protocols as a particular case of the one represented in Figure 5[4]:

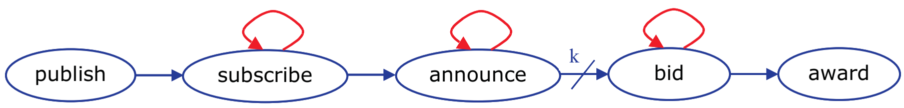

Fig. 5. General steps of an auction

It is clear that it is necessary to have a notion of *timeout*, as the whole auction mechanism is based on it. If we implement auctions by means of dialogue primitives with an associated *transaction* time, we still have to relate it to the *actual* time. To this respect, the blackboard comes in hand, since it can be used to assign primitives both an actual timestamp and a transaction time, and keep them bound somehow. In the sequel, we will assume that agents are equipped with a predicate $actual\_time(Transaction\_time, Actual\_time)$ that keeps track of such relationship.

5 Extended Negotiation Language And Framework

The coordination space If we want to extend this framework to tackle auctions, we need a language, and in addition to it we need a communication layer that provides

[3] More generally, we could assume that bidders can subscribe up until the auction is closed, i.e., up until all the objects have been either awarded or declared unsold.

[4] We omit the closure phase for lack of space. Note that such phase does not play a role, anyway, for the purposes of this paper.

a support for multi-casting. For this we will use a blackboard. Although we will not commit to any particular blackboard or coordination framework, such as for instance a tuple space can be, we will make on the blackboard some assumptions. In particular, we assume that the blackboard is able to assign a transaction time to incoming messages, to store transactions, to manage conflicts in case of multiple concurrently incoming messages (for instance by allowing only one of them), and we will also assume that the blackboard can be programmed in order to allow only those messages that stick to the protocol. For instance, if an English auction is on, where price can only go up, and it must be at least a certain percent higher than that of the previous bid, the blackboard can be programmed to reject those primitives that represent bids that are below the minimum threshold.

We also introduce in the framework a *locking* mechanism, implemented again within the blackboard, that allows bidding only after obtaining the blackboard lock. This is expressed in the agent programs by a logical predicate, $obtain\_lock(\ at(\ Actual\_time\ ),\ Auction\_ID,\ Transaction\_time\ )$. The semantics of such predicate is: true if the agent obtains the right to write on the blackboard (i.e., to multi-cast) at time $Actual\_time$, the $Transaction\_time$-th primitive of $Auction\_ID$. An example of use of such predicate is in Section 6.

Multi-cast primitives As opposed to dialogue primitives, the communication acts of auctions must be multi-casted, which requires a suitable support, as we pointed out previously. Therefore, we will assume that agents do not write directly in a blackboard, not before obtaining from the blackboard the permission to write. In fact, the blackboard will coordinate the agent requests, by allowing one message (primitive) at the time, and by giving it a unique transaction time.

The format of multi-cast primitives must be different from that of one-to-one primitives, since the recipient is in general not unique. As we already said, the second parameter will not be a single agent, but a group of agents. Multi-cast primitives will have in general this format: $tell(Sender,\ Group,\ Subject,\ Time)$, where $Sender$ is an agent, $Group$ is a group of agent, intended to be the recipients of the message, $Subject$ expresses the content of the message, and $Time$ is again a transaction time.

We are now ready to introduce a *language* for auctions, i.e., a set of allowed dialogue primitives that agents use to implement an auction:

- $tell(\ Auctioneer,\ all^5,\ publish(\ auction(\ Auction\_ID\ ),\ items(\ \{\ i_1, i_2, \ldots, i_m\ \}\ ),\ protocol(\ english \mid dutch \mid fpsb \mid vickrey\ ),\ beginning\_at(\ initial\_time\ ),\ timeout(\ 2'\ )\ ),\ Time\ )$
- $tell(\ Bidder,\ Auction\_ID,\ subscribe(\ Auction\_ID\ ),\ Time\ )$
- $tell(\ Auctioneer,\ Auction\_ID,\ announce(\ Item,\ Price,\ Timeout\ ),\ Time\ )$
- $tell(\ Bidder,\ Auction\_ID,\ bid(\ Item,\ Price\ ),\ Time\ )$
- $tell(\ Auctioneer,\ Auction\_ID,\ award(\ Item,\ Price,\ Bidder\ ),\ Time\ )$

[5] Here and in the following, *all* could mean agents all the agents known or reachable by the auctioneer. We assume that the auctioneer has some visibility of the environment; we have not dealt explicitly so far with such issue, though.

– $tell(\ Auctioneer, Auction\_ID, close, Time\ )$

Given this language, an example of an English auction is the following[6]:

$\rightsquigarrow$ $tell(\ a, all, publish(\ auction(\ auction_1\ ), items(\ \{\ nail\ \}\ ), protocol(\ english\ ),$
$beginning\_at(\ \texttt{Jan 3}^{\texttt{th}}, 2002, 14:30\ \texttt{GMT}\ ), timeout(\ 2'\ )\ ), 1\ )$
$\rightsquigarrow$ $tell(\ b_4, a, subscribe(\ auction_1\ ), 2\ )$
$\rightsquigarrow$ ...
$\rightsquigarrow$ $tell(\ a, auction_1, announce(\ nail, 100, 14\ ), 13\ )$
$\rightsquigarrow$ $tell(\ b_4, auction_1, bid(\ nail, 100\ ), 14\ )$
$\rightsquigarrow$ $tell(\ a, auction_1, announce(\ nail, 110, 16\ ), 15\ )$
$\rightsquigarrow$ $tell(\ b_9, auction_1, bid(\ nail, 110\ ), 16\ )$
$\rightsquigarrow$ $tell(\ a, auction_1, announce(\ nail, 120, 18\ ), 17\ )$
$\rightsquigarrow$ $tell(\ a, auction_1, award(\ nail, 110, b_9\ ), 18\ )$ [7]

In this example, the auctioneer starts with publishing the auction $auction_1$ for a certain point in the future ($Jan\ 3^{th}, 2002, 14:30\ GMT$), and collects subscriptions. Then it starts the auction by announcing the first (and only) item, a $nail$. Bidders keep replying to further announcements, until a timeout is reached after the last one. We could give similar examples for the other auctions, but we do not have enough space.

In the case of English and Dutch auctions, as in this example, the auctioneer announces the new (minimum) price and waits for a bid. If the timeout comes the auctioneer posts in the blackboard the award message. We will assume that some aspects, such as filtering unsuitable bids, are managed by the blackboard (bids for items not being sold in that auction, bids from unsubscribed agents, bids for a price that is too low, ... will be rejected by the blackboard). We admit that this is a strong assumption on the environment, and needs to be further investigated. Moreover, such behaviour hardwired in the blackboard, could require a dynamic (re-)configuration. It is not obvious at all who and how should to it. Another important issue, with this respect, is how to merge such non-logical element within the logical framework.

We can assume of course that bidders can bid more than announced by the auctioneer (highest bid wins anyway). This is up to the agent. In the case of FPSB and Vickrey auctions, that we could not represent here, the timeout would be expressed in terms of transaction time, as the current time $+n$, n being the number of expected bidders. This is one more reason why it is required to subscribe an auction: n is actually the number of subscribers. This could raise some problems, in that it would require some assumptions on the presence of all subscribers for the whole duration of the auction, which is maybe too strong in a multi-agent setting. Anyway, it is possible to relax this condition, and define a protocol in which not all subscribers must bid. We could imagine that if there are less bids than bidders ($k < n$), the auctioneer will be in charge to make sure that

[6] See Appendix for more examples

[7] As we said before, we will not deal with the closure of auctions, which could be expressed by a primitive such as $tell(\ a, auction_1, close, 19\ )$, uttered by the auctioneer.

before too late (*actual* timeout) n messages are anyway posted in the blackboard, in order to reach the (*transaction*) timeout. She will therefore put $n - k$ messages atomically together with the award message, just before it is published.

6 An Example: Implementation Of The English Auction Protocol

In this section, we will give here an example of a possible implementation of the English auction protocol. Due to the space constraints, we will only write the basic rules and constraints of the two kinds of agents: the bidder and the auctioneer. Let us start with the auctioneer program.

auctioneer (a) program:

$\mathbf{on\_sale(Item, Auction\_ID)} \leftarrow$
$tell(a, all, publish(auction(Auction\_ID), items(Items),$
$protocol(Protocol), beginning\_at(Initial\_time), timeout(Timeout)), Time)$
$\wedge\ Item \in Items$
$\wedge\ actual\_time(Actual\_time)$
$\wedge\ Initial\_time < Actual\_time$
$\wedge$ *not* $timeout\_expired(Item, Auction\_ID, Timeout)$
$\wedge$ *not* $(tell(a, Auction\_ID, award(Item, Price, Bidder), Time1)$
$\wedge\ Time1 < Time)$

An item is on_sale if it is included in the item list of an ongoing auction, and the timeout after the last bid has not expired.

$\mathbf{timeout\_expired(Item, Auction\_ID, Timeout)} \leftarrow$
$tell(a, Auction\_ID, announce(Item, Price1, Timeout1), Time1)$
$\wedge$ *not* $(tell(a, Auction\_ID, announce(Item, Price2, Timeout2), Time2)$
$\wedge\ Time2 > Time1)$
$\wedge\ actual\_time(Time1, Actual\_time1)$
$\wedge\ actual\_time(now, Actual\_time)$
$\wedge\ Actual\_time > Actual\_time1 + Timeout$

The timeout that refers to a certain item on sale at a certain auction is $expired$ if now is a later time than that of the last announcement, plus the declared $Timeout$.

$\mathbf{new\_price(Price, New\_price)} \leftarrow New\_price\ is\ Price + 0.1 * Price$

The auctioneer announces the new price of an item adding a ten percent to the last price.
$\mathbf{tell(Bidder, Auction\_ID, bid(Item, Price), Time)}$
$\wedge\ on\_sale(Item, Auction\_ID)$
$\wedge\ new\_price(Price, New\_price)$
$\wedge\ obtain\_lock(at(now), Auction\_ID, Time + 1)$

$$\Rightarrow \mathbf{tell(a, Auction\_ID, announce(Item, New\_price, Time+2), Time+1)}$$

This constraints says that if a participant bids at time $Time$, and the item in question is still on sale, the auctioneer next (i.e., at $Time + 1$) announces such item for a new price. Such announcement will hold up to $Time + 2$.

$\mathbf{timeout\_expired(Item, Auction\_ID, Timeout)}$
$\wedge\ tell(A, all, publish(auction(Auction\_ID), \ldots, timeout(Timeout)), \ldots)$
$\wedge\ tell(Bidder, Auction\_ID, bid(Item, Bidder\_price), Time - 1)$
$\wedge\ tell(A, Auction\_ID, announce(Item, New\_price, Time + 1), Time)$
$\wedge\ Bidder\_price \geq New\_price$
$\quad \Rightarrow \mathbf{tell(a, Auction\_ID, award(Item, Bidder\_price, Bidder), Time + 1)}$

This constraints says that after the timeout has expired the highest bid wins, and therefore the bidder is awarded the item.

$\mathbf{timeout\_expired(Item, Auction\_ID, Timeout)}$
$\wedge\ tell(A, all, publish(auction(Auction\_ID), \ldots, timeout(Timeout)), \_)$
$\wedge\ tell(A, Auction\_ID, announce(Item, \_, \_), Time)$
$\wedge\ not\ tell(\_, Auction\_ID, bid(Item, \_), \_)$
$\quad \Rightarrow \mathbf{tell(a, Auction\_ID, award(Item, \ldots, noone), Time + 1)}$

This constraints says that if there are no bids, the item is declared unsold.

bidder (*b*) program:

The bidder program is simpler, but it introduces two predicates whose implementation is not specified here, and will depend on the individual bidder programs (we assume that each bidder may adopt a different policy from the others). Such predicates are $calculate\_new\_price$ and $calculate\_bid\_time$.

$\mathbf{tell(a, Auction\_ID, announce(Item, Price, Timeout), Time)}$
$\wedge\ have\_subscribed(Auction\_ID)$
$\wedge\ not\ (tell(Participant, Auction\_ID, Anything, Time1) \wedge\ Time1 > Time)$
$\wedge\ calculate\_new\_price(Item, Price, New\_price)$
$\wedge\ calculate\_bid\_time(Auction\_ID, Bid\_time)$
$\wedge\ obtain\_lock(at(Bid\_time), Auction\_ID, Time + 1)$
$\quad \Rightarrow \mathbf{tell(b, Auction\_ID, bid(Item, New\_price), Time + 1)}$

This means that if the auctioneer has announced an $Item$ at a $Price$, at the (transaction) time $Time$, and after that no other participant made any move, if the bidder b can calculate a suitable price and can obtain the blackboard to bid at a certain time (Bid_time), then b will bid at (transaction time) $Time + 1$. The two above mentioned predicates and the blackboard will decide whether the agent will actually bid or not.

$$\mathbf{have\_subscribed(Auction\_ID)} \leftarrow$$
$$tell(b, a, subscribe(Auction\_ID), Time)$$

7 Discussion And Future Work

We have extended an existing framework for one-to-one agent negotiation to cope with one-to-many negotiation, in the particular case of auctions. There is much ongoing work on the negotiation area: those interested could refer to [11] for an introduction, and to [6] and [2] for a perspective on state of the art methods and challenges.

An other existing approach to negotiation via dialogue, that makes use of an argumentation system, is that of [15], and more recently [7], where the authors adopt a modal logic approach and focus on the mental states of agents that move towards an agreement, and on the way to persuade a counterpart in order to foster cooperation.

As far as auctions, there is much work on mathematical models, and protocol comparison in terms of efficiency, stability, etc., but to the best of our knowledge, only a little relate theoretically founded models to operational models, as in our case.

This is a preliminary work on the subject and needs to be further expanded in several ways. The main issue, in our opinion, is how to tackle the various non-logical elements, such as the initial time, the timeout, the blackboard lock, and merge them in the logical framework that we propose. This is a need because auctions are intrinsically non-logical in relying on the concept of timeout. As opposed to the case of dialogue primitives, in an auction setting the trigger that fires a dialogue constraint and makes a primitive being cast will necessarily depend not only on the existence of other incoming messages, but also on the *lack* of such messages. That is, on the fact that a timeout is met.

However, in our framework, we manage to separate such non-logical features from the logical reasoning bit, partly relegating them to the blackboard, partly implementing them inside logical predicates used by the agent.

As future work, we would like to integrate within this framework the two negotiation patterns, giving the agents the possibility to either retrieve the missing resources by means of dialogues, or by means of auctions. We aim at defining a comprehensive architecture in which an agent can choose to obtain a resource either through an auction or through a 'simple' sequence of dialogues.

Another direction of research is that of *policies*. We consider it important to provide the possibility to define policies that are orthogonal to the agent program used to multicast primitives, and to the constraints in particular. For instance, before committing to a bid, agents could reason in terms of available budget, need for other items, maximum price that they intend to pay for the item, importance of the goal, attitude they may have with respect to the other bidders, etc. Together with policies, *strategies* can be used to maximize an agent's payoff, e.g. agents can decide to bid as soon as possible, or as late as possible, with the adopted policy.

We intend to study properties of auctions, when different policies and strategies meet. Finally, we plan to give an implementation of this one-to-many framework, possibly in integration with the other dialogue-based negotiation framework.

Acknowledgements

We would like to thank Paola Mello, Maurelio Boari, and the anonymous referees, for their precious comments, suggestions, and support.

References

1. T. H. Fung and R. A. Kowalski. The iff proof procedure for abductive logic programming. *Journal of Logic Programming*, 1997.
2. N. R. Jennings, P. Faratin, A.R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation: Prospects, methods and challenges. *International Journal of Group Decision and Negotiation*, 10(2), 2001.
3. A. Kakas and P. Mancarella. On the relation between truth maintenance and abduction. In T. Fukumura, editor, *Proceedings of the first Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan*, pages 438–443, 1990.
4. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in AI and Logic Programming*, 5:235–324, 1998.
5. R. A. Kowalski and F. Sadri. From logic programming to multi-agent systems. *Annals of Mathematics and AI*, 1999.
6. S. Kraus. *Strategic Negotiation in Multi-Agent Environments*. MIT Press, Cambridge, MA, 2000.
7. S. Kraus, K. Sycara, and A. Evenchik. Reaching agreements through argumentation; a logical model and implementation. *Artificial Intelligence*, 104:1–69, 1998.
8. S. Parsons, C. Sierra, and N. R. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.
9. A. Rao and M. Georgeff. An abstract architecture for rational agents. In *Proceedings of the International Workshop on Knowledge Representation (KR'92)*, 1992.
10. S. Rochefort, F. Sadri, and F. Toni, editors. *Proc. International Workshop on Multi-Agent Systems in Logic Programming, in conjunction with ICLP'99, Las Cruces, New Mexico*. November 1999.
11. J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. MIT Press, Cambridge, Massachusetts, 1994.
12. F. Sadri, F. Toni, and P. Torroni. Dialogues for negotiation: agent varieties and dialogue sequences. In *Proceedings ATAL'01, best paper award, Seattle, WA*, August 2001.
13. F. Sadri, F. Toni, and P. Torroni. Logic agents, dialogues and negotiation: an abductive approach. In *Proceedings AISB'01 Convention, York, UK*, March 2001.
14. K. Satoh and F. Sadri, editors. *Proc. Workshop on Computational Logic in Multi-Agent Systems (CLIMA-00), in conjunction with CL-2000, London, UK*. July 2000.
15. K. P. Sycara. Argumentation: Planning other agents' plans. In *Proceedings 11th International Joint-Conference on Artificial Intelligence*, pages 517–523. Morgan Kaufman, 1989.
16. G. Weiss. *Multiagent Systems*. MIT Press, 1999.

Appendix: More Examples Of Automated Auction: Dutch, Fpsb, And Vickrey Auctions

An example of a Dutch auction is the following:

$\rightsquigarrow$ $tell(\ a, all, publish(\ auction(\ auction_2\ ), items(\ \{\ nail\ \}\ ), protocol(\ dutch\ ),$
$beginning\_at(\ \texttt{Jan } 3^{\texttt{th}}, 2002, 14:30\ \texttt{GMT}\ ), timeout(\ 2'\ )\ ), 1\ )$
$\rightsquigarrow$ $tell(\ b_4, a, subscribe(\ auction_2\ ), 2\ )$
$\rightsquigarrow$ $\ldots$
$\rightsquigarrow$ $tell(\ a, auction_2, announce(\ nail, 100, 3\ ), 12\ )$
$\rightsquigarrow$ $tell(\ a, auction_2, announce(\ nail, 90, 4\ ), 13\ )$
$\rightsquigarrow$ $tell(\ a, auction_2, announce(\ nail, 80, 5\ ), 14\ )$
$\rightsquigarrow$ $tell(\ a, auction_2, announce(\ nail, 75, 6\ ), 15\ )$
$\rightsquigarrow$ $tell(\ b_3, auction_2, bid(\ nail, 75\ ), 16\ )$
$\rightsquigarrow$ $tell(\ a, auction_2, award(\ nail, 75, b_3\ ), 17\ )$

An example of a FPSB auction is the following (the price is encrypted; example with 5 bidders):

$\rightsquigarrow$ $tell(\ a, all, publish(\ auction(\ auction_3\ ), items(\ \{\ nail\ \}\ ), protocol(\ fpsb\ ),$
$beginning\_at(\ \texttt{Jan } 3^{\texttt{th}}, 2002, 14:30\ \texttt{GMT}\ ), timeout(\ 2'\ )\ ), 1\ )$
$\rightsquigarrow$ $tell(\ b_4, a, subscribe(\ auction_3\ ), 2\ )$
$\rightsquigarrow$ $\ldots$
$\rightsquigarrow$ $tell(\ a, auction_3, announce(\ nail, 100, 12\ ), 7\ )$
$\rightsquigarrow$ $tell(\ b_1, auction_3, bid(\ nail, \$105\$\ ), 8\ )$
$\rightsquigarrow$ $tell(\ b_5, auction_3, bid(\ nail, \$103\$\ ), 9\ )$
$\rightsquigarrow$ $tell(\ b_3, auction_3, bid(\ nail, \$104\$\ ), 10\ )$
$\rightsquigarrow$ $tell(\ b_2, auction_3, bid(\ nail, \$106\$\ ), 11\ )$
$\rightsquigarrow$ $tell(\ b_4, auction_3, bid(\ nail, \$101\$\ ), 12\ )$
$\rightsquigarrow$ $tell(\ a, auction_3, award(\ nail, \$106\$, b_2\ ), 13\ )$

An example of a Vickrey auction is the following (same bids as before):

$\rightsquigarrow$ $tell(\ a, all, publish(\ auction(\ auction_4\ ), items(\ \{\ nail\ \}\ ), protocol(\ vickrey\ ),$
$beginning\_at(\ \texttt{Jan } 3^{\texttt{th}}, 2002, 14:30\ \texttt{GMT}\ ), timeout(\ 2'\ )\ ), 1\ )$
$\rightsquigarrow$ $tell(\ b_4, a, subscribe(\ auction_3\ ), 2\ )$
$\rightsquigarrow$ $\ldots$
$\rightsquigarrow$ $tell(\ a, auction_4, announce(\ nail, 100, 12\ ), 7\ )$
$\rightsquigarrow$ $tell(\ b_1, auction_4, bid(\ nail, \$105\$\ ), 8\ )$
$\rightsquigarrow$ $\ldots$
$\rightsquigarrow$ $tell(\ b_2, auction_4, bid(\ nail, \$106\$\ ), 11\ )$
$\rightsquigarrow$ $\ldots$
$\rightsquigarrow$ $tell(\ a, auction_4, award(\ nail, \$105\$, b_1 3\ ), 8\ )$