Tutorial on tuProlog agents in DCaseLP

To show how tuProlog agents can easily be programmed and executed in DCaseLP, we will go through, step by step, the creation and implementation of a small MAS prototype whose components are both **Java** and **tuProlog** agents.[1]

The MAS we are about to create will represent a simple *distributed marketplace* where there are agents willing to *buy* fruit and agents wishing to *sell* it. To simplify our example, we will suppose that the fruit sold in the market is only **oranges, apples** and **kiwi**; moreover, there are only **3 shops** and only **2 people buying**. The shops where people buy and the people themselves can all be represented by agents. We will call *buyer1* and *buyer2* the agents representing the people, while we will address with *seller*, *seller1* and *seller2* the agents representing the shops. Thus, our MAS consists of **5 agents**. These agents could be Java, tuProlog or Jess agents. We will programme the behaviour of **4** of these agents using **tuProlog**, while we will use **Java** for the remaining one, for example the agent *seller*.

What is the next step?

The next step we will carry out is to determine what kind of **interactions** can take place among our agents. All the interactions among the agents will imply **sending/receiving messages** through the JADE platforms in which the agents are deployed. So, what type of messages do we want to consider? Well, given that our example is simple, we can probably expect the agents that sell fruit to mainly **receive 2** different types of *requests* from the buyers:

- the **price** at which the fruit is sold;
- an amount of fruit to **buy**.

For the purpose of simplicity, we will assume that the buyers always buy the *same amount* of fruit: that amount can vary according to the type of fruit. Obviously, the sellers will have to *keep track of the amount of fruit that they have in stock* in order to be able to respond to the requests made by the buyers: if they

[1] The reader can find more details about the tuProlog agents in DCaseLP, this example and its execution, in the Master Thesis of Ivana Gungui (in English) that can be downloaded using this link: http://www.disi.unige.it/person/MascardiV/Download/Gungui.pdf.gz .

have enough fruit they will sell it, otherwise they will have to *inform* the buyer that the quantity of fruit they have in stock is not enough.
When a sale of fruit can be carried out, the seller will send a message to the buyer to *inform* it of the successful transaction. On the other hand, the buyer will need to *keep track of the amount of money that it has available*.
The messages that the agents exchange can then be summarised as follows:

INTERACTIONS in our MAS

Message	**buyers**	**sellers**
REQUEST **price**	send	receive
INFORM **price**	receive	send
REQUEST **buy**	send	receive
INFORM **bought**	receive	send
INFORM **no_more**	receive	send

Now that we have established the possible interactions that can take place in our MAS, we must decide what should be the initial situation of the MAS. What would happen in a market? The people intending to buy fruit would go round having a look at the prices of the different shops and then would decide to buy where the fruit is cheaper. Therefore, it seems a good idea if, at the beginning, the *buyers* request the price of each fruit to all the *sellers*: obviously the order with which these questions are carried out cannot be decided before executing the MAS and will have to be random.

Once one *buyer* knows the prices of the fruit, independently of what the other *buyer* is doing, it will send requests for buying the cheapest fruit. In our MAS, a

buyer will carry on trying to buy fruit while it has **money** that allows it to do so, while a *seller* will carry on selling until it has **enough fruit** to sell.

For what we have said until now, it can be gathered that the 2 tuProlog agents buying fruit, ***buyer1*** and ***buyer2***, behave in the same way and differ only in the quantity of fruit that they wish to buy for each fruit and in the quantity of money that they own. Thus, we can write the *theory file* for one of them, let's say ***buyer1***, and then copy it for ***buyer2*** modifying it accordingly.

Similarly, the 2 tuProlog agents selling fruit, namely ***seller1*** and ***seller2***, will have the same theory file, with different facts regarding the quantity of fruit in stock and the prices at which the fruit are sold.

We will see the Java agent selling fruit, that is ***seller***, at the end.

The buyer agent

We will first detail what the **knowledge base** of the agent will contain, and afterwards we will show the definition of the ***main*** predicate. The user must though remember that the clauses defining the `main` predicate will have to appear *at the beginning of the theory file*.

First of all, we decide the quantity that this agent will buy for each fruit so, in the text file containing the tuProlog theory for the agent ***buyer1***, we add the following lines:

```
buys(goods(oranges),quantity(2)) :- true.
buys(goods(apples),quantity(3)) :- true.
buys(goods(kiwi),quantity(12)) :- true.
```

The term **goods**(*term*) represents a type of fruit, while **quantity**(*term*) is used to express how many kilograms of the relative fruit the agent buys.

We will write in blue the data that will probably be different in the theory file of the agent ***buyer2***.

The buyer also needs to know how much money it owns, so we add this fact:

```
money(200) :- true.
```

To be able to *decide* which seller sells a fruit at the cheapest price, the buyer will have to *keep track* of the prices applied by each seller.
We can thus add:

```
price(seller,oranges,na) :- true.
price(seller1,oranges,na) :- true.
price(seller2,oranges,na) :- true.
price(seller,apples,na) :- true.
```

```
price(seller1,apples,na) :- true.
price(seller2,apples,na) :- true.
price(seller,kiwi,na) :- true.
price(seller1,kiwi,na) :- true.
price(seller2,kiwi,na) :- true.
```

We have used the constant ***na*** (**n**ot **a**vailable) to indicate the situation when the price applied by the seller is not yet known to the buyer. At the beginning, the buyers are not aware of the prices and, for simplicity, we suppose that they do know the names of the sellers.

Previously we have decided that the first task of the buyers will be to find out the prices of the fruit, but there is a situation that we must take into account: what happens in case the answer of a seller is lost or does not arrive before the buyer agent starts its cycle again; the buyer would repeat the request because it does not know the price yet. To avoid a buyer agent sending more than one request to the same seller, we use this auxiliary predicate:

asked(*fruit*,*seller*,*yes/no*)**.**

that will allow the agent to know if it has already sent a request for a price. Thus, we add the facts shown below:

```
asked(oranges,seller,no) :- true.
asked(apples,seller,no) :- true.
asked(kiwi,seller,no) :- true.
asked(oranges,seller1,no) :- true.
asked(apples,seller1,no) :- true.
asked(kiwi,seller1,no) :- true.
asked(oranges,seller2,no) :- true.
asked(apples,seller2,no) :- true.
asked(kiwi,seller2,no) :- true.
```

The constant **no** indicates that the agent *has not yet asked* the price of the fruit to the corresponding seller agent. In case the agent needs to know how much of a fruit it has bought, we can add the following:

```
goods_possessed(oranges,0) :- true.
goods_possessed(apples,0) :- true.
goods_possessed(kiwi,0) :- true.
```

Finally, in order to refer to the seller agents by using the constants **seller**, **seller1**

and **seller2**, instead of the strings representing their JADE addresses (GUIDs), we add:

```
address_name("seller@ai:1099/JADE",seller):- true.
address_name("seller1@ai:1099/JADE",seller1):- true.
address_name("seller2@ai:1099/JADE",seller2):- true.
```

The first argument of the predicate is the JADE GUID of the seller agent: in our example, the seller agents are all running in a JADE platform running on a computer named **ai**.

We could programme the agent to ask the Directory Facilitator agent (available in JADE) for the addresses of the seller agents and then store them in the knowledge base, but for simplicity we prefer to assume that the buyers already know the addresses of the sellers.
Finally, we add two more facts to the theory:

```
goods_list([oranges,apples,kiwi]) :- true.

sellers_addresses(["seller@ai:1099/JADE",
   "seller1@ai:1099/JADE",
   "seller2@ai:1099/JADE"]) :- true.
```

The first fact is used to know the list of fruit available in the market, while the second one tto know the list of the JADE addresses corresponding to the seller agents.

The main activities carried out by the buyer agent are, in order: *handling incoming messages*, *asking the price of the fruit to sellers* (performed until it knows the price at which each fruit is sold by every agent) and *buying fruit*.

The simple way of programming the behaviour of the agent is, therefore, to separate these tasks and deal with them separately by calling auxiliary predicates that will perform them. So, **at the beginning of the theory file** (where the definition of the `main` predicate must be!) we write the following rule:

```
main :- handle_msgs,ask_prices,buy_goods.
```

Now we will go through each auxiliary predicate in detail.

handle_msgs

This activity mainly consists on receiving a message and processing it in the ap-

propriate manner. The first action will therefore be the call to the `receive` predicate that we have defined in the ***TuJadeLibrary***. We will make use of another predicate to examine the message received (if there is one) and decide how to proceed.

We can define the `handle_msgs` predicate as follows:

```
handle_msgs :- receive(Performative,Message,Sender),
   select(Performative,Message,Sender).
```

The `receive` predicate is successful both if the agent has received a message and if it has not. If the agent has at least received a message then the `Performative`, `Message` and `Sender` variables are bound to information regarding the message received, otherwise they remain unbound. The `select` predicate has a multiple definition: one deals with the messages that are sent by the seller agents while the other one assures that the predicate never fails, avoiding the failure of the `main` predicate which would cause an error.

```
select(Performative,Message,Sender) :-
   bound(Performative),bound(Message),
   address_name(Sender,Name),unpack(Message,TermMsg),
   handle(Performative,TermMsg,Sender).

select(_,_,_) :- true.
```

If the `bound` predicate fails it means that no message has been received, so the agent will continue by evaluating the next predicate in the `main` goal, that is `ask_prices`. In case a message has been received, we must check if it has been sent by a seller agent, so we call the `address_name` goal: if it fails then the sender is not one of the seller agents, so the message is discarded; on the other hand, if it does not fail, we must handle its content.

Before examining the content, we call the `unpack` predicate, also defined in the ***TuJadeLibrary***, in order to create the `TermMsg` term corresponding to the tuProlog term that the seller agent intended to send but was converted into a string to be inserted as the content of the message.

At the beginning we have pointed out what are the interactions that take place among our agents, so we know that a buyer agent will receive only three different types of messages from the seller agents. Since the action that the buyer agent needs to carry out depends on the *predicate* that appears in the term that has been sent by the seller agent, we define the `handle` predicate as shown below:

```
handle("INFORM",price(Goods,Price),Sender) :-
   bound(Goods),bound(Price),address_name(Sender,S),
   retract(price(S,Goods,_)),
```

```
    assert(price(S,Goods,Price)).

handle("INFORM",bought(Goods),Sender) :-
    bound(Goods),address_name(Sender,S),
    price(S,Goods,P),
    retract(money(M)),
    retract(goods_possessed(Goods,X)),
    buys(goods(Goods),quantity(Q)),N is X+Q,P \= na,
    NM is M--P,assert(money(NM)),
    assert(goods_possessed(Goods,N)).

handle("INFORM",no_more(Goods),Sender) :-
    bound(Goods),address_name(Sender,S),
    retract(price(S,Goods,_)),
    assert(price(S,Goods,finished)).

handle(_,_,_) :- true.
```

Therefore, in case the agent receives a message that it was not expecting, this message is discarded and the agent will continue by evaluating the next goal in the `main` one, namely `buy_goods`.

The first clause defining the `handle` predicate is the one that deals with messages containing a term that applies the `price` predicate to two terms. This message informs the buyer about the price at which the sender sells a fruit. The only action taken by the buyer is to cancel the previously stored selling price, if any, of the corresponding fruit and store the price received in this message, by using the `retract` and `assert` predicates.

The second clause, instead, handles the message whose content is a term that applies the `bought` predicate to one term. This message has been sent to inform the buyer that the transaction has been successfully carried out, that is, the buyer has bought the fruit.

The resulting actions taken by the buyer are to decrement the money it owns by the amount of money used to buy the fruit, and to increment the quantity of fruit that it possesses. In this clause we evaluate the goal `price(S,Goods,P)` in order to know the price (`P`) of the fruit bought so we can decrement by that amount the money owned by the agent. We also evaluate the goal `buys(goods(Goods),quantity(Q))` so we know how much kilograms of that fruit the buyer usually buys, and we can update the fruit possessed by the agent accordingly.

The third clause has the purpose of managing the message whose content is made up with the `no_more`) predicate. This message has been sent from a seller agent to inform the buyer that it can no longer sell a fruit, since the quantity it

owns is less than the quantity that this buyer usually buys.

The only action taken by the buyer is to cancel the previously stored selling price, if any, of the corresponding fruit and substitute it with the constant `finished`, thus the agent knows if it can no longer buy that fruit from the corresponding seller agent.

If the received message does not have the expected performative (`INFORM`), then it is discarded and the agent evaluates the next goal in the `main` one; this is possible because the last clause assures that the `handle` predicate never fails.

ask_prices

This activity consists in asking the fruit prices to the seller agents by sending them a message with the performative `REQUEST`. The main `ask_prices` predicate is defined below:

```
ask_prices :- sellers_addresses(Sell),
   goods_list(GList),
   ask_prices(Sell,GList).
```

First of all, we retrieve the list of the JADE addresses of the seller agents; then, we retrieve the list of fruit sold in the market and, finally, we send to each seller a message (one for each fruit sold in the market) asking the price at which it sells the fruit.
To do so, we use the auxiliary `ask_prices` predicate that takes two arguments:

```
ask_prices([Seller|Others],GoodsList) :-
   ask_prices_to_seller(Seller,GoodsList),
   ask_prices(Others,GoodsList).

ask_prices([],_) :- true.

ask_prices_to_seller(Seller,[Goods|Others]) :-
   address_name(Seller,S),asked(Goods,S,no),
   pack(price(Goods),Str),
   send("REQUEST",Str,Seller),
   retract(asked(Goods,S,no)),
   assert(asked(Goods,S,yes)),
   ask_prices_to_seller(Seller,Others).

ask_prices_to_seller(Seller,[Goods|Others]) :-
   address_name(Seller,S),price(S,Goods,X),X \= na,
   ask_prices_to_seller(Seller,Others).
```

```
ask_prices_to_seller(Seller,[Goods|Others]) :-
   address_name(Seller,S),asked(Goods,S,yes),
   ask_prices_to_seller(Seller,Others).

ask_prices_to_seller(_,[]) :- true.
```

The auxiliary `ask_prices_to_seller` predicate is the one that actually sends a message.

One of the first actions carried out by this predicate is to call the `asked` goal in order to know if the agent has already sent, or not, a message asking the price of the fruit. Before calling the `send` predicate we have to call the `pack` predicate (both predicates are defined in the ***TuJadeLibrary***) on the term we want to send and on the string into which the term will be converted, so we can insert that string as content of the message.

Afterwards, we substitute the `asked` fact whose third argument was the `no` constant, with a new one whose third argument is the `yes` constant, in order to know in future that the agent has already sent a request for the price of the fruit (specified in the fact) to the seller (also specified in the fact).

The second clause defining `ask_prices_to_seller` checks whether the price of the given fruit is available or not. If the agent already knows the price then the next fruit in the list of the fruit sold in the market is examined.

On the other hand, if the price is still not present, but it has been requested, the agent cannot do anything regarding that fruit so, while waiting for the agent seller to reply, it checks if it knows the price of the other fruit: this is done by the third clause.

The final clause defining `ask_prices_to_seller` is used to know if the agent has asked the price of all the fruit sold by a seller agent.

buy_goods

This activity consists in buying fruit:

```
buy_goods :- sellers_addresses(SList),
   goods_list(GList),buy_goods(SList,GList).
```

First of all, we retrieve the list of the JADE addresses of the seller agents; then, we retrieve the list of fruit that the buyer intends to buy and, finally, we send for each

fruit, a message asking to buy it to the seller that sells that fruit at the cheapest price.

To do so, we use the auxiliary `buy_goods` predicate that takes two arguments:

```
buy_goods(SList,[Goods|Others]) :- money(M),M > 0,
   buys(goods(Goods),quantity(Q)),
   find_min_price(SList,Goods,30000,nobody,
   HonestSeller),HonestSeller \= nobody,
   buy_from_seller(HonestSeller,Goods,Q),
   buy_goods(SList,Others).

buy_goods(_,_) :- true.
```

At first, of course, we check if the agent has money, otherwise it does nothing and starts again the evaluation of the `main` goal. If the agent has money to spend, though, we consider each fruit that it wishes to buy and decide which seller charges the cheapest price for such fruit by calling the `find_min_price` goal. The auxiliary `find_min_price` predicate is defined as follows:

```
find_min_price([Seller|Others],Goods,Min,_,
   HonestSeller) :- address_name(Seller,S),
   price(S,Goods,Price),Price \= na,
   Price \= finished,Price < Min,
   find_min_price(Others,Goods,Price,Seller,
   HonestSeller).

find_min_price([Seller|Others],Goods,Min,Sel,
   HonestSeller) :- address_name(Seller,S),
   price(S,Goods,Price),Price \= na,
   Price \= finished,Price >= Min,
   find_min_price(Others,Goods,Min,Sel,
   HonestSeller).

find_min_price([Seller|Others],Goods,Min,Sel,
   HonestSeller) :- address_name(Seller,S),
   price(S,Goods,na),
   find_min_price(Others,Goods,Min,Sel,HonestSeller).

find_min_price([Seller|Others],Goods,Min,Sel,
   HonestSeller) :- address_name(Seller,S),
   price(S,Goods,finished),
```

```
    find_min_price(Others,Goods,Min,Sel,HonestSeller).

find_min_price([],_,_,Seller,Seller) :- true.
```

The first clause defining the `find_min_price` checks if the buyer already knows the price at which the given fruit is sold by the seller considered. This check is easily performed by solving the `price` goal. This call also allows the buyer to know if the seller still has the quantity of fruit that it wants to buy.

If the buyer can buy the fruit from the seller, then the price at which the fruit is sold is compared to the minimum price that the buyer has found for that fruit (stated by the third argument of the `find_min_price` goal). Obviously, at the beginning of the search for the cheapest price, we will need to have a very high minimum, this is why in the `buy_goods` goal we call the predicate with the minimum set to `30000`: we will make sure that at least one seller will sell the fruit at a price that is lower than this chosen amount.

Thus, if the price is lower than the current minimum, the buyer continues its search by calling the same goal but substituting the minimum price found until now with the price just considered, and substituting the fourth argument (that indicates the cheapest seller found until now) with the current seller.

If, on the other hand, the price charged by the seller that is currently considered is higher or equal to the current minimum price, then this situation is dealt by the second clause that will make the buyer totally ignore this seller and continue the search by considering the next seller.

The third clause deals with the case in which the buyer does not yet know the price that the current seller charges for the fruit: the buyer totally ignores this seller and continues the search.

If the current seller does not have the quantity of fruit that the buyer needs, then this seller is ignored and the search carries on.

Finally, the last clause is the one that terminates the search since all the sellers have been considered; the last argument of the goal is bound to the cheapest seller found until now, so if it is the `nobody` constant it means that the buyer is not going to buy the fruit (in fact, the first `buy_goods` clause fails).

After having considered a fruit, the buyer considers the next fruit in the list, until it has considered all the fruit it is interested in.
The auxiliary `buy_from_seller` predicate is the one that sends a request to buy a fruit to a seller agent and is defined below:

```
buy_from_seller(Seller,Goods,Quantity) :-
   address_name(Seller,S),price(S,Goods,Price),
   money(M),M > 0,NM is M - Price,NM >= 0,
   pack(buy(Goods,Quantity),Str),
```

```
    send("REQUEST",Str,Seller).

buy_from_seller(_,_,_) :- true.
```

Before sending a message requesting to buy the desired fruit, the agent must check if it has enough money: it subtracts the price of the fruit from the money it has and sends the request to buy only if the result is positive or equal to zero.

Finally, by resolving the `send` predicate, the agent sends the seller agent the message with which it requests to buy a certain quantity of the fruit.
The last clause assures that, if the money is not enough for buying the fruit, the agent will pass to the next fruit belonging to the list. When programming the behaviour of a tuProlog agent, the developer must remember that if a goal in the `main` predicate fails, an error will be displayed in a window that automatically pops-up and the agent is terminated; this is why most of the predicates we have defined never fail.

The tuProlog seller agent

As we have done for the buyers, we describe the theory file of the agent ***seller1***, since the one of agent ***seller2*** only differs in the prices charged for the fruit and the quantities of fruit it owns.

The seller agent only has one activity, that is to respond to messages that it receives from the buyer agents. Its `main` predicate is, therefore, simply defined as follows:

```
main :- handle_msgs.
```

Again, the developer must remember that this clause must appear *at the beginning of the theory file* given in input when loading the seller agent into a JADE platform.

Before defining the main `handle_msgs` predicate, we detail what will be the initial knowledge base of the agent. First of all, we decide how much fruit the agent sells when it carries out a sale of a particular fruit, and the price it charges for that fruit:

```
sells(goods(oranges),price(30),quantity(900)) :-
    true.

sells(goods(apples),price(150),quantity(40)) :-
    true.

sells(goods(kiwi),price(200),quantity(30)) :- true.
```

We adopt the same convention used when describing the buyer agents: we write in blue the data that will probably be different in the theory file of the agent ***seller2***.

Whenever it happens that a buyer tries to buy a fruit and the seller does not have the quantity that the buyer intends to buy, the seller will keep track of this, thus it will be able to immediately refuse to sell if it receives sale requests from that buyer in the future. For this purpose we assert the following facts:

```
available(oranges,buyer1,yes) :- true.
available(oranges,buyer2,yes) :- true.
available(apples,buyer1,yes) :- true.
available(apples,buyer2,yes) :- true.
available(kiwi,buyer1,yes) :- true.
available(kiwi,buyer2,yes) :- true.
```

The constant `yes` in the `available` facts means that the agent can sell the fruit to the buyer agents specified in the facts themselves.

Finally, we also assert the facts below in order to be able to convert the names of the buyer agents into their JADE addresses and vice versa.

```
address_name("buyer1@ai:1099/JADE",buyer1) :- true.
address_name("buyer2@ai:1099/JADE",buyer2) :- true.
```

Now we can define the `handle_msgs` that performs the main activity of this agent:

```
handle_msgs :- receive(Performative,Message,Sender),
   select(Performative,Message,Sender).
```

As we have said when defining the behaviour of the buyer agents, the `receive` predicate, that we have defined in the ***TuJadeLibrary***, bounds the `Performative`, `Message` and `Sender` variables only if the agent as actually received a message, otherwise it leaves them unbound.
The auxiliary `select` predicate is declared by these clauses:

```
select(Performative,Message,Sender) :-
   bound(Performative),
   bound(Message),address_name(Sender,S),
   unpack(Message,TermMsg),
   handle(Performative,TermMsg,Sender).

select(_,_,_) :- true.
```

If the agent has received a message, it first reads the name of the sender: if the sender is not one of the buyer agents then the message is discarded, otherwise

it is read and, before it is handled, the string it contains is converted into the corresponding tuProlog term by calling the `unpack` predicate, defined in the ***TuJadeLibrary***.

If, on the other hand, the sender is one of the buyers, then the agent deals with the message only if it is one of the two types of messages that the seller expects from the buyer agents, otherwise the message is simply discarded.
We declare the `handle` predicate as shown below:

```
handle("REQUEST",price(Goods),Buyer) :-
   bound(Goods),
   sells(goods(Goods),price(Price),quantity(AQ)),
   pack(price(Goods,Price),MsgString),
   send("INFORM",MsgString,Buyer).

handle("REQUEST",buy(Goods,Quantity),Buyer) :-
   bound(Goods),bound(Quantity),
   address_name(Buyer,B),
   available(Goods,B,yes),
   sells(goods(Goods),price(Price),quantity(AQ)),
   AQ >= Quantity,
   retract(sells(goods(Goods),price(Price),
   quantity(AQ))),
   NQ is AQ - Quantity,
   assert(sells(goods(Goods),price(Price),
   quantity(NQ))),
   pack(bought(Goods),MsgString),
   send("INFORM",MsgString,Buyer).

handle("REQUEST",buy(Goods,Quantity),Buyer) :-
   bound(Goods),bound(Quantity),
   address_name(Buyer,B),
   available(Goods,B,yes),
   sells(goods(Goods),price(_),quantity(AQ)),
   AQ < Quantity,retract(available(Goods,B,yes)),
   pack(no_more(Goods),MsgString),
   assert(available(Goods,B,no)),
   pack(bought(Goods),MsgString),
   send("INFORM",MsgString,Buyer).

handle(_,_,_) :- true.
```

The messages that the seller receives and that have a performative different from `REQUEST` are all discarded because are not of the kind that it expects.

The first clause is the one that handles a message sent from a buyer in order to ask the price at which the fruit specified in the content of the message is sold. The only action, in reply, taken by the seller is to send back to the buyer a message, with the `INFORM` performative and the `price` term transformed into a string as content, to inform it of the price at which the fruit is sold.

The second clause, instead, handles a request to buy fruit: the content of the message specifies the fruit and the amount (in kilograms) that the buyer agent is interested in buying.

First, the agent checks if it can sell that fruit to the buyer, by solving the `available` goal. Secondly, if the fruit can be sold, the agent verifies if the quantity it has in stock is greater or equal to the one requested: if it is, the sale is carried out. Before informing the buyer of the successful sale, though, the seller subtracts the quantity just sold from the amount that it had in stock, storing the result as the current amount in stock.

On the other hand, if the agent *knows* that it cannot sell the fruit to this particular buyer, then it just ignores the message.

The third clause is the one that manages a request to buy a quantity of fruit that is greater than the amount currently in stock. In this case the agent sends a message, whose performative is `INFORM` and whose content is the `no_more` term converted into string, to the buyer informing it that the sale cannot be carried out because the fruit in stock is less than the quantity the buyer is interested in buying.

Before sending the message, though, the agent updates its knowledge base: it stores the `available` fact with the constant `no` instead of `yes`, to indicate that it is no longer possible for it to sell to the buyer who sent the message the fruit that it requested.

The last clause has the purpose of forcing the success of the resolution of the goal in the cases not dealt with by the previous clauses.

The Java seller agent

We will now detail the main parts of the content of the file *Seller.java* that defines the Seller class and implements the agent ***seller*** of our MAS:

```
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;

public class Seller extends Agent
```

```
{ private int oranges = 5;
  private int apples = 5;
  private int kiwi = 10;

  private int orangesP = 105;
  private int applesP = 80;
  private int kiwiP = 100;

  private boolean orToB1 = true;
  private boolean orToB2 = true;
  private boolean apToB1 = true;
  private boolean apToB2 = true;
  private boolean kiToB1 = true;
  private boolean kiToB2 = true;
```

The classes imported are used to implement an agent runnable in a JADE platform.
The three integer fields, named with the name of the fruit, are meant to store the initial quantity of fruit that the agent has in stock.
The three integer fields, whose name is formed by the name of the fruit and an ending **P**, are meant to store the price at which the agent sells the fruit.

The boolean fields are used to keep track as to whether a given fruit is (or is not) sellable to the buyers. The first two letters in the name of the field indicate the fruit: `or` stands for oranges, `ap` for apples and `ki` for kiwi. Whereas, the last two letters represent the buyer: `B1` stands for *buyer1*, while `B2` stands for *buyer2*.

If, for instance, the value of the `apToB2` field is `false`, then it means that this agent cannot sell apples to buyer2. At the beginning, the agent can sell all the fruit to all the buyers.

```
protected void setup()
{ SellBehaviour p = new SellBehaviour(this);
  addBehaviour(p);
}
```

The *setup* method is invoked by JADE during the setup phase of the agent: in our case the only action performed is that of adding a custom behaviour, the SellBehaviour, to the task queue of the agent.

```
class SellBehaviour extends CyclicBehaviour
{ ..................
  public void action()
```

```
{ ACLMessage msg;
  msg = myAgent.receive();
  if (msg != null) handleMsg(msg);
}
```

The SellBehaviour is the only task that the agent has to perform and, since it extends the CyclicBehaviour available in JADE, it is repeated throughout the life of the agent.

The *action* method in the SellBehaviour class is invoked by JADE's scheduler when it withdraws such behaviour from the task queue corresponding to the agent. The main task of the seller agent is to receive messages: if the message queue is empty, it remains idle, otherwise it handles the received message by calling the *handleMsg* method that we have developed, passing the message to it as a parameter.

```
private void handleMsg(ACLMessage msg)
{ String sender = msg.getSender().getLocalName();
  if ((sender.equals("buyer1"))||(sender.equals("buyer2")))
  { CODE1 }
  else return;
}
```

The first action carried out by this method is to read the address of the sender agent: if it does not correspond to one of the buyer agents, then it returns immediately, otherwise it executes **CODE1** below:

```
{ String perf = msg.getPerformative();
  String cont = msg.getContent();
  int n = cont.length();
  if (!perf.equals("REQUEST")) return;
  try if (cont.substring(0,5).equals("price"))
  { CODE2 }
  else
  if (cont.substring(0,3).equals("buy"))
  CODE3
```

The performative and content of the message are read. First, the performative is checked: if it is not the REQUEST performative, the method returns. So, if the performative is the one expected, the method proceeds by checking the content: if it is a string that does not start with the word `price` or `buy`, then it returns without doing anything.

On the other hand, if the content begins with the word `price`, **CODE2** below is executed:

```
{ String goods = cont.substring(6,n-1);
  int price = price(goods);
  if (price == -1) return;
  msg.reset();
  msg.setPerformative(ACLMessage.INFORM);
  msg.setSender(myAgent.getAID());
  AID receiver = new AID();
  receiver.setLocalName(sender);
  msg.addReceiver(receiver);
  String content = new String("price("+goods+","+price+")");
  msg.setContent(content);
  myAgent.send(msg);
}
```

If the content of the message is the string `price(`*elem*`)`, with the *elem* string equal to `oranges` or `apples` or `kiwi`, then the agent sends a message to the sender. This message has the INFORM performative and the content "`price(`*elem*`,`*pr*`)`", where *pr* is the number corresponding to the selling price of *elem*.

In the above code, the private *price* method takes a string as an argument and returns an integer. The returned integer is -1 if the string in input is not one of the fruit in the market, otherwise it is the value of the field corresponding to the price of that fruit.

When the content of the message received by the agent begins with the word `buy`, then **CODE3** below is executed:

```
{ int comma = cont.indexOf(',');
  String goods = cont.substring(4,comma);
  if (n < comma+1) return;
  int qtity = Integer.parseInt(cont.substring(comma+1,n-1));
  if (qtity < 0) return;
  if (!available(goods,qtity))
  { if (!reported(goods,sender))
    { msg.reset();
      msg.setPerformative(ACLMessage.INFORM);
      msg.setSender(myAgent.getAID());
      AID receiver = new AID();
      receiver.setLocalName(sender);
      msg.addReceiver(receiver);
      String content = new String("no_more("+goods+")");
      msg.setContent(content);
```

```
        reporting(goods,sender);
        myAgent.send(msg);
    }
```

The first four lines of **CODE3** are for parsing the content of the message and obtaining from it the variables representing the fruit (`goods`) and the quantity (`qtity`) that the buyer wants to buy. If the quantity of fruit that the buyer has specified in the message is not positive, the agent does nothing.
If, on the contrary, it is positive, then the agent checks if it has enough fruit to sell: it calls the *available* method that, given the name of the fruit and the requested quantity, returns `true` only if the agent has enough fruit to sell.

If the seller has less fruit than the quantity requested by the buyer: it calls the *reported* method that, given the name of the fruit and the name of the buyer agent, returns `true` only if the agent believes it can sell that fruit to that buyer.
If the *reported* method returns `true`, it means that this is the first time that the agent realises that it cannot sell the fruit to the buyer. As a result, it not only updates the boolean value of the relative field, but also sends a message to the buyer.
This message informs the buyer that, from now on, the seller will not be able to provide it with the requested fruit.

When the *reported* method returns `false`, the agent does nothing.
When the message received contains the request for an amount of fruit available, then the sale is carried out and this is the code executed:

```
update_qtity(goods,qtity);
msg.reset();
msg.setPerformative(ACLMessage.INFORM);
msg.setSender(myAgent.getAID());
AID receiver = new AID();
receiver.setLocalName(sender);
msg.addReceiver(receiver);
String content = new String("bought("+goods+")");
msg.setContent(content);
myAgent.send(msg);
```

The *update_qtity* method, given the name of the fruit and the number representing the quantity that the buyer wants, updates the field containing the quantity of fruit possessed by the agent.
After the updating, the agent sends the buyer the message informing it that the fruit has been bought.
The manual of JADE contains all the details for developing Java agents.

The final step: execution

Once all the agents have been specified, they can finally be loaded into JADE and the developer can start **executing the obtained prototype**.
This can be done by typing the command:

```
java jade.Boot
   seller1:tuPInJADE.JadeShell42P(seller1.pl)
   seller2:tuPInJADE.JadeShell42P(seller2.pl)
   buyer1:tuPInJADE.JadeShell42P(buyer1.pl)
   buyer2:tuPInJADE.JadeShell42P(buyer2.pl)
   seller:Seller
```

from the directory where the files defining the agents' behaviour have been saved. Note that you should have installed both Java, Jade, and tuProlog, in order to be able to execute the prototype, and you should have changed your classpath system variable as described in the "tuPInJADE-readme" file, in the `DCaseLP\tuPInJADE` directory.

So, to develop tuProlog agents the developer just needs to write their `main` goal, together with their initial knowledge if present, in text files and then pass them as input when loading the JadeShell42P or JadeShell42PGui agents (= **tuProlog agents**) in JADE.

JADE offers more than one way to debug the MAS. For instance, the developer can exploit the *Sniffer Agent* to display the messages exchanged by agents selected by the user, as a sort of sequence diagram. This agent also allows to view the details of a message, like its content, the address of the sender, etc. The developer can choose to save to a file the sequence of messages viewed, and then reload it at a later time.

Another interesting agent useful in debugging the MAS is the *Introspector Agent*. This agent allows to monitor and control the life cycle of a running agent, and also to view the messages that it has sent and received. The Introspector Agent also offers the possibility to monitor the queue of behaviours of an agent, including their execution step-by-step.

The agents described in the previous sections were successfully executed and we have obtained what we expected. The textual output of the simulation consists of a set of messages printed by the agents, in order to show how the exchange of fruit goes on:

```
Agent seller has sold 8 apples to buyer2

Agent seller 1 has sold '2'oranges' to 'buyer1'
```

```
Agent seller has sold 8 apples to buyer2

Agent seller 2 has sold '3' 'apples' to 'buyer1'

Agent seller 2 has sold '3' 'apples' to 'buyer1'

Agent seller has sold 3 kiwi to buyer2

Agent seller 1 has sold '4' 'oranges' to 'buyer2'

Agent seller 2 has sold '8' 'apples' to 'buyer2'

Agent seller has sold 12 kiwi to buyer1

Agent seller 1 has sold '2' 'oranges' to 'buyer1'

Agent seller has sold 3 kiwi to buyer2

Agent seller 2 has sold '3' 'apples' to 'buyer1'

Agent buyer 1: bought '2oranges' from 'seller1';
had '200' cents and now has '170'

....
```

Both the Java and the tuProlog code for these agents can be found in the `DCaseLP\tuPInJADE\Tutorial` directory. If you want to re-use this code, please note that YOU CANNOT USE IT AS IT IS. In fact, you need to edit the tuProlog agents and change the addresses that appear inside them, so that the name of the computer where the JADE platform runs (AI in our code), is equal to the name of your own comupter.

The reader can find more details about the tuProlog agents in DCaseLP, this example and its execution in the Master Thesis of Ivana Gungui (in English) that can be downloaded using this link: `http://www.disi.unige.it/person/MascardiV/Download/Gungui.pdf.gz`.