Multi-threaded communicating agents in Qu-Prolog

K.L. Clark[1], P. J. Robinson[2], and S. Zappacosta Amboldi[1]

[1] Dept. of Computing, Imperial College, London
[2] School of ITEE, The University of Queensland, Brisbane

Abstract. In this tutorial paper we summarise the key features of the multi-threaded Qu-Prolog language for implementing multi-threaded communicating agent applications. Internal threads of an agent communicate using the shared dynamic database used as a generalisation of Linda tuple store. Threads in different agents, perhaps on different hosts, communicate using either a thread-to-thread store and forward communication system, or by a publish and subscribe mechanism in which messages are routed to their destinations based on content test subscriptions.
We illustrate the features using an auction house application. This is fully distributed with multiple auctioneers and bidders which participate in simultaneous auctions. The application makes essential use of the three forms of inter-thread communication of Qu-Prolog. The agent bidding behaviour is specified graphically as a finite state automaton and its implementation is essentially the execution of its state transition function. The paper assumes familiarity with Prolog and the basic concepts of multi-agent systems.

1 Introduction

Qu-Prolog is a multi-threaded Prolog designed specifically for developing distributed rational agent applications in which each agent can be multi-threaded. It started as a variant of Prolog for building interactive theorem provers and is the implementation language of the Ergo theorem prover [1]. We then added multi-threading and inter-thread communication via manipulation of the shared dynamic database and asynchronous messages [2] allowing the implementation of rational agent applications [3]. The final stage was to link Qu-Prolog with the Elvin [4] content-based message router to give us broadcast communication with message routing based on message pattern subscriptions. This also gave us a message interface to applications written in imperative programming languages such as C and Java since Elvin has APIs to many programming languages.

A key discriminator between object and agents is that agents have their own thread of control [5]. We go further, and believe that they are

naturally multi-threaded. Each thread is used to encapsulate a key behavioural component of the agent. For example, we can have a thread for each other agent with whom the agent is interacting via messages. Each conversation thread then accesses and updates a shared belief store. On occasion it suspends waiting for a key update to be made by some other thread. Thus, an agent's internal threads coordinate using a Linda style belief store [6], which in Qu-Prolog is the database of dynamic clauses. There is a Qu-Prolog primitive, `thread_wait_on_goal`, which causes a thread to suspend until some goal dependent upon the dynamic database succeeds, usually after a clause is asserted or retracted. This generalises the Linda `rd` lookup operation but its implementation is very efficient as the dynamic database is internal to the agent.

Agent communication languages such a KQML [7] and Fipa ACL [8] assume agent to agent asynchronous communication. In Qu-Prolog this is supported by a thread to thread communication model. Each thread has a message buffer and a unique name similar to an email address. Messages sent to an internal thread are copied to the destination thread's buffer. Communication between threads in different Qu-Prolog processes uses McCabe's ICM [9] store and forward communication servers to route the message between processes, which can be on different hosts. Various message receive primitives enable a thread to periodically search and remove from its message buffer messages of interest. If need be it can suspend, with a timeout, until an acceptable message is received. Since we can have a thread for each conversation, a conversation thread can suspend waiting for a reply to its last outgoing message. It is automatically resumed when a reply is received.

Some agent applications require broadcast communication. That is, an agent wants to send a message to any other agent interested in the message content without knowing the agent's identity. An example is the contract net protocol [10]. For this style of communication, it is better to use a communications server that routes messages based on content rather that destination identification. Qu-Prolog's primitives for connecting to, subscribing, and sending notifications to an Elvin [4] server give us this facility. A thread registers content test subscriptions with the Elvin server. Any notification message sent to the server that satisfies one of these tests is then automatically routed to the thread and placed in its message buffer.

In this paper we illustrate the use of these three forms of inter-thread communication, and their utility in building agent applications. The application we use is an auction house with multiple simultaneous auc-

tions comprising bidding agents and auctioneer agents. The application makes essential use of the three forms of inter-thead communication in Qu-Prolog. Bidding agents participate in multiple simultaneous auctions, each one conducted by its own auctioneer agent. A bidding agent starts with a wish list of items it wants to purchase with a maximum price it will pay for each item. It also has a limit to the total amount it can spend purchasing items in all the auctions. The bidding agents are multi-threaded with a thread for each auctioneer. Each bidder agent has its desires (the items it wants to buy), beliefs (about its purchases and unspent and committed funds) and intentions (its concurrently executing bidding threads). The application thus serves as an exemplar for the implementation of simple BDI agents concurrently executing intentions.

In the next section we sketch the structure of the auction application and the multi-threaded architecture of each bidding agent. In the section 3 we give an introduction to Qu-Prolog's thread spawning and inter-thread communication primitives illustrating their use with fragments of code from the application. In section 4 we specify the crucial bidding behaviour as a finite state automaton in which the state transitions are triggered by messages. In section 5 we show how the bidding agents and the auctioneers are implemented in Qu-Prolog. As we shall see, the bidding behaviour is essentially an execution of the finite state automaton transition function defined as a three argument relation. We summarise and mention some related agent implementation languages in section 6.

2 Auction application

The overall architecture of the application and the architecture of each bidding agent is depicted in Figure 1. Each auction is conducted as an ascending English auction. In each round the auctioneer calls for bids at a certain price. The sender of the first bid received at that price level becomes the potential purchaser. The auctioneer then raises the bid price and calls for bids at the new level. If no bids are received within a certain time limit the item is sold to the potential purchaser. The application is fully distributed, each bidding agent and each auctioneer runs as a separate Unix process, possibly on separate hosts.

An auctioneer agent broadcasts its bid calls, sold and withdrawn messages as notifications to an Elvin server. These are routed to the bidding thread for that auctioneer within each bidding agent because of subscriptions lodged by these threads with the Elvin server. Bids are sent directly to the auctioneer as ICM messages.

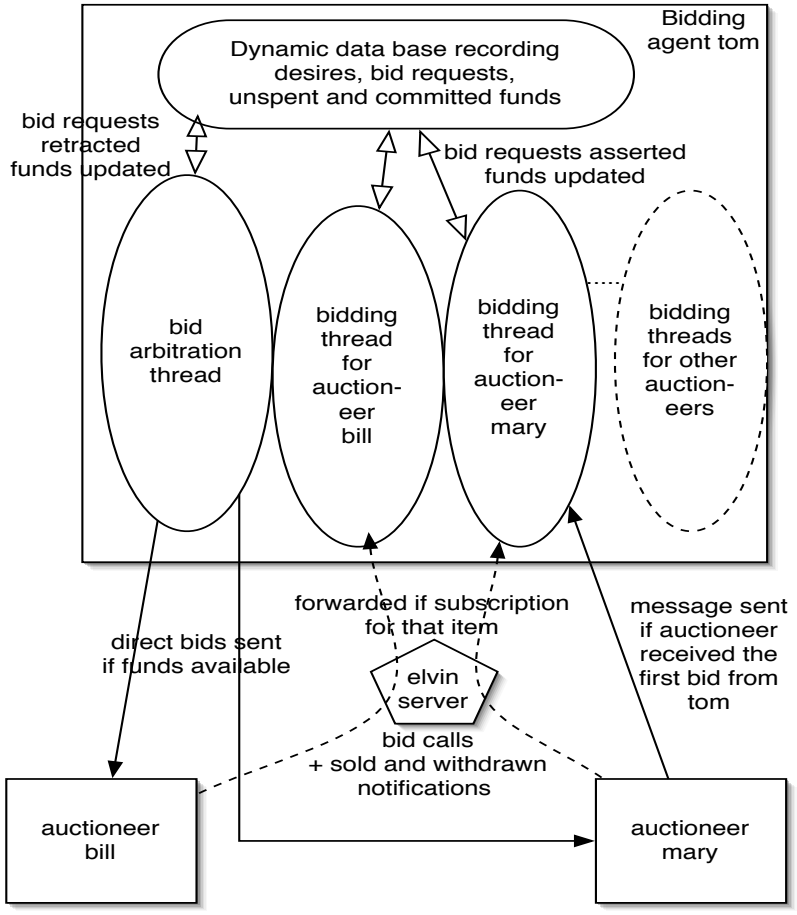

Fig. 1. Auction Architecture

Before sending a bid each bidding thread must check that there are available funds. Funds are reduced each time a bidding thread wins the auction for an item. Funds are provisionally committed when a bid is made, and released if the bid does not result in a purchase at that bid level. In the later stages this means that a bidding agent may have to wait before bidding in one auction until committed funds are released from another, and it may even have to skip a bidding round. This is because the agent cannot overspend its initial allocation.

The amount of the unspent and committed funds are held as dynamic clauses inside the agent. To allow the agent to make an overall judgement

about which item it should continue bidding for when funds become tight, and more than one item of interest is currently being auctioned, the decision to commit funds for a particular bid call is made by a separate *bid arbitration* thread. When the auctioneer specific thread receives a bid call at a price below or at its maximum price for the item, it informs the agent's bid handling thread that it wants to make a bid by asserting a bid request fact.

The bid arbitration thread suspends until there is at least one bid request. Its role is to find and process pending requests for which there are sufficient avaliable funds as quickly as possible so as not to miss a bidding round. If there is such a request for which there is sufficient remaining budget taking into account provisionally committed funds, it removes the request, atomically increases the committed funds by the value of the bid, and immediately sends the bid to the appropriate auctioneer indicating that any response should be sent to the thread that posted the request. A response will be sent only if the bid was the first one to be received by the auctioneer at the current call price. If the arbitration thread cannot find a request for which there are sufficient available funds it re-suspends until either a new bid request is asserted (because this may be at a lower price that is within the available funds), or committed funds are released by one of the auctioneer linked threads. This behaviour is achieved using a `thread_wait_on_goal` call as illustrated in 3.1.

The entire application has been implemented with a Tcl/Tk GUI visualisation, which is illustrated in Figure 2.

3 Threads and Inter-thread Communication

Qu-Prolog has several predicates for creating and controlling threads. The predicate `thread_fork(Name, Goal)` creates a new thread and executes `Goal` within the thread. For the thread executing the fork the call is deemed to always succeed and the forking thread immediately continues with the next call. The forked thread is terminated when `Goal` terminates (either with success or failure). If `Name` (an atom) is supplied then this name can be used as part of the address of the thread when using ICM communication. If it is not supplied, the system generates a name for the thread and unifies it with `Name`.

The predicate `thread_exit(Name)` terminates the thread with the given name. If `Name` is not supplied then the current thread is terminated.

Sometimes, one thread may need to carry out a computation (such as making several changes to the database) before giving control to other

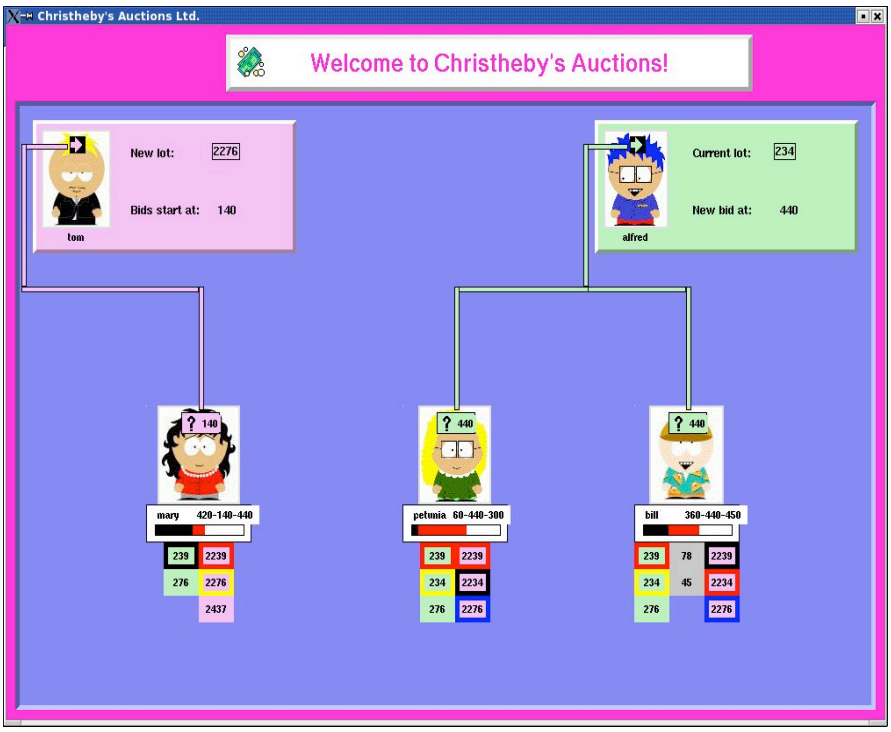

Fig. 2. Application Visualisation

threads. This can be achieved by using `thread_atomic_goal(Goal)`. When a thread enters such a call, no other thread will be given a time slice until `Goal` finishes executing (either in success or failure).

There are three mechanisms Qu-Prolog uses to communicate between threads. If the threads are within the same process then the threads can communicate using the dynamic database. Threads (in the same or different processes) can also communicate via messages using either ICM [9] or Elvin [4]. We now look at these in more detail.

3.1 Communication using the database

All threads within the same process share the dynamic database of the Qu-Prolog process and so when one thread asserts or retracts a clause, the effect is immediately visible to all the threads.

In some agent applications, the agent is implemented as several cooperating threads. One or more of these threads may be designed to wait for certain changes to the database before continuing with their execu-

tion. To achieve this behaviour we can use the single solution meta-call `thread_wait_on_goal(Goal)`. If `Goal` fails, `thread_wait_on_goal(Goal)` suspends until the dynamic database is changed in some way. `Goal` is then retried and will be retried on each update of the dynamic database until it succeeds. Of course, the call may never succeed, in which case the thread executing the `thread_wait_on_goal` call suspends forever.

We can often specify exactly what dynamic database predicates `Goal` depends upon. In this case `thread_wait_on_goal(Goal, PredList)` can be used. This will only be retried if some change is made to at least one of the dynamic predicates in `PredList`. These are the *watch* predicates of the call.

The code for the arbitration thread within a bidding agent which is responsible for making bids on behalf of the bidding threads uses both `thread_wait_on_goal/2` and `thread_atomic_goal/1`. The threads communicate by asserting and retracting `bid_requests/3` facts and by changing the `committed/1` and `budget/1` facts.

```
handle_bid_requests :-
  repeat,
  thread_wait_on_goal(choose_bid, [bid_request/3,committed/1]),
  fail.

choose_bid :-
    thread_atomic_goal(
      bid_request(Auctioneer, Item, Price),
      budget(Budget),
      committed(Committed),
      Price =< Budget - Committed,
      retract(bid_request(Auctioneer, Item, Price)),
      retract(committed(Committed)),
      NewCommited is Committed + Price,
      assert(committed(NewCommitted)),
      send_bid(Auctioneer, Item, Price)
    ).
```

The `committed/1` fact keeps track of the funds needed to cover sent bids that are not yet known to have succeeded or failed. The `budget/1` fact records funds that have not been spent. This is decreased only when the bidder wins the auction for some item and is changed by the bidding threads.

The two dynamic relations `bid_request/3` or `committed/1` determine whether or not the call to `choose_bid` succeeds. If there are no outstanding `bid_request`s, or if the value of the committed funds is such that

no such request can be covered from the remaining budget, the call to `choose_bid` suspends.

The call is resumed if either of these dynamic relations is updated (`budget/1` is never changed without `committed/1` being changed in the same atomic transaction). If the change was a new bid request, this may be for an amount that can be covered. If it was because the committed funds were reduced by a bidding thread when it learned its last bid did not result in a purchase at that price, funds may now be sufficient to cover a previous request.

The body of the `choose_bid` rule is executed as an atomic goal so that each time the call is entered or resumed no changes can be made to the dynamic database by the other bidder agent threads until it either succeeds or suspends; and, so that when it does succeed, the other threads will see a consistent update to the database. On success, a bid will have been sent, the `bid_request` that it has satisfied will have been retracted, and `committed` funds will have been increased by the amount of the bid. The `repeat/fail` iteration in the `handle_bid_requests` rule will now result in a new call of `choose_bid` inside the `thread_wait_on_goal`. It may immediately find another request it can satisfy. If not, it suspends until there is a change to one of the two dynamic relations it is watching for.

3.2 ICM Messages

The high-level peer-to-peer communication support of Qu-Prolog is based on the ICM. The ICM consists of one of more `icm` processes that act as message routers and an API that provides applications with ICM functionality. Using this, a process can register its name with an `icm` process and then send and receive messages via the `icm` processes.

ICM addresses have three main components: a thread name, a process name, and a machine address (the home of the process). An `icm` process uses the process name and home fields to determine the message destination. The process itself is responsible for managing the thread name field.

The Qu-Prolog implementation provides library support for the ICM API and manages an incoming message buffer for each thread within a Qu-Prolog process. The library provides two layers of support for ICM messages: a lower-level layer that provides the basic send and receive primitives, and a higher-level layer that further simplifies communication. In this paper we focus on the higher-level support.

In the higher-level layer the message send predicate is

```
Message ->> Address reply_to RTAddress
```

where `Message` is any Qu-Prolog term and `Address` is a Qu-Prolog term representing an ICM address. The `reply_to` part is optional and is used if the recipient of the message should forward a reply to some other thread.

The most general form for a Qu-Prolog address is

```
ThreadName:ProcessName@HostName
```

where the home part (`HostName`) can be dropped if the message is to another process on the same host. The process name (`ProcessName`) can also be dropped if the message is to another thread within the same process. The special address `self` can be used for a thread to send a message to itself. For agent applications, where each agent is a separate Qu-Prolog process, the process name is the agent name. Communication to local threads of a process does not use the ICM servers. The message is immediately placed at the end of the message buffer of the internal thread.

If a message is sent to a process that does not exist, one of the `icm` processes will store the message until the process is created. Similarly, if a message is sent to a thread that does not exist within a running process, the process will store the message until the thread is created.

In the `handle_bid_requests` clause given earlier, the `send_bid` action is defined by the clause:

```
send_bid(Auctioneer,Item,Price):-
   auctioneerAddress(Auctioneer,AuctioneerICMAdress),
   bid(Item,Price) ->> AuctioneerICMAddress reply_to Auctioneer,
   bid_sent(Item,Price) ->> Auctioneer.
```

`Auctioneer` is the name of the bidding thread that has made the bid request. The bidder has a relation, `auctioneerAddress` that stores the ICM address of the message interface threads of each auctioneer. It has facts such as:

```
auctioneerAddress(tom,thread0:tom@'zeus.doc.ic.ac.uk')
```

Here the name of the bidder thread `tom` is the name of the Qu-Prolog process that is the auctioneer agent.

The bid message is sent to the auctioneer agent, which will be on another host, with the `reply_to` set to be the bidding thread that made the request. This is so that the response the auctioneer will send, if this is the first bid received at that price, will go directly to that thread. In addition, a `bid_sent` message is put in the requesting thread's message buffer to alert it to the fact that a bid has been sent.

3.3 Elvin Messages

The high-level subscription/notification communication support of Qu-Prolog is based on Elvin. An Elvin notification is a list of field name, value pairs and is sent to an Elvin server that determines which processes are subscribed to this notification, and sends the notification to each of these processes. A process can subscribe to notifications they are interested in by using the Elvin subscription language. A subscription is a logical formula describing properties of notifications of interest - for example, notifications that contain a particular field or a particular value for a field.

Qu-Prolog threads can subscribe to Elvin notifications and any matching notifications are placed in that threads incoming message buffer. In order to distinguish Elvin notifications from ICM messages in the threads incoming message buffer, the sender and reply-to addresses are both set to the atom `elvin`.

Unlike ICM, Elvin has no memory of past notifications. A thread only receives those notifications satisfying a subscription that are posted *after* it has registered the subscription with the server.

In the auction application the bidder thread corresponding to auctioneer `tom` will send subscriptions to Elvin such as:

```
elvin_add_subscription(auctioneer==tom && item==lot2239)
```

The Elvin router will then forward any notification that contains an `auctioneer` field with value `tom` and an `item` field with value `lot2239` to this thread.

The auctioneer broadcasts to interested bidders by posting Elvin notifications with calls such as:

```
elvin_add_notification(
   [message_type=call,item=lot2239, price=204, auctioneer=tom])
```

The receiver of such a notification must make no assumption about the order of `field=value` pairs which may be changed by the Elvin server. So, a Prolog recipient must extract pairs from the message list using the `member/2` list membership relation.

3.4 Processing the Message Buffer

There are several predicates that process the incoming message buffer for a thread. The simplest of these are:

```
Message <<- Address reply_to RTAddress
Message <<= Address reply_to RTAddress
```

where the reply-to fields are again optional.

A `<<-` call removes the first message in the message buffer and tries to unify the arguments with information contained in this message (including the addresses). It suspends of there is no message, to be resumed when a message arrives. It fails if any of the unifications fail. A `<<=` call searches the message buffer looking for a message that unifies with the supplied patterns and removes the first such message. If no (unifying) message is found, the call suspends until a new message arrives. The second message receive never fails. Both are single solution calls.

Qu-Prolog also has a powerful `message_choice` operator that provides case analysis search of the message buffer with different response calls linked to different messages patterns. The program below is called by the auctioneer to handle bid messages in a bid round after it has broadcasted a bid call for an item `Item` at price `Price`.

```
handle_bids(Item, Price) :-
  message_choice (
    bid(Item, Price) <<- _ reply_to Bidder ->
       accept_bid(Item, Price, Bidder)
    ;
    bid(I,P) <<- _ :: (I\=Item;P\=Price)  ->
       handle_bids(Item, Price)
    ;
    timeout(7) ->
       bid_timeout_for(Item)
  ).
```

The argument of the `message_choice` operator has the same structure and similar semantics to the if-then-else construct in Prolog except that each test is a message pattern with an optional test following the `::` operator.

The first `->` rule matches any message of the form `bid(Item,Price)` which is a bid for `Item` at the current call price `Price`. The `reply_to` associated with the message, the address of the bidding thread within the bidder for this auctioneer, is assigned to the variable `Bidder`. It is used by `accept_bid` to record the bidder's identity and to send an acknowledgment message. The sender is ignored since this is the bidder's arbitration thread. The second `->` rule matches any `bid` message for a different item or a different price. This it so that such a message - usually a late bid for a previous round - can be discarded. The `handle_bids` program is then

recalled to search for, and if need be wait for, the first valid bid for this round. The last rule specifies a timeout in seconds on how long the auctioneer will wait for a message that can be handled by the first two rules. If no such message has arrived within seven seconds then `bid_timeout/1` will be called.

Use of a timeout rule is optional, and if it is not supplied, `message_choice` will block until some message that can be handled by one of its rules arrives.

4 Bidding behaviour

Figure 3 is a finite state automaton which characterises the bidding behaviour of a thread `A` within a bidding agent monitoring the announcements of auctioneer `AA`. The thread starts by posting Elvin subscriptions as given in section 3.3. These ensure that whenever `AA` broadcasts a message via Elvin about an item `I` the bidder wants to buy that message will appear in the thread's message buffer. Auctioneer announcements about items the bidder does not want to buy are not seen.

`A` then behaves in accordance with the finite state machine of Figure 3 starting at the `desire(I,MP)` node. `MP` is the maximum price it is prepared to pay for item `I`. It stays in that control state until it receives a message concerning any such item `I`. The state transitions are triggered by messages that `A` receives either via Elvin or directly. In the figure all messages are denoted as simple functor terms such as `call(I,P)` even though the actual message may be a list of attribute value pairs.

If the bidder agent has arrived at the auction after auctioneer `AA` has started, the first message it receives could be a `wd(I)` (withdrawn) or a `sold(I)` message for the item `I`. In this case the bidder enters the `exit(I)` state for the auction of `I` and then re-enters the finite state machine at the `desire` node to wait for a bid call for another item it wants to buy. It enters the `exit(I)` state whenever it receives either of these messages regarding item `I`. We assume that the auctioneer always starts the bidding at the reserve price for an item so that a withdrawn message is only sent when there are no bids received in the time the auctioneer allows for a bid response to a call.

More usually, the initial state is left when a `call(I,P)` message is the first message received asking for bids for item `I` at bid level `P`. If `P>MP` the behaviour moves into the `exceed_max(I)` state and subsequent bid calls for item `I` are ignored. The bidding behaviour moves into the

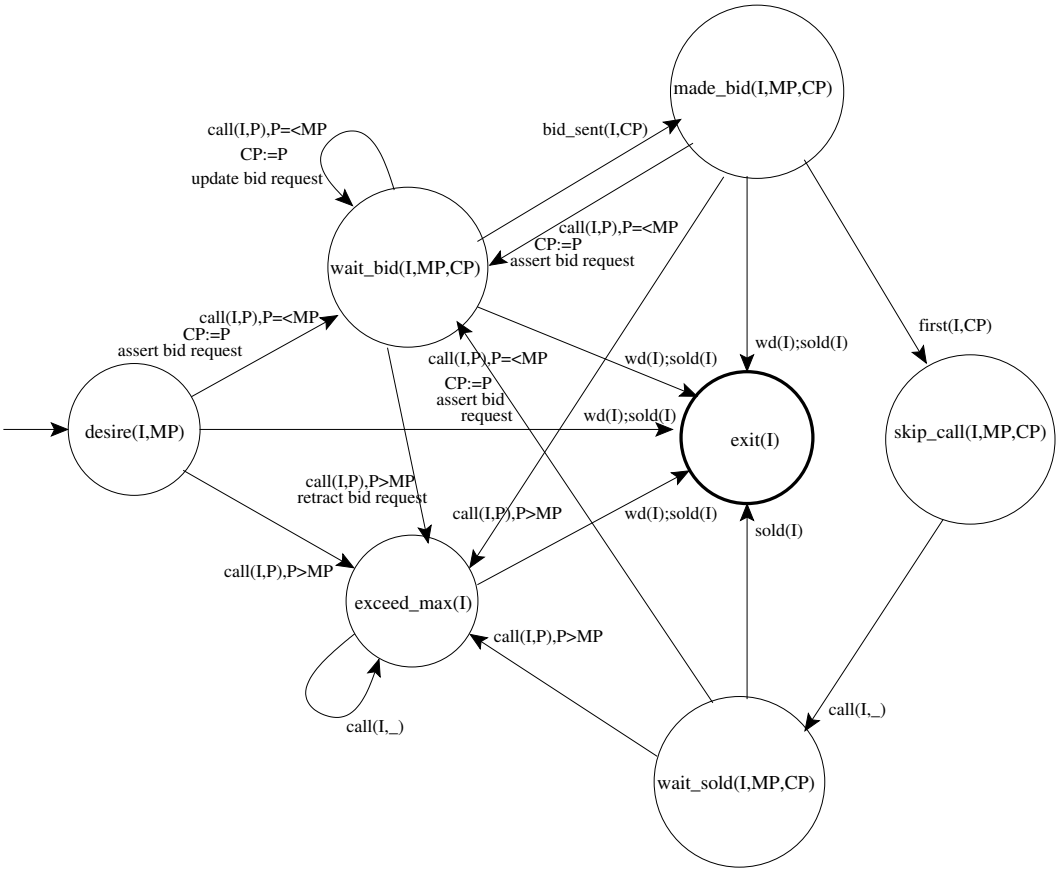

Fig. 3. Bidding behaviour

`exceed_max(I)` state whenever a bid call is received for `I` at a price above the bidder's maximum price `MP`.

The other transition from the initial state is to the `wait_bid(I,MP,CP)` state. This happens if `call(I,P)` is received and `P=<MP`. `CP` just records the current bid price `P`. As part of the state transition the thread will also assert a `bid_request(A,I,CP)` fact. This is a request to the bid arbitration thread to send a `bid(I,CP)` message to the auctioneer `AA` on its behalf. The arbitration thread executes the program discussed in section 3.1.

In the `wait_bid(I,MP,CP)` state thread `A` may receive a `bid_sent(I,CP)` internal message from the arbitration thread. The arbitration thread places this message in `A`'s message buffer immediately after sending the bid to auctioneer `AA` as an ICM message. In that case, the behaviour

moves to the made_bid(I,MP,CP) state. Alternatively, it may first get a new message from its auctioneer concerning item I. This happens if the auctioneer AA has received a bid at the current bid price CP from another bidder, or no bids were received at this bid level within the auctioneer's time limit, *and* either event occurred before the arbitration thread decided to send a bid for item I and to inform A by inserting a bid_sent(I,CP) in its message buffer.

If the time limit was reached, auctioneer AA may broadcast a sold(I) message, indicating a sale to another bidder - the one that got in the first bid at the previous bid call price, or it broadcasts a wd(I) message (when no bids were received in time in response to its first call for bids for I).

If a new call(I,P) message is the first message to be received in the wait_bid(I,MP,CP) state, another bidder has got its bid in first. Thread A retracts its bid_request(A,I,CP), if it is still there. Then, if P=<MP, CP is updated to be the new P value, and a new bid request fact is asserted at the new CP level for consideration by the arbitration thread. If P>MP, the behaviour moves to the exceed_max(I) state.

The previous bid request fact may no longer be in the dynamic database because it has been retracted by the concurrently executing arbitration thread prior to sending A a bid_sent message and a (too late) bid to auctioneer AA. In this case the arbitration thread will also have added CP to the committed funds, so these will have to be reduced by this amount. As part of this roll back operation, the sent_bid(I,CP) message, that will have been placed in A's message buffer by the arbitration thread, might as well be removed from the buffer. If not, it will be repeatedly skipped over and ignored.

If a bid_sent(I,CP) message is the first message received in the state wait_bid(I,MP,CP), the behaviour moves into the made_bid(I,MP,CP) state to wait for a possible first(I,CP) message to be received from auctioneer AA letting it know that the bid(I,CP) message that was sent by the arbitration thread was the first bid received. If it gets this message, the behaviour moves into the skip_call(I) state as the bidder is now a potential purchaser of I and will win the auction if no one bids in response to the next bid call for I at the raised price. Receipt of the next call(I,_) message simply moves the behaviour from the skip_call(I) state to the wait_sold(I,MP,CP) state. In this state, a sold(I) message broadcast by the auctioneer indicates that no bids were received in response to the last bid call and hence that thread A has won the auction for item I. In the resulting transition to the exit(I) state, the committed and

unspent funds are both decreased by amount CP, this double update being performed atomically.

The other message that might be received in the wait_sold(I,MP,CP) state is a new call(I,P) message. This will be broadcast by the auctioneer if it has received a bid in response to the raised price call. Bidding thread A has therefore not won the auction at price level CP. It now either moves into the exceed_max(I) or the wait_bid(I,MP,CP) state, with CP updated to the new call price P, depending upon whether or not P exceeds its maximum MP for item I. In either case, it decreases its committed funds by amount CP, releasing the allocation that was made for its last bid.

5 Auction implementation in Qu-Prolog

5.1 The bidding agents

Each bidding agent program has facts for the predicates:

auctioneer(A,AH): AH is the ICM handle for auctioneer A
desire(A,I,MP): Bidder wants item I from A at max. price MP

giving housekeeping details about the auctioneers and the purchasing desires of that bidder. It also has initial facts for dynamic predicates:

budget(B) : B is amount left to spend in all auctions
committed(C) : C is amount currently committed in outstanding bids

The initial budget fact records the total amount each bidder has to spend when they arrive at the auction. The amount initially committed will be 0.

Each bidder agent starts by executing a call to the program:

```
bidder :-
    thread_fork(arbiter, handle_bid_requests),
    forall(auctioneer(Auctioner,_),
             thread_fork(Auctioneer,
                         (post_subscriptions_for(Auctioneer),
                          monitor_auction) ).
```

The first thread_fork launches the bid arbitration thread executing the program given in section 3.1. The forall then launches a bidding thread for each auctioneer with the name of the auctioneer. This thread starts by posting subscriptions to the Elvin server as exemplified in 3.3 so that it will be sent just those Elvin messages for items it wants to buy from its auctioneer. It then executes the program monitor_auction.

```
monitor_auction :-
   bidding_behaviour_from(desire(I,MP)),
   monitor_auction.
```

The `bidding_behaviour_from(desire(I,MP))` call is to a program that will follow the behaviour described by the finite state machine of section 4, starting at the state `desire(I,MP)`. At this stage values of `I` and `MP` are unknown. They will be bound when the first message has been processed.

The call will terminate when the behaviour reaches the state `exit(I)`. The `monitor_auction` program then recurses to re-enter the bidding behaviour at the state `desire(I,MP)`[1].

The `bidding_behaviour_from` program is just a recursive program that walks over the finite state automaton of Figure 3 until the `exit` state is reached. We assume that the state transitions of the machine are defined by a `next_state/3` relation.

```
bidding_behaviour_from(exit(_)).
bidding_behaviour_from(State) :-
   essence_of_next_message(State, M),
   next_state(State, M, NxtState),!,
   bidding_behaviour_from(NxtState).
```

An example rule for `essence_of_next_message` is:

```
essence_of_next_message(desire(I,MP), Msg):-
   ElvinMsg <<= elvin,
   member(item=I, ElvinMsg),
   thread_symbol(A),      % get auctioneer name of this thread
   desire(A, I, MP),      % find bidder's max price for I
   ( elvin_call_message(ElvinMsg, P), Msg=call(I,P)
    ;
     elvin_sold_message(ElvinMsg), Msg=sold(I)
    ;
     elvin_withdrawn_message(ElvinMsg), Msg=wd(I)
    ).
```

where:

[1] Our running auction implementation has the auctioneer broadcast an `auction_over` notification when it has no more items to auction. The above `monitor_auction` program suspends if the auctioneer just stops sending out calls for bids. It will terminate in failure, and hence cause the bidding thread to terminate, if an `auction_over` notification is broadcast.

```
elvin_call_message(ElvinMsg, P) :-
   member(message_type=call, ElvinMsg),
   member(price=P,ElvinMsg).
elvin_sold_message(ElvinMsg) :-
   member(message_type=sold, ElvinMsg),
....
```

In the `desire` state only an Elvin message can be received. Its 'essence' is one of the terms used in the finite state machine of Figure 3. The term constructed for the Elvin message

```
[message_type=call, item=lot2239, price=204, auctioneer=tom]
```

is

```
call(lot2239,204).
```

Example `next_state` rules are:

```
next_state(desire(I,MP), call(I,P), wait_bid(I,MP,P)):-
   P=<MP,
   thread_symbol(A),
   assert(bid_request(A, I, P)).
next_state(desire(I,MP), call(I,P), exceed_max(I)):-
   P>MP.
next_state(desire(I,MP), M, exit(I)):- M=sold(I); M=wd(I).

next_state(wait_bid(I,MP,P), bid_sent(I,P), made_bid(I,MP,P)).
next_state(wait_bid(I,MP,P), call(I,NewP), wait_bid(I, MP, NewP)):-
   NewP=<MP,
   thread_symbol(A),
   thread_atomic_call(
     (remove_bid_request(A, I, P), assert(bid_request(A,I,NewP)).
next_state(wait_bid(I,MP,P), call(I,NewP), exceed_max(I)):-
   NewP > MP,
   thread_symbol(A),
   thread_atomic_call(remove_bid_request(A, I, P)).

remove_bid_request(A, I, P):-
   (retract(bid_request(A, I, P)) -> true
       ;     % retract failed, request already retracted by arb. thread
       retract(committed(F)),
       NewF is F-P,
       assert(committed(NewF)),
       bid_sent(I,P) <<= arbiter)).
```

A `remove_bid_request(A,I,P)` call removes `P` from the committed funds if the bid request message has already been deleted by the arbitration thread (the `retract` fails) and it also discards the `bid_sent` message which will be in its message buffer, as per the discussion in section 4.

As these example rules show, it is relatively easy to produce the `next_state` rules from the finite state machine and the discussion of section 4.

5.2 The auctioneers

Each auctioneer program has facts for the predicate:

`item(I,RP)`: Item `I` has reserve price `RP`

detailing all the items that the auctioneer has to auction. The reserve price is the minimum price at which the item can be sold and is the price used to start the bidding. Each auctioneer program also has a fact for `my_name/1`, recording the auctioneer's name.

Compared to the bidding agents, the auctioneers execute a quite simple behavioural program. Each executes `auction`.

```
auction :-
  (retract(item(Item,RPrice)) -> send_bid_call(Item,RPrice) ; true).

send_bid_call(Item,Price) :-
    myname(Name),
    elvin_add_notification([message_type=call,item=Item,
                            auctioneer=Name,price=Price]),
    handle_bids(Item,Price).
```

where `handle_bids` is the program given in section 3.4.

As discussed there, this invokes `accept_bid(Item,Price,Bidder)` if the first bid received at the `Price` was from `Bidder`, and this was received within seven seconds of the call notification. This program asserts the fact:

```
potential_purchaser(Bidder,Item,Price)
```

after retracting any other such fact about `Item`. `Bidder` is now the potential purchaser of `Item` at `Price`. `accept_bid` also sends a `first` message directly to `Bidder` to inform them of this. It then increments `Price` by a fixed amount to give a new call price `NewPrice` and then executes `send_bid_call(Item,NewPrice)`.

At some stage the `timeout` rule of `handle_bids` will be triggered when no bids are received in time. This invokes `bid_timeout_for(Item)`. If a potential purchaser has been recorded for `Item` an Elvin notification is sent that the item has been sold and the `potential_purchaser` fact is

replaced by a `purchased` fact; else a notification is sent that the item has been withdrawn. The latter only occurs if no bids are received in time following the first bid call. The auctioneer program then iterates with a new call to `auction`. It terminates when there are no more items to be auctioned.

6 Concluding Remarks

We trust we have demonstrated the expressiveness of Qu-Prolog for programming distributed agent applications in which the agents are subject to real time constraints, such as timely reaction to a bid call, and where agents may have to concurrently interact with several other agents taking into account limited shared resources, such as money to spend. Such agents can be programmed using multiple internal threads communicating via the shared dynamic database, or via messages, in order to co-ordinate access and use of the resources they must share.

In the case of our bidding agents a separate arbitration thread is in charge of allocating the shared money resource, and shared dynamic predicates and message passing are used to co-ordinate the threads. The program for the arbitration thread uses a simple strategy to allocate funds. It allocates to the first bid request it finds for which sufficient funds are available at that time. It is quite easy to change this to take into account preferences for items, and, say, the difference between the current call price and the maximum price the bidder is prepared to pay. It can then choose the pending request with maximum utility computed as a function of its preference rating and the price differential.

Qu-Prolog's interface to the ICM message servers enables us to seamlessly perform private agent to agent communication across the internet using symbolic names for threads similar to email addresses. Our application uses the ICM system to send private bid messages and acknowledgments for first bids. Using proxy servers, the ICM system [9] allows messages to be routed through fire walls and to be automatically downloaded to agents on mobile devices, such as laptops, when they connect to the network.

The interface to Elvin enables us to quickly build agent applications that need to use a message broadcast mechanism with messages routed to agents based on receiver subscriptions. It also gives us a mechanism for quickly building hybrid applications. One such might be an agent monitoring application in which sensor software written in Java or C posts notifications routed to the monitoring agents. The role and function of the

monitoring agents can be changed without changing the sensor software simply by changing subscriptions.

Lodging appropriate subscriptions or sending notifications to the Elvin server gives an easy method for an agent to join an existing community of agents and components. Posting subscriptions is the joining mechanism for bidding agents at the auction. Posting notifications is the joining mechanism for the auctioneers.

To join the community an agent needs to know the message format being used. In open agent applications KQML or Fipa ACL based messages may be being used. The structure of Elvin notifications allows almost direct representation of such messages, which are also based on a list of attribute/value pairs. If the application uses KQML messages, the Elvin server takes over some of the role of a KQML facilitator [7].

The features of Qu-Prolog that support the writing of non-resolution inference systems are illustrated in [3] but they are more fully explained in the Qu-Prolog User Guide. The system and its documentation are down-loadable from:

```
http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html
```

Qu-Prolog is used as the programming language used for a course on multi-agent systems at Imperial College and it was the language used by one of the winning submission [11] for the Agent Programming Competition of CLIMA VI. It is also being used in the ARC Center for Complex Systems at the University of Queensland for the simulation of insect behaviour, and for the proto-typing of free flight air traffic control in which aeroplane agents negotiate over flight levels and flight paths to avoid near misses and collisions. Currently the language does not have a finite domain constraints but it does have delayed calls linked to unbound variables and primitives to retrieve the delayed calls. We plan to add finite domain constraints using these features.

6.1 Some Related Languages

SICStus-MT [12] is a multi-threaded version of SICStus Prolog in which threads each have a single message buffer, called a port. As in Qu-Prolog, a thread can scan the buffer looking for a message unifying with a given message pattern, suspending if no such message is found. However, port communication is restricted to threads within the same Prolog process. SICStus Prolog does have a Linda package, but the Linda store is for

external communication between different Prolog processes, or between a Prolog process and a process implemented in another programming language, not for internal thread coordination. For communication between processes in different languages we would use Elvin or the ICM API.

BinProlog[13] is a multi-threaded Prolog with a tight coupling to Java and communication between threads in Prolog or Java via Linda tuple stores. It also supports mobile agents via thread migration between different BinProlog processes.

QuP^{++} [14] is an object oriented extension of Qu-Prolog that allows a class structure with multiple inheritance to be used to construct multi-threaded agent applications. In QuP^{++} a class instance is an active object with at least one thread of control. This thread handles messages from other objects, can launch new internal threads, and can create new active objects.

Mozart-Oz[15] is a multi-paradigm distributed symbolic programming language with support for logic programming, functional programming and constraint handling. It is being used for distributed agent applications. Mozart-Oz is multi-threaded with the threads sharing a common store of values and constraints. The store is used for inter-thread communication. Constraints are posted to the store and the store can be queried as to whether some particular constraint is entailed by the current constraint store. A thread executing such a query will suspend until the store entails the constraint.

The CIAO system [16] uses the dynamic Prolog database for communicating between threads in the same process. Whereas Qu-Prolog uses `assert` and `retract` to update the database and `thread_wait_on_goal/2` to wait for changes to named dynamic predicates, the CIAO system requires the dynamic predicates that can lead to a thread suspension to be declared as `concurrent`. Changes to clauses for concurrent predicates are used for stream communication between threads. A normal call to a concurrent predicate will suspend if it cannot succeed, even on backtracking. Thus, a thread will suspend when a call to a concurrent predicate has 'seen' all the clauses for the predicate that have so far been asserted by the other threads. This allows the dynamic database to be used to communicate a stream of data between threads, as an incrementally asserted set of facts, with automatic suspension of consuming threads that run ahead of the producers. In Qu-Prolog we would achieve this by using ICM or Elvin message communication.

April [17] is a multi-threaded hybid functional/imperative programming language that also uses ICM servers to communicate messages between threads in different applications.

Erlang [18] is a functional multi-threaded language with a single message buffer for each thread. The `message_choice` operator of Qu-Prolog is modeled on the message receive primitive of Erlang.

Go! [19] is a multi-threaded multi-paradigm functional, logic and OO programming language in which threads can communicate using mailbox objects, or dynamic relation objects that act as Linda stores. Each mailbox, which is typically private to a thread, can have any number of linked dropbox objects that can be shared with other threads and used by them to post messages to the mailbox. A thread will suspend waiting for a particular message to be posted to a mailbox.

References

1. P. J. Robinson. Ergo Reference Manual. Technical report, ITEE, University of Queensland, http://www.itee.uq.edu.au/~pjr/HomePages/ErgoFiles/cover.html.
2. Keith L. Clark, Peter J. Robinson, and Richard Hagen. Multi-threading and message communication in Qu-Prolog. *Theory and Practice of Logic Programming*, 1(3):283–301, 2001.
3. P. J. Robinson, M. Hinchley, and K. L. Clark. Qu-Prolog: An Implementation Language with Advanced Reasoning Capabilities. In M. Hinchley et al, editor, *Formal Appraches to Agent Based systems, LNAI 2699*. Springer, 2003.
4. B. Segall et al. Content based routing with elvin4. In *Proceedings AUUG2K*. Canberra, Australia, Downloadable from http://elvin.dstc.com/doc/papers/auug2k/auug2k.pdf, 2000.
5. M. J. Wooldridge and P.Ciancarini. Agent Oriented Software Engineering: the State of the Art. In P.Ciancarini and M. J. Wooldridge, editors, *Agent Oriented Software Engineering*, volume 1957 of *LNCS*, pages 1–28. Springer-Verlag, 2001.
6. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
7. T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings 3rd International Conference on Information and Knowledge Management*, 1994.
8. FIPA. Fipa communicative act library specification. Technical report, Foundation for Intelligent Physical Agents, www.fipa.org, 2002.
9. F.G. McCabe. *ICM Reference Manual*. Fujitsu laboratories Ltd. Downloadable from: http://sourceforge.net/projects/networkagent/, 1999.
10. Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In Alan H. Bond and Les Gasser, editors, *Readings in distributed aritficial intelligence*, pages 357–366. Morgan Kaufmann, 1988.
11. S. Coffey and D. Gaertner. Using pheromones, broadcasting and negotiation for agent gathering tasks. in this volume. 2006.
12. Jesper Eskilson and Mats Carlsson. SICStus MT—A Multithreaded Execution Environment for SICStus Prolog. In C. Palamidessi, H. Glaser, and K. Meinke,

editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1490 of *Lecture Notes in Computer Science*, pages 36–53. Springer-Verlag, 1998.
13. Paul Tarau. BinProlog 9.x Professional Edition: User Guide. Technical report, BinNet Corp., 2002. Available from http://www.binnetcorp.com/BinProlog.
14. K. L. Clark and P. J. Robinson. Agents as Multi-threaded Logical Objects. In T. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*. LNAI 2699, Springer, 1998.
15. Peter Van Roy and Seif Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*. http://www.mozart-oz.org/papers/abstracts/diplcl99.html, 1999. Part of International Conference on Logic Programming (ICLP 99).
16. D. Cabenza M. Hemenegildo and M. Carro. On the uses of attributed variables in parallel and concurrent logic programming systems. In L Sterling, editor, *Proceedings of ICLP95*, pages 631–645. MIT Press, 1995.
17. F.G. McCabe and K.L. Clark. April - Agent PRocess Interaction Language. In N. Jennings and M. Wooldridge, editors, *Intelligent Agents*, pages 324–340. Springer-Verlag, LNAI, 890, 1995.
18. J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall International, 1993.
19. K. L. Clark and F. G. McCabe. Go! – a Multi-paradigm programming language for implementing Multi-threaded agents. *Annals of Mathematics and Artificial Intelligence*, 41(2-4):171–206, 2004.