Supply Chain Multi-Agents System Coursework

Introduction

The smartphone supply chain problem that this coursework is trying to address is that each customer would order several Phablet or a Small Smartphones with a unit price, due date and per day penalty cost to a manufacturer. The manufacturer task is to decide what smartphones order to accept or reject to maximise profit per day. The manufacturer would also order components from supplier one or supplier two. Supplier one sells Screens, RAM's, batteries and storages and delivers these supplies to the manufacturer in the next day after order placed. Supplier Two only sells RAM's and storages with a less price than supplier one but provides these components four days after the order placed. Other constraints are that there is a per day per parts warehouse cost, and the manufacturer can only assemble 50 smartphones per day.

The smartphone supply chain problem is fundamental for many reasons. This problem is a supply chain management problem which plays a considerable part in today's global economy, which produces trillions of dollars in transactions yearly around the globe. The Smartphones supply chains are very dynamic which subject to market fluctuations where customer demands for smartphones decreases or fewer components are available from suppliers, operational contingencies like delays in supply delivery, losses of capacity or quantity of problems and finally, to modification in strategies implemented by customers or suppliers (Sadeh, N.M., Arunachalam, R., Eriksson, J., Finne, N., & Janson, S. (2003)).

A Multi-Agents Systems is perfect for addressing the smartphone supply chain problem as its functionality can enhance the smartphone supply chain performance by automatically evaluating vast options such as deciding what customers offer to accept, number of components to buy which supplier to buy parts from and in what order to assemble and ship smartphones to customers under constraints with aim to maximise profit than what a human manager can do. The proof is that the 2003 supply chain trading agent competition shown that multi-agents systems can provide solutions that can effectively evaluate the massive amount of buying components from suppliers, customer bidding options, assembling and shipping products to customers under constraints (Sadeh, N.M., Arunachalam, R., Eriksson, J., Finne, N., & Janson, S. (2003)).

Model Design

Agent Types

- Customer – This agent type role in the supply chain is to order 1 to multiple Small or Phablet Smartphones with a unit price and specific RAM, Storage, Screen and Battery sizes.
- The Manufacturer – Each day, this agent type role in the supply chain is to receive orders from the customer and decide what order to accept or reject to maximise

profit. The Manufacturer role is also to order components from supplier one or supplier 2, assemble the parts within the warehouse to Smartphones and ship each smartphone to the required customers. Finally, the manufacturer would calculate the profit made on that day and then let the Ticker and Seller agent know that the manufacturer finished for the day.

- The Ticker Agent – This agent type would send a message to customers, manufacturer, supplier one and supplier two that a new day has arrived and then increment the day. When each agent sent a message "Done" to the Ticker Agent, the agent would do the previous tasks again until the day is equal to 100.
- Supplier One – This agent type receive purchases request from the Manufacturer to buy Screen, Storage, RAM or battery. The agent would then deliver each purchased components to the warehouse.
- Supplier Two - This agent type receives purchases request from the Manufacturer to buy Storage or RAM. The agent would then deliver each purchased components to the warehouse.

Ontology Diagram

As shown in Appendix one, the concepts are Smartphone, Small Smartphone, Phablet Smartphone, Component, RAM, Screen, Battery and Storage. Predicates are Received and Delivered. The Small Smartphone, and Phablet Smartphone inherits from the Smartphone concept. The RAM, Screen, Battery and Storage inherits from the Component concept. The actions are Order, Assemble, Purchase and Deliver. The order action is used by customers to order smartphones from the manufacturer. The ship action is handled by the manufacturer to ship assembled smartphones to each customer. The purchase action is used by the manufacturer to purchase components from Supplier one or Supplier two. The deliver action is handled by supplier one or supplier two to deliver ordered parts to the warehouse. Predicates are Received and Delivered. Received predicate would allow a Supplier to check that the manufacturer has accepted a component or not. The Delivered predicate would enable a manufacturer to verify that a customer has received the ordered smartphones.

Sequence Diagram

Appendix two represent a sequence diagram that shows the communication protocol between the agents. The Ticker Agent sends an Inform message to all agents with content of ("new day") to let all agents know that a new day has started. Each customer agent sends a REQUEST message with an Order action to the Manufacturer to order smartphones. The Manufacturer would reply with ACCEPT_PROPOSAL ("Accept") or REJECT_PROPOSAL ("Decline") message to the customer to let the customer know whether their order is accepted or rejected. The manufacturer would send a REQUEST with a Purchase Action to supplier one and supplier two to purchase an appropriate number of components from each supplier. Both Supplier one and Supplier Two would reply with an ACCEPT_PROPOSAL ("Accept") message to the manufacturer to let the manufacturer know that their request is accepted. Both suppliers would send a REQUEST message with a Deliver action to the Manufacturer to deliver the ordered components to the warehouse. The manufacturer would reply to both suppliers with ACCEPT_REQUEST ("Success") message to let both

suppliers acknowledge that their request is accepted. The manufacturer would send a REQUEST with a ship action to the customers to ship the assembled smartphones to them. The customer would send an INFORM ("done") message to the Ticker agent so the Ticker Agent would acknowledge that the customer finished for the day. The manufacturer would send an INFORM("done") message to the Ticker Agent, Supplier One and Supplier Two agents for these agents to know that the manufacturer finished for the day. The Supplier One and Supplier Two would send INFORM ("done") to the Ticker Agent so the Ticker Agent can acknowledge that it is time to start a new day.

Model Implementation

All of the communication protocols within the behaviours are intended to emulate the communication protocols within the sequence diagram shown in Appendix 2.

The OffersServer behaviour is a sequence behaviour of the Manufacturer Agent would receive a REQUEST message with an Order action from each customer and extract the message and decide what order to accept or reject. If order accepted, the manufacturer would reply with an ACCEPT_PROPOSAL ("Accept") message to the Customer. Otherwise, the manufacturer would send a REJECT_PROPOSAL ("Decline") message to the customer. This behaviour would go on sequentially until all customers have ordered least one smartphone. This behaviour is shown in Appendix 3.2.

The OrdersServerOne behaviour is a OneShotBehaviour of the Manufacturer agent that loops through the current day accepted ordered Smartphones, and within the loop, the behaviour transmits a REQUEST message with a Purchase Action to Supplier One to order the appropriate Screen and Battery with appropriate sizes and calculates the components price and sends the price to SupplierOne as message content. This behaviour is shown in Appendix 3.3.

The OrdersServerTwo behaviour is a OneShotBehaviour of the Manufacturer Agent that loops through the current day accepted ordered Smartphones, and within the loop, the behaviour transmits a REQUEST message with a Purchase action to Supplier Two to order the appropriate Storage and RAM with appropriate sizes and calculates the components price and sends the price to Supplier Two as message content. This behaviour is shown in Appendix 3.4.

The recieveComponentsOne behaviour is a sequential behaviour of the Manufacturer agent that receives REQUEST message with a Deliver Action from supplier one sequentially until the number of replies is equal to the number of components ordered from Supplier or the number of parts ordered is 0. Each message received, the behaviour would extract the message and add the received component to an array list. The Manufacturer would then reply to the message with an ACCEPT_PROPOSAL ("Success") message. This behaviour is shown in Appendix 3.7.

The recieveComponentsTwo behaviour is a sequential behaviour of the Manufacturer agent that receives REQUEST messages with Deliver action from supplier two sequentially until the number of replies is equal to the number of components ordered from Supplier two, or the

number of parts ordered is 0. Each message received, the behaviour would extract the message and add the received component to an array list. The Manufacturer would then reply to the message with a ACCEPT_PROPOSAL ("Success") message. This behaviour is shown in Appendix 3.8.

The assembleandship behaviour is a OneShotBehaviour of the Manufacturer agent which uses the appropriate components from the warehouse to assemble smartphones. Each constructed smartphone delivered to the relevant Customer through a REQUEST message with a Ship Action. Each time the manufacturer transmits smartphones to a customer, a profit would increase by (number of smartphones * unit price). The used components removed from the warehouse. This behaviour is shown in Appendix 3.9. From this Appendix, you can notice I have implemented the constraint where the manufacturer can only assemble and ship upto 50 smartphones in one day by making sure today number of assembled smartphones + ordered smartphones quantity <= 50 before assembling the smartphones. Another constraint is shown that number of smartphones able to be assembled if it is less than or equal to number of components within the warehouse by making sure ordered smartphones quantity is equal to each appropriate warehouse component type quantity. Another constraint you will also notice that the behaviour will calculate the penalties for late delivery by incrementing total penalty cost by (penalty * (wait time - dueDate)) * quantity.

EndDay Behaviour of the Manufacturer Agent is a OneShotBehaviour that will calculate profit, the per day per components cost, reduce delivery time of each non-delivered part by one and if delivery time is 0, component added to warehouse. The warehouse cost is a constraint which is calculated by looping through the components in the warehouse and adding each component quantity * 5 to the total warehouse cost. The EndDay behaviour would decrease total profit by daily warehouse cost, components cost and daily per day penalty cost. Decreasing total profit by these values is part of the constraint of calculating the total profit correctly. The total profit would then printed on the console. The Behaviour would finally send INFORM ("done") messages to SupplierOne, SupplierTwo and the TickerAgent to let these agents know that the manufacturer finished for the day. This behaviour is shown in Appendix 3.10.

The OrderSmartphones behaviour of the Customer Agent would send a REQUEST message with an Order action to the Manufacturer to buy a random number of smartphones ranging from 1 to 50 which can be a Phablet or Small Smartphone with specific sizes of the battery, storage, screen and RAM components. This behaviour is shown in Appendix 3.1. In Appendix 3.1, you can notice that I have implemented the constraint where the small Smartphone must have 5" screen and 2000mAh battery while a phablet Smartphone must have a 7" screen and 30000mAh battery by hardcoding these values. You can also notice I have implemented that each smartphone must have one RAM and storage size.

EndDay behaviour of the Customer Agent is a OneShotBehaviour that will send an INFORM ("done") message to the TickerAgent to let the TickerAgent acknowledge that a customer finished for the day. This behaviour is shown in Appendix 3.14.

The DeliversServer behaviour is a CyclicBehaviour of the SupplierOne Agent that receives a REQUEST message with a Purchase Action, extract the message and add the ordered screen and battery within the Purchase Action to an Array List. The behaviour would then reply with an ACCEPT_REQUEST ("Accept") to the Manufacturer. It would then send a REQUEST message with a Deliver action to the Manufacturer to send each ordered screen and battery to the manufacturer through a for loop. This behaviour is shown in Appendix 3.5. You can also notice from the appendix that I have made sure the Supplier One has to set the delivery time of each component to one by 'component.setDeliveryTime(1)' so the component is delivered to the warehouse one day after order.

The DeliversServer2 behaviour Is a CyclicBehaviour of the SupplierTwo Agent that receives a REQUEST message with a Purchase Action, extract the message and add the ordered RAM and storage within the Purchase Action to an Array List. The behaviour would then reply with a ACCEPT_REQUEST ("Accept") to the Manufacturer. It would then send a REQUEST message with a Deliver Action to the Manufacturer to transmit each ordered RAM and storage to the manufacturer through a loop. This behaviour is shown in Appendix 3.6. You can also notice from the appendix that I have made sure the Supplier Two has to set the delivery time to four of each component by 'component.setDeliveryTime(4)' so each component delivered to the warehouse four days after order.

The EndDayListener behaviour of the Supplier One and Supplier Two is a Cyclic Behaviour that would wait for an INFORM ("done") message from the manufacturer. When a message received, INFORM("done") message transmitted to the TickerAgent and all cyclic behaviours including this one deleted. This behaviour shown in Appendix 3.17.

SynchAgentsBehaviour is Sequence Behaviour of the Ticker Agent that would send an INFORM("new day") message to each agent to acknowledge a new day have begun; the day incremented. This behaviour would keep going on sequentially until the day is equal to 100. This behaviour is shown in Appendix 3.16.

The TickerWaiter Behaviour of Manufactuer, Customer, Supplier One and Supplier Two would wait for a INFORM ("new day") message from the Ticker Agent and then the daily behaviours would be initiated. These behaviours can be shown in Appendix 3.11, 3.12, 3.14 and 3.15.

Manufacturer Control Strategy

Each day, the manufacturer will check each customer offer; if the (due date is >= 4 or per day penalty for later delivery is equal to 0), and the todayQuantity + number of smartphones requested by customer is <= 50, the offer accepted. Otherwise, the proposal rejected. The strategy would help the manufacturer to maximise profit as the manufacturer can buy components from supplier 2 which offers prices lower compared to supplier 1 without worrying about per day penalty for late delivery so, therefore, decrease supplies cost, total penalty cost and maximise profit while at the same time, reduce the warehouse storage cost as the manufacturer only buy 50 or less component's per day which would minimize storage overflow as a manufacturer can only assemble up to 50 smartphones per day. This strategy aspired by part of the supply chain trading agent competition where the

agent acknowledges submitting high price bids might increase profit but reduces the number of customers accepting a request and at the same time submitting low bids causes amount of customers accepting submissions more than what the agent can satisfy.**(**Sadeh, N.M., Arunachalam, R., Eriksson, J., Finne, N., & Janson, S. (2003)).

The number of each component that the manufacturer would buy from each supplier in a day is the same as the number of smartphones that ordered in that day. This strategy would maximise the manufacturer profit as it would decrease the number of unnecessary components in the warehouse, which would reduce the warehouse per-parts per-day cost so, therefore, maximise the total profit of each day. This strategy inspired by the supply chain trading competition where trading agents could desire to stock up components to offer fewer delivery times to potential customers. Also, trading agents might want to prevent future supply shortages, whether due to production delays at one or more suppliers (Sadeh, N.M., Arunachalam, R., Eriksson, J., Finne, N., & Janson, S. (2003)).

The fact that the manufacturer only accepts orders from customers where the due day must be either least four days or the per day penalty must be 0 pounds; the manufacturer does not need to worry about the delivery time of supplier two. The manufacturer would order RAMs and storages from supplier two and order other components from supplier one. This strategy would help the manufacturer to maximise profit as supplier two sell RAMs and storages in a price which is less than the amount that the supplier 1 sells them so that the components cost would reduce and maximise manufacturer profit per day. This strategy influenced by the supply chain trading competition where the agent forecast future customer and supply market conditions instead of just looking at their own constrain such as the number of smartphones it can assemble per day to maximise profit as much as possible. (Sadeh, N.M., Arunachalam, R., Eriksson, J., Finne, N., & Janson, S. (2003)).

Each day, the manufacturer assemble and ship orders from the earliest accepted order until the number of smartphones assembled is equal or nearly to 50. This strategy would help the manufacturer to prevent any delays in supply deliveries and to be more flexible for future customers orders. Therefore, this strategy would help the manufacturer to maximise profit as avoiding any delays in shipping smartphones to customers indicates that the orders would be delivered to customers soon as possible and so reduce any likely per day penalties and maximise the amount of profit per day. This strategy influenced by the supply chain trading competition where agent could assemble activities earlier to prevent any potential delays within supplies deliveries and at the same time, be more flexible to future customer requests (Sadeh, N.M., Arunachalam, R., Eriksson, J., Finne, N., & Janson, S. (2003)).

Experimentation

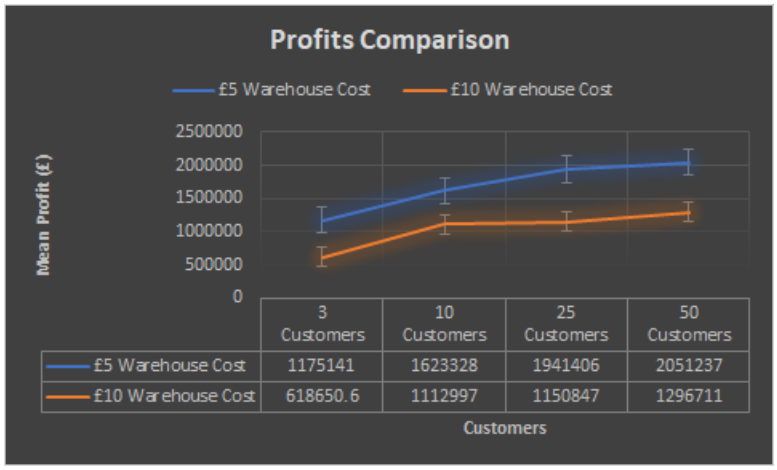
I am planning to evaluate the performance of two features of the manufacturer agent control strategy. I am planning to evaluate the performance through eight experiments. At each trial, the number of customers or per day per component warehouse cost parameters would vary. At each test, the number of customers would range from 3, 10, 25 and 50 and the per component per day warehouse cost would be either £5 or £10.

One reason that I like to vary these parameters as I like to evaluate the part of the manufacturer control strategy where it accepts customers offers if (due date is >= 4 or per day penalty for later delivery is equal to 0), and the todayQuantity + number of smartphones requested by customer is <= 50 would still manage to maximise profit when the number of customers and the per day per component warehouse storage cost increases as I believe that more offers would be accepted when there are more customers while total warehouse costs would not increase as much so the profit be maximised.

Another reason I like to vary the number of customers and the per day per components warehouse cost as I have intention to evaluate how efficient the part of the Manufacturer Control Strategy where the number of each component that the manufacturer would buy from each supplier in a day is the same as the number of smartphones that ordered in that day would be able to decrease the number of unnecessary components in the warehouse to reduce the daily warehouse per-parts per-day cost so, therefore, maximise the total profit of each day. My experimental hypothesis is that when the number of customers increases, the profit would increase and when the per day per component warehouse cost increases, the profit would decrease but not by a large margin.

Each experiment, I would run the program 30 times, and at each run, I would record the final profit on the 100th day. When each test finished, the average profit (£) and the standard error calculated. Finally, the mean and standard error of each experiment is plotted on a line graph. The x-axis would represent the Mean Profit (£), the y-axis would represent the number of customers, error bars would represent the Standard Error of the profits, the legends would be £5 Warehouse Cost and £10 Warehouse Cost.

Experiment Results



Conclusion

The design can be expanded to handle a more realistic supply-chain model by adding multiple manufacturers which are competing. Customers would order from Smartphones from manufacturers that have decent reputation such as delivering smartphones on time. There will also be various suppliers that sell similar components with different delivery times and parts cost. Per day, the Manufacturer would choose which suppliers to buy parts from where decision-based on maximising profit as much as possible. I would also modify

the Phablet and Small Smartphones ontologies to make sure that the phablet Smartphones has a 7” Screen and 3000mAh battery and to make sure that Small Smartphones has a 5” Screen and 2000mAh battery instead of hardcoding these values. This way of implementing the constraints are significant if engineer is using these ontologies in a different supply chain multi-agents' system. Customers would use a min-max algorithm to decide the most optimal smartphones attributes(unit price, quantity, Storage size and RAM size) instead of randomly choosing these values which is more realistic in a real-life supply chain.

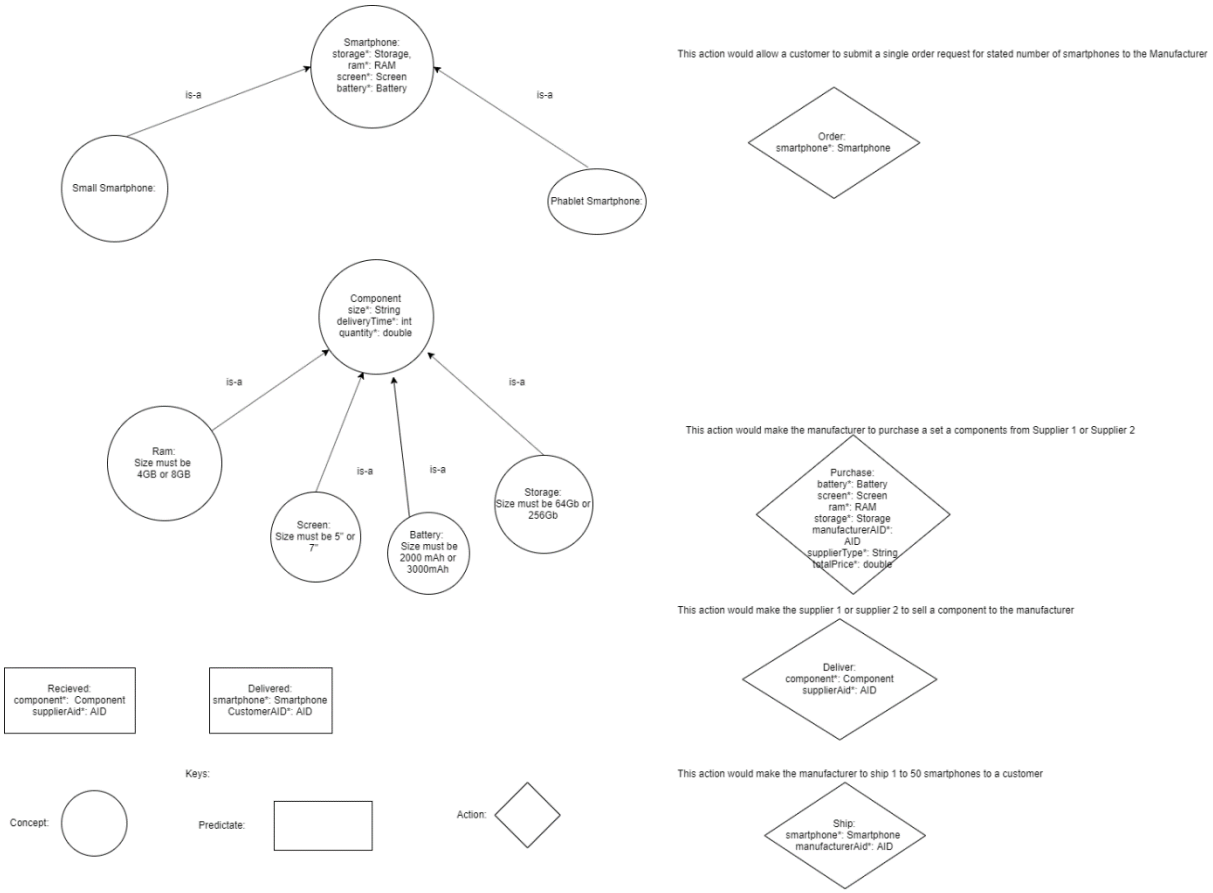
The experimentation stage shows that the profit of the manufacturer increases when the number of customers increases but decreases when the warehouse cost rises. Therefore, I believe that the Manufacturer Control Strategy can be modified to increase profit gains much more when number of customers increases and profit decreases much less when the warehouse cost rises. I would change how the manufacturer decide what customers' orders to accept or reject. The manufacturer would still check that the (todayQuantity + number of smartphones ordered) <= 50 and the due date is >= 4 but would also check whether ordered smartphone is phablet or small. If the smartphone is low, the unit price must be >= 250, and if the smartphone is phablet, the unit price must be >= 300. I believe this strategy would maximise the manufacturer profits much more than the manufacturer profits within the experimentation stage as the higher the unit price, the higher the profit especially when the number of customers increases. Alternatively, the manufacturer can predict the number of smartphones and type of components that potential customers would order by looking at statistics of previous customers' orders so the Manufacturer can order number of parts from appropriate suppliers beforehand to prevent warehouse storage overflow in order to reduce the per day per parts warehouse cost and gain much more profit compared to the earnings in the experimentation stage as most parts would be used to assemble and ship smartphones straight after it is delivered.

References

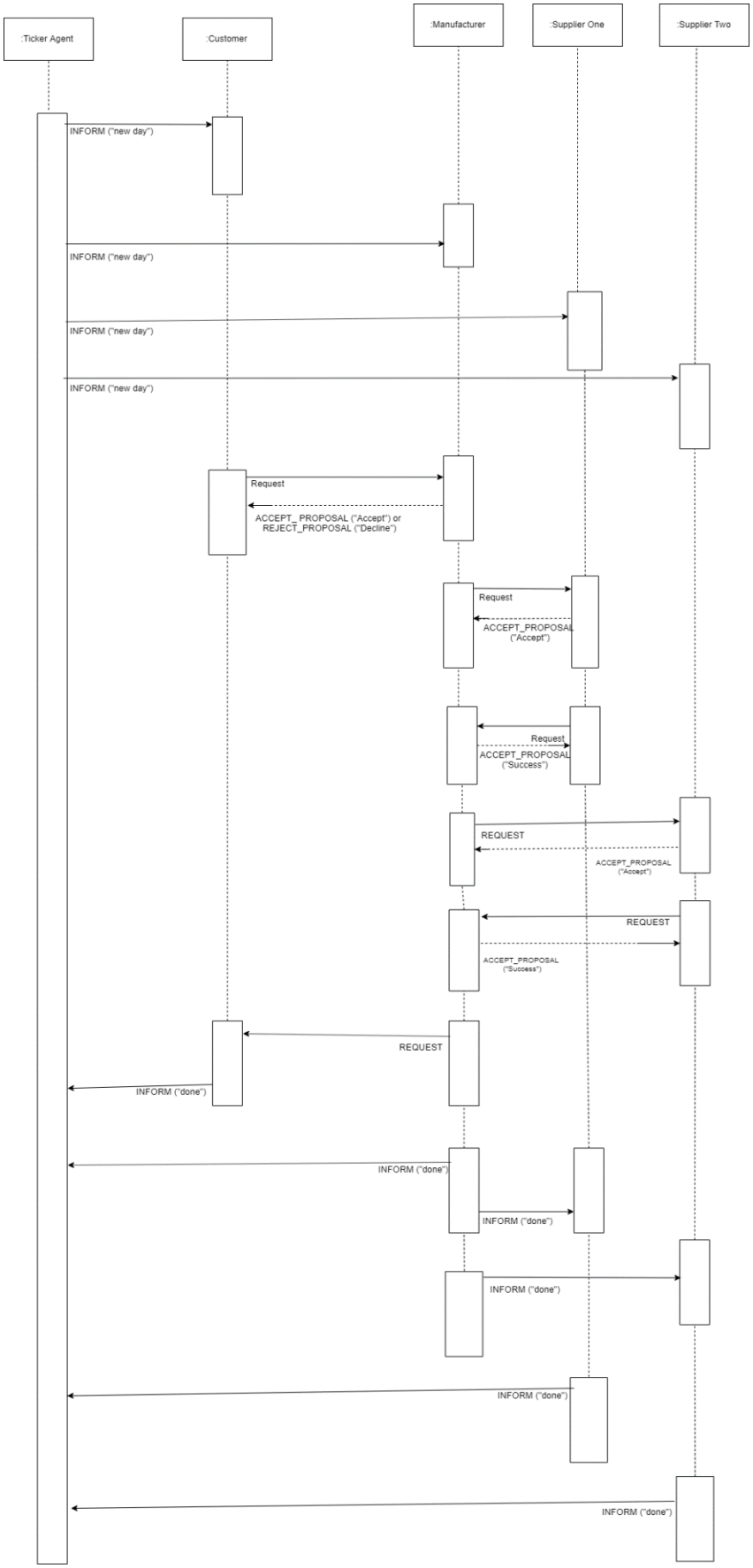
Sadeh, N.M., Arunachalam, R., Eriksson, J., Finne, N., & Janson, S. (2003). TAC-03 - A Supply-Chain Trading Competition. AI Magazine, 24, 92-94.

Appendixes

Appendix 1 – Ontology Diagram



Appendix 2 – Sequence Diagram



Appendix 3.1 – OrderSmartphones Behaviour Source Code

```
/*
	 * This Behaviour would send an Order Action request to the Manufacturer to
buy a
	 * random amount of smartphones ranging from 1 to 50 which can be a
	 * Phablet or Small Smartphone with specific
	 * sizes of the battery, storage, screen and RAM components
	 */
	private class OrderSmartphones extends OneShotBehaviour{

		public OrderSmartphones(Agent a) {
			super(a);
		}

		@Override
		public void action() {
			// TODO Auto-generated method stub
			double randomValue = (Math.random() * 1);
			if(randomValue < 0.5) {
				ACLMessage enquiry = new
ACLMessage(ACLMessage.REQUEST);
				enquiry.addReceiver(manufacturers.get(0));
				enquiry.setLanguage(codec.getName());
				enquiry.setOntology(ontology.getName());
				Small_Smartphone ss = new Small_Smartphone();
				Screen screen = new Screen();
				//Making sure screen size is 5 inch
				screen.setSize("5inch");
				ss.setScreen(screen);
				Battery battery = new Battery();
				//Making sure the battery size is 2000mAh
				battery.setSize("2000mAh");
				ss.setBattery(battery);
				RAM ram = new RAM();
				Storage storage = new Storage();
				double randomValue2 = (Math.random() * 1);
				//Making sure the ram and storage has only one value
				if(randomValue2 < 0.5) {
					ram.setSize("4GB");
					storage.setSize("64GB");
				}
				else {
					ram.setSize("8GB");
					storage.setSize("256GB");
				}
				ss.setRam(ram);
				ss.setStorage(storage);
				Random random1 = new Random();
				double numberOfSmartphones =
Math.floor(random1.nextInt((50 - 1) + 1) + 1);
				ss.setQuantity(numberOfSmartphones);
				Random random2 = new Random();
				ss.setPrice(Math.floor(random2.nextInt((500 - 100) +
100) + 100));
				Random random3 = new Random();
				ss.setDueDate(Math.floor((random3.nextInt((10 - 1) + 1)
+ 1)));
				Random random4 = new Random();
```

```
                        ss.setPerDayPenality(numberOfSmartphones *
Math.floor(random4.nextInt((50 - 1) + 1) + 1));
                    ss.setCustomerAID(myAgent.getAID());
                        Order order = new Order();
                        ss.setWaitingTime(0);
                        order.setSmartphone(ss);
                        Action request = new Action();
                        request.setAction(order);
                        request.setActor(manufacturers.get(0));
                        try {
                         getContentManager().fillContent(enquiry, request);
//send the wrapper object
                         send(enquiry);
                        }
                        catch (CodecException ce) {
                         ce.printStackTrace();
                        }
                        catch (OntologyException oe) {
                         oe.printStackTrace();
                        }
                    }
                    else {
                        ACLMessage enquiry = new
ACLMessage(ACLMessage.REQUEST);
                        enquiry.addReceiver(manufacturers.get(0));

                        enquiry.setLanguage(codec.getName());
                        enquiry.setOntology(ontology.getName());
                        Phablet_Smartphone ps = new Phablet_Smartphone();
                        Screen screen = new Screen();
                        //Making sure the screen size is 7 inch
                        screen.setSize("7inch");
                        ps.setScreen(screen);
                        Battery battery = new Battery();
                        //Making sure the battery size is 3000mAh
                        battery.setSize("3000mAh");
                        ps.setBattery(battery);
                        RAM ram = new RAM();
                        Storage storage = new Storage();
                        double randomValue2 = (Math.random() * 1);
                        //Making sure the ram and storage both has one value
                        if(randomValue2 < 0.5) {
                               ram.setSize("4GB");
                               storage.setSize("64GB");
                        }
                        else {
                               ram.setSize("8GB");
                               storage.setSize("256GB");
                        }
                        ps.setRam(ram);
                        ps.setStorage(storage);
                        Random random1 = new Random();
                        double numberOfSmartphones =
Math.floor(random1.nextInt((50 - 1) + 1) + 1);
                        ps.setQuantity(numberOfSmartphones);
                        Random random2 = new Random();
                        ps.setPrice(Math.floor(random2.nextInt((500 - 100) +
100) + 100));
                        Random random3 = new Random();
```

```
                        ps.setDueDate(Math.floor((random3.nextInt((10 - 1) + 1)
+ 1)));
                        Random random4 = new Random();
                        ps.setPerDayPenality(numberOfSmartphones *
Math.floor(random4.nextInt((50 - 1) + 1) + 1));
                        ps.setCustomerAID(myAgent.getAID());
                        ps.setWaitingTime(0);
                        Order order = new Order();
                        order.setSmartphone(ps);
                        Action request = new Action();
                        request.setAction(order);
                        request.setActor(manufacturers.get(0));
                        try {
                              getContentManager().fillContent(enquiry,
request); //send the wrapper object
                              send(enquiry);
                        }
                        catch (CodecException ce) {
                              ce.printStackTrace();
                        }
                        catch (OntologyException oe) {
                              oe.printStackTrace();
                        }
                  }
            }
      }
```

Appendix 3.2 – OffersServer Behaviour

```
    /*
     *The OffersServer behaviour within the Manufacturer Agent is a Sequence
Behaviour which would receive an Order Action request from each
     *customer and extract the message and check whether or not that the
(todayQuantity + Customer smartphones order quantity <= 50) and
     *the (Due Date is >= 4 or per day penalty is equal to 0). If the condition is
true, the manufacturer would add the smartphone to an ArrayList
     * and reply with an ACCEPT_PROPOSAL ("Accept") message to the Customer.
Otherwise, the manufacturer would send a REJECT_PROPOSAL ("Decline")
     * message to the customer. This behaviour would keep recieving messages from
customers until the number of replies is equal to numbers
     * of customers
     */
      public class OffersServer extends Behaviour{
            int numOfReplies = 0;

            public OffersServer(Agent a) {
                  super(a);
            }
            @Override
            public void action() {
                  MessageTemplate mt =
MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
                  ACLMessage msg = myAgent.receive(mt);
                  if(msg != null) {
```

```
                        numOfReplies++;
                        ACLMessage reply = msg.createReply();
                        try {
                                ContentElement ce = null;
                                // Let JADE convert from String to Java objects
                                // Output will be a ContentElement
                                ce = getContentManager().extractContent(msg);
                                if (ce instanceof Action) {
                                        Concept action = ((Action)ce).getAction();
                                        if(action instanceof Order) {
                                                Order order = (Order) action;
                                                Smartphone smartphone =
order.getSmartphone();
                                                if((quantity +
smartphone.getQuantity()) <= 50 && (smartphone.getDueDate() >= 4 ||
smartphone.getPerDayPenality() == 0) ){

        todaySmartphones.add(smartphone);
                                                        smartphones.add(smartphone);

        reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
                                                        reply.setContent("Accept");
                                                        quantity +=
smartphone.getQuantity();

                                                }
                                                else {

        reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
                                                        reply.setContent("Decline!");
                                                }
                                                myAgent.send(reply);
                                        }
                                }
                        }
                        catch (CodecException ce) {
                                ce.printStackTrace();
                        }
                        catch (OntologyException oe) {
                                oe.printStackTrace();
                        }
                }
                else {
                        block();
                }
        }

        @Override
        public boolean done() {
                return (numOfReplies == customers.size());
        }
}
```

Appendix 3.3 – OrdersServerOne Behaviour

```
/*
        * The OrdersServerOne is a OneShotBehaviour that loops through the current
day accepted ordered Smartphones,
```

```
        * and within the loop, the behaviour transmits a Purchase Action Request
to Supplier One to order the appropriate
        * Screen and Battery with appropriate sizes and calculates the components
price and sends the price to SupplierOne as message content.
        */
       public class OrdersServerOne extends OneShotBehaviour {
              public OrdersServerOne(Agent a) {
                     super(a);
              }
              @Override
              public void action() {
                     for(Smartphone smartphone : todaySmartphones) {
                           Screen screen = smartphone.getScreen();
                           Battery battery = smartphone.getBattery();
                           battery.setQuantity(smartphone.getQuantity());
                           screen.setQuantity(smartphone.getQuantity());
                           numberOfComponentsOrdered += 2;
                           ACLMessage purchaseMsg = new
ACLMessage(ACLMessage.REQUEST);
                           for(AID supplier : suppliers) {
                                  purchaseMsg.addReceiver(supplier);
                           }
                           purchaseMsg.setLanguage(codec.getName());
                           purchaseMsg.setOntology(ontology.getName());
                           Purchase purchase = new Purchase();
                           purchase.setManufacturerAID(myAgent.getAID());
                           purchase.setBattery(battery);
                           purchase.setScreen(screen);
                           purchase.setSupplierType("Supplier1");
                           double screenPrice = 0;
                           double batteryPrice = 0;
                           double totalPrice = 0;
                           if(screen.getSize().equals("5inch")) {
                                  screenPrice = 100;
                           }
                           else {
                                  screenPrice = 150;
                           }
                           if(battery.getSize().equals("2000mAh")) {
                                  batteryPrice = 70;
                           }
                           else {
                                  batteryPrice = 100;
                           }
                           totalPrice = (screenPrice + batteryPrice) *
smartphone.getQuantity();
                           totalComponentsCost = totalComponentsCost - totalPrice;
                           purchase.setTotalPrice(totalPrice);
                           Action request = new Action();
                           request.setAction(purchase);
                           for(AID supplier : suppliers) {
                                  request.setActor(supplier);
                           }
                           try {
                                  getContentManager().fillContent(purchaseMsg,
request);
                                  send(purchaseMsg);
                           }
                           catch(CodecException ce) {
```

```
                        ce.printStackTrace();
                    }
                    catch(OntologyException oe) {
                        oe.printStackTrace();
                    }
                }
            }
        }
```

Appendix 3.4 – OrdersServerTwo Behaviour

```
/*
        * The OrdersServerTwo behaviour is a OneShotBehaviour that loops through
the current day accepted ordered Smartphones,
        * and within the loop, the behaviour transmits a Purchase Action Request
to Supplier Two to order the appropriate Storage
        * and RAM with appropriate sizes and calculates the components price and
sends the price to Supplier Two as message content.
        */
       public class OrdersServerTwo extends OneShotBehaviour {
            public OrdersServerTwo(Agent a) {
                super(a);
            }
            @Override
            public void action() {
                for(Smartphone smartphone : todaySmartphones) {
                    Storage storage = smartphone.getStorage();
                    RAM ram = smartphone.getRam();
                    ram.setQuantity(smartphone.getQuantity());
                    storage.setQuantity(smartphone.getQuantity());
                    numberOfComponentsOrdered2 += 2;

                    ACLMessage purchaseMsg = new
ACLMessage(ACLMessage.REQUEST);
                    for(AID supplier : suppliersTwo) {
                        purchaseMsg.addReceiver(supplier);
                    }
                    purchaseMsg.setLanguage(codec.getName());
                    purchaseMsg.setOntology(ontology.getName());
                    Purchase purchase = new Purchase();
                    purchase.setManufacturerAID(myAgent.getAID());
                    purchase.setStorage(storage);
                    purchase.setRam(ram);
                    purchase.setSupplierType("Supplier2");
                    int ramPrice = 0;
                    int storagePrice = 0;
                    double totalPrice = 0;
                    if(storage.getSize().equals("64GB")) {
                        storagePrice = 15;
                    }
                    else {
                        storagePrice = 40;
                    }
                    if(ram.getSize().equals("4GB")) {
                        ramPrice = 20;
                    }
                    else {
                        ramPrice = 35;
```

```
                        }
                        totalPrice = (storagePrice + ramPrice)*
smartphone.getQuantity();
                        totalComponentsCost = totalComponentsCost - totalPrice;
                        purchase.setTotalPrice(totalPrice);
                        Action request = new Action();
                        request.setAction(purchase);
                        for(AID supplier : suppliersTwo) {
                              request.setActor(supplier);
                        }
                        try {
                              getContentManager().fillContent(purchaseMsg,
request);
                              send(purchaseMsg);
                        }
                        catch(CodecException ce) {
                              ce.printStackTrace();
                        }
                        catch(OntologyException oe) {
                              oe.printStackTrace();
                        }
                  }


            }
      }
```

Appendix 3.5 – DeliverServer Behaviour of Supplier One Agent

```
/*
       * The DeliversServer behaviour Is a CyclicBehaviour that receives a
Purchase Request message,
       * extract the message and add the ordered screen and battery within the
purchase Request message to a Array List.
       * The behaviour would then reply with a ACCEPT_REQUEST (“Accept”) to the
Manufacturer. It would then send a Deliver Action Request message to the
       *  Manufacturer in order to send each ordered screen and battery to the
manufacturer through a for loop.
       */
      public class DeliverServer extends CyclicBehaviour{
            public DeliverServer(Agent a) {
                  super(a);
            }
            @Override
            public void action() {
                  // TODO Auto-generated method stub
                  MessageTemplate mt =
MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
                  ACLMessage msg = myAgent.receive(mt);
                  if(msg != null) {
                        ACLMessage reply = msg.createReply();
                        try {
                              ContentElement ce = null;
                              ce = getContentManager().extractContent(msg);
                              if(ce instanceof Action) {
                                    Concept action = ((Action)ce).getAction();
                                    if(action instanceof Purchase) {
```

```
                                                Purchase purchase = (Purchase)
action;
                                                Screen screen =
purchase.getScreen();
                                                Battery battery =
purchase.getBattery();
                                                componentsOrders.add(screen);
                                                componentsOrders.add(battery);

        reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
                                                reply.setContent("Accept");
                                                myAgent.send(reply);
                                                DFAgentDescription buyerTemplate =
new DFAgentDescription();
                                                ServiceDescription sd = new
ServiceDescription();
                                                sd.setType("Manufacturer");
                                                buyerTemplate.addServices(sd);
                                                try{
                                                        manufacturers.clear();
                                                        DFAgentDescription[]
agentsType1  = DFService.search(myAgent,buyerTemplate);
                                                        for(int i=0;
i<agentsType1.length; i++){

        manufacturers.add(agentsType1[i].getName()); // this is the AID
                                                        }
                                                }
                                                catch(FIPAException e) {
                                                        e.printStackTrace();
                                                }
                                                for(Component component :
componentsOrders) {

                                                        ACLMessage msg2 = new
ACLMessage(ACLMessage.REQUEST);

        msg2.setConversationId("Delivery");
                                                        for(AID manufacturer :
manufacturers) {

        msg2.addReceiver(manufacturer);


                                                        }

        msg2.setLanguage(codec.getName());

        msg2.setOntology(ontology.getName());
                                                        Deliver deliver = new
Deliver();

        deliver.setSupplierAID(myAgent.getAID());
                                                        //Making sure the delivery
time is 1
                                                        component.setDeliveryTime(1);

        deliver.setComponent(component);
                                                        Action request = new
Action();
```

```
                                                request.setAction(deliver);
                                                for(AID manufacturer :
manufacturers) {

        request.setActor(manufacturer);

                                                }
                                                try {

        getContentManager().fillContent(msg2, request);
                                                        send(msg2);
                                                }
                                                catch(CodecException ce2) {
                                                        ce2.printStackTrace();
                                                }
                                                catch(OntologyException oe) {
                                                        oe.printStackTrace();
                                                }

                                                }
                                        }
                                }
                        }catch (CodecException ce) {
                                ce.printStackTrace();
                        }
                        catch (OntologyException oe) {
                                oe.printStackTrace();
                        }
                }

                else {
                        block();
                }
        }
}
```

Appendix 3.6 – Deliver Server 2 Behaviour

```
/*
        * This behaviour receives a Purchase Request message,
        * extract the message and add the ordered RAM and storage within the
purchase Request message to a Array List.
        * The behaviour would then reply with a ACCEPT_REQUEST (“Accept”) to the
Manufacturer.
        * It would then send a Deliver Action Request message to the Manufacturer
in order to send each ordered RAM and storage to the
        * manufacturer through a for loop.
        */
        public class DeliverServer2 extends CyclicBehaviour{
                public DeliverServer2(Agent a) {
                        super(a);
                }
                @Override
                public void action() {
                        // TODO Auto-generated method stub
                        MessageTemplate mt =
MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
                        ACLMessage msg = myAgent.receive(mt);
```

```
                    if(msg != null) {
                          ACLMessage reply = msg.createReply();

                          try {
                                ContentElement ce = null;
                                ce = getContentManager().extractContent(msg);
                                if(ce instanceof Action) {
                                      Concept action = ((Action)ce).getAction();
                                      if(action instanceof Purchase) {
                                            Purchase purchase = (Purchase)
action;
                                            RAM ram = purchase.getRam();
                                            Storage storage =
purchase.getStorage();
                                            componentsOrders.add(ram);
                                            componentsOrders.add(storage);

      reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
                                            reply.setContent("Accept");
                                            myAgent.send(reply);
                                            DFAgentDescription buyerTemplate =
new DFAgentDescription();
                                            ServiceDescription sd = new
ServiceDescription();
                                            sd.setType("Manufacturer");
                                            buyerTemplate.addServices(sd);
                                            try{
                                                  manufacturers.clear();
                                                  DFAgentDescription[]
agentsType1  = DFService.search(myAgent,buyerTemplate);
                                                  for(int i=0;
i<agentsType1.length; i++){

      manufacturers.add(agentsType1[i].getName()); // this is the AID
                                                  }
                                            }
                                            catch(FIPAException e) {
                                                  e.printStackTrace();
                                            }
                                            for(Component component :
componentsOrders) {

                                                  ACLMessage msg2 = new
ACLMessage(ACLMessage.REQUEST);

      msg2.setConversationId("DeliveryTwo");
                                                  for(AID manufacturer :
manufacturers) {

      msg2.addReceiver(manufacturer);
                                                  }

      msg2.setLanguage(codec.getName());

      msg2.setOntology(ontology.getName());
                                                  Deliver deliver = new
Deliver();

      deliver.setSupplierAID(myAgent.getAID());
```

```
                                                        //Making sure the delivery
time is 4
                                                        component.setDeliveryTime(4);

        //System.out.println("Component quantity is " + component.getQuantity());

        deliver.setComponent(component);
                                                        Action request = new
Action();
                                                        request.setAction(deliver);
                                                        for(AID manufacturer :
manufacturers) {

        request.setActor(manufacturer);
                                                        }
                                                        try {

        getContentManager().fillContent(msg2, request);
                                                                send(msg2);
                                                        }
                                                        catch(CodecException ce2) {
                                                                ce2.printStackTrace();
                                                        }
                                                        catch(OntologyException oe) {
                                                                oe.printStackTrace();
                                                        }

                                                }

                                        }
                                }
                        }catch (CodecException ce) {
                                ce.printStackTrace();
                        }
                        catch (OntologyException oe) {
                                oe.printStackTrace();
                        }
                }
                else {
                        block();
                        flag = true;
                }
        }
        }
```

Appendix 3.7 – recieveComponentsOne Behaviour

```
/*
        *keep receiving Deliver Request Action messages that from supplier one
sequentially until the number of replies is equal to the
        *number of components ordered from Supplier one in the same day or the
number of parts ordered is 0.
        *Each message received, the behaviour would extract the message and add
the received component to an array list.
        *The Manufacturer would then reply to the message with a ACCEPT_PROPOSAL
(“Success”) message.
        */
        public class recieveComponentsOne extends Behaviour {
```

```
		private int count = 0;
		boolean block = false;
		public recieveComponentsOne(Agent a) {
			super(a);
		}
		@Override
		public void action() {
			// TODO Auto-generated method stub
			if(numberOfComponentsOrdered != 0) {
				MessageTemplate mt =
MessageTemplate.MatchConversationId("Delivery");
				ACLMessage msg = myAgent.receive(mt);
				if(msg != null) {
					ACLMessage reply = msg.createReply();
					try {
						ContentElement ce = null;
						ce =
getContentManager().extractContent(msg);
						if(ce instanceof Action) {
							Concept action =
((Action)ce).getAction();
							if(action instanceof Deliver) {
								count++;
								Deliver deliver = (Deliver)
action;
								Component component =
deliver.getComponent();

	//System.out.println("Component Quantity is " + component.getQuantity());
								System.out.println("Delivery
time is " + component.getDeliveryTime());

	deliveredComponents.add(component);

	reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
								reply.setContent("Success");
								myAgent.send(reply);
							}
						}
					}catch (CodecException ce) {
						ce.printStackTrace();
					}
					catch (OntologyException oe) {
						oe.printStackTrace();
					}
				}
				else {
					block();
				}
			}
			else {
				block = true;
			}
		}
		@Override
		public boolean done() {
			// TODO Auto-generated method stub
			return (count == (numberOfComponentsOrdered) || block ==
true);
```

```
		}
	}
```

Appendix 3.8 – recieveComponentsTwo

```
 /*
	* This behaviour keeps receiving Deliver Request Action messages from
supplier two sequentially until the number of replies is
	* equal to the number of components ordered from Supplier two on the same
day, or the number of parts ordered is 0.
	* Each message received, the behaviour would extract the message and add
the received component to an array list.
	*  The Manufacturer would then reply to the message with a ACCEPT_PROPOSAL
(“Success”) message.
	*/
	public class recieveComponentsTwo extends Behaviour {
		private int count = 0;
		boolean block = false;
		public recieveComponentsTwo(Agent a) {
			super(a);
		}
		@Override
		public void action() {
			// TODO Auto-generated method stub
			MessageTemplate mt =
MessageTemplate.MatchConversationId("DeliveryTwo");
			ACLMessage msg = myAgent.receive(mt);
			if(numberOfComponentsOrdered2 != 0) {
				if(msg != null) {
					ACLMessage reply = msg.createReply();
					try {
						ContentElement ce = null;
						ce =
getContentManager().extractContent(msg);
						if(ce instanceof Action) {
							Concept action =
((Action)ce).getAction();

							if(action instanceof Deliver) {
								Deliver deliver = (Deliver)
action;

								Component component =
deliver.getComponent();

								System.out.println("Delivery
time is " + component.getDeliveryTime());

	deliveredComponents.add(component);

	reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
								reply.setContent("Success");
								myAgent.send(reply);

								count++;
							}
						}
					}catch (CodecException ce) {
						ce.printStackTrace();
					}
```

```
                                catch (OntologyException oe) {
                                        oe.printStackTrace();
                                }
                        }
                        else {
                                block();
                        }
                }
                else {
                        block = true;
                }

        }
        @Override
        public boolean done() {
                // TODO Auto-generated method stub
                return (count == (numberOfComponentsOrdered2) || block ==
true);
        }
}
```

Appendix 3.9 – assembleandship

```
/*
         The assembleandship behaviour is a OneShotBehaviour of the Manufacturer
agent which uses the components within the warehouse and uses these
         components to assemble from 1 to 50 smartphones.
        * Each constructed smartphone delivered to the appropriate Customer
through a Ship Action Request message.
        * Each time the manufacturer transmits
        * smartphones to a customer, a profit would increase by number of
smartphones * unit price. The used components would be removed from the
        * warehouse.
        */
       public class assembleandship extends OneShotBehaviour{

                public assembleandship(Agent a) {
                        super(a);
                }
                @Override
                public void action() {
                        for(int j = 0; j < smartphones.size(); j++) {
                                Smartphone newSmartphone = smartphones.get(j);
                                //This make sure number of smartphones that is being
assembled today is less than or equal to 50
                                if((numberOfAssembledSmartphones
+smartphones.get(j).getQuantity()) <= 50) {
                                        for(int i = 0; i < warehouse.size(); i++) {

       if(smartphones.get(j).getBattery().getSize().equals(warehouse.get(i).getSiz
e())
                                                                        &&
smartphones.get(j).getBattery().getQuantity() <= warehouse.get(i).getQuantity()) {

       newSmartphone.setBattery(smartphones.get(j).getBattery());
```

```
                                        }

        if(smartphones.get(j).getRam().getSize().equals(warehouse.get(i).getSize())
&&

        smartphones.get(j).getRam().getQuantity() ==
warehouse.get(i).getQuantity()) {

        newSmartphone.setRam(smartphones.get(j).getRam());
                                        }

        if(smartphones.get(j).getScreen().getSize().equals(warehouse.get(i).getSize
()) &&

        smartphones.get(j).getScreen().getQuantity() ==
warehouse.get(i).getQuantity()) {

        newSmartphone.setScreen(smartphones.get(j).getScreen());
                                        }

        if(smartphones.get(j).getStorage().getSize().equals(warehouse.get(i).getSiz
e()) &&

        smartphones.get(j).getStorage().getQuantity() ==
warehouse.get(i).getQuantity()) {

        newSmartphone.setStorage(smartphones.get(j).getStorage());
                                        }

                                    }
                                    //smartphone is only shipped if it has all of the
needed components
                                    if(newSmartphone.getBattery() != null  &&
newSmartphone.getRam() != null && newSmartphone.getScreen() != null &&
                                                    newSmartphone.getStorage() != null)
{

        newSmartphone.setQuantity(smartphones.get(j).getQuantity());

        newSmartphone.setCustomerAID(smartphones.get(j).getCustomerAID());

        newSmartphone.setDueDate(smartphones.get(j).getDueDate());

        newSmartphone.setPerDayPenality(smartphones.get(j).getPerDayPenality());

        newSmartphone.setPrice(smartphones.get(j).getPrice());

        newSmartphone.setWaitingTime(smartphones.get(j).getWaitingTime());
                                        //part of the totalProfit calculation
                                        totalProfit += (newSmartphone.getPrice() *
newSmartphone.getQuantity());
                                        smartphones.remove(j);
                                        numberOfAssembledSmartphones =
numberOfAssembledSmartphones + newSmartphone.getQuantity();
                                        double waitTime = 0;
                                        waitTime = newSmartphone.getWaitingTime()
- newSmartphone.getDueDate();
                                        if(waitTime > 0) {
                                                double penaltyCost = 0;
```

```
                                        //penalty cost is calculated
                                        penalty_cost = (int) (wait_time *
newSmartphone.getPerDayPenality());
                                        total_penalty_cost += penalty_cost;
                                }
                                else {

                                }
                                for(int i = 0; i < warehouse.size(); i++)
{

      if((warehouse.get(i).getSize().equals(newSmartphone.getBattery().getSize())
|| warehouse.get(i).getSize().equals(newSmartphone.getRam().getSize()) ||
warehouse.get(i).getSize().equals(newSmartphone.getScreen().getSize()) ||
warehouse.get(i).getSize().equals(newSmartphone.getStorage().getSize()) &&
warehouse.get(i).getQuantity() == newSmartphone.getQuantity())) {
                                                warehouse.remove(i);

                                        }
                                }
                                ACLMessage msg = new
ACLMessage(ACLMessage.REQUEST);

      msg.addReceiver(newSmartphone.getCustomerAID());
                                msg.setLanguage(codec.getName());
                                msg.setOntology(ontology.getName());
                                msg.setConversationId("Ship");
                                Ship ship = new Ship();
                                ship.setManufacturerAid(myAgent.getAID());
                                ship.setSmartphone(newSmartphone);
                                Action request = new Action();
                                request.setAction(ship);

      request.setActor(newSmartphone.getCustomerAID());
                                try {
                                        getContentManager().fillContent(msg,
request);
                                        send(msg);
                                }
                                catch(CodecException ce) {
                                        ce.printStackTrace();
                                }
                                catch(OntologyException oe) {
                                        oe.printStackTrace();
                                }

                        }

                }
        }
      }
```

Appendix 3.10 – EndDay Behaviour of the Manufacturer Agent

```
public class EndDay extends OneShotBehaviour {

        public EndDay(Agent a) {
                super(a);
```

```
		}

		@Override
		public void action() {
			System.out.println("Warehouse size is " + warehouse.size());
			//Calculates the per-day per-component warehouse storage cost
			for(int i = 0; i < warehouse.size(); i++) {
				int totalWareHouseCost = 0;
				Component component = warehouse.get(i);
				warehouseCost += (10 * component.getQuantity());

			}
			for(int k = 0; k < deliveredComponents.size(); k++) {
				//Component is added to the warehouse when delivery
time is 0
				if(deliveredComponents.get(k).getDeliveryTime() == 0) {
					Component component = deliveredComponents.get(k);
					warehouse.add(component);

					deliveredComponents.remove(k);
				}
				else {
					Component component = new Component();
					//component delivery time decremented by 1

	component.setDeliveryTime(deliveredComponents.get(k).getDeliveryTime() -
1);

	component.setQuantity(deliveredComponents.get(k).getQuantity());

	component.setSize(deliveredComponents.get(k).getSize());
					deliveredComponents.set(k, component);
				}
			}
			for(int j = 0; j < smartphones.size(); j++) {
				Smartphone smartphone = smartphones.get(j);

	smartphone.setWaitingTime(smartphones.get(j).getWaitingTime() + 1);
				smartphones.set(j, smartphone);
			}
			todaySmartphones.clear();
			numberOfAssembledSmartphones = 0;
			numberOfComponentsOrdered = 0;
			numberOfComponentsOrdered2 = 0;
			quantity = 0;
			//part of the totalProfit calculation
			totalProfit -= warehouseCost;
			totalProfit -= totalComponentsCost;
			totalProfit -= totalPenaltyCost;
			System.out.println("The total profit is £" + totalProfit);
			warehouseCost = 0;
			totalComponentsCost = 0;
			totalPenaltyCost = 0;
			ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
			msg.addReceiver(tickerAgent);
			msg.setContent("done");
			myAgent.send(msg);
			ACLMessage supplierOneDone = new
ACLMessage(ACLMessage.INFORM);
```

```
                        supplierOneDone.setContent("done");
                        for(AID supplier : suppliers) {
                                supplierOneDone.addReceiver(supplier);
                        }
                        myAgent.send(supplierOneDone);
                        ACLMessage supplierTwoDone = new
ACLMessage(ACLMessage.INFORM);
                        supplierTwoDone.setContent("done");
                        for(AID supplier : suppliersTwo) {
                                supplierTwoDone.addReceiver(supplier);
                        }
                        myAgent.send(supplierTwoDone);
                }

        }
```

Appendix 3.11 – TickerWaiter Behaviour of the Manufacturer Agent

```
/*
         * This behaviour would wait for a 'new day' message from the Ticker Agent!
         * When the 'new day' message is received, daily behaviours would operate
sequentially
         */
        public class TickerWaiter extends CyclicBehaviour {
                //behaviour to wait for a new day
                public TickerWaiter(Agent a) {
                        super(a);
                }

                @Override
                public void action() {
                        MessageTemplate mt =
MessageTemplate.or(MessageTemplate.MatchContent("new day"),
                                        MessageTemplate.MatchContent("terminate"));
                        ACLMessage msg = myAgent.receive(mt);
                        if(msg != null) {
                                if(tickerAgent == null) {
                                        tickerAgent = msg.getSender();
                                }
                                if(msg.getContent().equals("new day")) {
                                        SequentialBehaviour dailyActivity = new
SequentialBehaviour();
                                        dailyActivity.addSubBehaviour(new
FindCustomers(myAgent));
                                        dailyActivity.addSubBehaviour(new
FindSuppliersOne(myAgent));
                                        dailyActivity.addSubBehaviour(new
FindSuppliersTwo(myAgent));
                                        dailyActivity.addSubBehaviour(new
OffersServer(myAgent));
                                        dailyActivity.addSubBehaviour(new
OrdersServerOne(myAgent));
                                        dailyActivity.addSubBehaviour(new
OrdersServerTwo(myAgent));
                                        dailyActivity.addSubBehaviour(new
recieveComponentsOne(myAgent));
                                        dailyActivity.addSubBehaviour(new
recieveComponentsTwo(myAgent));
```

```
                                dailyActivity.addSubBehaviour(new
assembleandship(myAgent));
                                dailyActivity.addSubBehaviour(new
EndDay(myAgent));
                                myAgent.addBehaviour(dailyActivity);
                        }
                        else {
                                myAgent.doDelete();
                        }
                }
                else{
                        block();
                }
        }
}
```

Appendix 3.12 - TickerWaiter Behaviour of the Customer Agent

```
public class TickerWaiter extends CyclicBehaviour {

        //behaviour to wait for a new day
        public TickerWaiter(Agent a) {
                super(a);
        }

        @Override
        public void action() {
                MessageTemplate mt =
MessageTemplate.or(MessageTemplate.MatchContent("new day"),
                                MessageTemplate.MatchContent("terminate"));
                ACLMessage msg = myAgent.receive(mt);
                if(msg != null) {
                        if(tickerAgent == null) {
                                tickerAgent = msg.getSender();
                        }
                        if(msg.getContent().equals("new day")) {
                                //spawn new sequential behaviour for day's
activities
                                SequentialBehaviour dailyActivity = new
SequentialBehaviour();
                                //sub-behaviours will execute in the order they
are added
                                dailyActivity.addSubBehaviour(new
FindManufacturers(myAgent));
                                dailyActivity.addSubBehaviour(new
OrderSmartphones(myAgent));
                            //dailyActivity.addSubBehaviour(new
getSmartphones(myAgent));

                                dailyActivity.addSubBehaviour(new
EndDay(myAgent));
                                //dailyActivity.addSubBehaviour(new
CollectOffers(myAgent));

                                //dailyActivity.addSubBehaviour(new
EndDay(myAgent));
                                myAgent.addBehaviour(dailyActivity);
                        }
                        else {
```

```
                                //termination message to end simulation
                                myAgent.doDelete();
                        }
                }
                else{
                        block();
                }
        }

}
```

Appendix 3.13 – EndDay Behaviour of the Customer Behaviour

```
public class EndDay extends OneShotBehaviour {
        public EndDay(Agent a) {
                super(a);
        }
        @Override
        public void action() {
                ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                msg.addReceiver(tickerAgent);
                msg.setContent("done");
                myAgent.send(msg);
        }
}
```

Appendix 3.14 – SynchAgentsBehaviour

```
public class SynchAgentsBehaviour extends Behaviour {

        private int step = 0;
        private int numFinReceived = 0; //finished messages from other
agents
        public int day = 0;
        private ArrayList<AID> simulationAgents = new ArrayList<>();
        /**
         * @param a  the agent executing the behaviour
         */
        public SynchAgentsBehaviour(Agent a) {
                super(a);
        }

        @Override
        public void action() {
                switch(step) {
                case 0:
                        System.out.println("Ticker ending message");
                        //find all agents using directory service
                        DFAgentDescription template1 = new
DFAgentDescription();
                        ServiceDescription sd = new ServiceDescription();
                        sd.setType("Customer");
                        template1.addServices(sd);
                        DFAgentDescription template2 = new
DFAgentDescription();
                        ServiceDescription sd2 = new ServiceDescription();
                        sd2.setType("Manufacturer");
                        template2.addServices(sd2);
                        DFAgentDescription template3 = new
DFAgentDescription();
                        ServiceDescription sd3 = new ServiceDescription();
```

```
                        sd3.setType("Supplier");
                        template3.addServices(sd3);
                        DFAgentDescription template4 = new
DFAgentDescription();
                        ServiceDescription sd4 = new ServiceDescription();
                        sd4.setType("SupplierTwo");
                        template4.addServices(sd4);


                        try{
                                DFAgentDescription[] agentsType1  =
DFService.search(myAgent,template1);
                                for(int i=0; i<agentsType1.length; i++){

      simulationAgents.add(agentsType1[i].getName()); // this is the AID
                                }
                                DFAgentDescription[] agentsType2  =
DFService.search(myAgent,template2);
                                for(int i=0; i<agentsType2.length; i++){

      simulationAgents.add(agentsType2[i].getName()); // this is the AID
                                }
                                DFAgentDescription[] agentsType3  =
DFService.search(myAgent,template3);
                                for(int i=0; i<agentsType3.length; i++){

      simulationAgents.add(agentsType3[i].getName()); // this is the AID
                                }
                                DFAgentDescription[] agentsType4  =
DFService.search(myAgent,template4);
                                for(int i=0; i<agentsType4.length; i++){

      simulationAgents.add(agentsType4[i].getName()); // this is the AID
                                }




                        }
                        catch(FIPAException e) {
                                e.printStackTrace();
                        }
                        //send new day message to each agent
                        ACLMessage tick = new ACLMessage(ACLMessage.INFORM);
                        tick.setContent("new day");
                        for(AID id : simulationAgents) {
                                tick.addReceiver(id);
                        }
                        myAgent.send(tick);
                        step++;
                        day++;
                        break;
                case 1:
                        //wait to receive a "done" message from all agents
                        MessageTemplate mt =
MessageTemplate.MatchContent("done");
                        ACLMessage msg = myAgent.receive(mt);
                        if(msg != null) {
                                numFinReceived++;
```

```
                        if(numFinReceived >= simulationAgents.size()) {
                                step++;
                        }
                }
                else {
                        block();
                }

        }
}

@Override
public boolean done() {
        return step == 2;
}


@Override
public void reset() {
        super.reset();
        step = 0;
        simulationAgents.clear();
        numFinReceived = 0;
}


@Override
public int onEnd() {
        System.out.println("End of day " + day);
        if(day == NUM_DAYS) {
                //send termination message to each agent
                ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                msg.setContent("terminate");
                for(AID agent : simulationAgents) {
                        msg.addReceiver(agent);
                }
                myAgent.send(msg);


        }
        else {
                reset();
                myAgent.addBehaviour(this);
        }

        return 0;
}
}
```

Appendix 3.15 – TickerWaiter Behaviour of Supplier One

```
/*
 * This behaviour would wait for a new day and then all behaviours would be
operating until the end of the day.
 */
public class TickerWaiter extends CyclicBehaviour {

        //behaviour to wait for a new day
        public TickerWaiter(Agent a) {
                super(a);
```

```
		}

		@Override
		public void action() {
			MessageTemplate mt =
MessageTemplate.or(MessageTemplate.MatchContent("new day"),
					MessageTemplate.MatchContent("terminate"));
			ACLMessage msg = myAgent.receive(mt);
			if(msg != null) {
				if(tickerAgent == null) {
					tickerAgent = msg.getSender();
				}
				if(msg.getContent().equals("new day")) {
					CyclicBehaviour os = new DeliverServer(myAgent);
					//doWait(1000);
					myAgent.addBehaviour(os);

					//myAgent.addBehaviour(new
DeliverServer(myAgent));
					ArrayList<Behaviour> cyclicBehaviours = new
ArrayList<>();
					cyclicBehaviours.add(os);
					myAgent.addBehaviour(new
EndDayListener(myAgent,cyclicBehaviours));
				}
				else {
					myAgent.doDelete();
				}
			}
			else{
				block();
			}
		}
	}
```

Appendix 3.16 – TickerWaiter Behaviour of Supplier Two

```
/*
	 * This behaviour would wait for a new day and then all behaviours would be
operating until the end of the day.
	 */
	public class TickerWaiter extends CyclicBehaviour {
		//behaviour to wait for a new day
		public TickerWaiter(Agent a) {
			super(a);
		}

		@Override
		public void action() {
			MessageTemplate mt =
MessageTemplate.or(MessageTemplate.MatchContent("new day"),
					MessageTemplate.MatchContent("terminate"));
			ACLMessage msg = myAgent.receive(mt);
			if(msg != null) {
				if(tickerAgent == null) {
					tickerAgent = msg.getSender();
				}
				if(msg.getContent().equals("new day")) {
					CyclicBehaviour sb = new DeliverServer2(myAgent);
					//doWait(1000);
```

```
                                myAgent.addBehaviour(sb);

                                //myAgent.addBehaviour(new
DeliverServer(myAgent));
                                ArrayList<Behaviour> cyclicBehaviours = new
ArrayList<>();
                                cyclicBehaviours.add(sb);
                                myAgent.addBehaviour(new
EndDayListener(myAgent,cyclicBehaviours));
                        }
                        else {
                                myAgent.doDelete();
                        }
                }
                else{
                        block();
                }
        }
    }
```

Appendix 3.17 – EndDayListener Behaviour

```
/*
         * This behaviour would wait for a new day and then all behaviours would be
operating until the end of the day.
         */
        public class EndDayListener extends CyclicBehaviour {
                private int buyersFinished = 0;
                private List<Behaviour> toRemove;

                public EndDayListener(Agent a, List<Behaviour> toRemove) {
                        super(a);
                        this.toRemove = toRemove;
                }

                @Override
                public void action() {
                        flag = false;
                        MessageTemplate mt = MessageTemplate.MatchContent("done");
                        ACLMessage msg = myAgent.receive(mt);
                        if(msg != null) {
                                //we are finished
                                componentsOrders.clear();
                                ACLMessage tick = new ACLMessage(ACLMessage.INFORM);
                                tick.setContent("done");
                                tick.addReceiver(tickerAgent);
                                myAgent.send(tick);
                                //remove behaviours
                                for(Behaviour b : toRemove) {
                                        myAgent.removeBehaviour(b);
                                }
                                myAgent.removeBehaviour(this);
                        }
                        else {
                                block();
                        }
                }
        }
```