The Zeus Agent Building Toolkit

ZEUS Methodology Documentation Part II

Case Study 3

Maze Navigator

Creating Rule-based Agents with ZEUS

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

Simon Thompson, simon.2.thompson@bt.com

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 a simple maze solving application, and thus illustrates two particular aspects of ZEUS:

- how to implement a centralised information service that can be queried using agent messages
- how to control the behaviour of an agent using rules

In contrast to the PC Manufacture application, this case study involves no tasks; instead we implement *reactive* behaviour that is encoded in the agents using rules. The participants and their abilities are:

The first agent will be called **Environment**, which is responsible for storing a model of the maze. It also will store the location of the agents inside the maze, and will respond to their queries about the maze. The agent should also display a graphical depiction of the maze and the location of the agents within it.

The other type of participant is the **Navigator** agent. These agents are trying to escape from the maze, but have no model of what it looks like, hence they must send messages to the Environment and make their decision about where to move next based on the response received. They will each contain a rule base of problem-solving behaviour that will govern their movements. Each Navigator agent should also possess a user interface that will report its current status to the user. This example assumes the existence of three Navigator agents, called Red, Green and Blue.

The first half of this case study describes how the Maze application is designed and realised; Section 2 explains how an appropriate role model is chosen and configured to match the required specification, which forms 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. 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

We start the analysis process by comparing the desired system to the features of existing role models. In this case we have an agent that is a centralised provider of information, and one or more other agents that communicate with it. This would seem to be a simpler arrangement than those described by the Business Process role models, and closer to those in the Information Management domain.

2.1 Select Role Models

One of the simplest Information Management role models is the Shared Information Space. So we ask, would this provide a suitable basis for our application? The role model consists of a centralised information resource, and its responsibilities seem to match those of our planned maze Environment agent. There are also Subscriber roles that seem an appropriate basis for the Navigator agents, so this role model would seem a good choice.

Now 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. However there is no need for a broker given the Subscriber agents will probably know of the Environment agent a priori. Then we assign roles to agents, which is a simple task given the role model, namely:

Agent Name	**Roles Played**
Environment	Publisher (Model, View, Controller)
Navigator	Subscriber

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

Next we use the role model description to identify the responsibilities of each role the agent will play. These can be categorised as social or domain responsibilities, the former involving interaction with other agents, and the latter involving some local application-specific activity. Hence the responsibilities involved in this application are:

PUBLISHER – Social Responsibilities	
Origin	**Responsibility**
Controller	To accept incoming registrations
Controller	To respond to registrations
Controller	To receive information requests from subscribers
Controller	To respond to information requests from subscribers
PUBLISHER – Domain Responsibilities	
Origin	**Responsibility**
Controller	To query model in response to subscriber's information requests
Model	To encapsulate some repository of information
View	To visualise the contents of the model [additional from specification]

Then we consider the Subscriber role's responsibilities:

SUBSCRIBER – Social Responsibilities	
Origin	**Responsibility**
Subscriber	To register self with the Publisher
Subscriber	To request information from the Publisher when necessary
SUBSCRIBER – Domain Responsibilities	
Subscriber	To perform its application specific activities
Subscriber	To update the user on current progress [additional from specification]

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 a responsibility refinement phase, which is followed by a knowledge-modelling phase; we shall consider them in turn.

3.1 Problem Design

Now that we have identified the responsibilities required for each agent, we can 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 with the responsibilities of the Publisher roles.

3.1.1 Realising the Publisher roles

The roles of the Publisher primarily concern the handling of Subscriber requests, as shown in the table below. These entries will illustrate that matching a problem to an existing solution requires some expertise in the construction of agent systems, (expertise that has hopefully been captured within this case study).

PUBLISHER SOCIAL RESPONSIBILITIES

Responsibility:	To accept incoming registrations
Origin:	Controller
Problem:	Listening for and recording the addition of a new subscriber
Solution:	Add Fact Monitor to External Program (**IMPL-4**)
Explanation:	As all incoming messages are added to the agent's Resource Database registration can be implemented through the Fact Monitor interface. This has a method that is triggered when new facts are received, whereupon appropriate action can be taken (see the next entry).
Responsibility:	To respond to registrations
Origin:	Controller
Problem:	To send a message back to the subscriber
Solution:	Create an Initial Agent Resource (**DEF-3**) and Specify the Presence of a Rulebase (**DEF-2**) and Implement a Message Sending Rule (**RULE-1**)
Explanation:	When the rule is triggered (see previous entry) the agent will send a message back to the subscriber acknowledging registration. The fact sent to successful subscribers will have been supplied to the Publisher as an initial resource.
Responsibility:	To receive information requests from subscribers
Origin:	Controller
Problem:	To monitor the arrival of new movement requests
Solution:	Add Fact Monitor to External Program (**IMPL-4**)
Explanation:	Using a similar approach to the handling of registrations, the Fact Monitor can also detect incoming queries and react accordingly. In this case by querying the Model and triggering the rules that send a response (see next entry).
Responsibility:	To respond to information requests from subscribers
Origin:	Controller
Problem:	To send Message [to: Subscriber, about:]
Solution:	Implement Message Sending Rules (**RULE-1**)
Explanation:	When the rule is triggered (in this case based on information retrieved from the Model) the Controller can send a message back to the subscriber providing the information it requested.

As these solutions are not provided by the generic ZEUS agent class they must be realised manually by configuring the agent using the ZEUS Agent Generator tool and writing the necessary program code. How this is achieved is explained in the next section, meanwhile we can consider the Publisher's domain responsibilities.

PUBLISHER **DOMAIN RESPONSIBILITIES**

Responsibility:	To query model in response to subscriber's information requests
Origin:	Controller
Problem:	To extract information from the model to satisfy the subscriber queries
Solution:	`Implement Adapter method (`**`IMPL-4`**`)`
Explanation:	The adapter method will provide the application specific code that translates the Subscriber's request into Model queries, which should then trigger the rules that send a response back to the Subscriber. The adapter method will be linked to the agent through its ZeusExternal interface.

Responsibility:	To encapsulate some repository of information
Origin:	Model
Problem:	To store information
Solution:	`Allocate Initial Agent Resources (`**`DEF-3`**`) or` `Connect Agent to an External Resource (`**`IMPL-3`**`) or` `Implement Model Externally (`**`IMPL-4`**`)`
Explanation:	Agents can store information in a variety of ways; it can be held as facts in the Resource Database, in an External Database, or in Java code that is accessible to the agent's ZeusExternal interface. The choice will depend on the type of information being stored. In this case, we shall store the maze externally in a 2-dimensional Boolean array.

Responsibility:	To visualise the contents of the model
Origin:	View
Problem:	To provide a graphical depiction of the state of the model
Solution:	`Implement Model GUI`
Explanation:	The maze representation is too application specific to be interpreted by the Visualiser, so a custom interface will have to be written that displays it. This can be instantiated from the agent's ZeusExternal program.

3.1.2 Realising the Subscriber Role

This involves adopting the same approach as before, considering each responsibility in turn:

SUBSCRIBER **SOCIAL RESPONSIBILITIES**

Responsibility:	To register self with the Publisher
Problem:	Send message to Publisher [about: registration]
Solution:	`Know acquaintances (`**`ORG-1`**`) or Use Facilitator, and` `Implement Message Sending Rules (`**`RULE-1`**`)`
Explanation:	The identity of the Publisher can be set statically when the agents are generated, or discovered through the Facilitator at run-time. The Subscriber

	can then send a message to it registering itself.

Responsibility:	To request information from the Publisher
Problem:	To send the Navigator's next move to the Maze Environment agent
Solution:	`Create an Initial Agent Resource (`**`DEF-3`**`) and` `Specify the Presence of a Rulebase (`**`DEF-2`**`) and` `Implement Message Sending Rules (`**`RULE-1`**`)`
Explanation:	The initial resource will store a move template that will be used to inform the Environment of the move the Navigator wishes to make. The move serves as an implicit request for information "I'm moving North" - to which the reply will be along the lines of "there are obstacles north and east". The preconditions of these rules encode the maze-solving behaviour of the agent, and are discussed next.

SUBSCRIBER DOMAIN RESPONSIBILITIES

Responsibility:	To perform application specific activity
Problem:	Implementing application functionality – in this case escaping the maze, and informing the user of its current progress
Solution:	`Implement Behaviour Rulebase (`**`RULE-1`**`)`
Explanation:	The expertise required to navigate around the maze is stored in the form of rules. In this example we assume that the responses from the Environment agent will inform the Navigator of the obstacles in its path. Hence these rules will use the information gained from the Environment agent to determine the next move that will be made.

Responsibility:	To update the user on current progress
Problem:	To provide a graphical depiction of the state of the model
Solution:	`Implement External Program (`**`IMPL-4`**`)`
Explanation:	A small user interface can be created to display what the agent is currently doing. This will need to implement the Fact Monitor interface in order to detect what information is being sent and received.

By the end of this process we should now have a clearer idea of how to realise all the agents' responsibilities using the ZEUS Agent Generator. But before we can begin that we need to consider the ontology the application will use.

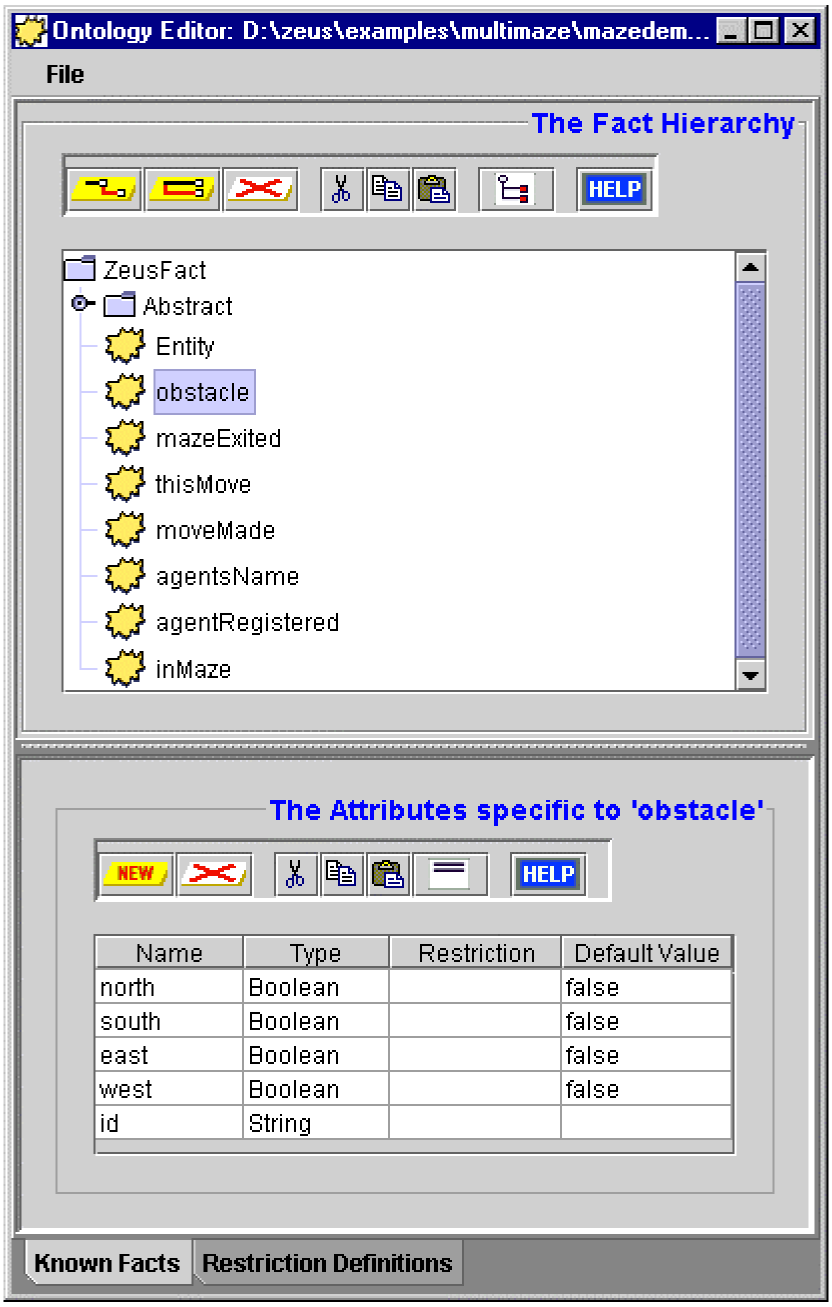
3.2 Knowledge Modelling

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 Maze Navigator application relate to the concepts that represent the maze and movement within it. They do not need to inherit attributes from other facts, and so all will be sub-facts of the root fact (ZeusFact). There is also no need from any value constraints either. Hence the information that needs to be entered is listed in the following table. Once entered into the Ontology Editor this application's ontology will look looks like that shown in Figure 1.

Fact	Attributes	Default	Meaning
Obstacle	north: south, east, west : *Boolean;* id : *String*	false	Whether there is an wall in these directions
MazeExited	id : *String*		Identity of the agent that has left the maze
ThisMove	north: south, east, west : *Boolean*; id : *String*	false	The attribute set to true is the direction moved by this agent
MoveMade	Moved : *Boolean;* id : *String*	false	Internal flag indicating a valid move has been made
AgentName	name : *String*		Name of the agent attempting to register
AgentRegistered	name : *String*		Name of the registering Environment agent
InMaze	IsInMaze : *Boolean*; name : *String*	false	A flag for each registered agent, true if still in maze



Name	Type	Restriction	Default Value
north	Boolean		false
south	Boolean		false
east	Boolean		false
west	Boolean		false
id	String		

Figure 1: Screenshot of the Ontology Editor displaying the Maze Demo ontology

Then 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.

To skip this section, you can load the previously created project definition file, called **mazedemo.def** in the examples/mazedemo directory. (But if you do try defining some of the agents manually do not overwrite the multimazedemo.def file - otherwise you will not be able to load the prewritten rulebases).

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 and create a child fact called **obstacle**; (see entry **ONT-3**)
- Select the **obstacle** fact and create four new attributes called **north**, **south**, **east** and **west**; all should be of type **Boolean** and have the default value of **false**.
- Ensuring the newly created 'obstacle' fact node is selected, click on the "Add New Peer Fact" button. Rename the entry to **mazeExited**, and then create a new attribute called **id** of type **String**.
- Repeat the above action to create new facts for the following: **thisMove**, **moveMade**, **agentName**, **agentRegistered** and **inMaze**. The attributes and default values involved are listed in the table in section 3.2.

The contents of the Ontology Editor should now look like that shown in Figure 1, and so can be saved and used as the basis for our agent application.

- Save the Ontology as 'mazedemo.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 Environment Agent

The design for the Environment agent was outlined in section 3.1.1 when we considered how to release the responsibilities of the Publisher 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 **Environment**.

The skeleton of a new generic ZEUS agent now exists inside the Generator.

- Double-click on the 'Environment' 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 social and domain responsibilities of our design to determine 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 activities **DEF-2** and **DEF-3** specified in the design, which can be realised by the following:

- Click on the 'Create New Task' button in the Task Identification panel, and select the 'New Rulebase' option, (this process is described in activity **DEF-2**).
- A new entry will appear in the table, double click on it to rename it to **respondMove**.

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 **agentRegistered** as the fact type from the fact hierarchy.
- This fact will be used to store a reference to the Environment agent which will be sent to all agents who successfully register, so edit the **name** attribute and set the value to **Environment**.

This concludes the Agent Definition process, as there are no Organisation or Co-ordination aspects to configure for this agent; so you can now save the agent definition by clicking on the Save icon.

Defining the Environment Agent's Rulebase

During the Agent Definition phase we specified the existence of the **respondMove** rulebase, and so the next step is to actually define it. Rulebases are collections of rules that have been grouped together for convenience, usually because they have related functions. Each rule consists of a s-expression encompassing a pattern and an action; so when the agent possesses data that matches a pattern then the associated rule will be fired.

As rules are application dependent, they can take a lot of work to produce, although if you are implementing a relatively well-known application you should be able to find the rules required in A.I textbooks and web resource pages.

The rules for the Environment agent do not originate from the Shared Information Space role model (which only describes a generic information distribution system). So instead the rules must be hand-written according to the behaviour desired by the developer. The rules for this example have been prewritten and can be found in the file respondMove.clp; these are:

Rule Name	Trigger	Action
LegalMove	valid move received from a Navigator	message sent informing Navigator of obstacles at new location
IllegalMove	invalid move attempted by Navigator	message sent reiterating obstacles at

		current location
RespondReg	Navigator registration received	Navigator recorded as being in Maze
SendInMaze	Navigator recorded as being in Maze	message sent informing Navigator that it is in the maze
SendExited	Navigator reaches Maze exit	message sent informing Navigator that it is no longer in the maze

These code for the **respondMove** rulebase is shown below, note the pattern => action format of the rules, and the presence of fact and attributes.

```
(:Rulebase respondMove
   (firstLegalMove
      ?moveFlag <- (moveMade (moved true) (id ?agName))
      ?obst <- (obstacle (north ?var176) (west ?var177) (east ?var178) (south ?var179)
(id ?agName))
      =>
      (send_message (receiver ?agName) (content ?obst) (type inform))
      (retract ?moveFlag)
      (retract ?obst)
   )
   (illegalMove
      ?moveFlag <- (moveMade (moved true) (id ?agName))
      ?lastMove <- (thisMove (north ?var19) (west ?var20) (east ?var21) (south ?var22)
(id ?agName))
      ?obst <- (obstacle (north ?var176) (west ?var177) (east ?var178) (south ?var179)
(id ?agName))
      =>
      (send_message (receiver ?agName) (content ?obst) (type inform))
      (retract ?moveFlag)
      (retract ?obst)
      (retract ?lastMove)
   )
   (respondReg
      (agentsName (name ?agName))
      ?ar <- (agentRegistered (name ?envName))
      =>
      (send_message (receiver ?agName) (content ?ar) (type inform))
      (assert (inMaze (isInMaze true) (name ?agName)))
   )
   (sendInMaze
      ?im <- (inMaze (isInMaze true) (name ?var269))
      =>
      (send_message (receiver ?var269) (content ?im) (type inform))
   )
   (sendExited
      ?ex <- (mazeExited (id ?varH))
      =>
      (send_message (receiver ?varH) (content ?ex) (type inform))
   )
)
```

The **respondMove** rules will be entered into the Agent Generator through the Rulebase editor, either by double-clicking on the **respondMove** 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. Once the Rulebase editor is open the rules can be manually entered, (although it would be quicker just to import the rules by loading the **multimazedemo.def** project definition file).

At this point it is worth noting that this example was developed using rules in order to illustrate the Rulebase creation process and the ZEUS rule engine. ZEUS also supports alternative implementation approaches, (such as using a co-ordination protocol using register speech acts to handle the registration process), which may or may not be equally efficient.

4.2.2 Creating the Navigator Agents

This section assumes that the *multimazedemo.def* project definition file has been loaded; this contains definitions of the three Navigator agents: Red, Green and Blue. This file will also automatically load the Navigator agents' rulebases into the Agent Generator tool, which are:

- Register - which contains a rule to register the agent and one to de-register the agent when it eventually exits the maze
- Navigate - this rulebase contains the agents maze solving behaviour; it consists of a starting rule and sets of rules for following walls, choosing directions at junctions and deciding what to do when a particular direction is blocked.

How these rules function is somewhat beyond the scope of this case study, but the reader is free to browse through the rules in the Rulebase Editor if interested. This document continues by considering what utility agents need to be created.

4.3 Utility Agent Configuration

From the main Agent Generator window 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 Environment and Navigator agents there are a Name Server, and a Visualiser.

Only the ANS is essential to this application, but the Visualiser is always a useful agent to include. The runtime parameters of these agents can be configured during this stage as appropriate by clicking on the 'Utility Agent' tab panel; this process is described in section 4 of the Realisation Guide.

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 Environment 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 information used by the Environment agent is stored in an external Java program rather than an external database. Hence the `Database External` field is left blank, and the name of the class implementing the external interface is entered in the `External Program` field, in this case it is called mazeControl. The Navigator agent has a class called navGUI entered in its `External Program` field, which will implement its user interface.

Important note: the class names entered during this stage must be accessible to the Java runtime environment's class-loader. No changes will be necessary if you enter **.** (the current directory) into your local CLASSPATH environment variable.

If you have loaded the project definition file you will also notice that the checkboxes in the `Create GUI?` fields of the Environment and Red Navigator agents are selected. This creates an agent viewer interface for these agents, allowing you to inspect their activity online. You may choose which agents have Agent Viewers according to the free memory on the machine that will host the agents. Finally, you can choose what icon will represent the agents when they appear in the Visualiser, these have already been set to icons in to the 'zeus/examples/multimazedemo/gifs' directory.

This concludes the configuration of the task agents, so we can now generate the Java source code for our 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.

Your first option is to choose the directory into which the source code will be written. You could choose to write the code into the zeus/examples/mazedemo directory, this will replace the existing agent definitions (but existing files will not be over-written, 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 Environment agent in Section 3 we can see several implementation activities:

- `Add Registration Fact Monitor to External Program [`**`IMPL-4`**`]`
- `Add Request Fact Monitor to External Program [`**`IMPL-4`**`]`
- `Implement Adapter method to service requests [`**`IMPL-4`**`]`
- `Implement Model Externally [`**`IMPL-4`**`]`
- `Implement GUI Externally`

The code implementing each of these aspects will be linked to the Environment agent through its external program interface, (**mazeControl.java** in the zeus/examples/mazedemo directory). The next section will explain how each aspect has been implemented.

Implementing the Environment Agent's External Program

The external program, provides a means of programmatically interacting with the agent, using a mechanism that is described in section **IMPL-4A** of the Realisation Guide. The first issue to consider is how the Environment agent will respond to Registration and Request messages. This can be realised by implementing the `factAddedEvent` method of the FactMonitor interface - this will be triggered whenever a new fact is added to the Environment agent's Resource Database, (i.e. whenever it receives a message containing a registration or move); this code is shown in Figure 2.

The request handling code invokes the Adapter method, which is called **move**(…). This method tests whether the move request made by the Navigator agent is legal, and if so the move is enacted. This will result in the agent's location being updated, and the new obstacles around the agent being asserted in the fact database. This will ultimately trigger a new response from the Navigator agent and the cycle will continue (until the maze exit is reached).

Another aspect that must be implemented is the Model, this stores a representation of the maze and is the primary reason that the Environment agent exists at all; (if the Navigator agent had a complete model of the maze it wouldn't have to communicate with the Environment). The model is implemented as a 2-dimensional Boolean array with appropriate acccessor methods, and can be seen in the file **mazeModel.java**. The variable storing the model is instantiated when the mazeControl external program is created by the Environment agent.

The final aspect to be implemented is the maze GUI. The maze GUI is implemented in the file **mazeView.java**, and like the model is instantiated and stored by the mazeControl external program. The view not only depicts the state of the maze, but also allows the user to change the position of the walls and exit at run-time by clicking inside the grid.

SOURCE CODE

```
public void factAddedEvent (FactEvent fe)
  {
    Fact currentFact = fe.getFact();
    String agentName = context.whoami();
    String factType = currentFact.getType();
    System.out.println("Fact = " + factType);
    if (factType.equals("thisMove"))
    {
      String north = currentFact.getValue ("north");
      String south = currentFact.getValue ("south");
      String east = currentFact.getValue ("east");
      String west = currentFact.getValue ("west");
      String id = currentFact.getValue ("id");
      boolean nBool = north.equals("true");
      boolean sBool = south.equals("true");
      boolean eBool = east.equals("true");
      boolean wBool = west.equals("true");
      try
      {
        Thread.sleep(500); // pause for a second
      }
      catch (Exception e) {;}

      // now invoke a function that tests whether it is a legal move
      // side effect the obstacles around the agent are asserted in fact database
      move (id, nBool, eBool, sBool, wBool);
      // a message should now have been sent to the originating agent
      // telling it what new obstacles lie in it's path!
    }
    else if (factType.equals ("agentsName"))
    {
      String id = currentFact.getValue("name");
      int c = (int)(mazeInUse.mazeWidth*Math.random());
      int r = (int)(mazeInUse.mazeHeight*Math.random());
      mazeInUse.registerAgent(id, c, r);
      mazeGui.repaint();
      OntologyDb ont = context.OntologyDb();
      Fact obst = ont.getFact(Fact.FACT, "obstacle");
      obst.setValue("north",mazeInUse.northVal(id));
      obst.setValue("east",mazeInUse.eastVal(id));
      obst.setValue("south",mazeInUse.southVal(id));
      obst.setValue("west",mazeInUse.westVal(id));
      obst.setValue("id", id);
      ResourceDb rdb = context.ResourceDb();
      rdb.add(obst);  // add it to the fact db
    }
  }
```

Request handling code - invokes the adapter method that queries the Model

Registration code - places Navigator in the maze and informs it of obstacles at its starting location

Figure 2: The Implementation of the Environment agent's factAddedEvent method

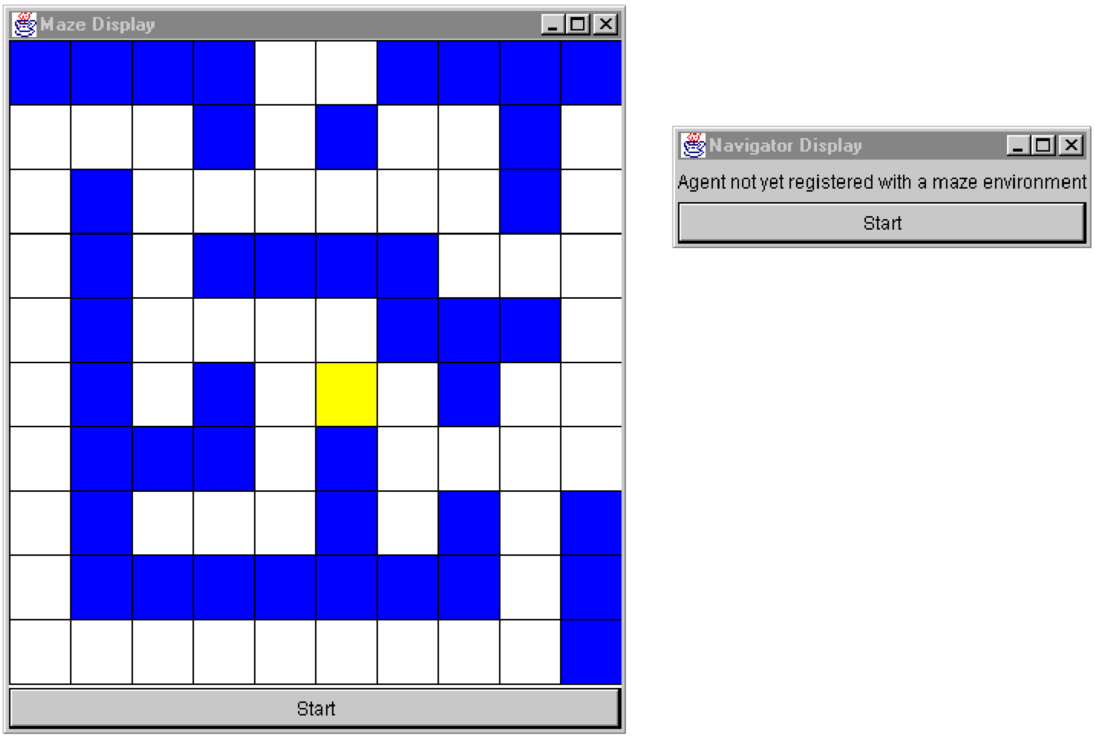
Once the external components have been implemented we can compile the generated source code and external program implementations. Assuming no errors are reported, the application can now be run using the agent launching scripts created by the Generator tool at the same time as the agents. How to run agent applications is explained in the ZEUS Runtime Guide, and is outlined during the next section.

5 Running the Maze Solver 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.) You can also enter the command **'run2'**, if you want to start the Visualiser agent - but this is not necessary.
- Enter the command **'run2'**, this will start the Environment agent and three Navigator agents. As well as four more background processes being launched, you should also see an Agent Viewer window for each of these agents appearing on screen.
- If you want to use the Visualiser, (which is not really necessary for this application because the Agent Viewer windows can tell us all we need to know), enter the command **run3**.

Once launched the agents will register themselves with the Name Server, and their user interfaces will appear, which look like those in Figure 3. The agents will now wait for the user to instruct the Navigator agent to register with the Environment agent. Press the **Start** button on the Navigator agent panel will send a message to the Environment agent registering its presence in the maze. A red circle will then appear in the maze display indicating the presence of a Navigator agent, and words to the same effect will appear in the Navigator's user interface.



Agent not yet registered with a maze environment

Figure 3: The maze Environment agent's display (left) and Navigator agent interface (right)

Once the Navigator has registered itself, the **Start** button of the Maze Display window will become active. Once this is pressed, the Environment agent will send a message to the Navigator informing it of the obstacles at its current location. As these are the preconditions for making a move, and so the Navigator will reply with a move request. After confirming the received move is valid the Environment agent will update the position of the agent in the maze, and you will see the red circle moving onscreen to the new position. The Environment agent does not tell the Navigator agent its new position however, (as the Navigator has no model of the maze this information is irrelevant), instead it sends a description of the obstacles at the new location, which the Navigator assesses when determining its next move.

The query-response cycle continues until the maze exit is reached. The Navigator agent will then be de-registered from the maze by the Environment agent, which disables its Start button until the Navigator agent has re-registered.

Observing Runtime Behaviour

The Agent Viewer interface of each agent provides the best means for observing what is happening 'behind the scenes' as the agents are running. For instance, the Mail In and Mail Out components will show what messages are exchanged, and their content. The conditions and actions of the maze-solving rules can be seen by observing the Navigator agent's Rule Engine component, whilst the Environment agent's Rule Engine will show how it registers agents and responds to their queries. Also of interest are the Resource Databases of each agent, as these components store each agent's transient knowledge and current actions.

5.1 Potential Extensions to this Example

One of the attractions of this example is that can be readily extended with new features; if the reader is keen to explore the possibilities of ZEUS, we recommend attempting the following:

1. One Step Look-Ahead

At present the Navigator agents are only aware of the exit when they encounter it. A logical extension would thus be for the Environment agent to inform the Navigator if it is within one space of the exit. This would require a change to the ontology, so the compass directions stored in the Obstacle concept can take the values true, false and exit. The rules for the Environment and Navigator agents would then need to be changed accordingly.

2. Collaborative Problem Solving

There are a number of ways of improving the efficiency of the maze solving agents, and the most interesting from an agent perspective is collaboration. For instance, when an agent reaches the maze exit it could retrace its steps leaving a 'pheromone' trail that others could follow once they encounter it.

Alternatively you could implement a shared memory space for the agents, which would be akin to the blackboard of the original agent systems. Each agent would place the information gained from the Maze Environment about its local position on the blackboard, from which agents would then base their decisions on where to move next. Whilst the creation of the blackboard would be relatively easy, the rules that take advantage of the information would be more challenging.

3. The MineHunter Variation

A particularly interesting variation on the standard maze solver is the MineHunter game, where the player begins on one side of a minefield and must move to the other side. Every time the player moves they are informed of how many mines are in adjacent spaces - but not in which directions, hence they must infer which spaces have mines by a process of elimination. (This game is slightly different to MineSweeper, which has no notion of player location and movement). A description of a Java implementation of the MineHunter game can be found here:
http://cs.wheatonma.edu/nbuggia/coursework/java/apps/minehunter/minehunter.html

Although the original MineHunter game supported just a single player, the most interesting extension would be using multiple agents. This would provide a global problem that agents could be solve more efficiently by co-ordinating their efforts. Implementing this would require a shared memory space, and some means of co-ordinating the agents so that the best candidate spaces were investigated, in order, by the agents closest to them.

Concluding Remarks

The intention of this case study was to illustrate how a typical Shared Information Space application can be implemented. In this case the application involved maze-solving, but the same rules and agents can easily be adapted to any problem where information is not uniformly distributed, like a digital library for example, or a distributed sensor net.

Those who have run the maze demo will have noticed that the behaviour encoded into the Navigator agents is not particularly sophisticated, as our goal was a rulebase that was simple to understand, and thus extend. The last section has provided some suggestions for the brave and the adventurous, and we look forward to seeing what can be realised.

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

Jaron Collis (jaron@info.bt.co.uk)
Simon Thompson (simon.2.thompson@bt.com)