6. THE DOTNETSIM MODELLING ENVIRONMENT: THE MODELLING COMPONENT

Chapter overview 120

6.1. Objectives of the DotNetSim modelling component 121

6.2. DotNetSim modelling environment: the modelling component 123

6.2.1. Customising Visio™ 125

6.2.2. Drawing the diagram 126

6.2.3. Reading the diagram 128

6.2.4. Describing the model 130

6.2.5. Reviewing the model 133

6.2.6. Reporting the model 135

6.2.7. General operations on the model 137

6.2.8. Simulating the model 137

6.2.9. Utilities 138

6.2.10. Help 138

6.3. Comments on the implementation of the modelling component 138

6.3.1. Packages as objects of other packages 139

6.3.2. Applications object models 141

6.3.3. Visual Basic for Applications 142

6.4. The chapter in context 143

CHAPTER OVERVIEW

The DotNetSim modelling component, together with the DotNetSim stencil component, prototypes a software environment within which DES systems are modelled as sets of states which transit in response to the occurrence of time-stamped events. As explained in chapter 5, the Event Graphs paradigm was chosen to describe such systems, hence, the DotNetSim modelling component implements the functionality required to capture the logic of DE models diagrammatically as Event Graphs and to place the modelling data in relational databases. It automatically loads the Event Graph stencil and uses the functionality of this modelling notation to draw DES as Event Graphs. Upstream from the DotNetSim modelling component there might be descriptive lists of the DE model, written within a Microsoft application which supports Automation. The lists are readable and automatically convertible into Event Graphs by the DotNetSim modelling component. This modelling component deals with Excel-based descriptive lists, which it reads by getting a reference to the corresponding range of cells. Downstream from the modelling component is the DotNetSim simulation engine (see chapter 7), which runs the simulation of the DE model by getting references to the appropriate objects of the corresponding Visio document. Also downstream are other Microsoft applications, namely Word™, PowerPoint™ and, once again, Excel™, to which the model may be reported if required.

The DotNetSim modelling component provides the tools to draw an Event Graph directly on a Visio drawing documents or to convert automatically a descriptive list of a model, produced in Excel, into Event Graphs. It also provides the tools to traverse the Event Graph and store it in a relational database together with other modelling

data read directly from within this component.

This chapter presents and describes the functionality implemented by the DotNetSim modelling component, emphasizing the integration of the different packages. Finally, some comments on the prototype's implementation are made and lead to further discussion on the value of the integration of different Microsoft applications for modelling DES systems.

6.1. OBJECTIVES OF THE DOTNETSIM MODELLING COMPONENT

With the DotNetSim stencil component, the DotNetSim modelling component prototypes a software environment within which the logic and the dynamics of DE models can be captured diagrammatically by Event Graphs and stored in relational databases. The DotNetSim modelling component employs the Event Graph extended modelling notation implemented by the Event Graph stencil (see chapter 5) to represent DE models as networks of events and inter-event relationships. The logic structure of a DE model is captured diagrammatically either directly from the user's Visio drawing or from a descriptive list written in another Microsoft application and converted automatically into Event Graphs. The dynamic behaviour of a DE model, which is described by the state transition triggered by the events and the characteristics of the time that passes through the edges, is read upon dragging and dropping the stencil's masters onto the Event Graph or from the descriptive list of the model. This data is then stored in the tables of custom properties associated with each event and edge.

Thus, the modelling component of the DotNetSim prototype consists of a collection of VBA programs which provide tools for performing the following main operations:

- Capture the logic of a DE model diagrammatically in an Event Graph
- Traverse the Event Graph to determine the sequence of the events
- Read the data which describe its temporal behaviour
- Place the diagram and the modelling data into a relational database to facilitate the review of the model and its placement into other packages
- Run the DE simulation, invoking the DotNetSim simulation engine component

Additional generic tools were implemented to provide utilities and perform general operations on the visualisation of the diagram and the customisation of Visio™. The prototyping of this modelling component aims, however, to derive modelling tools from generic software tools built in widely-used Microsoft applications in order to enable DE models to be devised by allowing the integration of data originated in different Microsoft packages. Microsoft applications are then customised and integrated to create, by object-orientation, a modelling environment for DES systems. Microsoft Visio™ constitutes its central core, which manipulates the applications from which the modelling data is sourced and in which the models are placed. The DotNetSim modelling component supports the integration of Visio documents with Excel spreadsheets and Word and PowerPoint documents. The integration is bidirectional so that, for example, Excel™ reads the names of the masters currently in an Event Graph stencil and Visio™ places the description of its diagrams in Excel™. Integration with other applications, namely Project™ and Access™, would also be possible by applying the object-oriented principles that underline Microsoft Automation.

Fig. 6.1 depicts the DotNetSim prototype's architecture focussing on the

modelling component of the DotNetSim environment. The DotNetSim modelling component loads the Event Graph stencil and bi-directionally exchanges data with various Microsoft applications to capture the logic and dynamics of DE models in Event Graphs and relational databases. Eventually, it invokes the DotNetSim simulation engine which, after executing, returns control to the modelling component.

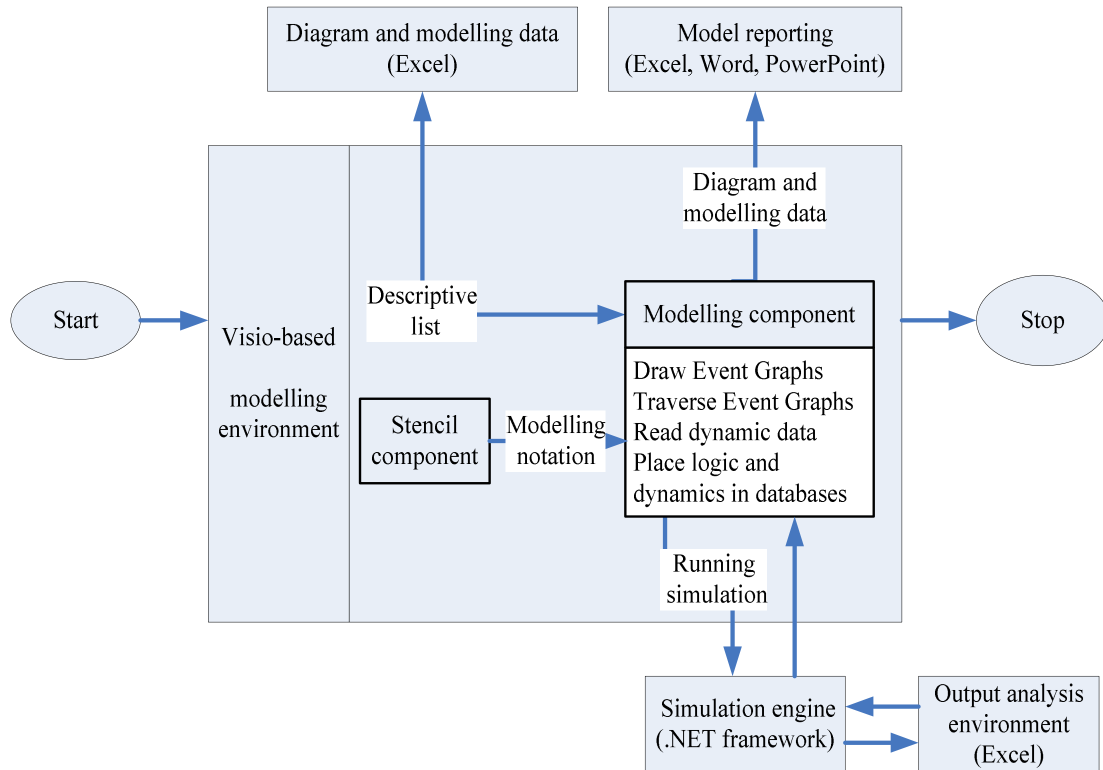

Fig. 6.1: Zooming the modelling component in DotNetSim's architecture. The execution of the DotNetSim starts and ends within the Visio-based modelling environment, passing through the stages 'Modelling – Simulation – Output analysis'

6.2. DOTNETSIM MODELLING ENVIRONMENT: THE MODELLING COMPONENT

The DotNetSim modelling component is implemented on the Visio template EGmodelling.vst, which is the container of the VBA components that implement the functionality required by the Event Graph paradigm for modelling and running DES systems. EGmodelling.vst is a class of document which is instantiable into new documents and the documents based on the EGmodelling template inherit the whole

following modelling functionality:

- Drawing the diagram: The diagram is drawn by the user in Visio drawing pages or it is automatically generated from a tabular description of the model in Excel™ or other Microsoft application.
- Reading the diagram: The diagram is traversed to number the events and the edges, to store the sequence of the events and to label the nodes and edges so that it complies with the Event Graph modelling notation.
- Describing the model: The modelling data such as the state variables, the state transition actions and the simulation parameters are prompted to the user.
- Reviewing the model: The diagram data and the modelling data are placed into a relational database in Visio, Excel™ or other appropriate Microsoft application in order to facilitate a synthesised review of the model.
- Reporting the model: Data is extracted from the diagram and listed in Excel™ and the diagram is pasted into Word documents and PowerPoint slides. Also, a summary of the model is output in a Word document and an Excel™ workbook.
- Performing general operations: The diagram can be resized, zoomed or deleted. Also, a pan and zoom window may be used to magnify part of the diagram.
- Simulating the model: The model is executed by an event-based simulation engine written in C#.

Apart from this functionality associated with Event Graph modelling, the EGmodelling.vst exploits Visio customisation by allowing selective loading or

visualisation of the Visio built-in tools. It also allows Excel files to be created and opened from within Visio™ and it demonstrates the invocation of an web-based help system also from within Visio™. The DotNetSim modelling component has no contextual user help or assistance, but a test HTML help was created and compiled by resorting to HTML HELP WORKSHOP [70, 72] to investigate the effectiveness of its integration with Visio™.

These functions are displayed in the Event Graphs menu shown in Fig. 6.2. This menu is generated and appended to the Visio menu bar on creating a drawing document based on the EGmodelling template.

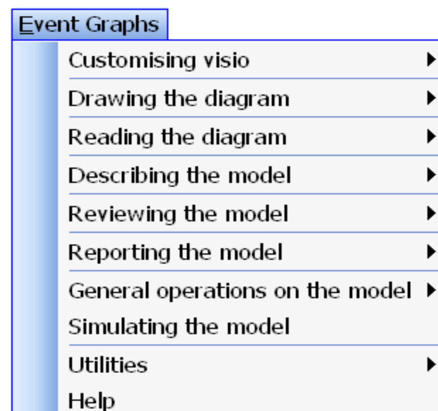

Fig. 6.2: Event Graphs menu

6.2.1. CUSTOMISING VISIO™

To limit the generic features of Visio™ to those needed by each DE model and also to maintain the simplicity of the graphical user interface, the EGmodelling template starts with just the basic features needed for modelling. It automatically unloads or hides the stencils, menus and tool bars that are not directly used by the Event Graphs. However, at any time, other tools can be activated if required, as shown in Fig. 6.3.

Other generic features of Visio™, such as those that display the properties of the diagram's pages and shapes, can be used by selecting the corresponding commands on the Event Graph menu.

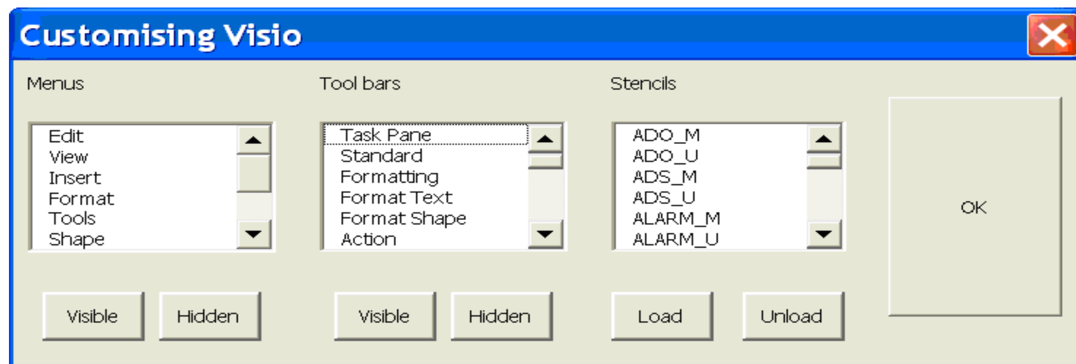

Fig. 6.3: The boxes list the menus, tool bars and stencils available for the current Visio installation

6.2.2. DRAWING THE DIAGRAM

Event Graphs are manually drawn in the Visio drawing pages or generated automatically from the corresponding descriptive lists managed by any Microsoft package which supports Automation. Both processes resort to the Event Graph stencil. The DotNetSim prototype implements the automatic generation of any Event Graphs described in Excel worksheets. Graph generation from other Microsoft applications such as Project™ or Word™ are similar, but have to comply with the respective application object model and its VBA variation.

An Excel template, EGdiagram.xlt, provides the format for describing Event Graphs. Excel files based on this template can be created from within Visio™ or may be created manually. The list format is basically the headings of the columns which correspond to the properties of the graph's events and edges as described for the master shapes. Two further fields must be included in this list: the name of the master which represents the event or the edge and the level – a function of the y-coordinate – which each event occupies in the drawing page. The Excel template contains VBA code to generate a listbox from the current masters in the Event Graph stencil and to display it whenever a master's name is to be entered. This event handler pops up the listbox shown in 6.4 whenever an Excel cell of the master name's column is selected.

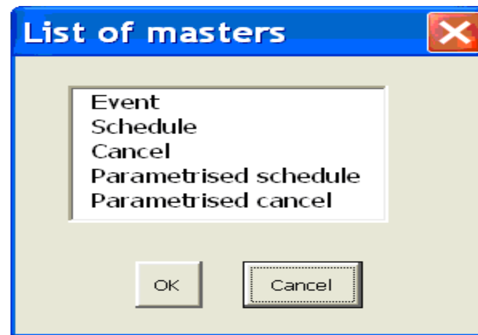

Fig. 6.4: ListBox generated in Excel™ from the masters currently in the Event Graph stencil

Creating and opening Excel files, based in this template from within Visio™, requires bidirectional data passing from Visio™ to Excel™ (creating and opening the Excel file from Visio™) and vice-versa (generating the list of the masters currently in the Event Graph stencil). Fig. 6.5 lists the VBA statements for instantiating Excel™ and creating an EGdiagram-based workbook from within Visio™.

```
Set xlapp = CreateObject("Excel.application")
Set Wb = xlapp.Workbooks.Add("C:\simengine\Exceltemplates\EGdiagram.xlt")
```

Fig. 6.5: VBA statements that create an EGdiagram-based workbook from within the Visio™

Fig. 6.6 lists VBA statements for instantiating Visio™ and reading the masters' names applicable to the Event Graph stencil from within Excel™.

```
Set vsapp = CreateObject("Visio.Application")
stencilEG= vsapp.StencilPaths + “Event-Graph.vss"
Set egst = vsapp.Documents.OpenEx(stencilEG, visOpenRW + visOpenDocked)
Set eust = egst.Masters.Item(1)
Set R = Sheets(1).Range("A2")
R.Cells(1,1) = eust.Name
```

Fig. 6.6: VBA statements that read a master's name of the Event Graph stencil from within Excel™

If working in Visio™ with a model file created in Excel™, VBA code instantiates Excel™, reads the descriptive list and generates the Event Graph. Simultaneously, the properties of the events and edges are stored in the corresponding shapes sheets. Table 6.1. displays part of the list read by Visio™ to generate the diagram shown in Fig. 6.7.

Number	Level	Node/Edge	Name	Description	Origin	Destination	Delay time
1	1	Event	Run	1st patient arrival			
2	1	Event	Arrival	Patients arrive every ta minutes			
3	1	Event	Reception	Recording patients			
4	2	Event	Start-Nurse	Preliminary screening			
5	2	Event	Finish-Nurse	End screening			
6	2	Event	Start-Doctor	Medical observation			
7	2	Event	Finish-Doctor	End observation			
8	3	Event	Leave	Leaving the emergency service			
9		Schedule			1	2	
10		Schedule			2	2	ta
11		Schedule			2	3	
12		Schedule			3	4	tn
13		Schedule			4	5	
14		Schedule			5	8	
15		Schedule			5	6	
16		Schedule			6	7	td
17		Schedule			7	8	td

Table 6.1: Data extracted from a created Excel file

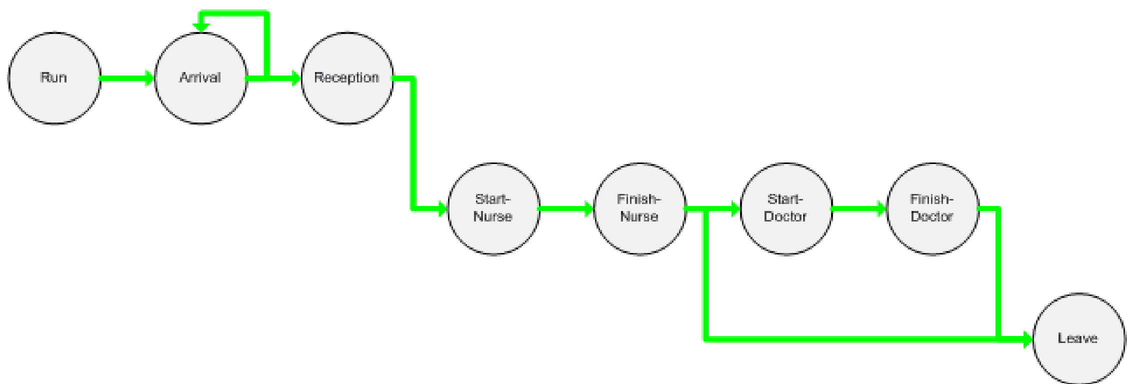

Fig. 6.7: Event Graph generated from table 6.1

6.2.3. READING THE DIAGRAM

Reading a diagram consists of traversing it so that the sequence of events is determined, i.e. the structure of the scheduling and the cancelling relationships between events is automatically captured into the edges' properties. Events and edges are numbered sequentially. The numbers of the events connected by each edge are stored in the origin and destination properties of the corresponding edge, as shown in

Fig. 6.8. Edges that do not link two events display an error that invites the user to revise their connection points.

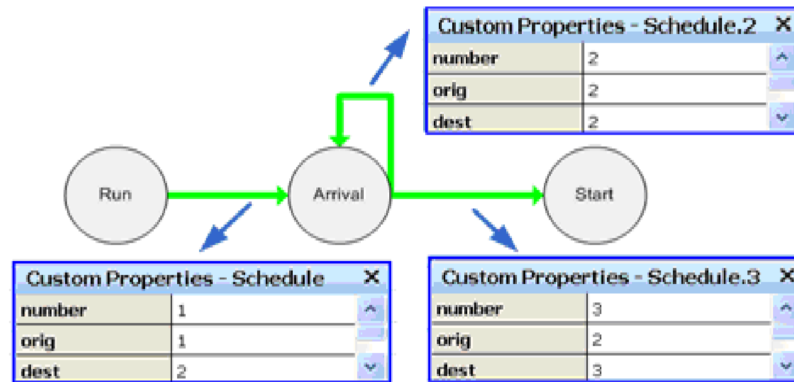

Fig. 6.8: Origin and destination properties of three edges of an Event Graph

Next, the diagram's events and edges are labelled to facilitate its understanding and to comply with the modelling notation of the Event Graphs. The name of the event and the parameter it passes to the adjacent edge are inscribed in the event's circle, as shown in Fig. 6.9. The event description and the state transition are displayed on double clicking the graph's nodes in conformity with the event handler registered in the master definition and implemented in the modelling template.

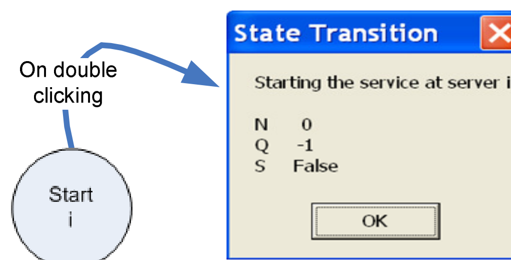

Fig. 6.9: State transition of an event

The labels of the edges display, at most, the variable that represents the delay time or its value if it is deterministic, the head-condition between brackets and the parameter that passes through the edge to the destination event. On double clicking a scheduling edge, the delay time or the parameters of the delay time's distribution function are prompted as shown in Fig. 6.10.

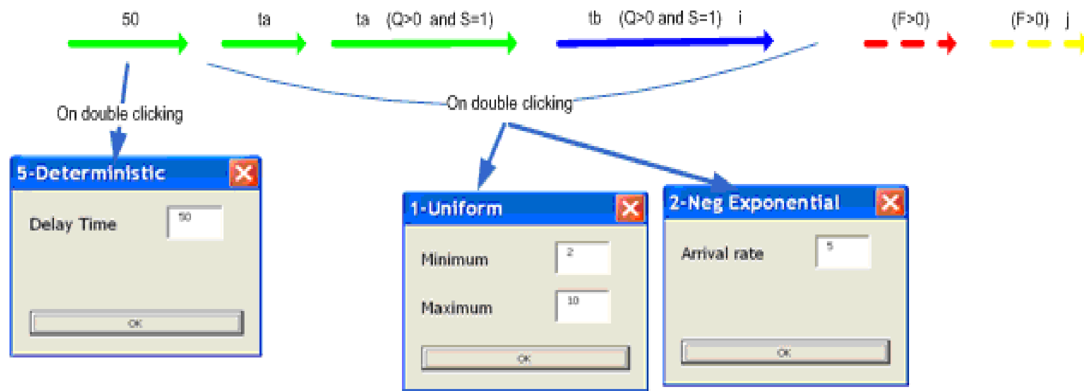

Fig. 6.10: Labelling of the edges

6.2.4. DESCRIBING THE MODEL

The modelling data such as the model name and description, the definition of state variables, the event transition actions and the simulation parameters are prompted to the user and stored as custom properties of the diagram's first page. This data is split into four parts which should be supplied in sequence:

- General attributes: Two forms are displayed in sequence to collect the name and description of the model. This can be easily extended to prompt and store other properties such as the modellers' names, creation and revision dates.

- State variables: The name, description, type and the maximum value of each state variable are prompted and stored as custom properties of the diagram's page. Four types of variables are allowed: integer, boolean, string and array of integers, as shown in Fig. 6.11.

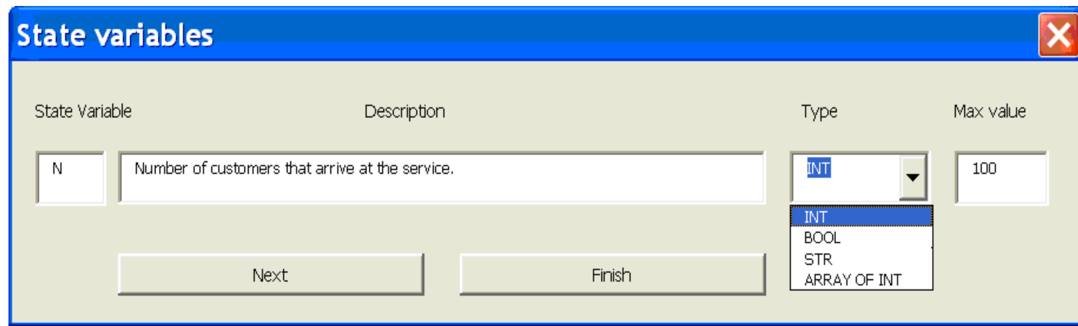

Fig. 6.11: User form that prompts the definition of a state variable

- State transition: The state transition definition displays for each event the set of its state variables so that these state changes can be defined (see Fig. 6.12).

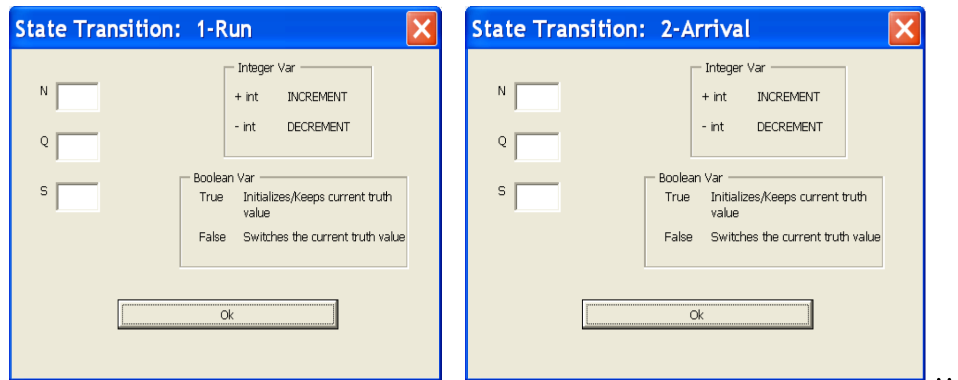

Fig. 6.12: Successive display of user forms to prompt the state transition triggered by each event

The custom properties of each event are extended to record all the state variables and the corresponding changes. Changes to integer variables are stored as increments or decrements; 0 means that the event does not change the state variable. Changes to boolean variables keep or switch the current truth value; true keeps the current truth value and false switches it. Changes to string variables represent their new values. Fig. 6.13 shows the state transition associated with the start of a service in a single server system.

Custom Properties - Event.3	
number	3
name	Start
Description	0
Parameter	0
N	0
Q	-1
S	FALSE

Fig. 6.13: In a single server system, starting serving a customer does not change the number of customers in the system, decrements the length of the queue as one customer is dequeued and changes the status of the server to busy

- Simulation parameters: Finally, the simulation parameters such as the run length, the random seed and the number of replications to execute are prompted as shown in Fig. 6.14.

Single server and single queue system

Simulation run length 100 Unit of time days

Random seed 2 Replications 5

Finish

Fig. 6.14: The simulation parameters as prompted for the single server system

Custom Properties - ThePage	
Modname	Single server system
Moddesc	Component of a DES System in which a single server provides
N	INT
DescN	Number of arrivals
MaxiN	500
Q	INT
DescQ	Length of the single queue
MaxiQ	500
S	BOOL
DescS	Status of the server {0-busy, 1-idle}
MaxiS	0
Run	500
Unit	hrs
Seed	2
Replic	10

Fig. 6.15: Modelling data stored for a single server system

The simulation parameters are also stored as custom properties of the diagram's first page. Fig. 6.15 illustrates how the modelling data is stored as

custom properties of the diagram's page.

6.2.5. REVIEWING THE MODEL

The diagram and its associated modelling data are now organised into a set of tables (shown in Fig. 6.16) which facilitates a synthesised review of the model. It also eases the placement of the model in other applications, e.g. Excel™.

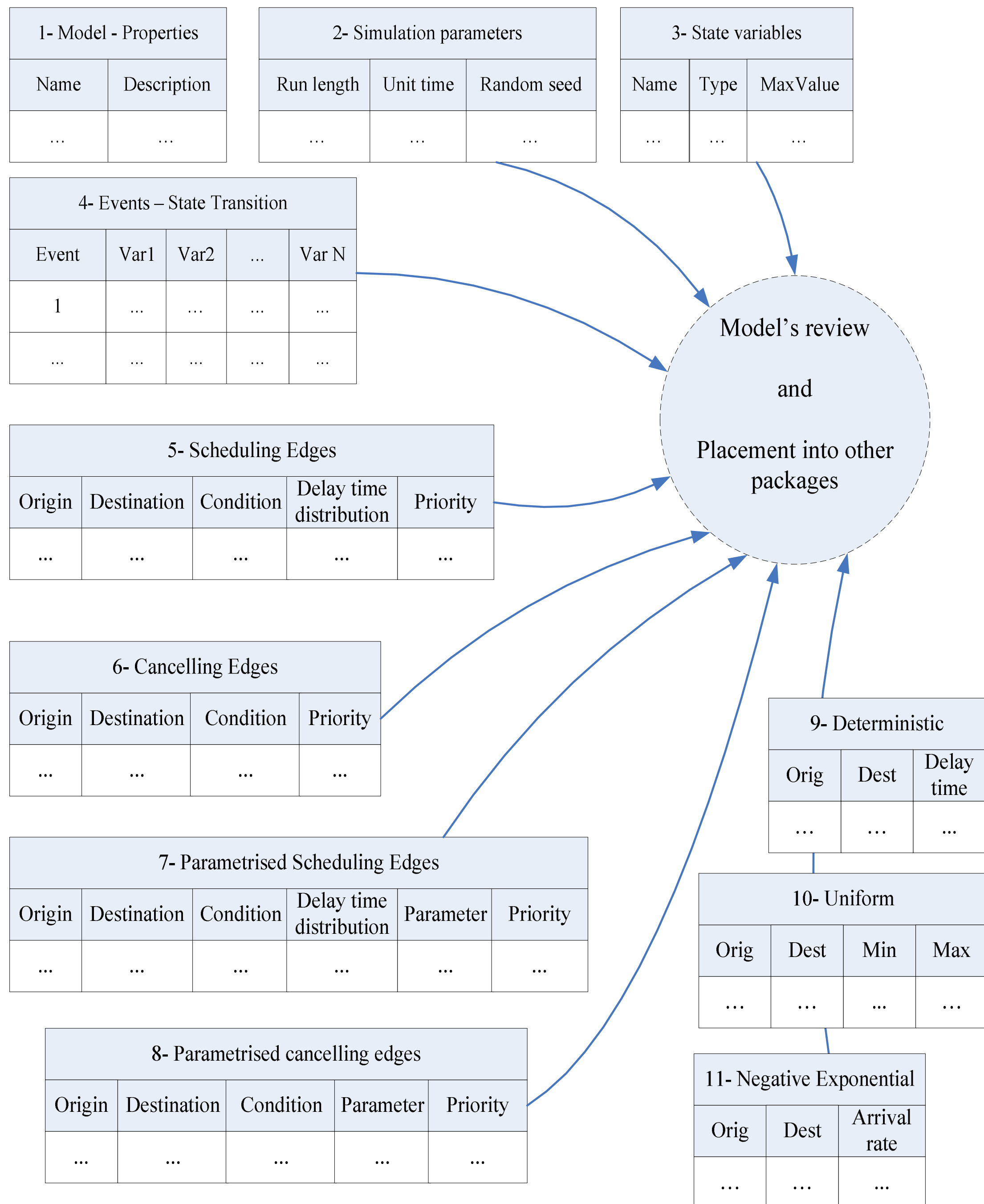

Fig. 6.16: Set of tables that store the data extracted from an Event Graph and corresponding modelling data

These tables are stored in arrays which are displayable either from within Visio^{TM}, as shown in Fig. 6.17 or in an Excel workbook. The latter is built on the EGmodel Excel template, especially created for this prototype and shows each table in a separate worksheet.

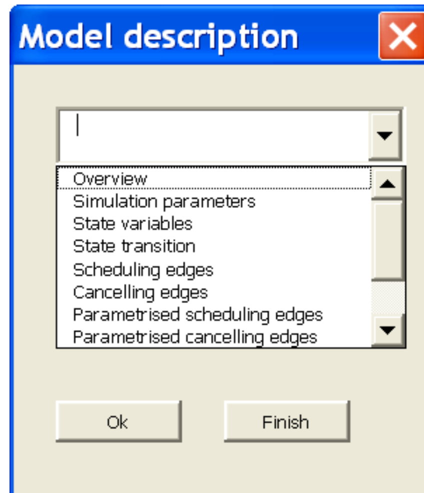

Fig. 6.17: List of the Visio arrays that describe the model and are displayable on a selection basis

To further facilitate the review of the model, the changes to the state variables can be highlighted along the Event Graph, i.e. each event is coloured to show the changes it triggers on a selected state variable. A toolbar, the tracing toolbar, is generated to allow the selection of a state variable and to show its changes in the diagram. The tracing toolbar displays the colours' legend and two buttons, V and R, for selecting a state variable and re-setting the original colours respectively. Fig. 6.18 depicts the tracing toolbar and the changes triggered on the state variable which stands for the length of the queue in a single server system.

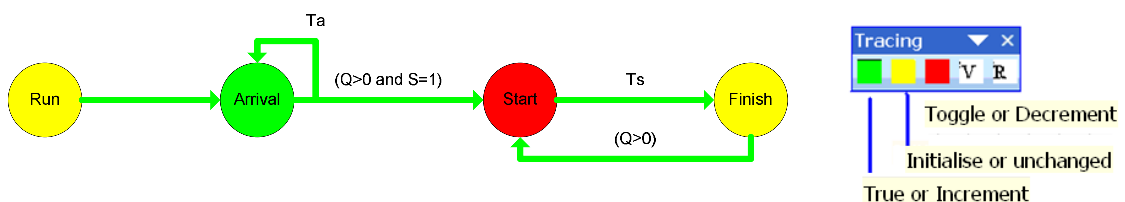

Fig. 6.18: Changes triggered on the length of the queue in a single server system

6.2.6. REPORTING THE MODEL

Inserting data in the model into written reports or slide presentations is done by manipulating the objects of other applications from within Visio™, so that Event Graphs and modelling data can be displayed in other Microsoft applications, namely Excel™, Word™ and PowerPoint™. The EGmodelling template provides VBA code which implements the following functionality:

- Listing a diagram into an Excel worksheet: The current diagram is read and converted into a list of properties such as the type of the shapes and their levels in the drawing page. This list, together with the custom properties of the shapes, is written in an Excel file based on the EGdiagram template. The Visio VBA procedures prompt the name of the Excel file, open the file and write the data in predefined columns (see Fig. 6.19).

```
Fich = InputBox("Diagram Excel file ")
Set sh = ActiveDocument.Pages(1).Shapes(1)
Set xlapp = CreateObject("Excel.application")
Set Wb = xlapp.Workbooks.Open(Fich)
Set r = Wb.Sheets(1).Range("D2")
r.Cells(1, 1) = sh.Cells("Prop.name").ResultStr("")
```

Fig. 6.19: VBA statements which write the name of a shape into a range of an Excel worksheet from within Visio™

  This is the inverse operation of the Event Graph generation described above.

- Copying a diagram to Word documents and PowerPoint presentations: The current diagram is copied and may be pasted either into new or existing Word files and PowerPoint files. It will be placed in the current position of the cursor. Fig. 6.20 was automatically pasted from the Visio active

document to this Word document.

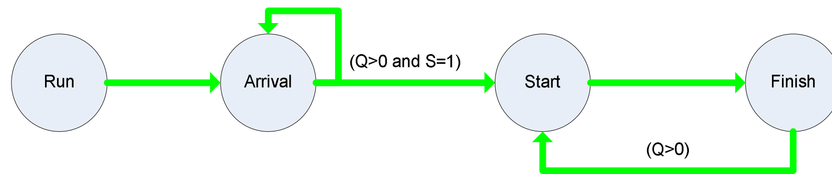

Fig. 6.20: This diagram was automatically pasted from the active Visio document

Visio™ selects the current diagram and either instantiates the Word™ or the PowerPoint™ applications or get references to already created objects; methods are invoked on these objects to instantiate new documents or get references to the active documents (see Fig. 6.21) and, finally, the diagram is pasted into them. This applies to other Microsoft applications that support automation. However, the invocation of the properties and methods exposed by each application's object model depends on the VBA variation.

```
Set Vs = Visio.Application
Set diagsel = Vs.ActiveWindow.Selection
diagsel.Copy
Set ppapp = GetObject(, "PowerPoint.application")
Set np = ppapp.ActivePresentation
Set pp = np.Slides(ppapp.ActiveWindow.Selection.SlideRange.SlideIndex)
pp.Shapes.Paste.Select
```

Fig. 6.21: VBA statements that get a reference to the active PowerPoint presentation and paste the diagram into it

- Summarising the model in a Word document: The properties of the model, namely its name and description, the state variables and their transition and the simulation parameters, are written into the active Word document. A set of Visio's VBA procedures reads the data from the custom properties of the drawing's page, gets a reference to the active Word document and writes the

data in the it. The VBA code could well be extended to define a style sheet for formatting this document from within Visio™.

6.2.7. GENERAL OPERATIONS ON THE MODEL

General operations on the diagram visualisation and the deletion of the current model were also coded in VBA so that they can be executed from the Event Graph menu. The diagram, or parts of it, can be resized or zoomed to improve its readability. Thus, the whole diagram can be enlarged or reduced, its edges can be resized to make their labels easier to read and parts of the diagram can be selected in a pan window and zoomed as shown in Fig. 6.22. Also, the current diagram can be deleted and its custom properties removed to erase the corresponding modelling data.

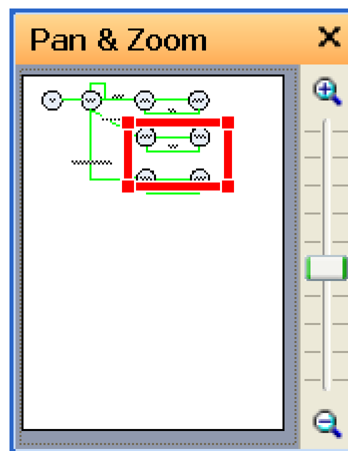

Fig 6.22: Pan and zoom window to select the part of the diagram to be zoomed

6.2.8. SIMULATING THE MODEL

The model data is read by the simulation engine described in detail in chapter 7. This prototype implements an event-based simulation engine in C#, but other schemas can be substituted. The simulation engine gets a reference to the Visio-based DE model and reads the data that defines its logic and dynamics by invoking the appropriate methods of the custom properties. The simulation results are returned to an Excel workbook for data analysis and reporting (see chapter 8).

6.2.9. UTILITIES

Excel files based on EGdiagram and EGmodel templates can be created, opened and closed from within the EGmodelling-based Visio documents. These utilities are executed by Visio's VBA modules which instantiate the Excel™ application and invoke its methods to perform general operations on Excel files.

6.2.10. HELP

As a prototype, the DotNetSim modelling component has no contextual user help or assistance, though this could obviously be added. To demonstrate this, a test help web site (see Fig. 6.23) was created and compiled using the HTML HELP WORKSHOP which is an authoring tool from Microsoft [70, 72]. This help system runs from within EGmodelling Visio template.

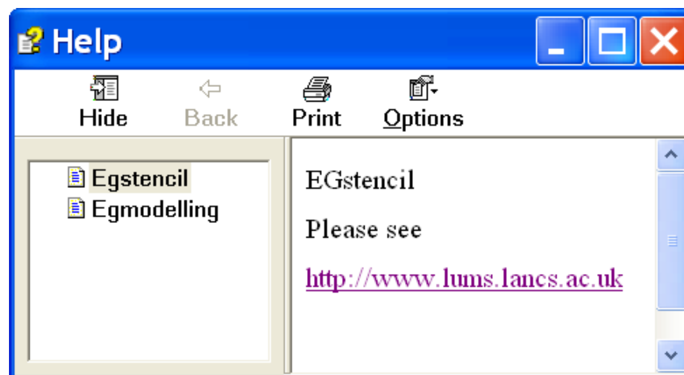

Fig. 6.23: HTML help that is launched by a Visio VBA module

6.3. COMMENTS ON THE IMPLEMENTATION OF THE MODELLING COMPONENT

The modelling component of the DotNetSim prototype provides an environment in which DE models can be defined by integrating data originated in different Microsoft packages. Microsoft Visio™ provides the basis for its central core, which manipulates the applications from which the modelling data is sourced and in which the models are

placed. DotNetSim supports the integration of Visio documents with Excel spreadsheets and Word and PowerPoint documents. The integration is bidirectional so that, for example, Excel™ reads the names of the masters currently in an Event Graph stencil and Visio™ places the description of its diagrams in Excel™. Integration with other applications, namely Project™ and Access™, would also be possible by applying the object-oriented principles which underline Microsoft Automation.

6.3.1. PACKAGES AS OBJECTS OF OTHER PACKAGES

Simple purpose-written VBA modules allow the instantiation of other applications within Visio™. These applications are presented to Visio™ as abstract classes whose instantiations become the entry points to the corresponding hierarchical object models. By setting a variable of the type of an application, Visio™ gets a reference to that application, which, in fact, is the root of a tree of sub-classes and it therefore acquires the capability of manipulating the whole hierarchy of objects. Fig. 6.24 illustrates how Visio™ extends its functionality by referencing the object model of another application. By instantiating Excel™, Visio™ can set and get the properties of Excel's objects and invoke their methods. The EGmodelling prototype uses this capability to allow Visio™ to collect modelling data from Excel worksheets to automatically draw diagrams on this data and to insert data and diagrams in Word documents and PowerPoint presentations.

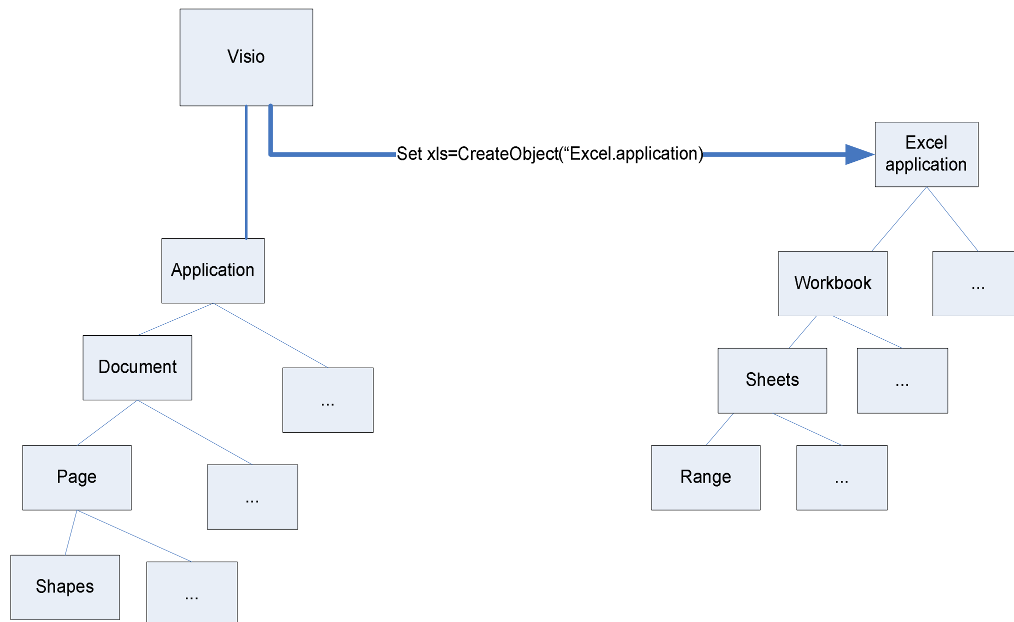

Fig. 6.24: By instantiating Excel™, Visio™ gains control over the object model of an instance of Excel™

The Visio™ ability to manipulate other applications facilitates the gradual refinement of DE models. A diagram, for example, can be drawn and re-drawn automatically by Visio™ as additional data is typed in Excel™. The following process may be iterated until a satisfactory graphical representation of the model is attained:

1. Initially, a diagram is sketched in elementary form in Visio™, by dragging and dropping events and edges on the drawing page;
2. Visio™ extracts the modelling data from this sketch, causal relationships included, and writes it in an Excel sheet.
3. The modeller types additional data in the Excel sheet
4. Visio™ collects this data and automatically re-draw the diagram

Thus, diagrams can be successively refined by combining sketches with data lists, which serves the gradual development of the models as recommended by the principle of parsimony [85, 87].

The DotNetSim prototype also relies on instantiating packages as objects of other

packages to generate reports on the models. Diagrams and modelling data are sent by Visio™ to Word™ and PowerPoint™ by invoking the appropriate methods on the objects of these applications' objects. For example, Word documents and Word tables, PowerPoint presentations and slides are created by Visio™, which also places on them diagrams and modelling data.

6.3.2. APPLICATIONS OBJECT MODELS

The object models built into the widely-used Microsoft applications are hierarchical structures of classes of objects rooted in the applications themselves. The whole models unfold into subclasses which successively define objects as elements of their containers. A shape in Visio™ and a range of cells in Excel are, for example, defined as objects contained respectively within a page and a worksheet. Thus, all the applications which this modelling prototype integrates have classes of objects that derive from others by inheritance and which encapsulate their implementation from other programming figures. It might be argued that, apart from the absence of polymorphism, the object models are fully object-oriented in all these applications. However, the object-orientation does not go beyond the built-in objects in any of these packages. The new classes created by the user are just containers of properties and methods that, at most, can be nested. This incomplete object-orientation was a major drawback in the implementation of the DotNetSim modelling component. The different variants of the edges, for example, could not be defined as a substructure of classes and subclasses.

The object model of Visio™ is, in some ways, more favourable for user-defined data structures than other Microsoft packages. Some objects, such as pages and shapes, have sheets of properties attached to them. These sheets consist of sets of three-dimensional arrays of properties organised in sections, rows and columns.

Additional sections and rows can be added by the user in certain circumstances, but the number of columns is always fixed.

In spite of its fixed structure, the custom properties section attached either to the diagram pages or the shapes was frequently used in this prototype to define the attributes of the DE model and the Event Graphs modelling notation. The custom properties operate as user-defined data types, i.e. value types that contain the data itself. They cannot, however, reference objects on a heap so a custom property cannot be the entry-point to a dynamic structure of related properties. For example, the custom property which stores the name of the delay time distribution cannot point to a class that characterises the appropriate distribution function.

The events section attached to the shapes was also used in this modelling prototype to define the shapes' response to certain events, such as double clicking a shape or dropping a new shape. For example, in this prototype, double clicking a scheduling edge prompts the user for the parameters of the delay time distribution.

6.3.3. VISUAL BASIC FOR APPLICATIONS

The negative side of the implementation of this prototype is the VBA programming language. VBA was used because we intended to integrate VBA-enabled applications by resorting to the generic development tools supported by the corresponding vendor. There may be several reasons for Microsoft to use VBA as the language for customising and extending the functionalities of many of its packages [76] and, among them, is that VBA has a simple and friendly development environment. However, VBA, due to its procedural nature and to the lack of proper dynamic data structures, imposes a huge programming effort to implement more complex algorithmic solutions. VBA classes, modules and user forms are useful for encapsulating code but they lack inheritance and polymorphism primitives. In

addition, the syntax incoherence of VBA is a major delay obstacle in manipulating objects within each application and across them. Actually, the different syntax applied to different object models contributes to the existence of several VBA dialects, one per application.

Nonetheless, VBA allows the integration of the different packages. The development of this modelling prototype showed that the integration of widely-used Microsoft packages is possible and may serve the purposes of different DE modelling graphical methodologies. A template was developed to create an Event Graph based modelling environment, other templates may be developed for other graphical modelling paradigms and to suit other simulation worldviews.

6.4. THE CHAPTER IN CONTEXT

This chapter describes the modelling component of the DotNetSim prototype which together with the Event Graph stencil component forms the DotNetSim graphical modelling environment. The DotNetSim modelling component is an extension of the Visio™ which uses the Event Graph stencil and OO integrates with other Microsoft applications to provide the functionality required to capture the logic of a DE model as an Event Graph. The diagrammatical and the modelling data are placed in a relational database which the DotNetSim simulation engine component reads by applying the OOP principles.

The next chapter discusses the implementation of the DotNetSim simulation engine component which prototypes an event-based simulation executive that runs, through simulated time, the model captured within the DotNetSim modelling environment.