Simulation and Verification of a Dynamic Online Auction

Bo Chen and Samira Sadaoui
Department of Computer Science, University of Regina
Regina SK S4S 0A2, Canada
{chen112b, sadaouis}@cs.uregina.ca

ABSTRACT
This paper deals with the formal specification, simulation and model-checking verification of an agent-based online auction. The need to understand dynamic behavior in auctions is increasing with the popularization of Internet auctions. This provides a strong motivation for the simulation of these complex systems. Since the auction interaction protocol is not trivial, it is suitable to use formal methods to ensure its correct functioning. Therefore, we investigate on the applicability of well-established techniques and tools from distributed systems, such as the formal specification language LOTOS, to specify and verify properties of interaction protocols in multiagent systems, and simulate their behavior before the implementation.

KEY WORDS
Formal methods, LOTOS, software agents, online auctions.

1 Introduction

This paper deals with the formal specification, simulation and model-checking verification of an agent-based online auction. The need to understand dynamic behavior in auctions is increasing with the popularization of Internet auctions [15, 13]. This provides a strong motivation for the simulation of these complex systems. Multiagent technology is considered as a viable solution for large-scale industrial and commercial applications. Indeed, they are suitable for modeling business process and commerce activities such as online auction systems where sell and buy agents can negotiate on the behalf of the customers.

There is still an increasing need for rigorous modeling techniques that permit the complexity of agent systems to be effectively managed, and principled methodologies to guide the design process [11, 3, 14]. In fact, a multiagent system needs well defined mechanisms of management, communication (message exchange) and synchronization of agents staying on different platforms. Consequently, the interaction protocols and lifecycle of agents have been introduced. For instance, there are several communication protocols defined by FIPA (Foundation for Intelligent Physical Agents) for different applications, such as Request Interaction Protocol and English Auction Interaction Protocol [6]. Considering the agent protocols are not trivial, it is suitable to ensure their correctness using formal methods which can specify and verify the properties of concurrent systems, and simulate their behavior before the implementation [5].

The formal description technique LOTOS [1] is ideally suited to the specification of multiagent systems. LOTOS brings many potential advantages: a high level of abstraction, structuring capabilities, requirements capture with several specification styles, specification simulation, verification and refinements. There are many supporting tools for LOTOS, and in this paper, we will focus on the CADP environment [8]. CADP promotes the use of LOTOS to model interaction protocols in multiagent systems. It is a set of tools that assists the user through the design process: compilation, interactive and guided simulation, test generation, and most important, it can perform efficiently verification by equivalence and temporal logic model checking. CADP allows rapid prototyping by generating the C code which can be embedded in real applications.

The rest of this paper is organized as follows: Section 2 presents the informal description of an agent-based online auction protocol inspired from [13, 6]. In Section 3, we produce a LOTOS specification which provides a non-ambiguous description of the interaction protocol. Section 4 reports about the protocol validation, including the interactive and goal-oriented simulation, and also generation of test suites for the protocol implementation. In Section 5, we verify the correctness (based on safety and liveness properties) of the interaction protocol. Section 7 concludes the paper with some perspectives.

2 Agent-Oriented Auction Protocol

We are interested here in the English auction protocol [15, 13, 6] where the bidding is open, the duration is fixed, the bidding price increases gradually, only one or no winner is chosen at the end of the auction. We consider here a single auction with the sale of one item by one seller to n buyers who submit their bids to the auctioneer. A buyer can send more than one bid, being in general influenced by the bids of others. The auction process consists of several steps given below.

RequestOrder. A sell agent presents to the administrator agent a sell order including the description of the item, starting price, reserve

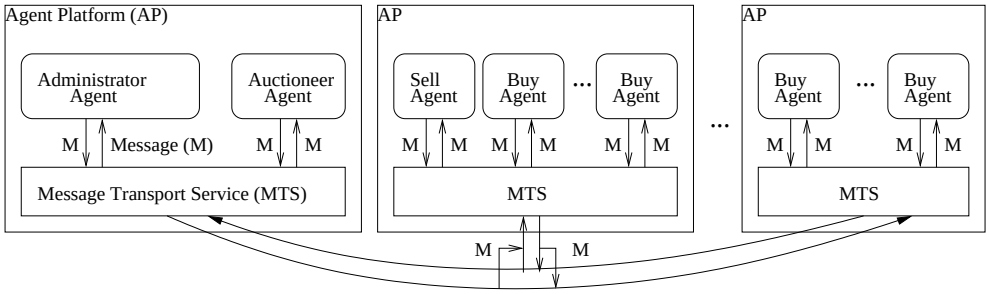

Figure 1. Architecture of the auction interaction protocol.

price, bid increment, payment method, shipping method, contact information, etc.

AcceptOrder/RejectOrder. The administrator agent may accept or reject the order.

RequestAuction. If the order is accepted, the administrator agent creates an auctioneer agent who is given all the auction rules.

InformAuction. The administrator agent looks for the registered buy agents and sends to them a description of the auction, including the auctioneer's ID who will be responsible for the execution of the auction.

Subscribe. If a buy agent is interested in this particular auction, he/she has to subscribe to the auctioneer agent.

RefuseSubscription. The auctioneer can refuse to let a buy agent participate in the auction.

CallForProposal. Once the auction starts (specified by the start time), the auctioneer asks the participants to place bids by issuing a CallForProposal message. A participant who receives a CallForProposal means that his/her subscription has been accepted.

CancelSubscription. A buy agent can inform the auctioneer to be removed from the auction.

AcceptCancel/RejectCancel. A buy agent with the best bid is not allowed to cancel his/her subscription.

Propose. A buy agent can send a proposal message to the auctioneer, including the information "I propose that the bidding level be raised to the purchase price Z and I assert that I am able to pay Z for the item".

CheckProposal. After receiving a proposal, the auctioneer processes the bid by checking for its validity, e.g. checking whether the increment bid has been respected or not.

AcceptProposal/RejectProposal. If the proposal is valid, the auctioneer notifies all participants with information about the current best bid and the agent who holds it, and asks for another bid by broadcasting a CallForProposal message. Otherwise, the auctioneer replies a RejectProposal to an invalid bid and gives the reason why the proposal is rejected.

InformResult. After the auction ends, the auctioneer notifies all the participants and the administrator agent about the auction result. Then, the sell and buy agents will complete the auction transaction through payment server. It the transaction is successful, the notification about the completion of the auction is sent to all participants.

The architecture of the auction interaction protocol is illustrated in figure 1. The basic software elements are agents: administrator, auctioneer, seller and *n* buyers. Agents communicate by sending or receiving messages on unidirectional communication channels. The agent communication is asynchronous point-to-point message passing. The buy agents execute concurrently in order to place bids. Agents behave to an event-reaction scheme: when receiving an event, an agent executes the appropriate reaction which sends messages to other agents.

The Message-Transport-Service (MTS) supports the inter-agent communication based on the standard agent communication language [12], such as FIPA ACL [7]. This language is expected to be universal, e.g. being platform and agent independent. MTS is a service provided by the Agent Platform (AP) to which an agent is attached. It supports the transportation of messages between agents on a given AP and between agents on different APs. A message contains three parts as **a sender, a set of receivers** and **a content**. The sender and receivers refer to agent names which are unique and unchangeable for each agent. MTS handles the transferring, addressing, buffering of messages, and also error messages. Therefore, agents are free of carrying out these tasks.

3 Auction Protocol Specification

LOTOS is the ISO standardized formal specification technique [1] for describing communication protocols and distributed systems. LOTOS combines a process calculus (as defined for CCS and CSP) with an abstract data type language [2]. The process part, describing process composition, defines the external visible behavior of a system. A concurrent system is described as parallel processes interacting by rendezvous. Processes manipulate data values and exchange them at interaction points called gates. Each process behavior is specified with operators defined in table 1.

We present here the formal specification of the English online auction protocol. The architecture of the auction specification is shown in figure 2. The process MTS stores messages into an agent's message buffer. The buffer allows removing deadlocks in the agent communication. Therefore, the agent can read the messages from the buffer at any time. Also, an agent can send a message to another agent which can be in any state (activated or not). For our specification, the content of a message has some of the following six parts: the request type, agent ID, auction item,

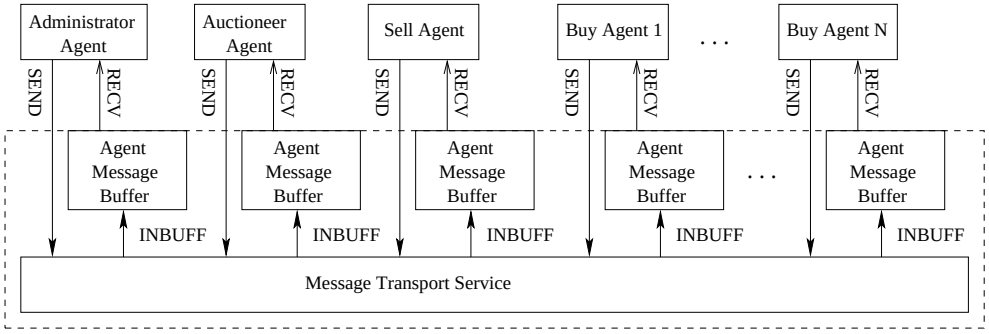

Figure 2. Architecture of the auction specification

Table 1. Process operators

Syntax	Operators
stop	Inaction
G !V ?X:S ; B	Interaction on gate G, sending value V and receiving a value of sort S in variable X, then execute B
B1 [ ] B2	Choice between B1 and B2
$[E] \rightarrow$ B	If E is true then execute B else stop
B1 |[G1 ... Gn]| B2	Synchronization on gates G1, ..., Gn
B1 ||| B2	Interleaving
exit	Successful termination
B1 >> B2	Sequential composition
P[G1...Gn](V1....Vn)	Process call with gate parameters G1...Gn and value parameters V1...Vn

starting price, reserve price and bid increment.

The specification uses ten data types to describe the messages and the agent communication: agent-oriented such as Agent, AgentSet; session-oriented such as AuctionItem, AdministratorSession, SellSession, BuySession, AuctioneerSession; message-oriented such as Message, MessageContent, MessageBuffer. All agents, modeled as LOTOS processes, are activated in parallel. The data part and behavior part are defined separately. In the appendix, the architecture of the protocol as well as the buyagent process are specified in LOTOS.

The process composition is based on the resource-oriented style [4] whereas the individual process uses the state-oriented style [4]. The resource-oriented style supports modularity, parallel composition and gate hiding. In our specification, we have several distinct resources: Administrator, auctioneer, sell and buy agents. Each resource is naturally a self-contained entity which can be implemented by different authorities. The state-oriented style provides an insight of how many states (situations) each agent may have, and what can an agent do in each state, e.g. what kinds of messages are sent and how to deal with each received message. It is worth to note that the auction specification supports the subscription and unsubscription of a buyer at any time during the auction. However, the current winner is not allowed to cancel his/her subscription; a call for proposal is only sent to the subscribed buyers; after the auction ends, the auctioneer will refuse all the late bids (that may be due to the network failure) and which are answered with a RejectProposal.

It is important to notice that the dynamic creation of a process (recursive call occurring at any side of the parallel operator $|[]|$) is supported by the language LOTOS but not by the CADP compiler. This why in our LOTOS specification, the auctioneer is not created by the administrator, instead it is present from the beginning of the auction.

4 Auction Protocol Validation

Simulation allows checking the conformance of the specification to the initial requirements at early development stages. The interactive simulation of LOTOS specifications allows tracing and monitoring all the possible execution sequences, and detecting errors. We give below an example of simulation that leads to an auction failure:

- `SEND !MSG (SELLER, ADD (ADMINISTRATOR, NIL), CNT (REQUESTORDER, ITEM1, 1, 1, 2))`

  `(*Seller sends a RequestOrder to administrator to sell item1, starting price is 1, bid increment is 1, and reserved price is 2*)`
- `RECV !ADMINISTRATOR !SELLER !CNT (REQUESTORDER, ITEM1, SELLER, 1, 1, 2)`

  `(*Administrator receives the sell order*)`
- `SEND !MSG (ADMINISTRATOR, ADD (SELLER, NIL), CNT (ACCEPTORDER, ITEM1))`

  `(*Administrator sends to seller an acknowledge to accept the order*)`
- `SEND !MSG (ADMINISTRATOR, ADD (AUCTIONEER, NIL), CNT (REQUESTAUCTION, ITEM1, SELLER, 1, 1, 2))`

  `(*Administrator submits the order to auctioneer for processing*)`
- `RECV !AUCTIONEER !ADMINISTRATOR !CNT (REQUESTAUCTION, ITEM1, SELLER, 1, 1, 2)`

  `(*Auctioneer receives the auction request*)`
- `SEND !MSG (AUCTIONEER, ADD (ADMINISTRATOR, NIL), CNT (ACCEPTAUCTION, ITEM1))`

  `(*Auctioneer sends an acknowledge to administrator*)`
- **i**

(*Auction starts, internal action*)

- **i**

  (*Auction ends and no buyer has joined this auction, internal action*)

- SEND !MSG (AUCTIONEER, ADD (ADMINISTRATOR, ADD (SELLER, NIL)), CNT (AUCTIONFAILURE, ITEM1, 0, NULL))

  (*Auctioneer informs administrator and seller about the auction failure*)

The tool EXIBITOR, integrated in CADP, allows generating one or all possible scenarios that satisfy a user-defined goal. For instance, if the goal concerns the "auction failure" which is expressed as <until> [SEND !MSG.*.AUCTIONFAILURE.*], EXIBITOR will produce all the execution sequences that lead to an auction failure, e.g. the auction ends and the best bid is lower than the reserve price, the sell order is refused by the administrator agent, or the auction ends and no buyer has subscribed.

The tool TGV, also integrated in CADP, allows deriving test suites from a user-defined test purpose. These tests are used to assess the conformance of a final implementation with respect to its formal specification. For instance, if we consider the following test purpose SEND !MSG.*.AUCTIONFAILURE.*, the tool will then produce all the test cases that lead to an auction failure. We notice that the maximum bids is limited in order to cope with the state explosion problem.

5 Auction Protocol Verification

With the simulation, we can not prove that the system is error free. It is then necessary to use more powerful techniques of automated analysis such as the verification. Indeed, the agents and their communication should satisfy some important correctness properties, such as the safety properties (something bad never happens) and liveness properties (something good eventually happens). Some properties are local to a single agent, and some are global to the whole protocol i.e. the agent composition. The correctness properties can be described using a temporal logic.

In LOTOS, a specification is compiled and translated into a finite labeled transition system (LTS) which encodes all its possible execution sequences. CADP provides model checking tools to automatically verify the LTS against temporal logic formulas which express the possible action sequences a LTS can perform.

Some typical safety properties are called invariants and express that every state of the LTS satisfies some "good" property such as deadlock and livelock free. For the auction system under study, we have identified seven correctness properties listed in the table below. These properties are expressed in regular free-u calculus, the temporal logic accepted by the model-checker EVALUATOR.

Safety properties:

- The absence of deadlock, i.e. that every state has at least one successor. This property is expressed as [true*]<true> true
- The absence of livelock, i.e. useless non-progressive internal cycles. This property is expressed as <true*>@("i")
- A proposal can not be submitted by a buyer before his/her subscription. This property is expressed as before(PROPOSE, SUBSCRIBE) where the macro command before denotes the precedence of actions
- The auctioneer can not accept a subscription after the auction ends. This property is expressed as <true*.AUCTIONFAILURE or AUCTIONSUCESS. true*. ACCEPTSUBSCRIBE> true.
- An unsubscribed buyer can not receive a CallForProposal. This property is expressed as not(<true*. ACCEPTCANCELSUB_BUYER. true*. CALLFORPROPOSAL_BUYER> true)

Liveness properties:

- There exists an execution sequence that leads to an auction success. This property is expressed as <true*. AUCTIONSUCCESS> true
- After a subscription, a buyer can receive a CallForProposal. This property is expressed as <true*. SUBSCRIBE_BUYER. true*. CALLFORPROPOSAL_BUYER> true

With the tool EVALUATOR, we can also verify if a temporal ordering of actions is correct or not. For example, the following sequence of actions is correct:

```
before (ACCEPTORDER, ACCEPTAUCTION)
and before (ACCEPTAUCTION, ACCEPTSUB)
and before (ACCEPTSUB, AUCTIONSUCCESS)
and before (ACCEPTPROPOSAL, AUCTIONSUCCESS)
and before (CALLFORPROPOSAL, AUCTIONSUCCESS)
and before (PROPOSE, AUCTIONSUCCESS)
```

6 Conclusion and Perspectives

In this paper, we have shown that we can make use of existing techniques and tools from distributed systems (such as LOTOS and CADP tool set) to design interaction protocols in multiagent systems. Moreover, LOTOS is suitable to describe synchronous as well as asynchronous systems [9]. With formal specifications, the semantics of a system is describe precisely without any concern for implementation

details, providing a basis for verification (model checking and temporal logic) and validation (through simulation and test generation) of the functionality of the system.

In future work we plan to use the specification language E-LOTOS [10] in order to take into account the time i.e. duration of the auction. Now, we are working on the production of a generic reusable LOTOS specification that describes any interaction protocol in multiagent systems. This specification can be specialized to derive, for instance, the specification of any auction protocol such as Dutch, Vickrey or English.

References

[1] *ISO LOTOS - A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour*. International Organization for Standardization- Information Processing Systems Open Systems Interconnection, Geneve, July 1987.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *in P.H.J. van Eijkand, C.A. Vissers and M. Diaz, eds., The Formal Description Technique LOTOS (North-Holland, Amsterdam)*, pages 303–326, 1989.

[3] F. Brazier, D. Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modeling Multi-Agent Systems in a compositional Formal Framework. In *International Journal of Cooperative Information Systems, vol. 6, n0 1*, 1997.

[4] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specifications, Their Implementations and Their Tests. *Protocol Specification, Testing and Verification, VI, IFIP*, 1987.

[5] M. A. Cornejo, H. Garavel, and R. Mateescu. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In *Proceedings of the 32nd conference on Winter Simulation*, pages 1772–1777. Society for Computer Simulation International, 2000.

[6] FIPA. English Auction Interaction Protocol Specification. In *Foundation for Intelligent Physical Agents, htpp://www.fipa.org/specs/fipa00031*, 2001.

[7] FIPA. ACL Message Structure Specification. In *Foundation for Intelligent Physical Agents, htpp://www.fipa.org/specs/fipa00061*, 2002.

[8] H. Garavel. An Overview of the EUCALYPTUS Toolbox. In *Proc. of COST247, International workshop and Applied Formal Methods in System Design, University of Maribor, Slovenia, June*, 1996.

[9] J. He and K. J. Turner. Verifying and testing asynchronous circuits using lotos. In T. Bolognesi and D. Latella, editors, *Proc. Formal Methods for Distributed System Development (FORTE XIII/PSTV XX)*, pages 267–283, London, 2000. Kluwer Academic Publishers.

[10] I. JTC1/SC21/WG7. *Final Commite Draft on Enhancements to LOTOS*, May 1998.

[11] D. Kinny and M. Georgeff. Modelling and Design of Multi-Agent Systems. *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages ATAL-96, 1193*, 1996.

[12] Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: The Current Landscape. *IEEE Intelligent Systems*, 14(2), April 1999.

[13] H. Mizuta and K. Steiglitz. Agent-Based Simulation of Dynamic Online Auctions. In *Proceedings of 32nd Conference on Winter Simulation*, pages 1772–1777. Society for Computer Simulation International, 2000.

[14] N. Szirbik and G. Wagner. Steps Towards Formal Verification of Agent-based E-Business Applications. In *MOCA 01, Workshop on Modelling of Objects, Components, and Agents, August*, 2001.

[15] M. Vetter and S. Pitsch. An Agent-Based Market Supporting Multiple Auction Protocols. In *Workshop on Agents for Electronic Commerce and Managing the Internet-Enabled Supply Chain, Third International Conference on AUTONOMOUS AGENTS, Seattle, Washington, May 1-5*, 1999.

Appendix

Architecture Specification

```
hide BuffIn in (
((SellAgent[SEND,RECV](seller,sellerSession(initial of
SelSessSt,administrator,null of Item,0 of Money,0 of
Money,0 of Money))
|[RECV]|
AgentMessageBuffer[BuffIN,RECV](seller,<>))
|||  (
BuyAgent[SEND, RECV](buyer1,buyerSession(initial of
BuySessSt,null of Agent,null of Item,0 of Money,0 of
Money,0 of Money,null of Agent,0 of Money,1 of Money))
|[RECV]|)
||| (
BuyAgent[SEND,RECV](buyer2,buyerSession(initial of
BuySessSt,null of Agent,null of Item,0 of Money,0
of Money,0 of Money,null of Agent,0 of Money,2 of Money))
|[RECV]|
AgentMessageBuffer [BuffIN, RECV](buyer2,<>))
|||  (
AdministratorAgent[SEND, RECV](administrator,admSession(
initial of AdmSessST,add(buyer1,add(buyer2,nil)),
auctioneer,null of Item,null of Agent,0 of Money,
0 of Money,0 of Money))
|[RECV]|
```

```
AgentMessageBuffer[BuffIN, RECV](administrator,<>))
|||  (
AuctioneerAgent[SEND,RECV](auctioneer,auctioneerSession(
initial of AucSessSt,administrator,nil,null of Item,
null of Agent,
0 of Money,0 of Money,0 of Money,0 of Money,
null of Agent))
|[RECV]|
AgentMessageBuffer[BuffIN,RECV](auctioneer, <>)))
|[SEND, BuffIN]|
MessageTransportation[SEND,BuffIN](null of Agent,nil,
null of MsgContent))
```

BuyAgent Specification

```
process BuyAgent[SEND,RECV](id:Agent,as:BuySess):exit:=
Let st:BuySessST=getBuySt(as),ar:Agent=getAuctioneer(as),
    it:Item=getItem(as),sp:Money=getSPrice(as),
    bi:Money=getBIncr(as),bp:Money=getBPrice(as),
    bd:Agent=getBidder(as), mb:Money=getMyBid(as),
    max:Money=getMax(as)
in (
[st eq initial] -> exit
[]
[(st eq bidReady) and (id ne bd)] ->
 (
  choice newbid:Money
  []
  [(newbid ge bp) and (newbid le max)] ->
    SEND!msg(id, dd(ar, nil),cnt(propose,it,newbid,id));
    BuyAgent[SEND,RECV](id,buyerSession(bidProposed,ar,it,
    sp,bi,bp,bd,newbid,max))
 )
[]
[((st eq bidReady)and(id ne bd))or(st eq subAccepted)]->
     SEND!msg(id,add(ar,nil),cnt(CancelSub,it));
     BuyAgent[SEND,RECV](id,buyerSession(
     cancelled, ar, it, sp, bi, bp, bd, mb, max))
[]
[(st eq informReceived)] -> (
  SEND!msg(id,add(ar,nil),cnt(subscribe,it,id));
  BuyAgent[SEND,RECV](id, buyerSession(subscribed,ar,it,
  sp,bi,bp, bd, mb, max))
[]
BuyAgent[SEND,RECV](id,buyerSession(initial of BuySessSt,
ar,it,sp,bi,bp,bd,mb,max)))
[]
RECV!id?sender: Agent?c:MsgContent; (
let ac:MsgType=getMsgType(c) in (
[ac eq auctionDone] -> (
BuyAgent[SEND,RECV](id,buyerSession(initial of BuySessSt,
ar,it,sp,bi,bp,bd,mb,max)))
[]
[ac eq callProposal] -> (
[(st ne initial) and (st ne cancelled)] -> (
let sbp: Money=getBPrice(c),sbd:Agent=getBidder(c) in (
 [sbd eq id] ->  BuyAgent[SEND, RECV](id, as)
 []
 [sbd ne id] -> (
  BuyAgent[SEND, RECV](id, buyerSession(bidReady,ar,
  it, sp, bi, sbp, sbd, mb, max)))))
[]
[(st eq initial) or (st eq cancelled)] ->
BuyAgent[SEND, RECV](id, as))
(* end of callForProposal *)
[]
[ac eq rejectSub] -> (
BuyAgent[SEND,RECV](id,buyerSession(initial of BuySessSt,
ar, it, sp, bi, bp, bd, mb, max)))
[]
[ac eq acceptSub] -> (
BuyAgent[SEND, RECV](id, buyerSession(subAccepted,ar,
it, sp, bi, bp, bd, mb, max)))
[]
[ac eq acceptCancelSub] ->
BuyAgent[SEND, RECV](id, buyerSession(initial of BuySessSt,
ar, it, sp, bi, bp, bd, mb, max))
[]
[ac eq rejectCancelSub] ->
(BuyAgent[SEND, RECV](id, buyerSession(initial of BuySessSt,
ar,it, sp, bi, bp, bd, mb, max)))
[]
[ac eq rejectProposal] -> (
BuyAgent[SEND, RECV](id, as))
 []
 [ac eq acceptProposal] -> (
 BuyAgent[SEND, RECV](id, buyerSession(bidReady,
 ar, it, sp, bi, bp, id, mb, max)))
[]
[(ac eq auctionFailure) or (ac eq auctionSuccess)] ->
BuyAgent[SEND, RECV](id, buyerSession(initial of BuySessSt,
ar, it, sp, bi, bp, bd, mb, max))
[]
[ac eq informAuction] -> (
[st eq initial] ->(
let sar:Agent=getAuctioneer(c), sit:Item=getItem(c),
ssp:Money=getSPrice(c),sbi:Money=getBIncr(c) in
(BuyAgent[SEND,RECV](id,buyerSession(informReceived, sar,
 sit, ssp, sbi, 0, null, 0, max)))
[]
[st ne initial] -> BuyAgent[SEND, RECV](id, as)))
(* end of message processing *)
) (* end of let in *)
) (* end of hide *)
endproc
```