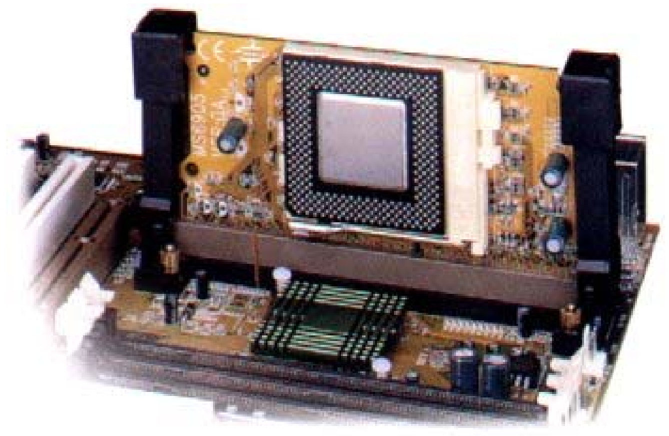
The Zeus Agent Building Toolkit

ZEUS Methodology Documentation Part II

Case Study 2

PC Manufacture

A Supply Chain Simulation



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

Intelligent Systems Research Group, BT Labs

Release 1.01, November 1999

Index

Document History

Version 1.01 - Added Index; updated section 5

1 Specification

This case study describes the implementation of an application that simulates the manufacture of Personal Computers (PCs). Building a PC tends to involve the acquisition and integration or many components, usually from different manufacturers, thus providing an excellent example of a supply chain. This scenario will illustrate how certain primitive components must be acquired before the more complex components of which they are a part are assembled.

For simplicity, we shall assume a very simple model of PCs, namely that they consist of a keyboard a monitor, a CPU (motherboard), and either a laser or inject printer. Furthermore, we shall assume that each type of component is a commodity, i.e. there is no inherent difference between two instances of the same item.

A key difference between this example and the FruitMarket application is that the participants in this example will be capable of performing specific tasks. The participants and their abilities are described next:

The first participant is called **Dell**[1], it represents a computer manufacturer that buys the necessary components for building a PC and assembles them. This suggests a single task, which we shall call **MakeComputer**. Its initial resources are 2000 US-style keyboards and 2000 UK-style keyboards.

The second participant is **Taxan**, a representative of a monitor manufacturer. We shall ignore its suppliers and assume it has all the resources necessary to build monitors without having to collaborate, we shall call this task **MakeMonitor**. It has initial stocks of 2000 CRTs (Cathode Ray Tubes), 2000 SVGA and 2000 VGA adapter cards.

Another participant is **Intel**, a representative of an integrated circuit manufacturer who supplies fully populated motherboards, complete with CPUs of various speeds. We shall ignore its suppliers and assume it can produce motherboards without further collaboration through a task called **MakeMotherBoard**. It starts with stock of 1000 each of the 3 speeds of CPU: 300, 350 and 400 MHz.

The next participant is **HP**, a representative of a printer manufacturer. To make this scenario more interesting we shall assume it has subcontracted the production of the printer cartridges, and performs a single task called **MakePrinter**. Its initial resources will be 2000 ink jet printer cases, and 2000 laser print cases.

The final participant is **CartridgeCorp**, the manufacturer of ink and laser toner for printers, thus it is capable of two tasks: **MakeInkCartridge** and **MakeTonerCartridge**. It possesses no initial resources.

In this example we shall assume the existence of an internal market, hence the costs of items will be credited and debited, although no pre-set budgets exist. We shall assume all transactions are conducted in Euros (€), using the following baseline for prices:

- Keyboards (US and UK) cost €10
- All types of Graphics Adapters (VGA/SVGA) cost €10
- CRTs cost €60
- CPUs of 300, 350 and 400 MHz cost 25, 30 and 35 respectively
- Inkjet mechanisms and cases cost €150, whilst those for laser printers cost €175

This application also contains some preferred supplier relationships that need to be represented:

- CartridgeCorp and HP collaborate on a non-contractual basis
- Taxan and Intel have a supply contract with Dell

[1] We named this participant after the eponymous PC manufacturer because our group uses Dell machines and we like them a lot. You can probably take a guess at what kind of monitors, CPUs and printers we use too.

About this Case Study

This document can be read in two ways: if the reader is interested in learning how the PC Manufacture 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. Although a brief outline the specification suggests that the key challenge will the production and transfer of resources on demand to those that need them. This would point towards one of the Business Process role models: of which the Role Model Guide currently lists Simple Supply Chain and Contractual Supply Chain.

2.1 Select Role Models

From the initial specification we notice that production and supply is based on need rather than competitive bidding. This would tend to suggest the Simple Supply Chain rather than its contractual variation, whose characterising features are sub-contracting and tendering.

With a particular role model in mind we can begin to associate roles to participants, starting with the standard *Zeus Application* role model from which it inherits. Almost mandatory is an agent in the *Agent Name Server* role to ensure agent location independence. In a simple supply chain brokers are not essential, as suppliers and producers tend to know each other a priori; but it would seem sensible to include it, as it will make it easier to add new agents to the application at a later date.

Next we consider the roles in the Simple Supply Chain role model. An interesting feature of this model is that there are two levels of abstraction: for instance, the Supply Chain (SC) Participant role is actually an aggregation of five simpler roles like Consumer and Supplier. The easiest way to see what roles are played by the domain participants is to draw out the full supply chain and thus determine who are the suppliers, who are the producers, and who do both. This is illustrated below:

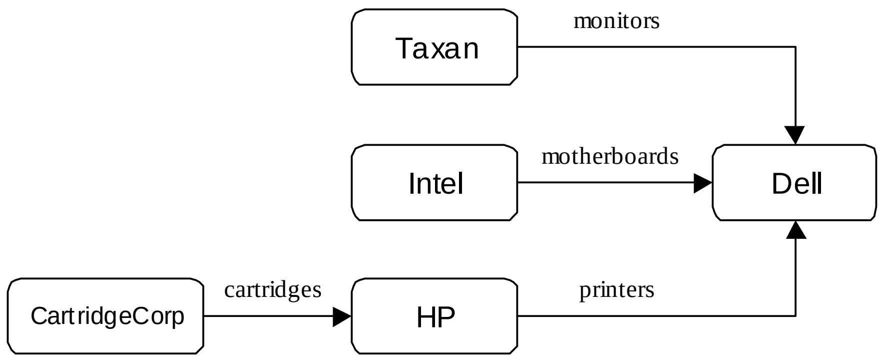


Figure 1: The Participants of the PC Supply Chain, showing resources produced and consumed

From Figure 1 we can see that the Dell company requires resources that it can not manufacture locally. It also does not seem to act as a supplier, hence it seems to best fit the SC-Head role. Whereas both Intel and Taxan supply Dell without relying on external producers, hence they would seem to play SC-Tail roles.

The HP company requires external resources, (printer cartridges), but also acts as a supplier to the Dell company; consequently HP plays the SC-Participant role. Whereas its supplier, CartridgeCorp produces its wares with collaboration making it a SC-Tail.

One aspect to notice about this analysis is that the participants are modelled at the highest level, i.e. HP is encapsulating all of negotiation, production and supply. But if the specification was more detailed we might not be able to justify allocating all the roles to a single agent. For instance, what if we learnt that HP had two offices, one to handle production, and another to handle orders and acquisition, then we might opt for an agent to fill the negotiation roles and another to fill the production and supply roles. In such cases we can use principles such as "Spheres of Responsibility" to justify our analysis decision, (see the guidelines on "What is an Agent?" in the Role Modelling Guide).

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

Agent Name	**Roles Played**
Dell	SC Head (Negotiation Initiator, Consumer)
HP	SC Participant (Negotiation Initiator, Negotiation Partner, Supplier, Producer , Consumer)
Intel	SC Tail (Negotiation Partner, Supplier, Producer)
Taxan	SC Tail (Negotiation Partner, Supplier, Producer)
CartridgeCorp	SC Tail (Negotiation Partner, Supplier, Producer)

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 *Simple Supply Chain* role model we obtain the list of responsibilities for the 2 constituent roles of an agent in the Supply Chain Head role. 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:

SUPPLY CHAIN HEAD – Social Responsibilities	
Origin	**Responsibility**
Negotiation Initiator	To be aware of acquaintances and their capabilities
Negotiation Initiator	To initiate negotiation on the supply of a resource
Negotiation Initiator	To negotiate the terms of supplying a specified resource
Negotiation Initiator	To pass information on resource orders (to own Consumer role)
Consumer	To receive delivered resources

SUPPLY CHAIN HEAD – Domain Responsibilities	
Origin	**Responsibility**
Negotiation Initiator	To initiate demand
Consumer	To consume delivered resource

The next role to consider is the Supply Chain Participant, the role played by those agents who both produce resources and consume resources from others. This is slightly more complicated, as it involves responsibilities from its five constituent roles:

SUPPLY CHAIN PARTICIPANT – Social Responsibilities	
Origin	**Responsibility**
Negotiation Initiator Negotiation Participant	To be aware of acquaintances and their capabilities
Negotiation Initiator	To initiate a request to supply a specified resource
Negotiation Initiator	To negotiate the terms of supplying a specified resource
Negotiation Initiator	To pass information on resource orders (to own Consumer role)
Negotiation Participant	To respond to requests to supply a specified resource
Negotiation Participant	To negotiate the terms of supplying a specified resource
Negotiation Participant	To pass information on resource orders (to own Producer role)
Producer, Supplier	To transfer produced resource (from Producer to Supplier)
Consumer	To receive delivered resources

SUPPLY CHAIN PARTICIPANT – Domain Responsibilities	
Producer	To produce requested resources
Consumer	To consume delivered resource

The third role is the Supply Chain Tail, which is intended for agents that will supply resources without consuming resources that have been produced externally. This involves responsibilities from its constituent three roles:

SUPPLY CHAIN TAIL – Social Responsibilities	
Origin	**Responsibility**
Negotiation Participant	To be aware of acquaintances and their capabilities
Negotiation Participant	To respond to requests to supply a specified resource
Negotiation Participant	To negotiate the terms of supplying a specified resource
Negotiation Participant	To pass information on resource orders (to own Producer role)
Producer, Supplier	To transfer produced resource (from Producer to Supplier)

SUPPLY CHAIN TAIL – Domain Responsibilities	
Producer	To produce requested resources

Now we have a list of the responsibilities of each intended agent the design process can commence.

3 Application Design

The application design process consists of two phases. The first is an agent design phase where the roles assigned to each agent in the previous stage are refined by mapping each of the responsibilities to a generalised problem, and then choosing the most appropriate solution. Once this activity has been performed for each agent we can begin the second phase, the creation of the application ontology.

3.1 Problem Design

By now we have identified 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.

3.1.1 Agents playing the Supply Head Role

In this application there is one agent that will playing the Supply Head role, the Dell Agent. We begin by considering its social responsibilities.

SUPPLY CHAIN HEAD SOCIAL RESPONSIBILITIES

Responsibility:	To be aware of acquaintances and their capabilities `[Optional]`
Origin:	Negotiation Initiator
Problem:	Knowledge [about: Supplier, about: Required Resource]

Solution:	`Provide knowledge about acquaintances (`**`ORG-1`**`) and`
	`Provide knowledge about acquaintance abilities (`**`ORG-2`**`)`

Explanation:	Static relationships between agents need to be specified when the agents are generated. Otherwise the agents can be left to discover each other through the Facilitator at run-time.

Responsibility:	To initiate negotiation on the supply of a resource
Origin:	Negotiation Initiator
Problem:	Initiate Dialogue [with: Supplier, about: Required Resource]
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 negotiate the terms of supplying a specified resource
Origin:	Negotiation Initiator
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. As the Supply Chain Head role is buying a resource, the **GrowthStrategy** is one of the most appropriate. This begins 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 inform local consumer of resource delivery status
Origin:	Negotiation Initiator
Problem:	Sending Message [to: Consumer Role, about: Resource Delivery terms]
Solution:	Automatic – part of default ZEUS Task Agent functionality
Explanation:	Once agreement has been reached on the supply of a resource the agent's planning and scheduling diary is updated accordingly.

Responsibility:	To receive delivered resources
Origin:	Consumer
Problem:	Interface to handle incoming resource

Solution:	`Create Consumption Task (`**`DEF-2`**`) and`
	`Define attributes of Task (`**`TASK-1`**`)`

Explanation:	The negotiation process is driven by the consumer role's need of a particular resource. Hence we need to create a task whose preconditions are the resources that need to be acquired at run-time.

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. This illustrates a general principle of using ZEUS: when a solution is not automatically provided it will need to be realised manually: by configuring the agent using the ZEUS Agent Generator tool.

SUPPLY CHAIN HEAD DOMAIN RESPONSIBILITIES

Responsibility:	To initiate demand
Origin:	Negotiation Initiator
Problem:	Starting the supply chain

Solution:	`Use 'Add Goal' facility of Agent Viewer or`
	`Implement external program (`**`IMPL-4A`**`)`

Explanation:	Supply chains require a trigger to initiate demand. Since resource demand in ZEUS agents is governed by the presence of outstanding goals the solution is to create a new goal for the agent. This can be achieved by either using the New Goal facility of the agent viewer, (which requires no implementation) or by writing an external program that will create goal and pass it to the agent.

Responsibility:	To consume delivered resource `[Optional]`
Origin:	Consumer
Problem:	An implementation that utilises resource
Solution:	`Implement task body (`**`IMPL-2`**`)`
Explanation:	Once the resource has been supplied it can be consumed, as the way in which the resource is consumed is domain-specific it will need to be implemented by the developer. However, resources do not need to be consumed, they can simply be passed along the supply chain - if this is the case no consumption implementation is needed.

The above entries illustrate 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.

3.1.2 Agents playing the Supply Chain Participant Role

This application has only one agent playing the Supply Chain Participant role: the HP agent. We adopt the same approach that led to the design of the Dell agent, considering each responsibility in turn:

SUPPLY CHAIN PARTICIPANT SOCIAL RESPONSIBILITIES

Responsibility:	To be aware of acquaintances `[Optional]`
Origin:	Negotiation Initiator, Negotiation Participant
Problem:	Knowledge [about: Supplier, about: Required Resource]
Solution:	`Provide knowledge about acquaintances (`**`ORG-1`**`) and` `Provide knowledge about acquaintance abilities (`**`ORG-2`**`)`
Explanation:	Static relationships between agents need to be specified when the agents are generated. Otherwise the agents can be left to discover each other through the Facilitator at run-time.

Responsibility:	To initiate negotiation on the supply of a resource
Origin:	Negotiation Initiator
Problem:	Initiate Dialogue [with: Supplier, about: Required Resource]
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. The Negotiation Initiator role implies the use of the **Initiator** version of the FIPA defined Multi-Round Contract Net, (the justification for this is discussed in the Application Realisation Guide).

Responsibility:	To negotiate the terms of supplying a specified resource
Origin:	Negotiation Initiator
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 the Negotiation Initiator role the agent will be acquiring a resource from its subsidiary at cost price, hence the **Default-No-Profit** strategy will avoid needing to haggle with its supplier.

Responsibility:	To initiate negotiation on the supply of a resource
Origin:	Negotiation Participant
Problem:	Initiate Dialogue [with: Supplier, about: Required Resource]
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 **Respondent** version of the FIPA defined Multi-Round Contract Net.

Responsibility:	To negotiate the terms of supplying a specified resource
Origin:	Negotiation Participant
Problem:	Evaluate [Commodity, Price]
Solution:	`Equip agent with negotiation strategies (`**`COORD-2`**`)`
Explanation:	In the Negotiation Participant role this agent will be negotiating on the supply of a resource for profit. Hence the **DecayStrategy** is appropriate, this begins bidding at a low price and gradually increases it is accepted or a certain period of time elapses.

Responsibility:	To inform local producer of resource delivery status
Origin:	Negotiation Participant
Problem:	Sending Message [to: Producer Role, about: Resource Delivery terms]
Solution:	Automatic – part of default ZEUS Task Agent functionality
Explanation:	Once agreement has been reached on the supply of a resource the agent's planning and scheduling diary is updated accordingly.

Responsibility:	To inform local consumer of resource delivery status
Origin:	Negotiation Initiator
Problem:	Sending Message [to: Consumer Role, about: Resource Delivery terms]
Solution:	Automatic – part of default ZEUS Task Agent functionality
Explanation:	Once agreement has been reached on the supply of a resource the agent's planning and scheduling diary is updated accordingly.

Responsibility:	To receive delivered resources
Origin:	Consumer
Problem:	Interface to handle incoming resource
Solution:	`Create Consumption Task (`**`DEF-2`**`) and` `Define attributes of Task (`**`TASK-1`**`)`
Explanation:	The negotiation process is driven by the consumer role's need of a particular resource. Hence we need to create a task whose preconditions are the resources that need to be acquired at run-time.

Responsibility:	Transfer of produced resource (from Producer to Supplier)
Origin:	Producer, Supplier
Problem:	Sending Message [to: Supplier Role, about: Resource availability]
Solution:	Automatic – part of default ZEUS Task Agent functionality
Explanation:	Once a resource has been produced or received it should be made available to the supplier role so it can be transmitted to the next entity in the supply chain.

SUPPLY CHAIN PARTICIPANT DOMAIN RESPONSIBILITIES

Responsibility:	To produce requested resources `[Optional]`
Origin:	Producer
Problem:	An implementation that creates a representation of a resource
Solution:	`Implement task body (`**`IMPL-2`**`)`
Explanation:	The standard behaviour of a task is to create a new resource objects for each of its effects. If the production of a resource involves some domain-specific process, (such as reading from an external database), this should be implemented in the task body by the developer. If no such process is applicable, no implementation is needed.

Responsibility:	To consume delivered resource `[Optional]`
Origin:	Consumer
Problem:	An implementation that utilises resource
Solution:	`Implement task body (`**`IMPL-2`**`)`
Explanation:	Once the resource has been supplied it can be consumed by domain-specific code, or simply be passed along the supply chain - if this is the case no consumption implementation is needed.

3.1.2 Agents playing the Supply Chain Tail Role

Finally we consider the agents playing the Supply Chain Participant role, in this application there are three: Intel, Taxan and CartridgeCorp. As they fulfil the same roles, we do not need to design each individually, although when we come to specify them using the Agent Generator the attributes used will differ.

SUPPLY CHAIN TAIL **SOCIAL RESPONSIBILITIES**

Responsibility:	To be aware of acquaintances and their capabilities `[Optional]`
Origin:	Negotiation Initiator, Negotiation Participant
Problem:	Knowledge [about: Supplier, about: Required Resource]

Solution:	`Provide knowledge about acquaintances (`**`ORG-1`**`) and`
	`Provide knowledge about acquaintance abilities (`**`ORG-2`**`)`

Explanation:	Static relationships between agents need to be specified when the agents are generated. Otherwise the agents can be left to discover each other through the Facilitator at run-time.

Responsibility:	To initiate negotiation on the supply of a resource
Origin:	Negotiation Participant
Problem:	Initiate Dialogue [with: Supplier, about: Required Resource]
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 **Respondent** version of the FIPA defined Multi-Round Contract Net.

Responsibility:	To negotiate the terms of supplying a specified resource
Origin:	Negotiation Participant
Problem:	Evaluate [Commodity, Price]
Solution:	`Equip agent with negotiation strategies (`**`COORD-2`**`)`
Explanation:	As this role involves negotiation about supplying a resource for profit the **DecayStrategy** is appropriate. This begins bidding at a low price and gradually increases it is accepted or a certain period of time elapses.

Responsibility:	To inform local producer of resource delivery status
Origin:	Negotiation Participant
Problem:	Sending Message [to: Producer Role, about: Resource Delivery terms]
Solution:	Automatic – part of default ZEUS Task Agent functionality
Explanation:	Once agreement has been reached on the supply of a resource the agent's planning and scheduling diary is updated accordingly.

Responsibility:	To handle the production of resources
Origin:	Producer
Problem:	Interface to handle produced resources

Solution:	`Create Consumption Task (`**`DEF-2`**`) and`
	`Define attributes of Task (`**`TASK-1`**`)`

Explanation:	Once the agent has agreed to supply a particular resource, it fulfils its commitment by running the task that produces it. Hence we will need to create a task whose effects are the resources that need to be supplied.

Responsibility:	Transfer of produced resource (from Producer to Supplier)
Origin:	Producer, Supplier

Problem:	Sending Message [to: Supplier Role, about: Resource availability]
Solution:	Automatic – part of default ZEUS Task Agent functionality
Explanation:	Once a resource has been produced or received it should be made available to the supplier role so it can be transmitted to the next entity in the supply chain.

Then there are the domain responsibilities:

SUPPLY CHAIN TAIL DOMAIN RESPONSIBILITIES

Responsibility:	To produce requested resources `[Optional]`
Origin:	Producer
Problem:	An implementation that creates a representation of a resource
Solution:	`Implement task body` (**`IMPL-2`**)
Explanation:	The standard behaviour of a task is to create a new resource objects for each of its effects. If the production of a resource involves some domain-specific process, (such as reading from an external database), this should be implemented in the task body by the developer. If no such process is applicable, no implementation is needed.

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

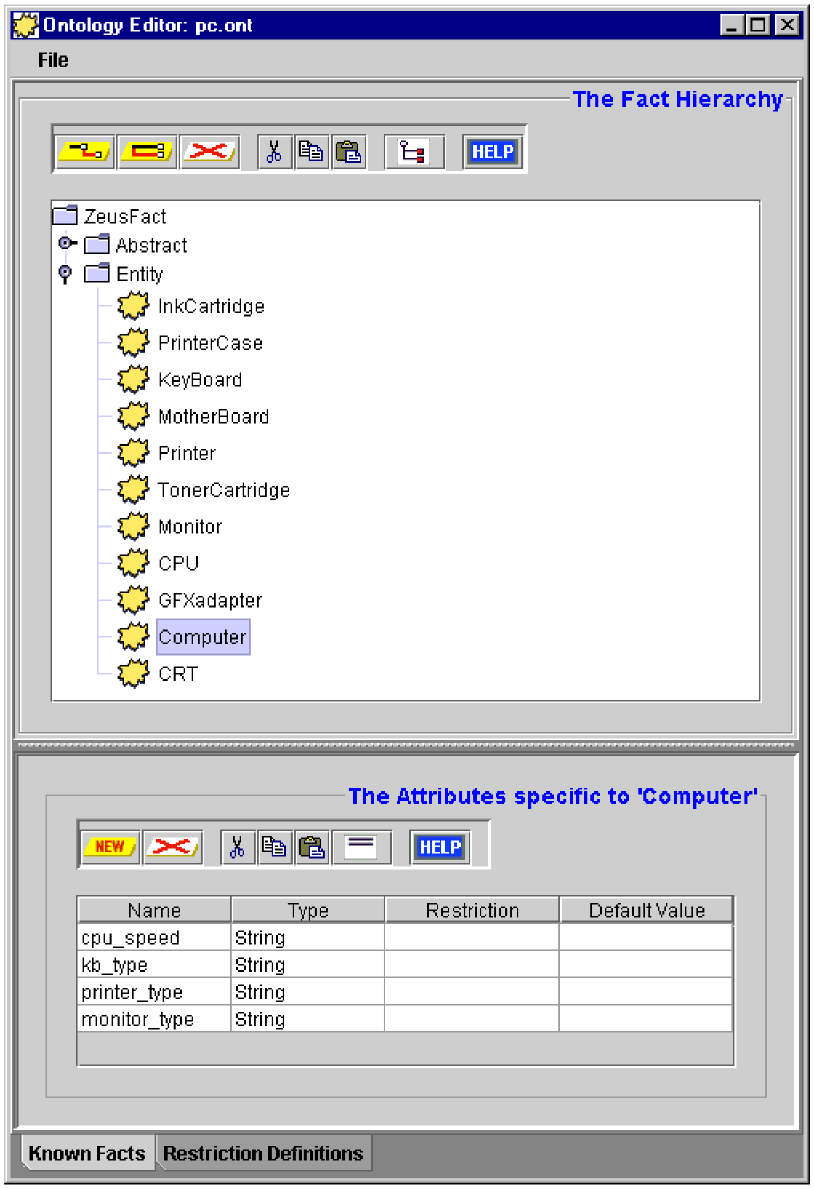
The concepts required for the PC Manufacture application relate to the pieces of hardware required to build a PC. As these are physical instances rather than abstract ones they will inherit from the **Entity** fact, this is a child of the root **ZeusFact** concept that provides all its children with 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 application this can be used to store information about the price of each commodity. The key concepts (and attributes) required are thus:

Fact	Additional Attributes	Constraints
InkCartridge		
TonerCartridge		
PrinterCase	Type : String	
Printer	Type : String	
KeyBoard	Type : String	[UK | US]
MotherBoard	Type : String	
Monitor	Type : String	
CRT		
CPU	Type : String	
GFXadapter	Type : String	
Computer	cpu_speed : String, kb_type : String, printer_type : String, monitor_type : String	

Typing and Constraints

Constraints provide a means of verification, restricting values to a subset of legal values. All physical objects have by default a cardinality constraint (there will always be 0 or more of them). So the table tells us that there is only one additional constraint to define, namely that the type attribute of KeyBoard facts must be either "UK" or "US". Other similar constraints are possible within this application, but will be ignored for the sake of simplicity.

At this stage in the design process we could also assign default values for each concept, but as different agents may attach different valuations there is no need to set these values at design time. Once entered into the Ontology Editor this application's ontology will look looks like that shown in Figure 1.



Name	Type	Restriction	Default Value
cpu_speed	String		
kb_type	String		
printer_type	String		
monitor_type	String		

Figure 1: Screenshot of the Ontology Editor displaying the PC Manufacture ontology

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, i.e. the ZEUS Agent Generator. 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, (as 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). The remainder of this section will walk-through the stages of the ZEUS agent development methodology describing how the PC Manufacture 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 **InkCartridge**; (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 **InkCartridge** node is selected, click on the "Add New Peer Fact" button. Rename the new entry to **PrinterCase**.
- Repeat the above action to create new facts for the following: **TonerCartridge**, **Printer**, **KeyBoard**, **MotherBoard**, **Monitor**, **GFXadapter**, **CRT**, **CPU** and **Computer**.

The contents of the Ontology Editor should now look like that shown in Figure 1. Next we need to add attributes for those facts that need them.

- Select the **PrinterCase** element and then click on the 'Add New Attribute' button in the lower attribute panel. Rename the new entry to **type**, then double click in the Type field and select the **String** entry.
- Repeat the above process for the **Printer**, **KeyBoard**, **MotherBoard**, **Monitor**, **GFXadapter** and **CPU** facts**.**
- Select the **Computer** fact and create four new attributes called **cpu_speed**, **kb_type**, **monitor_type** and **printer_type**; all should be of type **String**.

Finally we need to add the constraints.

- Select the **KeyBoard** fact and in the Restriction field of its **type** attribute enter `UK | US` and then in the default field enter `UK`

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 'pc.ont', (see 'How to Save an Ontology').

The ontology is now complete, so we can leave the Ontology Editor and begin the agent creation process.

4.2 The Agent Creation Process

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').

4.2.1 Creating the Dell Agent

The design for the Dell agent was outlined in section 3.1.1 when we considered how to release the responsibilities of the Supply Chain Head role. This section will step through the activities required to allow the user to become accustomed to using the ZEUS Agent Generator.

- The first step is to create a new agent, by clicking on the 'New Agent' button on the main Generator panel; then rename the new entry that appears in the agent table to **Dell**.

The skeleton of a new generic ZEUS agent now exists inside the Generator. In the phases that follow this generic agent will be configured to satisfy the application-specific responsibilities of its roles.

- Double-click on the 'Dell' 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 *Supply Chain Head*, (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 is one: `Create Consumption Task` (**`DEF-2`**), this involves the following:

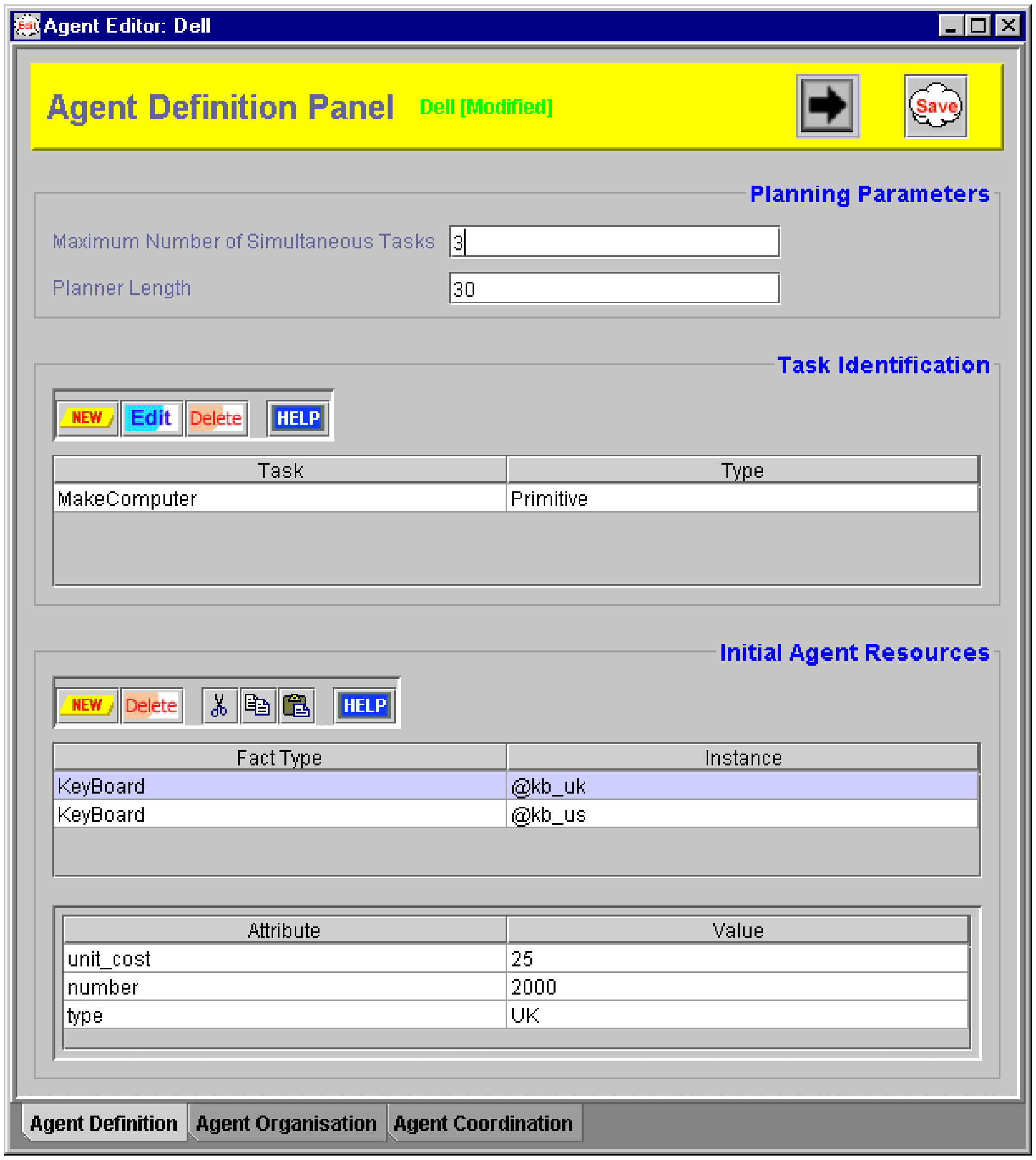
- Click on the 'Create New Task' button in the Task Identification panel, and select the 'New Primitive Task' option.
- A new entry will appear in the table, double click on it to rename it to **MakeComputer**.

Next we need to specify the initial resources of the **Dell** 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 **KeyBoard** as the fact type from the fact hierarchy.
- This will represent the amount of UK keyboards in stock, so double-click the new entry's `Instance` field, and rename it to something more comprehensible, like **kb_uk**.
- To set this agent's valuation of UK keyboards double click on the `Value` field of the **unit_cost** entry in the attribute table and change it to **25**.
- To set the number of UK keyboards held by the agent, double click on the `Value` field of the **number** entry in the attribute table, change it to **2000**.
- Now click on the "New Initial Resource" button again, and select **KeyBoard** as the fact type.
- This will represent the amount of US keyboards in stock, so double-click the new entry's `Instance` field, and rename it to something more comprehensible, like **kb_uk**.
- You will notice that the type field has defaulted to UK (the default for KeyBoard facts), edit this field and change it **US**.

- To set this agent's valuation of US keyboards double click on the `Value` field of the **unit_cost** entry in the attribute table and change it to **25**.
- To set the number of UK keyboards held by the agent, double click on the `Value` field of the **number** entry in the attribute table, change it to **2000**.

This concludes the Agent Definition process, by now the contents of the Agent Editor window should look like those in Figure 2. You should now save the agent definition by clicking on the Save icon.



Maximum Number of Simultaneous Tasks: 3
Planner Length: 30

Task	Type
MakeComputer	Primitive

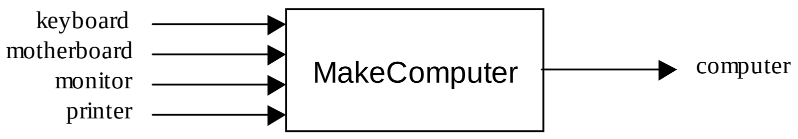
Fact Type	Instance
KeyBoard	@kb_uk
KeyBoard	@kb_us

Attribute	Value
unit_cost	25
number	2000
type	UK

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

The Task Description Process

During the Agent Definition phase we only specified the details of the **MakeComputer** task, not its details. Instead, the details are described during this phase in terms of the task's preconditions, effects and constraints. In terms of resources produced and consumed the MakeComputer task looks like this:



To enter this information into the Agent Generator we need to open the Primitive Task Editor, either by double-clicking on the MakeComputer entry in the list of known tasks in the root Generator window, or the its entry in the Identified Tasks table in the Agent Definition window. Now we can enter the task preconditions.

- Click on the **New** button in the Task Preconditions panel and choose the **KeyBoard** fact.

You have created a variable that will refer to the keyboard instance used when this task is invoked, hence best to give it a more meaningful name.

- Edit the Instance field, renaming the contents to **kb**.

Although there are no keyboard-producing agents planned for this application we can specify that keyboards must be obtained locally, (typically computer manufacturers tend to use logo-branded keyboards).

- Double click on the **Modifiers** field of the keyboard entry, this opens a window with several check-boxes, select the one marked 'must be in local database', and click on OK.

Next we enter the constraints relevant to keyboard consumption, in this application we assume that one is required per computer and that the keyboard type specified for the computer will restrict the type of keyboard used.

- In the attribute table underneath the keyboard entry double click the **number** field, and enter **computer.number** (see the Realisation Guide for details on the right mouse pop-up shortcut).
- Then double click the **type** field, and enter **computer.kb_type**

Next we can enter the motherboard precondition:

- Click on the **New** button and choose the **MotherBoard** fact, then rename the instance to **board**.

Next to be entered are the constraints relevant to motherboard consumption, namely that one is needed per computer and that the CPU speed specified for the computer will match the motherboard type.

- In the attribute table double click the **number** field, and enter **computer.number**
- Then double click the **type** field, and enter **computer.cpu_speed**

Next we can enter the monitor precondition:

- Click on the **New** button and choose the **Monitor** fact, then rename the instance to **monitor**.

Next we enter the constraints relevant to monitor consumption, here we assume that one is required per computer and that the type specified for the computer will match the type of the monitor.

- In the attribute table double click the **number** field, and enter **computer.number**
- Then double click the **type** field, and enter **computer.monitor_type**

The final precondition refers to the Printer:

- Click on the **New** button and choose the **Printer** fact, then rename the instance to **printer**.

Next we enter the constraints relevant to printer consumption, here we assume that one is required per computer and that the type specified for the computer will match the printer type.

- In the attribute table double click the **number** field, and enter **computer.number**
- Then double click the **type** field, and enter **computer.printer_type**

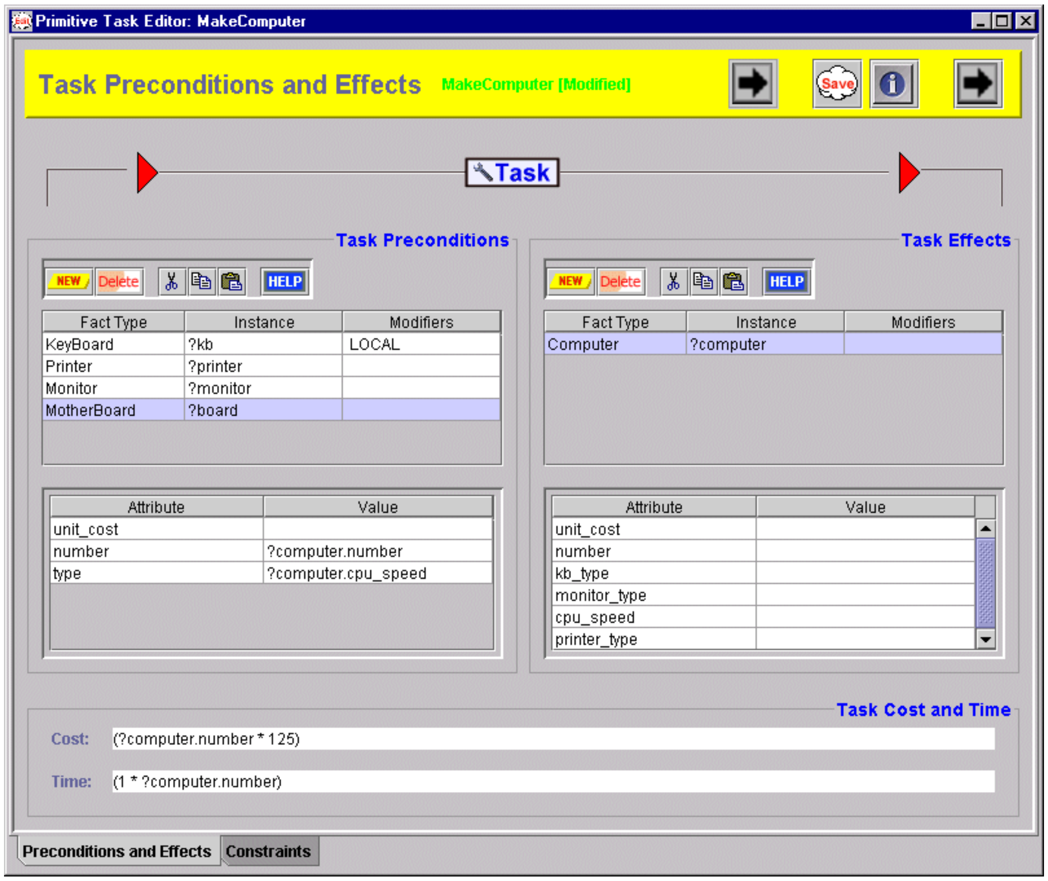
This completes the specification of preconditions. The next step is to specify the computer variable to which we have been referring when entering the precondition constraints.

- Click on the **New** button in the Task Effects panel and choose the **Computer** fact, then rename the instance to **computer**. This matches the computer variable mentioned in the preconditions.

There are no constraints applicable on the Computer effect so the next step is to specify the estimated costs in terms of time and money of invoking the task. We shall assume that each computer has a base cost of €125, and will be constructed within 1 time-grain, so:

- In the cost field enter **(?computer.number * 125)**
- And in the time field enter **(1 * ?computer.number)**

The **MakeComputer** task is now complete, and the Primitive Task Editor window should now look like that in Figure 3. You should now save the task and close the task editor.



Fact Type	Instance	Modifiers
KeyBoard	?kb	LOCAL
Printer	?printer	
Monitor	?monitor	
MotherBoard	?board	

Attribute	Value
unit_cost	
number	?computer.number
type	?computer.cpu_speed

Fact Type	Instance	Modifiers
Computer	?computer	

Attribute	Value
unit_cost	
number	
kb_type	
monitor_type	
cpu_speed	
printer_type	

Cost: (?computer.number * 125)

Time: (1 * ?computer.number)

Figure 3: A screenshot of the Primitive Task Editing after entering the 'MakeComputer' task

The Agent Organisation Process

Looking back at this agent's design we can see that agent acquaintances can be encoded by performing the ORG-1 and ORG-2 activities. So, open the Agent Editor again for the Dell agent, and move to the Agent Organisation panel.

From the specification we know that the Dell agent has two acquaintances, Taxan and Intel, which are bound by a supply contract to provide their resources on demand to Dell. In ZEUS we can use the subordinate relationship to represent this.

- Click on the 'New Acquaintance' button in the Acquaintances panel, and select the entry for **Taxan**. An entry for Taxan will now appear in the acquaintances table.

NEW

- To change its relationship double-click the **Relation** field, this will open a list of the possible relationships, choose the one called **subordinate**.
- Then click on the 'New Acquaintance' button again and select the entry for **Intel**. Then edit its relationship field so it is also **subordinate**.

NEW

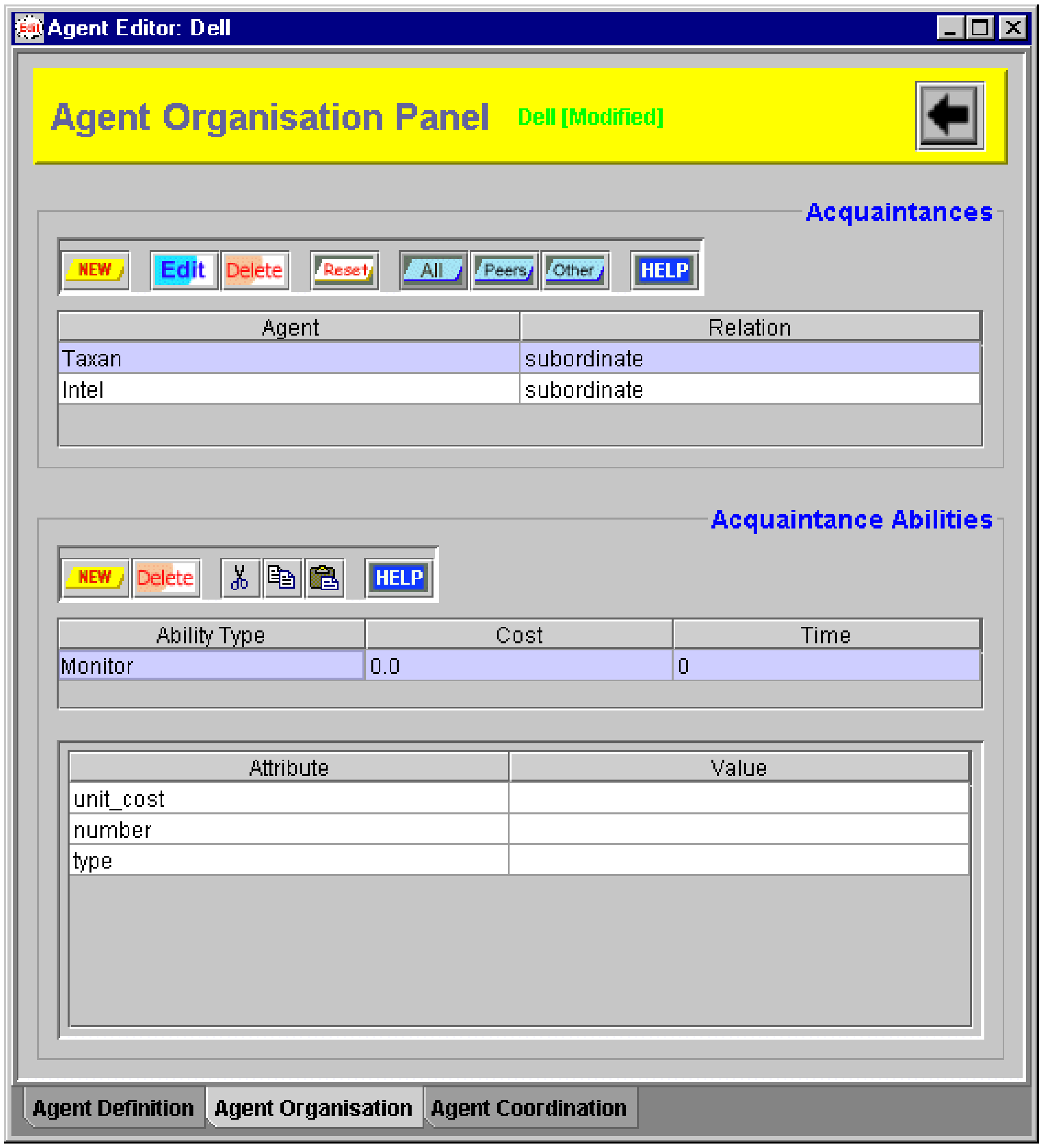
With the acquaintances entered we can move onto activity ORG-2: specifying the agents' capabilities, beginning with the abilities of the Taxan agent.

- Ensuring the Taxan entry is selected in the upper table, click on the 'New' button in the Acquaintance Abilities panel, and select the entry for **Monitor**.
- Now click on the Intel entry in the upper table, click on the 'New' button in the Acquaintance Abilities panel, and select the entry for **MotherBoard**.

NEW

NEW

By providing this information the Dell agent will know that the Taxan agent can produce Monitors and the Intel agent can produce MotherBoards; without this information the Dell agent would have to discover these capabilities through a Facilitator agent.



Agent	Relation
Taxan	subordinate
Intel	subordinate

Ability Type	Cost	Time
Monitor	0.0	0

Attribute	Value
unit_cost	
number	
type	

Figure 4: A screenshot of the Agent Organisation Panel after entering the 'Dell' agent acquaintances

This stage also enables us to provide the Dell agent with estimates of the cost and time involved by its acquaintances' capabilities; but this is not necessary in this application. The Agent Organisation Editor should now look like Figure 4, and its contents can be saved.

The Agent Co-ordination Process

Referring back to the *Supply Chain Head* design, we can see several solutions that 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 negotiation strategies [GrowthFunction] (COORD-2)`

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

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).

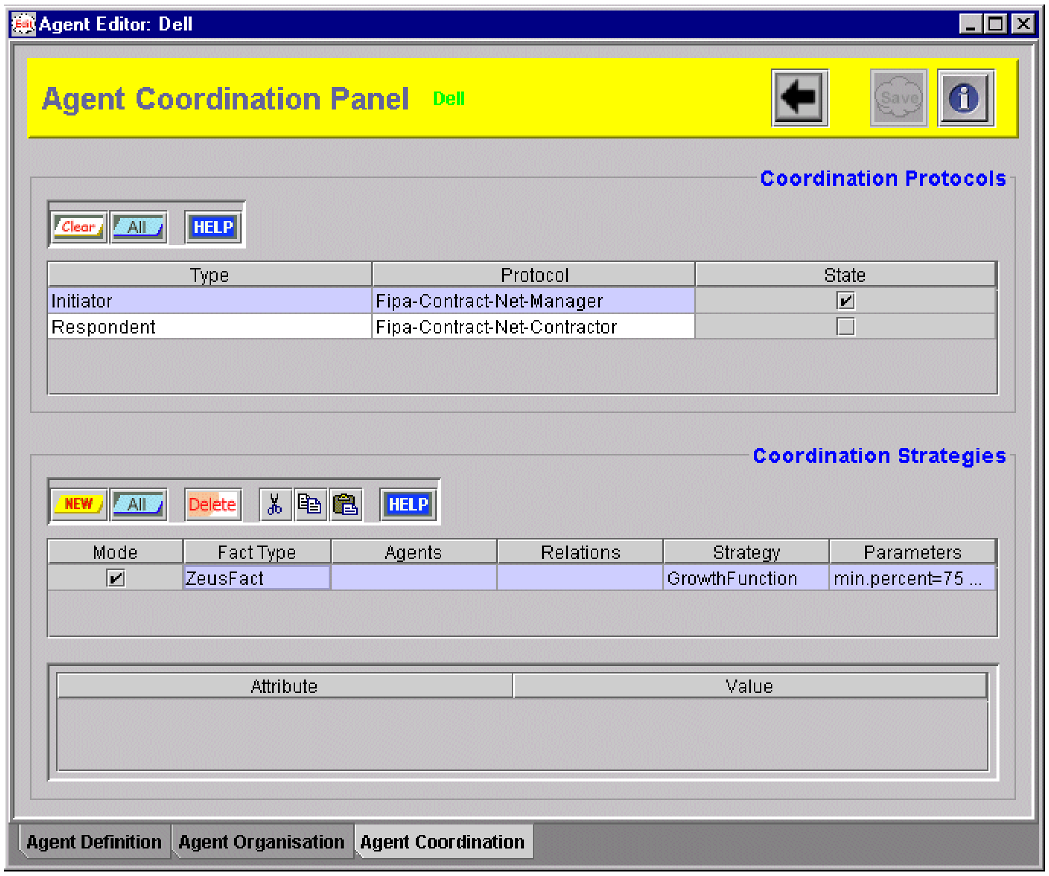
⇒ *step.default*: the increment granularity, determines the minimum increase between bids

⇒ *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 **75**.
- 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 **2**.
- Click on the `New` button to create a new entry, rename the entry in the `Key` field to **step.default** and press <return>. Then edit the `Value` field so that it contains the value **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 **95**.
- 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.



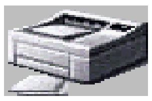
Type	Protocol	State
Initiator	Fipa-Contract-Net-Manager	✓
Respondent	Fipa-Contract-Net-Contractor	☐

Mode	Fact Type	Agents	Relations	Strategy	Parameters
✓	ZeusFact			GrowthFunction	min.percent=75 ...

Attribute	Value

Figure 5: The Agent Co-ordination Panel after configuring the buying behaviour of the 'Dell'.

As no constraints need be specified for this strategy, so the process of equipping this agent to sell is complete. In fact, the definition of the entire 'Dell' 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 create the other agents.



4.2.2 Creating the HP Agent

The same process that built the Dell agent can be used to create the HP agent. Looking back at the design derived from the agent roles we can see the only difference a supply chain Head and a Participant is that the latter needs a co-ordination protocol to sell its resources.

By now the reader should be familiar with how the ZEUS Agent Generator is used, (if not, the Realisation Guide provides detailed instructions). So for this agent only the steps necessary to realise it will be provided.

Agent Definition

The activities necessary at this stage are:

- Specify two primitive tasks exist: **MakeInkjetPrinter** and **MakeLaserPrinter** [**DEF-2**]

- Add a **PrinterCase** initial resource, name: inkcases, unit_cost: 150, number: 2000, type: ink
- Add a **PrinterCase** initial resource, name: lasercases, unit_cost: 175, number: 2000, type: laser

Agent Organisation

- Specify that **CartridgeCorp** is an acquaintance, with a relationship of **subordinate** [**ORG-1**]

Agent Co-ordination

The supply chain participant design tells us that the following are activities are necessary, equipping the agent to buy with an Initiator protocol and Default-No-Profit strategy, and equipping the agent to sell with a Respondent protocol and the Decay Function strategy.

Equipping the Agent to Buy

- Click on the checkbox to select the **Fipa-Contract-Net-Manager** protocol [**COORD-1**]
- Specify the strategy will be used for interactions with **subordinate** agent, and leave the strategy function as **Default-No-Profit**

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. As 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; here two parameters are interpreted differently from those described for the bidding process:

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

⇒ *max.percent*: the fraction of the commodity's perceived value that it is offered 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 **105**.
- 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 **2**.
- Click on the `New` button to create a new entry, rename the entry in the `Key` field to **step.default** and press <return>. Then edit the `Value` field so that it contains the value **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 **130**.

Now click on the OK button to commit the changes and dismiss the window. The agent has now been specified and can be saved.

Task Definition

Edit the **MakeInkjetPrinter** task, and configure it thus:

- Create an effect that is a **Printer** fact, name it **printer**, and set the attribute type to **inkjet**
- Add a **PrinterCase** precondition, name it **case** and apply the **Local** modifier. Then set the number to **printer.number** and the type to **printer.type**.
- Add a **InkCartridge** precondition and name it **ink**. We'll assume each printer comes with 10 spare ink cartridges so set the number to **(10*printer.number)**.
- Then set the task cost to **(45 * ?printer.number)** and the time to **(1 * ?printer.number)**.

Now save the task, close the task editor and edit the **MakeLaserPrinter** task, this involves:

- Create an effect that is a **Printer** fact, name it **printer**, and set the attribute type to **laser**
- Add a **PrinterCase** precondition, name it **case** and apply the **Local** modifier. Then set the number to **printer.number** and the type to **printer.type**.
- Add a **TonerCartridge** precondition and name it **toner**. We'll assume each printer comes with a single toner cartridge so set the number to **printer.number**.
- Then set the task cost to **(50 * ?printer.number)** and the time to **(1 * ?printer.number)**.

Now save the task and close the task editor. The HP agent and the tasks it will perform are now complete so we can move on to create the Supply Chain Tail agents.

4.2.3 Creating the Intel Agent

Looking back at the design derived from the agent roles we can see the only difference a supply chain participant (like HP) and a tail (like Intel) is that the latter does not need a co-ordination protocol to acquire its resources. Hence it is slightly simpler to specify.

Agent Definition

The activities necessary at this stage are:

- Specify a primitive task exists: **MakeMotherBoard** [**DEF-2**]
- Add a **CPU** initial resource, name: budget_cpus, unit_cost: 25, number: 1000, type: 300
- Add a **CPU** initial resource, name: medium_cpus, unit_cost: 30, number: 1000, type: 350
- Add a **CPU** initial resource, name: fast_cpus, unit_cost: 35, number: 1000, type: 400

Agent Organisation

- Specify that **Dell** is an acquaintance, with a relationship of **superior** [**ORG-1**]

Agent Co-ordination

The supply chain tail design tells us that for the agent to sell resources it will need to be equipped with a Respondent protocol and the Decay Function strategy. This involves the following:

- Click on the checkbox beside the entry for **Fipa-Contract-Net-Contractor**.
- Ensuring that the **Fipa-Contract-Net-Contractor** entry has been selected in the upper table and double-click on the `Strategy` field in the lower table. Then select the entry for **DecayFunction** from the list of available strategies.

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

- Enter a parameter called **min.percent** and assign the value **107**.

- Enter a parameter called **no.quibble** and assign the value **2**.
- Enter a parameter called **step.default** and assign the value **2**.
- Enter a parameter called **max.percent** and assign the value **115**.

Now click on the OK button to commit the changes and dismiss the window. The agent has now been specified and can be saved.

Task Definition

Edit the **MakeMotherBoard** task, and configure it thus:

- Create an effect that is a **MotherBoard** fact, and name it **mb**
- Add a **CPU** precondition, name it **cpu** and apply the **Local** modifier. Then set the number to **mb.number** and the type to **mb.type**.
- Then set the task cost to **(275 * ?mb.number)**.

Now save the task and close the task editor.

4.2.4 Creating the Taxan Agent

The Taxan agent involves exactly the same activities as the other supply chain tail agents, namely:

Agent Definition

The activities necessary at this stage are:

- Specify a primitive task exists: **MakeMonitor** [**DEF-2**]
- Add a **CRT** initial resource, name: crt, unit_cost: 60, number: 2000
- Add a **GFXadapter** initial resource, name: vga_card, unit_cost: 10, number: 2000, type: vga
- Add a **GFXadapter** initial resource, name: svga_card, unit_cost: 10, number: 2000, type: svga

Agent Organisation

- Specify that **Dell** is an acquaintance, with a relationship of **superior** [**ORG-1**]

Agent Co-ordination

The supply chain tail design tells us that for the agent to sell resources it will need to be equipped with a Respondent protocol and the Decay Function strategy. This involves the following:

- Click on the checkbox beside the entry for **Fipa-Contract-Net-Contractor**.
- Ensuring that the **Fipa-Contract-Net-Contractor** entry has been selected in the upper table and double-click on the `Strategy` field in the lower table. Then select the entry for **DecayFunction** from the list of available strategies.

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

- Enter a parameter called **min.percent** and assign the value **102**.
- Enter a parameter called **no.quibble** and assign the value **2**.
- Enter a parameter called **step.default** and assign the value **2**.
- Enter a parameter called **max.percent** and assign the value **125**.

Now click on the OK button to commit the changes and dismiss the window. The agent has now been specified and can be saved.

Task Definition

Edit the **MakeMonitor** task, and configure it thus:

- Create an effect that is a **Monitor** fact, and name it **monitor**
- Add a **CRT** precondition, name it **crt** and set its number attribute to **monitor.number**
- Add a **CRT** precondition, name it **card** and set the number to **monitor.number** and the type to **monitor.type**.
- In the cost field we can enter rules that provide different estimates depending on task parameters (in this case the monitor type); so enter the following: **(if (?monitor.type == sgva) then (120 * ?monitor.number) else (100 * ?monitor.number))**
- And in the time field enter **(1 * ?monitor.number)**

Now save the task and close the task editor.

4.2.5 Creating the CartridgeCorp Agent

The CartridgeCorp agent involves the same activities as the other supply chain tail agents, namely:

Agent Definition

This agent does not possess any initial resources, but can only produce one cartridge at a time, to specify this:

- Edit the maximum number of simultaneous tasks field so that it contains the value **1** [**DEF-1**]
- Next, specify two primitive tasks exist: **MakeTonerCartridge** and **MakeInkCartridge** [**DEF-2**]

Agent Organisation

- Specify that **HP** is an acquaintance, with a relationship of **superior** [**ORG-1**]

Agent Co-ordination

The supply chain tail design tells us that for the agent to supply resources it will need to be equipped with a Respondent protocol and a supply strategy. This involves the following:

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

In this case the agent will use the Default-No-Profit strategy, so nothing else needs to be entered. You can now click on the OK button to commit the changes and dismiss the window. The agent has now been specified and can be saved.

Task Definition

Edit the **MakeInkCartridge** task, and configure it like this:

- Create an effect that is a **InkCartridge** fact, and name it **ink**

No preconditions are necessary, we assume that production is handled outside the sphere of responsibility of the agent.

The default values for cost and time will suffice, so you can now save the task and close the task editor.

As all the task agents are now specified, so 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 PC manufacturing agents there are a Name Server, a Facilitator and a Visualiser.

The ANS and Facilitator are essential to this application, and the Visualiser is optional but always useful. 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.

The recycle period of the Facilitator governs the time interval between the messages it sends to each agent to update its world-view. Hence to reduce the amount of communication taking place, thus.

- Change the `Recycle Period` field of the Facilitator to **2.0**

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. Screenshots of the Utility Agent panels and more detailed instructions can be found in Section 4 of the Realisation Guide. Next, we turn our attention to the task agents.

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 Dell 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 supply chain agents are 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 the Dell agent requires a means of capturing the demand that will initiate the supply chain; this should be implemented in a separate class and identified in the `External Program` field.

- Double click on the Dell agent's `External Program` field and enter **DellPump**

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.

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.

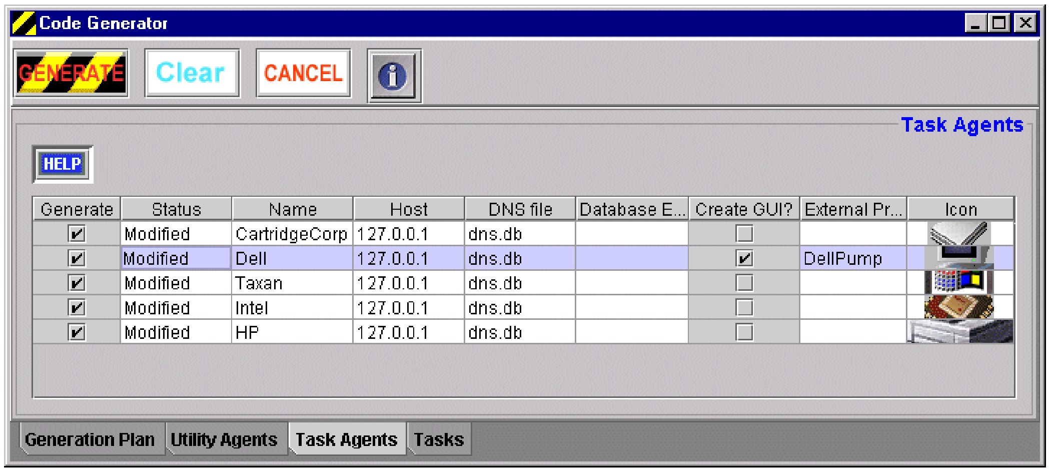
If you would like to gain a better insight into the mechanics of the supply chain then you can create GUIs for every other agent you are interested in observing.

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/pc/gifs' directory and choose the file named **computer.gif**

Repeat the above step for each of the other agents, using **cartridge.gif** for the CartridgeCorp agent; **monitor.gif** for the Taxan agent; **cpu.gif** for the Intel agent; and **printer.gif** for the HP agent.

This concludes the configuration of the task agents, and the Task Agent panel should look like that shown in Figure 6. Next, we shall generate the Java source code for our application.



Generate	Status	Name	Host	DNS file	Database E...	Create GUI?	External Pr...	Icon
☑	Modified	CartridgeCorp	127.0.0.1	dns.db		☐		
☑	Modified	Dell	127.0.0.1	dns.db		☑	DellPump	
☑	Modified	Taxan	127.0.0.1	dns.db		☐		
☑	Modified	Intel	127.0.0.1	dns.db		☐		
☑	Modified	HP	127.0.0.1	dns.db		☐		

Figure 6: A screenshot following the specification of the task agents

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.

Your first option is to choose the directory into which the source code will be written. As your agents are likely to be identical to those of the pre-generated agents supplied with ZEUS you could choose to write the code into the zeus/examples/pc directory. This will replace the existing agent definitions (but existing files are never over-written by the Generator, they are just renamed with a % postfix).

- If you want to change the target directory, 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).

Looking back at the design of the Dell agent in Section 3 we can see two potential implementation activities:

• a resource consumption task body	[Optional / **`IMPL-2`**]
• a supply initiation trigger	[Automatic / **`IMPL-4A`**]

In our simple simulation it is not necessary to implement the task body that will consume the supplied resource (in the real world this would equate to sending the finished PC out to the customer and updating the sales and delivery records).

However, some means of triggering the supply chain is not optional (otherwise the agents will remain idle). This can be achieved without implementing any additional code by using the **New Goal** menu option of the Dell agent's Agent Viewer interface; this will create a goal instructing the Dell agent to obtain a Computer fact with the desired parameters. As the design suggests, there is an alternative: to create an external program that will create a goal and send it to the agent through its interface. How the latter is achieved will be considered next.

Implementing the Trigger Program

The trigger program, (DellPump.java in the zeus/examples/pc directory), provides a simple example of how to issue instructions to an agent. The mechanism used is described in more detail in section **IMPL-4A** of the Realisation Guide, but essentially consists of an interface that allows the components of an agent to be directly accessed. The Java code involved is shown below:

SOURCE CODE

```
 Fact fact = context.OntologyDb().getFact(Fact.VARIABLE,"Computer");
fact.setValue("cpu_speed","400");
fact.setValue("kb_type","UK");
fact.setValue("printer_type","laser");
fact.setValue("monitor_type","svga");
fact.setNumber(1);
int now = (int)context.now();
Goal g = new Goal(context.newId("goal"), fact, now+8, 0,
                  context.whoami(), now+4);
context.Engine().achieve(g);
```

What is happening is that a new fact representing the desired item is created and used to create a new goal. The goal is then added to the agent's Co-ordination Engine, and will trigger the invocation of its MakeComputer task, and in the course of acquiring the necessary resources for this task negotiations with supplier agents will begin.

If you are building the application from scratch now is the time to compile the generated source code and external program implementation. 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 PC Manufacture 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 5 supply chain agents. As a result five more Java processes should now be running in the background. Whilst on screen an Agent Viewer window will appear for the Dell 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, and the Report Tool is useful for monitoring the status of each of the tasks in the supply chain. The ZEUS Runtime Guide describes how to use the various facilities of the Visualiser tool, but as these apply to any multi-agent application the remainder of this section will only consider the application-specific features of this example.

Once launched the agents will register themselves with the Name Server, and reply to the Facilitator and the Visualisers (the latter only if asked). No tasks will be performed until the user issues a goal to one of the agents, either through the Agent Viewer or an external program like the DellPump window. Assuming that a goal has been provided, what is going on behind the scenes?

5.1 How Negotiation Works

As explained in Section 3.4 of the ZEUS Realisation Guide, the interactions of the agents are governed by strategies. In this application the sellers who negotiate (i.e. who don’t use the No-Profit strategy) 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.

Using the PC Manufacture Demo

More detailed instructions on running this demo can be found in the readme.txt file in the zeus/examples/pc directory.

Concluding Remarks

The best way to understand the PC Manufacture 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 pc directory) makes calls to the ZEUS classes.

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

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