The Zeus Agent Building Toolkit

ZEUS Methodology Documentation Part II

Case Study 1

FruitMarket

An Agent Trading Application

Jaron Collis, jaron@info.bt.co.uk

Intelligent Systems Research Group, BT Labs

Release 1.01, November 1999

Index

"People say you can't compare apples and oranges. But why not?
They are both hand-held, round, edible, fruity things that grow on trees."
-- Anonymous

Document History

Version 1.01 - Added Index; updated section 5

1 Specification

This case study describes the implementation of a sample marketplace application called 'FruitMarket'. The objective is to enable the market participants to electronically trade tasty fruit over a network via their agents.

For simplicity, we shall assume only certain types of fruit will be traded: apples, oranges, pears, bananas and melons. Furthermore, we shall assume that each type of fruit is a commodity and traded in uniform size boxes, i.e. there is no inherent difference between two different boxes of apples.

To keep the example small-scale, we shall assume there are 3 participants that are each capable of buying and selling, these are:

The first participant will be called **OrchardBot**, the representative of an orchard that is attempting to sell some of its produce. We shall assume the produce has just been picked, and so it possesses a stock of 100 boxes of apples, 80 boxes of oranges and 60 boxes pears, but alas, no cash. It will attempt to sell its goods with a minimum profit margin of 10%.

The second participant is **SupplyBot**, a representative of several fruit producers that begins with a small stock of 30 boxes of apples, 10 boxes of pears, 30 boxes of oranges, 20 boxes of melons and 20 boxes of bananas. As the supplier needs to remain well stocked it has a budget of €100 to fund purchases. This agent's minimum profit margin is 5%.

The third participant is **ShopBot**, a representative of a supermarket chain. Its existing stock is down to 10 boxes of bananas, 5 boxes of pears and 10 boxes of melons; but it has a budget of €500 to purchase additional stocks. This agent's minimum profit margin is also 5%.

We shall assume all transactions are conducted in Euros, (hence the use of the € symbol), although as only one currency unit is involved the particular type used has little significance in this application. Furthermore to provide a baseline for prices we shall assume that:

- boxes of apples, oranges and pears each cost €5
- boxes of bananas cost €8
- boxes of melons cost €10

We shall also assume that the market is open, (i.e. agents can join or leave at any time), and agents are not initially aware of their counterparts. As this is an instructive example a means of analysing the activity of the marketplace would also prove very useful.

About this Document

This document can be read in two ways: if the reader is interested in learning how the FruitMarket example was created, or wants to try implementing it using the ZEUS Agent Generator, then continue reading with Section 2. This explains the how an appropriate role model is chosen and configured to match the required specification, this will form the basis for the application design, as explained in Section 3. Then section 4 describes the procedure by which this design is realised using the ZEUS Agent Generator tool.

Alternatively, if the reader is only interested in running the example then they should skip ahead to Section 5, which describes some significant features of the example and how to use it.

2 Application Analysis

Once we have a specification, the desired system can be situated within a domain that contains similar features and challenges. As the agents will exhibit trading behaviour it is obvious that this application is best situated within the *Multi-Agent Trading* domain. This domain contains two role models: Distributed Marketplace and Institutional Auction, hence the next step is to consider which roles from the domain's role models best fit our specification.

2.1 Select Role Models

From the initial specification we notice that the agents with trade between themselves rather than through an intermediary. This would tend to suggest that the *Trader* role of the *Distributed Marketplace* role model would better model the participants than the *Auction Participant* role, which would require the presence of an *Auctioneer* agent to function. Notice how role models can be used to support a decision on how to model the problem one way in preference to another.

From the *Distributed Marketplace* role model we can identify the roles with which the *Trader* agents will collaborate. The role model shows each *Trader* can play *Inquirer* and *Registrant* roles relative to a *Broker*. This suggests that a *Broker* agent should be incorporated into our solution, which is not inconsistent with our problem specification.

Next we need to consider the entities of the standard *Zeus Application* role model, upon which this role model is based. We will need an agent in the *Agent Name Server* role, but as the *Broker* duplicates the functionality of the *Facilitator* only one needs to be created.

It is possible for each of the identified roles to be played by an individual agent, and which interact together to accomplish a common cause. Alternatively, a single agent could play several roles. (See Guidelines on "What is an Agent?" in the Role Modelling Guide). In this example the latter arrangement seems more appropriate, i.e. to create one agent for each participant that will encapsulate all 4 roles of the *Trader*.

Having considered all the roles within the basic *Distributed Marketplace* role model we can move on to consider its common variations. One of these, the *Visualiser* variation would seem to satisfy the aspect of the specification that calls for some means of activity analysis. The other variation, the *Mediator*, is not alluded to in the specification and so shall be ignored in this application.

So, to summarise, after deciding on the domain and considering its constituent role models we have decided to base our solution on the *Distributed Marketplace* role model. Then we have decided to create several agents to fulfil the roles found within the role model:

Agent Name	**Roles Played**
OrchardBot	Trader (Buyer, Seller, Inquirer, Registrant)
SupplyBot	Trader (Buyer, Seller, Inquirer, Registrant)
ShopBot	Trader (Buyer, Seller, Inquirer, Registrant)
Broker	Broker (Facilitator)
Visual	Visualiser
ANS	Agent Name Server

Having identified what roles should exist within the application, we can begin thinking about how agents will realise each role.

2.2 List Agent Responsibilities

Each role played by an agent entails some responsibilities, e.g. resources that will need to be produced or consumed, interactions with external systems etc. Hence the next stage is to use the role descriptions to create a list of responsibilities for each agent.

From the descriptions in the *Distributed Marketplace* role model we can obtain the list of responsibilities for the 4 constituent roles of a *Trader* agent. As some responsibilities are shared between roles they only need to be considered once. The responsibilities involved can be categorised as social or domain responsibilities, the former involving interaction with other agents, and the latter involving some local application-specific activity; this results in the following:

TRADER - Social Responsibilities	
Origin	**Responsibility**
Seller-Registrant	To register and de-register presence in marketplace
Buyer-Inquirer	To request information on known sellers
Buyer	To send bids to potential vendors
Seller	To receive and respond to bids

TRADER – Domain Responsibilities	
Origin	**Responsibility**
Buyer	To facilitate the entry of user transactions
Buyer	To interpret vendor responses
Buyer, Seller	To exchange payment and ownership
Seller	To facilitate user selling preferences
Seller	To interpret bids

The next role to consider is the *Broker*:

BROKER - Social Responsibilities	
Origin	**Responsibility**
Broker	To receive notifications from participants
Broker	To respond to queries on market participants

BROKER – Domain Responsibilities	
Broker	To store information on market participants

Finally, as the functionality of the *Visualiser* role is already present in the standard Zeus Application role model we shall adopt the pre-built Visualiser tool; hence no design is necessary for this role.

3 Application Design

The design process is a process of refinement, mapping each of the responsibilities identified in the previous stage to a generalised problem, and then choosing the most appropriate solution.

3.1 Problem Design

Having obtained the responsibilities required for each agent, the next stage is to map each responsibility to the problem it attempts to solve. As these problems tend to be variations on common challenges, it is possible to reapply past tried and tested solutions to solve the problems in question. We begin by considering the Broker:

BROKER

SOCIAL RESPONSIBILITIES

Responsibility:	To receive notifications from participants
Origin:	Broker
Problem:	Receive and Store [Sellers, Commodities-For-Sale]
Solution:	`Configure the ZEUS Facilitator Agent (`**`UTIL-2`**`)`
Explanation:	By default the Facilitator will query agents pro-actively, but as this scenario calls for a reactive Broker this can be achieved by setting its cycle time to 0.

Responsibility:	To respond to queries on market participants
Origin:	Broker
Problem:	Match Requirements, Send Message [Buyer, Seller Identity]
Solution:	Automatic – functionality provided by ZEUS Facilitator Agent

DOMAIN RESPONSIBILITIES

Responsibility:	To store information on market participants
Origin:	Broker
Problem:	Store [Seller – Commodity, Price]
Solution:	Automatic – functionality provided by ZEUS Facilitator Agent

How then have these solutions been derived? In the case of the *Broker* it is through the reuse and adaptation of existing solutions from its parent role, the *Facilitator*. This is justifiable because the problems facing the *Broker* are very similar to those tackled by the *Facilitator*. This ethos of reusing existing solutions is central to most development toolkits. (**Note**: as the Broker is descended from the Facilitator it will be realised as a Utility agent rather than a Task agent, this is explained later).

The next role to consider is the Trader, which is descended from the Task Agent role. The Task Agent role is quite generic, and consequently there will be much less scope for reuse in the application specific *Trader* role. Instead, solving the *Trader* problems will require deeper knowledge of agent behaviours, protocols and strategies; this is summarised in the following table.

TRADER

SOCIAL RESPONSIBILITIES

Responsibility:	To register and de-register presence in marketplace
Origin:	Seller-Registrant
Problem:	Sending Message [to: Broker, about: Commodities-For-Sale]
Solution:	`Equip agent with appropriate co-ordination protocol. (`**`COORD-1`**`)`
Explanation:	By default Task Agents will reactively respond to Facilitator requests. To advertise proactively the agent will need to be supplied with a co-ordination protocol that sends a message to the Broker notifying it of all commodities available for sale.

Responsibility:	To request information on known sellers
Origin:	Buyer-Inquirer
Problem:	Sending Message [to: Broker, about: Commodities-Wanted]

Solution:	Automatic – part of default Task Agent functionality

Responsibility:	To send bids to potential vendors
Origin:	Buyer
Problem:	Initiate Dialogue [with: Seller, about: Commodity-For-Sale]
Solution:	`Equip agent with co-ordination protocol (`**`COORD-1`**`)`
Explanation:	To initiate and engage in a transaction dialogue these agents will need an appropriate co-ordination protocol. For the buyer the most appropriate is the **Initiator** version of the FIPA defined Multi-Round Contract Net, (the justification for this is discussed in the Application Realisation Guide).

Responsibility:	To receive and respond to bids
Origin:	Seller
Problem:	Engage in Dialogue [with: Buyer, about: Commodity-For-Sale]
Solution:	`Equip agent with co-ordination protocol (`**`COORD-1`**`)`
Explanation:	To reciprocate in a dialogue the selling agent will need a complementary co-ordination protocol. In this case the most appropriate is the **Respondent** version of the Multi-Round Contract Net.

From the above table we can see that one problem is solved by default by virtue of the functionality of the generic ZEUS agent, leaving three other solutions that need to be realised. The references (**`COORD-1`** and **`COORD-2`**) refer to sections of the ZEUS Application Realisation Guide.

Where a solution is not automatic, it must be realised through the ZEUS Agent Generator Tool, as described in the ***ZEUS Application Realisation Guide***.

The above table also demonstrates that matching a problem to an existing solution requires some expertise in the construction of agent systems. The purpose of the case studies in this document is to describe the common problems, and teach the reader when to apply or adapt existing solutions.

Finally, to complete the design of the Trader agent we need to consider its domain responsibilities.

TRADER

DOMAIN RESPONSIBILITIES

Responsibility:	To facilitate the entry of user transactions
Origin:	Buyer
Problem:	Information Entry [Commodity, Price, Number]
Solutions:	➢ Use default User Interface (Automatic) or
	➢ `Add Buying User Interface (`**`TACF-1`**` and `**`IMPL-4A`**`)`
Explanation:	Buying instructions can be issued in the form of goals, which can be entered through the agent's standard goal entry user interface. The alternative, which offers the possibility of a much more sophisticated front-end, is to create a custom GUI and link it to the agent using its external interface, (although this will require some programming).

Responsibility:	To interpret seller responses
Origin:	Buyer
Problem:	Evaluate [Commodity, Price]

Solution:	➢ `Equip agent with negotiation strategies (`**`COORD-2`**`)`
Explanation:	Agents can possess strategies that influence their dealings with others. The ZEUS toolkit provides several that are useful for trading negotiations, (see the Technical Manual for details). Alternatively if none seem suitable a custom strategy can be written and used. In this case the **GrowthStrategy** seems the most appropriate, it will begin bidding at a low price and gradually increase it until a vendor accepts or a certain period of time elapses. When implementing a trading strategy the developer will probably need to include information on the intrinsic worth of items, seasonal demand etc.

Responsibility:	To exchange payment and ownership
Origin:	Buyer, Seller
Problem:	Transfer [Money, Commodities]
Solution:	➢ Automatic (simulated by sending facts) *or*
	➢ `Implement Domain Function (`**`TASK-1`**` and `**`IMPL-2`**`)`
Explanation:	Once agreed, a transaction can be simulated by both parties exchanging messages that represent payment and the purchased commodity. A more realistic alternative is to perform the through some external program, a so-called 'Domain Function'. This could involve a transaction settlement system like Visa, and a delivery system like FedEx.

Responsibility:	To facilitate the entry of user selling preferences
Origin:	Seller
Problem:	Information Entry [Commodity, Price, Number]
Solution:	➢ Use default User Interface (Automatic) or
	➢ `Add Selling User Interface (`**`TACF-1`**` and `**`IMPL-4A`**`)`
Explanation:	As for entry of buying preferences

Responsibility:	To interpret incoming bids
Origin:	Seller
Problem:	Evaluate [Commodity, Price]
Solution:	➢ `Equip agent with negotiation strategies (`**`COORD-2`**`)`
Explanation:	Like buying behaviour, selling behaviour is governed by negotiation strategies. The strategy is likely to be used in conjunction with trading expertise like seasonal demand, pricing histories etc. In this case the **DecayFunction** seems appropriate, this will lower the asking price gradually until it matches an incoming bid.

By now, we have identified the means to realise all the agents' responsibilities. So we can move onto the next stage of the design process: knowledge modelling.

3.2 Knowledge Modelling

The next stage of the design process is to model the declarative knowledge that will be used by the agent roles. This stage should result in the concepts inherent to the application (termed Facts within ZEUS), their attributes and possible values (also known as constraints).

Concept Identification

The key concepts in the FruitMarket application are mentioned in the specification, namely apples, oranges, pears, bananas and melons. As these concepts refer to physical instances rather than abstract ones they will inherit from the **Entity** fact, this is a child of the root **ZeusFact** concept that provides all its children with a cardinality attribute that represents the number of each concept in existence.

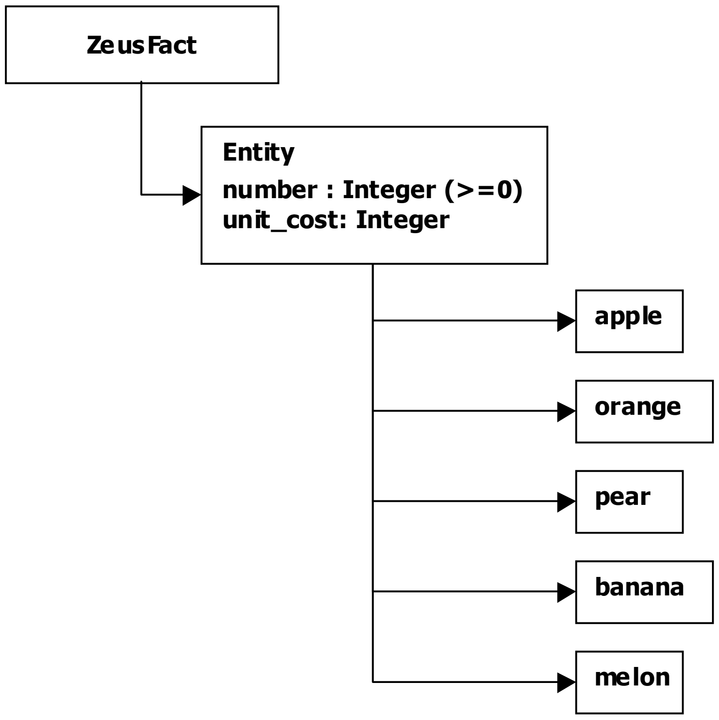
All Entity facts also possess an attribute that refers to the concept's inherent value. In our trading application this can be used to store information about the price of each commodity. (Although this is not appropriate for applications where there is no notion of inherent worth and prices are completely determined by supply and demand).

No other attributes are explicitly mentioned in the specification, although as an exercise the scenario could easily be extended with attributes like sell-by-date, colour, country of origin etc.

Typing and Constraints

Constraints provide a means of verification, restricting values to a subset of legal values. In this case the cardinality constraint (must be non-negative) already exists and so no additional constraints need to be defined. Also, as the value of each concept is not necessarily universal (different agents may attach different valuations) there is no need to provide a default value for the unit_cost attribute of each fruit. Instead the fruit valuations will be added when the agents are defined.

Hence the resulting facts, their attributes, types and constraints (also known as the 'Problem Ontology') looks like this:



Once the ontology is defined we can begin the process of realising the design.

4 Application Realisation

Once a design has been produced the next stage is to realise it using the available tools. In this case the tools in question are provided by the ZEUS toolkit, in the form of the Agent Generator editor. The realisation process combines the steps necessary to create a generic ZEUS agent with the steps necessary to implement the role-specific solutions identified during the previous phase.

The realisation process, (described in the ZEUS Application Realisation Guide), consists of the following activities:

1) Ontology Creation
2) Agent Creation, for each task agent this consists of:
   - Agent Definition
   - Task Description
   - Agent Organisation
   - Agent Co-ordination
3) Utility Agent Configuration
4) Task Agent Configuration
5) Agent Implementation

The purpose of these stages is to translate the design we have derived from the role models into agent descriptions that can be automatically created by the ZEUS Agent Generator tool. Thus, now is the time to launch the Agent Generator tool, (instructions on how to do this are provided in the Realisation Guide). Now, the remainder of this section will walk-through the stages of the ZEUS agent development methodology describing how the *FruitMarket* application can be implemented.

4.1 Ontology Creation

This stage involves using the Ontology Editor tool, as described in Section 2 of the Realisation Guide, (see entry **ONT-1**). Once the Editor has been opened we are ready to start defining the concepts of the *FruitMarket* ontology; this involves the following steps:

- First, click on the ZeusFact entry to show its child facts; now select the Entity fact and create a child fact called **apple**; (see entry **ONT-3**)

If you haven't already done so, click on the "Toggle Shown Attributes" button, this will show that the Commodity fact inherits a cardinality attribute (called **number**) and a value attribute (called **unit_cost**) from the Entity fact; consequently these attributes do not need to be added manually.

- Ensuring the newly created Apple node is selected, click on the "Add New Peer Fact" button. Rename the entry to **orange**.

- Repeat the above action, this time renaming the new entry to **pear**.

- Repeat the above action, this time renaming the new entry to **banana**.

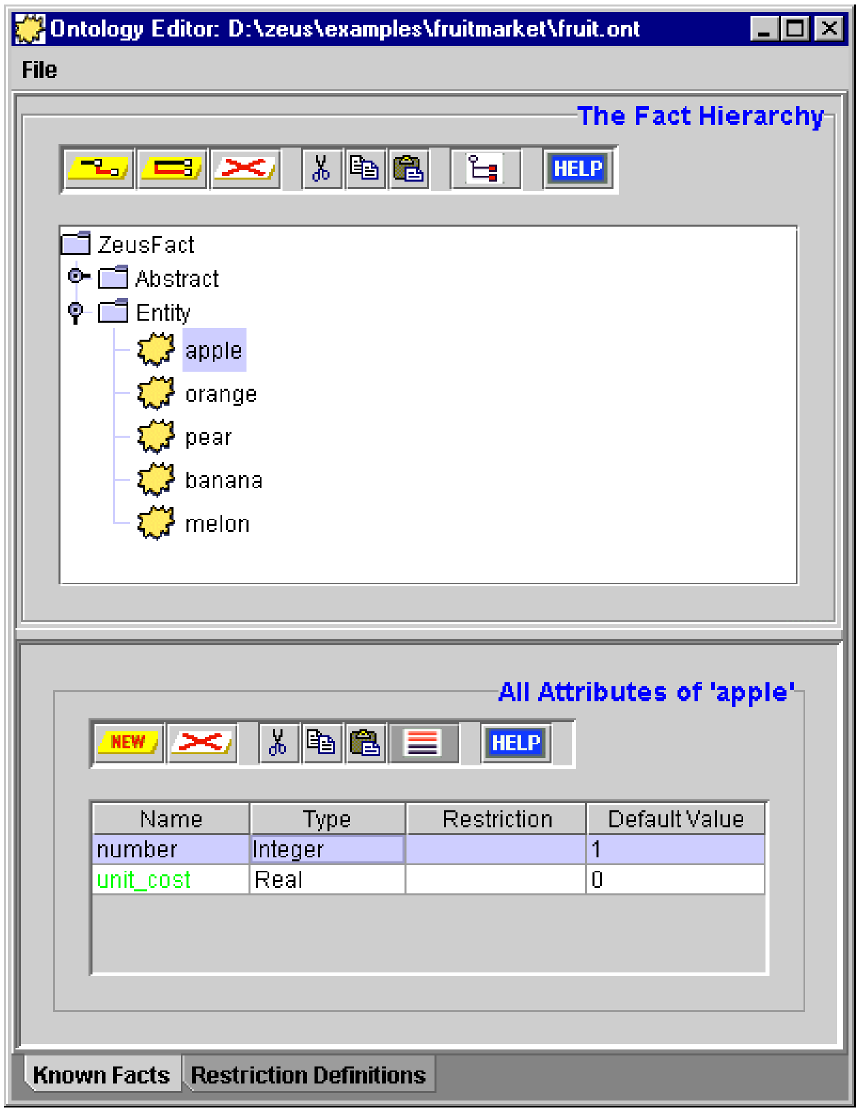
- Repeat the above action, this time renaming the new entry to **melon**.

The contents of the Ontology Editor should now look like that shown in Figure 1.

As the ontology now represents what was specified in section 3, it can be saved and used as the basis for our agent application.

- Save the Ontology as 'fruit.ont', (see 'How to Save an Ontology').

With the ontology created, we can leave the Ontology Editor and begin the agent creation process.


Ontology Editor: D:\zeus\examples\fruitmarket\fruit.ont

File

The Fact Hierarchy

- ZeusFact
  - Abstract
  - Entity
    - apple
    - orange
    - pear
    - banana
    - melon

All Attributes of 'apple'

Name	Type	Restriction	Default Value
number	Integer		1
unit_cost	Real		0

Figure 1: Screenshot of the Ontology Editor after the *FruitMarket* ontology has been entered

4.2 Agent Creation

The agent creation stage consists of several sub-processes that are repeated for each different task agent in the application. This process is described in Section 3 of the Realisation Guide, (see 'How to Create a New Agent'). To start, then:

- Create a new agent, and rename it to **OrchardBot**.

The result of this action is a new generic ZEUS agent. In the phases that follow this generic agent will be configured to satisfy the application-specific responsibilities of its roles.

- Double-click on the 'OrchardBot' entry to open the Agent Editor window, we can now begin the Agent Definition process.

The Agent Definition Process

This stage is documented in Section 3.1 of the Realisation Guide. First we need to look back at the design produced for *Trader*, (i.e. its social and domain responsibilities), to discover whether there are any solutions that require the agent's definition to be changed, (these are identifiable through their

`DEF-x` code references). As it happens, there are none - and so this agent can be left with the default settings.

At this stage we also have to specify the initial resources of the **OrchardBot** agent, (this process is described in activity **DEF-3**). The actions to be taken here are:

- Click on the "New Initial Resource" button, and select **Apple** as the fact type from the fact hierarchy.
- Now double-click on the new entry's `Instance` field, and rename it to something more comprehensible, like **appleStock**.
- To set the number of apples held by the agent, double click on the `Value` field of the **number** entry in the attribute table, change it to **100**.
- To set this agent's valuation of apples, double click on the `Value` field of the **unit_cost** entry in the attribute table and change it to **5**.
- Now click on the "New Initial Resource" button again, and select **Orange** as the fact type.
- Now double-click on the new entry's `Instance` field, and rename it to **orangeStock**.
- Ensuring that the entry for oranges is selected edit the `Value` field of the **number** entry in the attribute table, change it to **80**.
- Then to provide a valuation for oranges, edit the `Value` field of the **unit_cost** entry in the attribute table and change it to **5**.
- Now click on the "New Initial Resource" button again, and select **Pear** as the fact type.
- Now double-click on the new entry's `Instance` field, and rename it to **pearStock**.
- Ensuring that the entry for pears is selected edit the `Value` field of the **number** entry in the attribute table, change it to **60**.
- Then to provide a valuation for pears, edit the `Value` field of the **unit_cost** entry in the attribute table and change it to **5**.
- Now click on the "New Initial Resource" button again, and select **Money** as the fact type.
- Now double-click on the new entry's `Instance` field, and rename it to **cash**.
- Ensuring that the entry for money is selected edit the `Value` field of the **number** entry in the attribute table, change it to **0**. Times are obviously hard for OrchardBot.

Although the OrchardBot does not begin life with any bananas or melons, we shall include stock entries for these so that the agent will be capable of trading in all commodities.

- Click on the "New Initial Resource" button again, and select **melon** as the fact type.
- Now double-click on the new entry's `Instance` field, and rename it to **melonStock**.
- Ensuring the new entry is selected, change the `Value` field of the **number** entry to **0**.
- Now change the `Value` field of the **unit_cost** entry to **8**.
- Then click on the "New Initial Resource" button again, and select **banana** as the fact type.
- Now double-click on the new entry's `Instance` field, and rename it to **bananaStock**.
- Ensuring the new entry is selected, change the `Value` field of the **number** entry to **0**.
- Now change the `Value` field of the **unit_cost** entry to **10**.

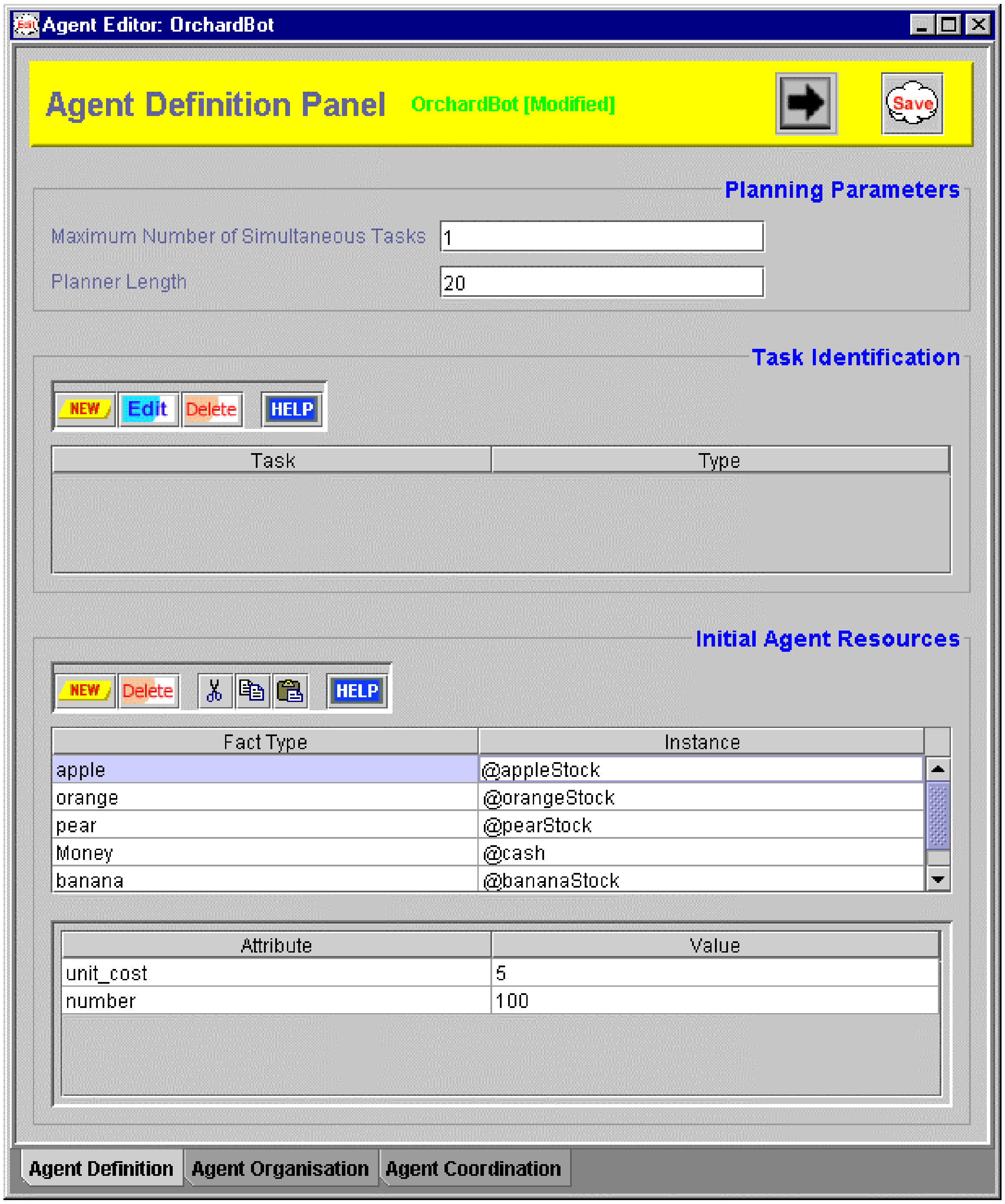
This concludes the Agent Definition process, by now the contents of the Agent Editor window should look like those in Figure 2.

The Task Description Process

Once again we look back at our design for the *Trader* agent to see if there are any tasks to implement. This reveals there is potentially one domain task – settlement and ownership transfer. However for this example we shall simulate settlement by message passing; consequently no tasks need be implemented and so no action needs to be performed at this phase.

The Agent Organisation Process

Looking at the Trader design we can see there are no solutions that affect the organisational layer of the *Trader* agent. Also, the problem specification did not state that any agents had prior knowledge of each other, and so no acquaintances need to be defined. Hence we can skip this stage of the development process too.



Maximum Number of Simultaneous Tasks: 1

Planner Length: 20

Task	Type

Fact Type	Instance
apple	@appleStock
orange	@orangeStock
pear	@pearStock
Money	@cash
banana	@bananaStock

Attribute	Value
unit_cost	5
number	100

Figure 2: A screenshot of the Agent Definition Panel after entering the 'OrchardBot' agent details.

The Agent Co-ordination Process

Again we refer back to the *Trader* design, where we can see several solutions which will affect the agent's co-ordination layer. The aspects that affect the agent's co-ordination layer are identifiable through their **`COORD-x`** code references, (the numbers of each refer to the order in which they should be realised, as some are dependent on the existence of others), namely:

```
•  Equip agent with co-ordination protocol [Initiator]      (COORD-1)
•  Equip agent with co-ordination protocol [Respondent]     (COORD-1)
•  Equip agent with negotiation strategies [GrowthFunction] (COORD-2)
•  Equip agent with negotiation strategies [DecayFunction]  (COORD-2)
```

In other words, the agent will need to be equipped to buy and sell.

Equipping the Agent to Buy

The design in Section 3 tells us that for an agent to buy a commodity it must begin a dialogue with a potential vendor. Hence this process involves equipping the agent an Initiator protocol, (this is described in the entry for activity **COORD-1**), namely:

- Click on the checkbox beside the entry for **Fipa-Contract-Net-Manager**.

Now we can define the negotiation strategy that will be used in conjunction with this protocol.

- Ensure that the **Fipa-Contract-Net-Manager** entry has been selected in the upper 'Co-ordination Protocols' table.

The lower 'Co-ordination Strategies' table will now show the applicability of that protocol, (as described in entry **COORD-2**). By default the `Mode` field will contain a ticked checkbox to indicate that this strategy will be used in conjunction with facts of the type described in the `Fact Type` field. As this field contains `ZeusFact` this strategy will be used to buy facts of any type.

Our specification does not suggest that the agent will use specialised strategies for particular interactions, so the `Agents` and `Relations` fields can be left unchanged.

- Now double-click on the `Strategy` field and select the entry for **GrowthFunction** from the list of available strategies.

Now we can define the parameters that determine how the GrowthFunction strategy runs, (these are described in the Technical Manual):

⇒ *min.percent*: the fraction of the commodity's perceived value at which the agent will begin bidding

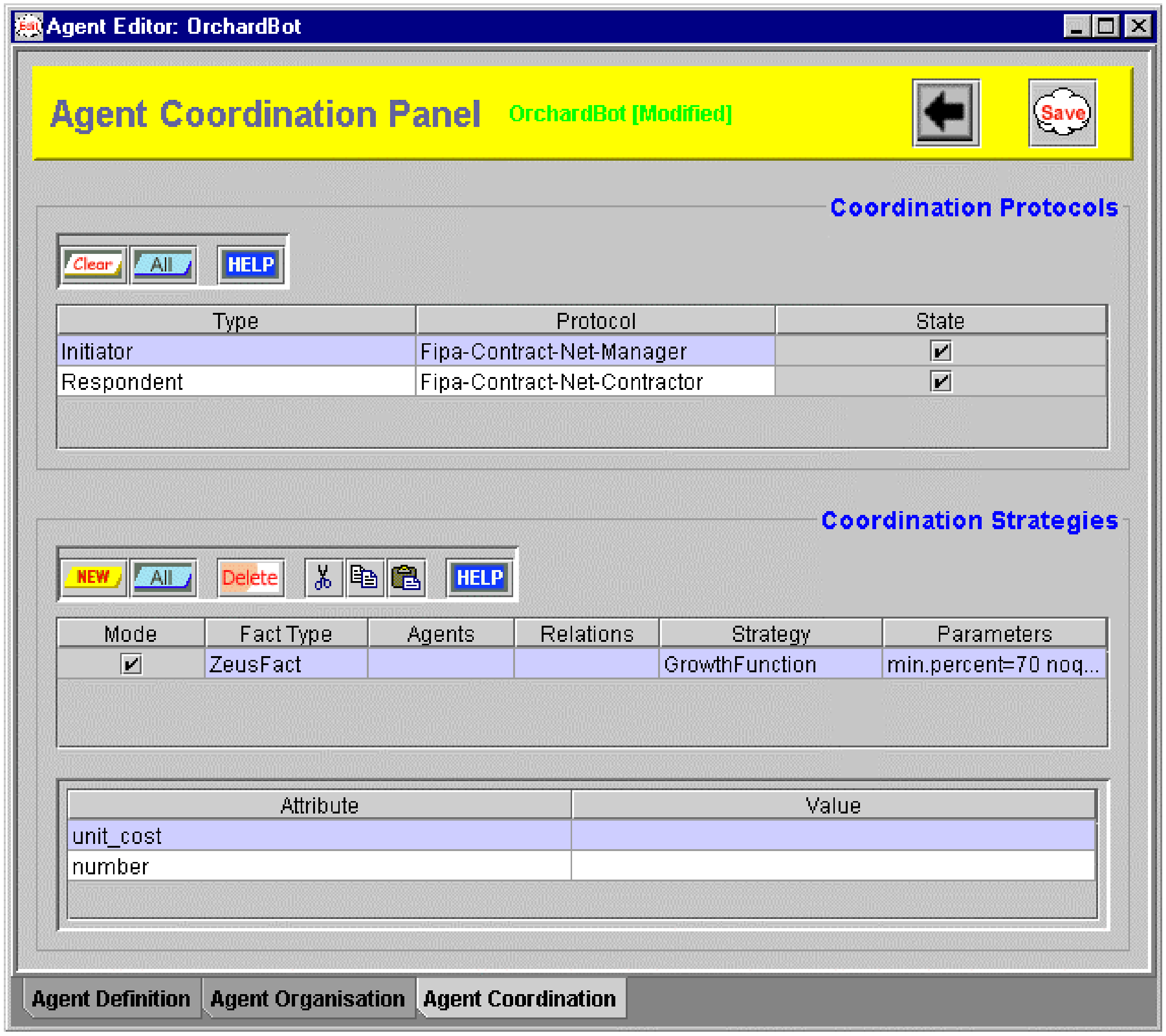
⇒ *no.quibble*: the difference in price considered insignificant by the agent; when the offer price higher than the bid price but within the 'no quibble' range the offer will be accepted. This avoids spending several rounds negotiating over proverbial pennies, (this can be important especially when negotiation needs to be resolved as soon as possible).

⇒ *max.percent*: the fraction of the commodity's perceived value up to which the agent will be willing to continue bidding

To enter these parameters, double click on the `Parameters` field beside the strategy name, this will bring up a window containing a table.

- Now click on the `New` button to create a new entry, then rename the `Key` field to **min.percent** and press <return>. Then edit the `Value` field so that it contains the value **70**.
- Click on the `New` button to create a new entry, rename the entry in the `Key` field to **no.quibble** and press <return>. Then edit the `Value` field so that it contains the value **0.2**.
- Click on the `New` button to create a new entry, rename the entry in the `Key` field to **max.percent** and press <return>. Then edit the `Value` field so that it contains the value **120**.
- Now click on the OK button to commit the changes and dismiss the window.

As there are no constraints applicable to this strategy the process of equipping this agent to buy is complete, and the Agent Co-ordination panel should look like that shown in Figure 3. The next step is to consider how to enable the agent to sell its resources.



Type	Protocol	State
Initiator	Fipa-Contract-Net-Manager	✓
Respondent	Fipa-Contract-Net-Contractor	✓

Mode	Fact Type	Agents	Relations	Strategy	Parameters
✓	ZeusFact			GrowthFunction	min.percent=70 noq...

Attribute	Value
unit_cost	
number	

Figure 3: A screenshot of the Agent Co-ordination Panel after configuring the buying behaviour of the 'OrchardBot'.

Equipping the Agent to Sell

The process by which the agent is equipped to handle incoming bids is very similar to the process that configured it with the ability to make bids. First we specify a protocol that describes the vendor's role in a negotiation dialogue.

- Click on the checkbox beside the entry for **Fipa-Contract-Net-Contractor**.

Now we can define the negotiation strategy that will be used in conjunction with this protocol.

- Ensure that the **Fipa-Contract-Net-Contractor** entry has been selected in the upper 'Co-ordination Protocols' table.

In the lower 'Co-ordination Strategies' table the `Mode` field will contain a ticked checkbox to indicate that this strategy will be used in conjunction with facts of the type described in the `Fact Type` field. As this field contains ZeusFact this strategy will be used to sell facts of any type.

Our specification does not suggest that the agent will use specialised strategies for trading with certain individuals, so the `Agents` and `Relations` fields can be left unchanged.

- Now double-click on the `Strategy` field and select the entry for **DecayFunction** from the list of available strategies.

Now we can define the parameters that determine how the DecayFunction strategy runs, (these are described in the Technical Manual):

⇒ *min.percent*: the vendor's reserve price, the smallest fraction of the commodity's perceived value that the agent will consider accepting. This factor will dictate the agent's minimum profit margin, and if it is less than 100% it will consider selling a commodity for less than its perceived worth.

⇒ *no.quibble*: the difference in price considered insignificant by the agent; when a bid price is lower than the asking price but within the 'no quibble' range the bid will be accepted. Like the corresponding factor in bidding strategies this avoids spending several rounds negotiating over trivial sums of money.

⇒ *max.percent*: the fraction of the commodity's perceived value at which the agent will offer it for sale

To enter these parameters, double click on the `Parameters` field beside the strategy name, this will bring up a window containing a table.

- Now click on the `New` button to create a new entry, then rename the `Key` field to **min.percent** and press <return>. Then edit the `Value` field so that it contains the value **110**.
- Click on the `New` button to create a new entry, rename the entry in the `Key` field to **no.quibble** and press <return>. Then edit the `Value` field so that it contains the value **0.3**.
- Click on the `New` button to create a new entry, rename the entry in the `Key` field to **max.percent** and press <return>. Then edit the `Value` field so that it contains the value **125**.
- Now click on the OK button to commit the changes and dismiss the window.

As there are no constraints applicable to this strategy the process of equipping this agent to sell is complete. In fact, the definition of the entire 'OrchardBot' agent is now complete and can be saved by clicking on the **Save** icon. Now close the Agent Editor and return to the Agent Generator window, where we can begin the next activity: creating the 'SupplyBot' agent.

Creating the SupplyBot Agent

There are two ways of creating this agent: one is to create a completely new agent, and then configure it by following the same process used to create the 'OrchardBot' agent. However, as the 'SupplyBot' and 'OrchardBot' agents share so much functionality there is a simpler alternative, to clone the existing 'OrchardBot' agent and configure the new agent accordingly. The agent cloning process is described in Section 3 of the Realisation Guide, and summarised below.

- Select the 'OrchardBot' entry in the Known Agents table and then click on the `Clone Agent` button.

A new entry will now appear in the table using a temporary name like 'OrchardBot$0'.

- Now select the new entry and click on the `Rename` button and choose the Rename Agent option. The Agent name will now be editable, double click on it and change it to **SupplyBot**, pressing <return> when finished.

Looking back at our initial specification, the only difference the 'SupplyBot' and 'OrchardBot' agents is the resources they possess. Hence we will need to edit the SupplyBot agent's initial resources.

- Double click on the SupplyBot entry to open the Agent Editor.

Now we can edit the contents of the Initial Resources table that have been copied over from OrchardBot, (and thus change the resources held by the SupplyBot).

- Click on the **appleStock** entry and change the entry in **number**'s `Value` field from 100 to **30**.
- Click on the **orangeStock** entry and change the entry in **number**'s `Value` field from 80 to **30**.
- Click on the **pearStock** entry and change the entry in **number**'s `Value` field from 60 to **10**.
- Click on the **melonStock** entry and change the entry in **number**'s `Value` field from 0 to **20**.

- Click on the **bananaStock** entry and change the entry in **number**'s `Value` field from 0 to **20**.
- Click on the **cash** entry and change the entry in **number**'s `Value` field from 0 to **100**.

At this point it is also possible to change the SupplyBot's fruit valuations, the process by which the values are configured should be obvious by now.

The next aspect that needs to be changed is this agent's minimum profit margin (which is slightly lower than that of OrchardBot), so move to the Agent Co-ordination panel.

- Select the entry for **Fipa-Contract-Net-Contractor** in the upper protocol table.
- Now double click on the `Parameters` field, to open the Parameter entry window. Then double click on the `Value` field of the **min.percent** entry and change its contents from 110 to **105**.

The definition of the SupplyBot agent is now complete and can be saved by clicking on the Save icon. Now close the Agent Editor and return to the Agent Generator window.

Creating the ShopBot Agent

Again we will create this agent by cloning an existing agent, in this case 'SupplyBot'. This process is summarised below.

- Select the 'SupplyBot' entry and then click on the `Clone Agent` button.

A new entry will now appear in the table using a temporary name like 'SupplyBot$1'.

- Now select the new entry, click on the `Rename` button and choose the Rename Agent option. The Agent name will now be editable, double click on it and change it to **ShopBot**, pressing <return> when finished.

As the only difference between the SupplyBot and ShopBot agents is in the resources they begin with, we will need to edit the ShopBot agent's initial resources.

- Double click on the SupplyBot entry to open the Agent Editor.
- Click on the **appleStock** entry and change the entry in **number**'s `Value` field to **0**.
- Click on the **orangeStock** entry and change the entry in **number**'s `Value` field to **0**.

Alternatively, we could have just deleted these two resource entries, but as we assume that the ShopBot will add to its stock of apples and oranges it makes sense to leave entries for these resources so we do not have to manually create them at runtime.

- Click on the **pearStock** entry and change the entry in **number**'s `Value` field to **5**.
- Click on the **melonStock** entry and change the entry in **number**'s `Value` field to **10**.
- Click on the **bananaStock** entry and change the entry in **number**'s `Value` field to **10**.
- Click on the **cash** entry and change the entry in **number**'s `Value` field to **500**.

The definition of the ShopBot agent is now complete and can be saved by clicking on the Save icon. Now close the Agent Editor and return to the Agent Generator window.

With all the task agents specified we can turn our attention to the application's utility agents.

4.3 Utility Agent Configuration

During this phase we can decide what utility agents will be created, and configure them accordingly. So, from the main Agent Generator window click on the button to open the Code Generator tool, this will display the Generation Plan: the agents that will be created when the code generation process is started. You will notice that as well as the OrchardBot, SupplyBot and ShopBot agents there are a Name Server, a Facilitator and a Visualiser. As this matches this application's requirements no new utility agents need to be added.

You may however want to configure the runtime parameters of these agents, this is achieved through the tables of the 'Utility Agents' tab pane, click on it now.

- Starting with the Agent Name Server, change its identifying name to **ANS**.

The I.P address in the `Host` field will default to the machine that is currently running the Agent Generator, which we shall assume to be where the Name Server will be running.

As there is only one Name Server, it is marked as the root, and provides the value of the application time-grain. This can be changed, but the default value is adequate for our application.

- For simplicity we shall assume that all the agents will be run from the same directory, thus the `Address File` does not need a pathname, just a filename. If this field is empty, edit it and enter **ns.db**, this provides the name of the file into which the name server will write its location.

Next we consider the Facilitator.

- First change its identifying name to **Broker.**

We shall assume the Facilitator will be running on the same host machine as the Name Server, so its `Host` field does not need to be changed.

The `DNS File` field contains the path and name of the file containing the Name Server's location. This field should contain the same contents as the Name Server's `Address File` field.

Looking at the design for the *Broker* agent we can see there is a requirement to reconfigure the Facilitator so that it is reactive rather than proactive.

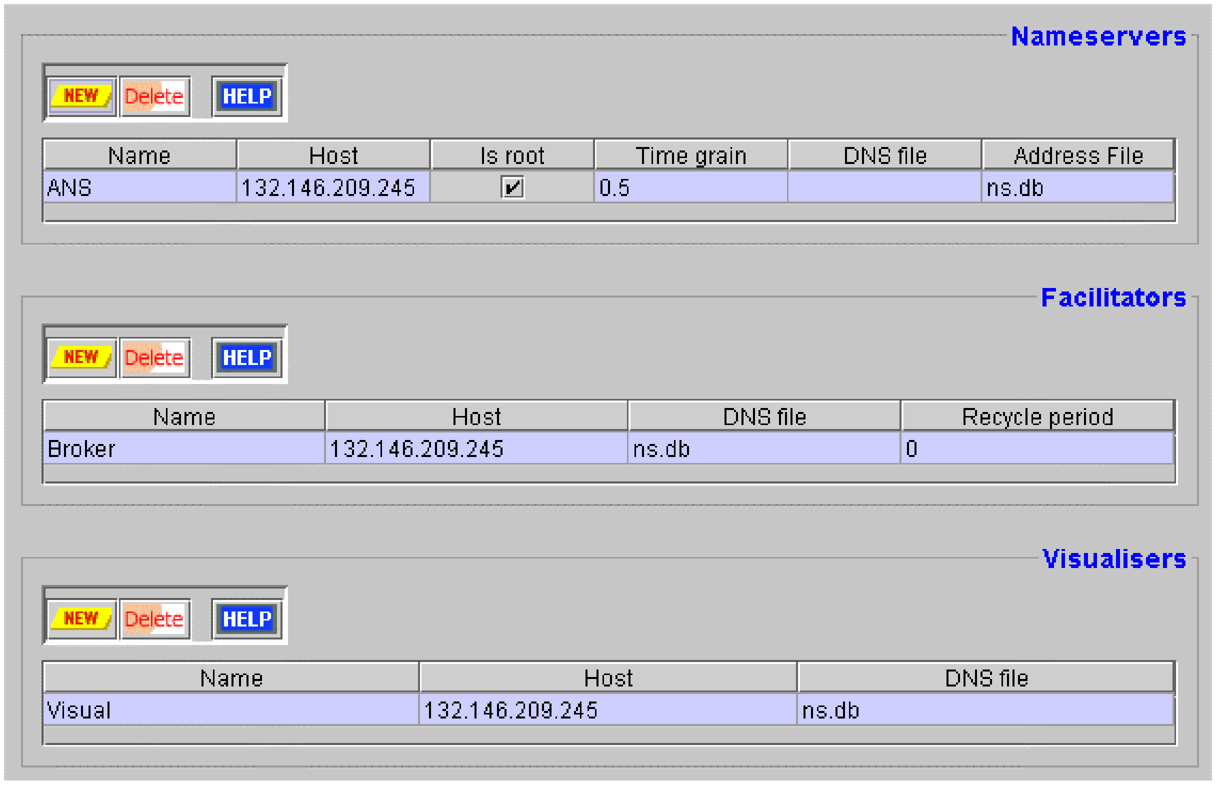
- Change the `Recycle Period` field of the Facilitator to 0.

Finally, we consider the Visualiser.

- Change its identifying name to **Visual**.

Whilst the host field can remain unchanged, and the `DNS File` field should contain the same contents as the Name Server's `Address File` field.

This concludes the configuration of the utility agents, and the Utility Agent panel should look like that shown in Figure 4. Next, we turn our attention to the task agents.



Name	Host	Is root	Time grain	DNS file	Address File
ANS	132.146.209.245	✓	0.5		ns.db

Name	Host	DNS file	Recycle period
Broker	132.146.209.245	ns.db	0

Name	Host	DNS file
Visual	132.146.209.245	ns.db

Figure 4: A screenshot following the specification of the utility agents for the *FruitMarket* application

4.4 Task Agent Configuration

This process is described in detail in Section 5 of the Realisation Guide, and performed through the Task Agents tab pane of the Code Generator tool, move to it now.

First, we will consider the 'OrchardBot' agent. If we assuming that it will run on the same host machine as the utility agents the contents of its `Host` field can be left changed. Likewise the `DNS File` field should contain the same contents as the Name Server's `Address File` field.

According to our design the resources of the *Trader* agents can be represented internally, and do not need to be obtained from external databases. Hence the `Database External` field can be left blank for each agent.

However, our design does indicate that each *Trader* agent will have a user interface, so this should be implemented in a separate class and identified in the `External Program` field.

Important note: the class names entered during this stage must be accessible to the Java runtime environment's class-loader. This may not be necessary if you enter **.** (the current directory) into your local CLASSPATH environment variable.

- Double click on the OrchardBot entry's `External Program` field and enter **orchardUI**

As this is a pedagogical example a UI that displays the agent's activity would be useful.

- Clicking the checkbox in the `Create GUI?` field will create an activity display for that agent when the code is generated.

Finally, you can choose what icon will represent the agent when it is visualised.

- Double click in the `Icon` field, this will open a file dialogue. Move to the 'zeus/examples/fruitmarket/gifs' directory and choose the file named **tree.gif**.

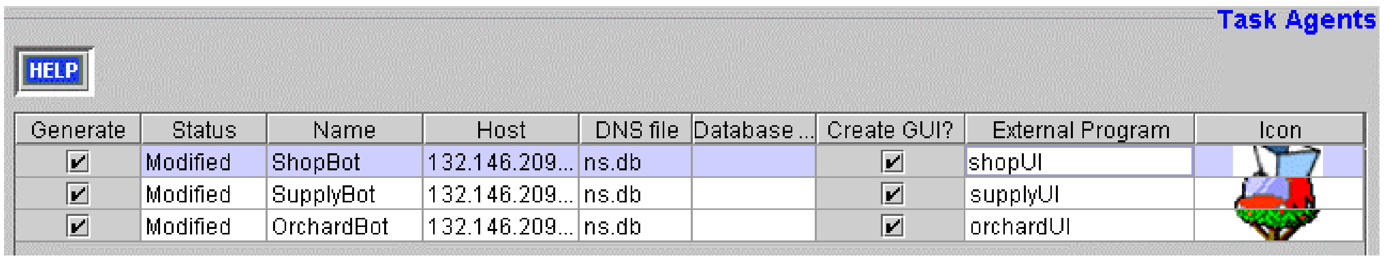
Next, we repeat the above steps for the SupplyBot agent.

- Double click on the SupplyBot entry's `External Program` field and enter **supplyUI**
- Check the checkbox in the `Create GUI?` field
- Double click in the SupplyBot's `Icon` field, and choose the file named **van.gif**.

Finally we configure the ShopBot agent.

- Double click on the ShopBot entry's `External Program` field and enter **shopUI**
- Check the checkbox in the `Create GUI?` field
- Double click in the ShopBot's `Icon` field, and choose the file named **cart.gif**.

This concludes the configuration of the task agents, and the Task Agent panel should look like that shown in Figure 5. Next, we shall generate the Java source code for our application.



Generate	Status	Name	Host	DNS file	Database ...	Create GUI?	External Program	Icon
✓	Modified	ShopBot	132.146.209...	ns.db		✓	shopUI	
✓	Modified	SupplyBot	132.146.209...	ns.db		✓	supplyUI	
✓	Modified	OrchardBot	132.146.209...	ns.db		✓	orchardUI	

Figure 5: A screenshot following the specification of the task agents of the *FruitMarket* application

4.5 Code Generation and Implementation

With all agents described we are ready to generate their implementations, this is described in detail in Section 6 of the Realisation Guide, and performed from the 'Generation Plan' tab pane of the Code Generator tool, move to it now.

First, choose the directory into which the source code will be written. As your recently defined agents are likely to be identical to those of the pre-written FruitMarket application you could choose to write the code into the zeus/examples/fruitmarket directory, replacing the existing agent definitions (existing files are never over-written by the Generator, rather they are renamed with a % postfix).

- Click on the `Target Directory` button and choose an appropriate directory, or type the full directory path into the field beside it.

Next, choose the operating system for which the agent launch scripts will be written.

- Click on either the `Windows` or `Unix` radio buttons as appropriate.
- Finally, click on the `Generate` button.

This will create Java implementations for each agent class listed in the 'Generation Plan' table. You will find the source code files in the directory you specified earlier. With the code now generated it is time to implement the application specific components suggested by our design (these are prefixed with the **IMPL** code). This reveals the following activity still needs to be performed:

- `Add Buying and Selling User Interfaces`
  ⇒ `See activity` **`IMPL-4A`**`: Implementing External Programs`

Implementing the External GUIs

Each agent requires a user interface to facilitate the entry of its owner's buying and selling preferences. Rather than describe how to implement these files, they have been pre-written and included as part of the ZEUS package (in the zeus/examples/fruitmarket directory). In keeping with the names defined in Section 4.4 these files are:

• orchardUI.java	- user interface implementation for the OrchardBot agent
• supplyUI.java	- user interface implementation for the SupplyBot agent
• shopUI.java	- user interface implementation for the ShopBot agent

As activity **IMPL-4A** explains in the Realisation Guide, the purpose of these files is to provide an interface between the class that implements each agent and the class that implements the user interface. This is achieved by passing an AgentContext object that contains references to each of the agent's internal components to the user interface class, as shown below:

SOURCE CODE

```
import zeus.actors.*;
import zeus.agents.*;


public class orchardUI implements ZeusExternal
{
   public void exec(AgentContext agent)
   {
     TraderFrontEnd thiswin = new TraderFrontEnd(agent);
   }

   public void showMsg(String message) {}

   public static void main(String[] args)
   {
     orchardUI win = new orchardUI();
     win.exec(null);
   }
}
```

User Interface instantiated with AgentContext

Of course, an alternative arrangement would be for the class implementing the GUI (TraderFrontEnd) to implement the ZeusExternal class, so having access to the AgentContext directly.

Acting on User Instructions

If you look at the file TraderFrontEnd.java, most of the user interface should be self-explanatory. One aspect that deserves some explanation is the code that obtains the user instructions from the interface and uses them to change the behaviour of the agent. For instance, when the Trade button on the Buying panel is pressed the following code is called:

SOURCE CODE

```
public void actionPerformed(ActionEvent evt)
{
  if (evt.getSource() == tradeBtn)
  {
    // trade button pressed: do some error checking first
    String fruit = fruitField.getText();
    if (!UI.agent.OntologyDb().hasFact(fruit))
      return;

    Fact[] tmp = UI.agent.ResourceDb().all(fruit);
    if (!buying && tmp[0].getNumber() == 0)
    {
      UI.display("Ooops, we're out of " + fruit + "s !");
      return;
    }

    // transaction is valid: commence trading
    Long askl  = askPriceField.getValue();
    Long timel = timeField.getValue();
    int ask  = (int)askl.longValue();
    int time = (int)timel.longValue();
    if (buying)
      beginBuy(fruit, ask, time);
    else
      beginSell(fruit, ask, time);
  }
}


public void beginBuy(String fruit, double ask, int time)
{
  UI.display("\nAttempting to buy: " + fruit);
  Fact fact = UI.agent.OntologyDb().getFact(Fact.VARIABLE, fruit);
  fact.setNumber(1);
  int now = (int)UI.agent.now();
  Goal g = new Goal(UI.agent.newId("goal"), fact, now+time, ask,
                    UI.agent.whoami(), (double)(now+time-0.5));

  UI.agent.Engine().buy(g);
}
```

ensures selected fruit is present in ontology

if selling, checks stocks

gets user trading parameters from GUI

create new goal using ontology definition and user parameters

pass goal to agent as new buying activity

From this we can see the process of getting an agent to bid for an item of fruit is relatively simple. After checking the desired fruit exists in the ontology, a new fact representing the desired item is created and then used together with the user's buying preferences to create a new goal. Finally the goal is sent to the agent's Co-ordination Engine as a new buying activity, this will cause the agent to begin a conversation using the 'Fipa-Contract-Net-Manager' protocol that we equipped the agent with earlier.

The procedure for triggering selling behaviour is very similar, with a new goal being created to represent the commodity being sold, and which is passed to the sell() method of the agent's Co-ordination Engine.

Monitoring Agent Events

Another useful feature of the TraderFrontEnd is to keep the user informed of ongoing trading activities, like the arrival of new bids and the responses made by sellers. This is achieved by implementing the ConversationListener interface; the method that listens for the beginning of a conversation is shown below:

SOURCE CODE

```
public void conversationInitiatedEvent(ConversationEvent event)
{
  String correspondent = event.getReceiver();
  String mode = " to ";
  if (correspondent.equals(agent.whoami()))
  {
    correspondent = event.getSender();
    mode = " from ";
  }
  Goal g = (Goal)(event.getData().elementAt(0));
  double cost = g.getCost();
  String f = g.getFactType();
  display("Conversation started...");
  display("[" + event.getMessageType() + "]" + mode +
          correspondent + " >> " + f + " @ " + cost);
}


public void conversationContinuedEvent(ConversationEvent event)
{
  .
  .
}
```

determines the conversation event's origin

obtains the subject of the conversation

displays selected information about the current conversation

The other method of the interface, conversationContinuedEvent(), is practically identical, although it will be triggered under slightly different circumstances (when a message is sent during an existing conversation).

Monitoring Changes to Resources

As well as receiving and acting on instructions, the TraderFrontEnd needs to present accurate information on the state of its agent's resources. In this example the changes in question involve stock arriving and being dispatched, and money being credited or debited. To cause a change, press one of the "Supply" buttons that appear beside the stock level of each commodity, this simulates the delivery of one item of that commodity by adding a new resource to the agent's own Resource Database (ResourceDb). This will trigger a *factAddedEvent*, which can be listened for by the TraderFrontEnd; in addition the *factDeletedEvent* will notify listeners that a particular resource has been removed and no longer exists locally. The following line registers the TraderFrontEnd's interest in these events:

```
UI.agent.ResourceDb().addFactMonitor(this,
  FactEvent.ADD_MASK|FactEvent.DELETE_MASK, true);
```

The excerpt of code illustrating the factAddedEvent is relatively simple, it retrieves the fact reference from the trigger event and passes it to a method that updates the appropriate GUI component. Changes to Money resources are slightly more complex, as credit transactions involve the creation of new separate Money resources these must be merged with the agent's existing Money resource to create a true figure for the amount of money owned.

SOURCE CODE

```
public void factAddedEvent(FactEvent event)  {
  Fact fact = event.getFact();
  if ( fact.getType().equals(OntologyDb.MONEY) ) {
    Fact[] tmp = UI.agent.ResourceDb().all(OntologyDb.MONEY);
    double cash = 0;
    for(int i = 0; i < tmp.length; i++ ) {
      tmp[i].resolve(new Bindings());
      PrimitiveNumericFn numFn =
        (PrimitiveNumericFn)tmp[i].getFn(OntologyDb.AMOUNT);
      cash += numFn.doubleValue();
    }
    cashLabel.setText("In Bank: " + Misc.decimalPlaces(cash,2));
  }
  else
    update(fact);
}


protected void update(Fact f)
{
  Fact[] tmp = UI.agent.ResourceDb().all(f.getType());
  int num = 0;
  for(int i = 0; i < tmp.length; i++ )
     num += tmp[i].getNumber();

  if (f.getType().equals("apple"))
    appleLabel.setText("Boxes of apples in stock: " + num);
  else if (f.getType().equals("orange"))
    orangeLabel.setText("Boxes of oranges in stock: " + num);
  else if (f.getType().equals("pear"))
    pearLabel.setText("Boxes of pears in stock: " + num);
  else if (f.getType().equals("banana"))
    bananaLabel.setText("Boxes of bananas in stock: " + num);
  else if (f.getType().equals("melon"))
    melonLabel.setText("Boxes of melons in stock: " + num);
}
```

merges all money resources

calls method to update display

uses type of changed fact to determine which field to update

If you are building the FruitMarket application from scratch now is the time to compile the generated source code and the external program implementations. Assuming no errors are reported, the application can now be run using the agent launching scripts created by the Generator tool at the same time as the agents. How to run agent applications is explained in the ZEUS Runtime Guide, and is outlined during the next section.

5 Running the FruitMarket Application

Assuming that all the agents will be running on the same machine, launching the application will involve the following process:

- Enter the command **'run1'**, this will execute the run1 script and start the Agent Name Server (ANS). No errors should be reported, and a new Java interpreter process should be running in the background. If a problem has occurred, check your system's configuration (e.g. CLASSPATH setting, install directory in the .zeus.prp file etc.)
- Enter the command **'run2'**, this will start the OrchardBot, SupplyBot and ShopBot agents, as a result three more Java processes should now be running in the background. Whilst on screen an Agent Viewer window and a Trading GUI window will appear for each agent.
- Enter the command **'run3'**, this will start the Visualiser and Facilitator agents. Two additional Java processes should be started, and a Visualiser Hub window should appear on screen.

Although the agents can run independently of the Visualiser, you may find it useful to launch the Society Viewer window to observe how the agents interact with each other. The Statistics tool may also be useful to plot price fluctuations over time, whereas for example the Report Tool has no relevance as no tasks are actually performed by the agents. The ZEUS Runtime Guide describes how to use the various facilities of the Visualiser tool.

The instructions in the ZEUS Runtime Guide for using the Visualiser tools apply to any multi-agent application. Hence the remainder of this section will only consider the application-specific features of the FruitMarket example.

Once launched the FruitMarket agents will display their front-end GUIs and await user instructions; they will do nothing autonomously apart from replying to register themselves with the Name Server, (and the Visualiser if one exists). Now users have two options, they can offer an item for sale or formulate a new bid.

Offering an Item for Sale

The procedure for tendering goods is relatively simple:

- Go to the "Sell Fruit" tab pane.
- Now choose the commodity to be sold either by typing its name into the **Sell** textfield or clicking on the **Choose** button and selecting it from the current ontology hierarchy.
- Next enter the reserve price of item (this is the lowest price you are willing to accept) in the **Reserve Price** field. This value will be kept secret and used by the negotiation strategy to create an asking price, which will be the price quoted to potential buyers.
- The elapsed time before the sale should be completed can be entered in the **Deadline** field, this sets the period of time that potential buyers have to submit bids and negotiate.
- Finally once the selling parameters have been entered click on the **Trade** button: this will notify the broker of the item for sale, and thus any interested parties.

Once offered for sale the agent waits for incoming bids until the deadline period expires, whereupon the item will be withdrawn from sale. How the agent responds to incoming bids is described later in this section.

Bidding for an Item

The procedure for bidding for goods is quite similar to that for selling:

- Go to the "Buy Fruit" tab pane.
- Now choose the commodity to purchase either by typing its name into the **Buy** textfield or clicking on the **Choose** button and selecting it from the current ontology hierarchy.
- Next enter the highest price you would be willing to pay for this item in the **Maximum Price** field. This value will be kept secret and used by the negotiation strategy to create a bid price, which will offered to the seller.
- The elapsed time before the purchase should be completed can be entered in the **Deadline** field, this sets the period of time by which negotiation should have concluded.
- Finally once the bidding parameters have been entered click on the **Trade** button: the agent will then consult the broker for the presence of matching items for sale.

If the commodity to buy is currently on the market the broker will send the interested bidder the identity of the seller, which it can contact directly and begin negotiation. Otherwise the bidding agent will ask to be informed if any appropriate items are placed on the market. Hence the order in which items are placed on the market is not significant – bidders can bid before sellers offer and vice versa, providing they do so before each other's deadline period expires.

5.2 How Negotiation Works

As explained in Section 3.4 of the ZEUS Realisation Guide, the interactions of the agents are governed by strategies. In the FruitMarket application sellers to determine their replies to incoming bids using the LinearInitiatorEvaluator strategy, whilst LinearRespondentEvaluator is used by potential purchasers to formulate their bids. These strategies are implemented in the eponymous files found in the zeus.actors.graphs package.

These strategies are linked to the agent's co-ordination engine by their implementation of the *StrategyEvaluator* interface class, and two significant methods: evaluateFirst() and evaluateNext(). The former method is used to calculate the initial negotiating position, whilst the latter uses the parameters set when the agent was created (as described in the COORD-2 section of the Realisation Guide), or runtime parameters (like those entered through the agent GUI). Sample implementations of these methods (from the LinearInitiatorEvaluator.java file) are shown next:

SOURCE CODE

```
public int evaluateFirst(Vector goals, ProtocolDbResult info) {
    this.goals = goals;
    this.protocolInfo = info;
    min       = getDoubleParam("min.percent",80)/100.0;
    max       = getDoubleParam("max.percent",120)/100.0;
    noquibble = getDoubleParam("noquibble.range",2);

    default_step =getDoubleParam("step.default",0.2);
    reserve_price=getDoubleParam("reservation.price",Double.MAX_VALUE);

    Goal g = (Goal)goals.elementAt(0);
    expected_cost = g.getCost();

    start_time = context.now();
    end_time = g.getReplyTime().getTime();

    g.setCost(0);   // hide true cost from bidders
    return MESSAGE;
}
```

the parameters entered at generation time

initialise the cost and time parameters

The evaluateFirst() method is intended to initialise negotiation rather than implement it: this is the function of the evaluateNext() method, as shown below:

SOURCE CODE

```
public int evaluateFirst(Vector goals, ProtocolDbResult info) {
    this.goals = ds.goals;
    if ( !ds.msg_type.equals("propose") )
      return FAIL;

    Goal g = (Goal)ds.goals.elementAt(0);
    offer = g.getCost();
    double now = context.now();

    if ( first_response ) {
       first_response = false;
       dt = now - start_time;
       end_time = end_time - dt;

       expected_cost = Math.max(offer,expected_cost);
       min_price = min*Math.min(reserve_price,expected_cost);
       max_price = Math.min(max*expected_cost,reserve_price);

       price = min_price;
       step = (max_price-min_price)*dt/(end_time-start_time);
       step = Math.max(step,default_step);
    }

    if ( offer < price + noquibble ) return OK;
    if ( !first_response )           price += step;

    if ( price < min_price )
      return FAIL;
    else if ( offer < price + noquibble )
      return OK;

    if ( now + dt >= end_time ) {
      if ( offer < max_price )
        return OK;
      else
        return FAIL;
    }

    g.setCost(price);
    return MESSAGE;
}
```

determine asking price and increment value

revise asking price slightly

running out of time, attempt to settle

The important fact to note about the evaluateNext() method is that it provides an interface between the agent and the code implementing its negotiation expertise. There is no need for expertise to be encoded here in the form of procedural Java code, indeed a more complex application might use this method to link the agent to an external program like a Case Reasoning Engine.

Typically the evaluateNext() method modifies the negotiation position in the light of new information, either in the form of new bids, or the passage of time bringing the deadline closer. It functions by changing the attributes of the current goal, and determines the next state of the agent's co-ordination engine by returning one of the following three types of message:

- OK - i.e. the bid or proposal was acceptable, agents now move into the settlement stage
- MESSAGE - i.e. the bid was not acceptable but close enough to warrant a counter-proposal
- FAIL - this ends the negotiation, typically because the bid was too low and the time deadline has expired

If the negotiation results in a price acceptable to both parties the co-ordination engine of each agent will move into the settlement stage. Here the resource in question is removed from the seller's ResourceDb and transferred to the buyer, whilst simultaneously the money resource of the buyer is debited and the amount credited to the seller. This process is part of the standard ZEUS agent interaction behaviour and so it does not need to be implemented by the user.

Try This…

In the FruitMarket example, negotiation will only occur when offers and bids are close enough to each other (remember how the seller was configured to ask for between 103 and 120% of its reserve price and the buyer to bid between 70 and 120% of its valuation). Try selling a box of apples at a reserve price of 8 to an agent willing to go as high as 12 and you will see negotiation in progress.

A brief walk-through of some of the FruitMarket features can be found in the readme.txt file that is located in the same directory as the example's source code.

Concluding Remarks

The best way to understand the FruitMarket application, and thus the ZEUS components from which it is built, is to experiment and explore. For instance, see where the application specific code (the Java code in the fruitmarket directory) makes calls to the ZEUS classes. The interfaces between the trading application and the ZEUS components reveal how the latter can be configured in what is probably the most succinct way possible.

In the meantime any errors, comments or suggestions are welcomed.

Jaron Collis (jaron@info.bt.co.uk)