Simulating Dynamic Systems
With Event Relationship Graphs

Donna L. Schruben
Lee W. Schruben

9th – Edition

SIGMA code download at www.sigmawiki.com

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

CONTENTS

SIGMA v
Hardware Recommendation v
File Name Extensions v

Chapter 1 SIGMA: Overview & Installation 1

1.1 The SIGMA Modeling Environment 1
1.2 Single-step Installing and Uninstalling SIGMA 1
1.3 A Quick Tour of SIGMA 2

Chapter 2 Discrete Event System Modeling 6

2.1 Background and Terminology for Systems Modeling 6
2.1.1 Systems 6
2.1.2 Models 8
2.1.3 Model Verification 9
2.2 Discrete Event Systems and Simulations 10
2.3 Event Graphs 13
2.4 Verbal Event Graph 16
2.5 Visual Power of Event Graphs 17
2.6 SIGMA 17
2.7 Exercises 18

Chapter 3 A Tutorial on the Basics of SIGMA 21

3.1 Starting a SIGMA Session 21
3.2 The SIGMA Environment 21
3.3 Exploring Our Carwash Model 22
3.3.1 State Variables 24
3.3.2 Vertices 24
3.3.3 Edges 25
3.3.4 Editing the Carwash Model 27
3.4 Using Text Files 27

Chapter 4 Running A SIGMA Simulation 29

4.1 Running the Model 29
4.2 Run Options 31
4.2.1 Description 31
4.2.2 Output File 31
4.2.3 Run Modes 31
4.2.4 Ending Conditions 32
4.2.5 Trace Variables 32
4.2.6 Initial Attributes 32
4.2.7 Random Seed 33
4.2.8 Output Plot 33
4.2.9 Command Buttons 35
4.3 Exercises 35

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Chapter 5 Event Graph Modeling 37

5.1 Enrichments to Our Basic Model 37
5.1.1 Multiple Identical Parallel Servers: BANK1.MOD 37
5.1.2 Batched Service: BATCHSIZ.MOD 39
5.1.3 Rework: REWORK1.MOD 39
5.1.4 Limited Waiting Space: BUFFERQ.MOD 40
5.1.5 Assembly Operations: ASSEMKIT.MOD 42
5.1.6 Different Servers Working in Parallel: SLOFAST0.MOD SLOFAST1.MOD 42
5.1.7 Periodic Resource Unavailability: FAILURE.MOD 43
5.2 Event Cancellation 45
5.2.1 Closing Time: CLOSEIT.MOD 45
5.2.2 Server with Intermittent Service Failures: BRKDN.MOD 46
5.3 Event Parameters and Edge Attributes 48
5.3.1 Many Servers of Many Types: SLOFAST2.MOD 49
5.3.2 Multiple Servers, One Line, Waiting Times: BANK2.MOD 50
5.3.3 Limited Rework: REWORK2.MOD 54
5.3.4 Generalized Assembly Operations: ASSEMKIT.MOD 54
5.3.5 Sequential Service with Blocking: TWOQUES.MOD, TWOQUES1.MOD, and TANDQ.MOD 55
5.4 Multiple Resources 56
5.5 A Problem in Finance: BONDRATE.MOD 56
5.6 Project Management (PERT/CPM): PERT.MOD 58
5.7 Modeling Transient Entities 60
5.7.1 Little's Law 60
5.7.2 Generalizing the Notion behind Little's Law: DELAY.MOD 61
5.8 Programming with Event Graphs 62
5.8.1 Boolean Variables 62
5.8.2 Conditional Expressions (If-Then-Else) 63
5.8.3 Do, While, and Nested Loops 64
5.9 Model Complexity and Model Size 66
5.10 Continuous Time Simulations: FISHTANK.MOD 66
5.11 Process Modeling 80
5.11.1 GPSS 68
5.11.2 SIMAN.MOD, SIMAN1.MOD and SIMAN2.MOD 69
5.11.3 Petri Net Simulation: PETRINET.MOD 70
5.12 Exercises 71

Chapter 6 Building Models, Verifying Simulations, and Sharing the Results of Simulation Experiments 77

6.1 Creating and Editing an Event Graph 77
6.1.1 Create Process Mode 77
6.1.2 Create Single Edge Mode 77
6.1.3 State Variables 77
6.1.4 Editing Vertices 78
6.1.5 Editing Edges 79
6.1.6 Moving a Simulation Graph 81
6.1.7 Copying Event Graph Models 81
6.1.7.1 SIGMA User Tools 82
6.1.8 Saving SIGMA Models and Output Plots 82

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

6.2 Dynamic Run-Time Model Building and Analysis 82
6.2.1 Changing Value of State Variable Values During Run 83
6.2.2 Changing Edges and Vertices During Run 84
6.2.3 Adding and Deleting Edges/Vertices During Run 84
6.3 Model Enrichment and Logic Checking 85
6.3.1 Setting up the Logic Checking Environment 85
6.3.2 Starting a Logic Checking Run 85
6.4 Automatic Translation of SIGMA Model 85
6.4.1 English Translation 85
6.4.2 Translate to C 86
6.5 Capturing a Simulation and Its Results 87
6.5.1 Printing Event Graphs, Simulation Plots, and Output Files 87
6.5.2 Using Spreadsheets and Word Processors 87
6.6 Exercises 87

Chapter 7 Using SIGMA Functions 91

7.1 Summary of SIGMA Functions 91
7.2 Reading Data (and Code) from Your Disk 93
7.2.1 Reading Data from Tables 94
7.2.2 Changing Code from Data files 94
7.2.3 Trace Driven Simulations 95
7.3 Interactive Execution 95
7.4 Bookkeeping Functions 95
7.5 Mathematics Functions 96
7.6 Cycling using the MOD function 96
7.7 Using Ranked Lists 97
7.7.1 Example: Sorting Data (SORT.MOD) 98
7.7.2 Example: A Priority Queue (PRIORITYQ.MOD) 98
7.7.3 Example: Time-Constrained Processing (TIMEOUT.MOD) 99
7.7.4 Example: A General Network of Queues or Jobshop 100
7.7.4.1 NETWORK.MOD 101
7.7.5 Example: Using the Conditional GET function, CGET 105
7.8 Generating Random Variables 107
7.9 Statistical Functions 107
7.10 Exercises 109

Chapter 8 Building Animations 115

8.1 Perspective and Basic Principles 115
8.2 Classes of Animated Objects 115
8.3 Tutorial: Animating Resident Entities 118
8.4 Tutorial Continuation: Animating Transient Entity Motion. 122

Chapter 9 Modeling Input Processes 127

9.1 Randomness 127
9.2 Trace Driven Simulations 127
9.3 Random Number Generators 128

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

9.4 Using Empirical Input Distributions 129
9.5 Using Parametric Input Distributions 129
9.6 Modeling Dependent Input 131
9.7 Sources of Data 131
9.8 Variance Reduction 133
9.9 Using Multiple Random Number Streams 133
9.10 Methods for Generating Random Variates 133
9.10.1 Distribution Function Inversion 133
9.10.2 Other Methods 135
9.10.3 Generating Non-Homogeneous Poisson Processes 135
9.11 Exercises 137

Chapter 10 Graphical & Statistical & Output Analysis 139

10.1 Keeping a Perspective 139
10.2 Elementary Output Charts 140
10.2.1 Step and Line Plots 149
10.2.2 Array Plots 141
10.2.3 Scatter Plots 141
10.2.4 Histograms 142
10.3 Advanced Graphical Analysis 142
10.3.1 Detecting Trends using Standardized Time Series 142
10.3.2 Dependencies 145
10.4 Using Statistics 145
10.5 Standardized Time Series 146

Chapter 11 Generating Source Code for SIGMA Models and Running Simulations from a Spread Sheet 151

11.1 Generating C Programs from SIGMA Models 151
11.2 Compiling SIGMA-generated C code 152
11.3 Running Large Experiments using Batch Files 153
11.4 Running a SIGMA simulation from a Spread Sheet 155
11.4.1 A Simple Spreadsheet User Interface 156
11.4.2 A More Elegant Spreadsheet Interface using VBA 161
11.4.3 Tutorial for the Service Center Spreadsheet Simulator 164
11.4.4 Creating the Interface 165
11.5 Replacing the SIGMA Pseudo-Random Number Generator 194
11.6 Exercises 194

Chapter 12 Advanced Programming Techniques 196

12.1 General Starting and Ending Conditions 196
12.2 Simultaneous Parallel Replications 196
12.3 Event Graph Reduction 198
12.4 Using Arrays of Arbitrary Dimension 200
12.5 Eliminating Event Scheduling 201
12.6 Exercise 202

Appendix A: Event Execution Sequence 204

Appendix B: Reading SIGMA-Generated C Programs 206

Appendix C: Overview of Visual Basic for Applications 221

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

SIGMA

SIGMA is based on the simple and intuitive Event Relationship Graph (sometimes called an ERG or Event Graph) approach to simulation modeling. The SIGMA project began as an effort to implement the notion of Event Relationship Graphs on personal computers and has evolved into a powerful and practical method for simulation modeling. SIGMA, the Simulation Graphical Modeling and Analysis system, is an integrated, interactive approach to building, testing, animating, and experimenting with discrete event simulations, while they are running. SIGMA is specifically designed to make the fundamentals of simulation modeling and analysis easy**.** SIGMA is able to translate a simulation model automatically into fast C source code that can be compiled and linked to the sigmalib.lib library to run from a spreadsheet or web interface. SIGMA can also write a description of a simulation model in English. SIGMA was developed without external or University funding.

Hardware Recommendations

SIGMA runs on specific versions of Microsoft Windows (Version 3.1/9+/0+/NT/XP or previous). Any computer capable of running these versions of Windows will run SIGMA. To take full advantage of SIGMA, the Windows' spreadsheet program, Excel, and a word processor are useful. A Microsoft Visual C/C++ Compiler Version 6.0 will allow you to compile SIGMA-generated C programs; there is no guarantee the compiled C code will run in other operating systems.

File Name Extensions

The specific function of SIGMA files are identified by the extensions to their filenames as in following table. .

Extension	*File Type Indicated*
`.BAK`	Backup copy of the last saved model
`.BAT`	Batch files
`.BMP`	Bitmaps used for animations
`.C`	Source code in C
`.DAT`	Data files for SIGMA models
`.EXE`	Executable programs
`.EXP`	Experiment file for batched runs (chpt. 11)
`.H`	Source code in C
`.LIB`	SIGMA C Function Library
`.MOD`	Sample SIGMA models
`.OUT`	SIGMA output files (if any)

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

1

SIGMA: Overview & Installation

Chapter 1 begins with an overview of the new SIGMA software system. Next, instructions for installing the software are given. This is followed by a brief tour of SIGMA, which will acquaint you with some of its basic features.

1.1 The SIGMA Modeling Environment

SIGMA is a unique and powerful simulation environment. Developed primarily for discrete event simulations, SIGMA has been proven able to represent any computer program, with modeling power referred to in computer science as Turing Complete.

SIGMA is based on the concept of an Event (Relationship) Graph. Event Graphs graphically capture the events taking place within a system and the relationships among these events. While Event Graphs may look similar to flow graphs, they are very different: Event Graphs are *relationship* graphs. Thus, a simple model can represent a very large and complex system.

SIGMA's most striking feature is that simulation models can be created, enriched, and edited *while they are running*. Events can be added, altered, or even deleted during a simulation run. Logic can be changed and errors corrected without stopping a run to change code and recompile. You can even pause and "replay" interesting events. Using SIGMA, a simulation model can be developed and verified in a fraction of the time it would take using conventional simulation languages.

Animation support is fundamentally different in SIGMA than in other simulation modeling environments. Animations are not created from simulation models using conventional add-on software; in SIGMA, the *animation and the simulation model are identical.*

In addition to graphical modeling, analysis, and animation, SIGMA also includes state-of-the-art graphical data tracking tools and allows pictures, graphs, plots, and data to be pasted into spreadsheets and word processors.

For speed and portability, SIGMA models can be automatically translated (with a mouse click) into a fast C code. Not only does this code allow models to run thousands of times faster, models can then be run from a spreadsheet and multiple experimental runs can be batched together. A SIGMA model can even write a description of itself in English.

Multiple SIGMA sessions can be run concurrently. You can copy and paste objects from one modeling session to another. In fact, models can be developed in one SIGMA session and then graphically integrated into another simulation model while that model is executing.

SIGMA supports the full simulation model life cycle: from model building and testing to output analysis, animation, documentation, and report writing. Discrete event simulation model building has never been easier, and the results from simulations have never before been so easy to observe and understand.

1.2 Single-step Installing and Uninstalling SIGMA

If you already have a copy of `SIGMA.EXE` on your computer, you may need to replace it with an updated version. If you have the installation CD, follow the READTHIS.TXT instructions as follows.

INSTALL: Copy the entire CD into your C drive (creating a folder called C:\SIGMA with three subfolders).

UNINSTALL: Simply erase the C:\SIGMA folder and all subfolders.

Sigma does not write anything into the Windows Operating System Registry.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

1.3 A Quick Tour of SIGMA

Chapter 3 contains detailed instructions for using SIGMA. To get a brief overview of the software, try the following steps:

1. *Start a* SIGMA *Session*. Click the Start button on the taskbar. Locate SIGMA from among your programs and click on the SIGMA button. The window that appears is the *simulation graph window*. You are now in SIGMA.

2. *Get into* Select or Edit *mode*. Your mouse pointer will look like a plus sign enclosed by a circle, ⊕, when it is in the *simulation graph window*. This shape indicates the Create Process mode. If you click the right mouse button, the mouse pointer will assume an arrow shape,⇱, which indicates Select or Edit mode. Click the right mouse button and get into Select or Edit mode.

3. *Read in a model*. Click once on the File menu, move the mouse over the Open command, and then click on the Event Graph command. A dialog box will appear. Scroll through the list of file names until you see `CARWASH.MOD`. Double-click on this model, which is a simple simulation of an automatic carwash with only two state variables: `QUEUE`, the number of cars waiting in line, and `SERVER`, the status of the machine, `0 = BUSY` and `1 = IDLE`. This model is represented in Figure 1.1.

NETWORK USERS: If you are running SIGMA from a network server, your default drive is probably write-protected. Thus, you will get a system error message if you run the simulation as described in Step 4. You must add a drive letter before the output file name in the Run Options dialog box. For example, you should change `UNTITLED.OUT` to `C:\UNTITLED.OUT` so you can write the output to your C: drive. (Click on the Options command under the Run menu. In the Run Options dialog box, add the drive letter to the name of the file in the Output File text box. Next, click on the OK button and proceed to Step 4.) You should not use a floppy drive as your default drive as this will slow the running of your simulation considerably.

Figure 1.1: A Simulation Model of a Carwash, `CARWASH.MOD`

WARNING: *You should not use a floppy drive or a write-protected network drive as your default drive.*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

4. *Run the simulation*. Click once on the Options command in the Run menu. The Run Options dialog box will appear. Click the OK & Run button at the bottom of this dialog box to start the run. The green button located at the bottom right also starts the run. (Respond OK if a dialog box appears with the question "Replace existing UNTITLED.OUT?" and No if asked to "Save changes to CARWASH.MOD.") As the model executes, you will see a logical animation of the simulation graph. (Notice that the value of the variable, QUEUE, is displayed above the ENTER vertex.) A simulation plot showing the graphical output for CARWASH.MOD will also appear. The numerical output will be written on your disk in a file called UNTITLED.OUT. To incrementally slow execution, press the [F2]. Key; press [F3]. to resume speed.

5. *View output file*. After the simulation run is complete, a dialog box will appear asking if you want to view the output trace now. Respond by clicking the Yes button.

   The output from the simulation will appear in a third window titled UNTITLED.OUT. To see this material in detail click the Maximize button. Afterward click the output Close button. Press [⇧Shift] and [F4] to return to the simulation graph and simulation plot windows or press the Windows/Tile command.

6. *Change the type of output plot*. Double-click anywhere within the area of the *simulation plot* window. An Output Plots dialog box will appear. Click the down arrow for the Plot Type drop-down list to see the various plot types available to you; and click on Histogram. Next, click on the OK button at the bottom of the dialog box. The output will be graphically represented as a histogram. Open the Output Plots dialog box again by double-clicking on the histogram and change the plot type back to Step Plot.

7. *Translate model to source code*. Activate the *simulation graph* window by clicking anywhere within that window. Next, click on the Translate command in the File menu. Two translation choices are available: English and C. You can translate your model with the click of a mouse. When you click on the English option; a dialog box will appear with the file name highlighted (CARWASH). Click the OK button to confirm the file name. Respond Yes if you are asked to replace an existing file and Yes when asked to see the translation now. Use the scroll bars to view the entire file and the Close button to close the translated file window. Next press the Windows/Tile command to view all the simulation graph and simulations plot windows

8. *Examine the Event Graph*. To see the details of the carwash Event Graph, double-click on the vertices (balls) and edges (arrows) in the *simulation graph* window. Click the Cancel button at the bottom of each Edit Vertex or Edit Edge dialog box, or click Close in the multiple edge dialog box you see after clicking on the double edge between the START and LEAVE vertices.

9. *Create another vertex while the model is running*. With the *simulation graph* window active, open the Run Options dialog box under the Run menu. Increase the Stop Time to 10000 by dragging the mouse across the Stop Time text box and entering 10000. Press the OK & Run button to start the simulation. Click OK if asked to replace existing CARWASH.OUT and NO if asked to save changes to the model. You may want to close the Trace Window, if it is active, to get a better view of the plot window. Look at the value of QUEUE in the *simulation plot* window. Click anywhere within the *simulation graph* window to activate it, and then click the right mouse button to get into Create Process mode. Click the left mouse button once to add a vertex. Click the right mouse button to get back into Select or Edit mode. Double-click on the newly-created vertex to open the Edit Vertex dialog box. Under the State Change(s) text box, enter QUEUE=80; then press Execute. The QUEUE in the simulation plot window will increase dramatically. Click the red Stop Run tool at the lower right of the screen to stop the run. Press [←Enter] several times to confirm ending the run and decline when asked to view output.

10. *Run an animation*. Click the Maximize button for the *simulation graph* window. Click the Open/Event Graph command under the File menu. Press No when asked to save changes to CARWASH.MOD. Locate the model called ROBOT.MOD and double-click on it to open it. Press the green Start Run tool to begin the simulation. If asked to replace existing ROBOT.OUT, press OK. Press Yes if a warning message appears. If you have a fast machine, you may want to slow down the animation by pressing [F2]. You also may want to move the Trace Window if it is in

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

the way. Press the red Stop Run tool to end the simulation. Press [←Enter] in response to the messages that appear at the end of the simulation.

11. Another visual representation of the simulation is as a process flowchart, run `FLOWPROC.MOD` to see how this looks. Double clicking on a vertex and changing its shape from a circle to a desired flowcharting symbol in the dropdown box is all that is required to create process flow charts.

12. *Exit* SIGMA. Exit by clicking on the SIGMA Close button. Do *not* save your model! Make sure to click the No button, when asked to save changes to the model.

You are now ready to learn how to build your own simulations with SIGMA. This can be a great deal of fun. Enjoy!

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

2

Discrete Event System Modeling

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it

FIRST LAW OF MENTAT*

This chapter contains an introduction to the key concepts and terminology of discrete event simulation. The event graph, a method of concisely organizing the elements of a discrete event simulation, is introduced. Using a simple waiting line as an example, an elementary event graph is developed and explained. The future events list, which is the master scheduler of events in a discrete event simulation, is examined in detail. A verbal description of an event graph is introduced as a first step in developing a formal event graph.

2.1 Background and Terminology for Systems Modeling

Here we will use computer simulation to study the dynamic behavior of systems –i.e., how systems change over time. Our focus will be on those systems where the status of a system changes at a particular instant of time; such systems are called discrete event systems. Discrete event systems can be found in areas as diverse as manufacturing, transportation, computing, communications, finance, medicine, and agriculture. Engineers, scientists, managers, and planners use simulation methodologies to design and test new systems and to evaluate existing ones, thus avoiding the expense and risks of physical prototypes and pilot studies.

2.1.1 Systems

It will be sufficient for our purposes to define a system as

> *a collection of entities that interact with a common purpose according to sets of laws and policies.*

The system may already exist, or it may be proposed. Using simulation, even theoretical systems can be studied. We intentionally do not define a system by the specific entities in it. Rather, we define a system by its purpose. Thus, we speak of a communications system, a health care system, a production system, etc. Using a *functional* definition of a system helps us avoid thinking of a system as having a fixed structure. Consequently, we view a system in terms of how it ought to function rather than how it has traditionally worked in the past. To design a new system it is necessary to free our thinking from the status quo.

The entities making up the system may be either physical or mathematical. A physical entity might be a patient in a hospital or a part in a factory; a mathematical entity might be a variable in an equation.

When developing simulation models of systems, it is useful to classify entities as being either *resident* entities or *transient* entities. Resident entities remain part of the system for long intervals of time, whereas transient entities enter into and depart from the system with relative frequency. In a factory, a resident entity might be a machine; a transient entity might be a part. Depending on the level of detail desired, a factory worker might be regarded as a transient entity in one model and a resident entity in another.

In describing a particular aspect of the dynamic behavior of a system, it is often useful to focus on the *cycles* of the resident entities. For example, we might describe the busy-idle cycles of machines or workers. Alternatively, we might focus on the *paths* along which transient entities flow as they pass through the system (e.g., parts moving through a factory). Transient entity system descriptions tend to be more detailed (and more informative) than resident entity descriptions. Each type of modeling has advantages: modeling resident entity cycles tends to be easy and efficient, while

---

* Dune, by Frank Herbert, Chilton Book Company: Radnor, PA. 1965.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

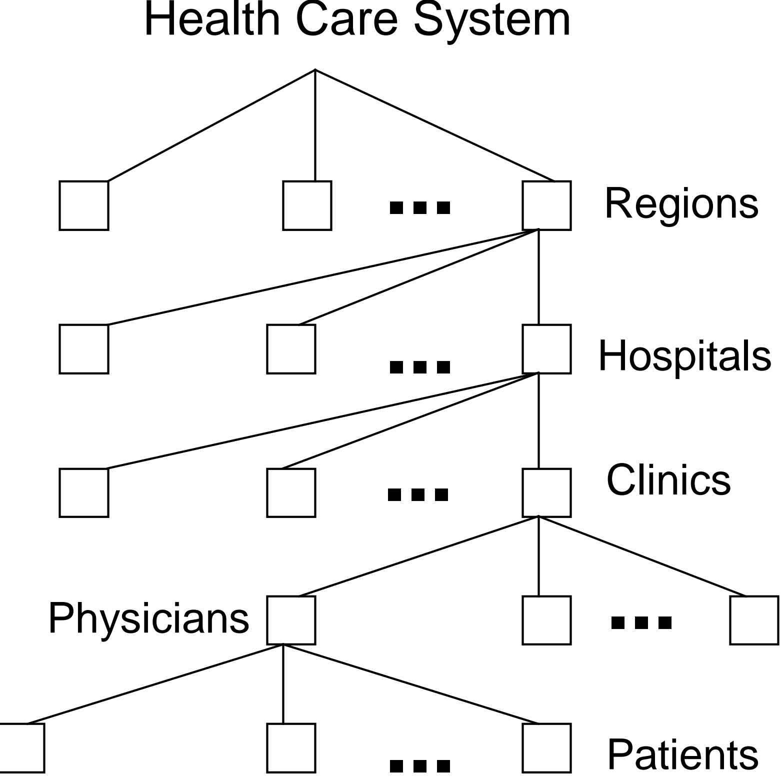
modeling the details of transient entity flow gives more information. Typically, a mixture of both viewpoints is used, but one or the other predominates.

In systems where there are relatively few resident entities and a great many transient entities, it is usually most efficient to study the cycles of the resident entities. Examples include semiconductor factories with thousands of wafers, communication systems with millions of messages, and transportation systems with tens of thousands of vehicles. In simulating such systems, the cycles of resident entities might be described by the values of only a few variables, while the flow of transient entities might require a great many variables to describe. On the other hand, systems where there are only a few transient entities and many resident entities (a power line inspection system or an airline maintenance facility) may be efficiently studied by examining the flow of transient entities.

It will often be the case that an initial model consisting only of resident entities is appropriate. Thus, first-cut system experiments can be done with a simple and efficient model. After a design has been roughed out and more details are desired, the model can be enriched into a transient entity flow model. Enriching the model to explicitly include transient entities generally makes the model larger, more complex, more prone to errors, and slower to execute. It may be wiser to use detailed transient entity models only for final refinements of a design. A well-designed simulation may have some sections that model only resident entity cycles and other sections that model the detailed flow of transient entities.

Entities are described by their characteristics (referred to as *attributes*). Attributes can be quantitative or qualitative. Moreover, they can be static and never change (the speed of a machine), or they can be dynamic and change over time (the length of a waiting line). Dynamic attributes can further be classified as deterministic or stochastic depending on whether or not the changes in their values can be predicted with certainty. It is sometimes useful to think of entities as belonging in sets owned by other entities. For example, in a factory a set of parts might be waiting in a line (queue) for a particular machine. Also, a set of machines (routing) might be required to process each part. Thus, each machine owns a set of parts, and each part owns a set of machines.

Figure 2.1: The Entity/Attribute Hierarchy for a Health Care System



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The level of detail with which we choose to describe a system determines whether a particular system component is thought of as an entity or an attribute of some entity. For example, we can examine regional health care systems at various levels. (See Figure 2.1.) We might think of the number of hospitals in the region as an attribute of the region. For greater detail, we might want to consider each hospital as a separate entity with the clinics offered as attributes. At an even finer level of detail, we might think of the clinics in the hospitals as distinct entities, with the physicians and patients as attributes of each clinic. Still greater detail is obtained by thinking of individual physicians and patients as entities described by their attributes of specialty, schedule, affliction, etc. The level of detail desired depends on our reasons for studying the system. It would not make sense to model a nationwide health care system at the level of individual patients; likewise, it would not be meaningful to model the laundry at a particular hospital from a national viewpoint.

The rules that govern the interaction of entities in a system that are not under our control are called *laws*. Similar laws are grouped in families, members of which are distinguished by *parameters*. Rules that are under our control are called *policies*; a family of similar policies may be distinguished by the values of their *factors*. When we experiment to determine the effects of changes in parameters, we are doing sensitivity analysis. When we experiment with changes in factors, we are doing optimization or design. Sensitivity analysis might examine changes in the rate of patient demand for an emergency room. Optimization in the same setting might focus on adjusting nurses' schedules to provide adequate coverage during peak periods. Unlike the real world, in simulation studies both types of experiments are conducted in much the same manner.

Often throughout this book we will use examples of simple, random systems consisting of a waiting line with one or more servers. (Notable exceptions are the financial risk analysis model, the critical path model, and the continuous time simulation of an aquatic ecosystem found in Chapter 5.) These stochastic, dynamic queues are found in many of the systems we are interested in studying: parts waiting for a machine, patients waiting for a doctor, jobs waiting for a computer, messages waiting for transmission, etc. The resident entities in the system are the machines, doctors, computers, data busses, . . . generically referred to as servers. The transient entities are the parts, patients, jobs, messages, . . . generically referred to as customers. Although not as obvious, the waiting lines, buffers, and queues are also resident entities in a queueing system. Laws describing the system might include the probability distribution of the time between successive customer arrivals. Policies might include the number of servers, the amount of waiting space, and the priority level different types of customers receive for service.

The *state* of a system is a complete description of the system and includes values of all attributes of entities, parameters of laws, factors for its policies, time, and what might be known about the future. The *state space* is the set of all possible system states. A *process* is an indexed sequence of system states; typically the index is time, but it might be a count of customers or some other system characteristic.

2.1.2 Models

We will define a *model* simply as

a system used as a surrogate for another system.

In typical computer simulation models, a system with mathematical entities is used as a surrogate for a system with physical entities. In this book when we use the word *system*, without qualification, we are referring to a real or hypothetical system that is the subject of the simulation study.

Mathematical models are conceptual abstractions of a particular aspect of a system. The mathematics we will be using include probability, statistics, and graphs. When we use the word *model,* without qualification, we will be referring to a graphical description of a system called an event graph. Finally, *simulations* will refer to computer programs developed from event graph models. Simulations will be our methodology for studying the model. A model serves as the interface between a system and a methodology for studying the system.

When evaluating a simulation, it is important to differentiate modeling a system from coding a model. Whether or not a simulation is "good" is more or less objective. A good simulation is a completely faithful rendition of a good model; nothing in the model is lost in the code. The process of testing if the simulation is good is called simulation verification, which is discussed in more detail later. This is often much more complicated than verification of other types of computer programs. However, there is no conceptual difficulty in defining a good program as being error free. Of course, the most one can honestly certify about any computer program is that it currently has no *known* errors.

Defining what constitutes a "good" model is much more subjective. A good model is based on good assumptions. Good assumptions make the simulation more efficient or the system easier to understand while costing little in terms of validity. Driving a good bargain between model simplicity and model validity is the essence of the art of modeling.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

A purist view of validity is that a model will be valid as long as it is based on explicit assumptions and the implications of the assumptions are well understood. From this academic viewpoint, a modeling error is not correctly stating and applying all model assumptions. If we take the pragmatist's view of a good model as contributing to correct decisions, it is possible to test the effects of including a particular detail or making a certain assumption by comparing the behavior of the simulation with and without the detail or assumption. Simulation is one of the few methodologies that allows testing the robustness of models with different assumptions. Perhaps the biggest danger in simulation modeling is including too much detail in the model. An experienced consultant in the field once remarked that he could tell a novice at simulation by the excessive amount of detail in his or her models.

A technique for keeping model details at a reasonable level is to focus on the *similarities* among the entities in the system rather than the *differences*. If transient entities (customers, jobs, messages, etc.) can be treated as identical, you can develop a valid model that merely keeps track of the numbers of transient entities at various stages of their progress through a system. This makes it unnecessary to have a detailed record for each individual entity. Such a model would require updating relatively few integers (the counts) instead of creating and maintaining separate records for every transient entity in the system. Similarly, treating resident entities (servers, machines, buffers, etc.) as identical allows you to maintain counts of the numbers of resident entities in the various states of their process cycles rather than keeping a record of the status of each entity. In situations where there are a great many transient entities in the system at one time, it might be necessary to treat transient entities as identical. It is certainly more efficient to have a single integer variable that counts transient entities than it is to create and maintain thousands of complete records.

It is natural to notice differences between entities in a system. However, a valuable modeling skill is the ability to recognize similarities. Differences are modeled only when they are essential to the validity of the study results. It is also natural to include detail unless there are solid reasons for assuming that it can be omitted. When building simulation models of complicated systems, it is good practice to require justification for including detail. Even when the differences in entities are thought to be important, a skilled modeler will be able to define groups of entities that can be treated as identical.

Sometimes the activity of developing a simulation model has as much value as the model itself. Building a model forces us to identify our objectives, determine constraints, quantify our knowledge, and expose our misconceptions. It could be argued that a study has merit even if its recommendations are never implemented and that a simulation model has value even if it never runs. Of course, this is of little comfort to the student who fails a homework assignment or an engineer who must try to find another job. It is vastly more satisfying to professors and employers if the simulation model runs and the recommendations from the study are adopted. The motivation for the development of event graphs was to make simulation models easier to build and verify. With event graphs, it is much easier to verify that your simulation program reflects the way you have modeled the system than it is to validate that the model actually can be used to imitate the relevant behavior of the real system.

2.1.3 Model Verification

An absolutely *valid* simulation model with all the detail and behavior of real life is probably not attainable, or even desirable. However, every simulation model should do what its creator intended. Ensuring that the computer code for the simulation model does what you think it is doing is referred to as the process of model *verification*.

There is a trade-off involved between validation and verification of a simulation model. Adding detail to a model makes the code more complicated. If correctly implemented, this detail will perhaps improve model validity. However, adding complex details can make code verification more difficult if not impossible. Identifying only the substantive details that are important to include in a simulation model is important and often subject to negotiation.

Really gross errors in a simulation code can be detected using standard statistical testing. For example, a classical paired *t*-test between the means of samples from the real world and those from simulation runs might be conducted. (A description of a *t*-test can be found in any introductory statistics textbook.) However, there may not be enough real-world data to reject a hypothesis that the model and real world data have nearly the same parameters. Moreover, the data itself may not be valid. (See the "Five Dastardly D's of Data" in Chapter 9.)

Translating a model from computer code into clear language is a good exercise. It is interesting to contrast what two different people think a model is doing. You can also compare what you think your model is doing with what *it* thinks it is doing using the English translations generated by SIGMA.

An excellent tool for helping to verify a simulation model is an informal exercise called a "Turing test," named after a man who conjectured on the possibility of not being able to distinguish computing machines from real people. For a Turing test, actual blank forms used in the day-to-day management of a system are filled in with either simulated or real data. Only the blanks that are *relevant* for the purposes of the study are different. Managers and other people familiar with the system are then asked to identify the real and simulated documents and *tell how each form was identified.* It is vital that everyone know in advance that this exercise is likely to be repeated several times.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

People familiar with these forms are usually more comfortable doing this exercise than they are in reviewing a computer code, evaluating a statistical analysis, or even observing an animation. Just the process of determining what data on each form are relevant to the study can make this exercise worthwhile.

A typical experience with a Turing test has been that the manager can immediately identify most, if not all, of the bogus forms! This is actually a good outcome; it indicates that the manager is paying attention, and it can lay the foundation for effective communication. A non-technical manager "winning" the first round may also help diffuse any antagonism that they may have developed toward the simulation project. What happens next is critical: when the manager tells how the simulated data was identified, changes to the simulation model should be made and the exercise repeated. It is the *repetition* of the exercise that is important, not the outcome of each iteration.

Statistical analysis to assess whether or not the outcome of the exercise is likely to result from guessing is presented in Schruben (1980). However, such formal analysis can actually be detrimental. It could easily inhibit communication and alienate the manager by moving the discussion to the unfamiliar ground of mathematical statistics.

2.2 Discrete Event Systems and Simulations

As stated earlier, systems in which changes occur at particular instants of time are called discrete event systems. In a simulation of a discrete event system, time is advanced in discrete (variable and often random length) steps to the next interesting state change; uninteresting time intervals are skipped over. This coarse level of detail permits the modeling of very large systems such as airports and factories.

A description of the *state* of a discrete event system will include values for all of its numerical attributes as well as any schedule it might have for the future. Changes in the state are called *events*. In a production system, events might include the completion of a machining operation (the state of a machine would change from "busy" to "idle"), the failure of a machine (the machine state would change to "broken"), the arrival of a repair crew (the machine state would change to "under repair"), the arrival of a part at a machining center (the machine might again become "busy"), etc.

The ability to identify the events in a discrete event system is an important skill, one that takes practice to acquire. Initially, you might use the following simple steps as a guide to identify system events:

1. State the purpose of your system. Be aware that there might be several (conflicting) purposes.
2. State the objectives of your study.
3. Design, at least qualitatively, the experiments you might want to run with your simulation.
4. Identify the resident and transient entities in your system and their important attributes; assign names to the attributes.
5. Identify the dynamic attributes and the circumstances that cause their values to change . . . these will be the events.

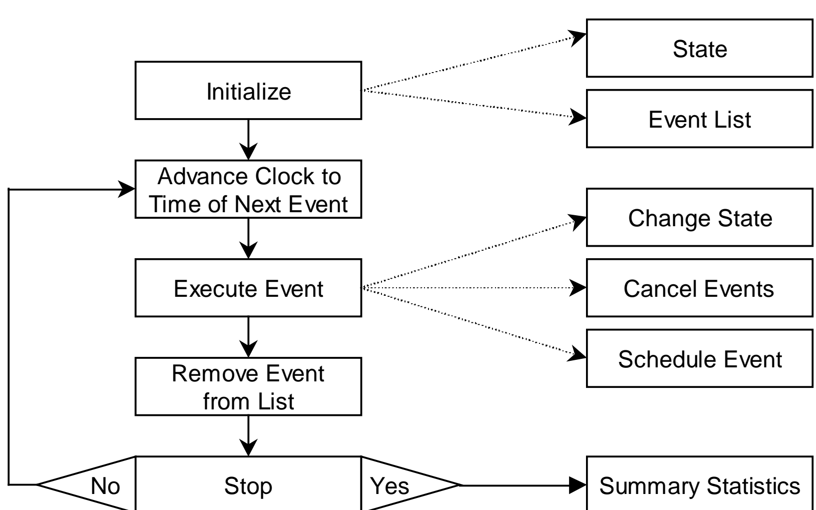
The building blocks of a discrete event simulation program are *event procedures***.** Each event procedure makes appropriate changes in the state of the system and, perhaps, may trigger a sequence of other events to be scheduled in the future. Event procedures might also cancel previously scheduled events. An example of event cancelling might occur when a busy computer breaks down. End-of-job events that might have been scheduled to occur in the future must now be cancelled (these jobs will not end in the normal manner as originally expected).

The event procedures describing a discrete event system are executed by a main control program that operates on a master appointment list of scheduled events. This list is called the *future events list* and contains all of the events that are scheduled to occur in the future. The main control program will advance the simulated time to the next scheduled event. The corresponding event procedure is executed, typically changing the system state and perhaps scheduling or cancelling further events. Once this event procedure has finished executing, the event is removed from the future events list. Then the control program will again advance time to the next scheduled event and execute the corresponding event procedure. The simulation operates in this way, successively calling and executing the next scheduled event procedure until some condition for stopping the simulation run is met.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The operation of the main simulation event scheduling and execution loop is illustrated in Figure 2.2.

Figure 2.2: Main Event-Scheduling Algorithm



Extended Example: We will follow the changes in a typical future events list by examining a simulation of a machine center with three identical machines (numbered `0`, `1`, and `2`) and two workers (worker `0` and worker `1`).

The types of events that might occur in this example are the `ARRIVAL` of the next part at the machining center, a machine "`STARTing`" or "`FINISHing`" work on a part, a `BREAKDOWN` of a machine, and a broken machine being `REPAIRED`. An actual simulation model would, of course, have other types of events.

For each event that pertains to a specific machine and/or operator, the machine number followed by the operator number (if appropriate) are listed as event attributes. At a particular time during a simulation run, the future events list might look like the one pictured in Table 2.1. The future events are logically sorted according to times that events are scheduled to occur. Here time will be measured in minutes.

The current time in this example is `3.00`, and the `ARRIVAL` of a part has just occurred at the center. This `ARRIVAL` event has "scheduled" the next `ARRIVAL` event to occur at time `3.37`. We can determine the status of each machine by scanning down the future events list and checking what lies in the future for each machine. (Recall that the machine number is designated by the first event attribute.) Machine `0` is due to `FINISH` processing the part it is currently working on at time `3.20`, so machine `0` must be busy. Likewise, machine `1` is busy and due to `FINISH` working on a part at time `3.40`. Finally, machine `2` will be `REPAIRED` at time `3.43`, so it is currently being fixed by the repair crew. We can see from the future events list in Table 2.1 that when the part arrived at time `3.00` none of the machines were available to start working on it. Thus, the part will join other parts in a queue waiting to be processed.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Table 2.1: Future Events List for a System with Three Machines
(Time = 3.00)

	Time	Event Type	Event Attributes
(Current time)	3.00	ARRIVAL	
	3.20	FINISH	0,1
	3.35	BREAKDOWN	1
	3.37	ARRIVAL	
	3.40	FINISH	1,0
	3.43	REPAIRED	2
	9.01	BREAKDOWN	0

Note that machine 0 is due to experience its next BREAKDOWN at time 9.01 and machine 1 is due for a BREAKDOWN at time 3.35 - before it can FINISH its current operation. Therefore, when machine 1 breaks down at time 3.35, the FINISH event for this machine at time 3.40 will have to be cancelled.

To see how this machining center simulation might proceed, we will now advance the current time to 3.20 and execute the FINISH event on machine 0. Looking at the second attribute of this FINISH event, we see that operator 1 becomes idle. Since we know that there is at least one part waiting, we can immediately START processing the next part. A new START event for machine 0 has been scheduled to occur at the current time of 3.20 with operator 1. The future events list is now like Table 2.2.

Table 2.2: Future Events List for a System with Three Machines
(Time = 3.20).

	Time	Event Type	Event Attributes
(Current time)	3.20	FINISH	0,1
	3.20	START	0,1
	3.35	BREAKDOWN	1
	3.37	ARRIVAL	
	3.40	FINISH	1,0
	3.43	REPAIRED	2
	9.01	BREAKDOWN	0

We next execute the START event for machine 0 at time 3.20 with operator 1. Suppose that a stored (or randomly generated) machine processing time for machine 0 is 1.20 minutes, then the FINISH event for this machine will be scheduled to occur 1.20 minutes from the current time of 3.20 or at time 4.40. The future events list after executing the START event at time 3.20 is shown in Table 2.3.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Table 2.3: Future Events List for a System with Three Machines
(Time = 3.20)

	Time	Event Type	Event Attributes
(Current time)	3.20	START	0,1
	3.35	BREAKDOWN	1
	3.37	ARRIVAL	
	3.40	FINISH	1,0
	3.43	REPAIRED	2
	4.40	FINISH	0,1
	9.01	BREAKDOWN	0

Next, we advance the time to `3.35` and execute the `BREAKDOWN` event for machine `1`. This `BREAKDOWN` event will cause the `FINISH` event for machine `1` scheduled at time `3.40` to be cancelled. We will assume here that the part is destroyed when the machine breaks down and that the worker becomes available for other work. If it takes five minutes for a repair crew to repair machine `1`, the future events list after the `BREAKDOWN` occurs is like that in Table 2.4. (Note the `REPAIRED` event for machine `1` has been scheduled at `8.35`, five minutes beyond the current time of `3.35`.) The simulation will now advance to time `3.37` when the next `ARRIVAL` event will occur.

Table 2.4: Future Events List for a System with Three Machines
(Time = 3.35)

	Time	Event Type	Event Attributes
(Current time)	3.35	BREAKDOWN	1
	3.37	ARRIVAL	
	3.43	REPAIRED	2
	4.40	FINISH	0,1
	8.35	REPAIRED	1
	9.01	BREAKDOWN	0

Do not worry if this all seems a bit mysterious for now. Discrete event simulation modeling is more than a simple exercise in computer programming. It is initially somewhat confusing for everyone. You will soon discover that it is relatively straightforward once you grasp the concept of an event and understand the relationships between events. For this, we will use event graphs.

2.3 Event Graphs

The three elements of a discrete event system model are the state variables, the events that change the values of these state variables, and the relationships between the events (one event causing another to occur). An event graph organizes sets of these three objects into a simulation model. In the graph, events are represented as vertices (nodes) and the relationships between events are represented as edges (arrows) connecting pairs of event vertices. Time sometimes elapses between the occurrence of events.

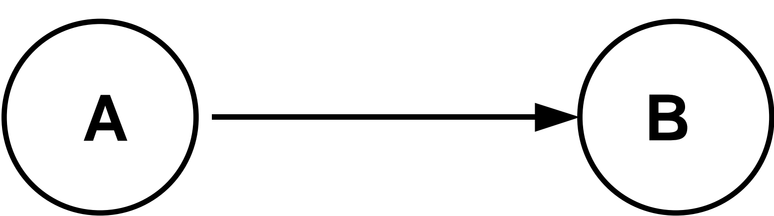
The basic unit of an event graph is an edge connecting two vertices. Suppose the edge represented in Figure 2.3 is part of an event graph. We interpret the edge between A and B as follows:

whenever event `A` *occurs, it might cause event* `B` *to occur.*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Basically, edges represent the conditions under which one event will cause another event to occur, perhaps after a time delay.

Figure 2.3: Simple Event Graph Edge



Using this notation, we can build a model that simulates a simple waiting line with one server (e.g., a ticket booth at a theater, the drive-in window at a fast-food restaurant, etc.). For our example, we will model an automatic carwash with one washing bay. The event graph of our carwash is represented in Figure 2.4.

We will begin our examination of this graph by discussing each vertex. The `RUN` vertex models the initialization of the simulation, the `ENTER` vertex models a car entering the carwash line, the `START` vertex models the start of service, and the `LEAVE` vertex models the end of service.

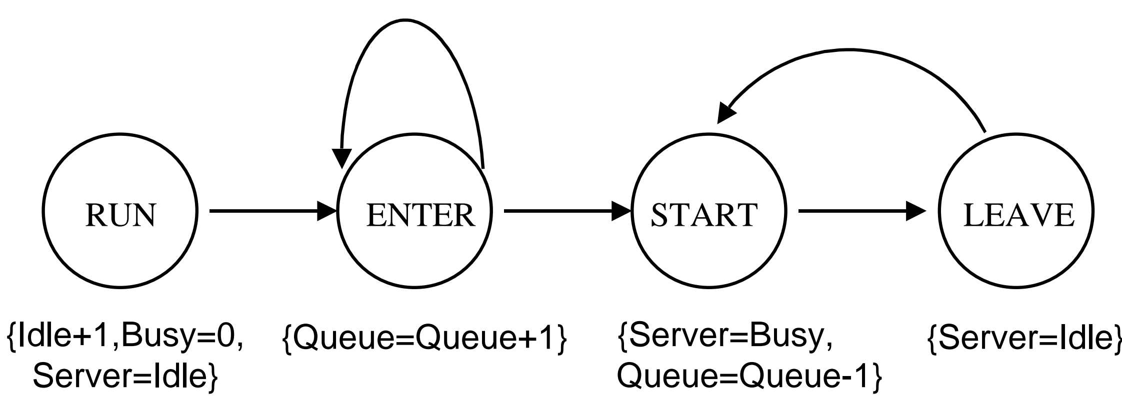
The state variables chosen to describe this system are:

`SERVER` = the status of the washing bay (busy, idle), initially set idle.

`QUEUE` = the number of cars waiting in line, initially set equal to zero.

To make our model more readable, we also define the constants, `IDLE=1` and `BUSY=0`.

Figure 2.4: Simple Event Graph of a Carwash



Next, we will focus on the changes in the state variables, shown in braces. The simulation `RUN` is started by making the washing bay at the carwash available for use {`IDLE=1,BUSY=0,SERVER=IDLE`}. Each time a car `ENTER`s the line, the length of the waiting line is incremented {`QUEUE=QUEUE+1`}. When service `START`s, the washing bay is made busy {`SERVER=BUSY`} and the length of the line is decremented {`QUEUE=QUEUE-1`}. Whenever a car has been washed and `LEAVE`s the washing bay, the washing bay is again made available {`SERVER=IDLE`} to wash other cars.

The dynamics of an event graph model are expressed in the edges of the graph. We read an event graph simply by describing the edges *exiting* each vertex (out-edges). In-edges take care of themselves. Continuing with our example, we look at each edge in Figure 2.4.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The simulation RUN is started by having the first car ENTER the carwash (edge from RUN to ENTER). If the ENTERing car finds the washing bay idle, service will START immediately (edge from ENTER to START). Each time a car ENTERs the carwash, the next car will be scheduled to ENTER sometime in the future (edge from ENTER to ENTER). The START service event will always schedule a car to LEAVE after that car has been washed (edge from START to LEAVE). Finally, if there are cars waiting in line when a car LEAVEs, the washing bay will START servicing the next car right away (edge from LEAVE to START).

The self-scheduling edge (the loop) on the ENTER event is the conventional way of perpetuating successive customer arrivals to the system. There will typically be some random time delay between customer arrivals.

After looking at the carwash model, you may have guessed that the state changes for an event vertex are typically very simple. Most of the action occurs on the edges of the graph. The conditions and delays associated with the edges of the event graph are very important; it is on the graph edges that the logical flow and dynamic behavior of the model are defined. For each edge in the graph we will need to define under what conditions and after how long one event might schedule another event to occur.

We will associate with each edge a set of conditions that must be true in order for an event to be scheduled. Also associated with each edge will be a delay time equal to the interval until the scheduled event occurs. Time will be measured in minutes for our examples. We have enriched the basic event graph to include edge conditions and edge delay times (see Figure 2.5). This edge is interpreted as follows:

if condition (i) *is true at the instant event* A *occurs, then event* B *will be scheduled to occur* t *minutes later.*

Figure 2.5: Conditional Event Graph Edge with a Time Delay

A —t— (i) —> B

If the condition is not true, nothing will happen, and the edge can be ignored until the next time event A occurs. You can think of an edge as nonexistent unless its edge condition is true. If the condition for an edge is always true (denoted as 1==1), the condition is left off the graph. We will call edges with conditions that are always true *unconditional* edges. Zero time delays for edges are not shown on the graph.

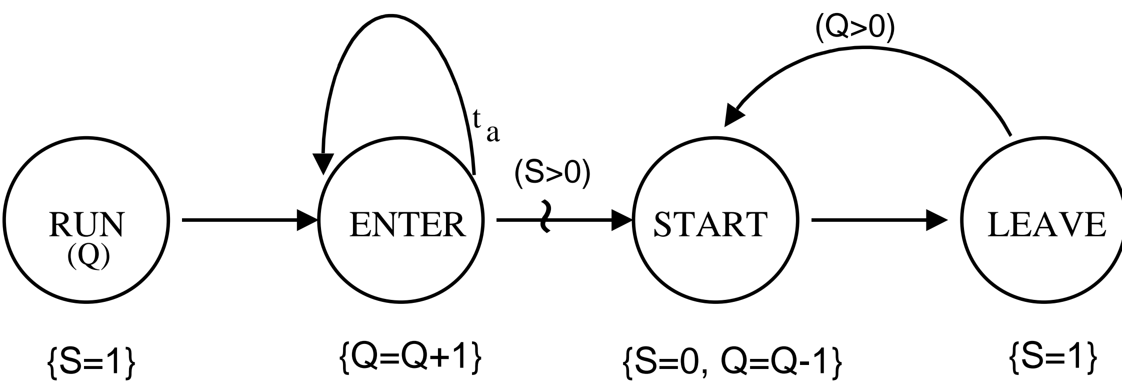
While you are learning to read event graphs, it might be a good idea to use the edge interpretation in the previous paragraph as a template for describing each edge. Once the edges in the graph are correct, the state changes associated with each vertex are typically easy to check.

Our carwash model with edge conditions and delay times is shown in Figure 2.6. The state variables SERVER and QUEUE are now denoted by S and Q, respectively, and the status of S is indicated by 1 or 0 (IDLE=1, BUSY=0). In addition, the time between successive car arrivals (probably random) is denoted by t_a and the service time required to wash a car is denoted by t_s. When values of t_a are actually needed, they might be obtained from a data file or generated by algorithms like those in Chapter 9.

In Figure 2.6, state changes associated with each vertex are enclosed in braces and edge conditions in parentheses. As you read the following description, identify a single edge with each sentence.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 2.6: Carwash Model with Edge Conditions and Delay Times



At the start of the simulation RUN, the first car will ENTER the system. Successive cars ENTER the system every t_a minutes. If ENTERing cars find that the server is available (S>0), they can START service. Once cars START service, t_s minutes later they can LEAVE. Whenever a car LEAVEs, if the queue is not empty (Q>0), the server will START with the next car.

Now re-read the above paragraph without looking at Figure 2.6. You will see that it is a concise description of the behavior of a queueing system. With practice, a system description can be read easily from the edges of an event graph. This is an excellent way to communicate the essential features of a simulation model and a good first step in model validation. With experience in reading event graphs, it becomes easier to detect modeling errors. This graph represents a completely defined simulation model. To run this model, only the starting and ending conditions for the run need to be specified.

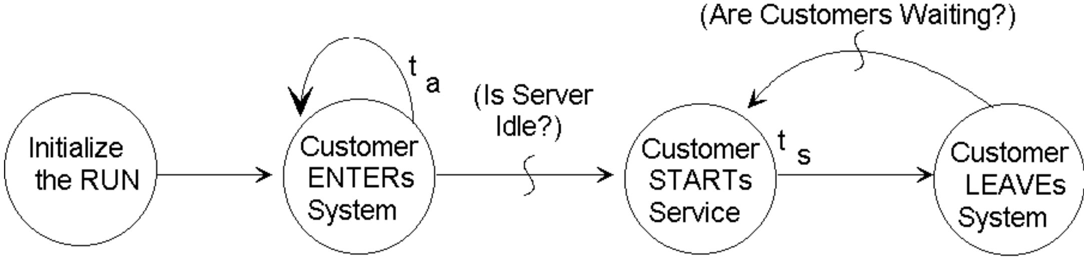
2.4 Verbal Event Graphs

Before designing your own event graph model, it is a vital that you develop a verbal description of your system. This description would include state changes associated with each vertex along with a verbal description of each edge condition and delay time on the graph. A verbal event graph for a generic single server queueing system is shown in Figure 2.7.

Developing a verbal description of your system is a necessary first step toward building a realistic and accurate simulation model. It will help you conceptualize the major components in the system, determine the key events and their interrelationships, and identify the state variables, edge conditions, and time delays necessary for the model. Note that state variables will need to be defined that permit testing of all edge conditions in your verbal event graph. Once you have constructed a detailed verbal description, the event graph model will be much easier to build.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 2.7: Verbal Event Graph of a Single Server Queue



t_a = Time Between Customer Arrivals (Possibly Random)

t_s = Time to Service a Single Customer (Possibly Random)

2.5 Visual Power of Event Graphs

The visual modeling power of event graphs is most appreciated after one recognizes the complicated details involved in a discrete event simulation. The fundamental concept in event graph modeling is to use a directed graph as a picture of the relationships among the *elements* in sets of expressions characterizing the dynamics of the system. Each vertex of the graph is identified with a set of expressions for the state changes that result when the corresponding event occurs. Each edge in the graph identifies sets of logical and temporal relationships between a pair of events.

2.6 SIGMA

SIGMA is a general event graph modeling environment that facilitates the development of correct simulation code. Reading an event graph facilitates model validation, and, since with SIGMA the model is the language, code verification is much easier using SIGMA than using traditional discrete event simulation coding methods.

Using modern simulation languages, it is quite possible to create complicated models of systems without the user having any idea how the simulation program actually works. Indeed a prime objective of some special-purpose simulators is to isolate the user completely from the simulation code. This is an advantage as long as the application does not extend beyond the domain of the simulator. However, once a certain level of skill in simulation modeling has been reached, many feel constrained by the inflexibility of high-level simulators.

The objective of event graph modeling using SIGMA is to provide a user-friendly environment in which to build, verify, and experiment with discrete event simulation models. Unlike most simulation software, SIGMA is designed to remove *all* of the mystery in discrete event simulation. Depending on your version, you may have access to the full source code of all SIGMA-generated simulation models. However, you can still choose to remain oblivious to the workings of the simulation program. With SIGMA the choice is yours. Initially event graph modeling may seem a bit abstract, but you will soon find it to be a simple yet powerful approach. SIGMA is suitable for both the beginner and the seasoned professional. While SIGMA is easy to learn and easy to use, it is powerful enough to allow for real growth as modeling skills evolve. Without learning any additional modeling tools, you will be able to model any discrete event system using event graphs.

Programming in the two dimensions of a graph has several advantages over traditional (line at a time) linear coding. Loops, conditional branching, function calls, and even notorious goto's are all easily represented as edges on a graph. In fact once you become familiar with SIGMA, you might consider using it as a general programming tool to create code that has nothing to do with simulation modeling. You might find that graphically programming in a plane makes more sense than writing code in the traditional linear method.

SIGMA makes simulation modeling much easier than directly writing code. With SIGMA, you interact with a simple graph like that in Figure 2.4. Details on edges (arrows) and event vertices (nodes) are available at the click of a mouse.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

SIGMA also has instant input checking. Unlike simple syntax checkers, SIGMA checks your computation by actually executing expressions when they are entered. This helps to ensure that while running your model you will not encounter frustrating spelling, syntax, or computation errors.

Computer programs used for discrete event simulation are distinguished from other types of computer programs by two fundamental characteristics. Discrete event simulation programs will have both logic for representing the passage of time (they are dynamic as opposed to static) and logic for representing randomness (they are stochastic as opposed to deterministic). Reflecting these two fundamental features of a discrete event simulation program, SIGMA does two computations automatically; one computation models time while the other models randomness. In SIGMA, there are only two variables you should not change: `CLK`, which represents the current value of simulated time, and `RND`, which represents a randomly chosen number between zero and one.

A program that has randomness but is static might be used to generate artificial data samples to study the behavior of a statistical procedure; such a program is commonly referred to as a Monte Carlo simulation. Systems that are modeled as continuously changing, such as chemical reactions, electrical pulses, and mechanical linkages, fall into the realm of continuous simulations, which model the progression of change using differential or difference equations. Continuous time simulations are typically dynamic but deterministic. An example of a stochastic continuous time simulation is given in Chapter 5.

2.7 Exercises

2.7.1 Model Evaluation

Read a recent article where discrete event simulation models are used to solve real-world problems. Some of the publications you might look at are *Management Science, Interfaces, Industrial Engineering, Operations Research, Material Handling, Medical Care, Computer Performance Evaluation, Expert Systems,* and *Simulation.* Conference proceedings in various fields of engineering also have articles about simulation; in particular, the *Winter Simulation Conference Proceedings* is a good source.

Collect or create the following:

(a) Photocopies of the title page and abstract of the article (explicitly give the source).

(b) A brief statement of the objective of the simulation project (in your own words).

(c) What programming language was used? Did they justify their choice?

(d) A paragraph describing what real-world data was needed for the model and how it was collected.

(e) A paragraph describing what experiments were run with the model. What factors were varied? What system performance measurements were taken? What analysis was done with these measurements?

(f) Give a one-page critique of the article and project. What would you have done differently?

2.7.2 Short Answers

(a) Give the most important advantage as well as the greatest disadvantage to having a simulation model with very few events as opposed to an equivalent model with many events.

(b) What two features generally distinguish a discrete event simulation program from other types of programs?

(c) What are the two broad classifications of entities in a discrete event system?

(d) Give one advantage and one disadvantage to modeling the flow of transient entities in a system.

(e) Does a model with few entities each having many attributes have more or less detail than a model with many entities each having fewer attributes? Which will generally take more memory to run? Why?

2.7.3 Identifying Discrete Event Systems

For a real system of your choice, describe the system in less than a page and give a one-sentence objective for studying this system with a simulation model. Identify the dynamic attributes of entities in the system and the events where these attributes change values. Give a rough event graph for the system. (Pick something interesting to you (but simple).

2.7.4 Simultaneous Events

In the carwash model, if `Q=5` and there is a time tie between an `ENTER` event and a `LEAVE` event, which event should be executed next? Why?

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

2.7.5 Events List

In the event graph of Figure 2.6, what is the maximum number of events that might be scheduled on the events list at one time?

2.7.6 Edge Interpretation

For Figure 2.5, which of the following statements are true?

Whenever event `A` occurs: if `t` time units later, condition `(i)` is true, then event `B` will be scheduled to occur immediately.

Whenever event `A` occurs: if condition `(i)` is true, then event `B` will be scheduled to occur `t` time units later.

Whenever event `A` occurs and condition `(i)` is false, event `B` will be scheduled to occur `t` time units after condition `(i)` becomes true.

2.7.7 Future Events List

Assume for the example in Table 2.4 that the times between the next three part arrivals are 2, 1.5, and 3.2 (each `ARRIVAL` event schedules the next `ARRIVAL` event in this model). Furthermore, assume that the processing time for each machine is 0.5 minutes and that there are currently 6 parts waiting in the queue for processing. Finally, assume that once machine 2 is repaired it will be 10 minutes before it breaks again. What does the future events list look like at time 8.00? What is the status of each machine? How many parts are waiting in the parts queue?

2.7.8 Simultaneous Event Errors

Assume that for the model corresponding to the future events list in Table 2.4 that the `FINISH` event made a machine idle and scheduled a `START` event if there were parts waiting in the queue. Also assume that an `ARRIVAL` event will schedule a `START` event if it finds an idle machine. What would happen if a part arrived at time 4.40, just as machine 0 finishes processing? What would happen if the `FINISH` event occurred before the `ARRIVAL` events?

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

3

A Tutorial on the Basics of SIGMA

An overview of the SIGMA software environment is presented in this chapter in tutorial format. All primary elements of event relationship graph modeling are explained. Technical details are presented in later sections.

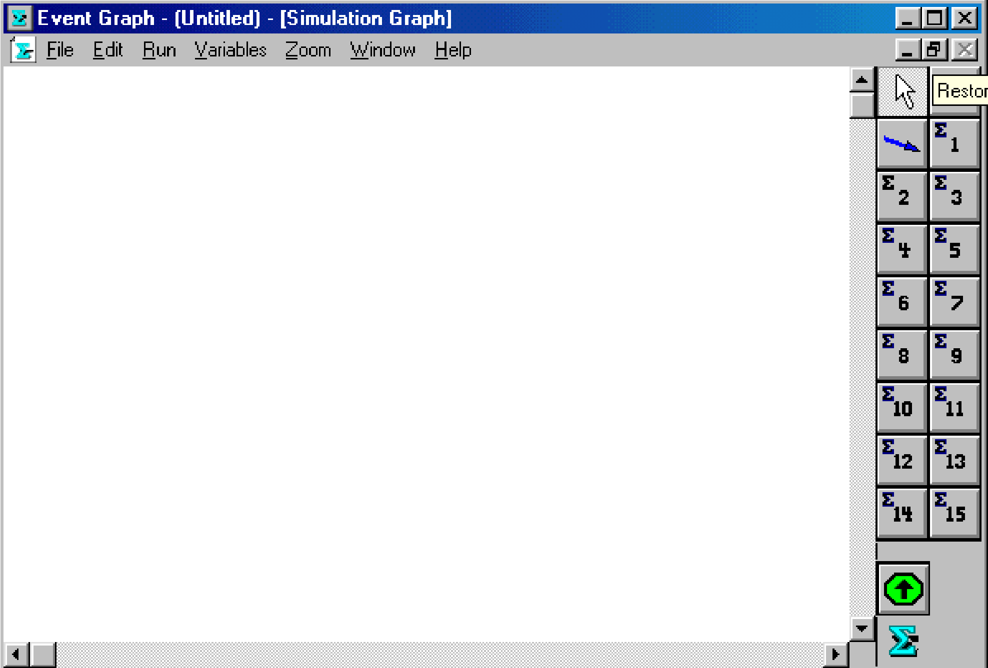
3.1 Starting a SIGMA Session

Double click on SIGMA.EXE to begin a SIGMA modeling session. SIGMA does not automatically load a model, but they can easily be recalled using the File/Recall menu command.

3.2 The SIGMA Modeling Environment

A *simulation graph* window similar to the one in Figure 3.1 will appear when a SIGMA session begins. This is the primary window for a SIGMA modeling session. Linked to this modeling session are *simulation plot* windows and *output* windows. Note: Several SIGMA modeling sessions may run simultaneously; thus, several simulation graph windows representing different models may be open at one time.

Figure 3.1: SIGMA's Simulation Graph Window

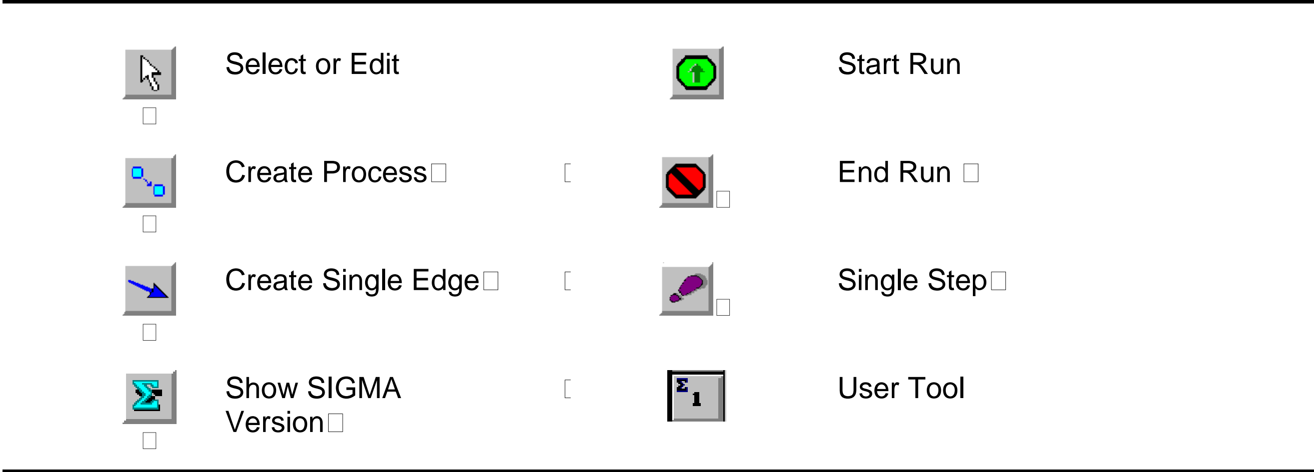


There are three regions in the *simulation graph* window: the model creation area (the region in the center of the window), the menu bar (located along the top of the window), and the toolbar (located along the right side of the window).

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The various push buttons in the toolbar are identified in Figure 3.2. The Start Run and End Run tools allow you to start or stop a simulation run; the Select or Edit, Create Process, and Create Single Edge tools let you quickly create a model; the Single Step tool allows you to watch the simulation as each event is executed; and the Run Time Information and Show SIGMA Version tools provide you with additional information. In addition, fifteen User Tool buttons allow you to add previously created models to a new modeling session.

Figure 3.2: SIGMA Toolbar



A window is activated when the mouse pointer is clicked anywhere on it. As the various windows are activated, the menu bar changes. The menu bar for the *simulation graph* window contains the following menus: File, Edit, Run, Variables, Zoom, Window, and Help. The *simulation plot* window contains the File, Edit, Options, Window, and Help menus; the *output* window contains the File, Edit, Search, Window, and Help menus.

When a SIGMA session is started, the mouse pointer is normally in Create Process mode (⊕). It is in this mode that the graphical components of a simulation model are created. If you click the right mouse button, the mouse pointer will change to Select or Edit mode (⇖). This mode is used to add or change information related to specific vertices or edges. Just double-click on a vertex or edge, and an Edit Edge or Edit Vertex dialog box will appear. Add information or make changes by clicking on the appropriate box and entering the data from the keyboard. Clicking the right mouse button will cause the mouse to alternate between the Create Process mode and Select or Edit mode. Pressing the appropriate push buttons on the tool bar will also activate the Create Process mode or the Select or Edit mode.

3.3 Exploring Our Carwash Model

We will use a previously created model, `CARWASH.MOD`, to examine the components of a SIGMA model. To begin, start a SIGMA session. Open `CARWASH.MOD` by pressing the File\Open command, pressing Event Graph, and then double-clicking on `CARWASH.MOD` from the list of previously created models in the dialog box.. This model is represented in Figure 3.3.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 3.3: An Event Graph for a Single Server Queue, `CARWASH.MOD`

Recall that `CARWASH.MOD` has only two state variables: `QUEUE`, the number of cars waiting in line, and `SERVER`, the status of the machine, `0=BUSY` and `1=IDLE`.. Four events are represented by the four vertices in the carwash model. The `RUN` vertex starts the simulation, the `ENTER` vertex models customers entering the carwash, the `START` vertex occurs when a car starts service in the washing bay, and the `LEAVE` vertex occurs when a car finishes service and leaves the washing bay. The time intervals between successive customer arrivals to the carwash are independent and random.

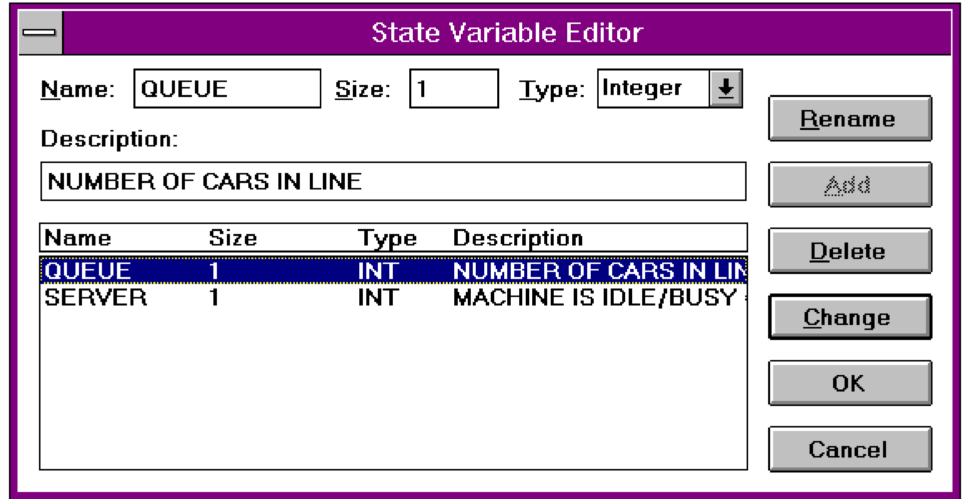
WARNING: *You should not use a floppy drive or a write-protected network drive as your default drive.*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

3.3.1 State Variables

Click the mouse on the Create/Edit Variables command under the Variables menu to see the list of state variables for this model in the State Variable Editor dialog box. Any state variables you define can be used anywhere in your SIGMA model. We refer to them as state variables to emphasize that they are accessible to all parts of the model. Clicking on one of the state variables in this dialog box will cause the details associated with that state variable to be displayed in the input boxes above (as in Figure 3.4.)

Figure 3.4: The State Variable Editor Dialog Box



The dialog box for state variables, like other SIGMA dialog boxes, has a line for a brief description of the object. These descriptions are important; they appear as comments in your SIGMA-generated simulation source code and make your model much easier to understand. Click on the Cancel command button to close the state variable dialog box and return to the simulation graph seen in Figure 3.3.

We will further explore this model by clicking the mouse on elements in the event graph. Make sure that the mouse pointer is in Select or Edit mode. Read the description in each dialog box as we examine the vertices and edges in the event graph..

3.3.2 Vertices

Double-click on the RUN vertex. This vertex, whose dialog box is shown in Figure 3.5, was the first vertex created in this model. SIGMA calls the first vertex created vertex number 1; it will always be executed first when the model is run. (This first vertex is colored green on the screen.)

The RUN vertex has a parameter: the state variable, QUEUE. When CARWASH.MOD is run, you will be asked to provide an initial value for the number of customers in the QUEUE when the system opens for service. All SIGMA state variables are initialized to be equal to zero. The only state change associated with the RUN vertex is to make the SERVER available (SERVER=1). To exit, click your mouse on the Cancel command button.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 3.5: The Dialog Box for the RUN Vertex

Edit Vertex 1

Name: RUN
Trace Event
Circle
Inactive Picture
Active Picture
Disable Animation
Transparent Bitmaps
Description: INITIALIZATION OF THE SIMULATION
State Change(s): SERVER=1
Parameter(s): QUEUE
Display Variables:
Display Location: Left
OK
Execute
Remove
Cancel

The ENTER vertex is where customers join the line. Open the dialog box for the ENTER vertex. This vertex simply increments the number of customers waiting in line (QUEUE=QUEUE+1). Note that QUEUE has been entered as a display variable, so its value will be shown while the simulation is running. Close this dialog box and then double-click on the START vertex.

The START service vertex is where the server is made busy (SERVER=0) and the number of customers waiting in line is decremented (QUEUE=QUEUE-1). Close the START vertex dialog box and open the LEAVE dialog box.

The LEAVE vertex makes the server available to serve other customer (SERVER=1). Note that if the variable, QUEUE, were defined as the number of customers in the total system (in service as well as in line), QUEUE would be decremented in the LEAVE vertex rather than in the START vertex. Close this dialog box.

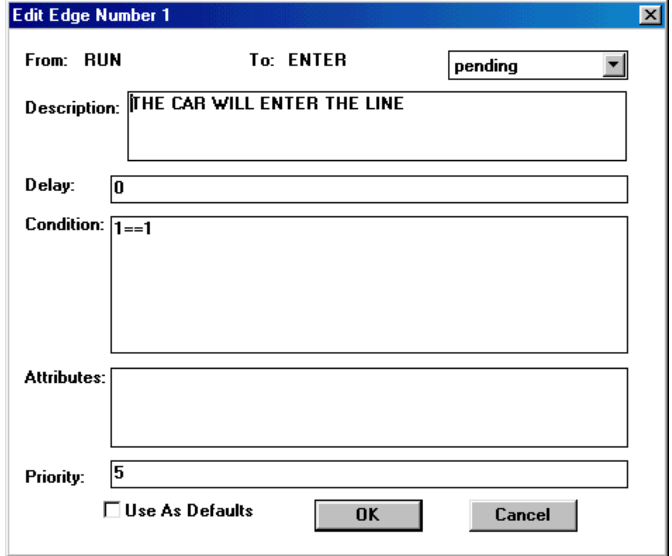
The event vertices are relatively straightforward in discrete event models. The complexity of the models comes at the edges of the event graph, where the dynamic and logical relationships between events are specified.

3.3.3 Edges

Next, we will examine each of the edges in our model. Figure 3.6 is the edge dialog box that you should see if you double-clicked on the edge between the RUN vertex and the ENTER vertex.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 3.6: Edge Dialog Box from RUN Vertex to ENTER Vertex Schedules the First Customer Arrival



We see that this edge has the condition (1==1). As in the C programming language, == is a test for equality in SIGMA. Thus, this edge unconditionally (1==1) schedules the first customer to ENTER the system. There is no time delay between starting the RUN and the first customer ENTERing the system. Here no attribute values are passed, and the execution priority, used to break time ties between events scheduled to occur at the same clock time, is set to a neutral value of 5**.** (Attributes and priorities are important aspects of SIGMA; they allow simple models to represent complex systems. Both are discussed in greater detail in Chapter 6.) Click the Cancel button on this edge dialog box.

Unconditional edges test the condition (1==1)*, which is always true.*

Double-click on the self-scheduling edge from the ENTER vertex to the ENTER vertex, which perpetuates customer arrivals. The most notable item in this dialog box is the delay time. Here, the Delay between successive customers ENTERing the system will be a time uniformly distributed between 3 and 8 minutes. This is done by making the delay time for this edge equal to the expression, 3+5*RND. RND is a SIGMA function providing a fraction that behaves like a random number between 0 and 1. Thus, 5*RND will be a number somewhere between 0 and 5; adding 3 will shift the range of this number to between 3 and 8. Close this dialog box.

Open the dialog box between the ENTER vertex and the START vertex. The item of interest on this edge is Condition: SERVER>0. This means that a customer ENTERing the system will START service without delay only if that customer finds that the server is available, i.e., if the condition SERVER>0 is true. Close this dialog box.

Double-click on the edge between the START vertex and the LEAVE vertex. Note that this edge is a double edge: with one edge from the START vertex to the LEAVE vertex and another edge in the reverse direction, from LEAVE to START. The dialog box for this multiple edge is shown in Figure 3.7.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 3.7: Multiple Edge Dialog Box between `START` and `LEAVE` Vertices

Select a Subedge

Edit Edge 4 (Edges between Vertex START and LEAVE)

Num	From	To	Cancelling?
4	START	LEAVE	No
5	LEAVE	START	No

Edit

Delete

Close

Double-clicking on one of the sub-edges in the multiple edge dialog box will produce the dialog box for that particular edge. For example, if you double-click on the line for the edge from `START` to `LEAVE`, a dialog box for that edge will appear.

The edge between the `START` vertex and the `LEAVE` vertex has a delay time of five minutes, indicating that it will take five minutes to wash a single car. Now cancel the dialog box for this sub-edge and look at the dialog box for the sub-edge from the `LEAVE` vertex to the `START` vertex.

The dialog box for the edge from `LEAVE` to `START` tells us that if there are customers waiting in line (`QUEUE>0`) whenever a customer `LEAVE`s the system, the server will `START` service on the next customer in line. To return to the event graph for `CARWASH.MOD` click the Cancel button on the dialog box and clicking the Close command button on the multiple edge dialog box.

3.3.4 Editing the Carwash Model

To get a feel for editing graphical objects, double-click on the edges or vertices in the graph. Then click on some of the items in the dialog boxes and use the keyboard to enter changes to the model. (Pressing the OK button will cause your changes to be temporarily recorded in the model.)

After you have edited a few dialog boxes, click on the File menu and click again on the Open/Event Graph command. Click No when asked if changes to `CARWASH.MOD` should be saved. (Clicking Yes will save the changes you just made to the model.) Next, scroll through the list of model file names until you see `CARWASH.MOD`. Double-click on `CARWASH.MOD` to reread the initial version of this model into SIGMA. The screen in Figure 3.3 should appear again.

If you saved your changes to the model, `CARWASH.MOD`, a backup copy is also saved in your directory as `CARWASH.BAK`. Read `CARWASH.BAK` into SIGMA and save it as `CARWASH.MOD`, if necessary.

3.4. Using Text Files

Now that we have gone over the model, `CARWASH.MOD`, in detail, you might be interested in looking at how it is stored on your computer. The model is stored in a text file by the same name, `CARWASH.MOD`. To look at this file, first open the File menu and click the Text Data/Output command. A dialog box will appear. Click on the down arrow in the drop-down list under List Files of Type and then click on the Other Files option to have all the files included in the list box of file names.

Use the scroll bar to locate `CARWASH.MOD` in the list box; click once on `CARWASH.MOD` to insert this file into the File Name input box. Next, click the OK command button. When the text file appears, you will notice that SIGMA models are saved simply as copies of the dialog boxes for each of the objects in the simulation. Close the file.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

4

Running A SIGMA Simulation

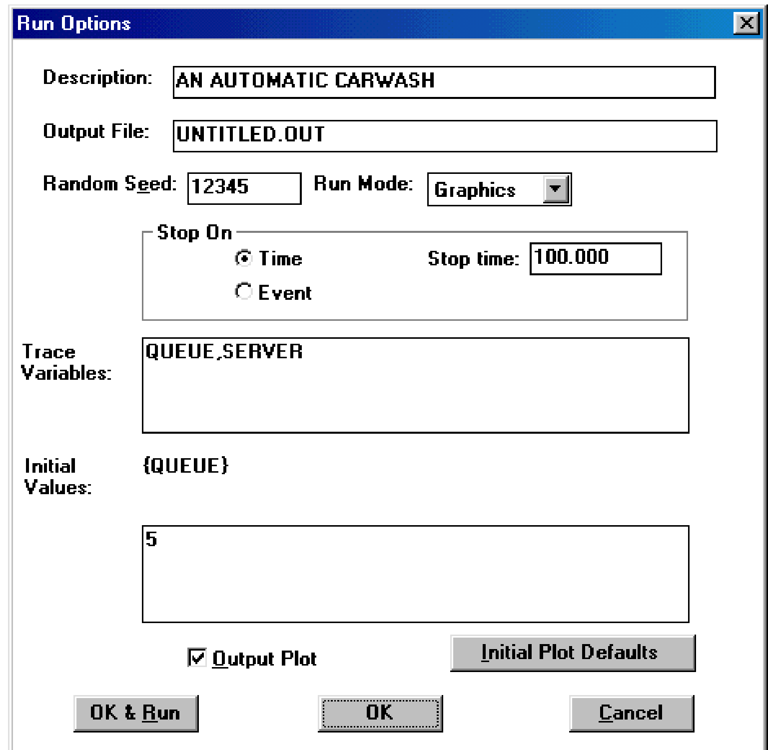
The Run Option dialog box controls how a simulation of a SIGMA model will run. The details associated with running a model are presented—in particular, the speed of the simulation, how the simulation terminates, and how the numerical data will be plotted.

4.1 Running the Model

Before you create your own simulations, you should know how to run SIGMA models. Here, we will continue to use the carwash model to show the basic components of a simulation run.

Start a SIGMA session and open CARWASH.MOD. Next, click once on the Run/Options command. A Run Options dialog box will appear like that in Figure 4.1.

Figure 4.1: Run Options Dialog Box for CARWASH.MOD



This dialog box control how CARWASH.MOD executes. The Description of the model is: AN AUTOMATIC CARWASH. The Output File is the name of the file on your default disk drive where the numerical output from the simulation will be written; here it is UNTITLED.OUT. The Random Seed is set at 12345, and the Run Mode chosen for this model is Graphics mode, meaning that we will see a logical animation of the model during execution. The Variables to be traced are QUEUE and SERVER.

The Initial Attribute for the model is 5. Recall from Chapter 3 that the first vertex created in CARWASH.MOD was the RUN vertex, which has the single input parameter, QUEUE. In order to run this model, we will need to specify an initial value for QUEUE as prompted in this dialog box. Here the initial value for the variable QUEUE is given as 5, meaning that there will be five customers in line when the carwash opens.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

In this model, the ending condition chosen is Stop On Time. The run will stop after the clock has advanced to a time equal to or greater than 100. This dialog box also controls output plots. Here the Output Plot check box is selected, so a simulation plot will appear when the model executes.

Click the OK & Run button at the bottom of this dialog box. (Respond OK if a dialog box appears with the question "Replace existing UNTITLED.OUT?" Also, press No if asked to save changes to the model.)

Watch the graph as the model runs. A simulation plot window with graphical data should appear beside the simulation graph window. (Recall that you can slow down the execution speed by pressing [F2] and speed it up by pressing [F3].) When the run is finished, you can look at the numerical output by clicking the Yes button when asked to "View output trace now?" Enlarge the output window by pressing the Maximize button; scroll through the file to view the entire contents.

A portion of the numerical output for this run is given in Figure 4.2; it is a record of the run history. There is an entry for each occurrence of every event that was monitored from Time 0 to Time 15. (For CARWASH.MOD, the Trace Event box for every vertex was clicked on in the Edit Vertex dialog box, indicating that all events were to be traced.) Here we see the simulated time, the name of each event that was traced, the number of times each event took place, and the values of the Trace Variables, QUEUE and SERVER, at those times. Note that there is a separate column for each traced variable.

Figure 4.2: Standard SIGMA Output File for the Model, CARWASH.MOD

MODEL DEFAULTS

```
Model Name:                 CARWASH.MOD
Model Description:          AUTOMATIC CARWASH
Output File:                CARWASH.OUT
Output Plot Style:          NOAUTO_FIT
Run Mode:                   GRAPHICS
Trace Vars:                 QUEUE,SERVER
Random Number Seed:         12345
Initial Values:             5
Ending Condition:           STOP ON TIME
Ending Time:                100.000
Trace Events:               ALL EVENTS TRACED
Hide Edges:
```

Time	Event	Count	QUEUE	SERVER
0.000	RUN	1	5	1
0.000	ENTER	1	6	1
0.000	START	1	5	0
3.483	ENTER	2	6	0
5.000	LEAVE	1	6	1
5.000	START	2	5	0
10.000	LEAVE	2	5	1
10.000	START	3	4	0
10.653	ENTER	3	5	0
15.000	LEAVE	3	5	1

To return to SIGMA, click the Close command under the Output Control menu.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

4.2 Run Options

The following sections provide additional information concerning the various options available under the Run Options dialog box.

4.2.1 Description

The Description text box allows you to provide a brief description of your model. Although not executable, all of the descriptions in SIGMA are important. The descriptions will appear as in-line comments in your SIGMA-generated simulation source code and English translation.

4.2.2 Output File

The Output File text box allows you to name the file in which you want the numerical output recorded. The default output file name is UNTITLED.OUT. The name of the event, the time each event executed, the number of times each event occurred, and the current values of all state variables that were chosen to be traced are recorded in your output file. (You choose the events that are to be traced by clicking the Trace Event check box On or Off in the dialog box of each vertex.) This output file is readable by a spreadsheet.

> **WARNING**: *Do not write your output over your model. It is advisable to end output files with* .OUT *and model files with* .MOD.

After the simulation run is complete, a dialog box will appear asking if you want to view the output trace now. If you click the Yes button, the output from the simulation will appear in a third window titled UNTITLED.OUT. Click the Maximize button to see the full screen. After examining the output, click the Restore button to return to the previous screens. When you have an opened output file on the screen, you can view the recent history of the output while the model is running. To do so, activate the *simulation graph* window, restart the simulation, and then press the "Refresh" button (in the upper left corner of the output window) while the model is running to update the output file during the run. Long output files should be read by a spreadsheet.

> **WARNING:** *You will not be allowed to save your output on a write-protected or an unrecognized disk drive. If you are using a network, you must specify your complete path, including the drive letter, so you do not write on the network drive.*

The output from SIGMA models is in standard ASCII format that is compatible with most statistical analysis packages. For very large output files, the View Text window in SIGMA will not be able to show the entire output file. If this is the case, a message will appear on the screen. Most likely, too many vertices have been traced. Turn off some traced events in the Edit Vertex dialog boxes or use any ASCII editor (e.g., Windows Notepad) to view the entire file. .

4.2.3 Run Modes

The four run modes available in SIGMA can be found in the Run Mode drop-down list. They are Single Step, Graphics, High Speed, and Time Steps. Note that the run mode can be *changed* during a run. Simply open the Run Options dialog box during the simulation, click on the Run Mode drop-down list, click on the new run mode, and click the OK command button.

In Single Step mode, the simulation will halt after each vertex is executed and wait until the Single Step tool is pressed. This run mode permits you to monitor state variable changes and the list of scheduled events. This mode is particularly useful when verifying the logic of a simulation. When you run a model in Single Step mode, the Single Step Window will automatically open on the screen. This window will show the current clock time, the event being executed, the number of times the event has executed, the value for each state variable being traced, and the pending

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

events list. Included in the window is the Single Step tool. Click the mouse on the Single Step tool or press [Enter] to advance the simulation to the next scheduled event and update the Single Step Window.

Close this window by clicking on the Single Step Control menu and then clicking the Close command or clicking the Close button.. Please note that if you close the Single Step Window during a run, the simulation will revert to Graphics mode. If you want to return to Single Step mode during a run, you must click on the Run Time Information tool to reactivate the Single Step Window.

In Graphics mode, vertices and edges change color as the simulation progresses. Edges change color when their conditions are tested as being true, and vertices flash when they are executed. You see the model being executed. Any event that does not execute during a run will remain shaded. The presence of vertices that never execute may indicate an error in your model logic. This run mode allows you to get a feel for how the model behaves. Hence, it is a good initial method for reviewing your model. If you choose to activate the Run Time Information tool during Graphics mode, a Trace Window that is very similar to the Single Step Window will appear. However, you do not need to do anything to advance the simulation; it progresses automatically. If the *simulation graph* window is active, pressing [F2] will slow the model execution and pressing [F3] will restore it to its original speed.

High Speed run mode bypasses the graphics in SIGMA and produces an output file. High Speed mode is useful particularly when running large simulations for long periods. While High Speed mode does produce a simulation plot as the model runs, it is wiser to turn the Output Plot check box Off so that the plot is suppressed or minimize the plot window. If execution speed is really important, compiling SIGMA-generated C source code is recommended.

Time Steps run mode is useful for working with spreadsheets. The graphs, plots, and output files are updated only when the simulated time advances.

Translate your model to C for dramatically faster run times. Also minimizing the number of windows open, including plots and graphs, will speed up the runs considerably.

4.2.4 Ending Conditions

Two methods are available within the Run Options dialog box to terminate a run. With Stop On Event, the simulation will terminate after a particular event occurs a specified number of times. With Stop On Time the simulation will run until the first event after the specified time.

To control the run duration based on a particular event, click on Event in the Stop On block of the dialog box. Two boxes will appear: a Stop Event drop-down list with all the events in the simulation and an Iterations input box. Suppose, for example, that our carwash simulation is to run until the tenth customer departs. Thus, the run would be controlled by the `LEAVE` vertex, and the number of executions of this vertex would be `10`. You would activate the Stop Event option, click `LEAVE`, and then type "10" in the Iterations box.

If you want to run your simulation for a specific time, click the Time option and enter the amount of simulated time the run is to take.

To stop in the middle of a run, click the mouse on the End Run tool. This stops the current simulation run; however, confirmation is requested to prevent accidentally halting the run. Of course, if the future events list becomes empty during a run, the run will also terminate.

4.2.5 Trace Variables

In the Trace Variables text box, you select the state variables to be recorded and displayed during the run. You list the state variable names (with subscripts in square brackets if appropriate) in a string separated by commas. It is advised to list only a few variables in each run.

4.2.6 Initial Attributes

In the Initial Attributes text box, you can specify the initial values for some state variables. These are the values for the parameters of the first vertex you created in this model. When you run a model, you must give initial values for all the variables you specified as parameters for the first vertex you created regardless of how you might have named them. Initial attribute values are entered as a string of numbers separated by commas.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

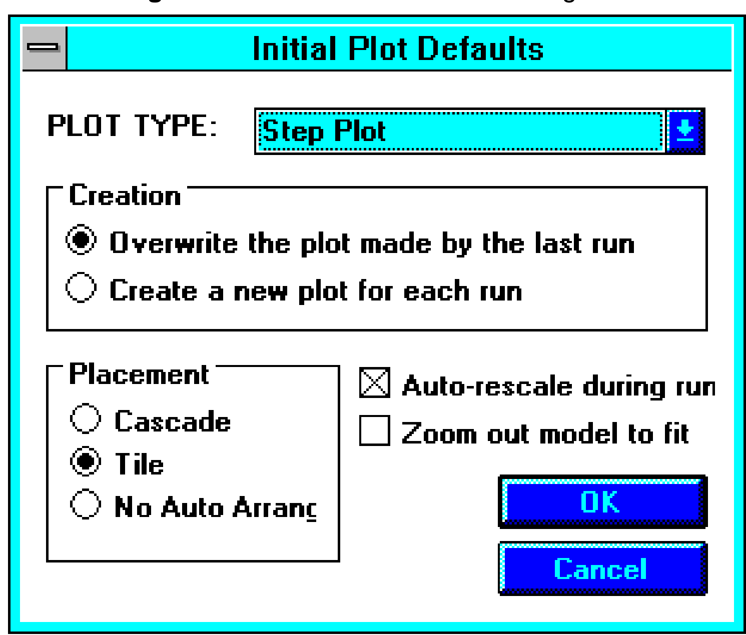
4.2.7 Random Seed

Another starting condition for a simulation is the seed for the pseudo-random number stream; this is any valid integer between 1 and 65000. Enter a random number seed in the Random Seed text box or use the default random seed – 12345.

4..2.8 Output Plot

Graphical plots of the data from a simulation are available to you while the simulation is running if the Output Plot check box is clicked On. (See examples of plots in Chapter 10.) A dialog box will appear when the Initial Plot Defaults button is clicked prior to the creation of a plot window. This box, shown in Figure 4.3, allows you to set up the plots as you wish.

Figure 4.3: Initial Plot Defaults Dialog Box



There are seven plot types available in the Plot Types drop-down list. They include step plots and line plots (which show how variables change over time), scatter plots (which show the relationship between pairs of variables), array plots (which show all elements of an array), histograms (which count the values of variables), autocorrelation plots (which show second-order dependence in the output), and standardized time series (which can be used to detect trends and initialization bias). Plot Creation options include: creating a new plot for each run or having a new plot overwrite the previous plot. Placement Options are also available.

Note that double-clicking anywhere in the *simulation plot* window during a run will open the Output Plots dialog box, shown in Figure 4.4. In it, you can change the plot type and its characteristics. Click the down arrow at the right of the Plot Type drop-down list to see the various plot types available to you. If you click on a different plot type and then click the OK button at the bottom of the dialog box, you will see a new graphical representation of the output.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 4.4: Output Plots Dialog Box

Output Plots

PLOT TYPE Scatter Plot

X Axis
Variable Time Batch Size 1
Label Time
Min 0 Max 390 Step 65

Y Axis
Variable QUEUE
Label QUEUE
Min 0 Max 6 Step 1

Title UNTITLED.OUT (QUEUE vs. Time)
Title Font Size Multiplier 1.000

Rescale OK Cancel

The Step Plot, Line Plot, and Scatter Plot have identical Output Plot dialog boxes. For these plots, you can select the X Axis (horizontal) and Y Axis (vertical) and their limits. The Array Plot, Histogram, Autocorrelation, and Standardized Time Series have slightly different dialog boxes.

Labels and Axes: You can select a font type for the title and axes labels for all plots by clicking the Select Font command under the Plot/Edit menu. The Title Font Size Multiplier box (on the Output Plots dialog box) can be used to make the title of the plot larger or smaller than the axes labels.

Data Truncation and Scaling: The Min (lower) limit and Max (upper) limit for the X Axis can be used to eliminate data from each end of a run that might be contaminated by run initialization or termination bias. Note that truncating biased data in this way is only effective after a run has been completed since these limits will be rescaled during a run (if the automatic rescale switch is turned on as it should be). Although it is much less common, you might also want to set maximum and minimum limits for the Y Axis to focus on a particular range of values or to plot several different output series on the same scale for comparison purposes.

Batching: In the X Axis group, you are also given the option of setting a batch size for the output data. Batching simulation output data is a common smoothing technique where adjacent and exclusive groups of observations are averaged. The resulting series of "batched means" can then be plotted and analyzed. A detailed example of using batched means to estimate confidence intervals is given in Chapter 10.

Histograms: The histogram plot gives the relative counts of observations in different intervals (called cells or "slices"). The Histogram Plot dialog box has as its Y Axis the count of the number of observations in each slice. You can change the number of slices to obtain smoother or more detailed histograms.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Autocorrelation Plots: The autocorrelation plot gives the correlations between observations in the output series as a function of how close they are to each other. The X Axis for this plot is the "lag" between the observations (a lag of 1 is for neighboring observations and a lag of 2 is for observations that have one observation between them, etc.). Setting a fairly small maximum value for the lag is a good idea for two reasons: computing the values for this plot will be faster with a small maximum lag, and because correlation estimates for observations that are far apart are not very accurate.

Trend Detection: The Standardized Time Series (STS) plot is a powerful tool for both confidence interval estimation and detecting trends in the output data. Standardized time series are sensitive to changes in the level of a sequence of observations. This is particularly useful in detecting initialization bias in a simulation output series. If the STS plot seems to be rather jagged and spends about the same time above and below zero, then there is probably not much of a trend in the output. If the STS plot is smooth and pulled off in either a positive or negative direction, an increasing or decreasing trend is indicated. A positive trend pulls the STS series to the positive side of zero, and a negative trend pulls the STS plot to the negative side of zero. For practical purposes, you can simply look to see if the STS plot tends to be positive (negative) during a run, indicating an increasing (decreasing) trend in the data.

Examples of using STS plots to detect very weak trends in the output data are presented in Chapter 10. A detailed discussion of using STS information to estimate confidence intervals also is deferred to Chapter 10.

4.2.9 Command Buttons

The command buttons in the Run Options dialog box are OK & Run, OK, and Cancel. The first saves changes and runs the model, the second saves changes and returns to the static simulation graph, and the third simply closes the dialog box without saving the changes.

4.3 Exercises

The models referred to in these exercises are SIGMA *models.*

4.3.1 A Semi-Random Walk

Build a simulation of a semi-random walk. The location of the walker on the line is given by the variable, `X`. Every step is in an opposite direction and has an expected step length equal to 4 feet. However, the steps to the right are uniformly distributed between 3 and 5 while steps to the left are exactly 4 feet long. Would you expect the location of the walker to change much over time?

4.3.2 Stopping a Run Based on Event Counts

Run `CARWASH.MOD` until 10 customers `LEAVE` the system.

4.3.3 Time Scaling

Without changing the run stopping criterion in the Run Options dialog box, double the effective run duration of `CARWASH.MOD`. (Hint: Rescale the unit for measuring time from minutes to half-minutes.)

4.3.4 Event List Dynamics

Twenty jobs were run through a computer system with one CPU. They all had one of three priorities (lowest number has higher priority and is executed first). A historical data file of these jobs is as follows:

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Job number	Time job was submitted	Job priority	CPU time used by the job
1	.00	1	.02
2	.00	2	.01
3	.01	3	.02
4	.02	2	.04
5	.04	3	.02
6	.05	2	.01
7	.07	1	.01
8	.12	1	.01
9	.14	2	.02
10	.15	1	.01
11	.15	2	.01
12	.17	1	.02
13	.18	3	.04
14	.21	1	.03
15	.22	2	.05
16	.23	1	.02
17	.23	1	.01
18	.26	1	.03
19	.33	2	.02
20	.34	1	.03

(a) By hand, schedule the job submission and job completion events as they would occur if all jobs had the same priority; assume that nothing else changes except the priorities. Schedule only the next job submission and next job completion event, not all future job submissions (that is, at most two events will be scheduled at any given time). At time 0.25 what is the state of the system, i.e., the number of jobs in the queue, the status of the CPU (busy/idle), and the events that are scheduled to occur in the future (and when they are scheduled)?

(b) Do part (a) with the execution priorities enforced.

(c) Assume that a second identical CPU has been installed and that the two processors operate in parallel on the same single queue of jobs. A single CPU will finish a complete job before taking the next job in the queue. With job priorities enforced, plot the queue size and number of idle CPU's every .01 time units.

4.3.5 A Theater with Limited Seating Capacity

Suppose that customers arrive at a movie theater at a uniform rate of one every 10 to 25 seconds and each is served in constant time of 20 seconds. There is limited seating in the theater, so the ticket window closes after it has sold 100 tickets. Model this queue.

4.3.6 A Restaurant

A popular restaurant, which does not accept reservations, serves parties in the order that they arrive. For dinner, parties arrive with a rate uniformly distributed between 5 and 15 minutes. Parties range in size from 2 to 6 people, each with the same probability of arrival. Service time for each party, regardless of size, is 2 hours. Assume the restaurant has a capacity for 12 parties. Model this system.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

5

Event Graph Modeling

This Chapter presents numerous ways to enrich and modify the basic queueing model introduced in Chapter 2. The basic single server queue model, CARWASH.MOD, can serve as a template for very sophisticated models. With a few modifications and enrichments, the carwash model can be made to simulate many systems, including those with multiple servers with identical capabilities, multiple servers working in parallel with different capabilities, batching operations, assembly operations, service failures, and closing times. Furthermore, SIGMA is not limited to modeling discrete event simulations. Indeed, any computer simulation can be modeling using SIGMA Here demonstration models have been included that represent continuous time modeling of an aquatic ecosystem, financial risk analysis in coupon bond pricing for an investment bank, and critical path evaluation of a construction project.

Several important features of event graphs are expanded upon including edge attributes, event parameters, and event canceling as well as methods for modeling multiple resources and transient entities. This chapter includes a discussion of the modeling technique of using Boolean variables, which allow alternatives to be expressed more easily, as in "if-then-else" statements, "do-while" loops, and nested loops.

5.1 Enrichments to Our Basic Model

You will find that the following discussions are much easier to understand if you read while looking at the models on your computer. The name of the model(s) corresponding to each section is given in the title of that section. You only need to consider one vertex at a time and its exiting edges. One of the major advantages of event graphs is that you do not have to look at the whole graph to understand what is happening Look at one vertex at a time along with its *exiting* edges—the graph takes care of connecting the model correctly.

5.1.1 Multiple Identical Parallel Servers: BANK1.MOD

We will enrich our original model of a single server queue with one server in order to represent several identical servers operating in parallel on a single waiting line of customers (e.g., tellers at a bank). We continue to use the same 1 = available and 0 = unavailable convention that we used with the single server; this allows us to easily change the number of servers available in our model. We simply let SERVERS denote the number of idle tellers in a simulated bank. No matter how many tellers we have in our model, the condition, SERVERS>0, will always mean that at least one server is available.

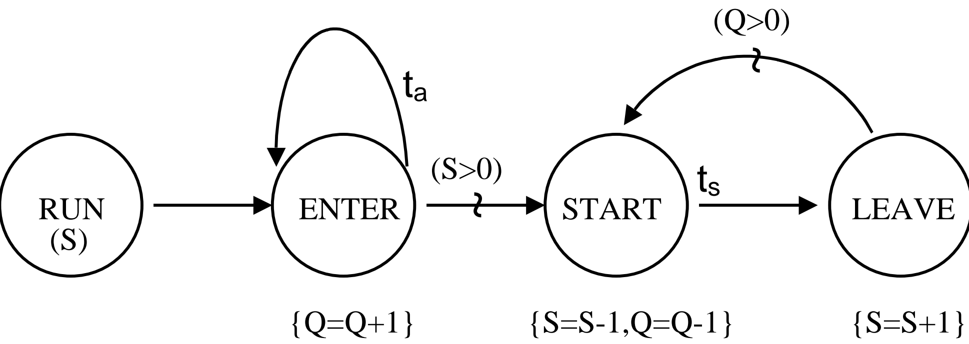
The only modification necessary of our basic model to change it from a single server system to a multiple server system is to set SERVERS equal to the total number of tellers in the system in the RUN vertex. We do this by first entering SERVERS as a parameter in the RUN vertex dialog box and then entering the number of total servers (3) in the Initial Values box in the Run Options dialog box. We initialize the number of servers as an input parameter so we can change the number of servers easily each time we run the model. All edge conditions remain the same; however, service times are now random.

The SIGMA graph for this system appears in Figure 5.1. The state variables used to describe this system are:

QUEUE	The number of customers currently waiting in line, initially equal to 0 by default.
SERVERS	The number of idle tellers, specified as an input parameter to the initial RUN vertex.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.1: Multiple Server Queue



As before, we will "read" this model by sequentially examining the state changes and exiting edges. First, let us look at the vertices.

In the RUN vertex, the number of SERVERS is read from the Run Options dialog box and the run started.

In the ENTER vertex, the arrival of the next customer is modeled with the state change, QUEUE=QUEUE+1.

In the START service vertex, the queue and the number of free servers are both decremented with state changes, QUEUE=QUEUE-1 and SERVERS=SERVERS-1.

When the service is finished, the customer can LEAVE. In the LEAVE vertex, the number of idle servers is incremented with the state change, SERVERS=SERVERS+1.

Next, we read the edges that exit from each vertex with one sentence per edge.

When the RUN is started, the first customer is unconditionally scheduled to ENTER right away.
Subsequent customers ENTER at intervals that are uniformly distributed between 3 and 8 minutes.

When a customer ENTERs, if there is an idle server (SERVERS>0), service will START without delay.

Once service STARTs, the customer will be scheduled to LEAVE after a time delay equal to the service time of between 2 and 6 minutes.

When a customer LEAVEs the bank, if there are still customers waiting (QUEUE>0), the server will immediately START service on the next customer.

Notice that the vertices for this example are identical to those in the carwash model. The only difference is that rather than setting the SERVER to busy or idle (0/1) when service starts and finishes, we increment and decrement the total number of idle servers. In fact, the single server carwash model is a special case of this model if the variable SERVERS is initialized to equal one in the RUN vertex.

We may wish to study the behavior of this system under different service speeds and demand rates. For the time being, we are only interested in resident entities - i.e., the servers and queues. We will observe both the utilization of the servers and the size of the queue. There is no reason here to model individual customers, as they are assumed to be identical; however, modeling customers is important in later examples. Modeling only resident entities simplifies a model considerably, as we shall see when we enrich this model to explicitly include each customer. With a little practice, either type of model is very easy to build.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

5.1.2 Batched Service: BATCHSIZ.MOD

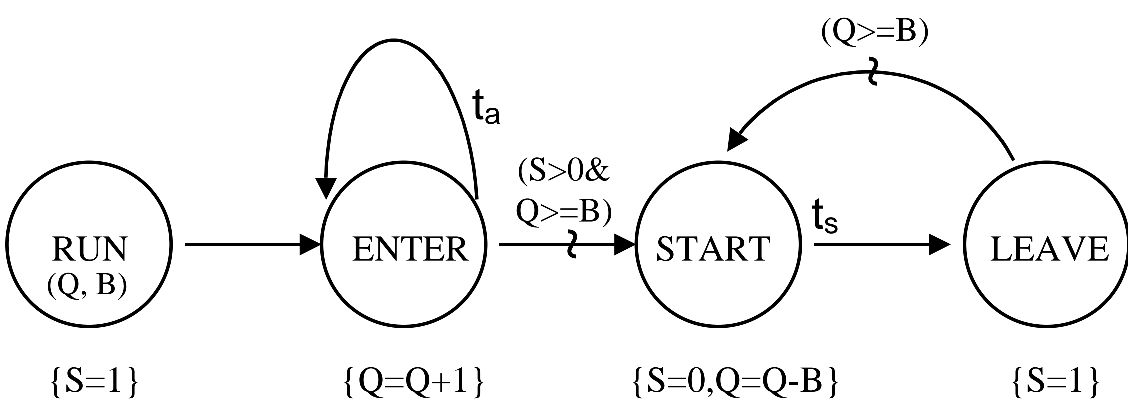
Batched service occurs when more than one item is processed at the same time. Examples include a furnace that processes multiple parts, an oven in a bakery, or a shuttle bus that transports airline passengers.

For this enrichment of the carwash model, we will denote batch size with the variable B. As a full batch is required for processing, we simply change the edge condition for the edge between ENTER and START to S>0&Q>=B. (Note that & is the symbol for a Boolean "and" operator that indicates that both conditions must be true.) Thus, if there is a server available *and* the queue is equal to or greater than the desired batch size, start service. We also would change the edge condition between LEAVE and START to Q>=B. The state change vector for the START event is now

```
S = 0, Q = Q-B
```

so that a full batch is removed from the waiting queue every time a START event takes place. This enrichment is shown in Figure 5.2.

Figure 5.2: Event Graph for a Queue with Batched Service



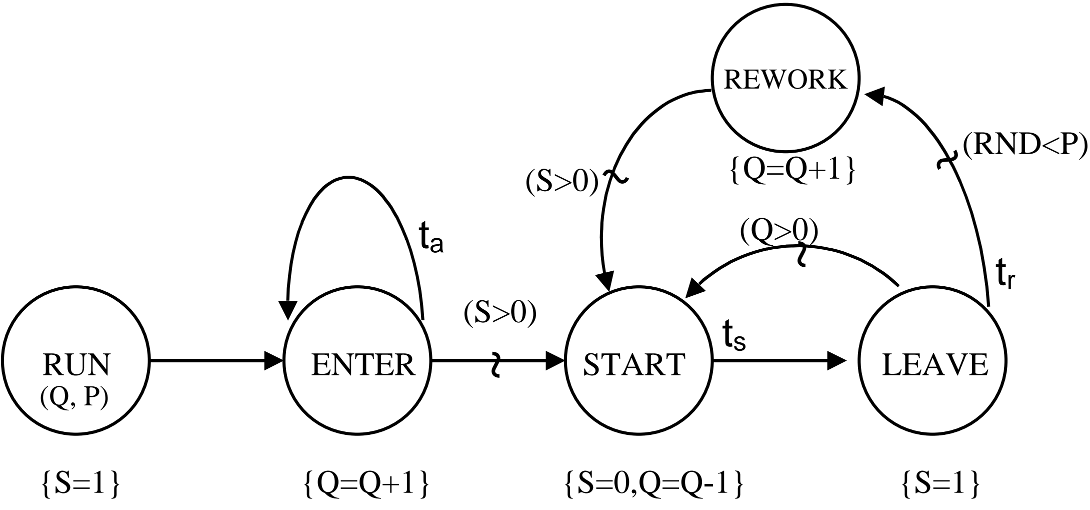
If we were simulating a system where it is possible to process a partial batch, the edge conditions for our original carwash model would not change. The state change for the START vertex would decrement Q by the minimum of the Q and B, Q=Q-MIN{Q;B}.

5.1.3 Rework: REWORK1.MOD

In some cases, it may be necessary to rework a part that has just been processed. This situation is most common when layers of material are applied to a part, as in a painting operation. The only change needed in the carwash model to create this new model is to add a feedback loop in the form of an edge going from LEAVE to a new vertex called REWORK and another edge from REWORK back to START as shown in Figure 5.3. In addition, the probability of rework P is an input variable to the model. A random number, RND, that falls (strictly) between 0 and 1 is drawn; if this number is less than P, rework is required. The probability that the condition RND<P is true equals P.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.3: Event Graph with Rework



Notice that if we model REWORK in this manner, LEAVE, REWORK, and ENTER might all schedule a START vertex at the same time. However, SIGMA automatically breaks time ties correctly, and the execution priority for the START vertex (assigned by the edges scheduling this event) will be higher than the execution priority for all other vertices. If not, an ENTER or REWORK vertex might schedule a START event (if the server already was made free by the LEAVE event) at the same time that the LEAVE event scheduled a START event. These START events would each, in turn, schedule LEAVE events, creating "phantom" extra servers in our model. If the START event is executed before any other events, there is no problem since this makes the appropriate edge conditions (S>0) false.

This model illustrates problems that may occur when two events are scheduled at the same time. Several popular professional simulation languages have no mechanism for breaking ties in the times that simultaneous events are scheduled. This is a bigger problem than one might think at first glance, since in a "balanced" system, time ties are intentionally designed into the system. The so-called Just In Time production systems invented in Japan are examples.

WARNING: *Watch out for unwanted side effects that come when events are scheduled simultaneously!*

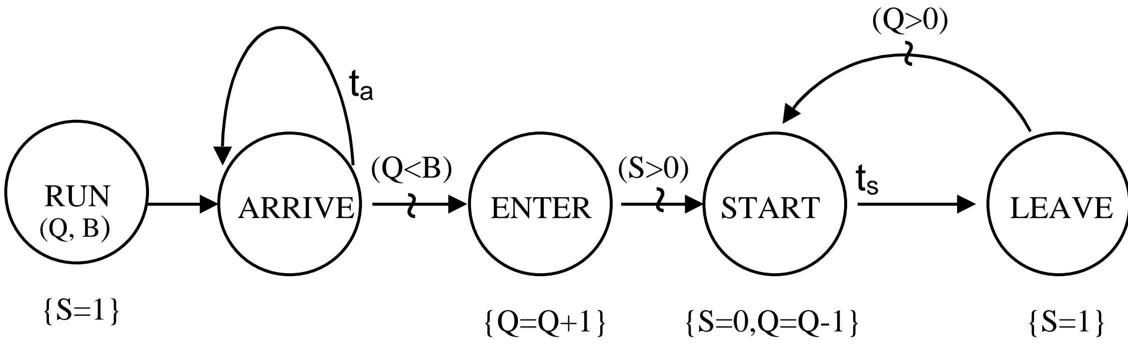
Sometimes rework requires an additional preparation time, called rework setup time. To model this additional rework preparation time (e.g., to remove paint), we simply place a delay on the edge from LEAVE to REWORK equal to the rework setup time, here t_r.

5.1.4 Limited Waiting Space: BUFFERQ.MOD

In our next model, we will simulate a system with limited waiting space in the queue. The amount of waiting space is called the buffer size.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.4: Event Graph with Limited Waiting Space



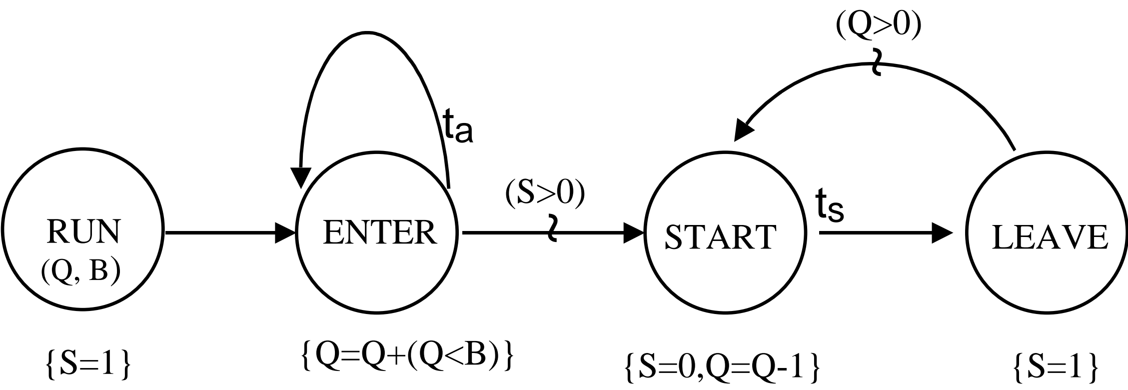
We define variable `B` as the buffer size and specify this as a parameter for the `RUN` vertex, to be initialized when the model is run. To create this model, we modified our basic carwash model by adding an `ARRIVE` vertex between the `RUN` vertex and the `ENTER` vertex. The self-scheduling edge used to create customer arrivals is moved from the `ENTER` vertex to the `ARRIVE` vertex. We also conditioned the edge from `ARRIVE` to `ENTER` to require that there be an empty space for the arriving customer to wait. See Figure 5.4.

It is also possible to model this system another way, using a Boolean variable. When using Boolean variables, a condition takes on the value of `1` if it is true and `0` if it is false. Again using the carwash model, we can incorporate limited waiting space by altering the expression for the state change in the `ENTER` edge from `Q=Q+1`, which models an arriving customer unconditionally joining the queue, to the expression

```
Q=Q+(Q<B)
```

If the condition `(Q<B)` is true, there is waiting space for an arriving customer and the Boolean variable in parentheses will be equal to `1`. Then `Q` will be increased by `1`. If the `Q` equals or exceeds the buffer size, there is no waiting space. Consequently, the condition will evaluate to `0`, and `Q` will not be incremented (`0` is added to `Q`). See Figure 5.5 for the event graph for this model.

Figure 5.5: Using Boolean Variables to Model Limited Waiting Space

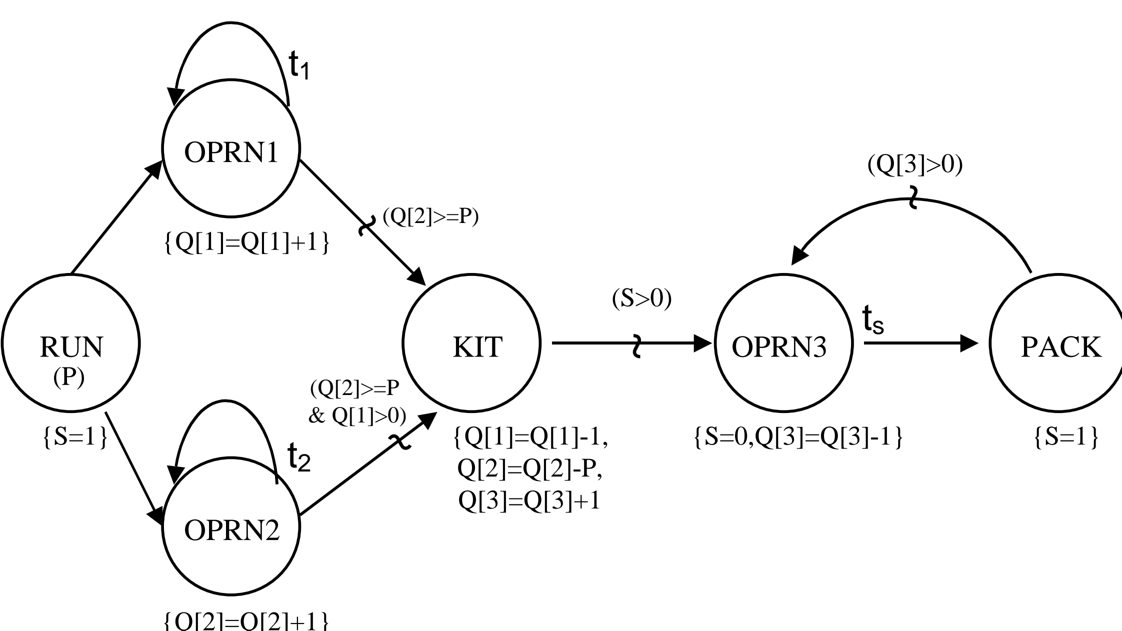


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

5.1.5 Assembly Operations: ASSEMKIT.MOD

In an assembly operation, several different types of parts are put together into a single unit. The collected parts for a finished assembly are sometimes called a "kit." A kitting and assembly operation, where one part from Operation 1 is joined with P parts from Operation 2 to form a kit for assembly Operation 3. is modeled in Figure 5.6.

Figure 5.6 Event Graph for an Assembly Operation



5.1.6 Different Servers Working in Parallel: SLOFAST0.MOD SLOFAST1.MOD

Some systems have two servers with different characteristics operating in parallel (e.g., machines with different average processing speeds, the high-speed check out line at the supermarket). In this example, there is a new, faster machine working with last year's slower model.

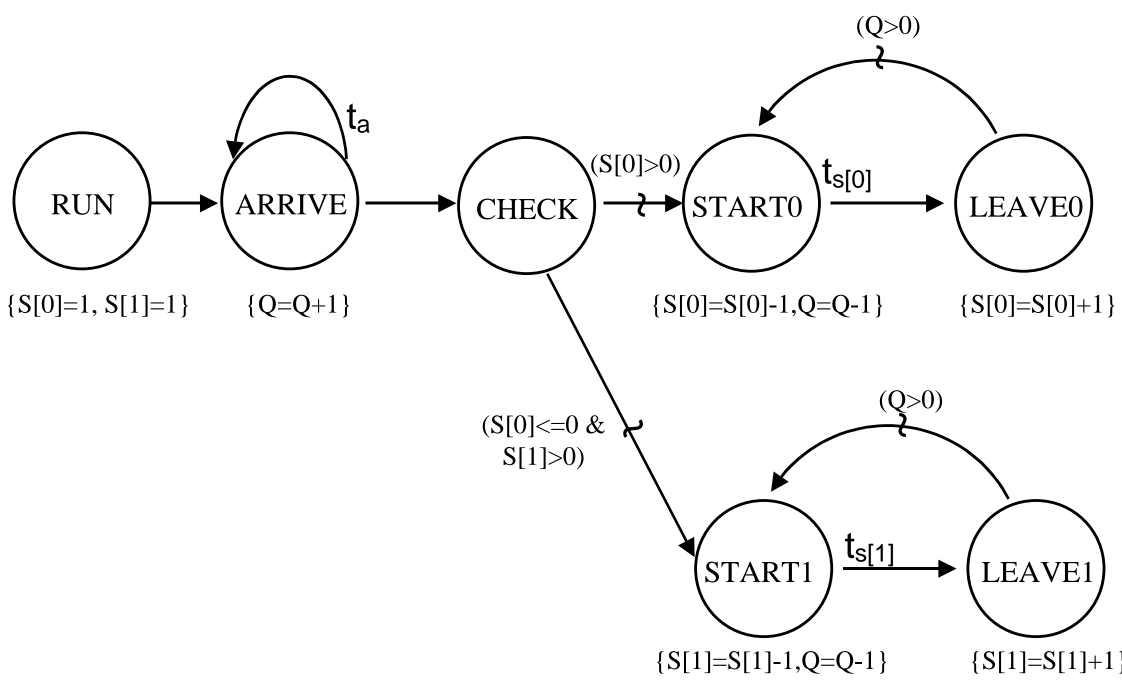
We will designate each type of machine with an index, A, that indicates the age of the machine. For the new machine, A=0, and for the older machine, A=1. The status of each machine is given by the values of state variables S[0] and S[1]. The status of a machine will be equal to 0 if the machine is busy and equal to 1 if the machine is idle. We will denote the average processing times for the two machines by t_s[0] and t_s[1]. We assume that when both machines are idle, the faster machine, (A=0), is preferred.

The most direct way to model these two types of machines operating in parallel is to add a CHECK vertex and a second pair of START and LEAVE vertices for the second server. This was done in Figure 5.7 and in the model SLOFAST0.MOD. From the viewpoints of the servers, this model with a single waiting queue and two servers is the same as a model with separate waiting lines for each server with line "jockeying" (customers in a queue will change lines if they see that the other server is idle).

Parts arrive at the ARRIVE vertex in the usual manner with a (perhaps random) interarrival time of t_a. We will increment the count, Q, of the number of parts in the queue at the ARRIVE event. At the CHECK vertex, we check to see if there is an idle machine. If the faster machine is available, work will start with that machine. In the START0 vertex, the state change S[0]=S[0]-1 makes machine 0 busy; S[0] goes from a value of 1 to a value of 0.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.7: Two Dissimilar Servers Working in Parallel



If the fast machine is busy and the slower machine is idle, work will start with machine 1 at the START1 vertex. The test for this is the condition S[0]<=0&S[1]>0 on the edge from CHECK to START1. When the conditions on both edges exiting the CHECK event are false, both servers are busy and nothing further will happen until the next event occurs (which will be an ARRIVE, LEAVE0, or LEAVE1 event).

Just as we enriched our model of the single server queue to include multiple identical servers working in parallel, we can easily enrich this model to include many servers of two types. We simply redefine S[A] to be the number of idle servers of type A and initialize our model (at the RUN vertex) with the total numbers of each type of server. This model is implemented as SLOFAST1.MOD .

5.1.7 Periodic Resource Unavailability: FAILURE.MOD

In Section 5.2.2, modeling of service failures will be used to illustrate event cancellation. Here we present another way to model service failures without using canceling edges that is more general, runs faster, and is easier to implement. Most simulators have the ability to model resources becoming unavailable, such as a machine going off-line or a worker going on a break. However, these simulators tend to have one or more of four major shortcomings

First of all, the time until failure is often simply measured as clock time, not the time the resource is working. This allows resources to fail even when they are not busy. This is usually nonsense—if a machine is idle, how can you tell that it has failed?. Secondly, the Time Between Failures (TBF) rather then Time To Failure (TTF) is sometimes modeled. This allows resources that are already broken to break yet again (again this is nonsense). Thirdly, there are only limited options on what is done with the job currently being processed when the resource goes off line. Several things can be done with jobs being processed when a resource goes off line (called the 4-Rs).

Reject:	the job in process is discarded.
Resume:	work on the job in process is continued.
Restart:	processing is started over.
Rework:	additional work needs to be done before the job can be restarted.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

An addition, a job can be Rerouted to another resource or Rescheduled to be processed by another resource (a 5th and 6th R?). Finally, most simulators do not allow for multiple failures while processing the same job.

The model, `FAILURE.MOD`, remedies all these shortcomings for multiple parallel resources. That model is rather complex and general; we will first discuss an event graph module for a single resource that overcomes the first three shortcomings. This model is an excellent application of Boolean variables in SIGMA.

In what follows we will enrich our elementary queueing model to permit quite general resource unavailability. For our model, we will define the following variables.

Q = number of jobs in line
S = status of the resource (1 if available, 0 otherwise)
PT = current processing time for I^{th} resource.
TTF = time to failure
TTR = time to repair
FAIL = failure flag (1 if failure occurs)

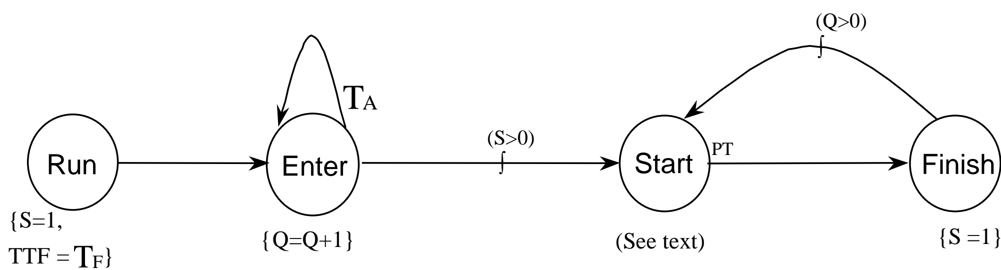
Input data to our model includes the following Random Variables, which will be generated when they are needed.

T_A = time between job arrivals
T_P = processing time
T_F = time to failure
T_R = time to repair

Modifying an event graph to model failures, while detailed, is very simple. Whenever a new job is started, we generate its processing time (`PT`) and check to see if the resource will fail before the job can be finished (`TTF<PT`); if so, we increase the job "processing" time to include the time to repair the resource and take care of the job currently being processed. If the resource is not due to fail before the job finishes, we simply decrement the remaining Time To Failure by the processing time for that job.

Figure 5.8 is the event graph for this model. An initial Time To Failure (`TTF`) for the resource is generated when the run is started. The processing time for a job includes any needed repair time for the resource and the time to dispose of or finish the job currently being processed. All other edges are the same as for non-failing resources.

Figure 5.8: Resuming Work after a Resource Failure



As mentioned earlier, there are only two changes to our basic model, the time to failure is set in the `Run` event and the processing time is computed in the `Start` event to include the repair time if a failure occurs while a job is being processed. The state changes for the `Start` event are as follows (with comments using generic times).

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
{S=0,                              / Make the server unavailable
Q=Q-1,                             / Remove job from queue
PT=T_P,                            / Generate job processing time
FAIL=(TTF<=PT),                    / Is there a failure during processing?
TTF=(FAIL==0)*(TTF-PT)             / No failure - decrement Time to Fail
     +(FAIL==1)*(T_F -(PT-TTF))    / Failure - set new Time to Fail
PT=(FAIL==0)*PT                    / No failure: process job
     +(FAIL==1)*(PT+T_R),          / Failure: resume processing after repair
```

If the job is discarded when there is a resource failure, simply replace PT with TTF in the sixth line in the above state change for the Start Event. PT is simply the time from when the resource starts working on a job until that resource again becomes available to process the next job (perhaps after a failure and repaired).

5.2 Event Cancellation

In some cases, the occurrence of an event will cause some previously scheduled event to be cancelled. For example, if a machine in a factory simulation happens to break, the completion of the current job will have to be cancelled. In event graphs, cancelling edges are shown as dashed arrows. Conditions under which the originating vertex will cancel the destination vertex are shown in the same manner as scheduling edges. If the edge condition for a cancelling edge is true, cancellation of the destination vertex is assumed to occur immediately; all the delay times for cancelling edges are equal to zero.

5.2.1 Closing Time: CLOSEIT.MOD

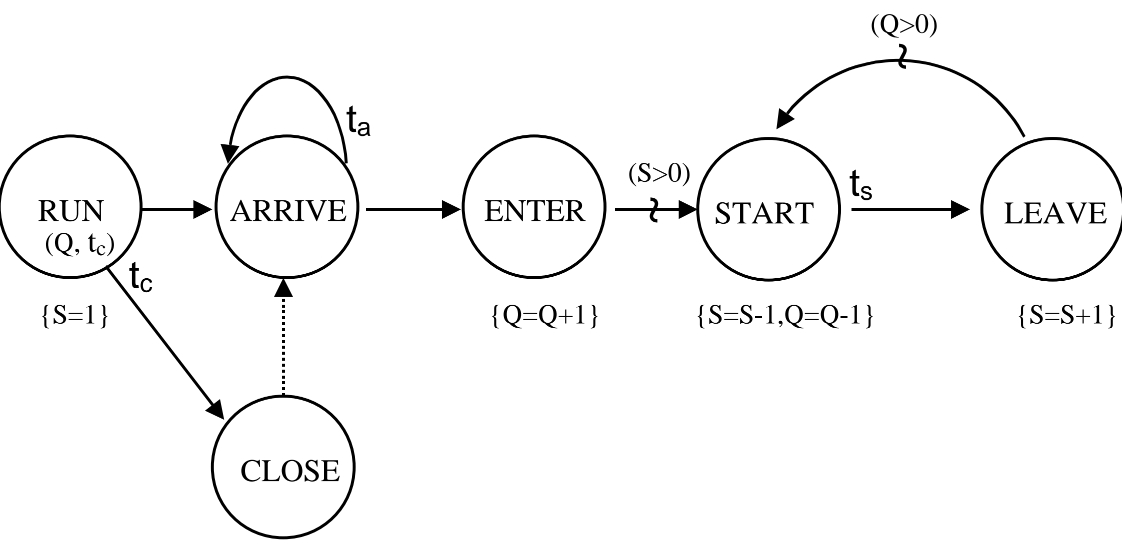
Here we expand the carwash model to include a daily closing time. When the simulated time, CLK, reaches the value of the input variable, CLOSING_TIME, we will no longer admit customers to the system. Of course, customers already waiting in line when we close will be served. There are several ways to model this. A straightforward way might be to add an ARRIVE vertex like we did in BUFFERQ.MOD and place a condition on the edge from ARRIVE to ENTER that the current time be less than the value of CLOSING_TIME. The problem here is that we will not know in advance how many customers are waiting in line when we close; therefore, we will not know when we should stop the simulation run. We could set the run time to a large number so that the customers in the system have ample time to leave the system before the simulation ends.

Since customer arrivals are generated by having each ARRIVE event schedule the next ARRIVE event, a more efficient model for closing the queueing system would be to cut off the stream of customer arrivals at CLOSING_TIME. However, placing the condition, CLK<CLOSING_TIME, on the self-scheduling edge that generates arrivals would not be correct. Before closing time, an ARRIVE event might be scheduled to occur after CLOSING_TIME.

A correct way to stop the arrival of customers at closing time is to cancel the next scheduled ARRIVE event (and thus all subsequent customer arrivals) at CLOSING_TIME. Any waiting customers will be serviced until there are no more events on the future events list; by default, the run will stop with an empty events list. (An empty future events list is a default run termination condition for almost all discrete event simulation systems, including SIGMA.) This model for closing a system is pictured in Figure 5.9

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.9: Queue with a Closing time



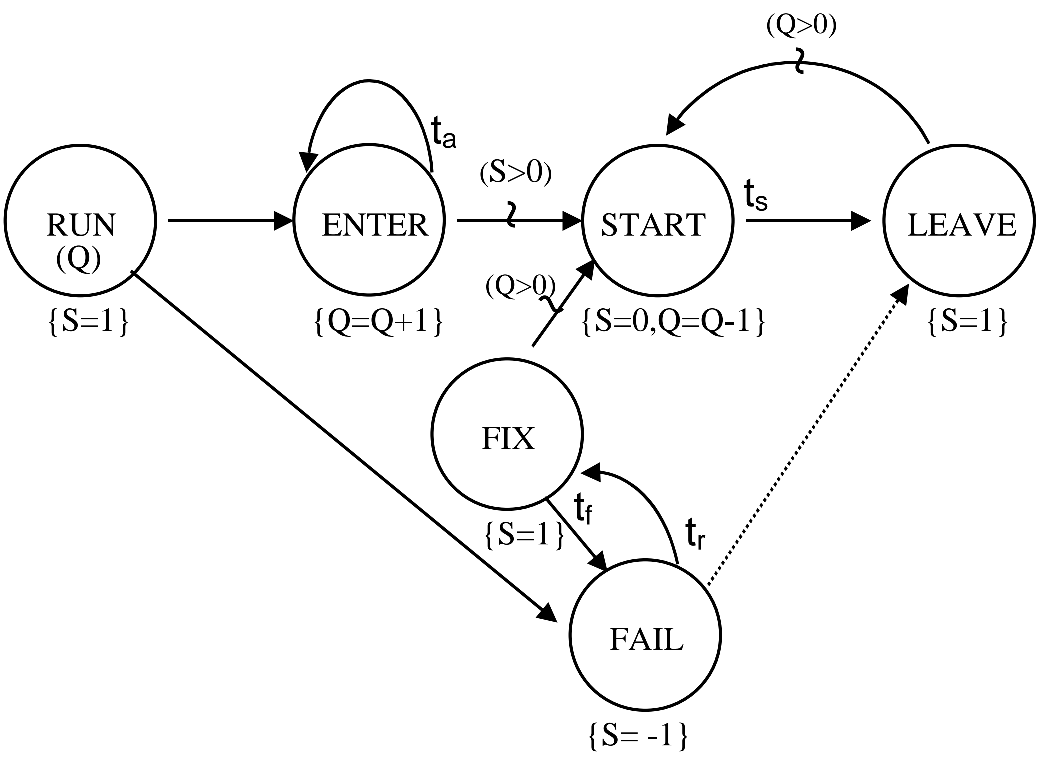
5.2.2 Single Server with Intermittent Service Failures: `BRKDN.MOD`

The model in this section is intended to illustrate another use of event canceling—to model the failure of a resource. A more general way of modeling resource failures is presented in Section 5.1.7. Another enrichment of the basic carwash model allows us to model periodic breakdowns of service by adding two vertices and expanding the definition of the status of the server to include "broken." (see Figure 5.10) Now server status is represented by `1` (available), `0` (busy), and `-1` (broken). The breakdown process is modeled with the following vertices:

> `FAIL` is the vertex where a service failure occurs. Any job that may be in progress and scheduled to `LEAVE` the system is cancelled. The part being worked on when the machine broke down is destroyed. The server's status is changed to "broken" with the state change, `SERVER=-1`.
>
> `FIX` is the vertex where the server is repaired. The state change, `SERVER=1`, makes the server once again available.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.10: A Queue with Service Failures



Additional edges added to the carwash model to simulate server breakdowns include the following:

An unconditional edge from RUN to FAIL is used to schedule the first service breakdown when the run starts (t_f)

An unconditional edge from FIX to FAIL is used to make the server break down after a randomly distributed time interval (t_f). This is regardless of whether or not the machine actually did any work during this interval.

A cancelling edge from FAIL to LEAVE is used to cancel the LEAVE event without delay if the server is busy. Note that if the server is idle when a failure occurs, we will attempt to cancel a nonexistent LEAVE event (which does nothing if the server is not busy).

An edge from FIX to START, conditioned by (QUEUE>0), is used to immediately start the server working after being repaired if there are parts waiting in the queue.

An unconditional edge from FAIL to FIX, with a delay time of t_r, is used to schedule the server to be fixed after the repair time.

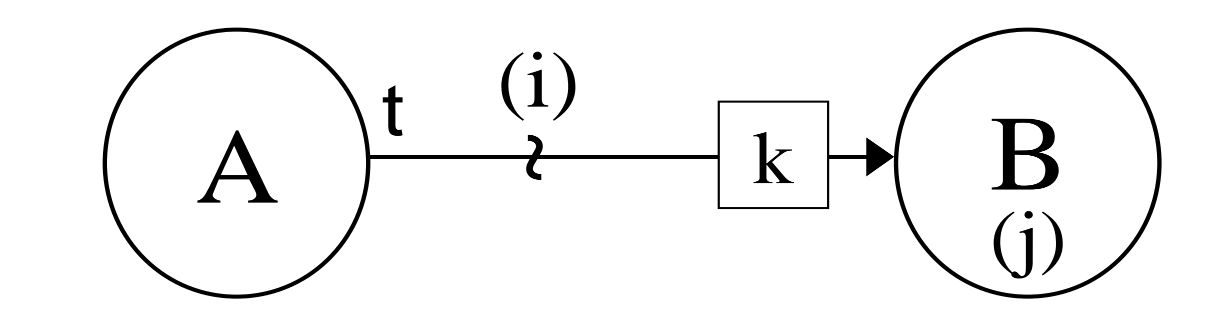
If we assume that a part in service is not affected by a machine failure, we would add a variable to keep track of the amount of service time remaining on a part when failure occurs. Also, it might be more realistic to model machine failures as being dependent on how much work was done since the last failure rather than on the elapsed time. This situation is modeled in Section 5.1.7 defining a new state variable to accumulate the working time until the next failure.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

5.3 Event Parameters and Edge Attributes

An important enrichment of our basic model is the parameterization of event vertices: similar events can be represented by a single vertex with different parameter values. For example, say there are several machines working in a factory. A "start processing" event might be parameterized to indicate which machine is involved. We might call this event `START(M)`. `START(3)` would indicate that machine number `3` is starting. Similarly, a "finish processing" event might be parameterized to indicate which machine has just become idle. We might call this event `FINISH(M)`; `FINISH(6)` would indicate the end of service on machine `6`.

Figure 5.11: Conditional Edge with Delay Time and Parameter Passing



We parameterize event graphs using the notation in Figure 5.11. The edge on a simulation event graph is now "read" as follows:

> *if condition* `(i)` *is true at the instant when* `A` *occurs, then event* `B(j)` *will be scheduled to occur* `t` *minutes later with parameter,* `j`, *equal to* `k`.

The values for the expression `k` are computed when event `B` is scheduled. The state variables `j` are set to these values when `B` is later executed. In general, the attribute, `k`, can be a string of the *values* of expressions, separated by commas, and `j` can be a vector of state *variables*; again, separated by commas.

Consider a network of waiting lines, such as those at an airport terminal. The number of customers in each queue in the system might be represented by a variable called `NUM_IN[QUEUE]`. Here `[QUEUE]` identifies a particular line and `NUM_IN` is an array of integers (an integer state variable with a size greater than one). Whenever a new customer joins a waiting line, the number of customers in that queue is increased by one. The vertex in the graph representing this event will contain the state change, `NUM_IN[QUEUE]=NUM_IN[QUEUE]+1`. A copy of the dialog box for this vertex is shown in Figure 5.12. This vertex will require a value for the parameter, `QUEUE`, telling which line in the system the customer joins. A value for the parameter, `QUEUE`, is passed as an attribute of the edge that scheduled this vertex.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.12: EVENT Vertex Dialog Box

With cancelling edges, like scheduling edges, it is sometimes necessary to designate which of several scheduled instances of an edge destination vertex is to be cancelled. If several machines are working and one breaks, an edge attribute can be used to identify which machine has broken so that its job completion vertex can be cancelled. If no edge attributes are specified, only the first scheduled occurrence (if any) of the edge destination vertex is cancelled. If attributes are given on a cancelling edge, the future events list is searched for an exact match by vertex type and attribute values; only the first scheduled event with an exact match of edge attribute values is cancelled. If you want to cancel all scheduled events of a particular type, an asterisk (`*`) is placed as the edge attribute. Setting an edge attribute to an `*` might be useful, say, if you wish to temporarily shut down a portion of the simulation (e.g., a fire drill in a factory would cancel all job completion events).

NOTE: *Pass array elements explicitly; if* `Q`=3, *pass* `WAIT[3]` *and* `Q`, *not* `WAIT[Q]`

Passing edge attribute values to parameters is illustrated in the following examples.

5.3.1 Many Servers of Many Types: `SLOFAST2.MOD`

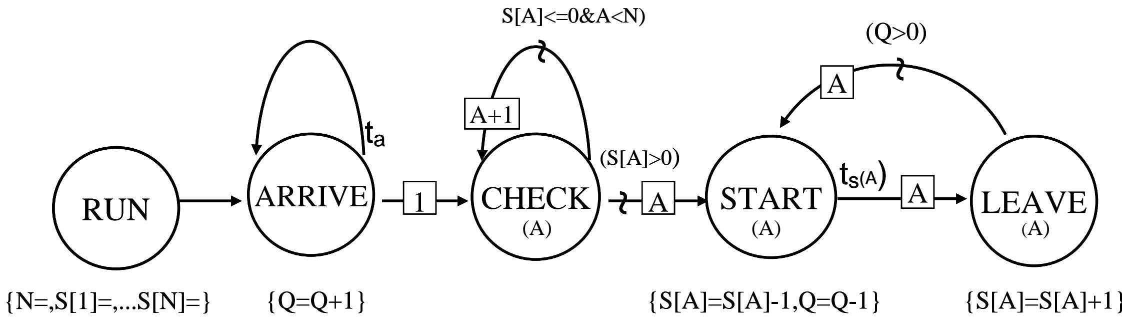
A generalization of the model with two types of servers (`SLOFAST0.MOD`) is a model of many types of servers with many servers of each type. Our example will be a production department with `N` different types of machines of different models and ages, indexed from 0 to N-1. There may be any number of each type of machine.

As before, we will designate the type of machine by its age, `A`, and let `S[A]` denote the number of idle machines of type `A`. If we merely added more event vertices for each type of machine, the model would become cumbersome when we simulate a system with many types of machines.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

This problem is easily solved using vertex parameters. For example, a START vertex with parameter A will denote the start of work with a machine of age A. A LEAVE vertex with parameter A will mean that a job has been finished by a machine of age A. When necessary, we will place vertex parameters in parentheses following the vertex name - e.g., START(A) and LEAVE(A). See Figure 5.13 for this model, which represents SLOFAST2.MOD. We have parameterized everything (state variables, edges, and vertices) that pertains to the type of server with the parameter A.

Figure 5.13: Many Servers of Many Different Types



This graph may look complicated when viewed as a whole. Perhaps the most confusing part of this model is the CHECK vertex; which checks each type of server in turn until either an idle server is found (S[A]>0) or all servers are checked (A>=N-1). These two conditions are the same as the compound condition (S[A]<=0&A<N-1) being false.

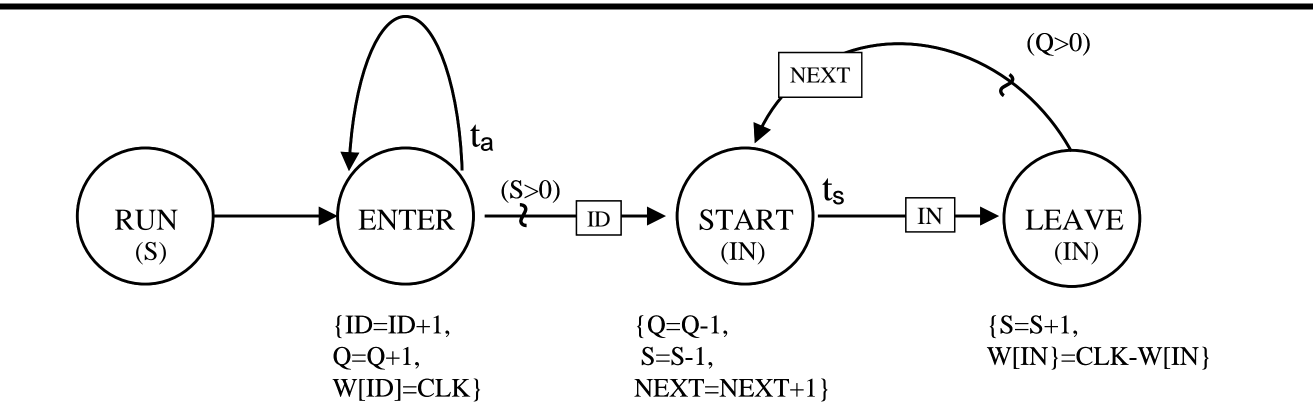
Notice that the graph structure for this model (vertices and edges) is similar to that of our original single server queue. We can think of these models as being in the same "family" of models. Using event parameters allows us to use simple graphs to model very large systems; we exploit the similarities of subgraphs by parameterizing them. We could use this model for very large systems with perhaps hundreds of servers of hundreds of different types and still be able to run the corresponding simulation program on a small personal computer.

5.3.2 Multiple Servers - Single Line with Waiting Times: BANK2.MOD

NOTE: This example is included to illustrate some subtle complexities in parameter passing. Modeling waiting times with general priorities for the different jobs waiting in line is easily done using the PUT and GET functions in SIGMA as explained in Chapter 7. While this example should NOT be used as a prototype for modeling customers in a queue, it is worth studying to help you develop effective simulation modeling skills. Modeling transient entities using parameters as in this example, while not as easy or general as using PUT and GET, does, however, result in a faster running simulation.

BANK2.MOD keeps statistics on the waiting times of individual customers in a multiple server queue (e.g., bank tellers). The event graph for this model is shown in Figure 5.14. This model is an enrichment of BANK1.MOD discussed earlier.

Figure 5.14: Multiple Server Queue with Customer Waiting Times



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The state variables for this model include:

QUEUE	The number of customers currently waiting in line.
SERVERS	The number of idle servers, initialized each time the model is run.

Additional variables that allow us to model the flow of transient entities are:

W[I]	An array of Waiting times for the Ith customer.
ID	The customer IDentification number.
IN	The identification number of a customer currently IN service.
NEXT	The customer NEXT at the front of the line.
IAT	Mean Interarrival Time
MST	Mean Service Time

Recall that CLK is the current simulated time, initially set at 0 and automatically updated at each event.Our graph now represents the flow of customers passing through the system. This model has the flavor of a process flow model, where the process being modeled is that of customers moving through a bank. This model has the same four vertices as the simpler resident entity model, BANK1.MOD, and their graph structures are very similar. The vertices, along with their state changes, are as follows.

The RUN vertex (the first vertex to be executed) is where we initialize the run parameters and start the run. The event parameters, SERVERS, IST, and MST, are input as initial conditions. The ENTER vertex models the arrival of new customers. The state changes for the ENTER event are as follows: ID=ID+1 - increments the customer ID counter; Q=Q+1 - increments the number in line; and W[ID]=CLK - marks the customer arrival time. The START vertex is the start of service for the customer with identification number = IN. The state changes are as follows: Q=Q–1-decrements the waiting line; S=S-1 - reduces the number of idle servers; and NEXT=NEXT+1 - updates the ID of the customer next in line. The event parameter, IN, is the customer going into service.

The LEAVE vertex models the end of service for the customer IN service. The state changes are as follows: S=S+1 - frees one server; and W[IN]=CLK–W[IN] – computes the customer waiting time. The event parameter, IN, is now the customer who was in service and is finishing.

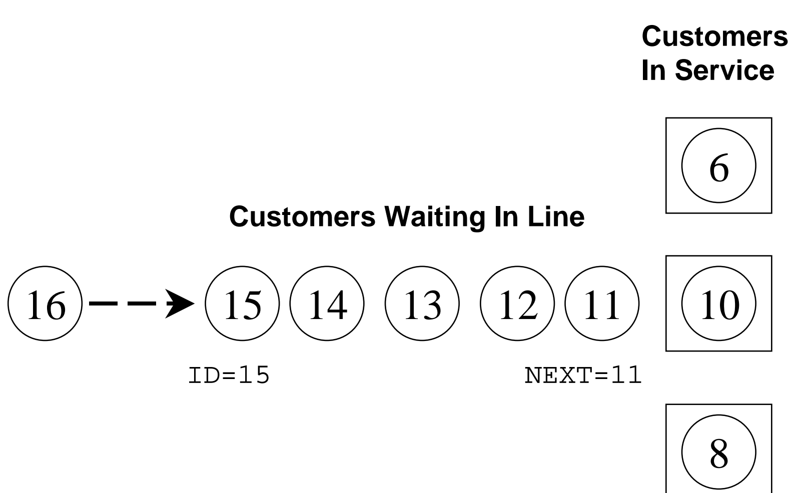
Here the situation is slightly more complex than in our previous models. However, it is still easy to read our event graph. The graph allows us to concentrate on one edge at a time. We can concisely describe the dynamics of the system with one sentence per edge. You should try to identify each sentence in the following system description with an edge in BANK2.MOD

The RUN begins when the first customer ENTERs the bank. Successive customers ENTER the system every t_a minutes. When a customer ENTERs, if there are idle tellers, then customer ID will START IN service. The same customer STARTing IN service will LEAVE after receiving service for t_s minutes. Whenever a customer IN service finally LEAVEs, if there are still other customers waiting, the NEXT customer in line can immediately START IN service.

To clarify these ideas, it is helpful to walk through an example with three servers. Consider this model at time 2.50 (CLK=2.50). Suppose that an ENTER event has just occurred. The entering customer found five customers waiting in line (Q=5) and no available servers (S=0). The customer at the front of the line is customer 11 (i.e., NEXT=11) and the last customer in the line is customer 15 (ID=15). The three servers are now serving customers 8, 6, and 10. The customer who just arrived would be customer 16 (ID+1) and Q would be increased to Q=6. Suppose the random time generated until the next arrival was 1.71. The next ENTER event would occur at time = 4.21 (2.5 + 1.71). The situation is illustrated in Figure 5.15

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.15: Numerical Example for `BANK2.MOD`



The future events list for this example is given in Table 1. It shows when the three customers currently in service will LEAVE the system and when the next customer will ENTER.

Table 5.1: Future Events List with Customers 6, 8, and 10

Time	Event	Priority	Attributes
3.243	LEAVE (customer 8)	6	8
4.210	ENTER (next arrival)	6	
5.593	LEAVE (customer 6)	6	6
6.478	LEAVE (customer 10)	6	10

At CLK=3.243 the LEAVE event occurs where customer 8 is the customer IN service (IN=8). At this LEAVE vertex, the total time in the system for customer 8 can be computed as the difference between the current value of the clock, CLK, and the time customer 8 arrived, W[IN]=W[8]. Since there are customers waiting in the queue (Q=6>0), a START event will be scheduled to occur without delay at time 3.243. The value of the vertex parameter, IN, which indicates which customer is starting "IN" service, will be the current value of NEXT (11) and will be passed by the edge from LEAVE to START. The START event will occur with parameter IN=11 and schedule a LEAVE event with customer 11 (say at time 8.243). The current value of IN is the attribute value for the parameter IN of the LEAVE event that has just been scheduled. Also, in the START event vertex, NEXT will be increased to 12 since customer 12 is now at the head of the line. The system and future events list now looks like Table 5.2. When the ENTER event occurs at CLK = 4.210, the identification number of the customer who is last in line is increased to ID=17.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Table 5.2: Future Events List After Customer 8 Leaves and 11 Starts Service

Time	Event	Priority	Attributes
4.210	ENTER (next arrival)	6	
5.593	LEAVE (customer 6)	6	6
6.478	LEAVE (customer 10)	6	10
8.243	LEAVE (customer 11)	6	11

We see that only the values of edge attributes are placed on the list of scheduled future events. All connection with the value of an attribute and the particular state variable or expression that was evaluated to get that attribute value is lost once the event vertex is scheduled.

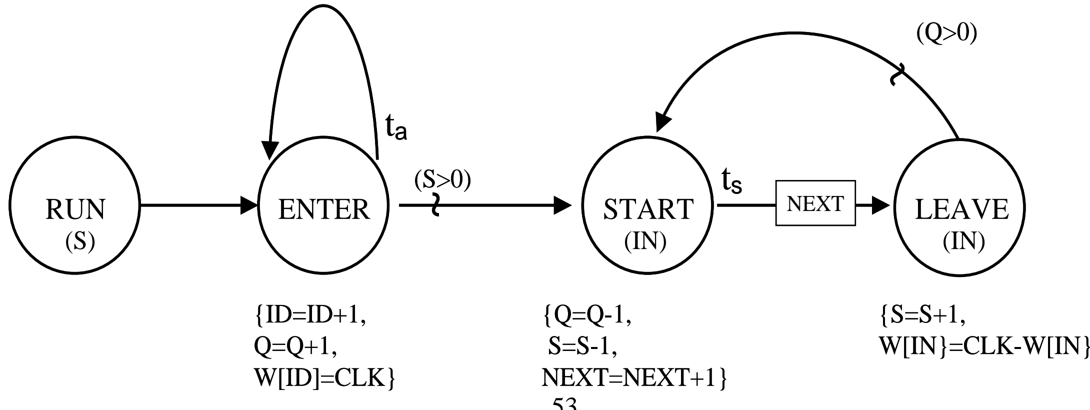
BANK2.MOD is a fairly complicated simulation. There are two SIGMA functions (called PUT and GET) that place transient entities (customers) in ranked lists (queues) and make this model very simple by eliminating the need for event parameters or edge attributes. Using these functions is a more straightforward way to model queues of transient entities with much greater ease and generality as explained in Chapter 7.

> *Only the **values** of edge attribute expressions are placed on the events list. These values are assigned to the list of vertex parameter* State Variables *when the event is later executed. Do not use array elements as edge attributes.*

Two models, BANK1.MOD and BANK2.MOD, illustrate how the same system can be modeled differently. The first example focuses only on the behavior of resident entities, those physical system components that remain a part of the system for long periods (e.g., servers in a queueing system). The second model enriches the first to include details on the flow of transient entities, those entities that occasionally enter and leave a system (e.g., customers in a queue). Each type of model has advantages: modeling resident entity cycles tends to be easy and efficient while modeling the details of transient entity flow gives more information.

The model, BANK2.MOD, actually has unnecessary event parameters. This was done in order to illustrate their use and perhaps make the model easier to understand. However, since the parameter IN is not used in the START event, it can be eliminated and an equivalent model is BANK3.MOD with only the START-LEAVE edge passing the attribute value of NEXT. Note that in BANK3.MOD the value of NEXT is not initialized to 1 but automatically started at zero. This model was simplified by assuming that customers will be served in the order in which they arrived (FIFO - First in first out). To

Figure 5.14- simplified (BANK3.MOD): Multiple Server FIFO Queue with Customer Waiting Times



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

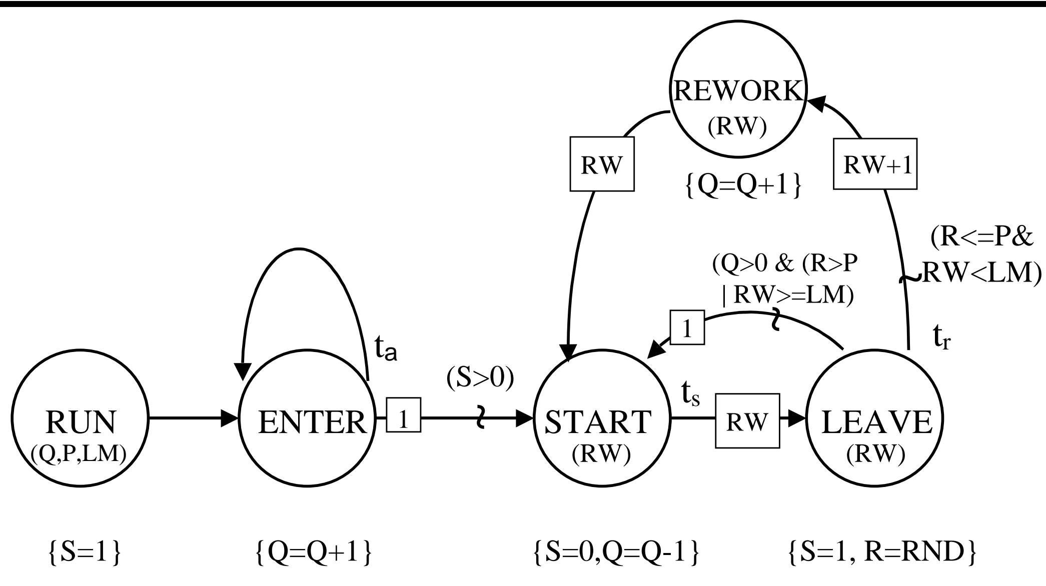
understand this: open two sessions of SIGMA at the same time and run BANK2.MOD in one and BANK3.MOD in the other and observe that their outputs are the same even though BANK3.MOD only has one edge passing the value an attribute, while BANK2.MOD has three such edges.

5.3.3 Limited Rework: `REWORK2.MOD`

Reworking does not always involve the removal of a coating that has just been applied; it sometimes involves the removal of original material. In a limited rework model, a particular part is not reworked more than a specified number of times before it is discarded.

In this model, we will assume that a part can be processed no more than the number of times specified by the limit, `LM`. Starting with the rework model (`REWORK1.MOD`), we can set up a counter, called `RW`, which is incremented each time rework is necessary on a part and reset to zero whenever a new part is started. This is easily done by making the `RW` counter an attribute of the `START` vertex. In the `REWORK` event, this counter is incremented from its current value by passing the edge attribute value of `RW+1` into the vertex parameter `RW` for the `REWORK` vertex. When a new part is started, a value of `1` is passed into the rework counter. This model is implemented in Figure 5.16. If an ENTER and a REWORK event happen to be scheduled simultaneously, it is important that the REWORK have priority to execute first, which will be followed immediately by a START event with the proper rework count, RW (because of the LIFO tie-breaking rule explained in Appendix A). Otherwise the rework count RW will be lost and a phantom server created as in Section 5.1.3.

Figure 5.16: Limited Rework of Jobs

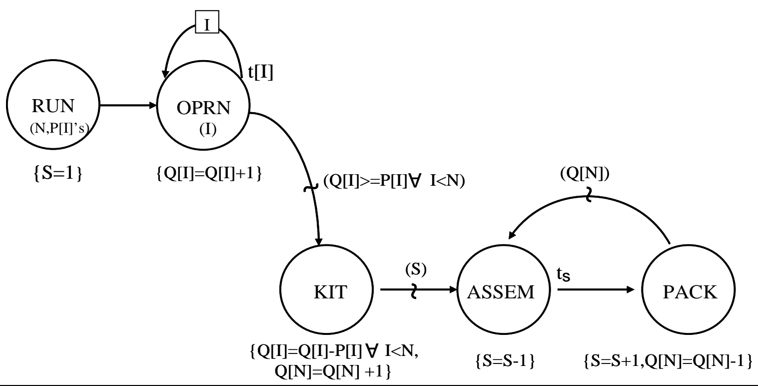


5.3.4 Generalized Assembly Operations: `ASSEMKIT.MOD`

Using vertex parameters, we also can generalize the assembly operation found in `ASSEMKIT.MOD`. We let `N` represent the number of different types of parts that go into the final assembly (see Figure 5.17). The number of parts needed from operation `I` will be given by the input variable `P[I]`. The graph for this model simply identifies the operation being finished in the `OPRN` event with the parameter `I`, denoting the type of part being produced. In this model, we use the mathematical notation $\forall$ to mean "for every"; thus the condition `(Q[I]>=P[I]`$\forall$`I<N)` is mathematical shorthand for "the number of finished parts in queue `I` is greater than the number of that type of part needed in the kit for every type of part."

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.17: Using Event Parameters in an Assembly Operation



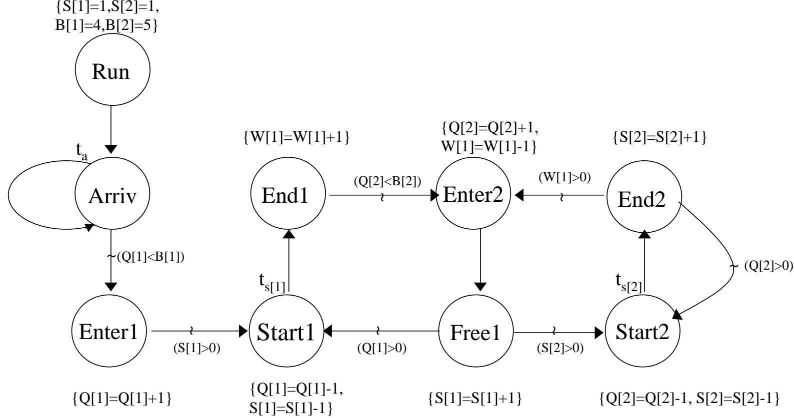
5.3.5 Sequential Service with Blocking: `TWOQUES.MOD, TWOQUES1.MOD, and TANDQ.MOD`

A queueing network where customers proceed through successive service operations along a single path is called a tandem queue; basically the queues are lined up one after another. An example of this might be a serial production line with several different machining steps. Typically, the service time for one of the steps is a bit slower than the others; this is called the bottleneck. Work will tend to pile up at the bottleneck created by the slower server, and successive servers will become starved for work. If there is limited waiting space between the servers (called a capacitated queueing system), blocking might also occur. A server becomes blocked when there is no space to unload a finished job. Both blocking and starving will degrade the performance of a tandem queue. Attempts to avoid both situations are major tasks to be faced when designing processing networks with limited waiting space.

We will start with the simple example of a capacitated tandem queueing network with just two types of servers. There is a limited amount of waiting space (buffer) for only `B[1]` jobs in front of the first (upstream) set of servers and waiting space for `B[2]` jobs in front of the second (downstream) set of servers. This is modeled in `TWOQUES.MOD`. We assume, as we did for our single capacitated queueing model, that customers who arrive when the first queue is full will simply leave.

The values of `B[1]` and `B[2]` will be input as parameters of the initial `RUN` vertex. We will use a single state variable, `S[I]`, to denote the number of idle servers of type `I`. We also will create another variable `W[I]` which is the number of servers of type `I` who are currently blocked and waiting for the downstream server to finish their current job so there will be a space to unload.

Figure 5.18: Tandem Queues with Buffers and Blocking



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The event graph for two queues in tandem, shown in Figure 5.18, looks a bit complicated but it is rather simple—again, read only one vertex and its exiting edges at a time and let the graph take care of keeping it all connected. Try to associate each sentence in the following description of the model's dynamics with a one or more edges in the event graph in Figure 5.18 - read a sentence, then look at the graph before reading the next sentence making sure that you understand the logic.

> The simulation Run is begins with the first customer Arrival. A new job Arrives every t_a minutes. If the Arriving job finds space in front of queue 1, it will Enter queue 1. Jobs Entering queue 1 can Start if there are idle servers. After Starting on process 1, a job will End after a delay of $t_{S[1]}$ minutes.
>
> After a job Ends process 1, if it then finds space in front of queue 2 it will Enter queue 2 and immediately Free its server for process 1. The newly Freed server for process 1 will Start work if there are more jobs waiting in Queue 1.
>
> The job Entering queue 2 can Start work at queue 2 if there is an idle server. Jobs that Start in process 2 with End processing $t_{S[2]}$ minutes later. When a job Ends processing at queue 2, it will Start the next waiting job, if any, in queue 2 making a waiting space in its buffer to Free any blocked servers for queue 1.

A more complex technique for modeling networks of queues with limited waiting space is to pass event parameters indicating whether or not a resource is blocked. This is illustrated in the model TWOQUES1.MOD.

We can generalize the tandem queue model to include any number of queues in series by having an additional vertex parameter that indicates which queue is involved in each event. For example, the ENTER1 and ENTER2 vertices can become a common ENTER vertex with the parameter, I, indicating which queue is being entered. This general tandem queue is modeled as TANDQ.MOD.

5.4 Multiple Resources

Sometimes several resources must be available for an event to happen. For example, in a factory it may be necessary that both a worker and a machine be idle before a part can be processed. Assigning 0 to the "not-available state" makes multiple resource constraints very simple to model. If several types of resources are needed to schedule an event, simply list the names of these resources on the scheduling edge in your SIGMA graph. If WORKERS is the number of idle workers, PARTS the number of parts available, and MACHINES the number of idle machines, then the edge condition, WORKERS & PARTS & MACHINES, is sufficient to check if all three resources are simultaneously available (assuming, of course, that there are no codes for states of the workers and machines that are negative numbers).

The reason this works is that in SIGMA (like C) zero indicates a condition is false and *any* non-zero value indicates it is true. Thus, the implicit condition (WORKERS & PARTS & MACHINES) is equivalent to the explicit condition (WORKERS>0 & PARTS>0 & MACHINES>0) - as long as there is no logic error that makes the number of workers, parts, or machines become negative! In fact, the implicit test is preferred to the explicit one in that it *allows* variables to go negative which turns out to be a great help in detecting errors in logic.

Our next model enrichment is a conceptual prototype for models where several resources are required to do a task. Here the machines are semi-automatic—they only need to be loaded and unloaded by workers. Once loaded, processing is automatic. Since one of the prime motivations for automation is to reduce labor requirements, we will assume that there are fewer workers than machines. We will add the event vertices, STRTL (start loading) and ENDL (end loading), to model the worker loading a job onto a machine and the vertices, STRTU (start unloading) and ENDU (end unloading), to model the worker unloading a machine. It is necessary that there be both an idle machine and an idle worker to start loading a job onto a machine.

5.5 A Problem in Finance: BONDRATE.MOD

So far we have focused on models of queueing systems that are applicable to such areas as manufacturing, health care, and communications. The applications of SIGMA are by no means limited to these systems. Here we will use SIGMA to analyze a problem in finance: determining fair pricing for coupon bonds in the presence of randomly fluctuating interest rates. The simple model we will develop can serve as a basis for more complicated cash stream analysis; indeed, this model can be enriched to study financial risk in general cash flow scenarios.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

In determining a fair price for a coupon bond (or any other investment), it is important to consider alternatives. For our example, we consider the decision faced by an investor who is offered shares in a variable rate mutual fund with a guaranteed minimum interest of 8%. Interest returns on this kind of investment are expected to remain about 15% over the next 5 years, but are highly variable. A more conservative option is being offered by an investment banker: a 5-year coupon bond that pays $10,000 at the end of each year with a $100,000 value upon maturity. How much should our investor be willing to pay for the coupon bond?

Discount factors are used to account for interest. With an interest rate of `I` for a particular year, cash is worth `(1+I)` as much at the end of the year as at the first of the year. Equivalently, cash at the end of the year is worth `1/(1+I)` as much at the beginning of the year; the factor, `R=1/(1+I)`, is called the discount factor for that year. Therefore, multiplying a cash payment at the end of the year by `R` gives its present value.

In addressing this problem, we will assume that yearly interest rates for the mutual fund can be modeled using a normal probability distribution with a mean of 15% and standard deviation of ±5%. Our simulation can be used to simulate thousands of replications of the next five years. For each replication we can determine the fair price for the bond. If the bond is offered at less than this price, the investor would have been better off buying it. In a few seconds, the investor can use our simulation to gain 5000 years of experience before making a decision. The simulation for this SIGMA is `BONDRATE.MOD`, shown in Figure 5.19.

Figure 5.19: Event Graph for Pricing of a Coupon Bond

Our model has the following input variables.

`LIFE`	Life of the bond until maturity
`INT`	Average annual interest rate over the life of the bond
`STD`	Standard deviation of the interest rate
`MINR`	Minimum interest from alternative investment
`PAYMENT`	The coupon bond yearly cash payment
`VALUE`	Face value of bond at maturity

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The following intermediate variables are used in our calculations.

N	Index for year during planning horizon
I	Normally distributed random future annual interest rate
R	Cumulative discount rate
NREP	Number of replications you wish to run
REP	Index for which replication currently is being run
CFLAG	Flag indicating if flow diagram desired, 1/0=yes/no)

The model computes the following for each year of the investment life.

CASH	End of year cash flow (only for cash flow diagrams)
PRICE	Fair price of bond on maturity
NPV	Net present value of cash flow (before taxes)

The dynamics of this model are simple; all of the action takes place in a single vertex loop where each year's interest rate is generated. Then the net cash flow for the year is discounted back to the present and added to the net present value of the investment. Initially, the cumulative discount factor is set to `R=1`. Next, a loop with year, `N`, going from `1` to `LIFE` is executed. For every year, the computations are the following.

`I=NOR{INT;STD}`	Generate the year's variable interest rate
`I=MAX{I;MINR}`	Guarantee the minimum interest rate
`R=R*(1/(1+I))`	Compute cumulative discount factor for the year
`CASH=PAYMT`	Place payment in the cash flow stream
`NPV=NPV+CASH*R`	Roll cash back to the present

In the last year when the bond matures, the face `VALUE` for the bond is collected. This `VALUE` is rolled back to the present and added to the net present value of the investment in exactly the same manner as for each annual cash payment.

`NPV=NPV+VALUE*R`	Roll cash back to the present

The result is the fair price of the bond, which is the net present value of the same amount of cash invested in the variable rate mutual fund for the simulated five-year period.

Of course, many more scenarios should be explored before funds are committed. These experiments should consider alternative models for the future interest rates. The model structure would remain essentially unchanged. Such experimentation only takes a few seconds.

Starting with this model, it is possible to build a very sophisticated, *customized,* financial risk model. In fact, a New York investment bank uses a SIGMA model to explore the pricing of "cocktail" bonds, which can be cashed in one of several foreign currencies.

5.6 Project Management (PERT/CPM): `PERT.MOD`

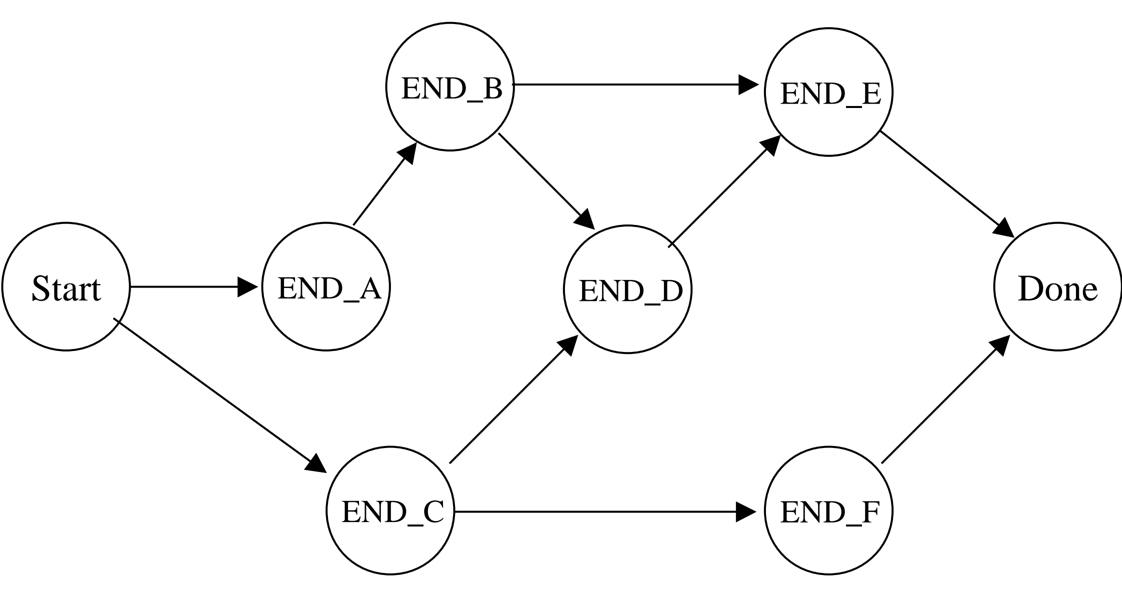
In a large project, such as the construction of a new building or the development of a major software package, there are typically several activities that may proceed in parallel as well as some precedence constraints among the different activities. When erecting a building, it might be possible to have the windows installed at the same time as the plumbing and electrical wiring; if so, these three activities can proceed in parallel. On the other hand, there might also be precedence requirements among activities: the scaffold must be finished and secured before starting the outside facing; the frame must be finished before starting work on the roof, etc. When planning or bidding for a large project it is important to know as much as you can about how long it might take to finish before resources are committed.

A popular technique for managing large projects is to study activity precedence networks, which use graphs showing the activities that must be completed before others can start (see Figure 5.20). The end of each activity is

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

represented by a node and the duration of the activities by the edges. In Figure 5.20, activity B must be finished before activities D and E can be started; therefore, the node, END_B, also represents the simultaneous starting of activities E and D.

Figure 5.20: Activity Precedence Network for a Project in PERT.MOD



The most common approaches to the static analysis of these networks are called PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method). These two similar techniques are discussed in most textbooks on management or industrial engineering. Traditional CPM and PERT assume that the activity durations are either known or, if random, independent of each other. Clearly, in a real project the activity times are random as well as dependent (e.g., bad weather, labor disputes, or competition for resources generally will delay more than one activity). Using simulation, we can "dry-run" a project hundreds or thousands of times to learn what might happen before we submit a bid or commit to a contract.

We will illustrate this with a simple example. Suppose we are planning a project that has the activity precedences given in Table 5.3 and graphed in Figure 5.20.

Table 5.3: Activity Precedence for a Project

Activity	Must Follow Activities	COUNTER[I]
A	None	0
B	A	1
C	None	0
D	B and C	2
E	B and D	2
F	C	1

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The event graph on your SIGMA screen for this project will look exactly like the activity precedence network in Figure 5.20. Associated with each vertex, `I`, is a counter of *unfinished precedent activities*, initialized to equal the number of incoming edges for the vertex.. We will call this activity counter, `N[I]`. The counter for the project `START` vertex is always `0`, and the counter for the `DONE` vertex in this example is `2` The state change for each vertex is simply to decrement its counter by one, `N[I]=N[I]-1`. All edges have the condition that the counter for the originating vertex has been reduced to zero `(N[I]==0)` indicating that there are no more unfinished precedent activities. The delay times are simply the activity times associated with activities that end at the destination vertices; they can have any distributions you want (e.g., they might be dependent on a weather pattern you generate).

5.7 Modeling Transient Entities

Most of our models so far have dealt primarily with resident entities: the servers and the waiting lines. Suppose that we want to know the average waiting time in line for each customer. The easy way to do this in SIGMA is to use the `PUT` and `GET` functions in Chapter 7. If we modeled each customer, their average waiting times would be easy to compute (as we did with `BANK2.MOD`). However, modeling transient entities in a queueing simulation may require more computer memory and significantly greater processing time. Furthermore, as the simulation becomes more congested, it will become less efficient. It is not uncommon for transient-entity-oriented network simulation languages to become so congested that they abort a run. When the run aborts, there is often no useful information given about that run, possibly just the message that the run has terminated with an error! Thus, the most interesting runs of the simulation (when the system was the busiest) and the most expensive (when the most customers were simulated) are exactly those runs that are aborted. Not observing the simulated system when it is congested can lead to an overly optimistic evaluation of system performance. Pure resident entity models are considerably more efficient; furthermore, this efficiency does not degrade as the system becomes congested.

5.7.1 Little's Law

A situation where we do not need transient entities is when we are only interested in aggregate information about the transient entities, such as the average customer waiting time. Suppose that we want to know the average time spent by our customers waiting for service during a simulation run. We will denote this average waiting time as `W`. To observe customer waiting times, we might enrich (and complicate) our simulation by including each customer in our model. Alternatively, we could leave our model like it is and use the idea behind "Little's Law" from queueing theory to indirectly estimate `W` without adding transient entities to our model. We will illustrate this with a straightforward dimensional analysis.

Say we have `N` customers in a resident entity simulation run that is `T` simulated hours long. Denote the observed average queue length as `L` customers. The average number of customers waiting in line during the run multiplied by the amount of simulated time gives us the total customer-hours spent waiting by all of the customers during the run. If `C` is the total simulated customer-hours, then

`C = L*T` (in customer-hours).

If we divide this total customer-hours by the total number of customers simulated, `N`, we get an estimate of the average time each customer spent waiting;

`W = C/N = (L*T)/N.`

Therefore, by keeping track of the average queue length in a pure resident entity simulation, we can estimate the average waiting time of the transient entities.

Although this is a popular way to estimate average waiting times in queues and it is used by some commercial simulation languages, it is not quite as accurate as directly collecting the waiting times of each customer. This is because some customers might be in line when the run ends, causing different numbers of customers to contribute to the numerator and denominator of the ratio `C/N` used to estimate `W`. Each customer waiting in line contributes to the total customer-hours waiting, `C`, continuously as they wait; however, the total cumulative customer count, `N`, is only incremented at discrete times. If we count customers when they leave, we might overestimate `W` since customers who have made partial contributions to the total customer-hour waiting time have not yet been counted. If we count customers when they enter the system, we would tend to underestimate `W` since customers who remain in line when the run ends will not make their

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

full contribution to the total customer-hours until they leave. We might adjust for this problem by generating the unfinished backlog of customer-hours of work for the server at the end of the run and add it to `C` or, equivalently, by cutting off the arrivals and letting the system empty as in the example in `CLOSEIT.MOD`.

5.7.2 Generalizing the Notion behind Little's Law (`DELAY.MOD`)

What if we are interested in more than just the average waiting times of jobs in our system? Suppose we also would like to estimate the probability that a job waits less that a specified time—that is, we are interested in estimating the cumulative probability distribution function. We can take a different approach to Little's law based on event counts.

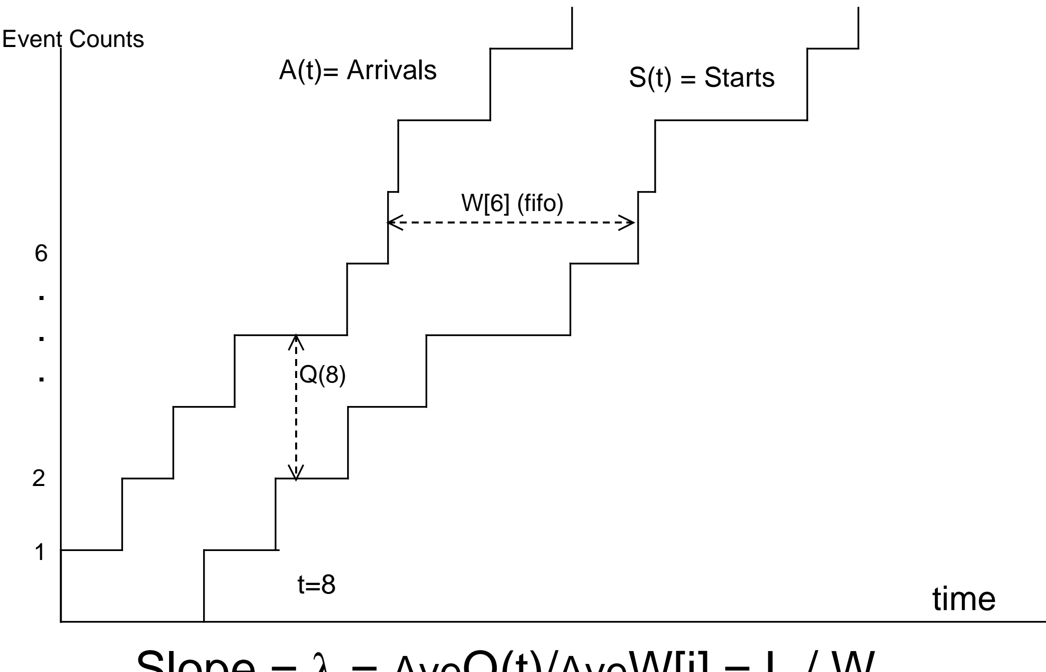
In Figure 5.21 we have plotted the count of the `Arrival` events, A(t), and the count of the `Start` events, S(t), in a simulated queue. The long-term average slope for A(t) plot is the number of `Arrival` events per minute or the job arrival rate, λ. Since the queue size neither grows infinite (then A(t) and S(t) would diverge) or negative (then A(t) and S(t) cross) in a stable queue, the long term average slope of these two plots must be equal.

We can estimate the common slope of A(t) and S(t) easily. The number of jobs waiting in the queue at any time t, Q(t), can be computed as the total number of job arrivals minus the number of jobs that have started service, Q(t)=A(t)-S(t) - the *vertical* distance between the two event counts. Also at the nth `Start` event the waiting time of the nth job can be computed as the *horizontal* distance between the S(t) and A(t). We estimate the slope of A(t) and S(t) in the usual manner, the average vertical distance (L) divided by the average horizontal distance (W). We therefore estimate the arrival rate (known slope), λ, with the ratio L/W. If the estimate is unbiased then in the long run we get $L = \lambda W$ or Little's law.

Notice that it is easy to estimate the length of the queue since we need only the event counts at a particular time - we don't have to store any information.

To compute the waiting times we need the horizontal (along the *time* axis) distances - we must store information.

Figure 5.21: Using Event Counts to infer Little's Law]



> *Computing with quantities on a vertical cut (at the same time) in the* `Event Counts` *plot does not require information storage.*

Now suppose that we wish to compute the probability distribution,

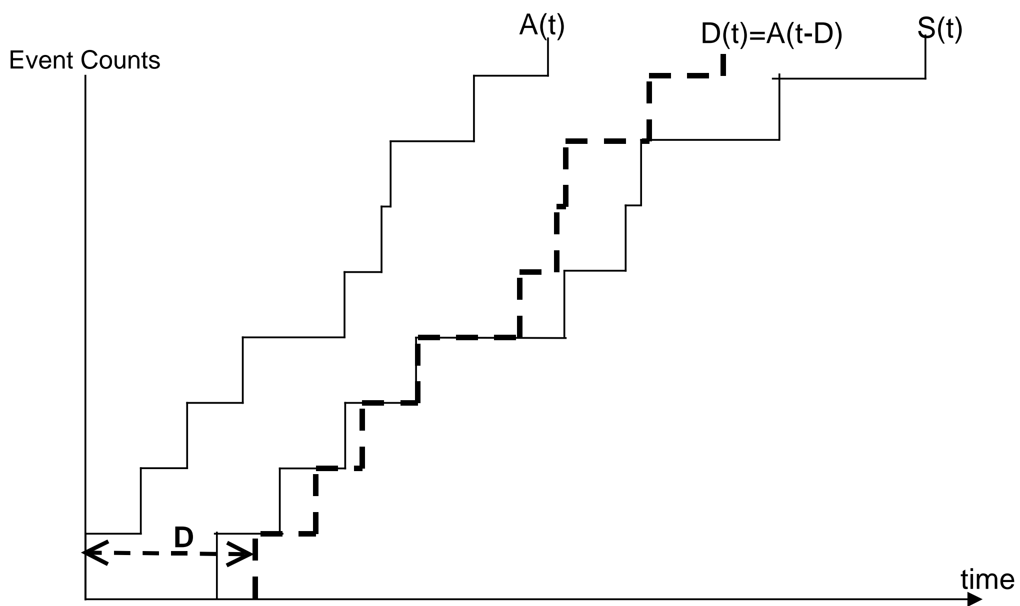
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

$$F_W(\tau) = \text{Prob}\{W \leq \tau\}$$

where W is the average time a job waits in the queue. We can easily do this without storing any information. This is done by adding a `Delay` event that is scheduled to occur τ time units after each `Arrival` event. The `Delay` event simply tells us that a job arrived τ minutes ago. If service has not started by that time, then the customer has waited in queue longer than τ. We keep a count of the `Delay` events that occur by time t, D(t). Whenever a `Start` event occurs and D(t)>S(t), then the job waited in line longer than τ. Let *W* be the total number of `Start` events where D(t)>S(t). Our estimator of $F_W(\tau)$ is simply *W*/S(t) which as t gets large will converge to the correct probability.

To understand how this works, study the model `DELAY.MOD` or draw a hypothetical D(t) plot, like that in Figure 5.22. D(t) will always be below A(t)-a job obviously cannot experience a delay in the queue until *after* it has arrived. D(t) will cross S(t) horizontally when a service starts before τ and vertically when a job is delayed longer than τ.

Figure 5.22: Adding the Delay Event Count to Estimate the Waiting Distribution]



D(t)>S(t) ⇔ Delay<D ⇒ Count to est. Prob{Wait<D} = CDF

5.8 Programming with Event Graphs

An event graph can be used as a general flow chart for designing computer programs that have nothing to do with simulation models. If all the edge delay times are zero, each vertex would be a block of assignment statements; edge conditions would show the logic flow. All branching is represented by edges in the graph, including `goto`'s, `if-then-else`, and function calls. The values of edge attributes passed to the vertex parameter variables act like values passed to functions, as in a C program. The main difference, and it is an important difference, between event graphs and traditional flow charts is branching. In SIGMA more than one branch might be taken, or perhaps none of a set of possible branches might be followed.

> *Recall that when edge conditions are tested they are considered false only if they are equal to zero. Expressions for edge conditions that are non-zero are always considered true.*

5.8.1 Boolean Variables

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Although condition testing is done on the edges of an event graph, it is possible to evaluate conditional expressions within a vertex by using Boolean variables. Boolean variables take the numerical "value" of a conditional expression such as (X>Y). The rule is simple: If the condition is true, the value of the Boolean variable is 1; if the condition is false, the value of the Boolean variable is 0. Embedding a logic condition in a state change means that the number 1 is substituted for the condition if the condition is true and a 0 is substituted for the condition if it is false.

For example, you might want to increment the length of a waiting QUEUE if and only if there are no idle SERVERS available. This can be done with the single statement

```
QUEUE=QUEUE+(SERVERS==0)
```

which is the same as setting QUEUE=QUEUE+1 if (SERVERS==0) and leaving QUEUE otherwise unchanged. This statement would model the queue increasing only if there are no servers available to a newly arrived customer. As another example, examine the expressions

```
R=RND,
Q=54+(R>.5)*3+(R>.7).
```

Note that here only one random number, RND, is generated. In SIGMA (and in C), this will have the following effect:

Q = 54 with a probability of 0.5 (R<=.5)
Q = 57 with a probability of 0.2 (.5<R<=.7)
Q = 58 with a probability of 0.3 (R>.7).

Another example would be to use three successively generated random numbers, X=(RND<.5)+2*(RND<.5)+4*(RND<.5), to generate a value of X uniformly over the integers 0 to 7. There are efficient ways to generate a sample from a simple discrete probability distribution.

A common use of Boolean variables in event graph modeling is in expressing alternatives, as in the "if-then-else" logic structure explained next.

5.8.2 Conditional Expressions (If-Then-Else)

There are several ways to write conditional expressions. Consider the following "if-then-else" sequence in C,

```
if (X>Y)  Z=W;
else Z=Q;
```

In SIGMA, one would either define an edge conditioned on X>Y as in Figure 5.23 or use the expression with embedded Boolean variables,

```
Z=W*(X>Y)+Q*(X<=Y)
```

in a vertex. If X>Y, the above expression will be equivalent to

```
Z=W*(1)+Q*(0)=W,
```

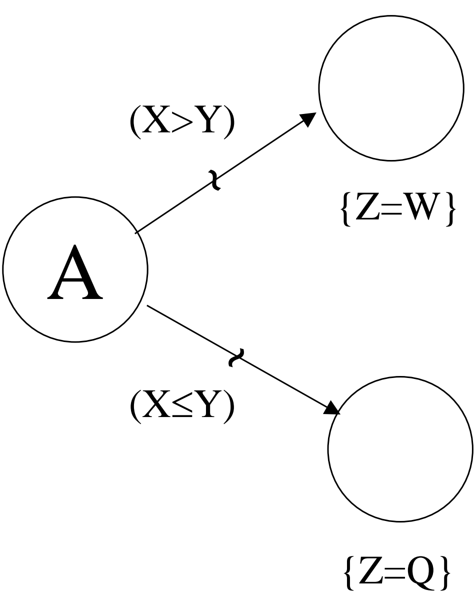
and if X≤Y, it would become

Z=W*(0)+Q*(1)=Q

as desired. Using embedded Boolean variables here is more efficient.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.23: `If-Then-Else` Graph Structure



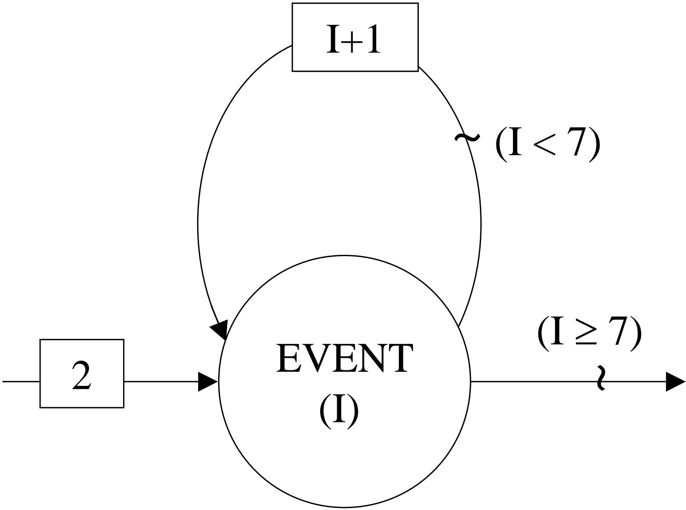
5.8.3 Do, While, and Nested Loops

"Do", "while", and "for" loops are easy to implement in event graphs. In fact, loops of code are naturally pictured in the graphs as loops. Figure 5.24 shows how the expressions in a block of code called `EVENT` can be executed with a parameter, `I`, that goes from a value of `I=2` to a value of `I=7`.

It is worth going through the logic in Figure 5.24 in detail. We want the state changes in `EVENT` to be executed for the last time when `I=7`. This will happen here because edge attribute expressions are evaluated only after edge conditions are tested as being true. When `I=6`, the condition on the loop, `(I<7)`, is true; therefore, the value of the attribute expression, `7=I+1`, is then passed to `EVENT` parameter, `I`. This is done by scheduling `EVENT` on the list of future events with an attribute of 7. When `EVENT` occurs the next time, the first thing that happens is its parameter, `I`, is set to 7. Unless the state changes in `EVENT` reduce the value of `I` (which would probably be a programming error), the condition `(I<7)` is now false and the loop is not scheduled.

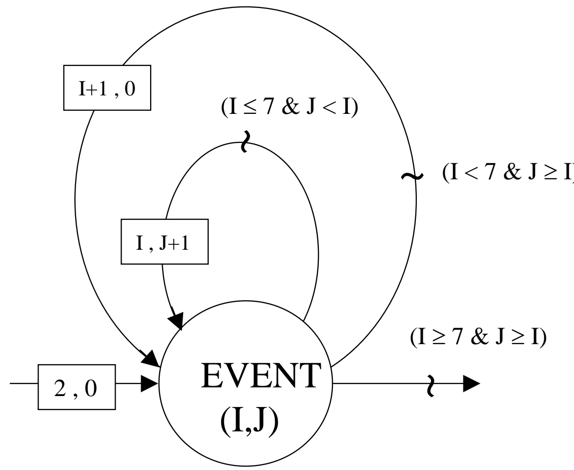
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.24: Do Event For `I=2 to 7]`



Nested loops are only slightly more complicated. In Figure 5.25 there are two loops, an outer loop indexed with the variable, `I`, and an inner loop indexed with `J`. The values of `I` range from the constants `2` to `7` as before; however, the values of `J` will range from `0` up to the current value of `I`.

Figure 5.25: Nested Loops: Do for I=2 to 7 and J=0 to I



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

5.9 Model Complexity and Model Size

Although some of the models discussed here may appear complicated at first, keep in mind that a large model is not necessarily a complex model. When building simulations, it is important to distinguish between a model that is inherently complicated and a model that is large but otherwise simple. Complexity occurs when a model is composed of many different *types* of events; a large simulation might be made up of many events but only a few different types of events.

A large simple model might be built from many copies of a few, essentially identical, subgraphs. It is the commonalities of these subgraphs that keep the model simple. Indeed, special-purpose-simulators are developed by focusing on the commonalities of the elementary components of a particular type of system. For instance, a typical commercial factory simulation package will have a generic subgraph to represent a machine as its basic building block. A crude model of a factory is built by putting together many machine subgraphs, each of which is distinguished only by different parameter values. The parameters for each machine (processing time, distribution of the time between failures, repair time distributions, etc.) can be selected from simple menus. The fundamental differences between simple simulators like that just described and block-oriented, general-purpose simulation languages are the number and the richness of the building blocks available with the language; however, both are limited to a finite set of preprogrammed modules. If you want to simulate a situation not already anticipated by the software vendor, you are out of luck.

Using event graphs, we can construct a simulator of a large scale queueing network by first developing an event graph of the basic building block for the network, a single service center. Using this approach, we can develop a simulation of an arbitrarily large queueing network of service operations with several different types of customers. Each customer will follow a different path through the service network and will have different sets of service times**.** (See the SIGMA `NETWORK` model `NETWORK.MOD` in Chapter 7 where the PUT and GET functions are discussed.) Starting with a simple queueing system model like those presented in this chapter, we will quickly build a simulator of an entire queueing network.

Once we are satisfied with the richness of the event graph of a basic building block, we can then use edge attributes and vertex parameters to make as many distinct copies of this subgraph as we want (the *picture* of our graph will not change). The complexity of our model will be determined by the detail we put into the basic building block; the size of our model is determined only by the number of parameterized copies of this basic event graph that our model generates during a run. It is useful to visualize the basic subgraph of a simulation model as a simulation "molecule" from which a larger model can be developed, much like a "crystal" made up of many molecules.

5.10 Continuous Time Simulations: `FISHTANK.MOD`

This book is concerned mainly with simulating discrete event systems. However, there is another class of systems where the values of state variables are modeled as changing continuously. Simulations of such systems are referred to as continuous time simulations. Continuous time systems can also be simulated using event graphs and the SIGMA software; an example is presented in this section.

While discrete event models typically are used to study systems that are on a "human scale," continuous time simulations are popular for very large scale and very small scale systems. On a small scale, continuous time simulations are used in the sciences and engineering to study such things as the stress on a robot arm or the dynamics of a collection of molecules. On a large scale, this type of simulation is used to study interactions in social, astronomical, or economic systems.

Discrete event simulations have the reputation of being considerably more complicated and *ad-hoc* than continuous simulations. This is largely due to the fact that continuous time systems have an explicit mathematical representation in the form of systems of differential and partial differential equations. Differential equations model the instantaneous rates of change for the state variables. These rates can depend both on time as well as the values of the state variables. In the simplest case, we might have a system with only one variable. The value of this variable at time, t, is denoted $X(t)$. The instantaneous rate of change in this variable - i.e., its time derivative, $dX(t)/dt$ - might be given as a (perhaps random) function, $f(.)$, of state and time,

$$dX(t)/dt = f(X(t),t).$$

Knowing the state value at time zero, $X(0)$, we can simulate future values of $X(t)$. On the computer, this is done recursively using a difference equation to approximate the change in state over a small time step. For a step size of Δt time units, our system can be modeled as

$$X(t+\Delta t) = X(t) + f(X(t),t)*\Delta t.$$

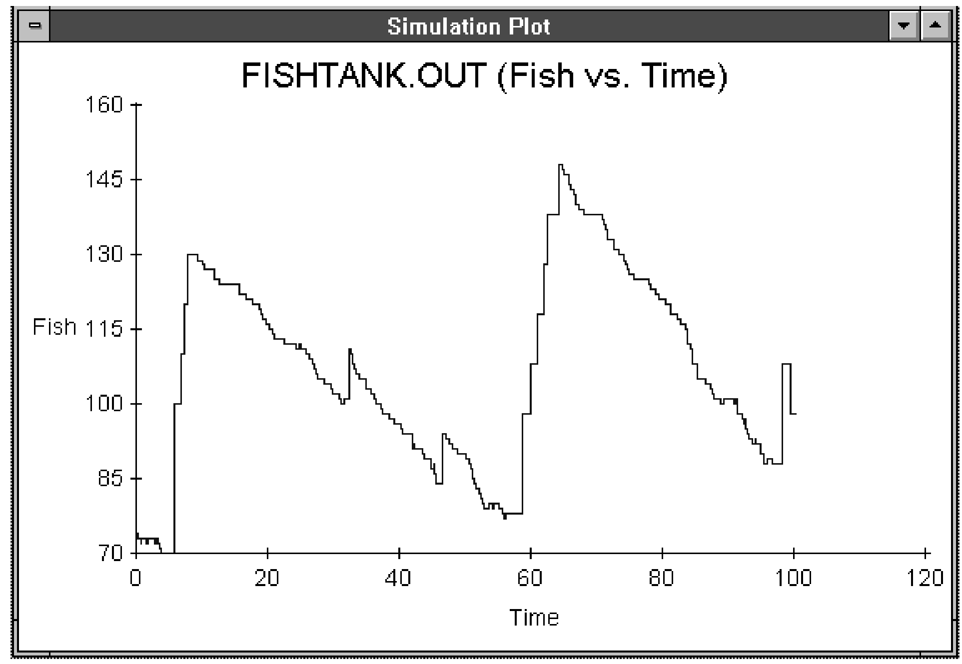
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Typically a continuous time simulation model will involve several state variables, and these will be coupled. We need to know the values of the state variables before we
can compute the changes, and we need to know the changes before we can compute the values. Therefore, the system of equations must be solved simultaneously. Approximate methods can be used to perform the integration to simulate the state variable trajectory. Consideration of such things as error accumulation requires carefully designed algorithms. There is a clear trade-off; a simulation with a smaller time-step will have smaller errors but will also take longer to run. Discussions of integration algorithms can be found in any number of texts on continuous time simulation.

A realistic but simple example of a continuous time simulation is a model of a large fish breeding aquarium by Jean-Didier Opsomer. This model is `FISHTANK.MOD`. The population dynamics of a tropical fish called a *discus* is modeled. The simplified food chain in this ecosystem model consists of phytoplankton (algae), zooplankton, daphnia, and small fish. Several fish may be purchased and introduced into the system at once; they are typically sold one at a time. Each population in the food chain is modeled with a linear feeding and piece-wise linear growth rate that depends on the relative sizes of each population. In addition to feeding rates, a rate-limiting nutrient, phosphate, is used. Termination events for this system are eutrophication (explosive algae growth) and extinction.

Using this model, the effects of fish introduction, feeding rate, and phosphate introduction are studied with the objective of determining appropriate target populations for the plankton and daphnia. Figure 5.26 shows how the fish population changes over time in this model. Like most ecosystem models, the population has a clear periodicity; the management goal is to keep the fluctuations in the different populations under control (and the fish population large) so that one of the termination events does not occur.

Figure 5.26: A Continuous Time Simulation of a Fish Aquarium, `FISHTANK.MOD`.



5.11 Process Modeling

Many popular simulation packages are based on what is known as a *process worldview*. (The process approach is also sometimes called network modeling.) With this modeling philosophy, the focus is typically on transient entities, such as customers in a service system or parts flowing through a factory. Steps followed by these transient entities as they move through a system are identified. A simulation model is developed by placing preprogrammed blocks of code in a sequence that approximates the actual processing sequence being modeled. The processing of a transient entity is conceptually viewed as the entity flowing from block to block. The process modeling viewpoint is possibly the purest form of transient entity modeling.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Process-oriented simulation languages provide a series of preprogrammed blocks that represent some of the typical processing stages in a queueing system. Examples of such blocks include a block to represent the entity waiting (i.e., a QUEUE block) and another to represent the start of service (i.e., a SEIZE block). Popular software packages that include a process-oriented feature include GPSS, Automod, ARENA, Promodel, SIMAN, Simeo, and SLAM.

Although each of the popular process modeling software packages is similar in function, each has its own terminology. For example, GPSS provides a GENERATE block, which introduces a transient entity into the system; ARENA calls this a CREATE block. Time delay is modeled using an ADVANCE block in GPSS and a DELAY block in SIMAN. GPSS and SIMAN both have QUEUE blocks to collect waiting time statistics, and both represent the entity waiting for service using SEIZE and RELEASE blocks. (Separate ENTER and LEAVE blocks are used in GPSS to model several identical parallel servers, which they call a "storage facility.") Time delays are modeled on the network arrows in SLAM, similar to the edge delay times used in event graphs.

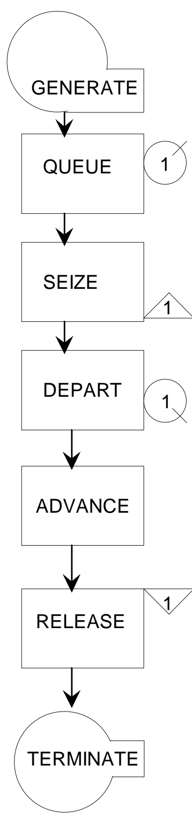
5.11.1 GPSS

A schematic diagram of the blocks in a GPSS model that represents a single server queue is shown in Figure 5.27. Here the customer enters the system in the GENERATE block and joins QUEUE number 1. If server (number 1) is idle, it is SEIZEd and the customer DEPARTs from the line. After the clock ADVANCEs for the service time, the server is RELEASEd and the customer exits the system at the TERMINATE block.

Event graphs, while much simpler than most process languages to learn and use, are more general. In fact, you can build versions of these process languages using SIGMA. See GPSS.MOD for an event graph of the GPSS single server queue pictured in Figure 5.27. This model uses the SIGMA PUT and GET functions, described in Chapter 7, for queue management.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 5.27: Block Diagram for a GPSS Model of a Single Server Queue



5.11.2 SIMAN.MOD `SIMAN1.MOD and SIMAN2.MOD`

SIGMA can be used to build a dynamic Windows subset of the manufacturing-specific simulation language, SIMAN (a product of Systems Modeling Co.). Two queuing models have been included here. `SIMAN1.MOD` and `SIMAN2.MOD`. The first is a basic model containing a subset of SIMAN blocks, where each block is a group of animated event vertices. `SIMAN2.MOD` was built by copying and pasting blocks from `SIMAN1.MOD`. The data for `SIMAN2.MOD` are in the text file, `DISTNS.DAT`. (See Chapter 7 to learn how to create data files that will be read by the `DISK{}` function.)

However unlike in SIMAN, you can change the probability distributions of a model *while it is running*. With `SIMAN2.MOD` on the screen, open the data file `DISTNS.DAT` (using File/View Output/Text File). Use the Window/Tile command to see both the simulation and the data file. Start the simulation and let it run in Graphics mode while you look at the output plot. Enter a new valid SIGMA expression, such as `3+BET{.5;.5}` for a Beta distribution (using capital letters), in the data file while the model is running. As soon as `DISTNS.DAT` is saved, the new data will be incorporated into the model and the resulting changes will appear in the output plot.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

FIGURE 5.28: SIGMA Simulation of a SIMAN Model.

Proponents of the process approach argue that it is natural to focus on the movement of transient entities since their motion "catches the eye." Once the definition of each of the preprogrammed blocks in the software package is understood, it is a simple manner to string them together into a simulated process. This is certainly one of the more easy to use modeling approaches for the beginner and largely explains its commercial success.

A disadvantage of the process modeling approach is that modeling certain situations may be difficult or impossible to simulate depending on the software you are using. You can only include in your model those features that the software vendor has anticipated and provided for with a specific block. Another drawback is that process modeling can be very inefficient. For example, a process model of a queue with hundreds of transactions can run much slower (or not at all) than the same model with only a few customers. This is because each customer that enters the system "generates" or "creates" a separate record in memory. Earlier remarks concerning the trade-offs between resident and transient entity models apply here. (Process models that use simultaneous resources are subject to deadlock.)

5.11.3 Petri Net Simulation `PETRINET.MOD`

`PETRINET.MOD` is a animated simulation of a Petri net that illustrates the so-called "activity" approach to simulation modeling. Here tokens are removed from "places" (balls) when "transitions" (bars) are enabled. This model can be used to simulate two tandem queues. Note that a transparent bitmap is used for some of the vertices. Ungroup one of the "places" to see how this is animated. (See the section on animation for more about transparent bitmaps.).

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

FIGURE 5.29: SIGMA Simulation of a timed Petri Net.

Event Graphs are more general than standard timed stochastic Petri Nets and it is easy to convert a Petri Net model to an Event Graph - Simply replace all PN Transitions with Timed Edges connected by at most one conditional edge for each PN Place.

5.12 Exercises

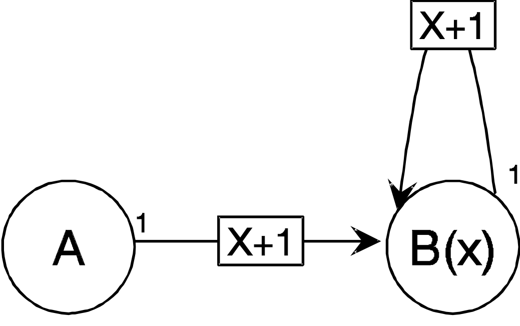
The models referred to in these exercises are SIGMA *models.*

5.12.1 Batch Arrivals to a Queue

Assume that cars arrive at a drive-up fast food restaurant with equal probability of having 1, 2, 3, or 4 customers in the car. Assume that service takes between 25 and 35 seconds per customer and cars arrive every 1 to 3 minutes (both service and interarrival distributions are uniform). Modify the carwash event graph in Figure 2.6 to model this system.

5.12.2 Parameter Passing

In the following event graph, plot the values of `X` at times `0` to `6.5`?



5.12.3 Future Events List

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

In a simulated queue with one line and five parallel servers (like BANK1.MOD):

(a) What is the minimum number of events on the events list while the simulation is running?

(b) What type of event(s) does this minimum list contain?

(c) What is the maximum number of events on the list during a run?

(d) What type of event(s) does this maximum length list contain?

5.12.4 Limited Waiting Space

Assume that the waiting line in BANK1.MOD has space for only 5 customers; there are three tellers. The maximum number of customers in the system is eight: three in service and five in line. Customers who arrive to find a full system are turned away. Change BANK1.MOD to reflect this constraint and re-run the first 30 customer service completions. Print your graph, output, and model. Record the number of arrivals that were turned away because the queue was full when they arrived. Again calculate the average waiting time for the first 10, 20, and 30 customers. Compute averages by hand or use a special statistics gathering vertex).

5.12.5 A Two Server Queue

Consider a queueing system with a single line and two identical parallel servers. The resident entity model, BANK1.MOD, and the transient entity model, BANK2.MOD, on your SIGMA disk are simulations of such a system.

Modify BANK2.MOD to have a finite capacity in this system of a total of at most 10 customers (a maximum of two customers in service and eight waiting in line). Arrivals that occur when the system is full are turned away. Customer arrivals occur according to a Poisson process with a constant rate of 1.5 customers per minute. (This is the same as exponentially distributed interarrival times with a mean of 1/1.5 = 0.667, given by .667*ERL{1}.) Each server has an exponentially distributed service time with mean of 1 minute. The arrival and service processes are all mutually independent. The objective for studying this system is to estimate the distributions for the customer waiting times and distributions for the server utilizations.

(a) Give the entities in this system (tell which are resident and which are transient). Give the attributes of each entity. Also list the sets of entities with their owners and members.

(b) List the events for this system. Give the state transformations for each event. Also give the conditions and delay times for scheduling each of the events. Show the relationships between events with an event graph.

(d) What event(s) should be scheduled initially to begin running this simulation for 100 simulated hours?

(e) What event(s) should be initially scheduled to run this simulation for 100 simulated customer service completions?

(f) What would happen if a customer tries to ENTER a full queue at the exact same time that another customer LEAVEs the server? Which of these two simultaneous events should be executed first?

5.12.6 Sensitivity Analysis

Customers arrive at an ice cream parlor at a rate uniformly distributed between 2 and 6 minutes and are served at a rate uniformly distributed between 3 and 8 minutes. Model this queue. After 4 hours, what is the average length of the queue? If another scooper is hired, what is the average length of the queue? How many scoopers must be hired to keep the average length of the queue below 2 customers?

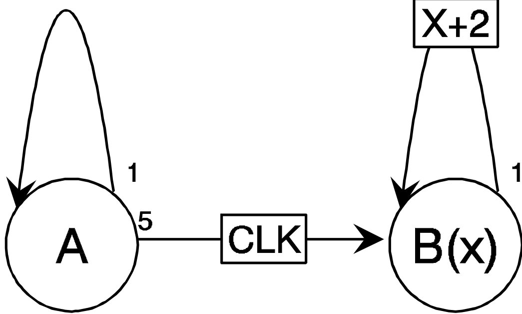
5.12.7 Fast Food

Customers waiting for fast food are very impatient. If they see 4 or more people waiting in line already, they will not enter. At peak hours, customers arrive at a rate uniformly distributed between 0.5 and 3 minutes. With current processes, each customer can be served in 2 minutes. Can the processes be speeded up enough to make sure the store never loses customers? (no). Can we speed up service so that we have a very low probability of losing customers? (yes). What is the necessary processing time for a specified probability for losing a customer that you think is reasonable? Build a model that can be used to answer this question.

5.12.8 Parameter Passing

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Give the number and types of events that will be on the list of scheduled events at time 21.5 in the following event graph.



5.12.9 A Bank

(a) Suppose that cars arrive at a bank's drive-through facility at a rate uniformly distributed between 2 and 10 minutes and each transaction requires 3 minutes. Model this queue.

(b) Now suppose that the bank added two more identical tellers and customers arrive at each one at a uniform rate between 5 and 15 minutes. Model this situation by modifying your answer for (a).

5.12.10 The Post Office

(a) Suppose customers arrive at a post office at a rate uniformly distributed between 2 and 3 minutes. A postal official can serve each customer at a rate uniformly distributed between 1 and 10 minutes. Model the service at the post office.

(b) Modify this model for a post office with 3 postal officials. Upon entry, all customers line up in one queue. The first person in the queue is served by the next available official.

(c) Modify this model again for a post office where one lazy clerk takes a 5 minute break (will not serve) after he serves 3 customers.

5.12.11 An Assembly Station

A factory mass produces widgets. Each widget has 3 component parts: `A`, `B`, and `C`. Each of these 3 types of parts is produced in its own area of the factory and then transported via assembly line to the assembly area. Parts `A`, `B`, and `C` arrive at the assembly area every 4, 5, and 10 minutes, respectively. A worker takes a time uniformly distributed between 5 and 10 minutes to assembly the widget. Model this production process.

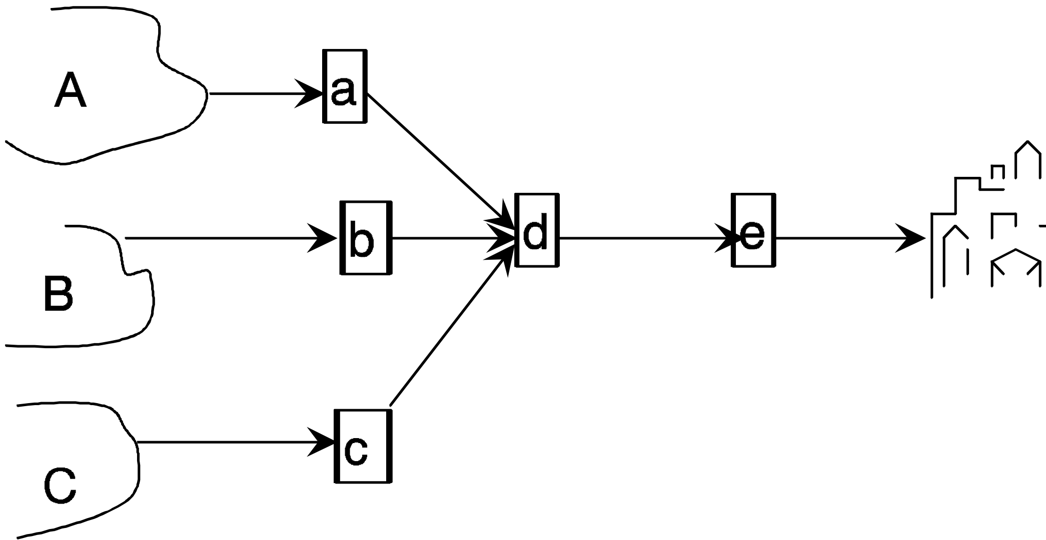
5.12.12 Communications Network Repair

A local area network has 100 identical links between work stations. Links fail at random intervals. The time until a working link fails has an exponential distribution with a mean of 5 days. Once a link fails, it must wait for a repairman to fix it. The time to repair a link has a uniform distribution between 1 and 2 hours. A single repairman is on duty at all times. Build a simulation of this maintenance system. Use your model to develop a trade-off curve between performance of the system (how should this be measured?) and the number of repairmen on duty. (Hint: Use the model, `CARWASH.MOD`, as a starting place where links (instead of cars) queue up for the single repairman (instead of a carwash machine). Also note that the rate that links fail depends on the number of working links at a given time. (This failure process is called a state-dependent Poisson process.)

5.12.13 A Water Reservoir System

Three water reservoirs (at `A`, `B`, and `C`) serve a city through the pipeline system shown below.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



Three identical (type 1) pumps at stations a, b, and c pump water from each reservoir to pumping station d. A large (type 2) pump at d pumps water to another type 2 pump at booster station e, which pumps water to the city. The time until failure of a type i pump (i = 1 or 2) is uniformly distributed between $12i$ and $4(i+1)^2$ months. Pump failures are independent, and pumps are not repaired until the water flow to the city actually stops. Model this system to predict when the city will run out of water due to pump failures.

5.12.14 Preventive Maintenance Policy Optimization

A conveyor system has 40 identical rollers, each with a failure distribution uniformly distributed between 20 and 45 days. When a roller fails, it must be repaired at a cost of $500. When preventive maintenance is performed on the conveyor, the cost is `$300+$50*N`, where `N` is the number of rollers replaced. A partial group replacement policy of performing preventive maintenance every `M` months and replacing all rollers over `A` months old is being considered. Model this system to try to suggest reasonable values for `M` and `A`.

5.12.15 Elimination of the Future Events List.

Create an event graph that is logically equivalent to the carwash model but does not schedule *any* events on the future events list. (Hint: Use a generic minimum function, `min(a,b)`, to determine the next event time.)

(a) What is the main advantage of your modified model? (Consider speed)

(b) What is the main disadvantage of your model?

5.12.16 Queues with Blocking

There are three identical and independent servers for each of two tandem queues. There is room for only 3 customers in queue 2; when queue 2 is full, no one can exit from any of the first set of three servers until a service completion for the second set of servers occurs.

(a) Identify the resident and transient entities along with their attributes. Present your entity-attribute hierarchy in outline form. Give the set of possible values for each attribute.

(b) Identify sets of entities and tell which entities (if any) "own" each set and what entities may be members of each set.

(c) Describe the discrete events in the cycles of activities for each resident entity. Also describe the events in the path of each temporary entity as they flow through the system. For each event tell what state changes occur and the conditions for these changes along with what further events will be scheduled or cancelled in a computer routine for the event.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

5.12.17 Simultaneous Events

In a queueing simulation, `BRKDN.MOD` (Figure 5.10), if an idle server is scheduled to start a break (`STARTBREAK` event) at the same time a customer arrival occurs (`ARRIVAL` event), which event should be executed first? No other events are on the events list and all other events have the same execution priority.

5.12.18 Nested Loops

(a) Create a nested loop with three index variables: `I` going from `1` to `4`, `J` going from `1` to `I`, and `K` going from `0` to `I*J`.

(b) Create an event graph which finds the sum of all the elements of a 3 by 5 array.

(c) Give the event graph for a nested loop where the inner loop moves twice as fast as the outer loop.

(d) Draw an event graph for a program that produces the entries in a table that translates from 1 to 20 degrees Celsius to Fahrenheit.

.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

6

Building Models, Verifying Simulations, and Sharing the Results of Simulation Experiments

Three objects must be defined in order for a SIGMA simulation to run: vertices, edges connecting the vertices, and state variables. In this chapter you will learn how to create and define these objects and how to make changes to them. Moreover, you will learn how to edit and build models while they are running. SIGMA's ability to let you interact with a simulation as it is processing is a valuable logic checking and model enrichment feature. Also valuable for logic checking is the Translate to English feature. This feature and others help those not familiar with your model better understand it.

6.1 Creating and Editing an Event Graph

6.1.1 Create Process Mode

The default mode for the mouse in SIGMA is the Create Process mode (⊕). This mode allows you to create a connected chain of events simply by moving the mouse and clicking in various locations within the *simulation graph* window.

If you are not in the Create Process mode, click the right mouse button or clicking on the Create Process tool in the toolbar. Clicking the left mouse button in the *simulation graph* window will cause a vertex to be created. If you move the mouse to another location and click the left mouse button again, another vertex will be created as well as an edge between the vertices. Clicking on a previously created vertex will create an edge connected to the last vertex clicked; no new vertex is created. As you continue clicking, SIGMA will create a chain of event vertices and connecting edges.

To make *self-scheduling* edges (edges with the same originating and destination vertex), simply click twice on the same vertex. This is useful when creating an input stream of arrivals to a queue.

If you are starting a new graph, your first mouse click will create the initial vertex, which is green. The variables in the parameter list for this first vertex are the initial attributes that must be entered in the Run Options dialog box.

6.1.2 Create Single Edge Mode

The Create Single Edge tool (on the SIGMA toolbar) allows you to create a single edge between two existing vertices or to create a new pair of vertices. It is useful for adding edges and/or vertices to different parts of your graph or connecting graphs copied from one SIGMA session to another. This tool is actually very similar to Create Process discussed previously ; however, here only one vertex at a time is remembered. You are first asked to select an originating vertex for the edge. If no vertex exists at the location where you click the mouse, one will be created. You then will be asked to select a destination vertex. Deactivate the Create Single Edge tool (click the right mouse button or click on the Select or Edit tool) before you attempt to enter or alter information about any edge or vertex.

6.1.3 State Variables

The Variables menu is used to define and edit the state variables in your simulation. We call these *state* variables to emphasize that they can be used in any expression, anywhere in your SIGMA model. All state variables in a SIGMA simulation must be defined before a SIGMA model will run.

When you click on the Create/Edit Variables command under the Variables menu, the State Variable Editor dialog box will appear. It is here where state variables are entered or edited. The insertion point will be in the Name text box. Enter the name of each new state variable here. Use the mouse or [Tab] to activate the Size, Type, and Description text boxes and enter the appropriate information. Clicking Add or pressing [Enter] will cause the state variable information to be entered in the list box below. The insertion point will automatically revert back to the Name text box. The number of state variables allowed may be limited by the available memory on your machine or the version of SIGMA you are using. Be sure to click OK to save changes to the model; if you do not, your changes will be lost.

To edit an existing state variable, click on that state variable in the list box. This will cause the state variable information to appear in the text boxes above. Use the mouse to activate any boxes where changes are needed, then enter the new information from the keyboard.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

EDITING STATE VARIABLES: *Be sure to press the* OK *button after clicking* Rename, Add, Delete, *or* Change. *If* OK *is not pressed, the changes will be lost.*

The Name you give to a state variable can be used in vertex state change expressions and in expressions for edge conditions, delay times, priorities, and attribute strings. State variable names have a maximum length of 8 characters. Names should begin with a letter but may be composed of letters and numbers. If you intend to automatically translate your model to C, you should use the appropriate naming conventions. Note that two functions reserved for SIGMA (`CLK` and `RND`) cannot be used as state variable names. In SIGMA, variable names and expressions are in upper-case. This is because reserved words in ANSI standard C are in lower-case, and we do not want to confuse C when we translate our models to source code. Compilers typically have some non-standard functions added to their libraries, so you must avoid using these function names.

You can select a Size for a variable. (If `Size > 1`, it is an array.) The maximum array size, typically four-dimensional arrays are allowed, may be limited by the free memory you have on your computer or the version of SIGMA you are using. Following the convention in C, in SIGMA we start indexing arrays with zero (not 1 as in FORTRAN). Array elements in a model can be recycled using the modulus function included in SIGMA. In SIGMA, a variable size can be from one to four dimensions. Two-dimensional variables are used for modeling tables of data. To define a variable that is a two-dimensional table, specify the maximum size of each dimension separated by a comma (e.g., `5,6`). The default size is a scalar of size `1`. You select the size of each array, within default limits. Array index values can be scalars or arrays of real or integer numbers. Real valued numbers are rounded down when used as indices for an array. If `X` is a one-dimensional array with `Size=5`, we reference the elements of `X` in SIGMA expressions as `X[0]` through `X[4]`. If `Y` is a two-dimensional table with `Size=5,6`, we can reference the entries as `Y[0;0]` through `Y[4;5]`. (Note that indices for elements of a table are separated by a semicolon.)

At the Type drop-down box, you can choose one of three types of variables currently supported in SIGMA. In addition to the usual integer and real (floating point) variables or arrays, you may create a general type of variable called a `USER` variable, which is beyond the level intended for this documentation.

The Description of each state variable is very important. It is strongly encouraged that at least brief descriptions be given whenever a SIGMA object is created.

6.1.4 Editing Vertices

Changes in the values of state variables occur only when a vertex is executed. Formulas for the state changes associated with a particular vertex are entered using an Edit Vertex dialog box, which is displayed when the mouse is double-clicked on a particular vertex. An Edit Vertex dialog box contains the following informational boxes:

Name and Description: Vertex names must be seven characters or less. Vertex descriptions should be included since they provide valuable comments in SIGMA-generated simulation source code.

Trace Event: The Trace Event mechanism allows the vertex to be traced in the numerical output file. On is the default. Thus, if no action is taken, all events will be traced in the numerical output file. Deactivate the vertex by clicking the Trace Event check box. Similarly, an event will not be traced if you click the vertex and then click the Trace Off command in the Edit menu. The Trace Event box can be turned On or Off in several vertices by holding down [Shift], clicking on the appropriate vertices, and then clicking on the Trace Off command in the Edit menu.

Inactive Picture/Active Picture & Transparent Bitmaps: These features are used for animations. Inactive & Active bitmaps allow you to assign two different bitmaps to each vertex so motion can be represented**.** Transparent bitmaps are white in color, (255,255,255) in Windows Paintbrush and RGB(0xFF,0xFF,0xFF) in C. Note that transparent bitmaps may slow model execution

New Vertex Shapes: A vertex can now have one of thirteen shapes, without the need for assigning bitmaps. Use the drop down list to change the original vertex shape from a circle to a square, diamond, oval, down triangle, up triangle, down arrow, up arrow, right arrow, left arrow, delay symbol, Enter symbol, or Exit symbol. Thus, SIGMA can be used as a general flowcharting tool that also runs a simulation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

State Change(s): There are typically one or more state variable changes associated with each vertex. The operators available for use are +, –, /, *, and ^ for add, subtract, divide, multiply, and exponentiation respectively. Different expressions are separated by commas.

Parameter(s): Parameters are the values of state variables passed as attributes of the scheduling edge. The parameter list is a string of state variables separated by a comma. The main value of vertex parameters is in defining the particular system entities to which a vertex pertains. For example, suppose that there were two identical machines in a simulated factory. The same START vertex may be used for both machines if the vertex has a parameter telling which machine is to start.

To solidify our understanding of vertex parameters, look at a vertex in a queueing simulation that occurs when a customer starts to be served. The state changes are given as the string,

QUEUE[S]=QUEUE[S]-1, STATUS[S]=0.

When this vertex executes, the number of customers waiting in the QUEUE for station, S, is decreased by one and the status of the server, STATUS, at station, S, is set equal to 0 (BUSY). A value for the parameter, S, (denoting which of several possible service stations is involved) is passed to the vertex as an attribute of the edge that originally scheduled this vertex.

Display Variables: A state variable and its value at a particular time can be displayed near a vertex during a run. The current value of the display variable is updated every time any event is executed. To display more then one variable, separate each with a comma. Display variables are useful for animation and debugging.

Locations: This drop-down list shows where a display variable will be located relative to a vertex: to the left, right, top, or bottom.

6.1.5 Editing Edges

Edges connect pairs of vertices in a simulation graph and define how one event may cause (or prevent) the occurrence of another event. There can be multiple edges between any pair of vertices in the model.

Information concerning an edge can be found in its dialog box. To open the dialog box, double-click on the edge. To change characteristics of an edge, click the mouse on the text box of interest and enter the change from the keyboard. An Edit Edge dialog box contains the following input boxes:

Originating and Destination Vertices: The originating and destination vertices for an edge are entered automatically from the event graph. You can change the originating and destination vertices by copying or moving edges on a graph.

Pending, Scheduling , and Cancelling Edge Types: The SIGMA default is the pending edge. Currently both pending edges and scheduling edges will schedule the destination vertex. A cancelling edge will cancel the destination vertex. (Cancelling edges are represented by dashed lines on the graphical model; pending and scheduling edges by solid lines.) When a vertex cancelling edge is executed, it cancels only previously scheduled vertices of the same type as the destination vertex of the edge. If the edges have any attribute expressions, only scheduled vertices with parameter values that are an exact match with the values of these edge attribute expressions are cancelled. If the attribute on a cancelling edge is entered as an asterisk (*), all scheduled occurrences of the destination vertex will be cancelled regardless of their parameter values. If a cancelling edge has no attributes, only the next scheduled vertex of the same type as the edge destination vertex (if any) is cancelled.

Edge Delay Time: This is the expression for the delay time between the occurrence of the originating vertex and the occurrence of the destination vertex. Cancelling edges have zero delay times; cancelling is immediate, regardless of a delay expression or execution priority.

If the expression for the condition on a pending or scheduling edge in SIGMA is true and the delay time expression is simply an asterisk (*), SIGMA will execute the vertex state change immediately. This is called *pre-emptive vertex execution*; the vertex is executed without being placed on the future events list. The effect is that the two vertices on this edge are condensed into a single vertex. Bypassing the future events list generally speeds up model execution without making the state changes for each vertex more complicated.

Caution should be used with pre-emptive vertex execution. Since pre-emptive vertex execution overrides execution priorities; potential problems in simultaneous vertex execution have to be considered. You should try to avoid having a cycle in your graph where all the edges have pre-emptive execution, such as a self-scheduling loop. The resulting

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

loop is likely to run until there is a stack overflow due to recursive function calls. Finally, there must be at least one event on the future events list or your SIGMA run will terminate; a generic STOP RUN event (scheduled when the run starts) is appropriate if all edges in a model have pre-emptive execution.

> If four or five pre-emptive vertex execution edges are placed in sequence, a stack overflow may occur. (While this is not a serious limitation, you may be able to avoid this by increasing your Windows stack size.)

Edge Conditions: The cancelling or scheduling of the destination vertex can be made conditional on the state of the system at the time the originating vertex executes. Edge conditions are entered as expressions. The operators you may use in edge conditions are +, –, /, *, ^, defined earlier in this chapter, along with :

==	(equal to)	>	(strictly greater than)
!=	(not equal to)	<=	(less than or equal to)
<	(strictly less than)	>=	(greater than or equal to)

You may combine conditions by using the Boolean operators & (and) and ¦ (or). For example, the edge condition, QUEUE>1 & STATUS==0, means that the destination vertex will be scheduled to occur (after the edge delay time) only if QUEUE is currently greater than one *and* STATUS is equal to zero. Readers familiar with C should note that the explanation mark (!) is not used by itself in SIGMA to negate a general condition and a single & and ¦ are used for Boolean operators rather than the double && and ¦¦ used in C.

Execution Priority: This is an expression for a scheduling edge that computes an execution priority for the destination vertex being scheduled. Edge priorities are used to break time ties in scheduled vertices. When two or more vertices are scheduled to occur at the same time, the event with the higher priority (the lower numerical value) will be executed first. Remaining ties are broken by giving precedence to the vertex that is scheduled last.

In SIGMA, the default execution priority is 5. The execution priority of events that are to be executed earlier should be changed to a higher priority (e.g., 1, 2.4, 3.6, or 4). The execution priority of events that are to be executed later should be changed to a lower priority.

If simultaneous events are executed in the wrong order, the logic of the simulation may be wrong. Consider the example where a customer arrives at a capacitated queue that is full and a service completion coincides with that arrival. If the customer arrival event is executed first, the customer will see a full line and depart. If the service completion event is executed first, a space will be made in line for the arriving customer and the customer will not depart. In simulations of serial systems (factories, communication networks, etc.), executing simultaneous events in the wrong order can cause the model to behave in a very strange manner.

No good "rule of thumb" for assigning execution priorities has been devised to replace thinking through the chain of consequences of simultaneous events. However, you should be alert to the possibility of simultaneous vertex problems whenever *the state changes in one vertex and the conditions tested in the exiting edges for another vertex involve the same variables.* Any two such vertices should have appropriate execution priorities passed to them. The order in which the different expressions are computed can be found in the Event Execution Sequence in Appendix A.

Edge Attributes: Edge attributes are values passed to vertex parameters. Each edge has an attribute list, and each vertex has a parameter list. These are simple but powerful modeling tools. The edge attribute list is a string of expressions separated by commas. When the originating vertex for an edge is executed, the expressions in the edge attribute list are evaluated. These will be the values assigned to the state variables in the destination vertex parameter list when that vertex is subsequently executed.

For example, consider an edge with attribute list:

X+1,5,QUEUE*N.

Say that the destination vertex for this edge has the parameter list:

Z,N,QUEUE.

Suppose when the originating vertex for this edge is executed, the state variables have values X=5, N=10, and QUEUE=2. When the destination vertex is subsequently executed, the values for the state variables Z, N, and QUEUE will be set equal to 6, 5, and 20. This assignment is made before any vertex state changes and regardless of the values for X and QUEUE when the scheduled vertex executed. The expression QUEUE*N was evaluated when the vertex was scheduled; the values

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

of N and QUEUE at that time were 10 and 2 respectively, so QUEUE*N was equal to 20. The number 20 is literally placed on the future events list as an parameter of the scheduled event.

> **IMPORTANT:** *Remember that any state variable in* SIGMA *may be changed indirectly, by passing values to vertex parameters, as well as directly, by the state changes in vertices.*

If an edge has more attribute expressions than its destination vertex has parameters, the extra edge attribute values are ignored. If an edge has fewer attribute expressions than needed, an error message occurs.

> ***Array elements must be specified explicitly -- e.g.,* A[3]*, not* A[Q]**.

6.1.6 Moving a Simulation Graph

All or part of a simulation graph can be moved around the graphical window relatively easily. First click on the vertex to be moved with the mouse, hold down the left mouse button until the pointer changes shape, and drag the vertex to the desired location. Edges are moved by highlighting the edge and then dragging it to a new *vertex*. To be successful, there must be a vertex to serve as a destination for the edge.

Multiple edges and vertices can be highlighted using [⇧Shift] and the mouse and moved by clicking the mouse on the group and dragging it to the new location. If you wish to move the entire graph, use Edit/Select All. Vertices in event graph models can also be moved one pixel at a time by highlighting the vertex and using the arrow keys.

6.1.7 Copying Event Graph Models

SIGMA allows you to copy and paste information within a single SIGMA modeling session or between SIGMA modeling sessions.

Copy all or part of an event graph by selecting the desired components, clicking on the Edit/Copy command, and then clicking on the Edit/Paste command. Move the copied subgraph to the desired location and then connect it to the original graph using the Create Single Edge tool. A single edge is copied by pressing [⇧Shift] and also holding down the mouse pointer on the desired edge until the mouse pointer changes shape and dragging the mouse to another vertex. Move edges that may occupy the same bounding rectangle out of the way so they are not copied also.

Copied state variables are added to the current set of state variables; duplicate names are deleted. Vertices that have been copied are renamed. Dialog box associated with the copied vertices contain all the information from the original vertices. The Copy-Paste feature is particularly useful when developing a model containing similar but complex vertices or edges.

> [Ctrl-c] *will copy information,* [Ctrl-x] *will cut it, and* [Ctrl-v] *will paste it to another area (or to and from another Windows application, such as Word for Windows).*

Event graph models (or parts of models) also can be selected, copied, and pasted between different SIGMA sessions. First, open the SIGMA session with the model to be copied. Highlight the components to be copied, then click on the Edit/Copy command. Open a second SIGMA session and use the Edit/Paste command to paste the first model into the new SIGMA session. Move the newly-pasted sub-graph to the desired location. It is suggested that you save intermediate graphs under different names. The Copy-Paste feature is also useful when several persons are working on different parts of the same simulation model. This permits the modeler (or a team of modelers) to create separate parts of a simulation model in different SIGMA windows and later connect them together into a larger simulation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

6.1.7.1. SIGMA User Tools

Another way to copy previously created models is to SIGMA's User Tools feature. SIGMA models can now be saved as User Tools. This option is easy to do and makes model building faster than ever.

There are fifteen User Tool buttons (like the one above) running vertically along the right side of the SIGMA screen. With a click of the mouse you can add an entire model into a SIGMA modeling session. In this way, you can re-use previously created models that have components similar to those desired in a new model.

To create a User Tool:

1. Open a SIGMA model that you would like to reuse.
2. Click on the Save As command under the File menu.
3. Save the model with the name TOOLN.TOL in the Save Model As dialog box (where N is the tool button number).

To illustrate, save a model as TOOL1.TOL. (Do not merely rename a model since SIGMA alters models to allow them to be appended if they are saved as type *.TOL.) Now whenever you press the Σ_1 tool button, the entire model you called TOOL1.TOL will be pasted into your current SIGMA modeling session. Note that if you press Σ_1 twice, both event graphs will be pasted into the same location. It is best to move each new model aside as it is added. Use the Create Single Edge button to join the separate event graphs together, and update the Edge and Vertex dialog boxes as needed with regard to passing attributes and parameters.

6.1.8 Saving SIGMA Models and Output Plots

If a model has not been saved previously, activate the *simulation graph* window and click on the File/Save or the File/Save As command. This action will produce a dialog box in which you can enter a name for the file where the model is to be saved. Again, the convention for naming files containing SIGMA graphical models is to use the .MOD file type (e.g., CARWASH.MOD). File names must be 8 characters or less and begin with a letter or number. A backup copy of the last model saved is retained with the extension .BAK.

To save an output plot, activate the *simulation plot* window, click on the FILE/Save command, and save the plot using the .PLT file type (e.g., CARWASH.PLT). You can retrieve a saved simulation plot file by activating the *simulation plot* window, clicking on the Open command under the File menu, and then double-clicking on the file name.

> *It is strongly suggested that all* SIGMA *models be saved with the file type,* ".MOD." *This is useful for debugging and documentation.*

6.2 Dynamic Run-Time Model Building and Analysis

SIGMA lets you interact with a model during a simulation. You can alter your model in virtually any way you wish *while it is running!* For example, with a running model, you can change an edge condition or delay time expression by double-clicking on the edge and making the necessary changes in the dialog box. If you do the same to a vertex, you can modify the state changes or displayed variables. You can change the run mode by making changes in the Run Options dialog box during a run. You can even add and delete edges or event vertices during a run.

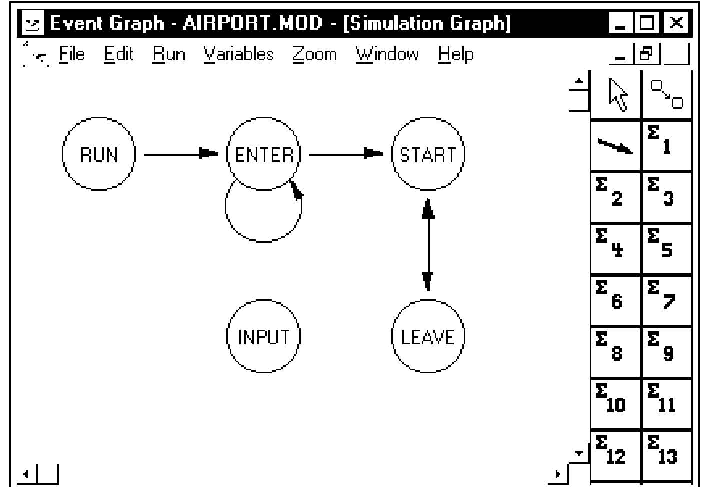
> *Changing a simulation during execution is simple: just make the changes as if the model were not running!*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

6.2.1 Changing the Value of State Variable Values During a Run

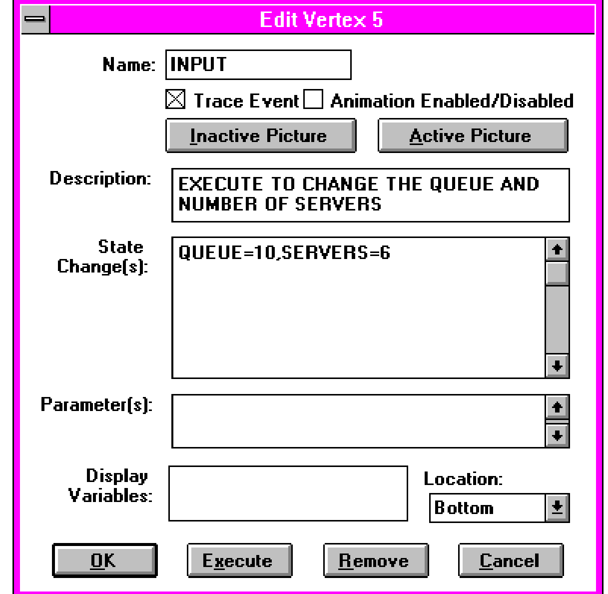
To illustrate the usefulness of changing variable values during a run, we will examine a simple model of a multiple-server single-line queue such as that found in an airport check-in line—AIRPORT.MOD. If you open AIRPORT.MOD, your screen should look like the one in Figure 6.1. (This model is very similar to BANK1.MOD but an INPUT vertex has been added.)

Figure 6.1: A Multi-Server Queueing Model - AIRPORT.MOD



Start this simulation. Watch the queue increase for a few seconds in the simulation plot window. Next, double-click on the INPUT vertex; it will not have executed since it is not connected to the graph. The Edit Vertex dialog box for the INPUT event should look like the one in Figure 6.2.

Figure 6.2: The Dialog Box for the INPUT Vertex



When you click on the Execute button in an Edit Vertex dialog box during a run, that vertex will be the next event executed. State changes for this vertex will then occur. Try this by pressing Execute in the INPUT vertex. You will see the QUEUE jump to 10 in the simulation plot, and the number of idle servers will be reset to 6. Continue to increase the number of idle servers in the running model until the system appears to stabilize. Do this by repeatedly editing and executing the state changes for the INPUT event. Note that in this model the variable, SERVERS, is defined as the number of *idle* servers. Thus, servers that are busy will stay in the system; only the QUEUE and number of idle servers are changed.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

To delete servers, double-click on the LEAVE vertex and close the resulting dialog box with the Remove button. This will remove the next pending LEAVE event, effectively removing one busy server.

The Execute and Remove commands buttons at the bottom of the Edit Vertex dialog box are useful for executing an event or removing an event from the future events list during a run. Since the executed event can itself schedule or cancel other events, this gives complete run-time control over the future events list.

It is good practice to include an isolated empty vertex, like the INPUT vertex in this example, for making variable changes during a run. You could use the ASK{} function discussed in Chapter 7 to do the same thing. However, that function is more for interactive games, where you want to prompt the model user to make a decision during a run.

> **NOTE:** *Change the values of variables during a run to feasible values. For example, you can change* QUEUE *to a negative number if you wish; however, it makes no sense to have a negative number of customers in line.*

6.2.2 Changing Edges and Vertices During a Run

Now try double-clicking on the self-scheduling edge for the ENTER event.You can change the time between arrivals by editing the delays time in the Edit Edge dialog box. The changes you enter will take effect as soon as you press OK. You make change to vertices similarly, just as if the model were not running. Again, changes take effect the next time the vertex is executed in the run, or you can force the vertex to execute immediately by clicking on Execute on the dialog box. In this manner you can ask "what-if" questions concerning changes in demand rate and number of servers *while the model is running*.

6.2.3 Adding and Deleting Edges and Vertices During a Run

To delete an edge click once on the particular edge and then press [Delete]. (This should be done carefully since you are now changing the model logic not just the values of variables.) Delete vertices the same way, by clicking once on the vertex you wish to delete and pressing [Delete].

> **CAUTION:** *Adding or deleting edges or event vertices during a run may change the model logic.*

You can connect a vertex to a running model or create a new edge. First click on the vertex that is to serve as the origin of the new edge. Next, click the *right mouse button* to get into Create Process mode. Then click where you would like to have a new vertex created. (Click on an existing vertex if you want only a new edge.) Continue clicking if you would like to spawn a series of new vertices and edges off an existing vertex. Press the *right mouse button* again to return to Select or Edit mode. Click on the new edges or vertices to open their dialog boxes. You will generally alternate right and left mouse clicks as you add new vertices and edit them in a running model.

> **NOTE***: Do not try to delete a vertex while there are grouped vertices in the model. See the chapter on animation for a discussion of vertex groups.*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

6.3 Model Enrichment and Logic Checking

SIGMA has very powerful simulation model enrichment and logic checking features.

6.3.1 Setting up the Logic Checking Environment

One of the most important things to check before you start to verify a simulation model is to insure that all the edges have enough attribute expressions for each of the parameters of the scheduled vertex. The *most* common error is not to have enough "initial attributes" in the Run Options dialog box for the parameters of the first vertex.

A model verification session will go as follows. Start a SIGMA session and open the model to be checked. (Here we will again use CARWASH.MOD.) Open the Run Options dialog box in the Run menu and set the Run Mode to Single Step. Next, make the run very short by setting the Stop Time to a small number (10 will do). This is to initialize data tracking. Start the run by clicking OK & Run. Press [←Enter] several times or click on the Single Step tool to execute some events. Press the End Run tool to stop the run. Click Yes when asked to view the output file**.**

At this point you should have three windows tiled in your SIGMA session: a numerical output window for the output file listed in your Run Options dialog box (UNTITLED.OUT), the simulation graph, and the simulation plot.

Click once on the simulation graph window to activate it. Next, click on View Output/Text File under the File menu and then click on CARWASH.MOD; this will open the ASCII text file for your model. (If you don't see CARWASH.MOD, press the drop down list under List Files of Types and click Other Files.) Under the Window menu select Tile. You should now have four windows showing: a simulation graph, a simulation plot, an output file, and a model text file. Activate the simulation graph window and set the Stop Time in the Run Options dialog box to a very large number, so your model will continue to run as you verify its logic.

6.3.2 Starting a Logic Checking Run

Press OK & Run to start the run. Make sure you are again in Single Step mode and step through your model by pressing [←Enter] or clicking on the Single Step tool in the Single Step Window. Every time the "Refresh" arrow [↓] (near the upper left corner of UNTITLED.OUT) is pressed during the run, the output trace will be updated and added to the output file.

If you have an edge that is not scheduling a vertex as you expected, click on the scheduling vertex (while the model is running) to open the Edit Vertex dialog box and set some Display Variables that are tested on the bothersome edge. (Enter the names of the variables separated by commas).

If you activate the *simulation plot* window and click on the New command under the File menu, you can open a copy of the simulation plot. This plot can be double-clicked to change its plot type. In this manner, you can view several plots at once. For example, one could show a histogram and another a scatter plot. Only one plot will be updated during a run; the other can used to record the run history.

The PAUSE{} function can be used to debug long runs. If you find a logic error at time 1000, you can trap this error by scheduling a dummy event with the state change X=PAUSE{} at time 999. Run the simulation in High Speed mode until PAUSE is hit, then change the run mode to Single Step, and continue logic checking.

6.4 Automatic Translation of SIGMA Model

SIGMA automatically translates a SIGMA model into both English and C code.

6.4.1. English Translation

SIGMA's automatic English translation feature is very useful for checking errors (use with a printed graph), verifying a model with persons not familiar with simulation, or supplementing physical animations.

This feature exploits SIGMA's graph structure to produce an English description of your simulation. Basically the two vertices on an edge act as the subject and object of a sentence while the edge conditions and delay times form prepositional phrases; the predicate is almost always "schedule" or "cancel."

To see the English translation of the carwash model, open CARWASH.MOD. Activate the English translation option by clicking the Translate command in the File menu and selecting English from the alternatives. A dialog box will appear with the active file highlighted in the File Name text box. Press the OK command button to see the English translation, press Yes to replace an existing file, and Yes again to view the file now.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The following English description of our basic carwash model was obtained using the Translate command on the graphical model.

```
The SIGMA Model, CARWASH.MOD, is a discrete event simulation. It models an
automatic carwash.

I. STATE VARIABLE DEFINITIONS.

For this simulation, the following state variables are defined:

QUEUE:   number of cars in line  (integer valued)
SERVER:  machine is idle/busy = 1/0    (integer valued)

II. EVENT DEFINITIONS.

1. The RUN(QUEUE) event models the initialization of the simulation. Initial
   values for, QUEUE, are needed for each run. This event causes the following
   state change(s):
   SERVER=1
   After every occurrence of the RUN event:
   Unconditionally, the car will enter the line;
   that is, schedule the ENTER() event to occur without delay.

2. The ENTER() event models the car entering the line.
   This event causes the following state change(s):
   QUEUE=QUEUE+1
   After every occurrence of the ENTER event:
   If SERVER>0, then start service with the idle machine;
   that is, schedule the START() event to occur without delay.
   Unconditionally, the next customer enters in 3 to 8 minutes;
   that is, schedule the ENTER() event to occur in 3+5*RND time units.

3. The START() event models the start of service.
   This event causes the following state change(s):
   SERVER=0
   QUEUE=QUEUE-1
   After every occurrence of the START event:
   Unconditionally, the car will be in service for 5 minutes;
   that is, schedule the LEAVE() event to occur in 5 time units.

4. The LEAVE() event models the end of service.
   This event causes the following state change(s):
   SERVER=1
   After every occurrence of the LEAVE event:
   If QUEUE>0, then start service for the next car in line;
   that is, schedule the START() event to occur without delay.
```

You can see that if you completed the sentences in the Description line in all of the dialog boxes for the vertices and edges in a simulation, the English translation of your model should be quite readable. Close this file.

6.4.2 Translate to C

Once you have verified the logic of your model, you can automatically generate a simulation program in the C Programming language using the same Translate feature discussed above. You specify the model (e.g., CARWASH.MOD) you wish to have translated and select a C as the target language; the rest is automatic.

To translate a model into C, activate the *simulation graph* window, click on the Translate command in the File menu, and click on C. A dialog box will appear with the file name highlighted (e.g., CARWASH.C). Click the OK button to

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

confirm the file name. Respond Yes if you are asked to replace an existing file and Yes when asked to see the translation now. Use the scroll bars to view the entire file.

6.5 Capturing a Simulation and Its Results

The ease of exporting data to word processors makes it much easier to develop reports for simulation experiments. Not only can the event graph be included in the summary reports, graphical plots, numerical output, and even an English translation of the model can be added to a written report.

6.5.1 Printing Event Graphs, Simulation Plots, and Output Files

It is very easy to print event graphs, graphical simulation plots, and numerical output files from models created in SIGMA. To print a graph, plot, or output file, just activate the appropriate window, click the Print command in the File menu, and respond to the Print dialog box.

6.5.2 Using Spreadsheets and Word Processors

SIGMA graphs, plots, and output can be incorporated into modern Windows spreadsheets and word processors. If you copy [Ctrl-c], an event graph and then paste [Ctrl-v]. it into a word processing file, you may be surprised at the results. Rather than the event graph, you will see the textual description of your event graph (the simulation model data). If, on the other hand, you activate the simulation graph, press [Alt-Print Screen], open a word processing file, and then click Edit, the event graph image will appear in your file.

SIGMA also allows you to copy, cut, or paste dialog box text between a modeling session and another application. This feature is particularly useful when entering lengthy, but similar, edge conditions. For example, you could write the text using Word for Windows, highlight the text and copy it using [Ctrl-c], then paste it into a SIGMA dialog box using [Ctrl-v].

To move output data from SIGMA into a spreadsheet: activate the *simulation plot* window in SIGMA, click on the Edit/Copy Data command, open the spreadsheet program, and click the Edit/Paste command. To export a simulation plot to a word processor: activate the *simulation plot* window, click on the Copy Plot command in the Edit menu, open a word processing file, and click on the Paste command in the Edit menu.

6.6 Exercises

The models referred to in these exercises are SIGMA *models. Exercises identified as mini projects are more extensive and may take considerably longer than the typical exercise.*

6.8.1 Event Execution Priorities

Modify CARWASH.MOD so that arrivals occur exactly every 2 minutes and make the execution priorities on every edge the same. Run the model for 50 minutes then look at the output. What happens when an ENTER event and a LEAVE event take place simultaneously? What event should be executed first if there is an ENTER event and a START event scheduled at the same time and QUEUE=1?

6.8.2 Pre-emptive Vertex Execution

Using pre-emptive vertex execution, modify the model, CARWASH.MOD, so that the logic of the simulation does not change but the number of events scheduled on the future events list is minimized. (Hint: Make some of the edge delay times equal to *; you will in effect incorporate the START vertex into *both* the ENTER and the LEAVE events).

Why might the output from a model that uses pre-emptive vertex execution be different from the output of the same model where pre-emptive execution is not used, i.e., all * delays are changed to 0? Why might the outputs be different even if the two models are logically equivalent in a probabilistic sense (output sample paths have the same probability of occurring)?

6.8.3 Pre-emptive Vertex Execution with Parameter Passing

Using pre-emptive vertex execution (edge delay times set equal to *), modify the multiple server queue, BANK2.MOD, so that the logic of the simulation is not changed but a minimum number of events are scheduled on the future events list.

6.8.4 System Reliability

Consider two systems, each with 50 identical components. System A is a serial system that fails when the first component fails. System B is a parallel system that fails when the last component fails. The times until failure, T, for the components

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

are independent and have beta distributions with parameters 0.5 and 0.2. System B is obviously more reliable; however, it is desired to estimate the expected difference between the times until system failure for the two systems.

(a) Model both systems and then estimate the difference between the failure times of the two system.

(b) Would the variance of the sample difference generated in part (a) be biased? If so, in what direction?

6.8.5 A Drive-in Bank Window (Mini Simulation Project)

A drive-in window is added to the side of the bank being modeled by BANK2.MOD. This window is serviced by the same tellers as before. The tellers rotate in serving customers at the drive-in window. Each teller will serve one drive-in customer and then return to service the line in the bank. If there are no customers at the drive-in window when a particular teller's turn comes up, s/he will miss a turn at the window.

Drive-in customers arrive according to the same pattern as the customers who arrive on foot (as described in the text) except that during the noon hour (12:00 to 1:00 P.M.) drive-in customers arrive at the rate of 2 per minute. Drive-in customers will balk if there are three or more cars waiting for service at the drive-in window. Of the balking drive-in customers, 80% will try to park their cars in the bank parking lot and walk inside for service. Once a former drive-in customer parks and walks inside, s/he will not balk unless the line has more than 20 customers in it.

The bank rents parking spaces in an adjacent parking lot for its customers. Currently the bank rents spaces for 10 cars. Customers desiring parking will wait only 30 seconds in a full lot before balking. Assume that it takes between 2 and 3 minutes (uniformly) for a customer to travel from the lot to the bank and vice versa (including time to park).

To better serve their customers, the bank is considering hiring additional tellers. Each teller costs the bank $1500.00 a month in salary and fringe benefits. As an alternative, the bank is considering renting additional parking spaces at $150.00 a month each.

Experiment with your simulation to advise the bank if they should hire tellers or rent parking spaces (or both). State your decision criteria and any additional assumptions that you need to make (e.g., tellers absent from work). Evaluate each of your assumption. (Are the assumptions conservative? Are they realistic? Can you confirm them with real-world data?)

6.8.6 Variability and System Performance

The model FLOWSHOP.MOD is a simulation of a flow shop where there are several parallel machines at each of N sequential stations. A part needs to be processed by only one machine at each station. Run the system for three stations in a series where there are 5 machines at the first station, 3 machines at the second station, and 4 machines at the third station. The mean processing times (in hours) at each station have uniform distributions with means of 0.2 for each machine at the first station, 0.1 for the second station, and 1.5 at the third station. The processing time range is ± 0.05 for all machines. Run several 8 hour shifts and increase the processing time range to see the effect of variability on the performance of the system.

6.8.7 A Queue with Service Breaks

Modify the model, BANK2.MOD, to more accurately simulate a ticket line at a major airport. At the ticket counter there are three agents. The time it takes an agent to serve a single passenger has an ERLANG (3) distribution with a mean of 15 minutes (5*ERL{3}). Every hour each agents takes a 5 minute break. They rotate breaks so one agent goes on break every 20 minutes. When an agent's break time is due, s/he will immediately stop whatever s/he is doing and go on break. If the agent is busy at the time, the passenger must wait until the agent returns from the break. An agent will finish the remaining service of a customer when s/he comes off break. Customers may not change agents once they start service even if the agent goes on break; the agent has their ticket. All passengers arrive at this departure desk in a taxi. The time between taxi arrivals has an exponential distribution with a mean of 1 minute. Passengers sometimes arrive in groups to share the taxi fare. Each taxi carries between 1 and 4 passengers. The likelihood of a group size of 1 is 0.6, the likelihood of a group size of 2 or 3 is 0.15, and the likelihood of 4 passengers arriving in a taxi is 0.1. Customers arrive in groups but each is processed alone. Run your model and watch the queue build up. Do you think the queue length will ever stabilize or do you expect it to continue to grow in an unbounded manner?

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

6.8.8 Project Management

In the project management model in Section 5.6, assume that the activity times have beta distributions with both parameters equal to 0.5 (see Chapter 7) and scaled to be between the limits given in the following table.

Activity	Must Follow Activities	Number of Workers	Time Range
A	None	1	1-3
B	A	1	1-5
C	None	1	2-4
D	B and C	2	3-7
E	B and D	1	2-4
F	C	2	4-6

(a) Run the simulation of this project 100 times and estimate the probability that the project will take longer than 14 days.

(b) Assume that there are only 3 workers and a task needs the number of workers specified in the table above (tasks cannot be done partially, they must be completed once started). Modify the model to account for this resource constraint (`PUT` activities in a queue when their precedence constraints are met but not enough workers are available to start; `GET` tasks from this queue when activities finish.)

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

7

Using SIGMA Functions

A number of predefined functions have been incorporated into the SIGMA simulation environment: functions for reading data files, runtime interactions, bookkeeping, mathematics, modeling priority ranked queues, generating random variables, and computing statistics. A summary of SIGMA functions is presented followed by a more thorough discussion of some selected functions. A table of models that illustrate these functions is at the end of this section.

7.1 Summary of SIGMA Functions

Functions can be placed in expressions, which can be used in state changes, edge conditions, edge priorities, and edge attributes. The functions are treated just like any state variable. Arguments of SIGMA functions are in braces { } and multiple arguments are separated by semicolons.

Functions can often be nested and used as arguments for each other. For example, `-6*LN{RND}` (a random number function used as the argument for a natural log) will approximate an exponentially distributed random variable with a mean of `6` (see Chapter 9 for details on random variate generation). Any of the expressions in these functions can, of course, be replaced by real constants, which are rounded down if integers are required. Note that you cannot use an array as an argument for a function. For example, you should break the function `MAX{X[I;J];Y}` into the two statements: `Z=X[I;J],MAX{Z;Y}`.

> NOTE: *Arguments of* SIGMA *functions are in braces* { } *and multiple arguments are separated by semicolons* (;).

SIGMA functions are defined below. (All arguments can be general expressions unless otherwise specified.)

`ASK{X}` will display the value of expression `X` during a run and ask if you wish to change it. You can type in a constant or any other valid expression. If you press Enter, the expression is unchanged. This is useful for debugging and interactive gaming. Example: `Q=ASK{Q+1}` defaults to `Q=Q+1` unless you enter some other expression or value for `Q`.

`AVE{X}` is the cumulative (count) average of the traced variable, `X`. Use `TAV{X}` for time averages.

`BET{X;Y}` generates a standard Beta variate with nonnegative parameters given by the values of expressions `X` and `Y`.

`CGET{Condition;List}` will get the values for the first entity, if any, on list, `L`, where the general condition, `CONDITION`, is true. This entity is removed and its attributes made available as the elements of the `ENT[ ]` array. This powerful list and data processing function for modeling priority queues and its companion functions, `PUT` and `GET`, are discussed in detail later in this Chapter.

`CLK` is the current simulated clock time (which is automatically updated to the time of each event occurrence) - this is commonly used as a state variable.

`COS{X}` gives the cosine of the expression, `X`, in real format.

`DISK{F;I}` reads the `I`-th number from disk file, `F`, where `I` is any valid integer-valued SIGMA expression. If `I=0`, the file is read sequentially. Data files can be read in any (random) order. Details on this function are presented later.

`ERL{X}` generates a standard pseudo Erlang random variate with shape parameter given by the value of expression `X` rounded down to the nearest positive integer. This is the sum of `X` exponential random variates each with mean 1, so `M*ERL{1}` is a pseudo exponential random variate with mean `M`.

`GAM{X}` generates a pseudo gamma random variate with fractional shape parameter given by the value of expression `X` with `0<X<1`.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

GET{O;L} gets values for the ENT[] (entity) array from list, L, according to option, O. (L is an integer.). More details on this and the companion function, PUT, are presented later. The options include:

FST "first" removes and places the contents of the first entry of list, L, into the ENT[] array. Any expression with the value of 1 can also be used.

LST "last" removes and places the contents of the last entry of list, L, into the ENT[] array. Any expression with the value of 2 can also be used.

KEY "key" uses the current value of ENT[0] to search for a match in list, L. If a match is found, KEY removes the entry from list, L, and places its contents in the ENT[] array. An expression with value of 3 can be used.

LN{X} gives the natural log of positive-valued expression X.

MAX{X;Y} gives the maximum of expressions X and Y.

MIN{X;Y} gives the minimum of expressions X and Y.

MOD{Y;X} gives the integer remainder when the value of expression Y is divided by the value of expression X.

NOR{M;S} generates a pseudo normal random variate with mean given by the expression M and standard deviation given by the expression S.

PAUSE{} halts the execution of a simulation run. This is useful for debugging, along with the Execute and Remove options in the Event Vertex dialog.

PI is defined to be 3.14159 and is used with the SIN{} and COS{} functions.

PUT{O;L} places all of the current contents of the ENT[] array in the list identified with the integer expression or constant L according to a ranking criterion given by the option O. More details on this and the companion functions, GET and CGET, are presented later. The options include:

FIF "first-is-first" places the current contents of the ENT[] array after the last entry of list L. An expression with the value of 1 can also be used.

LIF "last-is-first" places the current contents of ENT[] before the first entry of list L. Any expression with the value of 2 can also be used.

INC "increasing" places the current contents of ENT[] in list L, by increasing ranking of the value of ENT[RNK[L]], where RNK is the ranking array for the lists. An expression with the value of 3 can be used.

DEC "decreasing" places the current contents of ENT[] in list L by decreasing ranking of the value of ENT[RNK[L]]. Any expression with the value of 4 can also be used.

EVN "even" places the current contents of ENT[] ranked by increasing values of ENT[0] and then by attribute ENT[2] to break ties. This function is actually used to place events on the future events list, where ENT[0] is the time of the event and ENT[2] is the event execution priority used to break time ties. Any expression with the value of 5 can also be used.

RND uses the multiplicative congruential pseudo-random number generator, lcg(), to produce approximately "independent uniformly distributed" numbers strictly greater than 0 and less than 1. You choose an initial seed to start the generator in the Run Options dialog box.

SET{N} resets all variables to zero and the random number seed to the integer N. If no argument is given, the original random number stream continues to be used. This can be used to batch several runs in a designed experiment into a single run.

SIN{X} gives the sine of the real valued expression X.

STS{X} is the area under the standardized time series for the traced variable X.

TAV{X} is the cumulative time average for the traced variable X.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

TRI{X} generates a pseudo triangular random variate between 0 and 1 with a mode given by the expression X with 0≤ X≤1.

VAR{X} is the cumulative variance for the traced variable X.

7.2 Reading Data (and Code) from Your Disk

The DISK{} function is used to read data files. This function has two arguments: The first argument is the full name of the data file (including the drive and path if necessary), and the second argument is an integer-valued expression telling which entry is to be read. When the index is zero, the file is read sequentially. If the end of a file is reached, reading data continues at the beginning of the file; this is called wrapping around a file. The data file should contain numbers or expressions separated by at least one blank. Extra spaces and ends of lines are ignored.

Use an ASCII text editor such as Notepad (found in the Accessories file in Windows) to create your data file. If you use a word processor, save your file in "text" format since its default data format will likely not work. (Be careful to avoid "hidden DOS" file extensions - your text editor or word processor might save the file data.dat as data.dat.txt or data.dat.doc without you knowing it - then SIGMA will not be able to find the file!). Type in your values or expressions for the data you have collected. Each entry needs to be separated by at least one space, not a comma. You can use variable names or expressions, but they must have values at the time they are read by your model. Comment lines can be placed in data files by starting the line with //. Placing comments at the end of a file will speed processing. Note: In C, DISK does not read comments or expressions.

Use the Save command (under the File menu) to save your data. (i.e., JUNK.DAT). Make sure to note where the data file has been saved. You can view this output file in SIGMA by selecting the Open/Text command under the File menu. Select *.DAT or simply type in the name of the file. A new window with your file name will appear. You can edit the data from this window.

Some examples: a data file, named DATA, on your disk might look like the following:

```
.06  5  4.0  .3  2.  1
```

This is equivalent to the file

```
 .06  5  4.0
.3     2.  1
```

For a real-valued variable, X, eight successive evaluations of

```
X=DISK{DATA;0}
```

would in turn assign the eight values of 0.06, 5.0, 4.0, 0.3, 2.0, 1.0, 0.06, and 5.0 to X. After reaching the end of the file, reading would start again at the beginning of the file. The expression X=DISK{DATA;4} would assign a value of 0.3 to X since .3 is the fourth entry in the file. Finally, if I=4, X=DISK{DATA;2*I} would assign a value of 5.0 to X. (We wanted the 2*I=8th entry in the file, and wrapping around occurred.)

As another example: If the data file, QUE.DAT is

```
11 12 13 14 15
```

then the following would be the result of different state changes in event vertices

Q=DISK{QUE.DAT;4}	Sets the value of Q equal to 14.
Q=DISK{QUE.DAT;9}	Again, sets Q to 14 (wraps around at file end)
Q=DISK{QUE.DAT;0}	Reads file in sequence, wrapping around at the end.
Q=DISK{QUE.DAT;1+5*RND}	Q is randomly read from the set {11,12,13,14,15}

7.2.1. Reading Data from Tables

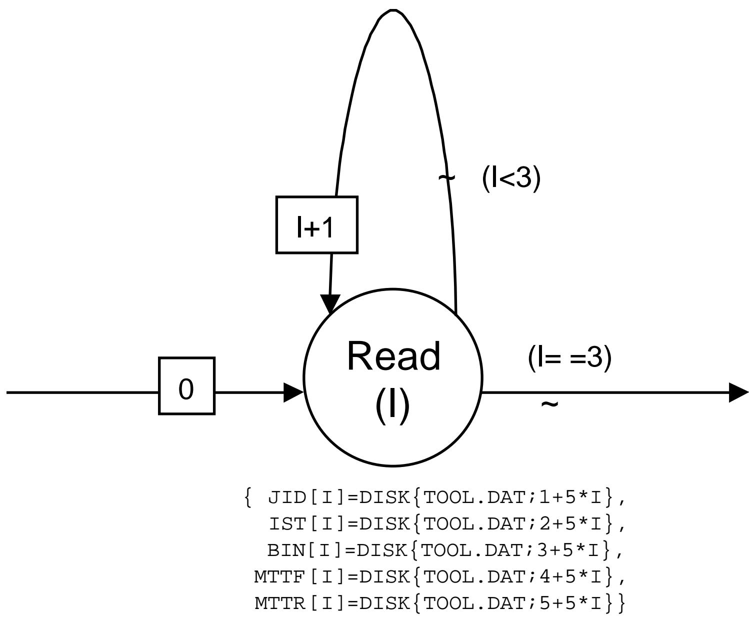
To read a data table, consider the file, TOOL.DAT containing the following text,

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
 1    3.5   4   14.4    15.2
 2    3.7   3   10.0    18.7
 3    2.5   2   15.4    16.6
 4    0.5   5    8.4    12.4
//JID   IST  BIN   MTTF    MTTR
```

This data can be read with the following Read event where the parameter, I, indicates the row of the table being read (See Figure 7.1). Note that there are five entries per row in the table and any comment lines are only at the end of the file!.

Figure 7.1: Reading the Data in Table, TOOL.DAT



7.2.2. Changing Code from Data files

Expressions can also be placed in data files and read using the DISK function. For example: if the file A.DAT is:

```
1 3 Q+4 5.6 3
```

Then the state change, Q=DISK{A.DAT;3} has the same effect as writing Q=Q+4. However, altering the right-hand side of the state change merely involves changing a data file (A.DAT) rather than changing the simulation code itself. Altering code is an extremely powerful and unique feature of SIGMA's DISK function, which does not translate to C. However, the other features of the DISK function do translate to C. Thus, the model and the data can operate together as a single object.

As a more elaborate example, consider the following state change:

```
QUEUE=QUEUE+DISK{SIZE.DAT;RND*5+1}
```

This statement will randomly select one of the first five entries in the file, SIZE.DAT, and add it to the value of the variable QUEUE. The second argument of this DISK function is an index that is a random integer (rounded down) between 1 and 5 inclusive. The above expression can be used, for example, to model arrivals of randomly sized groups of customers to a simulated service facility such as a bus station. The possible group sizes will be listed as five numbers in the disk file named SIZE.DAT. One of the five different group sizes will be selected at random each time the above state change is executed. Another way to do exactly the same thing is with the state change

```
QUEUE=DISK{Q.DAT;1},
```

where the first data entry in the file Q.DAT is the text expression

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

QUEUE+DISK{SIZE.DAT;RND*5+1},

which is the right-hand side of our previous state change.

7.2.3. Trace Driven Simulations

Sometimes, because a client demands it or to debug a model, you may want to drive a simulation model with data read from a file. If this data was generated by the system being modeled, the run is called a "trace driven" simulation. (More on this can be found in Chapter 9.) To run a trace driven simulation, simply place the DISK function as the edge delay as in Figure 7.2. It will read the Kth entry from the data file, simply increment the value of K in event A.

Figure 7.2: A Data Driven Simulation

You can, of course, randomize the reading of this data by replacing K with 1+M*RND where M is the number of entries in the data file. This can be rather slow as disk access is one of the slower computer operations.

7.3 Interactive Execution

ASK: ASK{E} allows you to change the value of the expression, E, during a run. SIGMA will prompt you for the change; you can enter any expression. Pressing [Enter] without entering an expression results in no changes being made.

Suppose you wanted to set a value for the variable, QUEUE, from the keyboard whenever a customer enters. The default expression for the state change when a customer enters is QUEUE=QUEUE+1. If you enclose the right-hand side of this equation with the ASK function, i.e., QUEUE=ASK{QUEUE+1}, you can enter any valid expression or number for the new value of QUEUE that you desire. This is illustrated in the model ASKDEMO.MOD. Of course, you do not need to be prompted by the ASK function to change a running model in SIGMA. Just double-click on an edge or vertex and enter the changes in the dialog box.

Another use of the ASK function is illustrated in FUNCDEMO.MOD. Here, the ASK function appears as an edge attribute on the edge from the DISKFN vertex to the NEXT vertex. The ASK function here allows you to override the value just read in from the disk, if you desire.

Besides allowing interactive simulation (gaming), the ASK function is very useful in debugging a model. You can try various values of variables while your model is running and see when the logic breaks down. For example in ASKDEMO.MOD, if the server is idle and you enter a value of zero when a customer ENTERs, you will have created a "phantom" customer arrival. This will subsequently cause the QUEUE to become negative when this customer (who did not join the QUEUE) later LEAVEs!

7.4 Bookkeeping Functions

SET: SET{N} resets your SIGMA variables to zero and the random number seed to N. If no argument is given, i.e., SET{}, then the random number stream continues with its next number. The simulation clock, CLK, is also set to zero. SET{} returns 1 if successful and 0 if it fails, so it can be used as an edge condition. (In a state change, it should appear only on the right hand side of an equation.)

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

SET is useful for running full experiments in batch mode. The model, FACTORAL, is a full 2 x 3 factorial experiment (normally requiring six runs) done in a single run with the SET function. SET{} does not translate to C. A very simple way to replicate is shown in the model INVENTRY.MOD, where several replenishment cycles are replicated; a self-cancelling edge with an edge condition of SET{} is all it takes to shut off the previous cycle.

CLK: CLK is the current simulated time, which is updated automatically when an event occurs. You cannot change CLK.

7.5 Mathematics Functions

PI: PI is defined in SIGMA as 3.14159. Thus, the Ith value of a cosine function at frequency F is COS{2*PI*F*I}. Here the frequency, F, is in oscillations per observation and is a positive number between 0 and .5; otherwise, the cosine and sine functions have arguments in radians.

RND: The function, RND, imitates the sampling of randomly-valued fractions called random numbers. Random numbers are assumed to be independent of each other and equally likely to occur anywhere on the interval strictly between zero and one; they are the key to modeling randomness. Random numbers are more properly called pseudo-random numbers since they are not truly random by any reasonable notion of randomness; they just look random.

For every run you make of a SIGMA model, you supply a "seed" (in the Run Options dialog box) for the random number generator that produces values of RND. Different seeds will produce different random number sequences.

Like the CLK function, you use RND as if it were an ordinary state variable. You also do not assign values to RND; they are given to you. We can use RND to model a random variable since every time RND appears in an expression it will have a different value.

Here are some examples: If X is defined as a real-valued state variable, then

X=2*RND and X=RND+RND

are not equivalent statements. X=2*RND takes a single random number and doubles it, producing a value for X that imitates a uniformly distributed random variable between 0 and 2. On the other hand, X=RND+RND takes two different random numbers and adds them. This produces a value of X that also falls between 0 and 2 but has a symmetric triangular-shaped distribution. The expression,

X=M+R*RND

will produce artificial values of X that are independent and uniformly distributed with a minimum value of M and a range of R, i.e., between M and M+R. If you define X to be integer-valued, it will be rounded down to the nearest integer from M to M+R-1. We will discuss random numbers in more detail in Chapter 9 where using multiple random number streams in a single model is explained.

7.6 Cycling using the MOD function

MOD: MOD{X;Y} gives the integer remainder when the rounded down value of expression X is divided by the rounded down value of expression Y. If X takes on successive values of 0,1,2,3,4,5,6,7,..., then MOD{X;4} repeats the cycle 0,1,2,3,0,1,2,3,0,1,2,3,....

One common use of the MOD function is to keep track of individual transient entities in a model (if you do not want to use SIGMA's PUT and GET functions). Array sizes are kept small by recycling indices. Say you want to keep track of individual customer waiting times in a simulated queue. Let WAIT[ID] be the waiting time of the customer whose identification number is ID. Assume that no more than M customers will be in the system at one time. When a customer arrives, make the following state changes in the corresponding ENTER vertex:

ID=MOD{ID+1;M},WAIT[ID]=CLK

When the customer departs, make the state change:

WAIT[ID]=CLK-WAIT[ID]

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Of course, the customer ID must be passed as an edge attribute throughout the graph to identify which customer is being served. If you recycle array indices in this manner, you must be careful not to reuse an index before it is released by a transient entity leaving the system.

It is important to check that the number of transient entities (customers) in the system never exceeds M. Even this will not insure that entity IDs are not duplicated. This is another advantage of resident entity models. This, in effect, means that we are modeling a capacitated queue with finite waiting room for, at most, M customers. If M is large, this is not a serious issue.

As another example of how the MOD function can be used, consider a bank where there are three tellers (numbered 0, 1, and 2). Each teller takes a five minute break once every hour, with a different teller going on break every twenty minutes. If the variable, NEXT, is the teller that is next to go on break, then the statement, NEXT=MOD{NEXT+1;3}, would make sure that the breaks rotated properly by generating the sequence, 0,1,2,0,1,2,0,....

As a final example, SIGMA was used to model a sensitive machining operation where a ten-second recalibration was needed for every five parts. If the normal processing time was seven seconds and N is the part number, the resulting processing delay is 7+10*(MOD{N;5}==0). The Boolean variable will be equal to 1 once every five parts, adding the ten-second recalibration time to the edge delay.

7.7 Using Ranked Lists

PUT and **GET**: Ranked queues occur whenever the order of service might differ from the order of entity arrival. SIGMA has two functions that make it very easy to model priority ranked queues and other types of lists: The PUT function puts entries onto lists and the GET function gets entries off from lists.

Two arrays, ENT[ ] and RNK[ ], are used by the PUT and GET functions. We will describe the purpose of these two arrays first.

The ENT[ ] array is used exclusively as a buffer for passing information about entities (typically customers) joining or leaving ranked queues using the PUT{} and GET{} functions described below. Attributes of individual transient entities (e.g. customers in a queueing system) can be assigned to the elements in the array ENT[ ]. For example, ENT[0] might be the customer arrival time, ENT[1] the class of service, and ENT[2] the amount of product to be purchased. The state change vector

```
ENT[0]=CLK, ENT[1]=CLASS, ENT[2]=DEMAND
```

might model the relevant attributes of a customer. The PUT function places the current values of the ENT[ ] array into a list, which can be thought of as a table with a new row being created by each time a PUT is made to the table. The GET function removes a row from this table and makes its contents available as the entries of the ENT[ ] array - one entry per column in the row just removed.

The array RNK[LINE] contains the index of the element of the ENT[] array that is to be used in determining a entity's position in the line designated by the integer, LINE.

The PUT{OPTION;LIST} function places the current contents of the ENT[] array in the LIST. The elements of the temporary buffer, ENT, are typically the values of attributes of entities that are joining the queue. LIST is a number, variable, integer expression, or function that identifies the queue to be joined. PUT{} options include:

FIF (first-is-first) or any expression with the value 1 - inserts the new entity (the current values of ENT[ ] array) after the last record on the LIST.

LIF (last-is-first) or any expression with the value 2 - inserts the new entity before the first record on the LIST.

INC (increasing) or any expression with the value 3 - the LIST is ranked by increasing values of ENT[K], where K=RNK[LIST] is the ranking entity attribute.

DEC (decreasing) or any expression with the value 4 - the LIST is ranked by decreasing values of ENT[K], where K=RNK[LIST] is the ranking entity attribute.

EVN (even) or any expression with the value 5 - when the values of ENT[0] for two entities are even, the tie is broken by increasing values of ENT[2], with remaining ties broken by "first-is-first".

GET{OPTION;LIST} removes a record from the specified LIST according to the OPTION chosen and places its contents in the ENT[ ] array. GET{} options include:

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

FST (first) or any expression evaluating to 1 - removes the first entry from the list and place its values in the ENT[ ] array for use in your model.

LST (last) or any expression evaluating to 2 - removes the last entry of LIST.

KEY (key) or any expression evaluating to 3 - removes the first entry of LIST (if any) whose value of ENT[0]matches the current value of ENT[0].

Both PUT{}and GET{}return 1 if successful and 0 otherwise. These functions should be used with care as part of an edge condition that might be false since PUT and GET are executed regardless. If used in a state change, they should appear only on the right-hand side of an equation, such as

QUEUE[N]=QUEUE[N]+PUT{FIF;N}.

This state change will increase QUEUE[N] (the length of the Nth queue) by 1 when the entity with attributes currently in the ENT[ ] array is put into this queue. The entity can be removed later (with attributes placed in ENT[ ]) with the state change

QUEUE[N]=QUEUE[N]-GET{FST;N}.

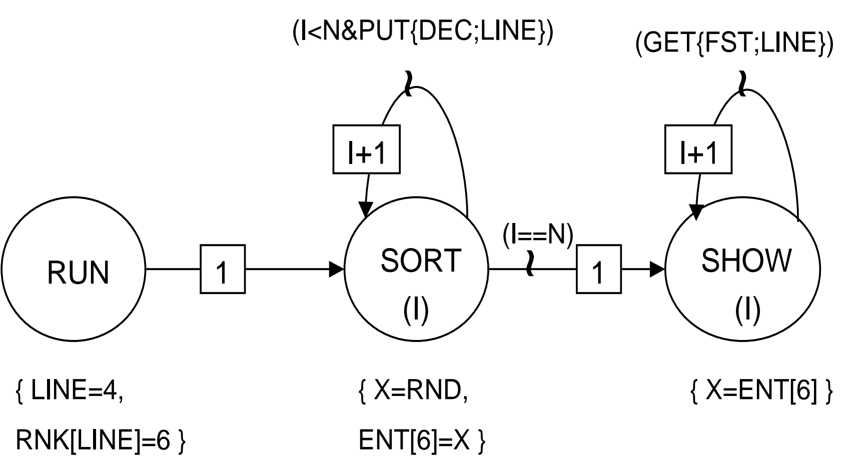
Redundant options are for clarity. For example, either pair (PUT{FIF;1} and GET{FST;1}) or (PUT{LIF;1} and GET{LST;1}) could be used for a first-come first-served queue.

Several sample models using SIGMA's ranked lists functions, PUT and GET, have been included.

7.7.1 Example: Sorting Data (SORT.MOD)

In the model, SORT.MOD, a SORT event will generate 100 random numbers (called X), placing them in decreasing order on a list. The SHOW event gets these numbers of the list sequentially and displays their values. This model is given in Figure 7.3.

Figure 7.3: Using PUT/GET Functions to Sort Numbers

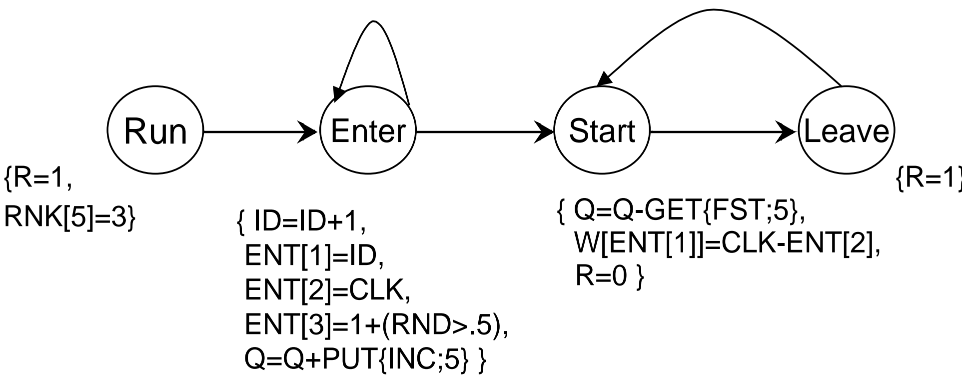


7.7.2 Example: A Priority Queue (PRIORITYQ.MOD)

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

PRIORITYQ.MOD models a queue with several classes of customers with different priorities. This model is in Figure 7.4 where the edge conditions and delays are omitted as they are identical to those in CARWASH.MOD. There are two types of customers, with 50% of the arrivals being in each class. (Note the Boolean expression to determine the customer class.) Arrivals are then PUT on the list in increasing order of their priority class. Whenever a service starts, the next customer in line is found using the GET function.

Figure 7.4: Modeling a Priority Queue. Note Queue 5 is ranked by ENT[3], which is customer priority (the value is either 1 or 2 with equal probability)



7.7.3 Example: Time-Constrained Processing (TIMEOUT.MOD)

Time-constrained processing is where two processes must be performed within a certain interval. Examples of time-constrained processes are common in semiconductor manufacturing, metal foundries, food processing, etc. where priming, pre-heating, or cleaning is required before performing a major processing step.

For this example, consider a heat and press sequence; parts arrive every t_a minutes and must be heated in a furnace (taking t_h minutes) before they can be pressed by a mold press machine (taking t_p minutes). The interarrival times and both processing steps are somewhat random. The sequence is time-constrained because there is a maximum cooling time limit of t_c after a part is heated during which the pressing step must start. If a part waits longer than t_c minutes after being heated before pressing starts, it must be returned to the furnace for reheating. We will identify each part waiting for the mold press with an ID number that is sequentially assigned after the part has been heated. F and P will denote the status of the furnace and press respectively (1=IDLE, 0=BUSY). QF and QP will denote the number of parts in the queue waiting for the furnace and press.

The event graph for this model, which is shown in Figure 7.5, is the complete simulation model. It might seem rather complicated at first glance; however, one of the most appealing features of event graph modeling is that the logic in each vertex, along with its exiting edges, can be verified in isolation. The graph naturally decomposes into vertices and sets of exiting edges that can be examined separately; the graph automatically ties everything together. We read the model by examining one vertex at a time along with its set of exiting edges; ignore the rest of the graph as you read each of the following paragraphs describing how each vertex in this model works. This model is called TIMEOUT.MOD.

The RUN vertex makes the furnace and press initially idle and requests a value for the cooling limit, t_c. The first part is then scheduled to ENTER the system.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

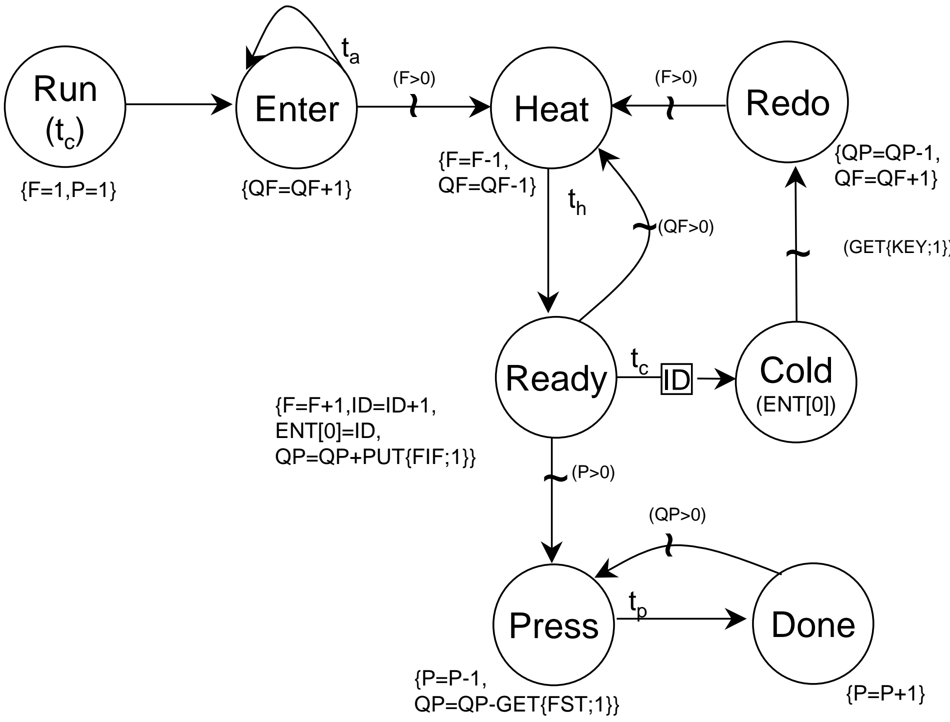
When a new part ENTERs the queue for the furnace, QF is incremented. If the furnace is idle (F>0), the part can be HEATed. The ENTER vertex also schedules successive new part arrivals.

When HEATing starts, the furnace becomes busy (F=0) and the number of parts waiting for the furnace, QF, is decremented. After a heating time of t_h, the part is READY for pressing.

When a part is READY for the mold press, the furnace is unloaded (F=1), an ID number is assigned to the part, and it is PUT into the queue for the mold machine, QP. If there are more parts waiting for the furnace, the next part can start to be HEATed. If the mold press is idle (P>0), then the PRESS operation can begin immediately. Cooling starts as soon as the part is READY for the press. If part ID does not start its PRESS operation before t_c, it will become COLD and need to be sent back to the furnace.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 7.5: Time Constrained Processing: `READY` puts jobs in Queue 1. Either the `PRESS` event will get them, or they will be `COLD` when the timer runs out in t_c time units.



When the `COLD` event occurs, the cooling limit for part `ID` has been reached. The condition on the edge from `COLD` to `REDO` checks if the cold part is still waiting to be pressed using the `GET` function with the `KEY` option. If this part is not still waiting in the mold press queue (list 1), then it must have already started its pressing step. In this case, the `GET` function will not find a match and returns a value of `0;` this makes the edge condition false, and the `REDO` event will not be scheduled. Note that the `ID` value of the part is passed from the `READY` vertex to the `COLD` vertex as an edge attribute. This `ID` value is placed in `ENT[0]` to be used as the match `KEY` in the `GET` function to see if that particular part is still waiting for the mold press when it became cold.

In the `REDO` vertex, if the cold part is still in the queue for the mold press (that is, part `ID` is still waiting in list 1), then it is removed from the press queue `(QP=QP-1)` and placed back in the queue for the furnace `(QF=QF+1)`. If the furnace is idle, then `HEAT`ing can begin.

The `PRESS` operation will `GET` a part out of the queue (if the `COLD` event has not already removed it) and make the press busy. After a pressing time of t_p,the part is `DONE`. If more parts are in the mold machine queue `(QP>0)`, the next part waiting (that has not cooled off) can begin its `PRESS` step.

The model, `TIMEOUTR.MOD,` models the same system without using any transient entities, using event cancellation instead. It is a faster model for this system and well worth studying; however, it is valid only if t_c is a fixed constant.

7.7.4 Example: A General Network of Queues or Jobshop

The model presented in this section illustrates many different aspects of event graph model building - `PUT` and `GET` functions, multidimensional arrays, `DISK` input, nested loops, branching, and event parameter passing. The model, while very simple, is simulating a more complicated system than those we have seen previously and may be a bit intimidating at first. Remember, event graphs allow us to concentrate only on one vertex or edge at a time; read the model logic by looking only at *one exiting edge* at a time.

Manufacturing and service systems that are composed of different processing centers where customers or jobs move from one processing center to another can be modeled effectively as networks of queues. Such systems are very common. In fact, several commercial simulation languages have been developed primarily to model queueing networks.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

In this section, we will start building our own queueing network simulator. The enrichments suggested by the exercises at the end of this chapter, if done correctly and efficiently, could result in a modeling tool that is as good or better than many commercial simulators. The key difference is that you will have complete control over your simulation program, so you can change or enrich it as you see fit.

In the material that follows, we will refer to jobs passing through different machine groups. For other applications (e.g., hospitals), substitute the appropriate transient entities (patients) and resident entities (departments) in the model description.

A common example of a queueing network is a production system called a *jobshop*. A jobshop has several different processing centers with one or more identical machines at each center. Often the machine centers are located in the same area. Different types of jobs have different routings through the jobshop. In this section, we will use a jobshop as our example of a queueing network. While we develop our model, keep in mind any changes you might want to make to simulate other types of queuing networks, such as patient flow in a health care system, baggage flow in an airport, or information flow through a bank - typically only the terminology and names of variables will change. A good jobshop simulation can come close to being a general "all purpose" discrete event system simulator.

7.7.4.1 `NETWORK.MOD`

The detailed discussion that follows refers to the model, `NETWORK.MOD`. A verbal event graph of that model is given in Figure 7.6, but your will need to open `NETWORK.MOD` on your computer to follow the details.

To make our model specific, we will consider a jobshop with five groups of different types of machines and identify each machine group with a particular value of the index, `G`. This jobshop will process three types of jobs. Each type of job will have a different routing and a different processing time. We will indicate the type of job with the index, `TYPE`. In this system, the resident entities are the machines and queues associated with each machine group. The transient entities are the jobs flowing through the system. Job processing times are modeled here with an Erlang distribution with different means depending on machine group and job type involved. This jobshop is the most advanced system modeled in Law and Kelton (1991).

We want to simulate systems with up to `MAXG` different machine groups. `MAXG=5` in our example, and we will number the machine groups `0` to `4`. A dummy sixth machine group (group `5`) is used to indicate that a job has left the shop. We will denote the number of idle machines in group `G` as `S[G]`which are initialized as parameters of the `RUN` event.

> *Comment:* If there were only one type of job in the system, we could get by with a model that includes only resident entities (the queues and machines). For each machine group, all machines are identical; therefore, we only need to keep track of the number of jobs waiting and the number of machines that are idle. As we know from our earlier models, pure resident entity models tend to be much faster and easier to understand than models that include transient entities. We can model a single job type easily (see `FLOWSHOP.MOD`). When there are multiple types of jobs, we still would not have to introduce transient entities into our jobshop model, but this involves some fancy bookkeeping beyond our current purpose of illustrating the use of priority queues. For model flexibility and efficiency, it is good practice to avoid using transient entities as long as possible; this can be done here by not drawing unnecessary distinctions between the jobs, just keep counts of the jobs in different states.

To make it easier to change input when running experiments, most of the specifications for our models will be read from disk files. Data driven models are much easier to use than models that must be updated each time a change is desired. Data input files sometime are called experiment files or experimental "frames". Chapter 11 explains how you can easily conduct large simulation experiments with many runs that use different input and output data files.

The successive steps in processing a job are called *tasks*; the integer, `TASK`, will be the position that a job is on its route. The routing for job type `0` is in Table 7.1. We use a mean processing time of -99 time units for pseudo-machine group 5 to make any error of using this group obvious.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Table 7.1: Job Type 0 Routing

TASK	Machine Group	Mean Processing Time
0	2	0.250
1	0	0.300
2	1	0.425
3	4	0.250
4	5=(exit)	-99

Comment: Note that jobs can visit a machine group more than once. A process where a job returns to the same machine group more than once is called a *re-entrant* flow. Re-entrant flow occurs where production involves building up layers of material as in automobile painting or semiconductor manufacturing.

The state variables we will be using are summarized in the Table 7.2.

Table 7.2: State Variables for Jobshop model

`MAXG`	Maximum number of machine groups
`G`	Index identifying each machine group
`S[G]`	Number of idle machines in each group
`Q[G]`	Number of jobs waiting for each group
`TASK`	Current task of a job on its route
`TYPE`	Type of job - determines routing
`ROUTE[TYPE;TASK]`	Machine group needed for each task
`MTIME[TYPE;TASK]`	Mean processing time for each task
`OUTPUT`	Count of the number of finished jobs
`R`	Random number to determine job type
`DELAY`	Cycle time - time a job spends in the system
`ARIV`	Arrival time of a job - to compute cycle time

Our model will have three different types of jobs. Each job type will have its own set of routings through the network and a different processing time at each machine group. Since the jobs waiting in a queue are not identical and a first-come-first-served dispatching policy is used, we must keep track of the order that tasks are waiting in each line. The number of job types can be increased without making the model any more complicated. In our example, we will have job types arrive according to the following probability distribution:

Probability {Type=0} = 0.3,
Probability {Type=1} = 0.5, and
Probability {Type=2} = 0.2.

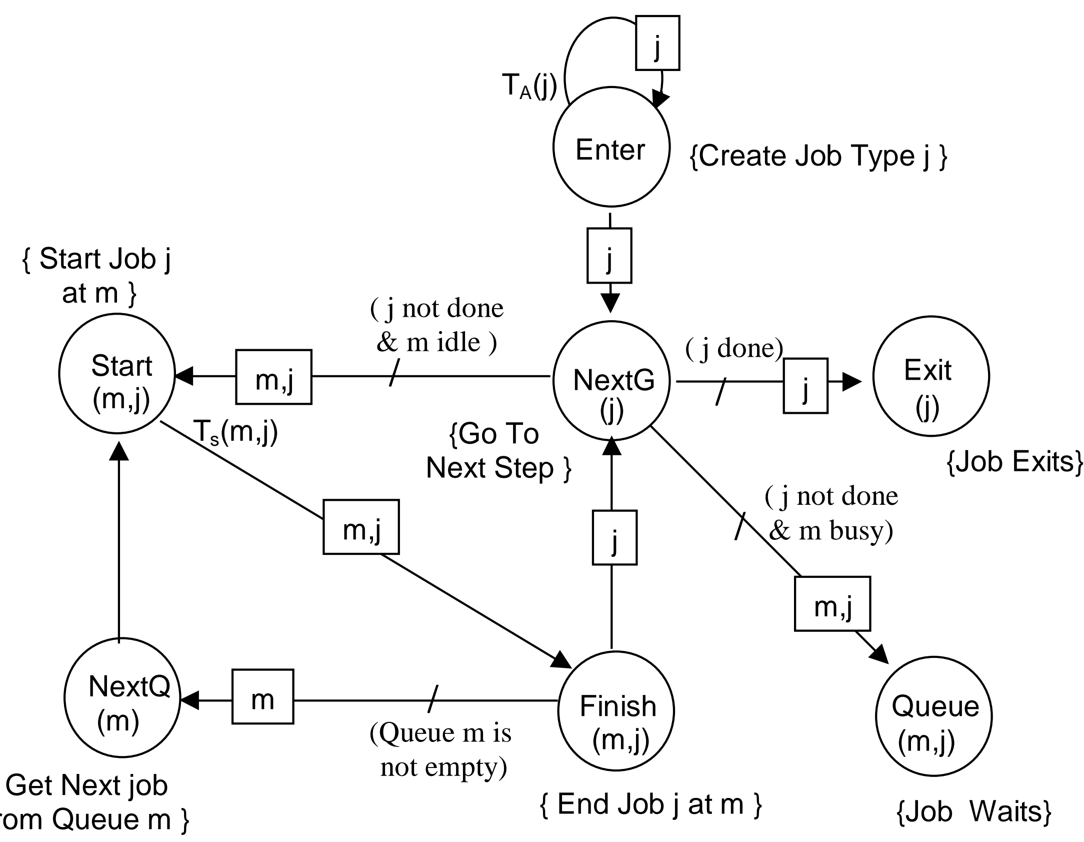
By ignoring the details and taking a high-level view of the event graph, we can see basically what is going on. The most distinctive feature of this graph are the two triangles in the center area, where two "activity" cycles can be seen. One cycle (`NextG, Start, Finsh`, and back to `NextG`) is followed by each job as it proceeds through the tasks on its routing. A job's task is incremented in the `Finsh` vertex and the machine group, `G`, changes whenever the job enters the vertex, `NextG`, to find the next machine group on its route.

The other activity cycle (`Start, Finsh, NextQ`, and back to `Start`) is the busy/idle cycle of each machine in a particular machine group. Around this cycle the machine group, `G`, does not change. The `TASK` changes each time the machine selects the next job in its queue at the `NextQ` vertex. Therefore, the upper cycle can be viewed as the activity of a transient entity (a job) moving between resident entities (from machine group to machine group). The lower cycle is the

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

activity of a resident entity (machine group) processing successive transient entities (jobs). We still do not identify particular jobs except by the TASK they are on, so this is basically a resident entity model.

Figure 7.6: Verbal Event Graph of General Jobshop, where Job Types, j are being Processed by Machine Types, m.



We will soon read the actual event graph for NETWORK.MOD, so please load NETWORK.MOD on your computer. This example is a good test of your comprehension of simulation modeling. Do not let the complexity of the graph discourage you; focus on only one exiting edges of each successive vertex at a time. The graph keeps everything else tied together; as we will see, this is a rather simple model of a very complicated system.

The two central cycles in the graph are where all the action occurs. The rest of the graph merely reads in data, computes waiting times, and maintains the queues. You can think of the job cycle and the machine cycle as being two "activities" that interact where they share a common edge (i.e., during processing). After a little practice, you should find that looking at the paths of temporary entities or the activity cycles of permanent entities on an event graph can tell you, pictorially, a great deal about how the system works.

We might want to collect job waiting times directly rather then use Little's Law (introduced earlier in Chapter 5). We would do this by adding transient entities to our model using the PUT{} and GET{} functions to manage the queues. The ENT[] array will serve as a buffer for the attributes of jobs that we wish to put into and get from the queues. We will define elements of this array as follows:

ENT[4] = type of job,
ENT[5] = next task for the job, and
ENT[6] = time the job entered the shop.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The other elements of the ENT array can be used for other job attributes. If we want to give some jobs priority over other jobs, we can set the values of the ranking job attribute with the RNK[ ] array and have priority ranking of the queue by job type or arrival time.

We explicitly designate the job type with the variable, TYPE, and designate each machine group with the variable, G. Job routings and timings are kept in two-dimensional tables. The tables, ROUTE[TYPE;TASK] and MTIME[TYPE;TASK], hold the machine groups and the mean processing times for each of the tasks, TASK, involved in processing each type of job, TYPE.

The total time spent in the system will be collected in an output variable called, DELAY. When a new job arrives, we mark its entry time with the variable, ARIV. Besides the use of the PUT{} and GET{} functions to manage queues, a major characteristic of this model is that the type of job, TYPE, its current task, TASK, and the time the job arrived, ARIV, are passed along the edges. The event vertices and the exiting edges for the new model.

Run Initiation events:

In the Run vertex, we will read initial values for some system constants and start our run. The state changes for this event are:

MAXG=DISK{ROUTES.DAT;0},Q[0]=0,Q[1]=0, Q[2]=0,Q[3]=0,Q[4]=0,Q[5]=0

The maximum number of machine groups is read from the data file, ROUTES.DAT and the initial queues are set to zero. The numbers of machines in each group are input parameters to the Run event. After the Run event, data for the routes will be read in the Input loop.

The Input vertex has a two dimensional nested loop reading in the ROUTE and MTIME tables by job TYPE and TASK. The data for these tables are in the disk files, ROUTES.DAT and MTIMES.DAT. The state change here is

```
ROUTE[TYPE;TASK]=DISK{ROUTES.DAT;0},
MTIME[TYPE;TASK]=DISK{MTIMES.DAT;0}
```

Once the data is read, the first job can Enter the system.

Job Flow Events:

The Enter event is where a new job arrives. Here we determine the job type with the state changes,

```
R=RND,    TYPE=(R>.3)+(R>.8)
```

To see how this works, recall that Boolean variables like (R>P) are set equal to 1 when they are true and to 0 whenever the condition is false. Assuming that RND is a true random number, the condition, RND<=.3, will be true 30% of the time, and both Boolean variables will be zero, making the TYPE=0+0=0. This causes 30% of our jobs to be type 0. Similarly, (.3<RND<.8) is true 50% of the time, so TYPE=1+0=1 for 50% of the entering jobs. Finally, (RND>.8) is true 20% of the time, making both Boolean conditions true. Thus, TYPE=1+1=2, for the 20% type 2 jobs.

After every occurrence of the Enter event, we loop to schedule the next job to Enter. We also schedule the NextG event to determine the machine group that is first on the job's routing. The Enter vertex passes the current clock time, CLK, into the variable, ARIV, along the edge to the NextG vertex. This sets the time that the job arrived in the system and will be used to compute how long it took to process.

The NextG vertex determines the next machine group on a job's route by looking up the machine group, G, in the ROUTE table,

```
G=ROUTE[TYPE;TASK].
```

If there are idle machines in group G the job can Start processing. If there are no idle machines in the next machine group, the job will join the queue in the JoinQ event by passing the type of job, TYPE, the current task, TASK, and the job's arrival time, ARIV, into elements 4, 5, and 6 of the ENT[] array, where they will be PUT into queue G. It G=MAXG, then the job is finished at a Leave event, which computes its time in the system.

The Start event is essentially the same as for our previous models of queues, it simply decrements the number of idle resources of type G.

```
S[G]=S[G]-1
```

The job arrival time, stored in the variable, ARIV, is merely passed through this vertex to the next vertex.

The Finsh event models the event where a task is finished at a particular machine group. This event causes another machine in group G to become idle and the task counter for the job to increment by one.

```
S[G]=S[G]+1, TASK=TASK+1
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

After every occurrence of the Finsh event, we determine the next machine group on the job's route in the NextG vertex by passing the current task and job type. Also, after the Finsh vertex, if Q[G]>0, jobs are waiting in the queue for the newly idle machine in this group, and we schedule the NextQ vertex where we select the next job waiting in the queue.

The Leave event computes the total time that the job spent in the system by subtracting its arrival time from the current clock time, CLK, with the state change,

DELAY=CLK-ARRIV.

Queue Maintenance Events:

The JoinQ vertex models the event of a job joining the queue for a particular machine group. This vertex has as its parameters, G, ENT[4], ENT[5], and ENT[6], which receive the values of G, J, T, and ARIV that were passed to it from the NextG vertex. The single state change,

Q[G]=Q[G]+PUT{FIF;G}

places these values for this job first in queue G and increments the value of Q[G] by the value, 1, returned from the PUT function.

The NEXTQ vertex models the selection of the next customer in queue G. The single state change,

Q[G]=Q[G]-GET{FST;G}

removes the data for the next customer and places it in elements 4, 5, and 6 of the ENT array and subtracts 1 (the value returned by the GET function) from the number of jobs waiting in Q[G]. The values of these elements of the ENT array are passed as edge attributes into the parameters, J, T, and ARIV, of the START vertex along with the machine group G.

Table 7.3 summarizes all of the state changes in the events in our general jobshop model. Note that this very complex system can be modeled with relatively few variables and simple state changes. As we will see in the next section: we can actually model this system with only two events using the powerful conditional CGET function.

Table 7.3: State Changes for a General Jobshop Model

Event	State Change
Enter	R=RND, TYPE=(R>.3)+(R>.8)
NextG	G=ROUTE[TYPE;TASK]
Start	S[G]=S[G]-1
Finsh	S[G]=S[G]+1, TASK=TASK+1
JoinQ	Q[G]=Q[G]+PUT{FIF;G}
NextQ	Q[G]=Q[G]-GET{FST;G}
Leave	DELAY=CLK-ARIV

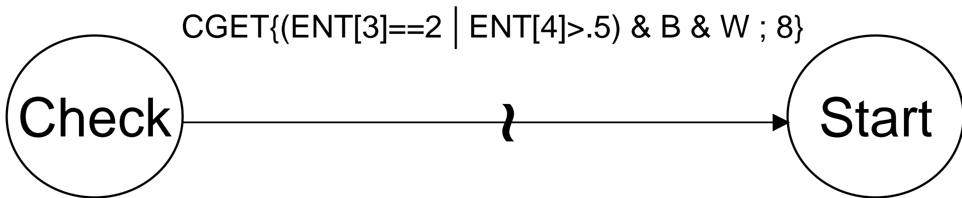
7.7.5 Example: Using the Conditional GET function, CGET

It is possible to get an entity (job or row) from a list (queue or table) conditional on the values of the attributes of the entity as well as on any other state variables. To do this use the CGET function. The form of this function is CGET{Condition;List} which gets the first entry in the list, List, where the Condition is true. This is useful

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

for example, if one wants to start the first job waiting in line 8, if any, with `ENT[3]=2` or `ENT[4]>.5`, provided the Buffer `B` is not full, and a Worker of skill `W` is Idle. This can be done with the event edge in figure 7.7.

Figure 7.7: Using the `CGET` Function as an Edge Condition

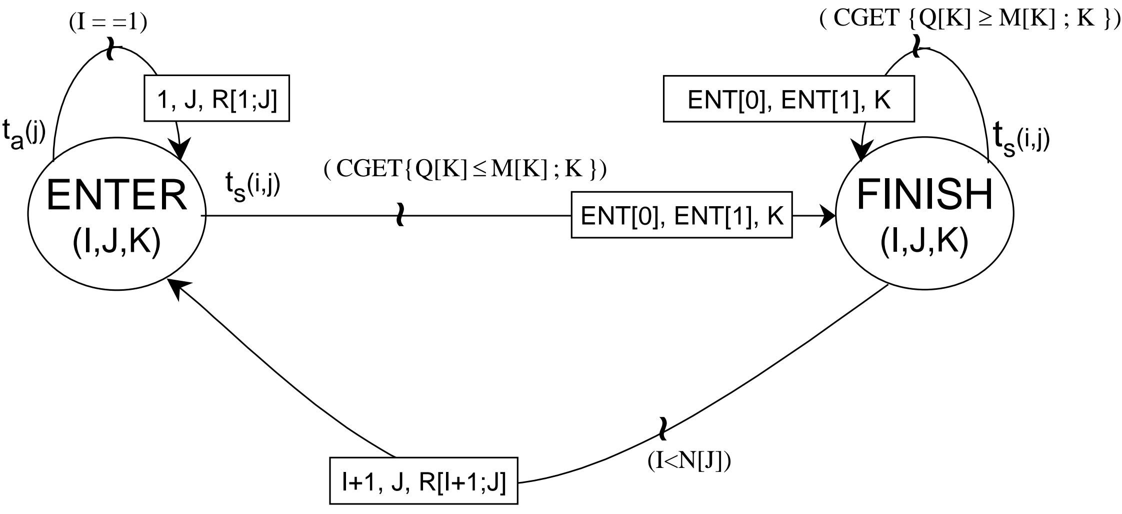


A general jobshop like that in the previous example can be modeled using only two vertices. To do this we have events for the jth job type at the ith step of N_j steps. Define the following state variables:

K	Number of types of processing resources.
J	Number of different job types, each with a different routing.
M_k	Number of idle resources of type k.
Q_k	Number of jobs waiting in queue for resource, k rank by increasing priority (RNK[k]=2).
$D_{j,k}$	Priority for job type j at resource k.
$t_a(j)$	Time between arrivals of type j jobs.
$t_s(i,j)$	Processing times at i^{th} step of job j.
$R_{i,j}$	The resource type needed by the jth job type at i^{th} step of N_j steps.

The jobshop model is given by the event graph in figure 7.8.

Figure 7.8: A General Jobshop Model using Conditional Gets, `CGET`.



Note that there are only two events with very simple state changes for this general model with multiple types of jobs being processed by multiple types of resources. The `ENTER` event computes job characteristics and `PUT`s it in a queue.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
ENT[0]=I,
ENT[1]=J,
ENT[2]=D[J;K],
RNK[K] = 2,
Q[K] = Q[K] + PUT{ INC ; K }
```

The `FINISH` event decrements the queue after being scheduled with conditional `GET`s from the queue

```
Q[K] = Q[K] -1
```

All the real work is done by the `CGET` function.

> Note: Having SIGMA automatically generate fast C code for a simulator is discussed in Chapter 11. Using the CGET function in C requires defining a simple function pointer as detailed in Section B17 of Appendix B.

7.8 Generating Random Variables

There are several functions in SIGMA that can be used to generate random variables. These include `BET{}` for beta variates, `TRI{}` for triangular variates, `ERL{}` for Erlang variates, `GAM{}` for highly skewed gamma variates, and `NOR{}` for normal variates. The use of these functions and the generation of other types of random input are discussed in Chapter 9 on input process modeling.

7.9 Statistical Functions

Several functions have been included in SIGMA for statistical analysis. These functions are primarily for model diagnostics and are not included in the C translations generated by SIGMA. Note that the variables used as arguments in these functions must have been used as traced variables for at least one previous run in the current SIGMA session. The more commonly used statistical functions included in SIGMA are as follows:

AVE{X}, which is the cumulative average of the traced variable, `X`,

VAR{X}, which is the cumulative variance of the traced variable, `X`,

TAV{X}, which is the time average of the traced variable, `X`, and

STS{X}, which is the area under the standardized time series for the traced variable, `X`.

The *cumulative* average simply counts the recorded values of a variable and averages them, whereas the *time* average takes into account how much simulated time the variable spends at a particular value. These functions can be nested in creative ways. For example VAR{AVE{X}} will be the variance of the average. This can be used, along with batching, to tell if the simulated run duration is long enough to obtain a desired relative or absolute precision of an estimator.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Table 7.4: Quick Reference to SIGMA Functions

Syntax	Operation	Input	Output	Models
ASK{X}	Displays the value of expression, X, during the run and allows you to change it	Any	any	ASKDEMO.MOD, FUNCDEMO.MOD
AVE{X}	Cumulative average of traced variable X	Any	real	IIDNORM.MOD
BET{X;Y}	Generates a Beta pseudo-random variate(p.r.v) with parameters X and Y	X,Y>0	real	
CGET{C,L}	Get first entry from list, L, where condition, C is true.	String, Integer	0/1	
CLK	The current simulated clock time	None	Real	
COS{Y}	Returns the cosine of Y (Y in radians)	Real	Real	FUNCDEMO.MOD
DISK{F;I}	Reads the Ith expression from the file F, to read sequentially set I=0 if I>number of entries, the reader will wrap around. The file name ,F, must include the disk and path if necessary.	String, Integer	Any	FUNCDEMO.MOD
ERL{X}	Generates an Erlang pseudo random variate with shape parameter X; an Exponential with mean M is generated using M*ERL{1}	Integer	Real	BANK2.MOD, PRIORITYQ.MOD
GAM{X}	Generates a Gamma pseudo random variate with fractional shape parameter X	0<X<1	Real	
GET{O;L}	Gets values for the ENT[] array from list L with option O: O=1 or FST (first), O=2 or LST (last), O=3 or KEY (key: match ENT[0])	String or int, integer	0/1	SORT.MOD, PRIORITYQ.MOD, NETWORK.MOD
LN{Y}	Returns the natural log of Y	any	Real	FUNCDEMO.MOD
MAX{Y;X}	Returns the maximum of X and Y	any	Real	FUNCDEMO.MOD
MIN{Y;X}	Returns the minimum of X and Y	any	Real	FUNCDEMO.MOD
MOD{Y;X}	Returns the integer remainder of Y is divided by X	integer	Integer	FUNCDEMO.MOD
NOR{M;S}	Generates a Normal (M,S^2) pseudo random variate	any*	Real	IIDNORM.MOD
PAUSE{}	Halts run when executed, for logic checking in high-speed mode	none	0/1	
PI	Predefined constant = 3.14159			
PUT{O;L}	Puts the present ENT[] array on list L according to option O: O=1 or FiF (first is first), O=2 or LIF (last is first), O=3 or INC (increasing by ENT[RNK[L}]), O=4 or DEC (decreasing by ENT[RNK[L]]), O=5 or EVN (breaks even ties)	(L) integer	0/1	SORT.MOD, PRIORITYQ.MOD
RND	Generates a Uniform (0,1) pseudo-random variate	none	Real	CARWASH.MOD
SET{X}	Sets all variables (including CLK) to zero and sets the random number seed to X; with no argument, the original stream continues	integer	0/1	FACTORAL.MOD, INVENTRY.MOD
SIN{Y}	Returns the sine of Y (Y in radians)	real	Real	FUNCDEMO.MOD
STS{X}	Area under the standardized time series for the traced variable X	any	Real	IIDNORM.MOD
TAV{X}	Time average of traced variable X	any	Real	IIDNORM.MOD
TRI{X}	Generates a Triangular p.r.v. on (0,1) with mode X	0<X<1	Real	
VAR{X}	Cumulative variance of traced variable X	any	Real	IIDNORM.MOD

Specify real parameters using a decimal (i.e. 3.0 instead of 3) to avoid problems in C translation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

7.10 Exercises

Exercises identified as mini projects are more extensive and may take considerably longer than the typical exercise.

7.10.1 A Priority Serving System

(a) The system administrators of a mainframe computer have devised of method of partitioning processing time. Every job submitted to the mainframe is assigned a priority: urgent and not urgent. When two jobs have the same priority, the first one submitted is processed first. However, when two jobs are of different priority, the job with the higher priority is always processed first. Urgent jobs arrive at a rate uniformly distributed between 3 and 5 hours and have required processing times uniformly distributed between 1 and 10 hours. Not urgent jobs arrive at a rate uniformly distributed between 30 minutes and 1 hour and have processing times uniformly distributed between 15 minutes and 2 hours. Model the mainframe queue assuming every job, once it is started, is finished without preemption.

(b) Now model the mainframe queue where jobs can be pre-empted by priority. In other words, if a job is submitted to the mainframe while a job of lower priority is processing, the lower priority job is discontinued until the job of higher priority finishes. When a pre-empted job returns to the mainframe, it does not need to be entirely reprocessed. It is processed only for the required time remaining when it was pre-empted. Use the `PUT` and `GET` functions in modeling this system.

7.10.2 Empirical Distributions

How can the function, `DISK{FILE;I}`, be used to generate a random variable from an empirical distribution function with data in the disk file, `FILE`? (Hint: Consider a random index, `I`.)

7.10.3 Condensing Events

Use pre-emptive event execution (edge delays equal to * to "condense" several event vertices into a single event) to make the simulation, `NETWORKR.MOD`, run as fast as possible without changing the output. Be careful about using * too often and causing a stack overflow as explained in the text.

7.10.4 Worker Assignment Problem

In the model, `TWOQUEUE.MOD`, there are 2 servers assigned to a constant time set-up operation followed by 6 workers assigned to a random time processing operation. Is this a good assignment of the 8 workers? Experiment with the model to determine an optimal allocation of the 8 workers, assuming that all of the workers are equally skilled and paid the same amount. Justify your answer (how do you define "optimal" and why?) as well as the way you designed your experiment.

7.10.5 Execution Priorities

Look at the model, `TWOQUEUE.MOD`. This is a model of two multiple-server queues in a series, where a constant time setup operation is followed by a random duration processing operation. The first set of servers has a constant service time of 5 minutes. The second set of servers has a random service time with an Erlang(1) (i.e., exponential) distribution with a mean of 4 minutes.

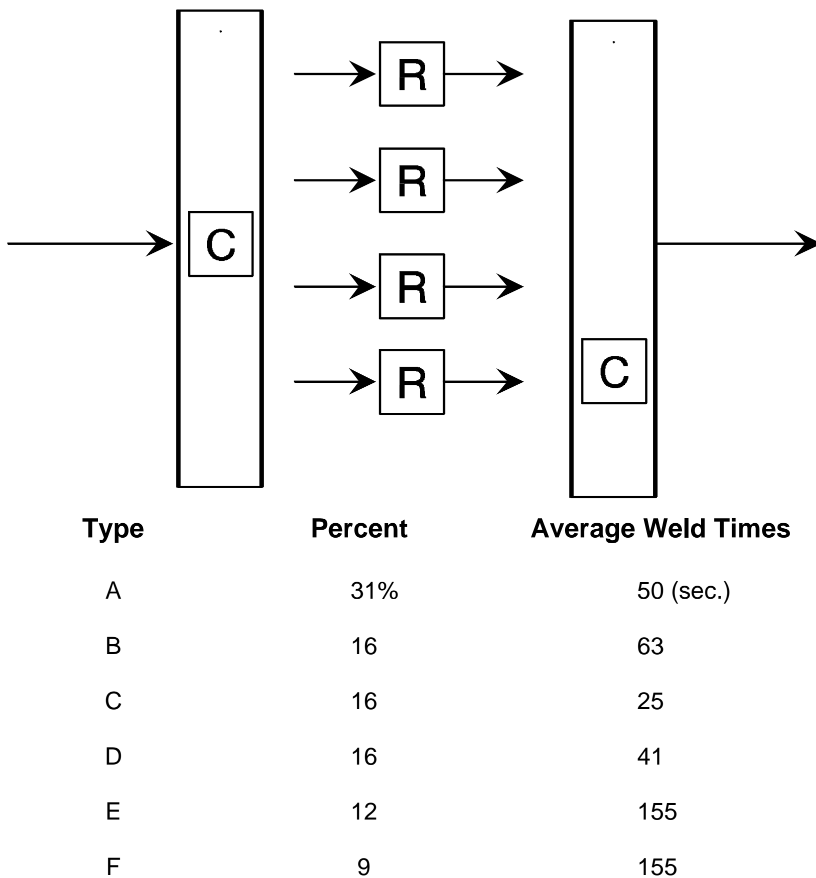
(a) Click on the double edge between the `STRT1` and `LEAV1` vertices. Change the execution priority on the edge from the `LEAV1` vertex to the `STRT1` vertex from `3` to a `5`. Notice that `QUEUE[1]` goes negative before time `60`. Why?

(b) Change the execution priority on the self-scheduling edge from `STRT1` to `STRT1` from `1` to `6`. Notice that the number of idle servers for queue `0` goes negative right away. Why?

7.10.6 Welding Line (Mini Simulation Project)

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The following system is used to weld hoods to automobiles. There are several parallel welding robots. The cars pass by the robots on conveyor lines. Transfers between lines are done using cranes that run north and south on fixed tracks. A schematic of the system for four welding robots is given below.



Type	Percent	Average Weld Times
A	31%	50 (sec.)
B	16	63
C	16	25
D	16	41
E	12	155
F	9	155

The input rate is 93 jobs per hour. There are six types of automobiles produced at this plant. The average welding times and percentages for each type of car are given below the figure. The time it takes for the crane to pick up a car or release a car is 8 seconds. For simplicity we consider only an approximation to the effects of crane acceleration. The first section of track traveled takes 9 seconds and every additional section traveled without stopping takes 6.5 seconds. A section of track is between each of the loading or unloading points on a crane track. For example, the crane takes 15.5 seconds to travel two sections of the track without stopping.

Cars must leave the welding operation in the same order that they enter it since components added later on in the assembly process must be for the right car. Set-up times for some of these later operations are very large, thus, requiring that set-ups start before cars arrive. The last crane in the schematic above is to sort the cars as they come out of the hood welding operation.

Write a simulation program for this system. For now, consider each of the times to be deterministic. Your model should allow up to three cranes per track and up to six parallel robots. You may use any computer language that you wish for this assignment. Flesh out your model and translate it to C to fill in the details (see Chapter 11). Use "entities" and the `PUT` and `GET` functions.

In the automobile production sub-system simulation model, add 33 seconds to each of the average welding times given. Now make the welding times uniformly distributed with a range of 20% of the mean. All confidence intervals are to be at the 90% level.

(a) Run the simulation of the automobile production system five times for 200 cars each. Use independently selected pseudo-random number generator seeds for each replications. Compute a confidence interval for the mean throughput rate of the system (rate that cars exit).

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

(b) Run the simulation for one run of 1000 automobiles. Compute confidence intervals for the mean throughput rate using the batched means method with 5 batches.)

(c) Comment on the above experiment, including identification of possible sources of error.

(d) Given that there are spaces for 15 automobiles in the system, decide how these 15 buffer spaces should be allocated. Justify your design and give confidence intervals for the throughput rate.

7.10.7 A Barge Unloading System (Mini Simulation Project)

River barges arrive at a warehouse carrying "piggyback" truck trailers mounted on flatcars. These flatcars are to be unloaded and mounted on railroad flatcars that will be taken away by trains. There are 4 trailers on each barge. The time between barge arrivals has a uniform distribution between 3 and 5 hours. A barge must wait for a single berth before it can dock. A single crane is used to move the trailers from the barge to the warehouse and from the warehouse to the flatcars. After being unloaded from the barges and mounted on flatcars, the trailers are taken to a large rail yard where they are assigned to trains according to their destination.

Once a barge has docked, 4 spaces in a warehouse must be free in order for unloading to start. There are 20 trailer spaces in the warehouse. The time to unload a barge is uniformly distributed between 1 and 3 hours. A train arrives at the warehouse to collect the waiting trailers. The time between train arrivals has a uniform distribution between 3 and 7 hours. The train has between 6 and 12 empty flatcars when it arrives. It takes 30 minutes to load each trailer onto a flatcar.

(a) Develop a resident entity simulation model that will keep track of the lengths of all important queues and utilizations of the warehouse space, berths, and crane. Clearly state any assumptions you are making in your model. Run this model for a time of 20 (hours) and trace the number of trucks unloaded. Run the model for 100 hours in High Speed mode and use the output file see how many trucks were unloaded from the barges. What is the major bottleneck: the berth, number of available flatcars, warehouse space, the crane?

(b) Consider the purchase of an automatic crane that cuts the barge unloading time by 50%. Model this (reduce the delay time between the start unload event and the end unload event). Run the new model for 100 time units and observe (from the output trace) the number of trucks unloaded. Do not create any new state variables.

(c) Add the necessary (two) event vertices that permit the crane to shut down every 10 hours for 1 hour of preventive inspection/maintenance. The crane will finish what it is doing before undergoing inspection.

(d) Eliminate the inspection/maintenance procedure and have the automatic crane break down every 8 to 12 hours (uniformly distributed). Have the repair time be uniformly distributed between 0.5 and 1.5 hours. If the crane is busy when it breaks, the current unloading operation must be restarted from scratch (use a cancelling edge). Run the simulation for 100 hours and compare with the systems in parts (a), (b), and (c). (This problem was suggested by G. Samorodnitsky.)

7.10.8 Buffer Capacities

In the model, NETWORKR.MOD, assume that each machine group has a finite capacity queue. For machine group, G, no more than B[G] jobs can be waiting. If the next queue for a job is filled, that job does not free its machine; it is blocked. Enrich your model to include this constraint. Be careful not to send two parts to a queue that only has space for one; when the second part arrives, it will find the queue full! (Hint: Reserve space in the next queue before freeing the current machine.) Assume that B[G] is 4 for all machine groups, and run at least 20 jobs through your model. Debug your model.

7.10.9 A Layout (Mini Simulation Project)

Using NETWORKR.MOD as a starting place, include the transit times between the various machines. Assume the job transit data is deterministic (conveyors?) and is provided in an array called TRANS(I,J), which is the time it takes to travel from machine group I to machine group J. The transit times are proportional to the distance between machine groups (state assumptions you are making). What data structures in C could be used effectively in this model? (Hint: The ENTITY structure). Tell how the structures will be used. How is your model improved with these structures (if at all)?

(a) Translate your model to C.. Use this simulation (or your model) to help determine an efficient high-level layout for a jobshop (i.e., the shapes, sizes, and locations for the different machine groups. Each machine takes 16 square yards of

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

space (including space for the job being processed and aisle space). Each job waiting in the queue takes 4 square yards of space. Transit between machine groups is by overhead crane, so travel is rectilinear not line-of-sight. Transit time between machine groups is 5 seconds per yard traveled between the head of each input queue. Write a report that explains and sells your design.

(b) Assume that you will make $10 for each type `1` job completed, $20 for each type `2` job completed, and $30 for each type `3` job completed. You may sequence the jobs in each queue in any manner you see fit. With the machines specified in the text, lay out the factory to maximize the rate that income is generated and minimize the size of the factory.

(c) To help solve this problem, design and run experiments to decide how to allocate waiting space for work-in-process (be creative). Explain the rationale for your experimental strategies: How did you initialize the system? Determine run lengths? Use variance reduction techniques? Compute confidence intervals?

7.10.10 Processing Priorities

Assume in `NETWORK.MOD` that completed type `1` jobs are worth twice as much money as type `2` jobs, which in turn are worth three times the money as type `3` jobs. It has been suggested that the jobshop operate by priority. The obvious scheme is to process waiting type `1` jobs before waiting type `2` jobs, with waiting type `3` jobs to be processed last. Make the necessary adjustments to your model for priority job processing. (Use the `PUT` and `GET` functions in SIGMA.)

(a) Make 5 runs of 8 hours each with the priority service rule and 5 runs of 8 hours with the current FIFO rule (a total of 10 one-day runs). Use independent streams for each run and estimate a confidence interval for the difference in total income for the day.

(b) Repeat this experiment using common seeds for each pair of runs (with and without the priority rule). Use independent seeds for different pairs of runs. Again estimate a confidence interval for the difference in total income for the 8 hour day.

(c) Comment on the differences between parts (a) and (b). What might be wrong with this experiment? Do you recommend the priority processing rule? Do you know a better one?

7.10.11 Motor Vehicles Department (Mini Simulation Project)

A State Motor Vehicles office has three clerks on duty. The office opens at 10:00 A.M. and closes its doors at 5:00 P.M. Customers already inside the office after closing are served. Between the hours of 10 and 12 (noon), walk-in customers arrive according to a Poisson process at a rate of $\lambda(t\text{-}10)$ customers per minute, where t denotes time in hours. Between the hours of 12 and 13 (i.e., over the noon hour), customers arrive at a constant rate of 2λ. Finally, between times 13 (1:00 P.M.) and 17 (5:00 P.M.), the arrival rate is $\lambda(17 - t)/2$.

Clerks process customers according to a uniform distribution between 0.5 and 2 minutes per customer. Occasionally a customer will call on the telephone and one of the clerks will take the call (possibly interrupting service for a customer). The time between phone calls is distributed as an exponential random variable with a mean of 5 minutes. It typically takes half the time to serve a call as to serve a walk-in customer. Every 20 minutes, one of the clerks is due for a ten minute break on a rotating basis (one break per hour for each clerk).

(a) With at most 10 runs of the model give an estimate of the capacity of the system as a function of the demand constant λ. Use whatever system performance measures you think are reasonable and justify them. Make and carefully state any additional assumptions about the system that you need; be sure to offer some justification for each assumption. A run is for a single day of operation. Carefully plan your experiment and give the rationale for your experimental design.

(b) As an aid in contract negotiations, consider the effect of adding an additional clerk or cutting the amount of break time taken by each clerk.

There is a plan to add a drive-up window to the office. This window is to be serviced by the same clerks that serve walk-in customers. Drive-in customers arrive according to the same pattern as the walk-in customers. Drive-in customers will balk if there are 3 or more cars waiting for service at the drive-in window. Of these balking drive-in customers, 80% will

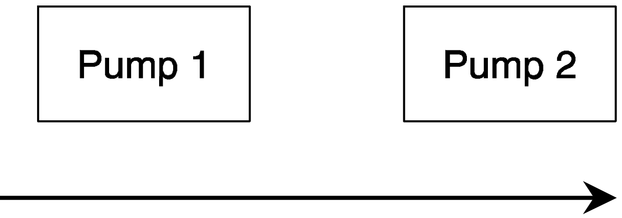
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

try to park their cars in the parking lot and walk inside for service. Once a former drive-in customer parks and walks inside, s/he will not balk unless the line has more than 10 customers in it. The parking lot has space for 10 cars. Customers desiring parking in a full lot will wait only 30 seconds before balking. Assume that it takes between 2 and 3 minutes (uniformly) for a customer to travel from the lot to the office and vice versa (including time to park).

Enrich you simulation to include the possible drive-in window. Use the your model to advise the State as to what they should do (hire more clerks?, build the drive-up window?, increase the size of the parking lot?, etc.).

7.10.12 A Turnpike Gas Station

A toll road gas station has 2 pumps in tandem but only 1 access lane. Cars arrive only from the left at intervals that are spaced *T* minutes apart. Pumping gas and paying takes *P* minutes. If both pumps in a lane are free when a customer arrives, the customer will use the "down stream" pump. A car at pump 1 cannot pass a car at the pump in front of it even if it has finished.



(a) What are the resident entities in this system? What are their attributes? What are the transient entities and their attributes? Describe, in words, the resident entity cycles and transient entity path(s). List any additional assumptions you are making about this system? Give an event graph model for the resident entities in this system. At a minimum, your graph could be used to measure the utilization of each pump.

(b) Create a SIGMA model of the turnpike gas station. Modify the problem so that if there are four or more cars in line when a customer arrives, the customer will go on to another station. Run you model until 20 cars have been served and analyze the queue size and utilization of each pump (charts, numbers, intuition, etc.). Start the system empty. Assume that car interarrival times are uniformly distributed between 1 and 5 minutes and service times are between 1 and 7 minutes.

7.10.13 A Turnpike Gas Station (continued) (Mini Simulation Project)

Enrich the preceding turnpike gas station model to include two access lanes, one on either side of the pumps. Potential customers arrive from the right at intervals that are randomly spaced. Assume that customer interarrival times are uniformly distributed between 2 and 6 minutes when coming from the left and uniformly spaced between 2 and 4 minutes when coming from the right. Service times are between 3 and 5 minutes at each pump.

(a) Customers cannot pass cars in front of them even if they have finished. Each pump can pump to only one car at a time in either lane and cars cannot drive around to change lanes. If there are more than four cars waiting in a lane, customers arriving from that direction will not stop. Draw an event graph of this system that can be used to study the resident entities (pumps and queues). Carefully justify your event graph model; do not just tell what you did and how, include why!

(b) Create a SIGMA model of your graph and print the English translation (`*.ENG`) and the event graph for your model. Run your model and discuss the output (what kinds of questions can you answer with your model?).

(c) Using the `PUT` and `GET` functions, collect waiting time data for customers in each lane. Present and discuss histograms and time plots of the waiting time data.

(d) Using event parameters, tell us how you would enrich your model to include several service islands.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

(e) Consider two service options: "full" service where service times are between 3.5 and 4.5 minutes at each pump and "self" service where service times are uniformly between 1 and 7 minutes at each pump. If both pumps are free when a customer arrives, that customer will use the "down stream" pump only half the time unless there is a "full" service attendant on duty. Create a SIGMA model of this system that can be used to study these two alternatives. Run your model under each alternative for 1 simulated hour. Write up your experiment and recommendations in less than five pages. How much can you afford to pay the extra attendant needed for the full service option (in terms of profit from lost sales)?

7.10.14 Parts Assembly

In an assembly operation, machines `A`, `B`, and `C` make parts that are joined together by machine `D`. It takes 3 parts from machine `A`, 2 parts from `B`, and 1 part from `C` for each assembly operation at machine `D`. All processing times have exponential (Erlang{1}) distributions. The mean processing times are input to the simulation as parameters of the first vertex. Build a simulation model of this simple assembly operation. Run your model for 10 complete assembly operations where the mean processing times for machines `A`, `B`, `C`, and `D` are `0.1, 0.2, 0.3,` and `0.4`. Which machine appears to be the bottleneck machine in this operation? What if the machine `D`'s processing time were reduced to 0.05 minutes? Translate your model to C or Pascal and run it until 1000 assemblies are finished. Does your bottleneck machine change once the system has warmed up?

7.10.15 The Texas Ferry Service (Mini Simulation Project)

Along the Texas Coast, an east-west four-lane highway was built to promote the tourist trade to the Padre Islands. Unfortunately, this highway must cross a wide channel used by large oil tankers. Providing free automobile ferry service is considered to be an alternative to building a bridge. There is room for up to 6 loading/unloading berths on each side of the channel. Whenever a tanker comes through the channel, the ferries have to wait until it passes. Evaluate the ferry option. Do a sensitivity analysis to various levels of demand. (How much can we spend on a bridge? How bad can it be?) Hint: Consider modeling aggregate arrival processes; e.g, ferry-loads/hour for the different size ferries.

Relevant data is as follows: Each berth costs $2,500,000 to build. There are three sizes of ferries that can use the berths:

Type 0 holds 25 cars and costs $1,500,000,
Type 1 holds 50 cars and costs $3,250,000,
Type 2 holds 100 cars and costs $5,500,000.

On Friday afternoons during a two-hour peak period, it is estimated that cars will arrive roughly according to a Poisson process with a rate of 20 per minute from the east (Corpus Christi toward the Padre Islands). The time between arrivals is thus `20*ERL{1}`. On Sunday afternoon over a four-hour period, roughly the same number of cars will return to Corpus Christi. At other times (during the tourist season) traffic in both directions is expected to arrive at a rate of about 6 cars per minute.

The ferries take between 12 and 18 minutes to cross the channel (depending on the tides). Tankers come by according to a Poisson process at a rate of 1 every half-hour, and it takes 8 minutes for a tanker to pass the ferry lanes. Cars can be loaded onto a ferry at a rate of 3 per minute and unloaded at a rate of 5 per minute.

Note: They decided to use ferries instead of build the bridge. Five berths on each side were built and 6 medium-sized ferries operate during the peak periods. Do you think this was a good decision? What factors, in addition to cost, might have guided their decision?

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

8

Building Animations

Animations created with SIGMA are fundamentally different from those using other simulation animation software. Most simulation modeling environments have a separate program for the simulation and an add-on program that does the animation; in SIGMA the simulation and the animation are identical. The event graph model becomes the animation. Animating a SIGMA model is extremely simple; you are limited only by your imagination and the speed and memory of your computer. In the tutorial that follows, we will develop a simple animation, use it to find an error in our model, and then fix the error.

8.1 Perspective and Basic Principles

Animations are one of the more entertaining aspects of simulation modeling. However, it is important to keep them in perspective. Animations are most useful during model development and testing. Their main strength is in helping to find gross logic errors in a model. Once a model is developed and experiments are being run, animations have almost no practical value. Output plots and spreadsheet statistics are the major tools for analysis. Finally, after a study is completed and different systems are being demonstrated, animations once again become very useful.

While animations are useful verification and demonstration techniques, there are some disadvantages to keep in mind. Animations have little analytical value. Modifying a system based on the experiences of a few minutes or even hours of animated operations is not wise. Even as a demonstration, the time spent by managers watching even the most carefully crafted animation is typically just a few minutes. The requirements for an animation will almost always force you to develop a more cumbersome model than necessary, with many details that are added simply to support the animation. Take, for example, a system with 100 machines processing thousands of parts an hour. We can *simulate* this system with an event graph having only three or four vertices. If one hundred machines are added, the model does not change To *animate* the system, a different representation will be needed for each machine. This is true regardless of the particular simulation software your are using. However, it is easy to add the detail necessary for an animation to an event graph by connecting additional subgraphs.

Animating a SIGMA model is simple. The balls that represent event vertices are replaced by two types of pictures: one image appears when the vertex executes, and the other appears when the vertex is inactive. These two pictures are in the form of bitmaps that are selected from the Edit Vertex dialog box by clicking on the Active Picture or Inactive Picture buttons. For fancy animations, the pictures can be actual scanned photographs; they can also be simple drawings created by a drawing program. In our tutorial, we will use pictures created with Windows Paint.

Each individual picture used in SIGMA cannot be larger than 64,000 bytes; however, you can use as many pictures as your computer's memory or version of SIGMA will allow. Hint: If you are creating pictures using Windows Paint, limit the picture sizes to less than two or three inches square using the Image/Attributes menu item. Save the picture as a 16 color bitmaps or monochrome bitmap (which are good for sequencing video files) using the Save As option.

The fancier the pictures, the more impressive the animation will be. However, remember that others will probably not spend more than a few minutes looking at your animation. You can use an inexpensive digital camera to take the pictures for your simulation (of course, crop the pictures so that the bitmaps are less than 64,000 bytes in size).

8.2 Classes of Animated Objects

To get started, it will be helpful if we think of the physical entities in a system as belonging to one of three classes:

1. S*tatic resident entities* (such as halls and windows in a building): These entities do not change their appearance when the state of the system changes during a simulation run.

2. *Dynamic resident entities* (such as machines): These entities may change their appearance when the state of the system changes, but they remain at the same location.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

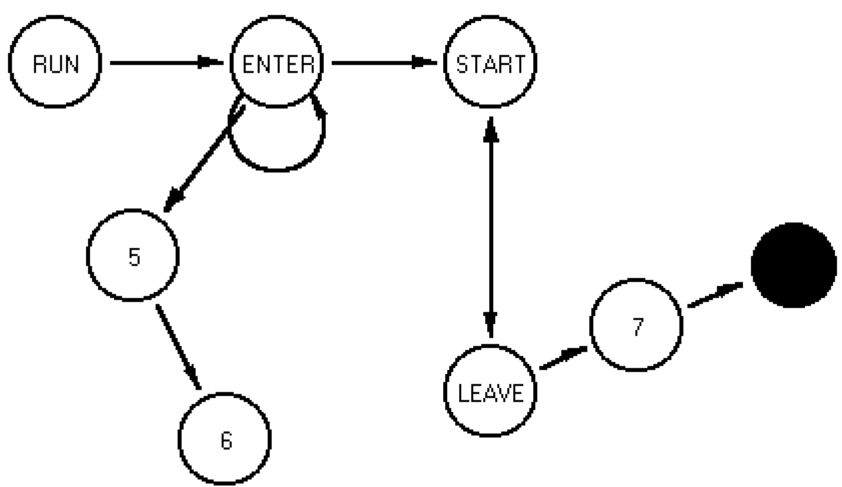
3. *Transient entities* (such as parts, customers, and cars): These entities may change their appearance and location while the simulation is running.

The three classes of physical entities are represented by different combinations of active and inactive pictures. For static resident entities, the active and inactive pictures are identical.

Dynamic resident entities typically will have different active and inactive pictures; the active picture represents a transitional state, and the inactive picture represents the state immediately after the vertex executes. Furthermore, a dynamic resident entity will often have several vertices stacked (or "grouped") on top of each other so that the picture of the current state of the entity is showing. You can think of the inactive pictures for the grouped vertices associated with a dynamic resident entity as the frames of a video of the entity and the active pictures as transitions between these video frames that smooth out the motion.

Transient entities will have an active picture that is a representation of the entity and a inactive picture that is blank. Some of the vertices in our original event graph might trigger some movement of a transient entity. For example, in our carwash model, the ENTER event might trigger a dirty car to enter the carwash and a LEAVE event might trigger a clean car to move away from the carwash. This movement is represented by a string of vertices along the transient entity path of motion triggered by the event. For example Figure 8.1 shows our original carwash model with a string of vertices attached to the ENTER and LEAVE events.

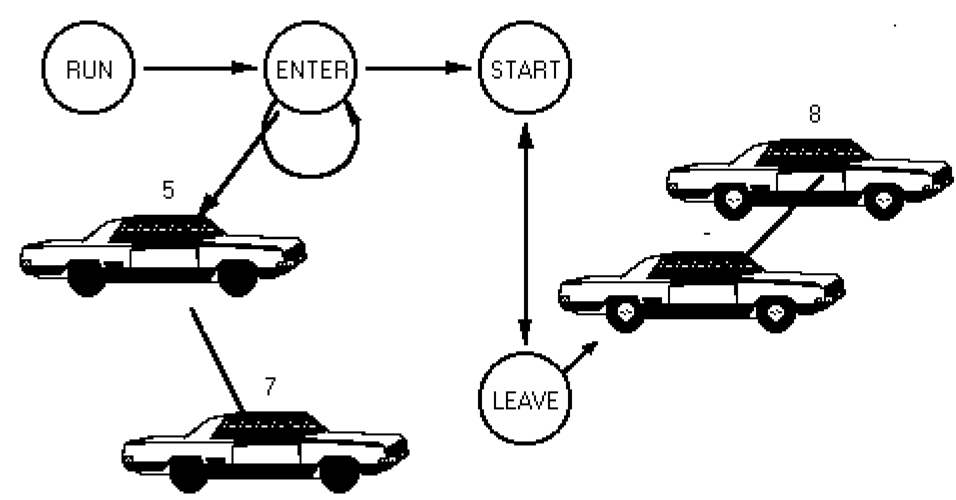
Figure 8.1: Strings of Vertices added to ENTER and LEAVE Events of CARWASH.MOD.Cars Entering and Leaving Facility.



When the active pictures are represented by clean and dirty cars and the inactive pictures are represented by blank pictures the size of the cars, the result will be like Figure 8.2. (The Select All button in the Edit menu was pressed to show all the animation elements.)

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 8.2: Strings of Vertices added with Active Pictures Showing; represents Cars Entering and Leaving the Carwash Facility.



These paths of motion are very easy to create. Simply use the Create Single Edge tool to create the first vertex in the string. Edit the active picture for the new vertex to be a bitmap of the transient entity and the inactive picture to be a *blank* bitmap the same size as the transient entity. Now select the Create Process tool and click a string of vertices that can later be moved along the desired path of motion. (The string of vertices will have the previous pictures as defaults.)

Bitmaps of the proper size are easily created using the Windows Paint utility and sized using the Attributes command in the Image menu. Clipping scanned photographs of the transient entity will make the animation look more like a video movie.

The delay times on the edges between these transient entity motion vertices can be set to represent actual move times, including acceleration and deceleration. The Time Steps run default mode along with a `TIMER` vertex should help make the motion smoother. (A "timer" is a self-scheduling vertex with an edge delay time of one time unit.). If you have a slow computer, you might change some of the edges between the transient entity motion vertices to be pre-emptive edges (`delay=*`). However, have no more than four or five pre-emptive edges execute in a sequence. If there are more than five in a sequence, increase your Windows stack to avoid a stack overflow.

The three types of entities are summarized in Table 8.1.

Table 8.1: Physical Entity and Picture Representations for Animations

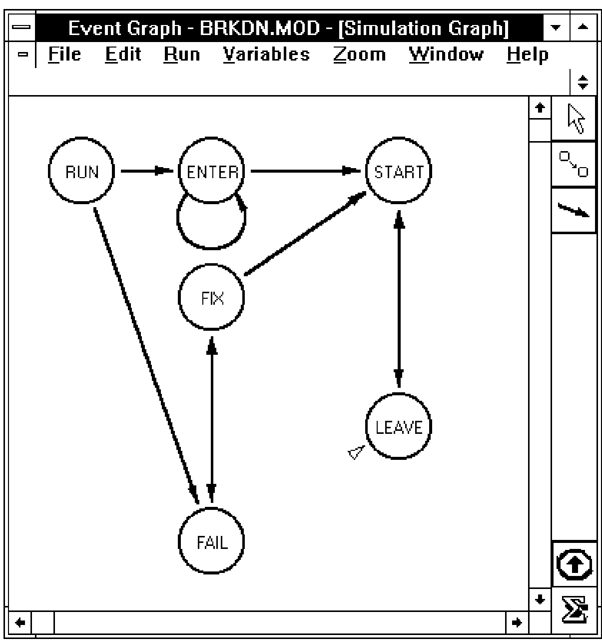
Physical Entity	Active Picture or Inactive Picture	"Groups" of Stacked Vertices
Dynamic Resident Entity	Transition/ Ending State	Yes
Static Resident Entity	Same Pictures for both	No
Transient Entity	Picture/blank	No

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

8.3 Tutorial: Animating Resident Entities

We will animate a simulation model from the SIGMA directory, BRKDN.MOD. This model represents a single machine, with periodic failures, that processes parts that arrive at random times. We will use each of the three basic types of animated objects discussed earlier, beginning with resident entities. The event graph for BRKDN.MOD is shown in Figure 8.3. Follow along on your computer as we animate this model.

Figure 8.3: Event Graph of BRKDN.MOD



Start a SIGMA *session.*

Open the model, BRKDN.MOD.

To represent the states of the queue (a dynamic resident entity), we will use a subgraph on your SIGMA directory called, QSIZE.MOD. You should minimize the current SIGMA window and open a *separate* SIGMA session from the Windows Start menu. Read QSIZE.MOD into this new SIGMA session. The event graph should look like Figure 8.4.

Click the Minimize *button on the* BRKDN.MOD SIGMA *session.*

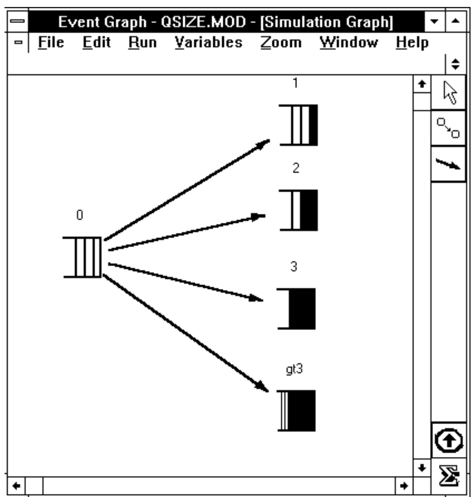
Return to the Start Menu *and open a second session.*

Click the Open/Event Graph *command under the* File *menu, locate* QSIZE.MOD*, and double-click on it.*

In QSIZE.MOD, simple active and inactive pictures drawn with the Windows Paint utility represent the different states of a queue. Vertices represent 0, 1, 2, 3, and >3 customers in the queue. The variable, QSIZE, represents the current size of the queue. If you explore QSIZE.MOD a bit, you will notice that vertex 0 has QSIZE passed to it as a parameter. Also note that the values of QSIZE are tested on each of the edges to see which picture is appropriate. Click on the vertices and edges to see how QSIZE.MOD works. It is very simple and serves no purpose other than animation. Do not "save" QSIZE.MOD if you mess it up, but rather "read" it into your SIGMA session again.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 8.4: Event Graph for `QSIZE.MOD`, with Inactive Pictures Showing



The next step will be to copy QSIZE.MOD and paste it into your other SIGMA session with the model BRKDN.MOD.

For QSIZE.MOD, *click the* Select All *command under the* File *menu.*

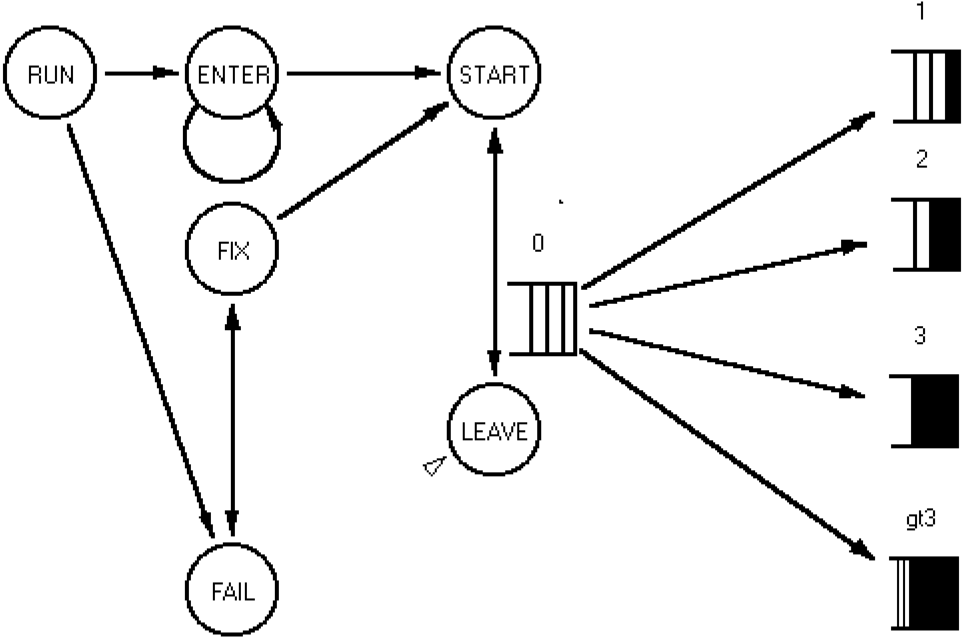
Click the Copy *command under the* Edit *menu.*

Click the Minimize *button on the* QSIZE.MOD *simulation window.*

Activate the BRKDN.MOD SIGMA *session.*

Press the Paste *command under the* Edit *menu for* BRKDN.MOD.

Figure 8.5: BRKANAM1.MOD After Pasting QSIZE.MOD into BRKDN.MOD



Hold the mouse button down on one of the newly copied vertices of QSIZE.MOD and drag the entire subgraph off to the side. (You might have to re-select these vertices by holding [⇧Shift] and clicking on them before you can drag the

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

selected subgraph out of the way.) If your screen looks something like Figure 8.5, the transfer was successful, and you can close the SIGMA session with QSIZE.MOD. The model in Figure 8.5 is called BRKANAM1.MOD in your SIGMA directory.

With the QSIZE.MOD *subgraph still highlighted, drag it off to the side of* BRKDN.MOD.

Click the Minimize *button on the* BRKDN.MOD *event graph.*

Close the QSIZE.MOD SIGMA *session.*

Re-open BRKDN.MOD.

Now we will simply wire the new QSIZE.MOD subgraph into our model by drawing two edges: One edge from the ENTER event to the 0 vertex of QSIZE.MOD and the other edge from the START event to the 0 vertex of QSIZE.MOD. These two edges should pass the attribute value of QUEUE (into the QSIZE parameter of vertex 0) to let the subgraph know the current number of parts in the system. The delay time for these edges should be an asterisk (delay = *), meaning pre-emptive execution, and they should be unconditional (Condition: 1 == 1). This model is BRKANAM2. Next, select all of the vertices in the QSIZE.MOD subgraph and stack them on top of each other.

Click on the Create Single Edge *tool.*

Create an edge from the ENTER *vertex of* BRKDN.MOD *to Vertex 0 of the* QSIZE.MOD *subgraph.*

Create another edge from the START *vertex of* BRKDN.MOD *to Vertex 0 of the* QSIZE.MOD *subgraph.*

Click on the Select or Edit *tool.*

*Double-click on each new edge; edit each dialog box so the delay time is *.*

Highlight the QSIZE.MOD *subgraph by lassoing the vertices or pressing* [⇧Shift] *and clicking the mouse.*

Click the Group Vertices *command under the* Edit *menu.*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Table 8.2: The Assignment of Bitmaps to Active and Inactive Pictures for `BRKANAM3.MOD`

Vertex	Representing	Type of Entity	Active Bitmap	Inactive Bitmap
Run	A Section of Track	Static Resident	TRACK.BMP	TRACK.BMP
Enter	A Customer	Transient Entity	CUSTOMER.BMP	BLANK.BMP
Start	Machine Working (Yellow)	Dynamic Resident	START.BMP	BUSY.BMP
Leave	Machine Available (Green)	Dynamic Resident	LEAVE.BMP	IDLE.BMP
Fail	Machine Broken (Red)	Dynamic Resident	FAIL.BMP	BROKEN.BMP
Fix	Machine Available (Green)	Dynamic Resident	FIX.BMP	IDLE.BMP

Now we will assign pictures to the other vertices according to Table 8.2. Note that the *static resident entity* of the track is represented by the `RUN` vertex and has the same active and inactive pictures. The `ENTER` vertex represents the *transient entity* of a part entering the system; it has a blank bitmap for its inactive picture. The four vertices (`START`, `LEAVE`, `FAIL`, and `FIX`) represent various events for the machine, which is a *dynamic resident entity.* These vertices have transitional bitmaps as their active pictures; their inactive pictures represent the state of the machine immediately after the event occurs. The resulting model is `BRKANAM3.MOD` in your SIGMA directory.

Double-click on the `RUN` *vertex to open the* Edit Vertex *dialog box.*

Press the Inactive Picture *button to open a dialog box with a list of bitmaps.*

Scroll through the list until `TRACK.BMP` *is located; double-click on it.*

Press the Active Picture *button, locate* `TRACK.BMP`*, and double-click on it.*

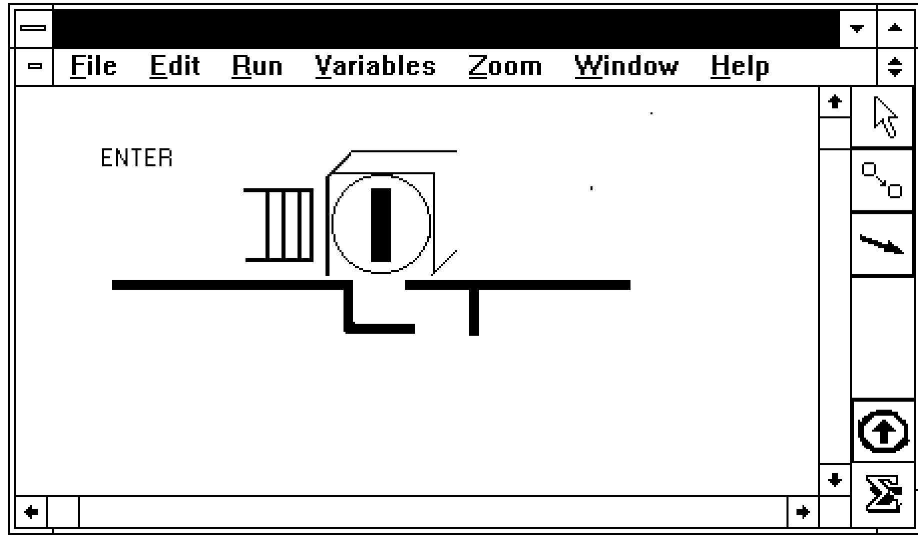
Press the OK *button at the bottom of the dialog box.*

Repeat this process with all the vertices, using the active and inactive pictures specified in Table 8.2.

Next, we will group the four vertices that represent the various states of the machine (`START`, `LEAVE`, `FAIL`, and `FIX`) into one machine, and then we will combine the objects on the screen. The animation, `BRKANAM4.MOD`, looks something like Figure 8.6.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 8.6: Event Graph for BRKANAM4.MOD, A Resident Entity Animation of BRKDN.MOD



Highlight the START, LEAVE, FIX, *and* FAIL *vertices using* [⇧Shift] *and the mouse.*

Click on the Group Vertices *command under the* Edit *menu.*

Click on the Select All *command under the* Edit *menu.*

Next, click the Hide Selected Edges *command also under the* Edit *menu.*

Drag the objects to locations similar to that in Figure 8.6.

You should load BRKANAM4.MOD and run it to see what happens. The simulation has become the animation. An add-on animation program would add an unnecessary layer of software between you and your model. Try clicking on some of the objects in BRKANAM4.MOD; you can Ungroup Vertices and Show All Edges using the commands under the Edit menu. You can double-click on vertices or edges and change their behaviors.

A final note about dynamic resident entities: in some cases it may be necessary to offset some of the pictures in a group representing a dynamic resident entity. For example, we may want to have a robot arm sticking out the side of a machine. The robot arm is part of the resident entity representing the states of the machine (the arm would be represented by a vertex that is grouped with the other machine state vertices). However, it has a blank inactive picture so that the arm would appear to retract. The arm is moved off to the side of the rest of the vertices in this group by slowly clicking on the group (so that it is not interpreted as a double click) until the arm appears. It is moved to the side, down, or up using the arrow keys. The other vertices in the group remain in their previous positions. Using the arrow keys to move a single vertex in a group can also give a sense of realism in animations of living things. (See SIGMA model, TOUCAN.MOD, where the body of the bird was moved slightly using the arrow keys to make it appear to "wiggle.") Moving only one of the vertices in a group with the arrow keys is a powerful animation technique. The saved model has vertex groups listed at the bottom of the file. This shows the group members with their "offsets" from the center of the group.

You will soon notice that the animated model, BRKANAM4.MOD, is a bit dull. This is because we have not yet added transient entities to our animation. It is the transient entities that move across the screen and bring our animations to life.

8.4 Tutorial Continuation: Animating Transient Entity Motion.

Adding transient entity motion is quite simple in SIGMA. As discussed earlier, you simply attach a string of vertices to an event that will represent the path of motion of the transient entity when the event occurs. The active picture for transient entity vertices is a picture of the entity. Here a simple colored bitmap of a bar is used to represent the part moving to and

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

from the machine. Transient entities have a *blank* inactive picture the same size as their active picture. The active picture flashes at the current location of the transient entity. This picture is then blanked out when the transient entity moves on. The vertices in such a string will typically have no state changes associated with them and are connected with an unconditional edge with pre-emptive execution (`delay = *`). Figure 8.7 illustrates adding a transient entity leaving the machine after the `LEAVE` event. (The Select All command was clicked to show the active pictures of the new vertices.)

Figure 8.7: Adding a String of Vertices to Represent Movement of a Transient Entity

> **IMPORTANT:** *Do not connect more than four or five vertices in a row with edges that have a delay of *. Insert an edge with zero delay to avoid a stack overflow condition.*

Open `BRKANAM4.MOD` *or continue with the model you have been developing.*

Click once on the group of vertices represented by the box. This will be the starting point for the path followed by a transient entity.

Since the starting point of the transient entity must be a vertex, click the Ungroup Vertices *command under the* Edit *menu to ungroup these vertices.*

Click the right button to get into Create Process *mode; click the left mouse button on the location of the first point of the transient entity path.*

Note that the vertex you create will be a duplicate of the last vertex created when building the model.

After getting into Select or Edit *Mod, double click on this new vertex.*

Assign `CUSTOMER.BMP` *to the* Active Picture *and* `BLANK.BMP` *to the* Inactive Picture. *Press the* OK *button in the dialog box to incorporate the new images into the event graph.*

Check if this is successful by clicking on a blank space; the transient entity image should disappear.

Click again under the number of the new vertex; the transient entity image should reappear.

Click on the right button to get into Create Process *mode.*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Click several times along the path to be followed by the transient entity. (All you should see are the edges of the transient entity path.)

Press the right button to get into Select or Edit *mode.*

Click on the Select All *command under the* Edit *menu.*

Click on the Hide Selected Edges *command under the* Edit *menu.*

Click on any blank space to deselect the vertices: All the transient entities should disappear, leaving just the numbers of each break-point in the path.

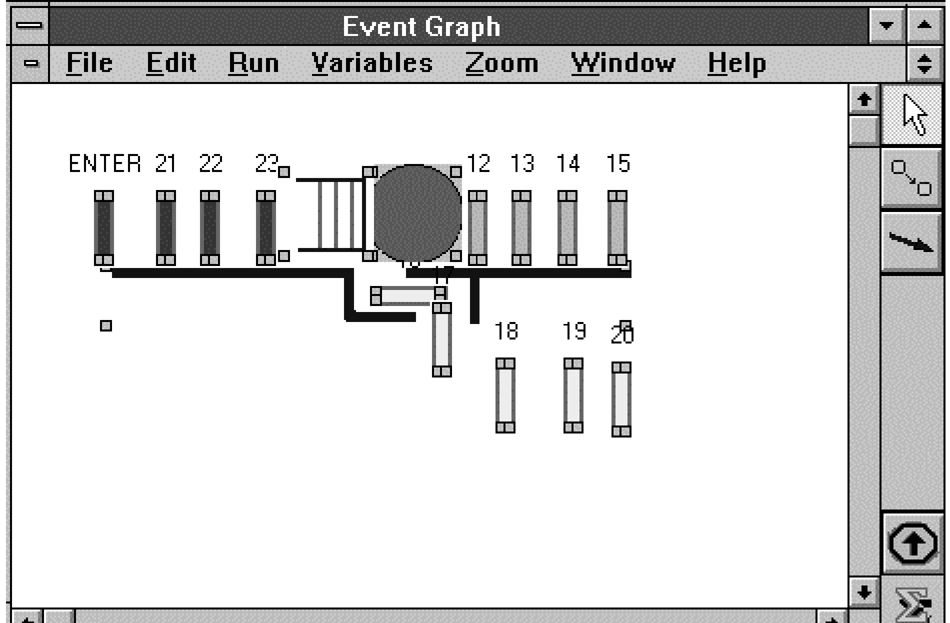
When you run the simulation, the numbers at the break-point vertices along the transient entity path will disappear.

HINT: *Put a very high execution priority (low number) on each edges along the transient entity path if you want the transient entity to complete its movement before any other vertex executes.*

Highlight the ungrouped boxes, and group them again.
Press the Start Run *tool to see the transient entity path in motion.*

A string of vertices was added to the `ENTER` event to show the part entering and to the `FAIL` event to show a part being discarded. The model is `BRKANAM5.MOD` in your SIGMA directory and is shown in Figure 8.8. Again, the Select All option was clicked to show all parts of the animation.

Figure 8.8: `BRKANAM5.MOD` with Strings of Vertices Representing Transient Entity Paths Added.



You should load and run `BRKANAM5.MOD` and observe how it works. The local display variables, `INPUT`, `GOOD`, and `BAD` were added to the model, in addition to `QSIZE`, to show where each part went.

With the animation, it is rather easy to see that there is an error in the model; the count of discarded bad parts is increased even if the machine fails when it is not working. Since what goes into the system must go somewhere, the following equation should hold:

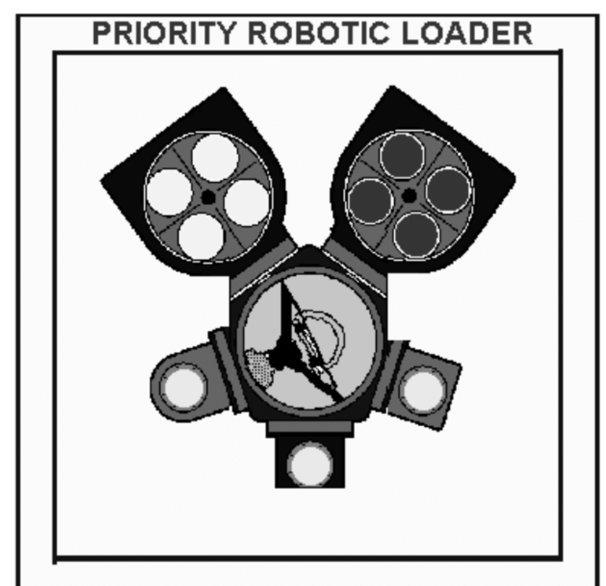
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
INPUT = GOOD + BAD + QSIZE + SERVER.
```

Increasing the number of BAD parts discarded even if the machine failed when empty will cause this balance equation to be invalid. Adding a vertex to indicate if the machine fails when empty or loaded solves this problem. (See BRKANAM6.MOD) Note that a puff of smoke is added when the machine fails, using the same technique for adding transient entities.).

Other examples of animations have been included with SIGMA, including one with the ubiquitous carwash, ACARWASH.MOD. Also see SIMAN.MOD for an animation of a process-oriented simulation language. ROBOT.MOD, in Figure 8.9, is an actual animation of a loading robot used in semiconductor manufacturing. The model represents an automated robot that processes wafers in vacuum chambers. Wafers enter the system at Station 1, are processed individually at Station 2, are moved to either Station 3 or Station 4 for batch processing, and then leave the system through Station 5.

Figure 8.9: An Animation Of Priority Loader Robot (Cluster Tool)



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

9

Modeling Input Processes

Discrete event simulations are typically both dynamic (they change over time) and stochastic (they are random). So far, we have concentrated on learning to model the dynamics of a system. In this chapter, we look at some of the issues and techniques for modeling randomness. Random numbers, trace-driven simulations, parametric input distributions, and empirical input distributions are discussed. The exponential autoregressive process is used to illustrate dependence in the input process. Various sources of data with which to model input processes are presented. A discussion on reusing random number seeds to reduce variance in model output is also included. The chapter closes with some techniques for generating random variates, including the generation of non-homogeneous Poisson processes. These processes are useful models for generating exogenous events to drive a simulation model.

9.1 Randomness

Although there is much literature in this area, here we will be brief. This is partly due to the fact that simulation modelers rarely write code for this part of their programs. Algorithms for imitating random sampling are well developed, and reliable codes are readily available. Furthermore, while the dynamics of most simulation models are unique, the stochastic logic tends to be the same. Almost all of the simulations we have discussed have had at least one random process as an input to the model. For instance, our carwash model was driven by customers arriving at randomly spaced intervals. We modeled the car arrival process by assuming that the intervals between arrivals were independent and had a particular uniform probability distribution. Realistic situations can be quite a bit more complicated: cars might arrive at a higher rate during rush hour, on weekends, or on days with nice weather. In this chapter we will review some of the considerations and techniques for generating random input processes to drive a simulation.

Broadly speaking, there are three popular approaches to modeling input processes: using pre-recorded data, using sample probability distributions, and using mathematical probability models. Pre-recorded data is also called a process "trace", a sample probability distribution is also called an "empirical" distribution, and mathematical probability models are sometimes referred to as theoretical or "parametric" models. All but trace-driven simulations require the use of random numbers.

9.2 Trace Driven Simulations

In a trace-driven simulation whenever a value for a random variable is needed by the simulation, it is read from a data file. When it is practical, this input file contains actual historical records. In our carwash example, the trace might be a file of the intervals between successive car arrivals recorded while watching the system. Sometimes only a portion of the input is trace driven. In a fire department simulation, the times and locations of calls might be read from a file with data from a dispatcher's log book while other inputs, such as equipment repair status and travel times, might be generated as they are needed.

Trace data is simple to read into SIGMA using the `DISK{}` function. Recall that the `DISK` function has two arguments. The first argument is the full name of the data file (with drive and directory path if necessary); the second is an integer index telling which entry is to be read. When the index is zero, the file is read sequentially, wrapping around to start at the beginning again when the end of the file is reached.

We could place the values (separated by at least one space) of times between customer arrivals at our carwash in a data file called `ARRIVAL.DAT`. We then could use the function, `DISK{ARRIVAL.DAT;0}`, as the delay time on the self-scheduling edge for the `ENTER` vertex that generates successive customers arrivals. If `ARRIVAL.DAT` contains only five observations and looks like the following

```
.32    2.6
.78    4.3  .85,
```

then the sequence read by the `DISK{ARRIVAL.DAT;0}` would be

```
.32, 2.6, .78, 4.3, .85, .32 (wrapped around) 2.6, ...
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

There are some distinct advantages to having the values of random processes read from a data file. Foremost, there is less concern with the validity of the trace input data than when the inputs are artificially generated. We never really know how individual input variables might actually be distributed. Furthermore, it is quite difficult to capture the dependencies between different input processes or between successive values of the same input process. (We will re-examine the question of dependent input later in this chapter.)

When attempting to validate that a model accurately represents the behavior of a real system, there is probably no better test than to simulate previous system behavior using past input data. All discrepancies between the performance of the model and the system can then be attributed to model assumptions or errors in the simulation code. Assumptions and code errors are only dangerous if they are hidden. Assumptions can be evaluated as to their potential impact on decisions and the benefits they offer in model simplification. Known coding errors can be corrected. Accurate representation of the past is a reasonable minimal expectation. However, just because your model closely imitates last year's performance with last year's input data by no means makes the model correct or even useful. Driving a simulation with an artificial data trace rather than historical data is a useful technique for debugging the logic of a model. Specific sequences of otherwise random events can be forced to reoccur in a model that is being tested or enriched.

Many of the disadvantages of trace driven simulations are more or less obvious; some are not. While historical data traces provide a valuable source of simulation input when developing or changing a model, traces are not a good general approach to driving a simulation model when it is being used for analysis. Trace driven simulations require storage space for the data buffers, or they can be very slow due to the significant overhead of reading input files. Historical data that is detailed enough to drive a simulation is probably not available and would be time-consuming and expensive to collect. Actual data is also subject to errors in observation or may be invalid merely due to the intrusion of the observer. Even if detailed data is readily available, the simulation still could not be independently replicated or run for a longer period than the interval for which actual data was collected. In using trace input, we are giving up two of the major advantages to simulation modeling: time compression and independent replication.

Using a historical data trace as input when considering alternative policies or designs is probably not valid. Most trace driven simulations are *closed* systems. That is, the laws governing the input processes are not dependent on the state of the simulation. On the other hand, the actual system most likely is an *open* system that influences its environment. Although they are run all the time, trace-driven, "what if" simulation experiments are usually not appropriate. All statements about the effect of a change are based on the implicit assumption that the change has no influence on the environment in which the system operates. This is analogous to assuming that a system operates in an inelastic economy; demand is not altered by supply, price, quality, etc. What we most likely want to know is how sensitive the new system might be to changes in the input. It is difficult to do input sensitivity analysis with trace input.

Another disadvantage concerns rare but important events (i.e., a single, very long repair time at a service center). Since an unusual event is, by definition, unlikely to be on any given historical data trace, we may never see its influence in our simulation runs. Perhaps worse, if such a rare event happens to be in our trace, it will occur in every system we simulate. We might choose an alternative that handles this unusual case well but is too expensive or does not perform well in more typical situations.

9.3 Random Number Generators

At the heart of stochastic modeling are random numbers. We define random numbers as positive fractions whose values are assumed to be independent of each other and equally likely to occur anywhere between zero and one. Random number generators are algorithms that imitate the sampling of random numbers. In SIGMA, `RND` has its values generated by such an algorithm. The algorithm simply takes an integer that you supply as the "seed" and recursively multiplies it by a fixed constant, divides by another constant, and uses the remainder as the next "seed." This remainder is also scaled to lie strictly between zero and one and used as the current value for `RND`. The random number generator we are using originated with Lewis, Goodman, and Miller (1969).

We will forego a discussion of the philosophy of random number generation. Suffice it to say that, by most common notions of what we mean by randomness, it is impossible to "generate" random numbers. Indeed, there have been some widely used algorithms that generate numbers that look far from random. Perhaps with the exception of the seed you give it, there is nothing whatsoever truly random in the values of `RND` or the outputs from any other random number generator; they just "look" random if you are not overly particular. Any random number generator that has passed all statistical tests for randomness simply has not been tested enough. Nevertheless, modeling the output from many random number generators as being true random numbers has been amazingly successful.

It is very easy to modify the SIGMA-generated source code in C to include multiple random number input streams. This is discussed when we introduce correlation induction techniques used in variance reduction in Section 9.8.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

9.4 Using Empirical Input Distributions

With this approach to input modeling, a sample of observations of an input variable is used to estimate an empirical probability distribution for the population from which the sample was taken. The customary estimate of the empirical probability distribution is to assign equal probability to each of the observed values in the sample. If the sample contains N observations, then the empirical probability distribution will assign a weight of 1/N to each of these observations.

Suppose that a trace of interarrival times of customers to our carwash is in the data file ARRIVALS.DAT. Sampling from the empirical distribution is equivalent to reading a randomly chosen value from this file. This is like shuffling and drawing from a data trace. In SIGMA this is done by making the index of the DISK function a random integer from 1 to N. For example, the file, ARRIVALS.DAT, may look like the following:

```
.3   .42   .2   .54   .79
```

A delay time of DISK{ARRIVAL.DAT;1+5*RND} will result in one of these five numbers (chosen at random) being used. Wrapping around the data file will not occur here since the index is never greater than 5. A considerably more efficient but perhaps less flexible approach is to place the data in an array and generate the index of the array uniformly. If the data is in the array, X, then X[1+5*RND] would select one of these values with equal probability. In SIGMA the index is automatically rounded down to the nearest integer.

Similar to using historical data traces as input, the big advantage to using sample distributions to generate input is that there is less concern over validity. However, with sample distributions we can replicate and compress time. The disadvantages to using the empirical input distributions are similar to the disadvantages to using trace input: the data might not be valid, sensitivity analysis to changes in the input process is difficult, we cannot generalize the results to other systems, and it is hard to model rare events. The one major advantage trace input has over empirical distribution sampling comes in modeling dependencies in the input processes. The trace will capture these dependencies whereas the empirical distributions will not.

9.5 Using Parametric Input Distributions

Efficient algorithms have been developed for imitating the sampling from a large number of parametric families of probability models. In Chapter 7, we saw some SIGMA functions for artificially generating samples that behave very much as though they were actually drawn from specific parametric distributions.

The values of parameters for these models determine the particular characteristics of the sample. This ability to easily change the nature of the input by changing a few parameter values is the primary advantage of using these models to drive a simulation. The variate generation algorithms in common use are fast and require very little memory. Furthermore, you can easily run replications, compress time, and generalize the results to other systems having the same structure. The major drawback to using parametric input distributions is that they can be difficult to explain and justify to people who have no background in probability and statistics.

Devroye (1986) provides a very complete reference on variate generation algorithms. The article by Leemis (1986) catalogs the relationships between dozens of probability laws.

There are several obvious classifications of probability models: finite or infinite range, continuous or discrete values. On a practical level we can also classify probability models as primarily providing good models for input processes or good models for output statistics.

Most of the common distributions that are used to model output statistics are derived from the normal (also called the Gaussian) distribution. In SIGMA the function NOR{M;S} will imitate sampling from a normally distributed population with a mean of M and a standard deviation of S. M can be any real-valued expression, and S can be any positive real-valued expression. Samples from other distributions such as the *t*, *F*, and χ*–square* can easily be derived from their relationship to the standard normal distribution (Leemis, 1986). A selection of parametric distributions that are good for input modeling are provided as SIGMA functions.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

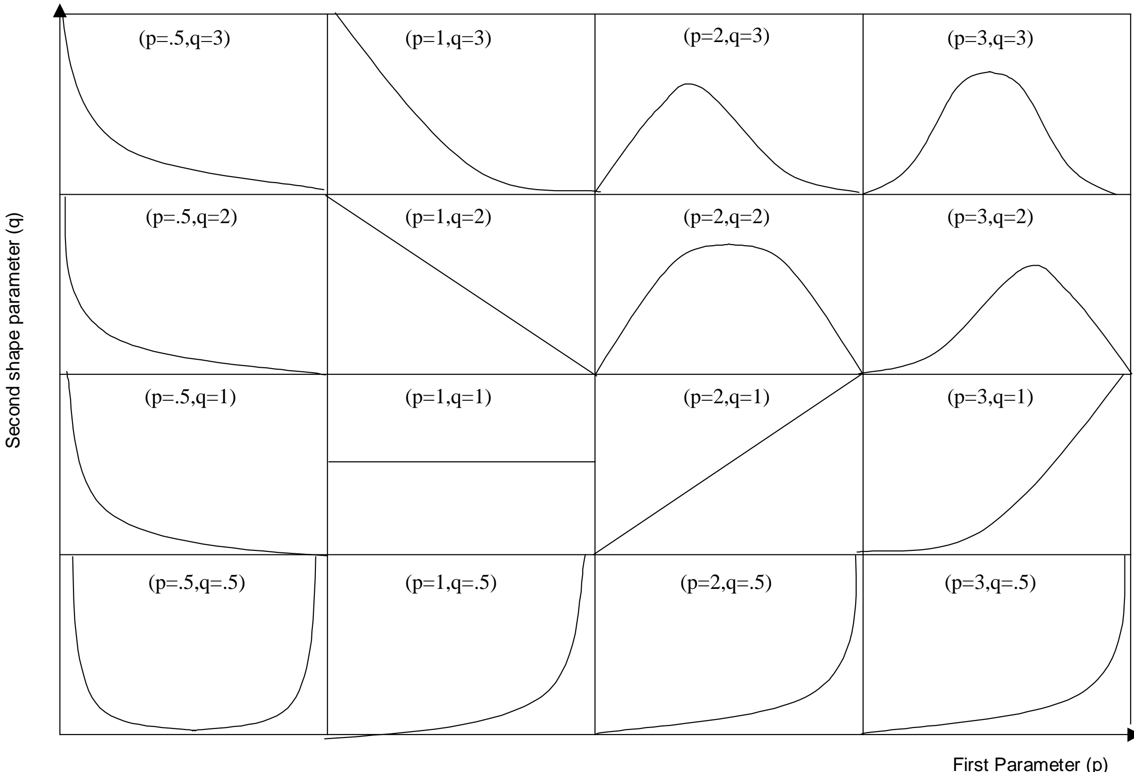


Figure 9.1: Standard Beta Density shapes $f_X(x) = \Gamma(p+q)/(\Gamma(p)\Gamma(q))\, x^{p-1}(1-x)^{q-1}$

The function `BET{P;Q}` imitates sampling from a beta random variate with parameters given by the positive real-valued expressions `P` and `Q`. This is a standard beta on the interval from 0 to 1 and can be scaled to the interval `(C,C+D)` in the usual way as `C+D*BET{P;Q}`. The beta distribution is one of the most useful in simulation input modeling because of the richness of shapes it can take with simple changes of its two parameters. Figure 9.1 provides a convenient "matrix" of beta distribution shapes for various values of its parameters.

`ERL{M}` will give a sample imitating an M-Erlang random variate. The parameter, `M`, can be any positive integer-valued expression. Multiplication by a real number, `A`, will move the mean from `M` to `A*M`. Since the *M*-Erlang is the sum of `M` independent exponentially distributed random variates with mean 1, `A*ERL{1}` will be an exponential random variate with mean, `A`.

If you want a sample that is even more highly skewed than one having an exponential, the function, `GAM{A}`, will imitate sampling from a gamma distribution with a fractional parameter. Here the shape parameter, `A`, is a real variable strictly between 0 and 1. The result can be multiplied by a scale parameter to give a variety of distribution shapes. For integer values of `M`, *M*-Erlang random variates have the same distribution as gamma variates.

The function, `TRI{C}`, will imitate sampling from a triangular shaped distribution over the range from 0 to 1. The mode (peak) of the distribution is at the value of the real-valued expression, `C`, which is between 0 and 1 inclusive. The linear function,

```
A+(B-A)*TRI{(D-A)/(B-A)}
```

imitates a sample from a triangular distribution between `A` and `B` with a mode at `D`.

It is easy to code other probability models with SIGMA. We will illustrate this with two very useful models, the multinomial and the lambda. The multinomial probability law models independent sampling, with replacement, where there are only `K` possible mutually-exclusive outcomes. The simplest example of multinomial sampling is drawing a numbered ball from a jar where after each sample the chosen ball is placed back in the jar. While there are more efficient algorithms available, multinomial sampling is easily done in SIGMA using the `DISK` function. To illustrate, suppose that there are only three possible outcomes from our experiment. We will see a 1 with a probability of 1/2, a 2 with a probability of 1/3, and a 3 with a probability of 1/6. We simply set up a data file called, `MULTINOM.DAT`, which has the following entries

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

1 1 1 2 2 3

The statement `X = DISK{MULTINOM.DAT;1+6*RND}` will assign the values of `X` with the given probabilities. The index of the above `DISK` function is a randomly chosen integer from 1 to 6 (rounding the index down is automatic).

The lambda (more properly the "generalized" lambda) distribution is like the beta in that it can take on a wide variety of shapes. This distribution is discussed by Ramberg, *et al*. (1979). The major difference between the lambda and the beta is that the lambda can take on an infinite range of values whereas the beta is restricted to take on values only within a specified interval. There are four parameters to the lambda that can be estimated using subjective data. Generation of a lambda variate is very easy. Suppose that you have defined real-valued state variables, `X` and `R`, along with the four lambda parameters, `L1`, `L2`, `L3`, and `L4`. The statements `R=RND`, `X=L1+(R^L3-(1-R)^L4)/L2` will give values of `X` that imitate sampling from the lambda.

You should exercise caution when using Erlang, exponential, gamma, lambda, and normal distributions for input modeling. Variates from these families can take on very large values. You need to check and/or control for reasonableness. For instance, if you are using an exponential (`ERL{1}`) variate to model the service time at a store, it is not reasonable for the service time to exceed some upper limit. No one is going to wait years for service. Truncating these distributions at some upper bound is advised. To illustrate, a vertex with the state changes,

```
X=ERL{1},
X=X*(X<5)
```

would produce a sample from an exponential variate truncated to be strictly less than 5. The truncated probability (the likelihood that an exponential variate exceeds 5) is added to the probability of zero occurring.

9.6 Modeling Dependent Input

The book by Johnson (1987) along with the articles by Lewis (1981) and McKenzie (1985) are devoted primarily to the generation of dependent input processes. To illustrate the critical importance of recognizing and modeling dependence in the input processes for a simulation, we will use a simple process called the exponential autoregressive (`EAR`) process (Lewis, 1981).

Successive values of an `EAR` process, `X`, with mean, `M`, and correlation parameter, `R`, are generated from the recursion

```
X=R*X+M*ERL{1}*(RND>R)
```

with an initial value of `X` given by the exponential `M*ERL{1}`. The values of this process will have an exponential distribution, but they are not independent. The correlation between two values of an `EAR` process that are `K` observations apart is R^k. At the extremes when `R=1`, the Boolean variable (`RND>R`) is always equal to zero and the above expression reduces to `X=X`. The process never changes value, so the serial correlation is a perfect 1. When `R=0`, (`RND>R`) is always equal to 1, and independent (zero correlation) exponential random variables are generated as `X=M*ERL{1}`. The `EAR` process is easy to use since its serial dependency can be controlled with the single parameter, `R`. Although histograms of the values of this process look like a simple exponentially distributed sample, the line plots of successive values of an `EAR` process look rather strange. As is obvious from the `EAR` process equation, the value of `X` takes large randomly-spaced jumps and then decreases for a while.

To see the effects of dependent input, consider our simple queueing model, `CARWASH.MOD`, where we change service times. We will use an `EAR` process, with mean, `M`, and a correlation parameter of `R`. This model is called `EAR_Q.MOD`. If you run `EAR_Q.MOD` with the same `M` but very different values of `R`, you will see a radical difference in the output series. Dependence in the service times has made the two systems behave very differently. When building this model, if we had looked only at the histograms of service times and ignored the serial dependence on service times, we might have had a very poor model.

9.7 Sources of Data

One of the most common excuses given for not successfully completing a simulation study is the lack of "real-world" data with which to model the input processes. While real data is valuable in establishing the credibility of a simulation, lack of data is not a good excuse for not proceeding with the study. You should be trying a wide range of reasonable input

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

processes to assess system performance sensitivity to changes in the environment. Furthermore, there are many sources of data that should not be overlooked when planning and conducting a simulation study. Each source of data has different associated costs, risks, and benefits.

At the least detailed level, there are physical constraints of the system being modeled, such as space limitations for a waiting line. This is reliable, low-cost information that gives design insight. It tells a great deal about the rationale for the way a system was designed. Unfortunately, it is static information and is of little help in modeling the system dynamics. At the next level of detail, there are the subjective opinions of persons involved with the system. This is low-cost (but unreliable), static data that provides behavioral insights about the people involved with designing, operating, and managing the system. Increasingly detailed information can be obtained from aggregate reports on system operations such as labor, production, and scrap reports. This is low-cost, verifiable, static data that can provide performance insight. Information that is useful in modeling the dynamics of the system can sometimes be obtained from artificial data, classical Industrial Engineering MTM methodology. This type of information can give standard times (with allowances for fatigue) for performing different manual operations in a system. The cost of this information is moderate, and you do not need to have access to the actual system to obtain it. However, its validity depends on the skill and experience of the person doing the analysis. This data uses detailed motion analysis and provides policy insights in how the system managers and designers *intend* people to perform their jobs. Finally, the most expensive source of data is direct observation. The validity of this data depends on the skill of the observer and the relationship between the observer and the person being observed. Direct observations of a system's operations might be collected manually with time studies or mechanically with sensors. This data provides the operational insights needed to accurately model system dynamics.

Alternate sources of information are sometimes overlooked. For example, part routing sheets can be used to verify traced job flows in a factory. Production records might be used to augment and validate data on the reliability of machines. Knowing the number of machine cycles in a particular time period from production records along with the total number of failures from maintenance records permit you to estimate the probability that a machine will fail on a given operation cycle. It is unlikely that these failures are independent; however, at least you have a starting place for your sensitivity analysis and a potential consistency check for verifying more detailed machine failure testing data.

When deciding on what types and how much data are needed for a simulation, sensitivity analysis is of great value. Change the values of an input parameter to your simulation. If the measured system output does not change significantly, you do not need a better estimate of that parameter. On the other hand, if the output is highly sensitive to variations in a particular input parameter, you had best devote some effort to estimating the true value or range of that parameter. Sensitivity is only one key factor in determining a need for more information about an input process. The other is the degree of control that you might have over the process. If the process is essentially beyond your control, detailed data collection of the current behavior of the process is probably not worthwhile. For example, customer arrivals at your carwash are not under your direct control. Knowing a great deal about the current demand is not critical. Hopefully, the demand for your carwash will increase dramatically from its present level once the improvements from your simulation study are implemented. You should run any prospective design against both high and low demand.

Finally, one needs to be alert for communication problems when collecting data. You might think that the data is about one thing when it is really about another. Or the data might have been translated, scaled, or simply recorded incorrectly. Fortunately, these are not fatal problems in carefully conducted simulation studies. We are going to change the input data during our sensitivity analysis experiments anyway.

To help keep the relative importance of real-word data in perspective it may be useful to remember the following *Five Dastardly D's of Data*. Data can be:

1. Distorted: The values of some observations may be changed or not consistently defined. For examples, travel times may include loading and unloading times, which would tend to overestimate the value of a faster vehicle, or a product demand data may include only backlogged orders, ignoring customers who refused to wait.

2. Dated: The data may be relevant to a system that has or will be changed. Perhaps factory data was collected for an older process or using last year's product mix.

3. Deleted: Observations may be missing from the data set. This might be because the data was collected over an interval of time, and events such as machine failures simply did not occur during the study period. Medical trial data might be censored by patients dropping out of the study for various and unknown reasons.

4. Dependent: Data may be summarized (i.e., only daily averages are reported). This may remove critical cycles or other trends in a data sequence or hide relationships between different sequences. For example, data from a surgical unit might give very accurate estimates of the distributions of preparation, operation, and recovery times. However, it may

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

fail to capture the fact that some procedures will tend to have large values for all three times while others procedures may tend to have all small values.

5 Deceptive: Any of the first four data problems might be intentional.

9.8 Variance Reduction

It is often possible to obtain significantly better results by using the same random number streams for different simulation runs. For example, you might want to compare the performance of two different systems. When doing so, it is a good general experimental technique to make "paired" runs of each system under the same conditions. In simulations you do this by using the same random number seed in a run of each system. This technique, called using common random numbers, extends to more than two alternative systems, You would re-use the same seed for a run of each system. To replicate, choose another seed and run each system again. This technique reduces the variance of estimated differences between the systems.

Another example where re-using random number seeds can help reduce the variance of the output applies when making two runs of the same system. As before, you use the same seeds for both runs in the pair. However, for the second run, use `1-RND` where `RND` was used before. If `RND` is a random number, then `1-RND` is also. Furthermore, there is a perfect negative correlation between `RND` and `1-RND`. If `RND` is a small random number, `1-RND` will be large; if `RND` is large, `1-RND` will be small. The pair of runs using `RND` and `1-RND` are called antithetic replications. The hope is that the negative correlation between the input streams will carry over to the output. If one run produces an output that is unusually high, its antithetic run will have an output that is unusually low. When the antithetic replicates are averaged, a run with an unusually high result is canceled by its antithetic replicate having an unusually low outcome and vice versa.

Re-using random number seeds falls under the general category of variance reduction techniques. For a discussion of common and antithetic random numbers as well as some other techniques (which tend to be much less successful in practice), see the text by Bratley, Fox, and Schrage (1987).

The chances that the beneficial results of correlated input streams carry over to the output are greater if the runs can be synchronized as much as possible. That is to say, we want any unusual sequence of random numbers in a run also to be used in the same manner in its commonly seeded replicate(s). For example, if one run in a queueing simulation has an unusual sequence of long service times that causes the system to become very congested, we would like its antithetic replicate to have an unusual sequence of short service times that reduces congestion in the system. We would like all systems using common random numbers to have the same experiences. Synchronization of runs is generally improved if we use different random number streams exclusively for different stochastic components of our simulation. For example, in a queue we might use one stream to generate interarrival times and another stream to generate service times. Thus, we will want to use more than one sequence of random numbers in our simulation. A detailed example is in Appendix B.18.

9.9 Using Multiple Random Number Streams (Development licensees only – others see Appendix B)

Function definitions in C make it very easy to change your SIGMA random number stream to a "vector" of random number streams. We discuss how to generate C simulation programs in Chapter 11. In your SIGMA-generated C code, replace `RND` with `RND[I]`, where `I` is an integer indicating which stream you want to use. For example, to draw a random number from stream 3, replace `RND` with `RND[3]`. Then "vectorize" your library functions by replacing `rndsd` with `rndsd[i]` and `RND` with `RND(I)` in `SIGMALIB.C` (for development licensees only) Note that `RND` is now a function and `rndsd` becomes an array. If you are using three different random number streams in your model, you would make two changes in the library header file for your C compiler (`SIGMALIB.H`, `SIGMAFNS.H`). The two replacement lines would be:

```
long rndsd[3]; /*makes rndsd a vector*/
#define RND(j) ((float)(rndsd[j] = lcg(rndsd[j]))*4.656612875e-10)
```

Like before, you still would have to read in the seeds for each stream when you run your model. You can always substitute the random number generator that comes with your compiler for `RND`. Appendix B.18 gives a general approach.

9.10 Methods for Generating Random Variates

In this section, techniques for generating random variates are presented. Included among the techniques is the generation of non-homogeneous Poisson processes.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

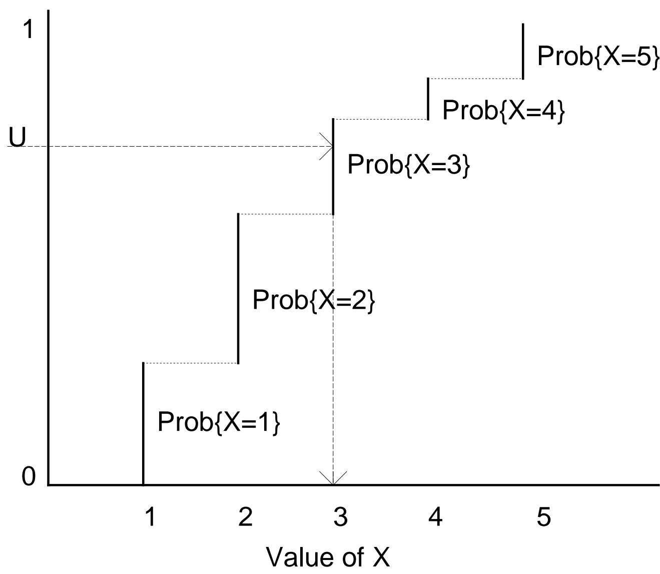
9.10.1 Distribution Function Inversion

The cumulative distribution function, $F_X(x)$, for the random variable, X, is the probability that the value of a random variable will be less than or equal to the function argument,

$$F_X(x) = \text{Prob}\{X \le x\}.$$

Discrete valued random variables can be generated by partitioning the interval between zero and one into sections whose widths are equal to the probability of each value. The value of the random variable is determined by the interval in which a pseudo-random number, U, falls. The probability of U falling in an interval is equal to the width of the interval, which, in turn, is the probability of the corresponding value occurring. This is equivalent to inverting the cumulative distribution function as illustrated in Figure 9.2.

Figure 9.2: The Cumulative Distribution Function of a Discrete Random Variable



The same technique can be applied to continuous valued random variables. For example, the cumulative distribution for an exponential random variable is,

$$y = F_X(x) = 1 - e^{-x/\mu}$$
$$\Rightarrow F_X^{-1}(x) = -\mu * ln(1 - y).$$

So, the inverse distribution function of a uniform random number will generate an exponential variate as

$$X = F_X^{-1}(U) = -\mu * ln(1 - U) \approx -\mu * ln(U).$$

When the cumulative distribution function is easily inverted, this technique is recommended. Unfortunately, not very many of the more commonly used probability distributions have easily inverted cumulative distribution functions. Exceptions that are particularly useful are order statistics from a sample of uniform random variables.

Order Statistics: Order statistics are sorted samples; the minimum order statistic is the smallest value in a sample. Suppose we want to know the first time that one of several identical independent components will fail. We could generate the lifetimes of each component and sort them to find the shortest. If there are many components, it would be easier to generate the minimum order statistic from the lifetime distribution.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Order statistics can be generated by evaluating an inverse cumulative distribution function at the corresponding uniform order statistic. The ith smallest of K uniform random variables has a beta distribution with parameters i and $K-i+1$. A common special case is where we want the smallest of K independent values of a random variable. The cumulative distribution function of the smallest of K independent uniform random variables is,

$$F_X(x) = \text{Prob}\{\text{smallest of } K \text{ independent Uniforms} \le x\}$$

$$= 1\text{-Prob}\{\text{smallest of } K \text{ Uniforms} > x\} = 1\text{-Prob}\{\text{all } K \text{ Uniforms} > x\}$$

$$= 1-(1-x)^K$$

Therefore,

$$F_X^{-1}(U) = 1-(1-U)^{1/K}.$$

If the lifetime of each of K independent components has an exponential distribution, the distribution of the time until the first component fails is equal to

$$X = -\mu * \ln(1-(1-(1-U)^{1/K})$$

$$= -\mu * \ln\{(1-U)^{1/K}\}$$

9.10.2 Other Methods

Other methods for generating random variates include acceptance/rejection, composition, and special relationships. Like inversion, these other methods are not always possible to use in their pure form, and special algorithms that combine these methods have been invented. The reference by Devroye (1986) contains many of these algorithms.

Acceptance/rejection involves bounding the probability density and generating a point uniformly within this bounding area. If the generated point falls below the density function, the horizontal value of the generated point is used as the random variate. If the point falls outside the density function, the generated point is rejected and a new point is tried. This is continued until an accepted point that falls within the region under the density function is found. This is analogous to throwing uniform points onto the bounding region and accepting those points that fall below the probability density. The algorithm for generating beta variates, `beta_fn`, in the SIGMA C library uses an acceptance/rejection algorithm due to Cheng (1978).

Composition involves breaking the probability distribution into regions from which it is relatively easy to generate variates. One of these regions is selected with the probability in that region and the variate is generated from that region.

Special properties exploit relationships between different types of random variables. Some commonly used relationships include generating an Erlang variate as a sum of exponentials, generating a geometric variate as the integer part of an exponential, generating a chi-square variate as the sum of squared normal variates, and generating a Poisson as the count of exponential variates whose sum falls within an interval. (When the Poisson rate is large, there are better methods for generating Poisson variates given in the references).

9.10.3 Generating Non-Homogeneous Poisson Processes

Random arrivals to a service system can often be modeled using a Poisson process. This process has proven to be a rather good model for many processes that occur naturally. It is also used for other types of exogenous events that drive a simulation model, such as equipment failures or flaws occurring in a piece of fabric. The parameter for a Poisson process is its rate, λ, expressed in the number of events per time unit (e.g., customer-arrivals/hour or flaws/square-foot).

There are numerous methods for generating Poisson processes which exploit different properties of these processes. The fact that the *times between Poisson events have exponential distributions* can be used to simply make the delay time for a self- scheduling edge have an exponential distribution with a mean equal to the inverse of the Poisson rate. For example, if the arrivals to a queue are going to be modeled as a Poisson process with an rate equal to `RATE`, then

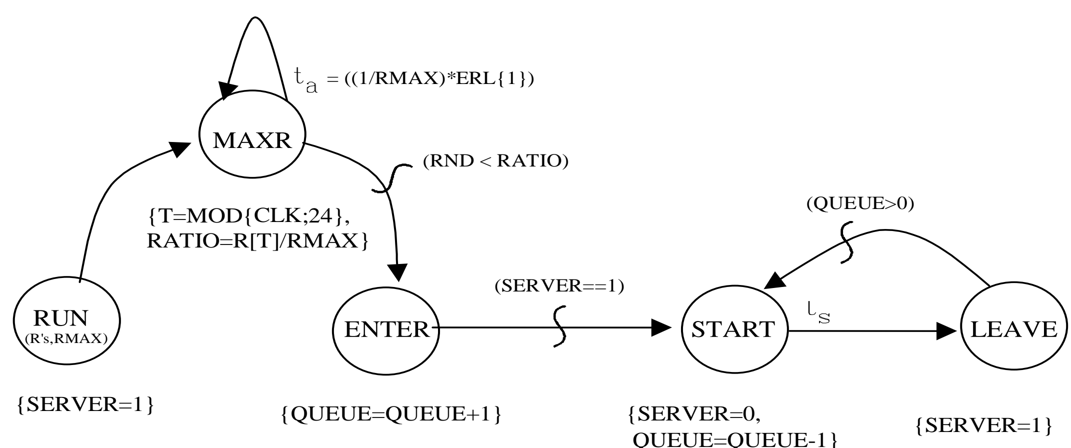
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

the edge delay time for the self-scheduling edge that models sequential arrivals would have a delay time of `(1.0/RATE)*ERL{1}` (recall that Erlang-1 and exponential distributions are same).

If we know the total number of Poisson events that occur in an interval of time, we can better match the process by conditioning on this knowledge. We do this using the fact that the distribution of the times of *K Poisson events in an interval have the same distribution as K uniform random numbers on the same interval*. Say we know that *K* Poisson events occurred in an interval between 0 and T. We can generate the minimum order statistic of *K* uniforms over the interval for our first event time, T_1. The second event time will be the minimum of the remaining *K-1* uniforms distributed over the remaining interval between T_1 and T. The rest of the *K* events in the original interval can be generated in this manner.

Unfortunately, processes in the real world do not often occur with a constant rate. The arrival rate may be relatively high during a rush hour and slack (or zero) late at night. A Poisson process with a changing rate is called a *non-homogeneous* Poisson process. An easy way to model a non-homogeneous Poisson process is by a technique called "thinning" (Lewis and Shedler, 1979). Here we simply generate Poisson events at the maximum rate and keep them with a probability proportional to the current rate. To illustrate: assume we know that during an 24-hour day, customers will arrive at our carwash with the following hourly rates: `R[0]`, `R[1]`,..., `R[23]` (the rate can be zero if the facility is closed). Let `RMAX` be the maximum of these rates. The event graph to generate Poisson arrival events according to this daily demand cycle is called `NONPOIS.MOD` (it is shown in Figure 9.3.

Figure 9.3: Non-Homogeneous Poisson Arrivals to our Carwash Model



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

9.11 Exercises

9.11.1 Chaos

A simple example of a "chaotic system" is the recursive equation: X = R*X*(1-X).

If you start off with a particular value of X, if R is set high enough, it becomes impossible to predict the value of X very far into the future. Starting the system with X = 0.2, try values of R = 0.5, 1, 2, and 4. Does the system behave differently for different values of R? Does "chaos" describe the behavior? (Do you think chaotic systems might make good pseudo-random number generators!)

9.11.2 Generating Discrete Random Variates

(a) Consider the probability distribution function: P(X=i)=i/6 where *i* =1,2,3

i. Create an event graph using RND to generate 25 random variates from this distribution.

ii. Create an event graph using DISK{} and a data file to generate 25 random variates of this distribution.

b) What is the probability distribution function of X?

U=RND and X=(U>0.2)+(U>0.5)

9.11.3 Generating Continuous Random Variates

(a) Let A and B be two random variables uniformly distributed between 0 and 1. Let X be the larger of A and B. Create an event graph which generates 20 values of X (try to be clever).

(b) Consider the following probability distribution function: *f(x)* = *x*/8 0<=*x*<=4
Create an event graph using RND to generate 25 random variates of this distribution.

(c) What is the delay time corresponding to an exponential rate with parameter 5?

(d) What does the probability distribution function of X look like if X is given by the following? U=RND and X=(2*RND)*(U<0.5)+(4+RND)*(U>=0.5)

(e) If U is a random variable with a uniform distribution between zero and one, what is the distribution of V=1/2-U/2?

(f) What will the following state changes do in SIGMA to the value of X? That is, give the value(s) of X and tell how the(se) values might occur. [Assume that all variables are REAL valued (i.e., floating point)].

```
R=RND
X=(RND<0.2)+(RND<0.5)*2+(RND<0.7)*4
```

g) What does the following SIGMA state change do?

```
X=(RND>0.5)*1+(RND<=0.5)*2
```

9.11.4 Generation of Minimum Statistics

There are 100 identical components operating independently in a system. Each component has a lifetime, X, that has an exponential distribution with a mean of 6 days. Generate the time until the first component fails from a single uniform random number, RND? (Hint: The minimum of N uniform random variables is given in Section 9.10.1, and X=-M*LN{RND} is an exponential random variable with mean, M.)

9.11.5 Testing Variate Generators

SIGMA has build-in generators for uniform (RND), gamma (GAM), and normal (NOR) random variates. Using the methods in texts referenced in this book (Bratley, Fox, and Schrage and Law and Kelton), perform at least three tests of these functions (at least two of the tests should be quantitative). Program and test generators for lambda variates.

9.11.6 A Variance Reduction Technique

Two runs are made of a simulated queue. The random number stream that was used to generate interarrival times in the first run is used to generate the service times for the second run. The random number stream that was used to generate service times in the first run is used to generate the interarrival times for the second run. Will the average customer waiting times from these two runs tend to have zero, positive, or negative correlation? Explain why.

9.11.7 Dependent Processes

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Run CARWASH.MOD where the interarrival times and service times are exponentially distributed with the same means as before but now follow EAR models with lag-one correlation of $p = 0.7$. Compare this system with CARWASH.MOD having deterministic service and CARWASH.MOD having *independently* distributed exponential service.

9.11.8 A Simple Recursion for Queue Times

Generate 90% confidence intervals for the average waiting time, E[W], in an M/M/1 queue with traffic intensity of 0.9. Use the recursion, W=MAX{W+S-A;0} starting with W=0, where S is the service time of the ith customer (exponential with mean =1) and A is the ith customer interarrival time (exponential with mean 1/0.9).

9.11.9 Dependent Input Data

You are building a model for a chain of automatic carwashes. These carwashes have deterministic service rates. A sample of car arrivals indicates that the times between car arrivals has an exponential distribution. Unfortunately, the people collecting the demand data simply tabulated a histogram of the interarrival times rather than recording the actual sequence of arrival times.

(a) What problem(s) do you anticipate this might cause in getting a valid model for the customer arrival process? Specifically, what information are you missing that you wish you had? Why?

(b) What if you are also told that the lag-1 serial correlation between successive interarrival times is 0.7? Given that an arrival event just occurred and the simulated time is CLK, how can you generate the time of the next car arrival (call this time T) using all of the information you have?

(c) Why might your approach not be acceptable to the people who own this chain of carwashes?

9.11.10 Generating Points in a Plane

Generate a sample of N independent points uniformly scattered over the area in the two-dimensional region in the plane bounded by the horizontal axis and an arbitrary function.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

10

Graphical & Statistical & Output Analysis

In keeping with our philosophy of utilizing pictures whenever possible, several graphical methods of presenting simulation results have been incorporated into the SIGMA software. The graphical output charts available to you while in SIGMA are line plots, step plots, scatter plots, array plots, histograms, autocorrelations, and standardized time series. This chapter discusses these graphical output plots and concludes with an explanation of standardized time series.

10.1 Keeping a Perspective

It is easy to become overwhelmed by the information produced by a simulation model. Different types of simulation output are most useful during different stages of a simulation study. Animations, graphs, and statistics all have their appropriate roles to play.

During the initial development and testing of the simulation model, animating the simulation logic while running SIGMA in Single Step or Graphics run mode is the most valuable. The logical animation in SIGMA is different from the physical animation of a system. Physical animations are useful in selling the simulation to prospective users and for catching gross logic errors in the simulation model. Physical animations using SIGMA are discussed in Chapter 8.

The Translate to English feature of SIGMA (found under the File menu) is an extremely effective tool for catching program logic errors or communicating the details of a model to persons not familiar with simulation. If the SIGMA-generated English description of your simulation is nonsense, it is likely that the logic in your model will not make sense either.

When evaluating alternative system designs at a high level, charts of the output are most useful. Here we are comparing the performance of very different systems (e.g., manual operations versus automated ones). Plots of the output not only offer information on overall system performance but also on the dynamics of the system. We can see, for example, if the manner in which we initialized the variables in our simulation has an inappropriate influence on the output. We will say more about the bias caused by initializing variables later in this chapter.

Once a particular design has been tentatively selected, it is important to do detailed sensitivity analysis and design optimization before a final recommendation is proposed. Here we are going to run a great many replications with different settings of the factors and parameters in our model. It is neither fun nor particularly informative to watch hundreds of different animations or look at hundreds of output plots. In the detailed design phase of a simulation study, numerical summaries of system performance in the form of output statistics are the most appropriate form of simulation output.

Finally, once a design has been finalized, the most effective form of output is a physical animation that lets people more fully understand the changes being suggested. Charts, statistical summaries, and the English description of your model can also be effective in helping you sell your ideas. The above discussion is summarized in Table 10.1.

Table 10.1: Typical Phases of a Simulation Project and Predominant Form of Output.

Phase of the Study	Predominant Form of the Output
Model building and validation	Logical animations and English Descriptions
System evaluation	Charts and plots
Detailed design	Statistics
Implementation	Physical animations

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

10.2 Elementary Output Charts

The Output Plot dialog box is discussed in detail in Chapter 4. In SIGMA there are five basic output plots and two plots for advanced analysis. The basic charts are:

1. Step plots, which show the values of traced variables during a simulation run.
2. Line plots, which are similar to step plots except straight lines are drawn between successive data points.
3. Array plots, which show values of each element in an array.
4. Scatter plots, which show the relationship between pairs of traced output variables.
5. Histograms, which count the relative frequencies that different values of a variable occur.

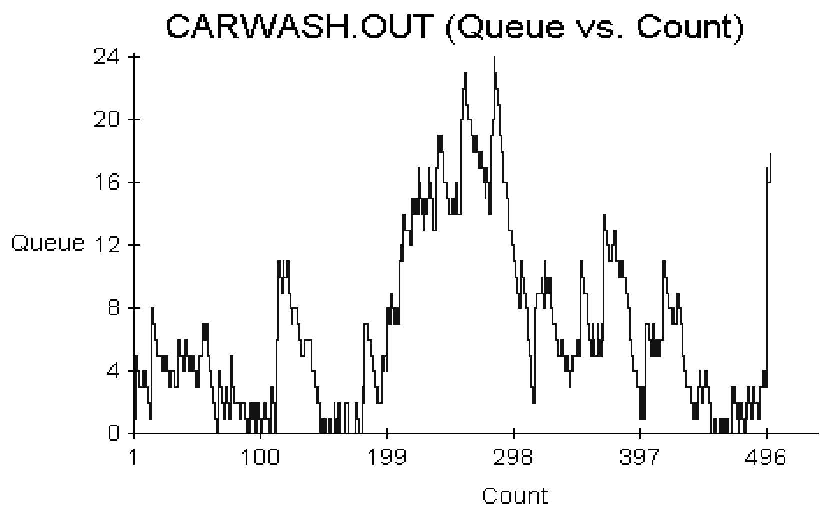
More advanced output analysis is possible using the following charts:

6. Autocorrelation functions, which shows dependencies in the output.
7. Standardized time series (STS), which can be used to detect trends.

10.2.1 Step and Line Plots

Step plots and line plots (also called index plots) are by far the most common form of graphical simulation output. Here we see how one variable changes during the simulation run. The variable of interest is chosen for the vertical axis of the plot and an indexing variable is chosen for the horizontal axis. The indexing variable is often simulated time, but other variables (such as customer identification number) might be used; the indexing variable should not decrease in value during the run for an index plot to make much sense. Figure 10.1 shows a line plot of the queue length in our simulated carwash (exponentially distributed service times were used here).

Figure 10.1: Queue Length as a Function of Customer Number



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

10.2.2 Array Plots

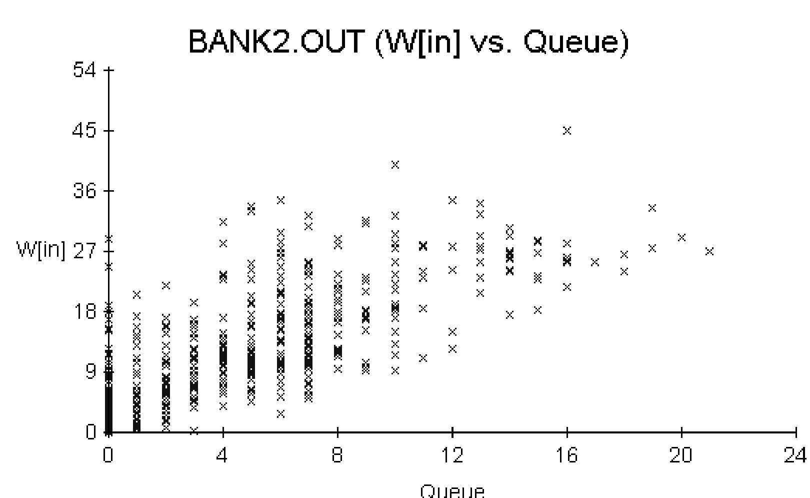
Array plots show the maximum and current values of each element in an array. This looks like a series of "thermometers". This is useful, say, when you have an array of queues. You can see how each queue size changes and effects the other queues. (See Figure 10.2) A good example is to look at the variable `Q` in the model, `FLOWSHOP.MOD`.

Figure 10.2: Array Plot of Jobs in each Queue in `FLOWSHOP.MOD`

10.2.3 Scatter Plots

When two variables are of equal importance, they can be chosen as the two axes of a scatter plot. A point is plotted for every observed pair of values for these variables. If the points tend to fall along a line with a positive slope, the two variables are likely to be positively correlated. (Small values of one variable are observed along with small values of the other variable and large with large.) Similarly, if the points in a scatter plot tend to fall along a line with a negative slope, negative correlation between the variables should be suspected. Figure 10.3 shows a scatter plot of the waiting time of each customer for our simulated bank in Chapter 5 and the number of customers in line when each customer departed; the expected positive correlation is evident.

Figure 10.3: Scatter Plot of Waiting Time and Line Length in a Bank

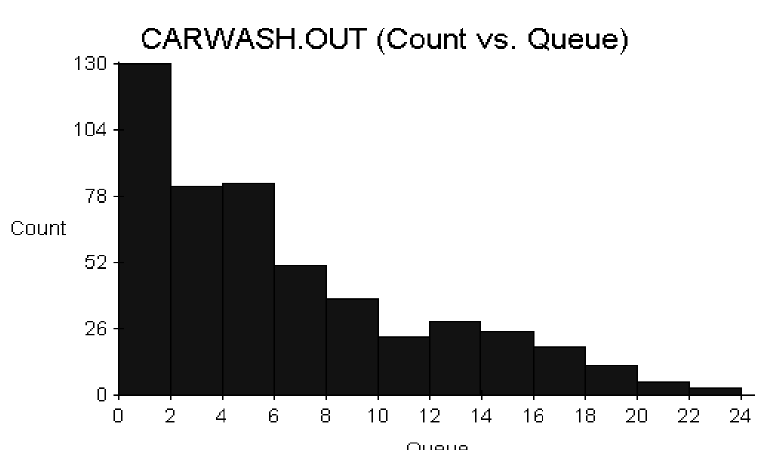


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

10.2.4 Histograms

Histograms show counts of the number of times the observed values of a variable fall within a specified interval. These counts show the relative frequency that values of a variable are observed. Figure 10.4 shows a histogram of the number of customers in the carwash model with exponentially distributed interarrival times.

Figure 10.4: Histogram of Number of Customers in Carwash Queue



10.3 Advanced Graphical Analysis

10.3.1 Detecting Trends using Standardized Time Series

Using a standardized time series (STS) plot, it is possible to detect trends in the output that might not otherwise be visible. The STS plot should appear to be symmetric about zero if there is no trend in the data (adequate initial truncation). When there is a trend, the STS plots will be pulled either above (increasing trend) or below (decreasing trend) the zero line. STS can also be used for other types of inference such as confidence interval estimation. The STS plots in SIGMA are the unscaled standardized time series for the selected output measurement. The best way to learn how to use STS plots is to look at a few simple output series. Try generating a sequence of independent random variables, say, with the one state change `X=NOR{0;1}`. Look at the line plot of the successive values of `X` and the corresponding STS plot. Now add a trend to the data, say, `X=CLK*NOR{0;1}` and look at the difference in the same two plots. Do this for some more interesting and subtle trends (exponential decay, quadratic, sudden shift, etc.) and see how the STS behaves. You will find that familiarity with the behaviors of STS plots is a valuable visual tool for output analysis.

Standardizing a time series is similar to the familiar procedure of standardizing or normalizing a scalar statistic. Standardizing a scalar statistic, such as a sample mean, involves centering the statistic to have a zero mean and scaling its magnitude to generic units of measurement called standard deviations. Limit theorems can be applied that give us the asymptotic (large sample) probabilistic behavior of correctly standardized statistics under certain hypotheses. This limiting model for scalar statistics is typically the standard normal probability distribution. This model can be used for statistical inference such as testing hypotheses or constructing confidence intervals. Here we extend this concept to the standardization of an entire time series. More information about STS can be found in Section 10.5, Standardized Time Series.

The value of standardizing time series comes from the fact that the same mathematical analysis can be applied to series from a variety of sources. Thus, the technique of standardization serves as a mathematical surrogate for experience with the data under study. No matter what the original time series looks like, the standardized time series will be familiar if certain hypotheses are correct. An unusual appearance of a standardized time series can be used to conclude that these hypotheses are not valid. The statistical significance of these conclusions can be computed in the same manner as with standardized scalar statistics.

To illustrate using STS plots to detect trends, consider first the STS plot in Figure 10.5. Since this plot is mostly negative, a clear downward trend in the data. is evident.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 10.5: STS Plot Indicating a Downward Trend in the Data

A line plot of the actual data is shown in Figure 10.6, where the downward trend indicated by the STS plot is at best only marginally apparent.

Figure 10.6: Plot of the Raw Data Series for the STS Plot of Figure 10.5

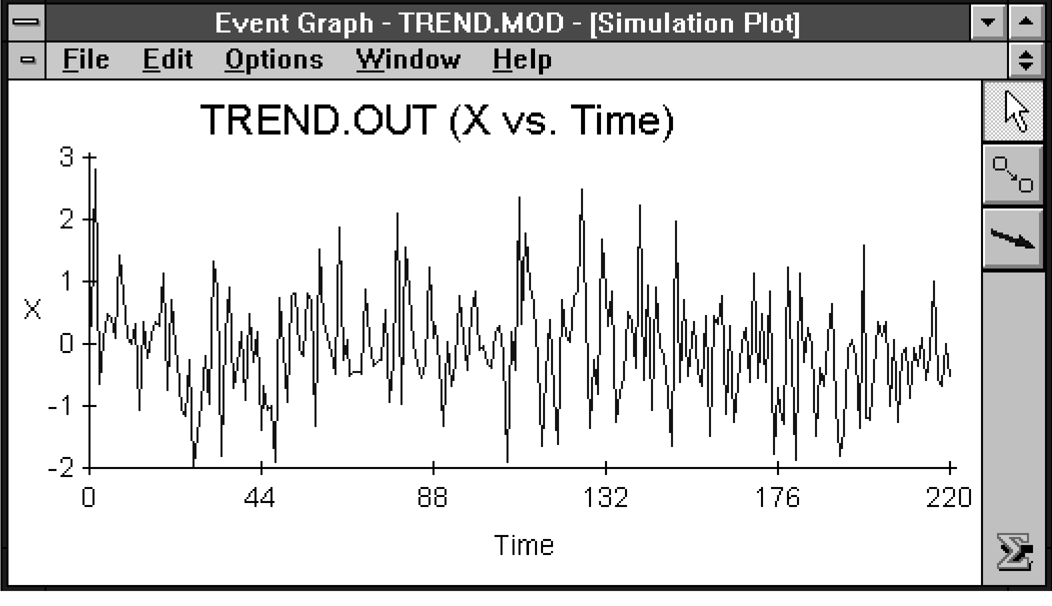


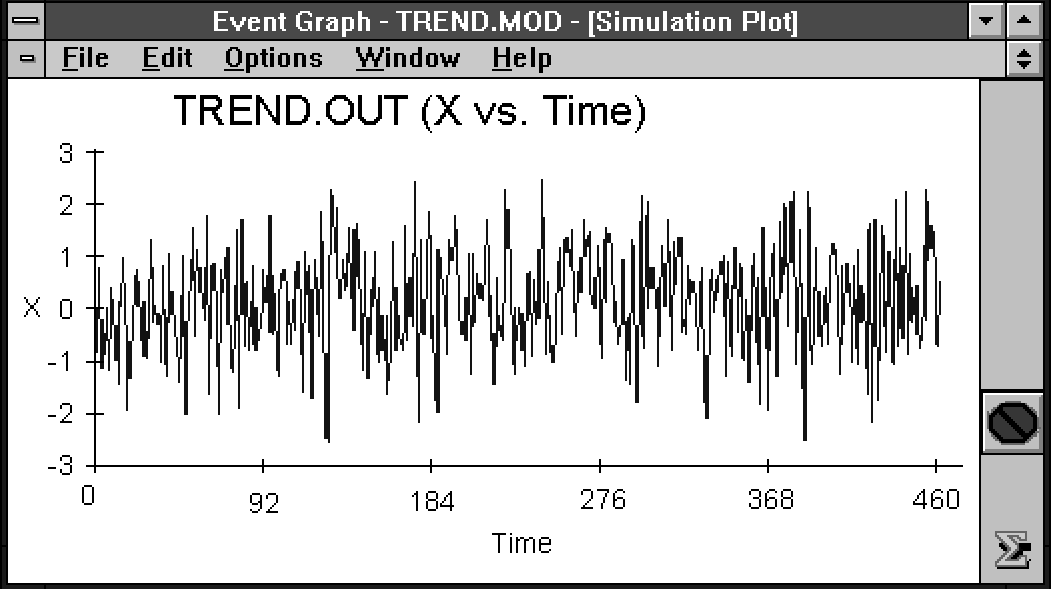
Figure 10.7 is an STS plot that indicates the presence of a strong increasing trend in the data. The STS plot is pulled in the positive direction by this positive trend in the data. Figure 10.6 should be compared to Figure 10.7, where a negative trend was indicated.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 10.7: An STS Plot indicating an Increasing Trend in the Data.

The raw data for the STS plot in Figure 10.7 is plotted in Figure 10.8, where the increasing trend in the data is again only slightly detectable.

Figure 10.8: Raw Data for the STS Plot in Figure 10.7



The sensitivity of STS plots to trend has a down-side: they can indicate a trend which may disappear as more data is collected. However, given the potential seriousness of simulation initialization bias causing an artificial trend in the output, it seems better to be able to detect trends easily at the risk of falsely indicating a non-existent trend.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

10.3.2 Dependencies

The autocorrelation function is a plot of the correlation between two observations in the same output series as a function of how far apart (lag) the observations are. For example, the lag 1 autocorrelation function is the sample correlation between two adjacent output measurements. The autocorrelation function should drop off sharply at a lag of 1 if the observations are not correlated. This would indicate that the batch size is large enough to remove correlations between successive batched means and initial transient output observations have been truncated. Crude 95% confidence bounds at $\pm 1.96/\sqrt{n}$ can be used as a very rough guide (Brockwell and Davis, 1987).

The standardized time series (STS) plots can also be used to visually assess whether or not there is significant positive serial correlation in an output series. The more jagged the STS plot appears, the less serial dependency in the output. If the STS plot is smoother than you are typically used to seeing, you can suspect either a serious trend in the data or significant positive serial dependency between successive observations of the output.

10.4. Using Statistics

Batching is where non-overlapping, adjacent, equal-sized groups of data are averaged. The resulting series of batched means will often be more independent, less erratic, and have an approximately normal distribution. If there are about twenty batches, there is a sufficient number of degrees of freedom for most inferences (Schmeiser, 1983). Batched means are computed after truncation. From within SIGMA, you are limited to 10,000 batches of output data.

Sufficient statistics and maximum likelihood estimators are used for common inference. The assumptions behind the statistics are probably more important than the numbers themselves and should be understood before too much faith is placed in their values. When looking up the simple formulas, you can review the assumptions. Do not let an unfamiliarity with statistics prevent you from looking at the charts. Averages of squares (scatter plots) are sufficient for batched means confidence interval estimation and other inferences. Approximately twenty to thirty large batches are recommended (Schmeiser, 1983). Batching tends to make the output better approximate an independent sample from a normal distribution.

To illustrate the effects of batching, 1000 observations of the EAR process discussed in Chapter 9 were simulated. The mean of the process, μ was 10. The sample mean was 9.43. The autocorrelation function of this process and the histogram showed us that this output series does not appear to be independent nor does it have the characteristic "bell shaped" distribution function expected of normally distributed observations. We also saw that the correlation between the observations appears to slowly decrease as the lag (time interval) between the observations increases. There appeared to be significant correlation between neighboring observations at lag 1. The sample standard deviation was 9.339. (This is approximately equal to the sample mean, as expected from exponential data.)

To see the effects of batching, averaged groups of 10 observations in a sample of 5000 from the same process. The sample mean of the batched process was 9.739.

Even with a batch size as small as 10, we saw from the autocorrelation plot that the data appears now to have very little serial correlation. We also saw that the sample standard deviation is 4.838. The histogram of the batched means showed the data is beginning to look like it came from a normal, bell-shaped distribution (as expected from the Central Limit Theorem of statistics). With averages of only 10 observations from a highly-skewed dependent exponential distribution, we begin to approximate an independent normal data set. The mean and variance of the batched process can safely be used to form a batched means confidence interval for the mean of the process, based on the usual *t*-statistic with 499 degrees of freedom. (We can use the normal approximation to the *t* distribution for such a large degrees of freedom parameter.) For example, a 90% confidence interval for the process mean, μ, is found to be,

$$\overline{X} - t_{499,0.95}\frac{S}{\sqrt{n}} \le \mu \le \overline{X} + t_{499,0.95}\frac{S}{\sqrt{n}}$$

Substituting our statistics into the above interval estimation formula, we get the following confidence interval for the true mean of the process.

$$9.739 - 1.645\frac{4.838}{\sqrt{500}} \le \mu \le 9.739 + 1.645\frac{4.838}{\sqrt{500}}$$

$$9.383 \le \mu \le 10.09$$

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The true value of $\mu = 10$ was within this confidence interval, as it was for the wider 90% confidence interval of $9.696 \le \mu \le 10.763$ constructed with 1000 unbatched observations.

In addition to averages, time averages, and standard deviations, The STS area, `A`, `STS{X}` can be used for confidence intervals as described by Goldsman and Schruben (1984). For *large* samples, `A` might behave like the standard deviation times a chi-square random variable with 1 degree of freedom independent of the sample mean. (Use independently seeded replications to increase the degrees of freedom.) STS is discussed in the next section.

The (unscaled) STS maximum, `M`, located at the `K`th of `N` batches might behave like the standard deviation times `(N-K)K/N` times a chi-square random variable with 3 degrees of freedom if `N` is very large (and the batch size is moderate). See Goldsman and Schruben (1984). This can be used for computing the STS maximum confidence interval estimator.

10.5 Standardized Time Series

As a guide to standardizing a time series, we will first review the procedure of standardizing a scalar statistic. We will use the familiar *t*-statistic as an example. The data will consist of *n* observations

$$Y_1, Y_2, \ldots, Y_n$$

that are independent and have identical normal distributions. We wish to make inferences about the unknown population mean, μ. The average of the data sample

$$\overline{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i$$

will be the statistic used for these inferences. The population variance, σ^2, is an unknown nuisance parameter.

Standardization involves the following steps.

1. *Center the Statistic:* The population mean, μ, is subtracted from the sample mean giving the random variable, $\overline{Y}_n - \mu$, which has an expected value of zero. (Strictly speaking, this difference between the average and the mean would not be called a "statistic" since it includes the parameter, μ.)

2. *Scale the Statistic Magnitude:* Since statistics can come in an almost endless variety of measurement units, we will need to express the statistic in a common unit of measurement called a standard deviation. The magnitude of the statistic is scaled by dividing by $\sqrt{\sigma^2/n}$. Our statistic is now

$$Z_n = \sqrt{n}(\overline{Y} - \mu)/\sigma$$

which is our standardized statistic. Standardized sample means will all have the same first two moments. The unknown scaling parameter, σ, can be either estimated or cancelled out of a ratio statistic; the cancelling out of this parameter in a ratio statistic is the more common approach and is followed here.

3. *Cancel the Scale Parameter:* The data is aggregated or batched into *b* exclusive adjacent groups of size *m* (if necessary, discard data from the front of the run so that $b = \lfloor n/m \rfloor$). The average of each batch is denoted as $\overline{Y}_{(i,m)}$, $i = 1, \ldots, b$. The usual unbiased estimator of the variance of the batched means is

$$S^2 = \frac{1}{(b-1)}\sum_{i=1}^{b}(\overline{Y}_{(i,m)} - \overline{Y}_n)^2$$

Inferences about the parameter, μ, are based on the random *t*-ratio,

$$T_{b-1} = \sqrt{b}(\overline{Y}_n - \mu)/S$$

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

4. *Apply Limit Theorems:* The limiting distribution of T_{b-1} is known. As $n \to \infty$ (making $m \to \infty$ since b is fixed), the distribution function of $(b-1)S^2 / \sigma^2$ converges to that of a χ^2 random variable with $b-1$ degrees of freedom. As $n \to \infty$, $\overline{Y}_n$ will converge to the constant μ from the law of large numbers. Also, from the Central Limit Theorem of statistics, the distribution function of Z_b will converge to that of a standard normal random variable. Thus, the distribution function of the ratio, T_{b-1} (being a continuous mapping) will converge to that of a t random variable with $b-1$ degrees of freedom. The unknown scaling constant, σ, is cancelled out of the ratio.

5. *Use the Limiting Probability Model for Inference:* The limiting distribution of T_{b-1} can be used for statistical inference and estimation.

The concept of standardization can be applied to an entire time series. The original series of observations is transformed into a standardized series of observations. We will hypothesize (and test) that the series is stationary. We also assume that there is some minimal amount of randomness in the process; however, we do not assume that the data is independent. The mathematical assumptions needed are given in a paper by Schruben (1983), where it is argued that many simulations on a computer will meet the imposed restrictions for applicability. Suppose we want to standardize the output from the ith run of a simulation, with i representing the run number. Let

$$\{\overline{Y}_{i,j} : j = 1,...,m\}$$

denote m stationary but perhaps dependent observations in the ith output time series. We will standardize the sequence of cumulative means up to and including the kth observations, given by,

$$\overline{Y}_{i,k} = \left(\frac{1}{k}\right)\sum_{j=1}^{k} Y_{i,j}$$

Similar steps to those in standardizing a scalar statistic are followed in standardizing a time series. The steps in standardization are as follows:

1. *Center the Series:* For run i, the sequence given by

$$S_{i,m}(k) = \overline{Y}_{i,m} - \overline{Y}_{i,k}$$

will have a mean of zero if the series has a constant mean.

2. (a) *Scale the Series Magnitude:* The scaling constant for dependent sequences (independent of the run i) that we use is defined as

$$\sigma^2 = \lim_{m \to \infty}[m\text{Var}(\overline{Y}_{i,m})]$$

which is just the population variance in the special case of independent identically distributed data. Magnitude scaling is done by dividing $S_{i,m}(k)$ by $\sigma\sqrt{m} / k$. The scaling constant is again unknown but will cancel out of our statistics as before.

There is one additional step required that was not necessary in the scalar standardization case. Different time series can be of different length, so we must also scale the length of the series. Thus, we have the additional step:

(b) *Scale the Series Index:* We will define the continuous index, $t = k / m$. Our previous index is thus given by $k = \lfloor mt \rfloor$. We also add the starting point $S_{i,m}(0) = 0$ so that the standardized series is 0 at t =

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

0 and $t = 1$. The result is that all standardized time series have indices on the unit interval and start and end at zero. We now have what we will call a standardized time series given by

$$T_{i,m}(t) = \lfloor mt \rfloor S_{i,m}(\lfloor mt \rfloor) / (\sqrt{m}\sigma)$$

where $\lfloor \cdot \rfloor$ is the greatest integer function.

3. *Cancel the Scale Parameter:* There are several functions that might be considered for the denominator of a ratio that cancels σ. We will consider here only one such function, the sum or limiting area under the function $T_{i,m}(t)$.

$$A_{i,m} = \frac{\sigma}{m}\sum_{k=1}^{m} T_{i,m}(k/m) = m^{-\frac{3}{2}}\sum_{j=1}^{m}[\frac{m+1}{2} - j]Y_{i,j}$$

4. *Apply Limit Theorems:* The standardized series, $T_{i,m}(t)$, will converge in probability distribution to that of a Brownian Bridge stochastic process. Thus, the Brownian Bridge process plays the role in time series standardization that the normal random variable played in scalar standardization. An important feature of the standardized series, $T_{i,m}(t)$, is that it is constructed to be asymptotically independent of the sample mean, $\overline{Y}_{i,m}$.

There are several functions of $T_{i,m}(t)$ that will also be asymptotically χ^2 distributed. The area, $A_{i,m}$, will have a limiting (as $m \to \infty$) normal distribution with zero mean and variance

$$V = \sigma^2 / 12$$

Therefore,

$$A_{i,m}^2 / V$$

will have a limiting χ^2 distribution with one degree of freedom.

Now consider where each of b independent replications (or b batches of data) are standardized in the manner above. We can then add the resulting χ^2 random variables, $A_{i,m}^2 / V$, for each replication or batch to obtain a χ^2 random variable with b degrees of freedom.

5. *Use the Limiting Probability Model for Inference:* In a manner similar to the scalar case, the standardized (scalar) sample mean of all of the data is divided by the square root of $A_{i,m}^2 / V$ over b to form a ratio where the unknown scale parameter, σ, cancels. For large values of m, the distribution of this ratio can be accurately modeled as having a t distribution with b degrees of freedom. The resulting (1-α)100% asymptotic confidence interval for the mean μ is

$$\mu \in \overline{\overline{Y}} \pm t_{b,1-\frac{1}{2}\alpha}(\frac{12}{bn}\sum_{i=1}^{b} A_{i,m}^2)^{1/2}$$

where $\overline{\overline{Y}}$ is the "grand mean" of all of the data in all batches or replications. More complicated, but superior, confidence intervals can be obtained by weighting the standardized time series as in Goldsman and Schruben (1990). The SIGMA function STS{X} is equal to $A_{i,m}$ for the output time series of values of X; this function can be used with the other weightings given in the reference.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Also, each of the replication or batch means can be treated as a scalar random variable and standardized and squared, giving another χ^2 random variable. Due to the independence of $T_{i,m}(t)$ and the $\overline{Y}_{i,m}$'s, these χ^2 random variables can be added, giving a χ^2 random variable with *2b-1* degrees of freedom. This can be considered as a "pooled" estimator of σ^2, which we will denote as *Q*. The same types of inferences can be made for the dependent simulation output series as were applicable in the independent data case. The resulting "*t* variate" is given by

$$T_{b-1} = (\overline{Y}_n - \mu) / \sqrt{Q/n}$$

Theoretical properties of confidence intervals formed using standardized time series are presented by Goldsman and Schruben (1984), which compares the standardized time series approach to conventional methods.

Standardized time series has been implemented in several simulation analysis packages, most notably at IBM (Heidelberger and Welch, 1983), Bell Labs (Nozari, 1985), and General Electric (Duersch and Schruben, 1986). These packages typically control initialization bias (see also Schruben, Singh, and Tierney, 1983, and Schruben, 1982) and run duration as well as produce confidence intervals. Other applications of standardized time series have been to selection and ranking problems (Goldsman, 1983) and simulation model validation (Chen and Sargent, 1984).

The asymptotic arguments above require that the batch size, *m*, become large as *n* is increased. The common method is to allow the batch size to grow as the sample size increases and keep the number of batches, *b,* fixed. Fixing *b* at 10 or 20 seems reasonable in most applications as long as the sample size is large (see Schmeiser, 1983).

The limiting probability model of a *t* random variable has the virtue that it is widely tabulated and has been studied extensively. There exist other limiting models that might be used, but none have been developed to the extent of the *t* model.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

11

Generating Source Code for SIGMA Models and Running Simulations from a Spread Sheet[1]

One of SIGMA's most powerful features is its ability to automatically generate source code for a simulation program from a graphical model. This chapter discusses the generation of very fast source code in C. Also discussed are ways to run large experiments using batch files, how to put your simulator into a spreadsheet and run simulations from a spreadsheet interface, how to run large experiments using batch files, and how to replace SIGMA's pseudo-random number generator. You do not have to know C to use this feature. If, however, you wish to understand the details, refer to Appendix B, which contains suggestions for reading a SIGMA -generated C program. The `SIGMALIB.LIB` library used here currently supports Microsoft Visual C/C++ Version 6.0 (or .NET).

11.1 Generating C Programs from SIGMA Models

SIGMA's ability to generate source code gives you the power and flexibility of using a common language compiler. Of course, SIGMA event graph models can be run directly from within SIGMA. Neither a compiler nor knowledge of C is needed to use SIGMA. However, SIGMA executes models interpretively (translating code as it goes); this is good for interactive model debugging but slow compared to compiled code. In fact, experienced SIGMA users will often rough out the logic of a model graphically, translate the model to C source code, and work directly with the generated source code. The compiled simulation can also be attached to a spreadsheet interface as discussed later in this chapter.

You create C simulations from your SIGMA model simply by clicking your mouse on the Translate command under the File menu and selecting C. You are prompted for the name of the file that will contain your C program. A file name with the extension "`.C`" should be entered. The rest is automatic and very fast. SIGMA-generated C models typically can be compiled and run without any modification. The library of support functions, `SIGMALIB.LIB,` is used in running these programs. Details of this library of functions are given in Appendix B.

Since there is a commonly accepted standard for the C language, your SIGMA-generated C simulation will be relatively easy to compile and run on different computer hardware. An introduction to reading C programs is given in Appendix B. If you do not know C, you might want to read this appendix now and look at SIGMA-generated C code. SIGMA-generated C code is extensively commented and very easy to read. (Comments in C are on lines starting with // or bounded between /* and */.) All keywords in C are in lower-case characters. Code that you create in SIGMA that is specific to your model will be written using upper-case characters. It is easy to recognize your code that was taken from your SIGMA event graph, just look for the upper-case characters. C is a very powerful language, and there are many features of C that are not used in creating SIGMA simulations. There are also a few features of SIGMA that do not translate easily into standard C code; these also are discussed at the very end of Appendix B.

> **Important:** *Code that you created that is specific to your* SIGMA *simulation will appear in upper-case characters. Generic C code is in lower-case.*

SIGMA-generated C models also have been run with the object oriented C++ language. As you become familiar with C and C++, it is recommended that you use your skills to streamline and enrich your SIGMA-generated simulation programs. SIGMA's intent is to give you a working simulation that is also fast; you can then modify your simulation according to your experience and programming skills. Even without modification, SIGMA simulations are often much faster than the same simulations written in other popular simulation languages. Learning to use C and C++ will allow you to bring a great deal of programming power to your simulations.

In addition to vastly improved execution speed, one of the major advantages of translating your SIGMA models into C source code is the ability to use data structures and pointers, which are not used in SIGMA. Arrays, pointers, and structures are introduced in Appendix B. Structures and pointers are used extensively in the SIGMA and the C.

---

[1] Wai Kin (Victor) Chan developed most of the material on Visual Basic

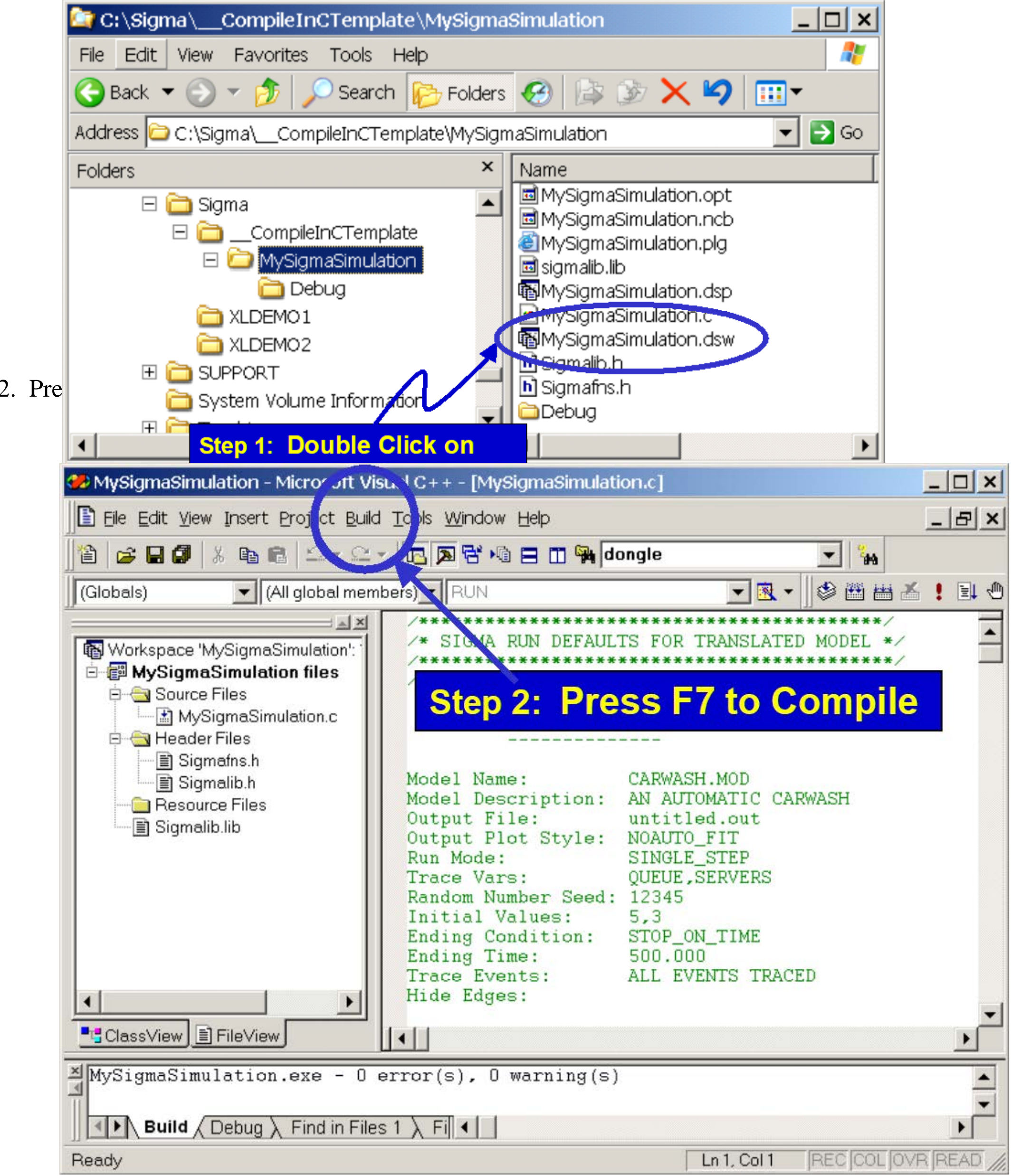
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

11.2 Compiling SIGMA-generated C code

Two-step Compiling shortcut: Details for using Microsoft Visual Studio to modify your C code are in Appendix B. Using Microsoft Visual C/C++ (Version 6 or .NET), if you replace the file named, `MySigmaSimulation.C,` in the subfolder `CompileInCTemplate` with your own sigma-generated C code, you can compile it by simply

Step 1. Run the "workspace definition file", `MySigmaSimulation.DSW`

2. Pre



Then close MSVC and you are done. Your simulation executable will be called `MySigmaSimulation.exe,` and is located in the "debug" subfolder. You can now rename this executable simulation file anything you wish and move it anywhere, as long as the name extension remains `.exe`. Your compiled model can run by simply double clicking on it., The `MySigmaSimulaton.C` file that comes with SIGMA is simply the file, `CARWASH.C,` that has been renamed after being generated from CARWASH.MOD. A sample experiment file `MySigmaSimulation.exp`, discussed next, is also included in the debug subfolder.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

For a concrete example: suppose that we want to run an experiment with our carwash simulation to study the effects of the initial run conditions. The only initial conditions in this model are the random number seed, the run duration, the initial number of cars waiting when the carwash opens each morning (QUEUE), and the number of bays (SERVERS). The simulation, Carwash.exe, was compiled from Carwash.C, which in turn was generated by SIGMA from CARWASH.MOD (Carwash.c was renamed MySigmaSimulation.c, compiled using the two step process given earlier and MySigmasimulation.exe was then renamed back to Carwash.exe).

You run the model by simply double clicking on Carwash.exe and answering several questions at the keyboard, responding to the following dialogue:

```
>[double click carwash.exe]
Running carwash.exe.
Looking for experiment file: carwash.exp
Not found, input data at the keyboard.

OUTPUT FILE (Enter File Name with Path):
[carwash.xls][←Enter]

WARNING: File carwash.xls must NOT be open!!
 If file does not exit it will be created.
Do you want new output to be appended to this file? (yes/[no])
[y][←Enter]

RANDOM NUMBER SEED (Enter Integer Between 1 and 65534):
[12345][←Enter]

STOPPING TIME (Enter number of time units until run termination):
[100][←Enter]

TRACE (1 = Trace Events, 0 = Summary Only)
[1] [←Enter]

Enter initial value for QUEUE:
[5][←Enter]

Enter initial value for SERVERS:
[2][←Enter]
```

If you are using an unlicensed copy of SIGMA, your run will be allowed to continue after a friendly reminder that non-educational use is prohibited without a license (☺). Your output will appear in a new spreadsheet called carwash.xls where you can do your analysis. The above process can be automated using Batch files as shown next.

11.3 Running Large Experiments using Batch Files

It is easy to use a SIGMA-generated C simulation to make an unattended series of runs with different input settings and place the outputs in different output files. This allows you to run large experiments with many different input conditions automatically (semi-automatically if you have an unlicensed copy of SIGMA). You can let SIGMA execute the required runs while you do something else. Input specification files are sometimes called "experimental frames" and they are simple to use. You simply create a text file with one line holding the input responses you would have typed into the keyboard for each run in your batch of experiments. We will see how to do this from a spreadsheet in the next Section.

The steps for directly running a batch of experiments using a compiled C SIGMA simulation is schematically shown as figure 11.1. Any data files and the single experiment control file are written using MS Notepad or some other text editor. (Do not use, for example, MS Word to create this file since it will be saved on disk using a proprietary format and not in plain text.) These files are read by the simulation executable that has been compiled from Sigma-generated C code as discussed earlier. The executable is run inside the MS

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Command (DOS) prompt. The output files from the simulation runs are typically analyzed using a statistical package or spreadsheet, the default output format being for Excel.

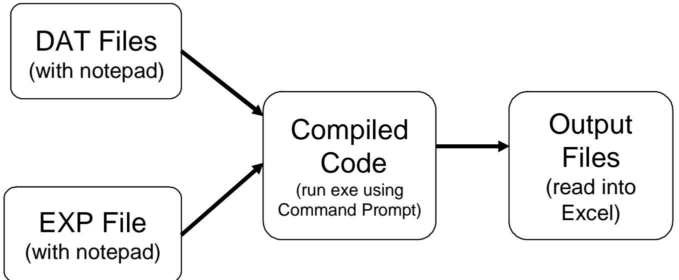
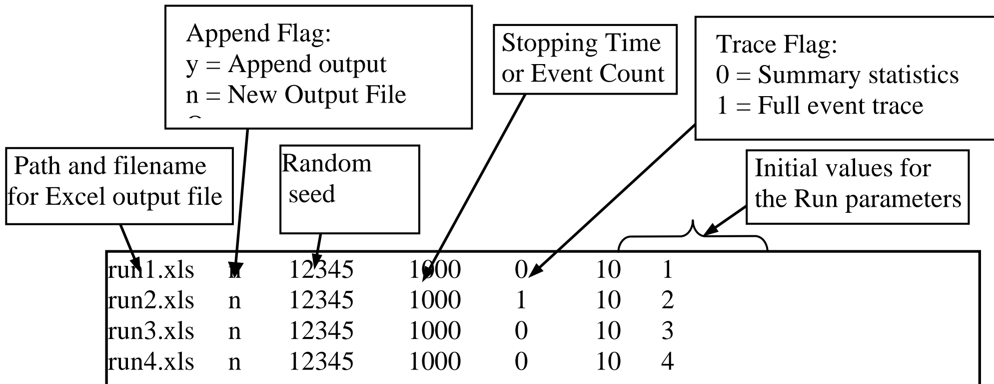


figure 11.1: No interface: the user is involved in every step, using <u>different</u> software

A batch experiment file can be used run your model as many times as desired, producing a separate output file for each run, perhaps with different inputs. Each line in this experiment file specifies the input for a single run of the simulation. It contains the name of the file where you want the output, the required random seed, stopping time, and values for any particular run parameters you require. An experiment file should be saved with the <u>same name as your executable</u> file, but with the `*.exe` extension replaced by `*.exp`. Again, you need to use a plain text editor like Notepad to write this file. An experiment file for a carwash model with the initial queue length and number of servers as parameters to the `RUN` vertex would look something like the following:



CARBATCH.EXP: This file will run the model four times with the output files, random seeds, stopping times, and trace flags as specified. The initial queue length (`QUEUE`) and number of servers (`SERVERS`) for each run are in the last two columns. The second run gives a full event trace, the others only summary output.

REMINDER: Make sure that your *.exp file has the <u>same filename</u> as your *.exe file and that both are in the <u>same folder</u> as all of the input data (*.dat) files read by DISK calls in your SIGMA model. All input files must be created using a plain text editor like Microsoft Notepad.

In the next section we will see how to build a user interface for you simulation in Microsoft Excel, hiding the messy details behind a spreadsheet. If you are using a spreadsheet interface for running your model, you should also make sure that it is also in the same folder with your other files.

Now, suppose we want to make four runs, a full factorial experimental design, consisting of replications at two values for SERVERS of 1 and 2 for `QUEUE` of 5 and 10. The duration of all runs will be 1000 minutes. We will use

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

common random number seeds for the runs made with each of the four combinations of initial conditions. (See Chapter 9 on variance reduction techniques for a discussion of commonly seeded replicates.)

Rather than repeat the dialogue in Section 11.2 four times, we can create an experiment file with the same name as our simulation and the extension .exp instead of .exe. First, we made a copy of the file, CARWASH.EXE and called it CARBATCH.EXE. Next, using Windows Notepad, we created an ASCII text file named CARBATCH.EXP with the following four lines (one for each run in our experiment).

```
run1.xls n  12345 1000  1     5     1
run2.xls n  12345 1000  1     10    1
run3.xls n  12345 1000  1     5     2
run4.xls n  12345 1000  1     10    2
```

The first field is the output file (SIGMA output by default is in Excel spreadsheet format). The next field is n if the output file is to be overwritten and y if it is appended. The next field is the random number seed. The next file is the run duration. The next field is a flag telling if you want a full event trace (1) or just summary statistics (0). The last two fields are the initial values for SERVERS and QUEUE for our four-run factorial experiment.

Clicking on CARWASH.EXE will now run the factorial experiment automatically. (Again: If you do not have a professional license, you will see an advisory.)

11.4 Running a SIGMA simulation from a Spread Sheet

The direct approach to running experiments just explained given in Section 11.3 is cumbersome and not satisfactory if the simulation is to be run by people who would like to use your simulator, but are not really interested in the details. A professional, commercial-grade, simulator needs to have an easy-to-use interface. Such an interface is relatively simple to create using Excel.

What you will produce is an Excel spreadsheet with the distinction of having a "run" button that runs a simulation in the background using information from the spreadsheet and producing results that are read back into the spreadsheet. The schematic for such an interface is given in Figure 11.2.

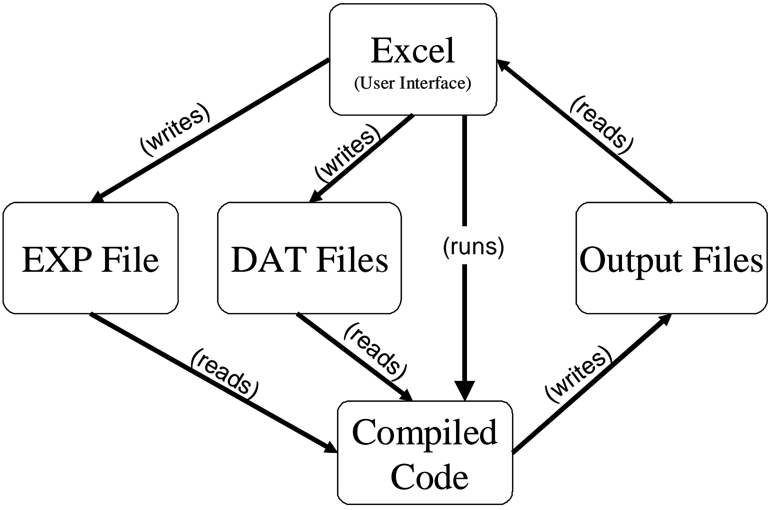


figure 11.2 Excel interface: the user sees only a dynamic spreadsheet

Here the user only sees their Excel spreadsheet and not the messy details of the simulation experiments, which are hidden behind the spreadsheet.

You can think of a simulation spreadsheet interface as like any other spreadsheet with the significant difference that it is *dynamic*. That is, the "run" button executes one or more simulators behind the scenes using information from the spreadsheet as input. You then update the spreadsheet by displaying the results of the simulation experiments. Comparing Figures 11.1 and 11.2 shows the obvious advantages of creating an Excel user interface to run your simulation experiments.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

11.4.1 A Simple Spreadsheet User Interface.

The spreadsheet interface for your simulation can be as simple or elegant as you desire; however, it must perform three essential functions: (1) writing the data files from the spreadsheet for the input to your simulation, (2) running the simulation code using these input files, and (3) reading the simulation output files back into Excel for analysis. Our first spreadsheet interface will perform these three tasks for our carwash simulator, MySigmaSimulation.exe, compiled from the C code, `MySigmaSimulation.c` translated by SIGMA from the model, `carwash.mod`. All three tasks will be done by simply clicking on a command button called "Simulate".

In the XLDEMO1 subfolder, run the spreadsheet MySigmaSimulation.xls to get acquainted , It runs batch experiments for the CARWASH model.

In section 11.4.2 we will demonstrate how to use "forms" in Visual Basic for Applications (VBA) to create a more elegant commercial-quality Excel user interface.

Pressing F1 brings up the help file in Excel if you need terminology defined

The spreadsheet interface and simulator for this example are included with SIGMA in the sub folder XLDEMO1. This spreadsheet will run a batch of experiments as described at the end of Section 11.3 and plot the output. Your own experimental file and simulator will be used. If your SIGMA model uses the DISK function for input, you will need to create one or more data files that are created exactly like the experiment file.

This spreadsheet interface will be called `MySigmaSimulator.xls`, and is also included in the XLDEMO1 folder. This simple spreadsheet has an experimental file, `MySigmaSimulation.exp` (similar to `carbatch.exp` in Section 11.3 with the number of SERVERS at the carwash varying) with a "Simulate" button for writing the input, running the simulation, and reading the output. The minimal interface spreadsheet template is a worksheet called MySigmaSimulation.xls and looks like Figure 11.3. Click on `MySigmaSimulation.xls` to launch this workbook. If you are asked whether to enable the Excel macros, select "Enable Macros"

For security, make sure the Excel interface is directly from a clean SIGMA folder that has not been modified maliciously protected by a virus checker.

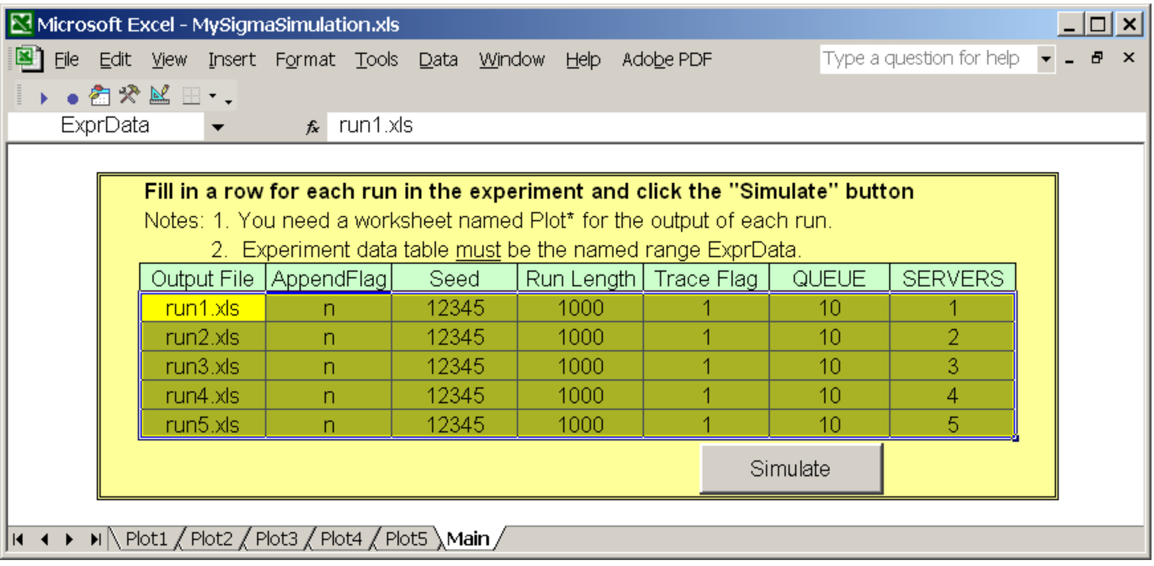
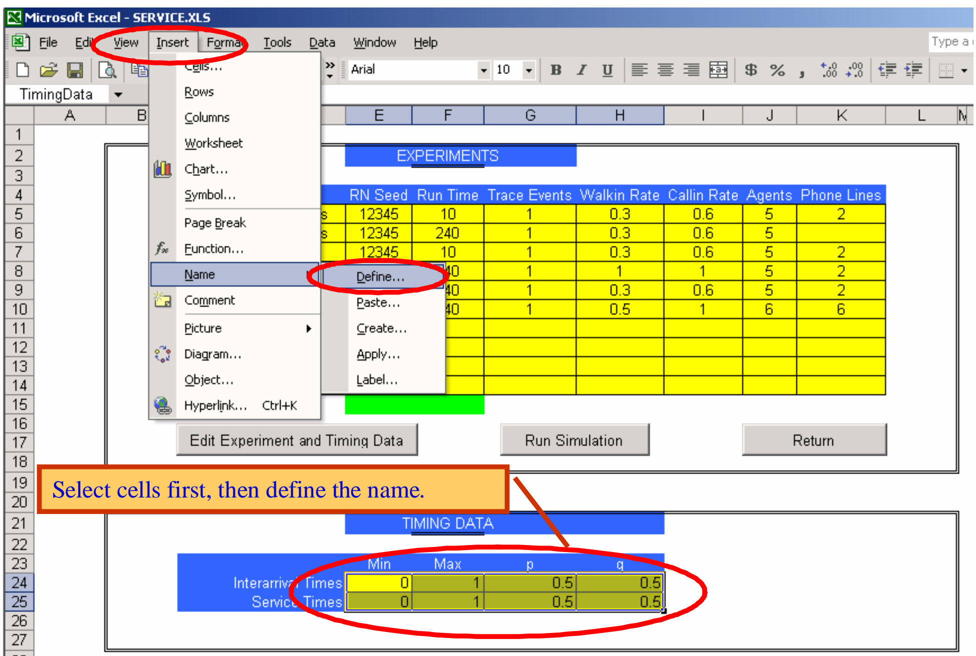


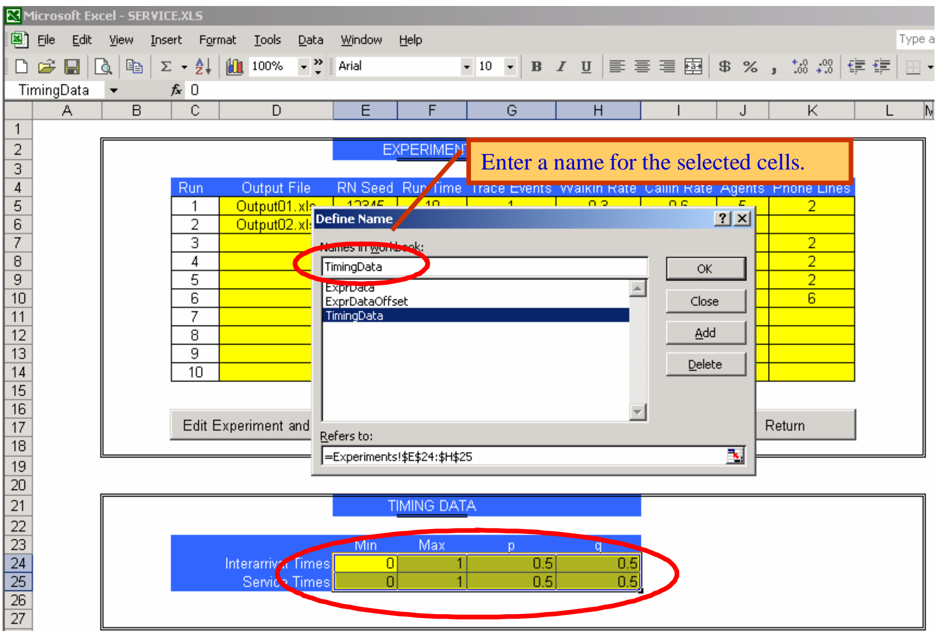
Figure 11.3 A simple dynamic spreadsheet experimental interface

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The entire table containing the experimental file for this Carwash simulator is named, *ExprData*. In figure 11.3, this table is highlighted and its name appears in the name box.

Working with "named cells" in Excel is very important for this example. To name a range of cells, simply do the following: highlight the cells first, then select "Insert → Name → Define," and enter the name of your choice. After you define a name, you can use it anywhere inside the workbook.





You can find out more details by pressing F1 for help and typing in "name cells" into the Answer Wizard.

Before proceeding with adding the "Simulate" button, a little background on Microsoft Office controls and Visual Basic for Applications (VBA) code modules is in order. Controls are objects that have properties, such as their appearance to the user, and methods, which are actions that are taken when the user does something with a control such as

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

clicking on a button. Controls are created using the control toolbox in Excel and includes the usual MS Windows objects (buttons, text boxes, etc.). This first spread sheet interface uses only command buttons to control the basic tasks of running the simulation experiments. Command buttons are created by activating the Tools→Customize→Control Toolbox, then selecting the Button control, and finally clicking on your spreadsheet where you want the command button to appear (try this).

You label command buttons by putting your spreadsheet in "design mode" and right clicking on the button and selecting CommandButton Object→edit. The command button "label" (what the user sees) and the "name" (how Excel recognizes the control) can be different as there are in this example.

Design Mode: When editing controls such as command buttons, it is important to have your spreadsheet in "design" mode as opposed to the usual "operating" mode. You put your spreadsheet into design mode by clicking on the icon that looks like the one here. Clicking it again puts your spreadsheet back into operational mode.

Modules are VBA code that perform general functions such as reading and writing files or running your simulation code. These code modules, called subroutines or "subs" for short, are in the modules section of the spreadsheet

To use this example as a template: The VBA code modules in MySigmaSimulation.xls should be copied from module1 folder in MySigmaSimulation.xls found in the VBA project explorer (activated by pressing Ctrl-R) and pasted into your spreadsheet code module. The code Module is created by using the insert→module menu command in your VBA editor. We will see shortly how to invoke the VBA editor.

The VBA code modules have comments explaining what the are doing. VBA comments are on lines starting with a single quote ('). The details of the VBA code are explained in the next section; however, you can skip the details and merely use these functions by copying them into your spreadsheet. Pressing the "Simulate" button performs the three essential steps in running a simulation.

Step 1. Write the input files for the simulation runs from data on the spreadsheet

With your spreadsheet in design mode, double click on the "Simulate" button and the VBA editor will appear. Look for the code that is executed when a user clicks this button.

```
WriteDataFile "Main", "ExprData", "MySigmaSimulation.exp"
```

Clicking the button when the spreadsheet is in normal operational model runs the experiment. The generic subroutine `WriteDataFile` has three arguments: (1) the worksheet where the input data is, (2) the name of range for the data table on that worksheet, and (3) the output file where this data is to be written.

Step 2. Run the simulation executable from the spreadsheet

The next step is the line of code,

```
RunExe "MySigmaSimulation.exe"
```

The generic code module, `RunExe` runs the program called MySigmaSimulation.exe by calling the Visual Basic function "`shell MySigmaSimlation. vbNormalFocus`". The spreadsheet waits for the run to finish. Then it loads the output into the spreadsheet and creates plots. It often takes longer for output to load into Excel then it did to generate it with a fast compiled SIGMA simulation executable.

In the next section we will look at code for the `WriteDataFile` and `RunExe` code modules in detail. For now it is important that you remember the following about this example.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

IMPORTANT

This spreadsheet runs the CARWASH simulator completely hidden from the user. The danger in doing this is that if something goes wrong (say, an input error) the user will never know it – the program just won't terminate.

Hiding execution is done by removing the following two lines in the translated C code (two lines after // experiments terminated)

.

```
scanf("%1s",&keytoclose);
fflush(stdin);
```

before the C code was compiled. The shell command was run with the vbHide option instead of vbNormalFocus option. Unless you have complete confidence with your simulator, use the vbNormalFocus option with the shell command, never vbHide.

Step 3. Examine the simulation output

After the run, the plot from the first run is shown using the code,

```
Worksheets("Plot1").Activate
```

The results of all of the experiments (here limited to four) can be viewed by clicking the tabs for the plot worksheets or using the command buttons.

You can use this spreadsheet as a template for your own dynamic spreadsheet simulator once you have used SIGMA to write the C code for your model. Your C code needs to be complied using the two-step compiler shortcut in Section 11.2 and tested with your own experiment file as discussed at the end of Section 11.3. The only changes necessary are for you to name the table for your experiment file, *ExprData* and make sure you have at least as many blank worksheets named "Plot*" for each run in your experiment for the simulation output. If your model needs other input data files (say, when your SIGMA model uses the DISK function for additional input, these data files can be written with additional calls to the `WriteDataFile` module with the appropriate file names and ranges before `RunExe` is executed.

11.4.2 A More Elegant Spreadsheet Interface using VBA

A simple customer service center is used to demonstrate how to build an Excel spreadsheet interface for running experiments with a simulation using Visual Basic for Applications (VBA) forms. This interface has all the essential features needed for creating a commercial special-purpose dynamic spreadsheet. It can be used as a template for building other spreadsheet interfaces for SIGMA-generated simulation executables by copying and pasting code from this example into spreadsheet code modules. The source codes for this example are in the XLDEMO2 subfolder that is included with SIGMA.

In this section, we first develop a simple SIGMA model of a customer service center. We will then go through a tutorial on how to use a dynamic spreadsheet interface for running experiments with this service center simulator. We will then show how to build the spreadsheet interface using VBA. Visual Basic is an extensive and ever expanding collection of objects for Microsoft Office programs. One does not actually "learn" VBA, like one might learn the language C. The best most users do is to "get acquainted" with the fundamentals of this language, and searche the VBA help file or the web for objects that do what they. Appendix C gives fundamental details of VBA with Excel. This section introduces VBA through the following example.

An Example: a Service Center Simulation

The SIGMA model used for this section is SERVICE.MOD. This model simulates a customer service center that has two types of customers, walk-ins and call-ins. The simulation model is designed to examine the effects on the system's performance of having different numbers of agents and phone lines. Beta probability distributions are used here to model the customer interarrival and service times; Beta distributions were chosen for their flexibility, but are easily changed (see Section 9.5). The agents are cross-trained to handle both walk-in and call-in customers, which are assumed here to take identically distributed service times (again, easily changed). Calls that come in when all agents are busy are automatically put on hold if any open phone lines are available; otherwise, they get a busy signal. Walk-in customers arriving when all agents are busy must wait in a single queue. Whenever an agent finishes service, if there are calls on hold, they will be answered before any further waiting walk-in customers are serviced. Giving call-in customers priority over walk-in

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

customers is a common, although very annoying, customer service desk behavior. The details of the simulation model follow:

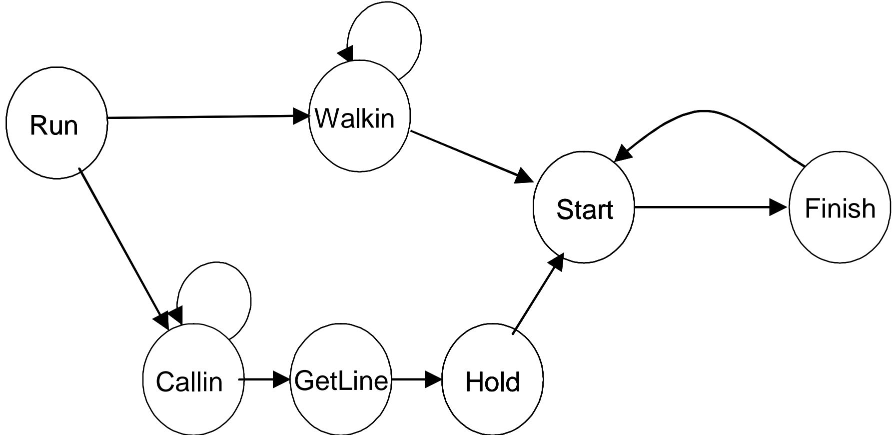
STATE VARIABLE DEFINITIONS

`AGENTS`	Number of idle agents
`LINES`	Number of open phone lines
`TYPE`	Customer type (0=Walkin, 1=Callin)
`QUEUE[TYPE]`	Number of customers waiting (TYPE=0=in line, 1=onhold)
`CIAR`	Call-in Arrival Rate
`WIAR`	Walk-in Arrival Rate
`C,D,P and Q`	Range and shape parameters for Beta distributions

EVENT DEFINITIONS

`RUN`	Initialize each run in an experiment
`Callin`	A call comes into the center
GetLine	A caller seizes an available phone line
`Walkin`	A walk-in customer arrives
`Start`	An agent starts service
`Finish`	Service finishes
`Hold`	A call is placed on hold

The event relationship graph for the model is as follows:



A verbal description of the model dynamics follows, with at most one sentence for each edge in the graph and the event names underlined.

Starting a simulation Run will schedule the first Walkin and first Callin Customer arrivals. Each type of arrival schedules subsequent arrivals of the same type with the appropriate random interarrival times. If a Walkin arrival finds an idle agent, service will Start without delay. If a Callin customer finds an idle line, it immediately will GetLine, and then be put on hold after a 3 second delay. If an agent is available when a call-in customer goes on hold, service Starts without further delay. Once service Starts it will Finish after a service time. When a service Finishes, if there are calls waiting on hold, they will Start being served, if there are no calls on hold and walk-in customers are waiting, a walk-in customer service will Start.

This simulation, SERVICE.MOD, is in the subfolder, XLDEMO2. You can read this model into SIGMA to examine it in greater detail and run it a few times. If you do this, note that the number of available LINES (if any) is decremented in the GetLine event and not in the Hold event. If LINES were instead decremented in the Hold event, a subtle error would occur when two call-in customers arrive less than 3 seconds apart and only one of the LINES is available. Both customers would then each *schedule* a Hold event after 3 second delays. The execution of these two Hold events would cause the number of idle LINES to be decreased twice, becoming negative. The rule of thumb illustrated here is to decrement

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

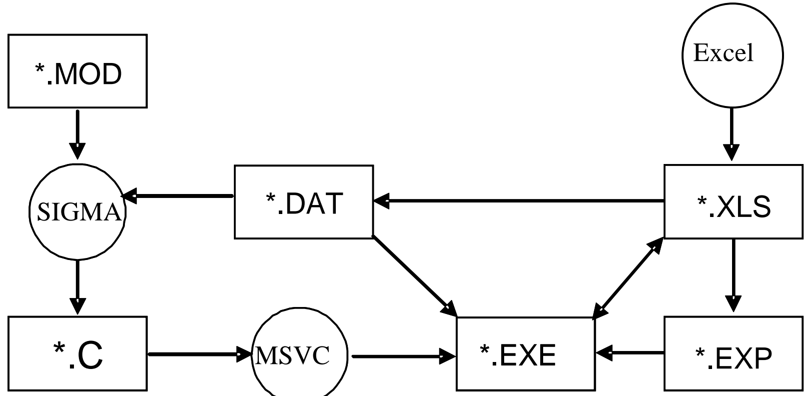
available resource counts immediately when their availability is tested, after a zero-delay top priority edge. (In Chapter 6 it shows how a delay of * will cause immediate event execution). A typical output plot from this simulation gives the number of calls on hold, customers in line, available agents and available phone lines.

This simulation has been integrated with a spreadsheet to make it easy to run multiple experiments. Here we are only running one model from the spreadsheet. In actual applications, many different complex simulations of different scenarios can be run from the same spreadsheet interface. A similar interface for a SIGMA model has been used to design a call-in center for a major bank with hundreds of agents on duty in different areas of the country with different sets of skills and training levels processing tens of thousands of customers a day. That simulation has two run modes (actually two different SIGMA models): one to optimize the number of agents in different skill classes and the other to test and verify a particular operational and agent scheduling scenario.

11.4.3 Tutorial for using the Service Center Spreadsheet Simulator

The spreadsheet interface used in the previous section was created using Visual Basic for Applications (VBA) code. VBA is distributed with Excel. The VBA codes included in the XLDEMO2 subdirectory can be used as a template for creating dynamic spreadsheets. Professional spreadsheet interfaces for SIGMA models have been developed that are much more sophisticated than this one, controlling the execution of several different simulators for different scenarios; however, all have the same basic function of writing data and experiment files and calling the simulation executable compiled from the automatically generated C code using the VBA "shell" command.

The relationships between different pieces of software (circles) and program types (boxes) involved in creating a dynamic spreadsheet are illustrated below

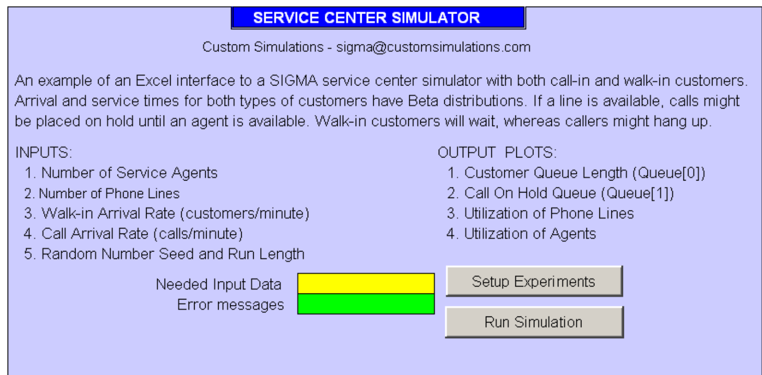


SIGMA runs a model (*.MOD) file and sometimes reads a data (*.DAT) file using the DISK function described in Section 7.2. SIGMA can automatically generate the source code in C (*.C) for the simulation that needs to be compiled by MSVC into a simulation executable (*.EXE) to run. The simulation executable will read the same data files as SIGMA. The simulation executable can also run a batch of experiments defined in an experimental (*.EXP) file. Excel can be used to create a spreadsheet (*.XLS) that writes any data and experiment files before calling the simulation executable. Details for creating such an interface are given after the service center simulator spreadsheet is explained. The Excel interface for this example is composed of three worksheets and two forms

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

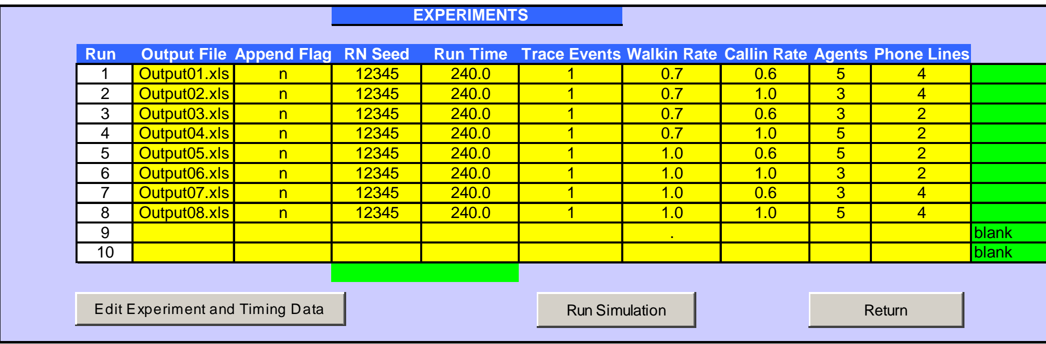
The Three Worksheets in the Simulator

To run this dynamic spreadsheet simulator, open `service.xls` and press the "enable macros" button if asked. There are three worksheets in the interface and two forms. We first introduce the worksheets and introduce the forms later. The first worksheet (referred to as Sheet1) gives a brief introduction to the simulator that looks like the following.



The interface is designed so that the user can clearly see what information they need to provide by coloring the cells yellow where user input data is required. If the input data is missing or incorrect, error messages are given in green colored cells.

There are two buttons on this introductory worksheet. Clicking the "Run Simulation" button will call the second worksheet (Sheet2) from where you can execute the simulation with whatever experimental data is currently on a third worksheet (this worksheet is discussed in detail later). Clicking on the "Setup Experiments" button calls this third worksheet, an "Experiments" worksheet, which is looks as follows.



EXPERIMENTS

Run	Output File	Append Flag	RN Seed	Run Time	Trace Events	Walkin Rate	Callin Rate	Agents	Phone Lines	
1	Output01.xls	n	12345	240.0	1	0.7	0.6	5	4	
2	Output02.xls	n	12345	240.0	1	0.7	1.0	3	4	
3	Output03.xls	n	12345	240.0	1	0.7	0.6	3	2	
4	Output04.xls	n	12345	240.0	1	0.7	1.0	5	2	
5	Output05.xls	n	12345	240.0	1	1.0	0.6	5	2	
6	Output06.xls	n	12345	240.0	1	1.0	1.0	3	2	
7	Output07.xls	n	12345	240.0	1	1.0	0.6	3	4	
8	Output08.xls	n	12345	240.0	1	1.0	1.0	5	4	
9						.				blank
10										blank

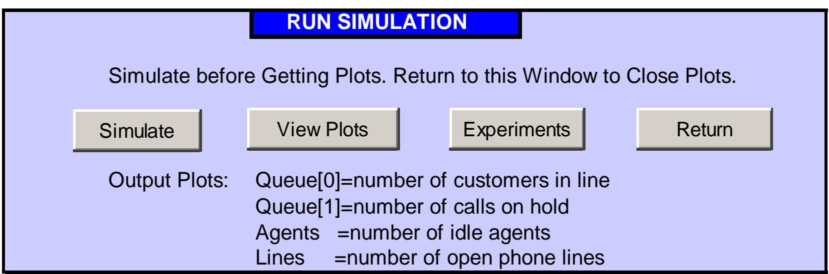
Edit Experiment and Timing Data | Run Simulation | Return

Each row of the table on this worksheet defines an experiment that is to be run in a batch of experiments. If there is any blank space for an experiment, the entire row will be ignored, and an error message will appear in the green error space to the right. The information in this table will be written to the simulation experiment file, `MySigmaSimulation.exp`, before MySigmaSimulation.exe is executed. Experimental files were discussed in Section 11.3. (The reader may or may not recognize that the experimental design being run in this example is called a fractional factorial experiment.) This spreadsheet also has a table of data that will be written to a "data" file called `timing.dat` before `MySigmaSimulation.exe` is run. This data file will be read by a DISK function in `MySigmaSimulation.exe` to determine the shapes and ranges of the probability distributions used for the random customer interarrival and service times.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

TIMING DATA				
	Min	Max	p	q
Interarrival Times	0	1	0.5	0.5
Service Times	0	1	2	3

After the data are entered, clicking the "Run Simulation" button on the introduction worksheet will bring up the Simulation worksheet. The "Run Simulation" worksheet looks like the following.



This worksheet has four buttons:

(1) the "Simulate" button first writes the experiment file, `MySigmaSimulation.exp` and the data file, `timing.dat`, and then runs the simulator, `MySigmaSimulation.exe`. You should do this to see how it works and make sure your system is set up correctly. You will first receive a message reminding that you need to have the some data in the tables in the Experiments and Timing Data worksheet before you run the simulator.;
(2) the View Plots button brings up a form to view and navigate among plots from the latest simulation run;
(3) the Experiments button switches to the Experiment worksheet; and
(4) the Return button goes back to the "Main" introduction worksheet.

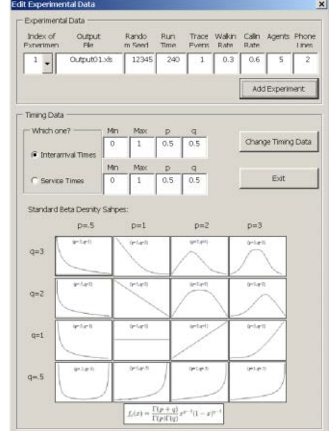
The Two Forms used in the Simulator

This spreadsheet interface has a form that allows the user to enter the experimental and timing data into the experiment and timing worksheets. This form is activated by pressing the "Edit the Experiment and Timing Data" button on the experiment worksheet. On the top of the form is the area for specifying the settings of a particular simulation run indexed from 1 to 10 by a combo box (drop-down box). Clicking the Add Experiment button copies the settings into the corresponding cells of the Experiment worksheet.

The second part of the form includes (1) two options buttons specifying which data distribution (arrivals or services) is to be modifed, (2) text boxes for entering timing data manually, and (3) sixteen Beta-distribution pictures for the user to select (clicking a picture automatically changes the values of above text boxes to reflect the distribution shape selected by the user). Clicking the Change Timing Data button copies the distribution settings into the corresponding cells of the Experiment worksheet.

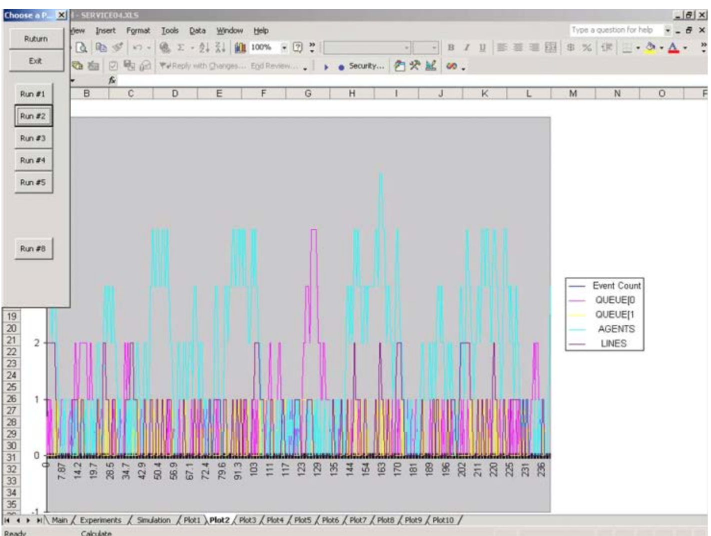
The form looks like the following

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



The two parameters for the distribution shape are chosen by clicking on a picture of the distribution when the appropriate (Interarrival Times or Service Times) radio button is activated. The distribution range (Min and Max) is entered in the text boxes here too. The timing data table on the worksheet is not updated until the “Change Timing Data” button is pressed and the experiment table is not updated until the “Add Experiment” is pressed.

A second form is activated by pressing the “View Plots” button. This is a simple form that displays the output plots from the experiment. The form and resulting output plots look like the following.

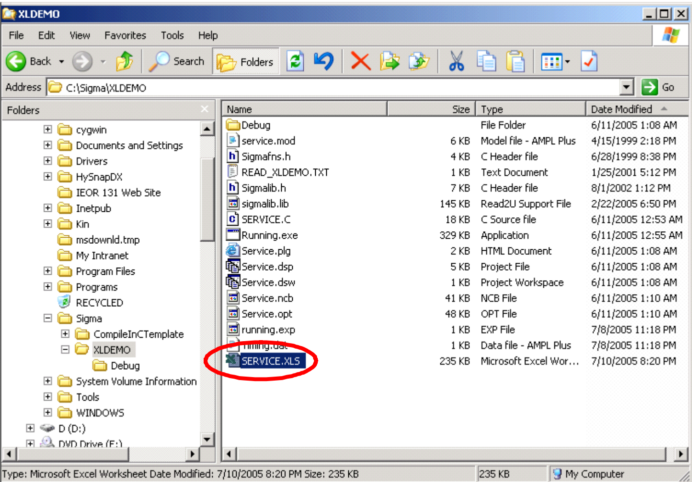


In the next section, we show how this spreadsheet interface was created using Visual Basic for Applications (VBA). VBA fundamentals are reviewed in Appendix C.

11.4.4 Creating the Interface

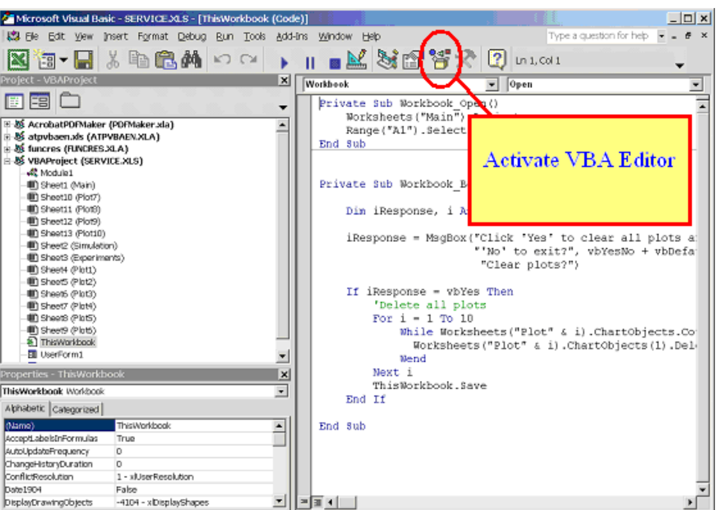
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Open the XL-interface by double clicking on SERVICE.XLS



If you are asked whether to enable the XL-interface's macros, select "Enable Macros" (for security, make sure the XL-interface has not been modified by others who might have added malicious macros).

To see the details on the interface, you should open the Excel tools/options menu dialog and check the formula bar, gridlines, scroll bars, etc. You can take an initial look at this code by opening the VBA editor. In Excel: press the VB Editor button Alternatively: hold the Alt key and press F11 (Alt-F11) to open VBA; hold Ctrl key and press R (Ctrl-R) to open project explorer; finally press F4 to view the Properties window. You should see the VBA editor looking somewhat like the following depending on your options.



Perhaps the best way to get acquainted with VBA is to watch code execute in the VBA editor. You can do this by setting "breakpoints" in VBA to halt execution of the code at a particular point. To set a breakpoint in the VBA editor: double click on an object in the VBA project (say UserForm1) which will place the cursor on some code related to a procedure for that object. Press F9 to set a breakpoint.

Now, if you go through the above tutorial another time, the code will stop when you hit each breakpoint. Press F8 to step through the code or F5 to continue to the next breakpoint. Holding the mouse for a short time over a variable name in the VBA editor will cause its current value to be displayed. The VBA editor makes it easy to understand the Visual Basic code, so you can copy and modify it for your own spreadsheet interface.

The steps taken in creating this dynamic spreadsheet simulator were the following.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Step 1. Create the simulator from the SIGMA model, Service.mod.

Step 2. Layout the worksheets and create navigation buttons

Step 3. Create a form to fill in experimental data.

Step 4. Have the spreadsheet write the simulation input files

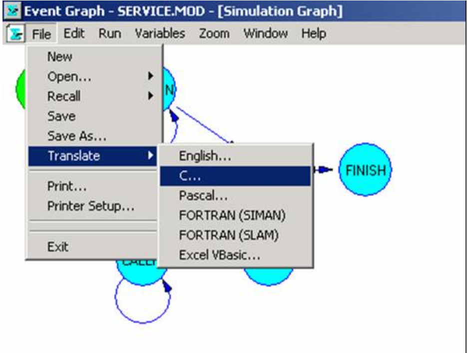
Step 5. Enable the Spreadsheet to execute the Simulator

Step 6. Create a form to display the simulation results after the runs end.

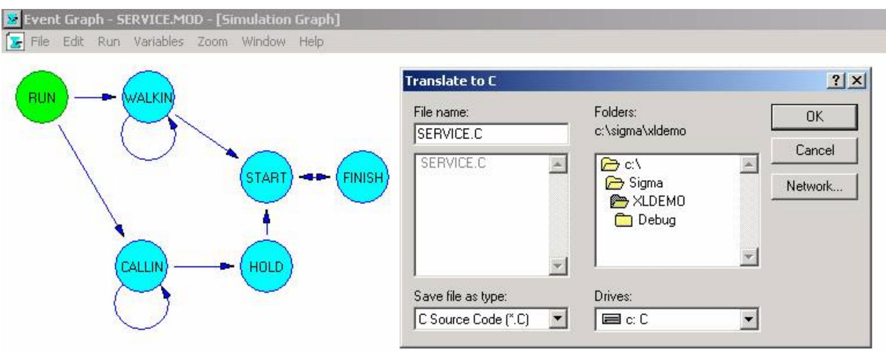
Step 7. Attend to miscellaneous details for a professional-looking simulator including: setting the default starting worksheet to the Main worksheet; changing the XL default drive and directory; and clearing plots before exiting to save space.

Step 1. Create the simulator from the SIGMA model, Service.mod.

After starting SIGMA, open the model `service.mod` included in the `C:\Sigma\XLDEMO2` folder. From the Sigma window, select "File → Translate → C..." to translate `Service.mod` to C code and name it `Service.c`.

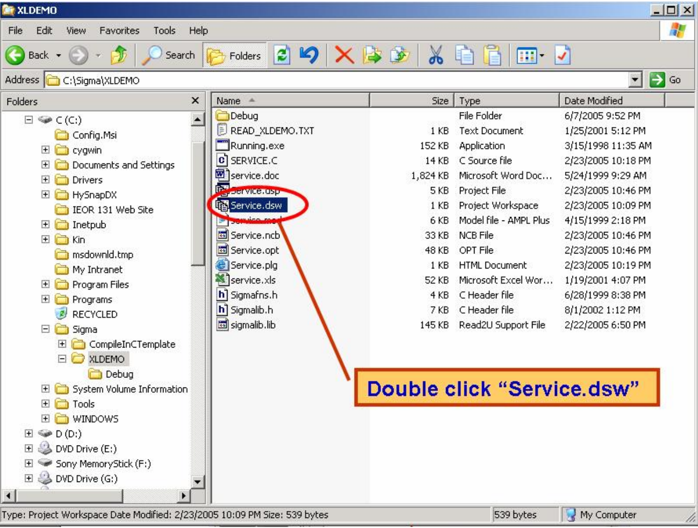


Save the translated `Service.c` to the folder "`C:\Sigma\XLDEMO2`". If your default directory is different, simply save the C file to your directory and keep in mind that whenever the following appears "`C:\Sigma\XLDEMO2`", it should be changed to your default directory.

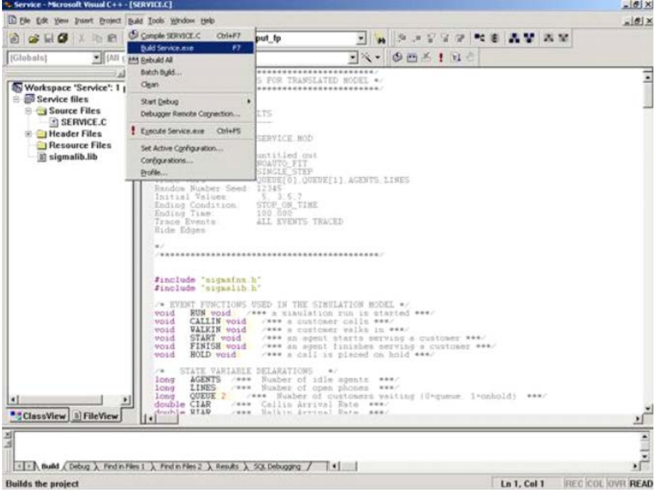


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Go to the Excel demo folder located in your SIGMA folder, by default, it is under "C:\Sigma\XLDEMO2". Double clickint on "MySigmaSimulation.dsw" will open Microsoft Visual C/C++ (VC) version 6 or later. Note: If you double click on MySigmaSimulation.dsw and a VC workspace does not open, this means all the files with a ".dsw" extension in your computer are not associated with the VC program. In this case, start VC directly from the Start menu and open "MySigmaSimulation.dsw" from VC.

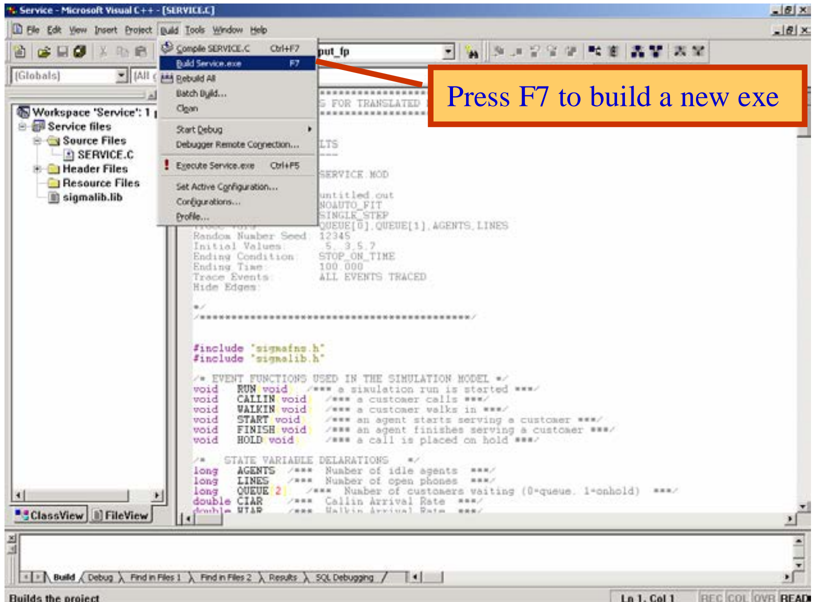


The workspace in VC you opened by clicking MySigmaSimulation.dsw should look as follows.



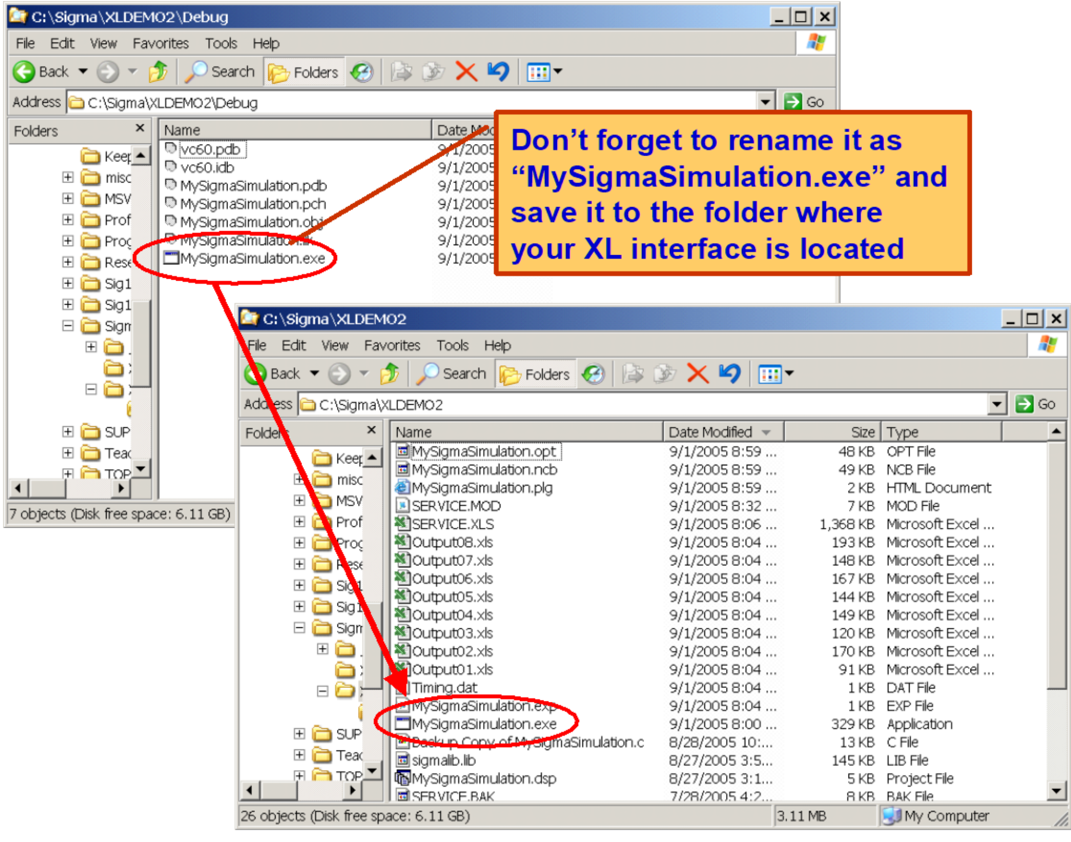
Press F7 to compile and build a new executable, which will be saved automatically as "MySigmaSimulation.exe" under the "C:\Sigma\XLDEMO2\Debug" folder. You will need to copy this new executable from the debug subfolder up to the "C:\Sigma\XLDEMO2" folder. If you choose to use a different folder name, it is a requirement that the interface, service.xls, and the simulator, MySigmaSimulation.exe, be in the *same* folder.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Do not forget to move the newly compiled simulator, `MySigmaSimulation.exe` from the debug subfolder to the folder where the Excel interface, Service.xls, is located.



Step 2. Layout the worksheets and create the navigation buttons

Using the Excel Format→Cells menu command, layout and put borders and colors in the tables and text in the worksheets. It is helpful to use different colored cells to indicate where your user needs to provide data (yellow cells here) and cells that will display error messages (green cells here). You can click on any green cell and, if the tools→options→formula bar box is checked, you will see the formula that displays the error messages (using =IF logical tests). To find out how these formulas work, press F1 to open the Excel VBA help file and search for the key words. For example: the green cell at the bottom of the RN Seed column in the Experiments table has the formula:

```
=IF(MAX(E5:E14)>65000,"Seeds<65000!","")
```

which will display the message "Seeds<65000!" if any seed in this column is larger than a maximum of 65000 (Again, press F1 for details.)

Navigation buttons:

Simple objects called a command buttons are used to navigate among the worksheets in the interface. These are easily added to worksheets. Open the controls tool bar by pressing and press .

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The next place you click on your spreadsheet will create a command button called, by default, “CommandButton1” looking like

CommandButton1

Right clicking on this button and selecting Common Button Object → Edit allows you to change its shape and rename it. Try this and rename the button, “MyButton” You can also right click and select “view code” to see the actions that take place when this button is clicked. It should look like:

```
Private Sub MyButton_Click()
End Sub
```

In most setups of VBA, key words are displayed in blue.

All VBA subroutines are initiated with the key word “Sub” followed by the name of the function, and they end with the key words “End Sub” (the Private key word indicates that this is your code). The parentheses after Click indicate that it is the name of a section of code. The generic name of the method of code for a button is “Click()” since the only thing you can do with a command button is to click it. Of course, no action is taken yet since none have been specified. Between these lines you can place VBA code describing the actions you want to be executed when this control object named “CommandButton1” is activated by the action “Click”.

For example, the code to activate the Main worksheet and select a cell, (here A1), for the focus looks like the following.

```
Private Sub CommandButton1_Click()
    Worksheets("Main").Activate
    Range("A1").Select
End Sub
```

When this button is clicked the spreadsheet will behave just as if you had tabbed to the Main worksheet, and clicked on cell A1.

Important exercise: Put your spreadsheet in “design mode” by pressing the design tool Now right click on the “Run Simulation” button and select “View Code”. The VBA editor should open and you should see the following code.

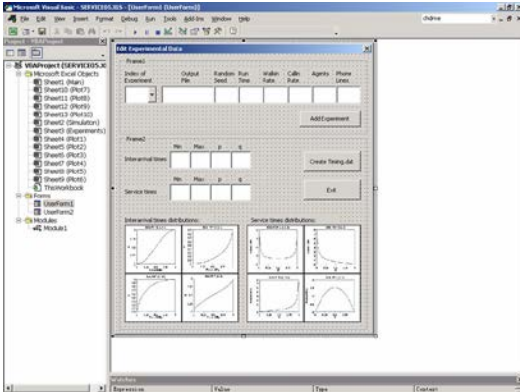
```
Private Sub RunSimulation_Click()
    Sheet2.Activate
    Sheet2.Range("A1").Select
End Sub
```

Make sure both your spreadsheet, `Service.xls`, and the VBA editor are both visible on your screen. You should then click on the first line in the above code to highlight it. Next press F8 repeatedly to step through the code. You should see the spread sheet activate the Run Simulation worksheet and give cell A1 the focus as each of the above lines is executed. Stepping through code by pressing F8 in this manner is how you can learn what the sample VBA codes in this spreadsheet do as well as debug your own. When you are done exploring code in this manner, do not forget to put your spread sheet back into “operation mode” by clicking the design tool once again.

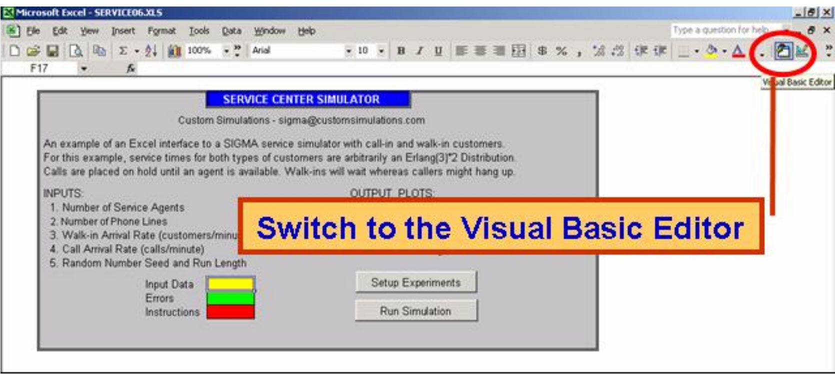
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Step 3. Create a form to fill in experimental data

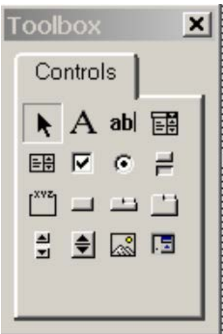
The form for editing the experimental data in this interface looks like the following.



The data in the timing and experiment tables on the second work sheet can be filled in directly; however, it gives the application a more professional look and makes it easier for the user if a form is created to populate these tables with data. The following shows how to create a form for filling in the experimental and timing data into the spreadsheet. From the Excel window, push the "Visual Basic Editor" button to switch to the Visual Basic Editor window. This button looks like



Inside the VBA Editor, open the project browser window. If the project browser is not automatically open, press Ctrl-R. Right click in the browser window and select "Insert → UserForm" to create a user form. Usually, the ToolBox menu will appear when you open the VBA Editor. However, if the control toobox does not appear, push the "ToolBox" button and this will show you the control tool box.

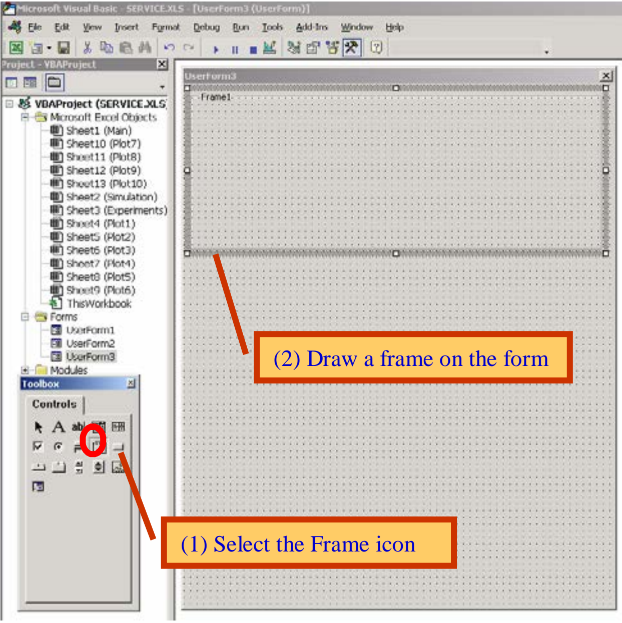


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

These are the familiar controls you see in most windows applications. You can use these to create a form showed below by adding text boxes, command buttons, and image control objects. We will next look at the controls used in this interface.

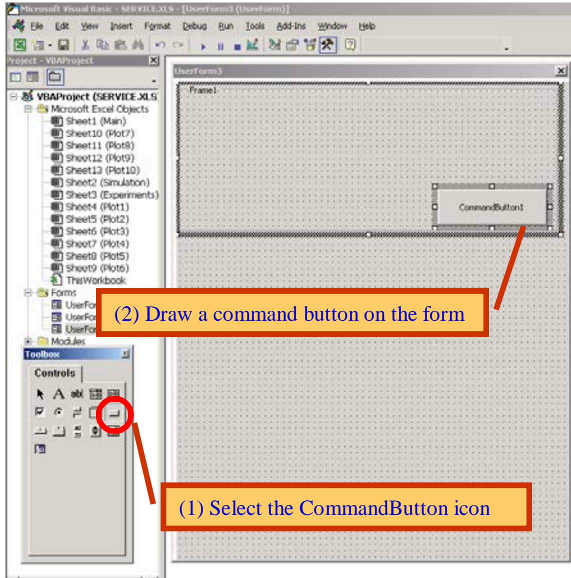
Frames

A Frame control, as a container for grouping controls, is used to visually separate areas of the form. Usually, you don't write much code for Frames; they just make your program look more professional. Don't try to draw a Frame over existing controls, you should draw the Frame first and then draw its constituent controls inside of it. Controls could also be cut and pasted into the Frame. To create a Frame on the form, simply select the Frame icon from the Toolbox menu and draw a frame on the form.



Command Button

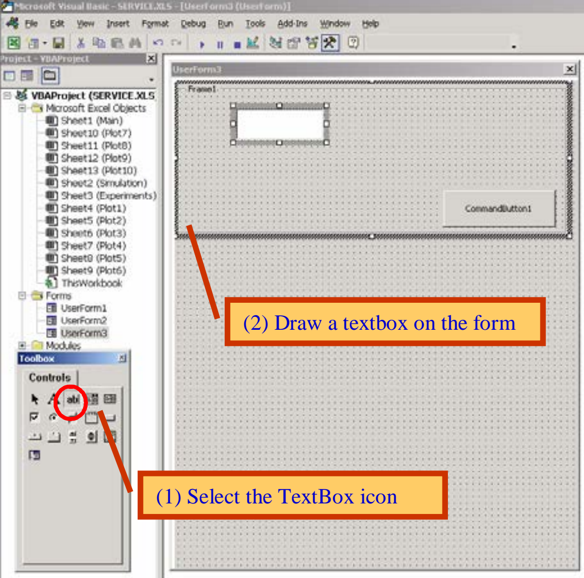
We have already seen command buttons. To create a command button, select the CommandButton icon and draw a button on the form. Double clicking the button brings you into the code associated with the button as discussed this earlier.



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

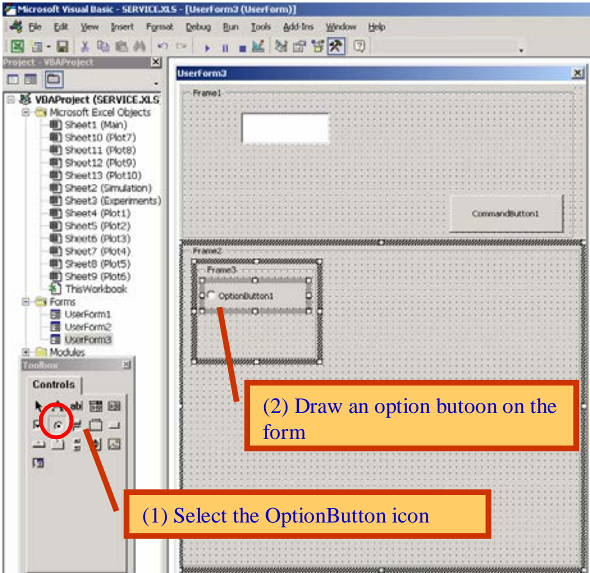
Textbox

To create a textbox, select the TextBox icon from the Toolbox menu and draw textbox on the form.



Option Buttons

To create an option (also called a “radio” or “toggle”) button, select the OptionButton icon from the Toolbox menu and draw an option button on the form.



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

ComboBox

To create a combobox (this type of control is also called a drop-down box or a list box), select the combobox option from the ToolBox and draw a comboBox on the form.

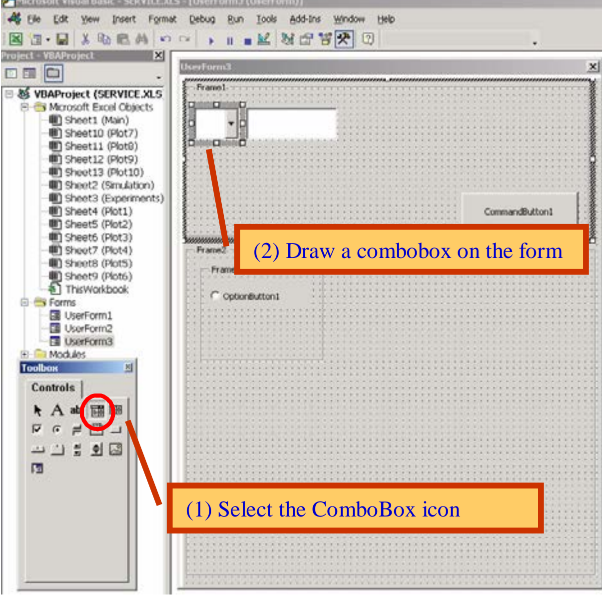
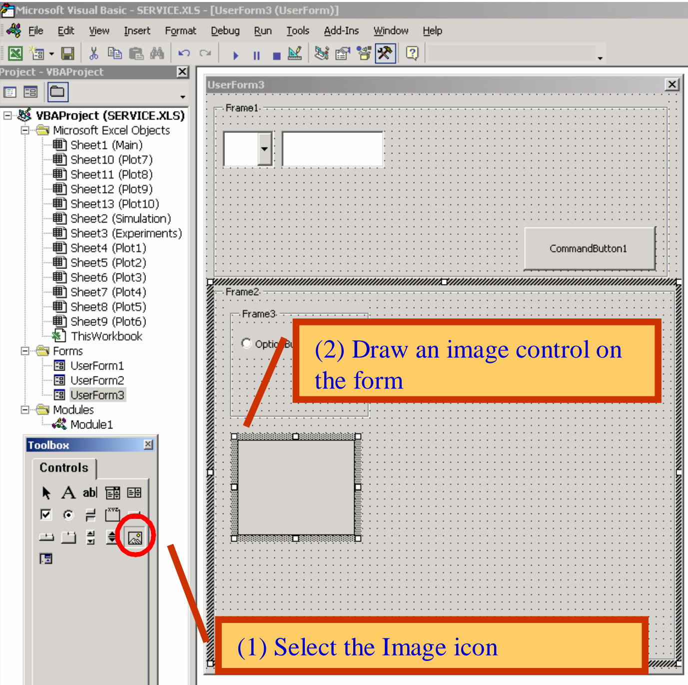


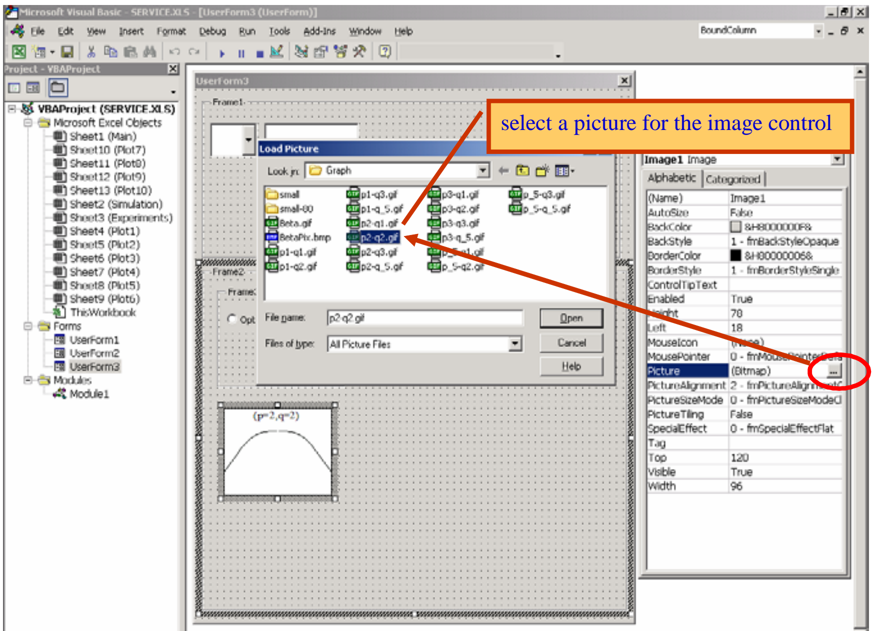
Image Control

To create an image control, select the Image icon from the Toolbox menu and draw an image control on the form.

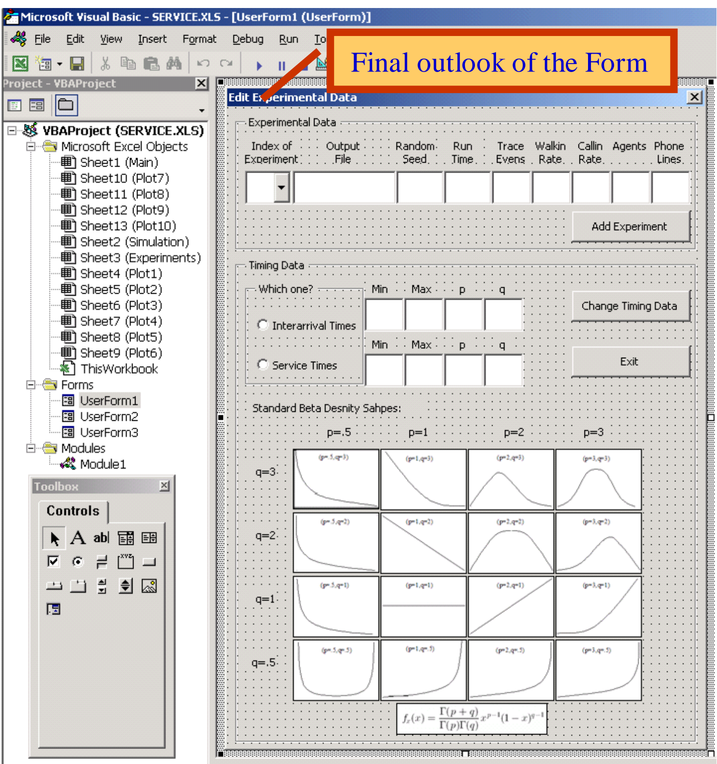


You specify a picture (provided by yourself or from a file) for an image control object from the properties menu of the object. The properties menu of an object is opened by right clicking the object and select "Properties."

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



The final form for our example looks like the following.



Initializing the Controls in the Form

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

A form in VBA is a collection of controls that function like the familiar dialog in Windows. The initial values of items in controls in a form are all set every time the form is activated. Selections and/or changes in the values are entered into the controls on the form. These new values for the controls are assigned to objects or variables when the form is exited (unless a cancel button is provided, in which case, the newly entered control values are ignored). After designing the form with its various controls, the next step is to fill in the initial values for all the controls on the form. This is done with codes into the *UserForm Initialize* function, which is a built-in function for each form.

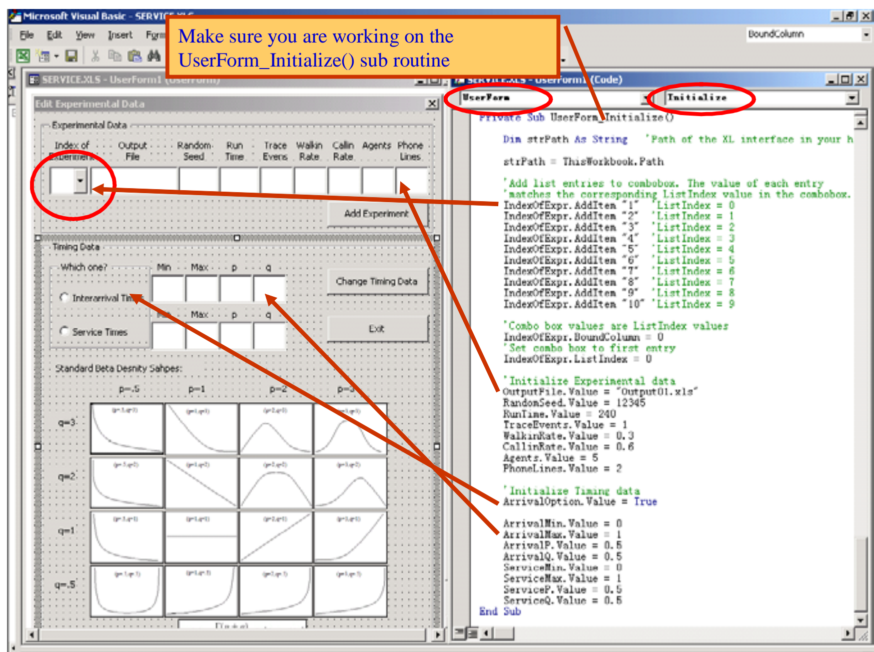
Double clicking on a blank spot on the form *where there are no controls* will open a window containing all the codes associated with the form. First, make sure that your are working on the "`UserForm_Initialize()`" subroutine by selecting "UserForm" and "Initialize" from the left-top corner and the right-top corner of the window respectively.

ComboxBox1, ComboBox2,… and TextBox1, TextBox2,…, etc. are the default names of these types of controls. Other names of the corresponding controls are intended to be descriptive. You can change the names of controls if you desire. To view the name of a control, simply right click on it and select "Properties" to view or change the name or other properties. In this example, IndexOfExpr is the name given to the combo box.

For comboboxes, use the code, `ComboBox1.AddItem "something"` to add an item to it where ComboBox1 is the name you have given your combo box. To initialize our combo box, we will use `IndexOfExpr.AddItem "something"` to add an item to it. The "value" of a combo box symbolizes the current selected item. The first column (column 0) contains the list indices. The second column (column 1) is the first element in each item, and so on. Usually, the "value" of combo box is chosen as the list index values; this is done by using the following code.

```
'Add list entries to combobox. The value of each entry
'matches the corresponding ListIndex value in the combobox.
IndexOfExpr.AddItem "1"  'ListIndex = 0
IndexOfExpr.AddItem "2"  'ListIndex = 1
IndexOfExpr.AddItem "3"  'ListIndex = 2…
```

For textboxes, use `TextBox1.Value = "some text"` to initialize theirs values. Information on initializing all controls are readily available in the help file found by pressing `F1` at any time.



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The complete code to initialize this form in our example is as follows. In VBA anything that follows a ' on a line is a comment. Comments will probably will appear in green. Use comments to explain to others, and remind yourself later, what you are trying to do.):

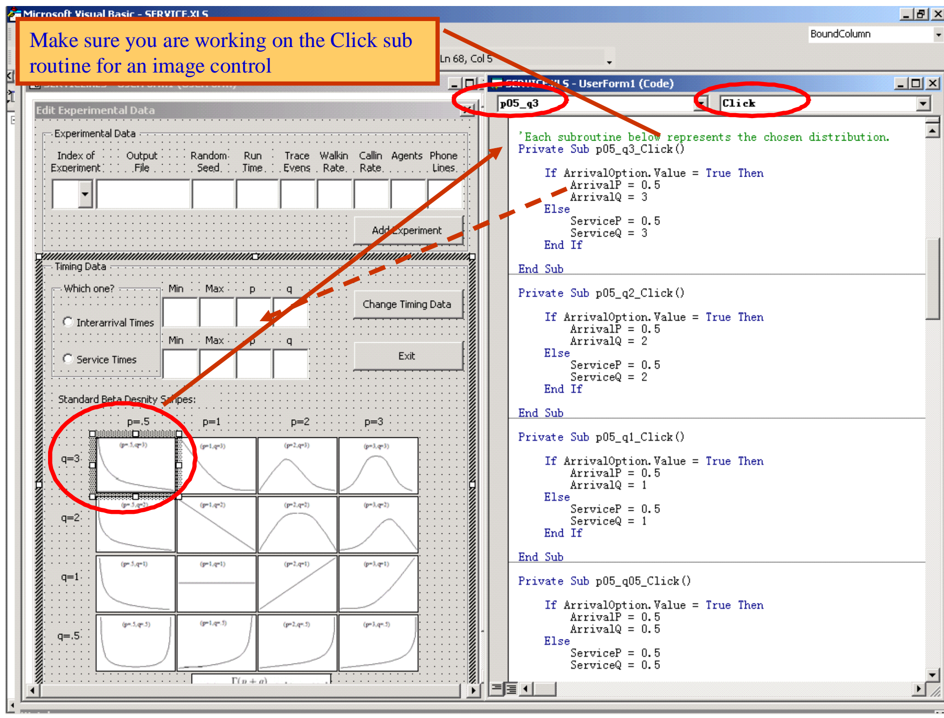
```
Private Sub UserForm_Initialize()
    Dim strPath As String    'Path of the XL interface in your hard-
drive
    strPath = ThisWorkbook.Path

    'Add list entries to combobox. The value of each entry
    'matches the corresponding ListIndex value in the combobox.
    IndexOfExpr.AddItem "1"  'ListIndex = 0
    IndexOfExpr.AddItem "2"  'ListIndex = 1
    IndexOfExpr.AddItem "3"  'ListIndex = 2
    IndexOfExpr.AddItem "4"  'ListIndex = 3
    IndexOfExpr.AddItem "5"  'ListIndex = 4
    IndexOfExpr.AddItem "6"  'ListIndex = 5
    IndexOfExpr.AddItem "7"  'ListIndex = 6
    IndexOfExpr.AddItem "8"  'ListIndex = 7
    IndexOfExpr.AddItem "9"  'ListIndex = 8
    IndexOfExpr.AddItem "10" 'ListIndex = 9

    'Combo box values are ListIndex values
    IndexOfExpr.BoundColumn = 0
    'Set combo box to first entry
    IndexOfExpr.ListIndex = 0

    'Initialize Experimental data
    OutputFile.Value = "Output01.xls"
    RandomSeed.Value = 12345
    RunTime.Value = 240
    TraceEvents.Value = 1
    WalkinRate.Value = 0.3
    CallinRate.Value = 0.6
    Agents.Value = 5
    PhoneLines.Value = 2

    'Initialize Timing data
    ArrivalOption.Value = True

    ArrivalMin.Value = 0
    ArrivalMax.Value = 1
    ArrivalP.Value = 0.5
    ArrivalQ.Value = 0.5
    ServiceMin.Value = 0
    ServiceMax.Value = 1
    ServiceP.Value = 0.5
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
    ServiceQ.Value = 0.5
End Sub
```

Creating actions for each of the Image Controls

A pair of option buttons (sometimes called radio or toggle buttons) is used to select whether a probability distribution shape is to be chosen for service times or for the interarrival times by the different image controls. The shape of a Beta random variable is determined by values of two parameters, p and q (see Chapter 9). This button has the default property "togglebutton.Value" and default method "Click". A togglebutton is activated by selecting it (clicking on it) and sets its Value to the Boolean constant, True, if it is inactive its Value is set to False. The names of the togglebuttons are ArrivalOption and ServiceOption.



For the selected option button we need to create a subroutine for each image object (the pictures of probability distributions) so that when a user clicks on a picture, the shape parameters the chosen distribution curve will be written into the text boxes corresponding to the active option button. To see the code for the image objects, double click on an image control, select the name of that image control (for example "p05_q3") and "Click" respectively from the left-top corner and the right-top corner of the window. This code will check if the radio button for the interarrival time distributions is active (making ArrivalOption.Value equal to "True") else the probability distribution for the service times will be selected). The code for one of these image objects (the one for parameters p=.05 and q=3) looks like the following:

```
'Each subroutine below represents the chosen distribution.
Private Sub p05_q3_Click()
    If ArrivalOption.Value = True Then
        ArrivalP.Value = 0.5
        ArrivalQ.Value = 3
    Else
        ServiceP.Value = 0.5
        ServiceQ.Value = 3
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
    End If
End Sub
```

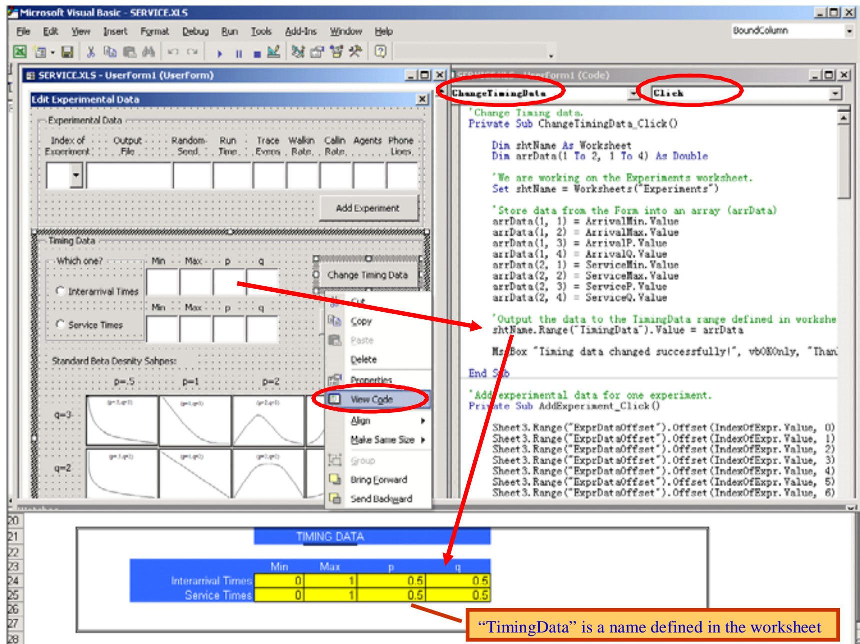
ArrivalP, ArrivalQ, ServiceP, and ServiceQ are names of text boxes for corresponding distribution shape parameters. The default property of a textbox (say ArrivalP) is its value, therefore, `TextBox1 = "text"`, is equivalent to `TextBox1.Value = "text"`. You should highlight the first line of this code and step through it pressing F8 as in the exercise earlier in this section.

The following is the sub routine associated with the "Change Timing.dat" button to write the `Timing.dat` file based on the distributions selected by the user. Double click on this button or right click and select "View Code" to bring up the window to see the codes.

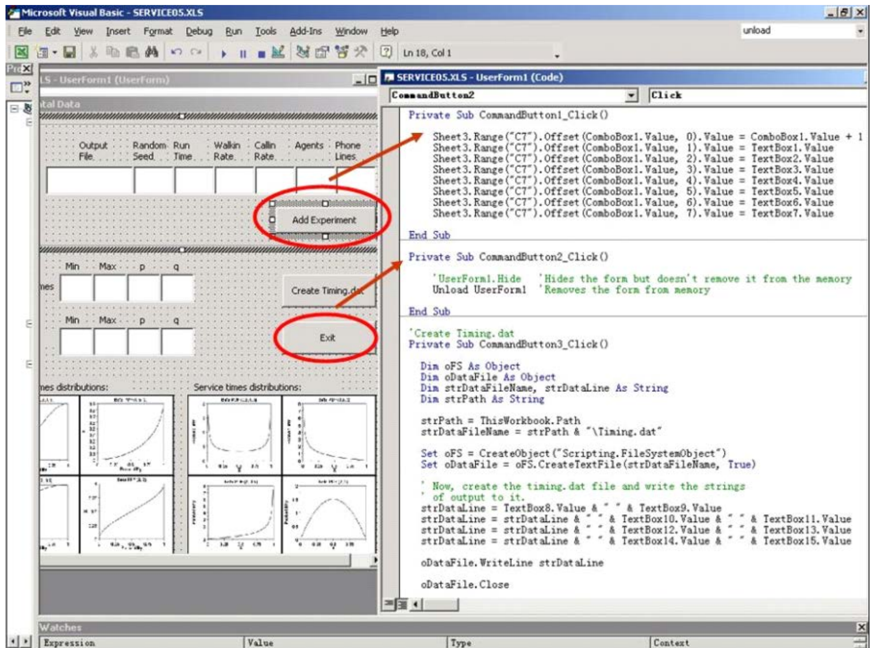
```
'Change Timing data.
Private Sub ChangeTimingData_Click()

    Dim shtName As Worksheet
    Dim arrData(1 To 2, 1 To 4) As Double


    'We are working on the Experiments worksheet.
    Set shtName = Worksheets("Experiments")

    'Store data from the Form into an array (arrData)
    arrData(1, 1) = ArrivalMin.Value
    arrData(1, 2) = ArrivalMax.Value
    arrData(1, 3) = ArrivalP.Value
    arrData(1, 4) = ArrivalQ.Value
    arrData(2, 1) = ServiceMin.Value
    arrData(2, 2) = ServiceMax.Value
    arrData(2, 3) = ServiceP.Value
    arrData(2, 4) = ServiceQ.Value

    'Output the data to the TimingData range defined in worksheet.
    shtName.Range("TimingData").Value = arrData


    MsgBox "Timing data changed successfully!", vbOKOnly, "Thanks!"

End Sub
```

Again, make sure you are working on the right sub routine; select the name of this button for example "`ChangeTimingData`", and again, right click on it and select "Properties" and view the name of a control and "Click" on the left-top corner and the right-top corner of the window. Step through this code pressing F8 with both the VBA editor and the spreadsheet visible as shown in the following figure.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



Once the user hits the "Add Experiment" button, the experimental data will be copied from the form to the worksheet. The "Exit" button will unload the form from the memory. The codes for doing these are shown below. To edit these codes, simply right click on the button and select "View Code." Of course, you also need to select the right subroutine to work on by selecting the name of the button and "Click" respectively from the left-top corner and the right-top corner of the window.



As mentioned in Section 11.4.1, there are three fundamental steps that the spreadsheet must do: (1) Write input files, (2) run the simulation, and (3) get the output. The VBA codes for the subroutines for these steps, `WriteDataFile` and `RunExe`, and, `GetResults`, are all contained in "module1". These codes are fairly complex and can simply be copied to your spreadsheet

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

applications by including module1 in your interface; they are very general utilities. The details for the three steps performed by these subroutines are discussed next and may be skipped by those not interested in fine points.

Step 4. Have the spreadsheet write the simulation input files

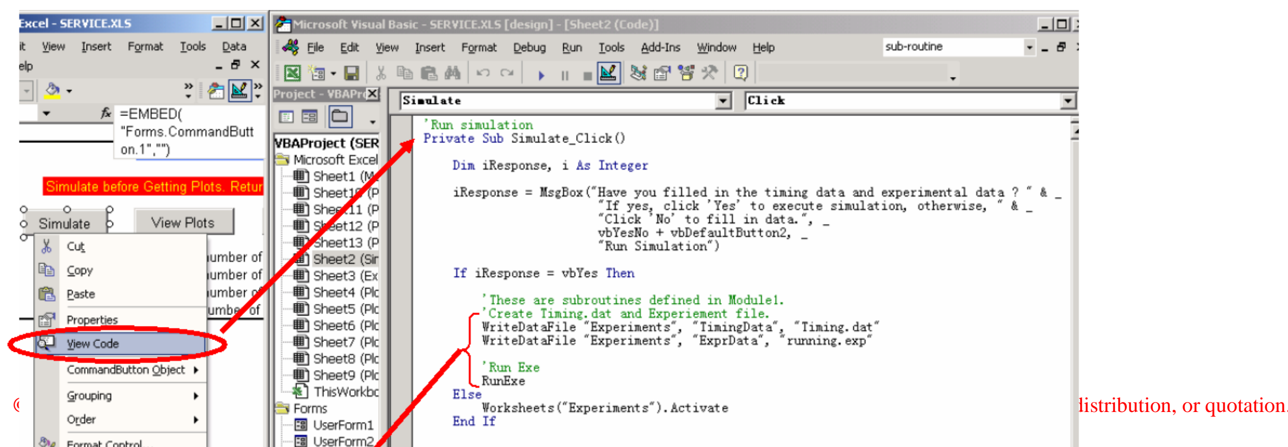
To run the simulator from the Simulation worksheet, click on the “Simulate” button in the Simulation worksheet. Clicking this button calls two subroutines, `WriteDataFile` that write data from the spreadsheet to your hard drive needed for the simulation and `RunExe` that executes the simulation executable. After the simulation has finished running, it then calls a subroutine, `GetResults` to display the output plots..

It is the convention (and we recommend too) that user-defined sub-routines are put into modules. The code for worksheets and forms should contain only what is necessary for performing certain actions, such as displaying an message or show/hide a form. As mentioned earlier, to create a module, right click “Modules” from the project browser window and select “Insert → Module.”

You can see the code for writing the input files and running the simulation by putting your spreadsheet is in “design mode” and right clicking on the “Simulate” button. This will show the following code.

```
'Run simulation
Private Sub Simulate_Click()
    Dim iResponse, i As Integer
    iResponse = MsgBox("Have you filled in the timing data and
experimental data ? " & _
      "If yes, click 'Yes' to execute simulation, otherwise, " & _
       "Click 'No' to fill in data.", _
               vbYesNo + vbDefaultButton2, _
                       "Run Simulation")
    If iResponse = vbYes Then

        'These are subroutines defined in Module1.
        'Create Timing.dat and Experiement file.
        WriteDataFile "Experiments", "TimingData", "Timing.dat"
        WriteDataFile "Experiments", "ExprData", "MySigmaSimulation.exp"

        'Run Exe
        RunExe
    Else
        Worksheets("Experiments").Activate
    End If
End Sub
```

Your screen should look somewhat like the following



listribution, or quotation.

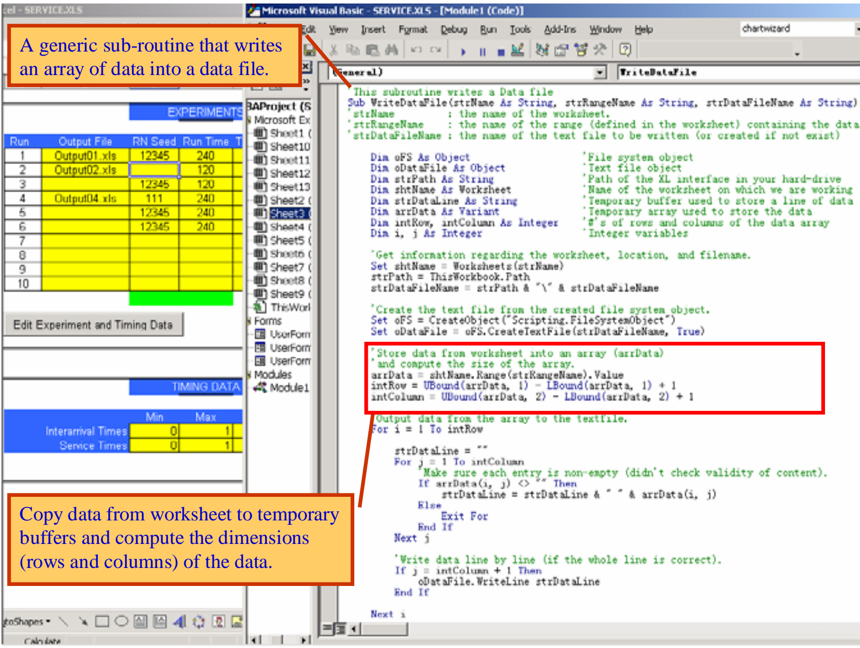
WriteDataFile subroutine: Creating the timing data file and experiment file

The sub-routine `WriteDataFile` shown in the following figure is a generic procedure that writes an table on the worksheet into a data file. It takes three arguments: (1) the name of the worksheet that contains the data table, (2) the name of the array (range) that contains the data table, and (3) the name of the data file to be created and to which the data is to be written. Given appropriate arguments, this sub-routine can be used to create the timing data file (`Timing.dat`) and the experiment file (EXP).

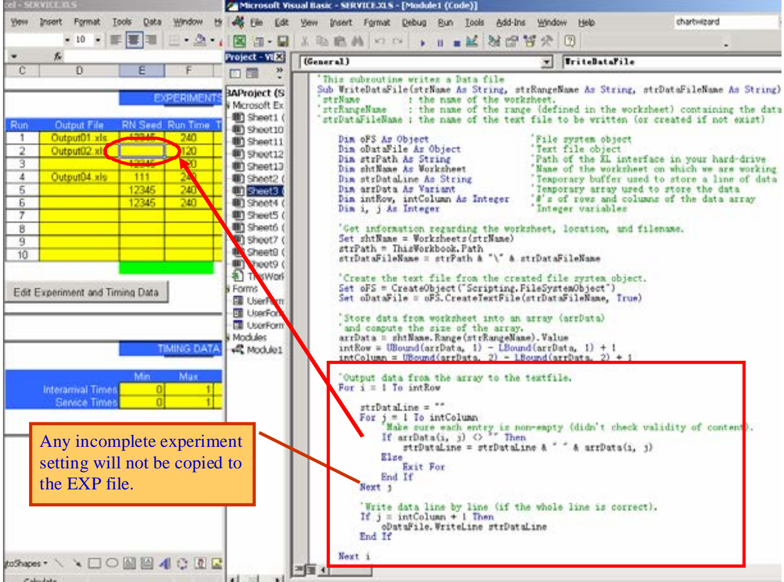
The `WriteDataFile` subroutine that is called by the above subroutine copies data from an Excel worksheet into files on the hard drive so the simulation can read it. The TimingData table is first copied to a file called `timing.dat` on your hard drive, and the data from the ExprData table is then written to a file called `MySigmaSimulation.exp` on your hard drive. (TimingData and ExprData are *named ranges* (discussed earlier) on the *Experiments* worksheet click F1 for help if you don't know what a range is in Excel.)

This sub-routine first creates the text file object. Then, it copies the data from the worksheet (the name of the array is "`strRangeName`" which is defined using a similar procedure given in **Error! Reference source not found.**) to the temporary buffer, called arrData. The dimensions (rows and columns) of this array is computed using built-in API (Microsoft Applications Programming Interface) functions `UBound()` and `LBound()`.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



A double loop is used to output data into the text file: (1) the outer loop goes through all rows of the data range line by line; and (2) the insider loop concatenates the elements (separated by a space) on a row into a string (called "`strDataLine`" in this example) and writes the string into the text file. If any element is missing in this row (in this example, the second element of the second row—the RN Seed—is missing), then this row

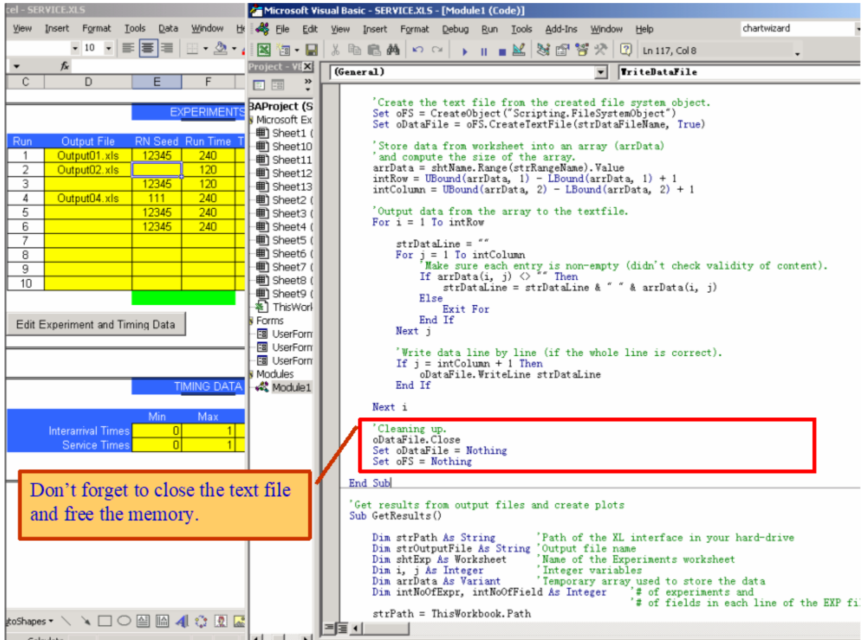


will be skipped (see the "`Exit For`" VBA command inside the insider loop and the "`IF` ..." condition before `oDataFile.WriteLine` command). The actual function that writes a string into a text file is the command "WriteLine" associated with the text file object "`oDataFile`" that we created earlier. (The o in front of these names indicates that these are objects according to the naming convention explained in Appendix C.)

> *Warning:* any incomplete experiment setting (e.g., missing any elements) in the EXP file will cause errors during the simulation and no output is generated in most of the time.

Do not forget to close the text file and free the memory after the file is written.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Step 5. Enable the spreadsheet to execute the simulator

We discussed the `WriteDatafile` subroutine earlier. Now we look at the details in the `RunExe` subroutine. The subroutine `RunExe` is also defined in Module1.

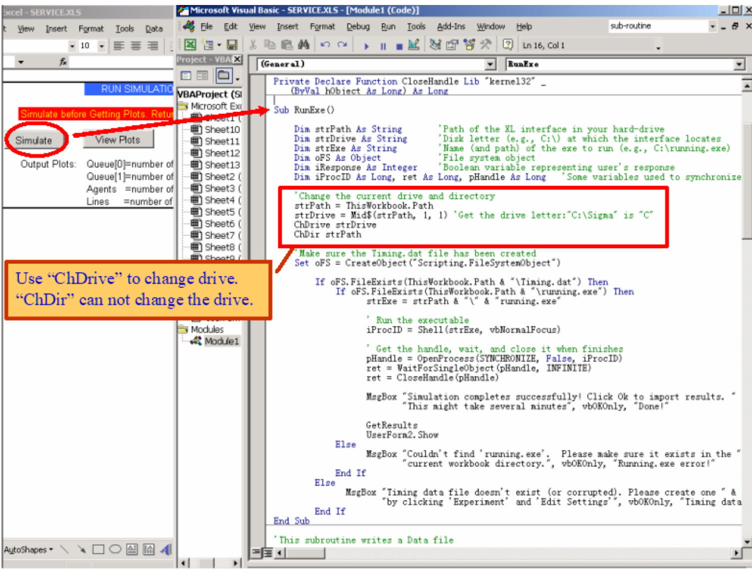
First Make sure the simulator and the input data are in the same folder: Once that the simulation input data is on your hard drive, the simulation is run by calling the subroutine, `RunExe`. However, this will not work if the simulator, `MySigmaSimulation.exe`, and the data are not in the same folder as the data. It is crucial that the spreadsheet, `Service,xls`, and the simulation executable, `MySigmaSimulation.exe`, be in the same folder and the director path to this folder be the default for Excel. There are two directory paths of concern and they have to be the same in order to run your simulation correctly. These two paths are (1) the Excel-path (Excel's default path) and (2) the Interface-path (the path at which your XL-interface and other necessary files locate—e.g., `Timing.dat`, `MySigmaSimulation.exp`, `MySigmaSimulation.exe`, and the output files). These two paths must match because otherwise all the output files generated by the EXE will be stored under the default directory of Excel—the Excel-path. If your XL-interface is under a different directory, it will not be able to access these output files unless other options are taken.

Having the whole path of the output file specified in the first column of EXP file will not solve this problem because the spaces within the path (if any) will cause the EXE to consider the whole path as several input parameters. One way to solve this problem is to change the default path of Excel manually within Excel. However, this method is not satisfactory either since the user has to change the path every time they move their XL interface to other directory.

The solution recommended here is to use the "ChDir" and "ChDrive" API commands build into VBA to explicitly change the default drive of Excel and path to where your XL spreadsheet interface and simulation executable are located. ChDir changes the default path of Excel to the path at which your XL interface locates. These two procedures have to be done separately because ChDir will not change the drive. The code to do this is simply

```
'Change the current drive and directory
strPath = ThisWorkbook.Path
strDrive = Mid$(strPath, 1, 1) 'Get the drive letter: e.g. in "C:\Sigma" is "C"
ChDrive strDrive
ChDir strPath
```

This code is found in Module1 in the Project Explorer (opened by pressing Ctrl-R in the VBA editor if it is not open).



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

To use them in your own interface, simply copy and paste them on the top of your module.

Execute the Simulator: The heart of the RunExe subroutine is the call to Shell.

```
Shell(strExe, vbNormalFocus)
```

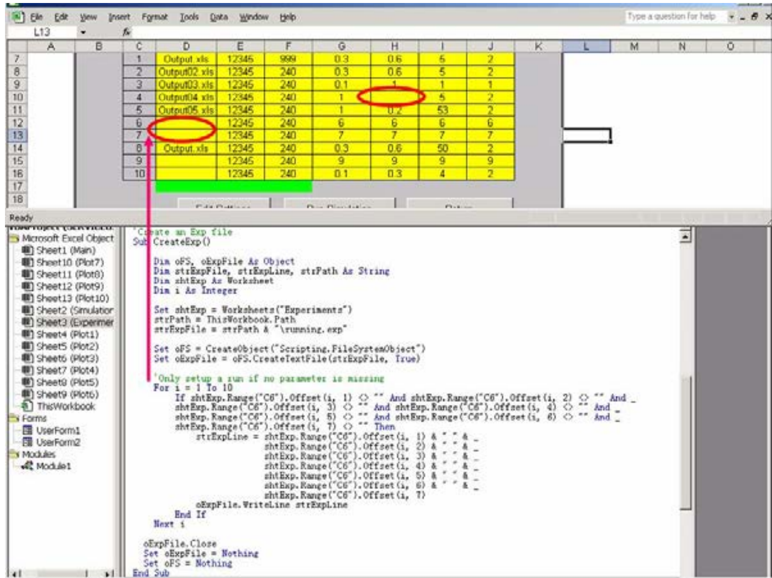
This API function call in this application will have parameter values of `strExe = c:\sigma\XLDEMO2\MySigmaSimulation.exe`, which is the name of the simulator program to be run an path, `vbNormalFocus` is a defined constant, which tells us to open a command window to watch the executable run. When using the Student version of SIGMA, you need to enter a slowly changing random integer when prompted to execute each run. Professional users can set this second parameter to vbHide and this window will open, but remain hidden from the spreadsheet user.

> Remember: If you completely hide your simulation execution from the user by using the vbHide option for the shell command, remove the following two lines in the translated C code (two lines after // experiments terminated)
>
> .
>
> ```
> scanf("%1s",&keytoclose);
> fflush(stdin);
> ```
>
> before compiling. Otherwise your executable will just run forever. It is much safer to use the vbNormalFocus command so the user sees what is happening during the simulation run

Again, to understand what each line of code does. Click on `RunExe( )` and press F8 repeatedly to single step through the VBA code and press F1 for more information.

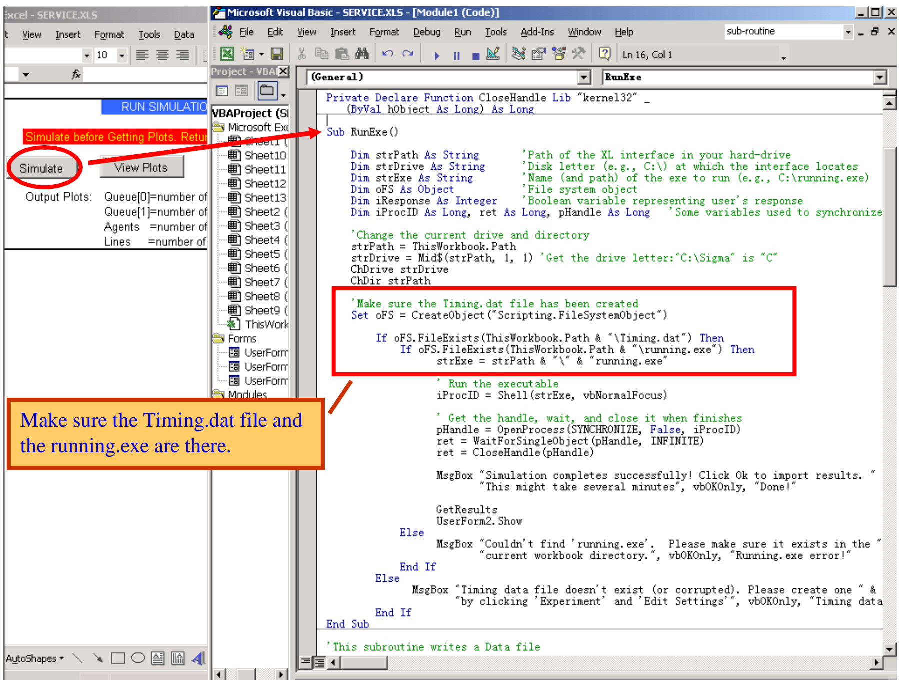
> Note: you can control the execution of several completely different SIGMA simulations from the same spreadsheet with multiple calls to the "Shell" API command selected using IF conditions.

Running an experiment without filling in all input parameters could cause errors. Therefore, when creating the EXP file, we have the following procedure to prevent this problem from happening. This procedure is embedded inside the sub routine "CreateExp()," which is defined inside module1.

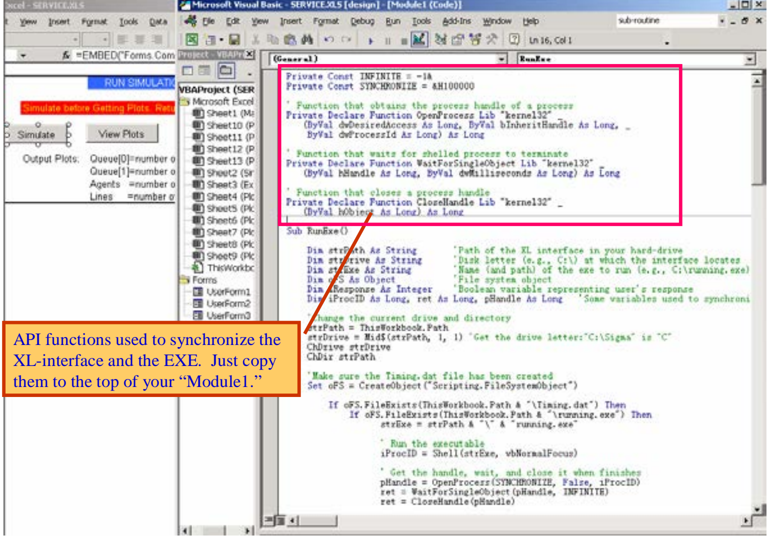


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Before running the executable, `RunEXE()` will make sure that both the `Timing.dat` and the `MySigmaSimulation.exe` are there. It does not check for the experiment file because users might want to input experimental data manually from the command prompt.

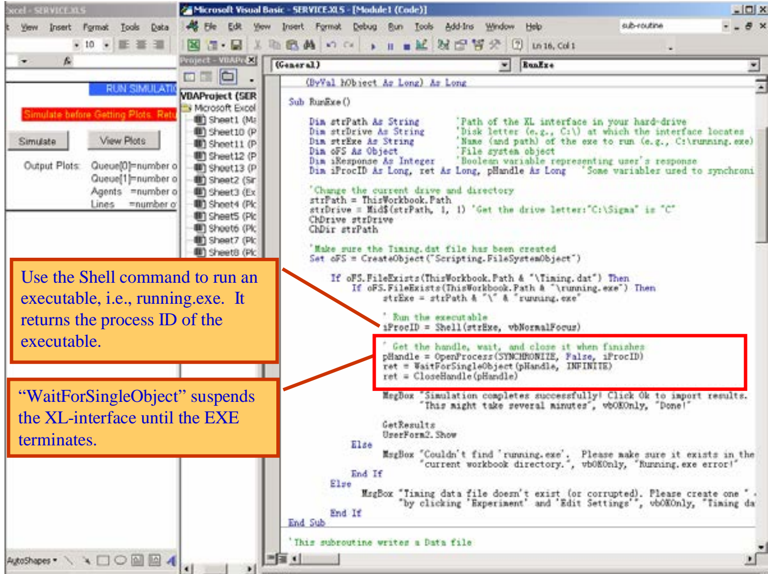


Make sure XL waits for the completion of the simulation: To have Excel wait for the executable to finish producing the output before trying to read it we use three API functions built into Excel (i.e., synchronizing these two processes). These functions are defined at the beginning of Module1.

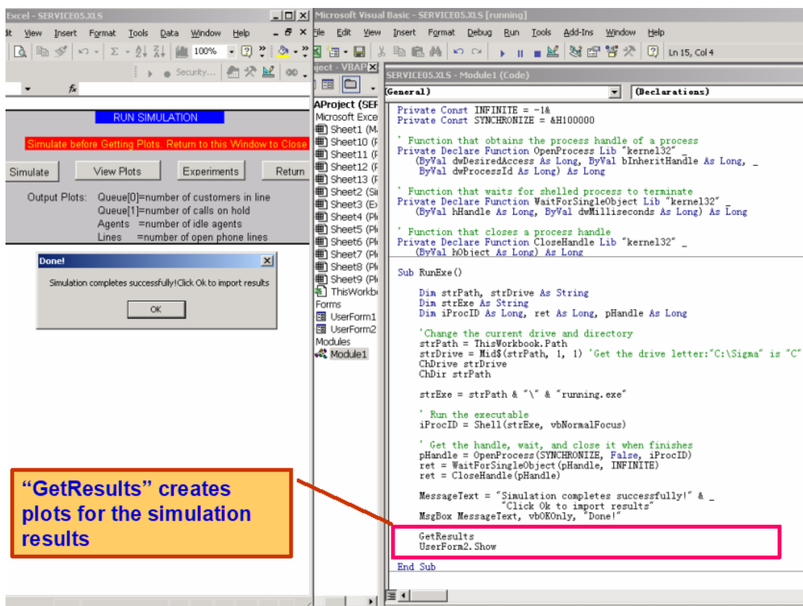


When the Shell command is used to run an executable (here, MySigmaSimulation.exe), the return value of this command is the process ID (called the "handler") of the executable. This ID is then used by the API function OpenProcess() (one of the three API functions defined earlier) to identify which process to synchronize. The API function WaitForSingleObject() will suspend the XL-interface, if the executable is still running, until the executable is finished. Finally, it releases the process ID handler used to identify the executable so as to free the memory. The code that waits for the simulation to finish is in the middle of this module and should be copied into a code module for your application.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

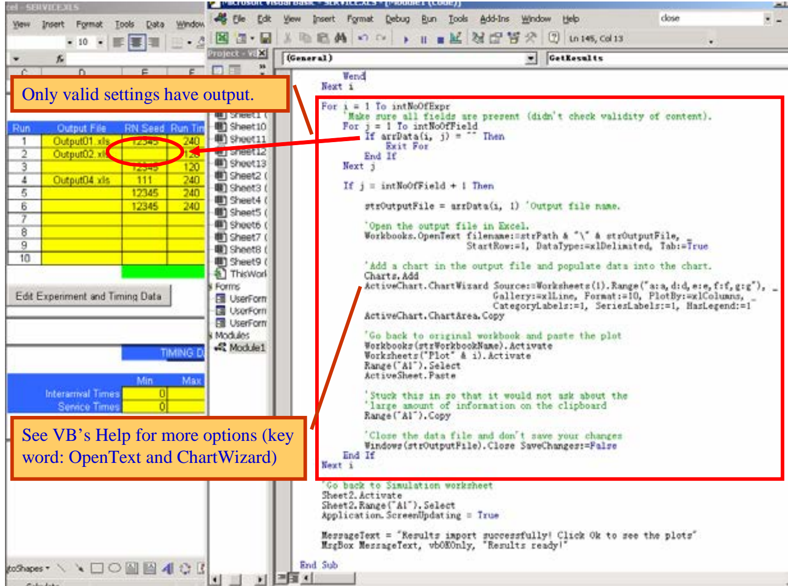


Import the results after the run into Excel: After the simulation finishes, a message box pops up to inform the user that the simulation run is finished and to warn the user that it might take several minutes to import the results from all output files in to Excel, which is done by the sub-routine `GetResults`.

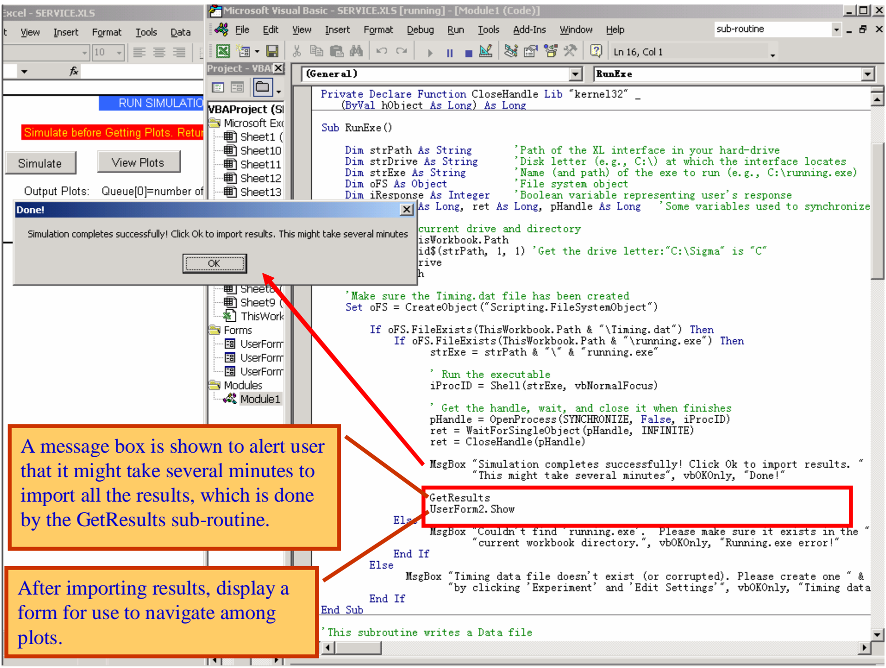


Creating plots showing the simulation results: The `GetResults` subroutine will open an output file, create a plot for the output, copy this plot to the main XL-interface, and finally close the output file. If n ($n \leq 10$) experiments are performed, this procedure will be repeated n times, creating n plots in your XL-interface, each of which is pasted on one of the 10 built-in worksheets. If more than 10 experiments are needed, then the workbook can be modified to accommodate more experiments and plots.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

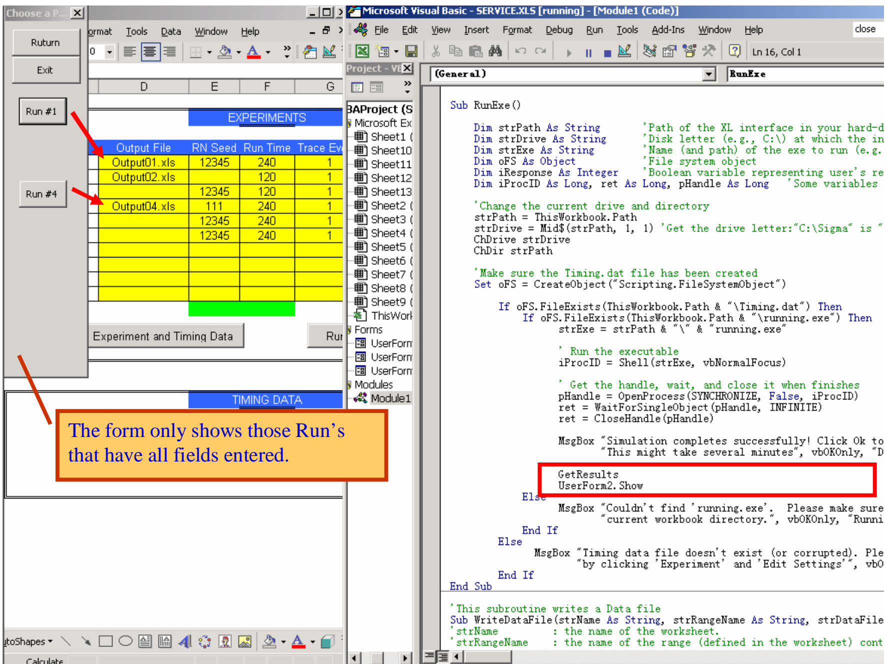


During the importing of the output files from the simulation runs, a message box warns that it often will take longer to import the data into Excel then it took for the simulator to create the output in the first place.



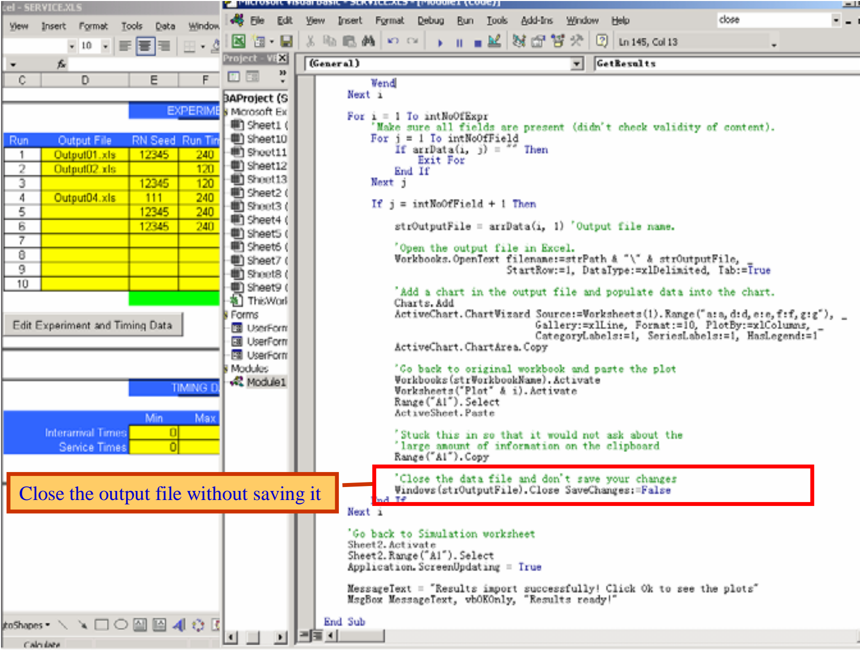
This form only displays buttons if the input row for run in the Experiment file was correct.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



Disable the Excel prompt to save the files: To disable the annoying message Excel issues when closing an unsaved modified output file without saving it, you can use the following code.

```
    'Stuck this in so that it would not ask about the
    'large amount of information on the clipboard
           Range("A1").Copy
   'Close the data file and don't save your changes
           Windows(strOutputFile).Close SaveChanges:=False
```

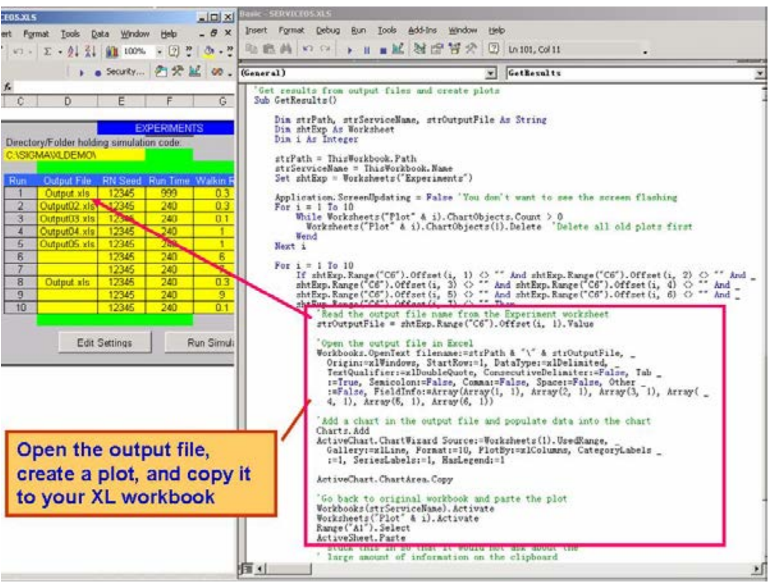


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Step 6. Create a form to display the simulation results

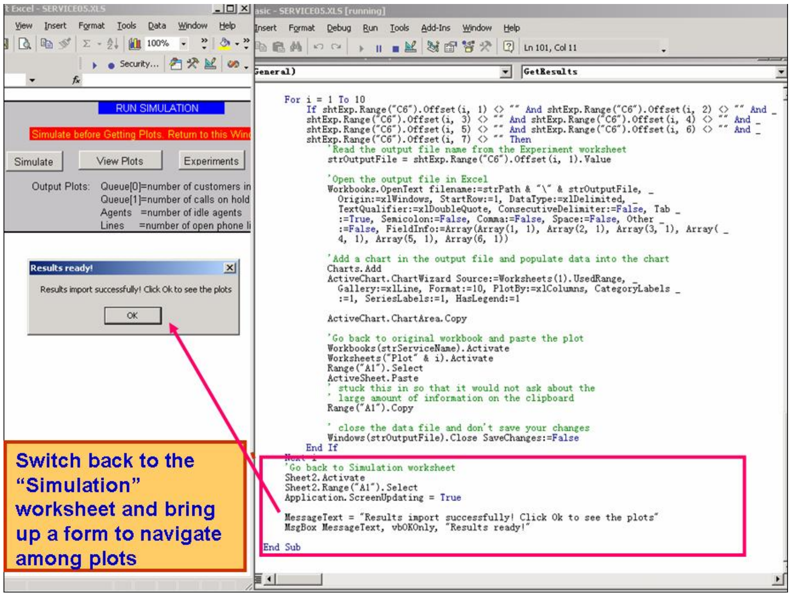
When a the batch of experiments in a run finishes, the "View Plots" button is clicked to show UserForm2 that is used to select which run output in the experiment to view. The `GetResults` subroutine created plots for the simulation results. The output from each simulation run is shown easily by clicking the corresponding button. Again, you should click on `GetResults` and repeatedly press F8 to learn what this code does.

The View Plots button on the simulation worksheet opens an output file, creates a plot for the output, copies this plot to the main XL interface, and finally closes the output file. If you run n (≤10) experiments, this procedure will be repeated n times, creating n plots in your XL interface.



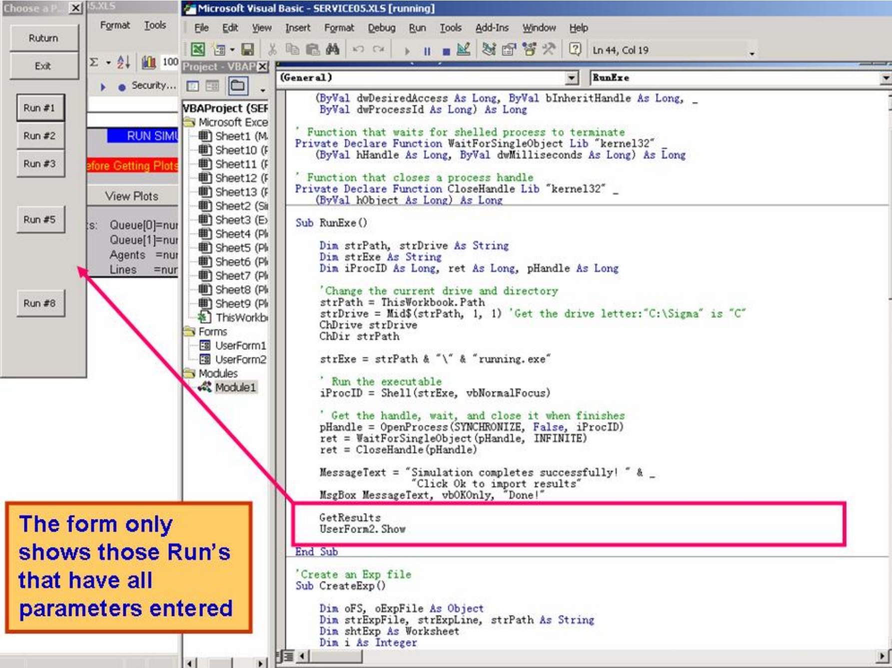
Selecting among the different Output Plots..

When all plots are ready, the interface switches back to the "Simulation" worksheet and brings up the navigation bar.

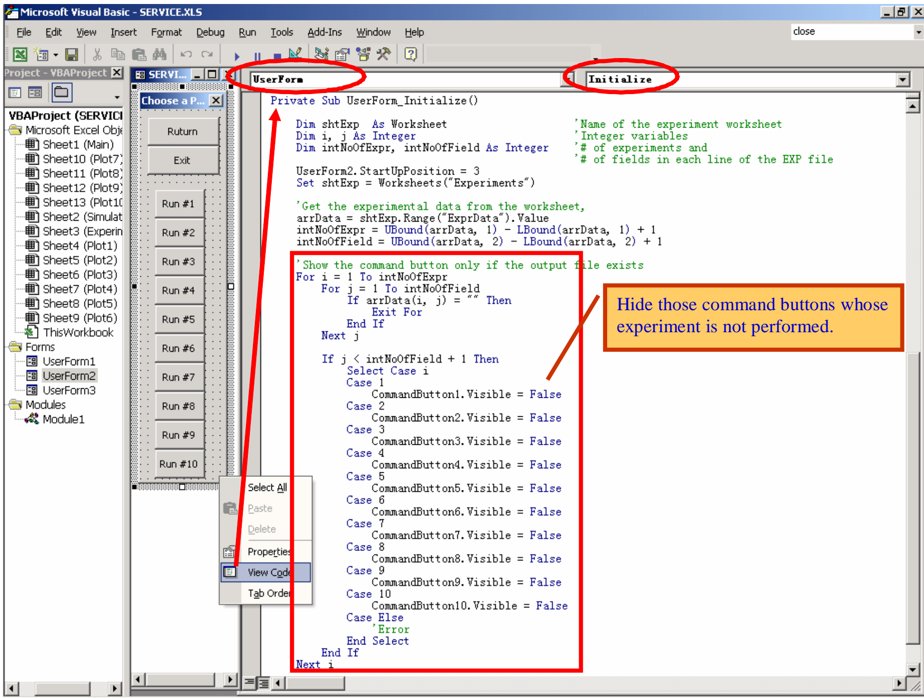


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Only those experiments that have been performed have output plots.

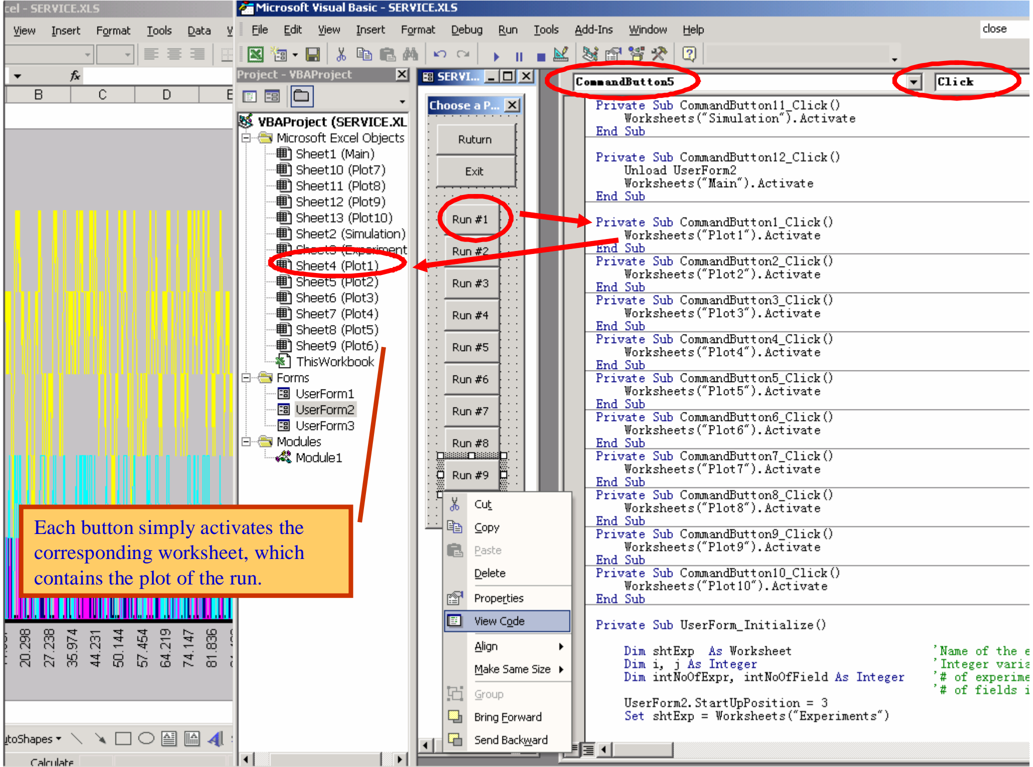


The Initialize sub routine checks the "Experiments" worksheet and disables those buttons for which the corresponding experiments are not performed.

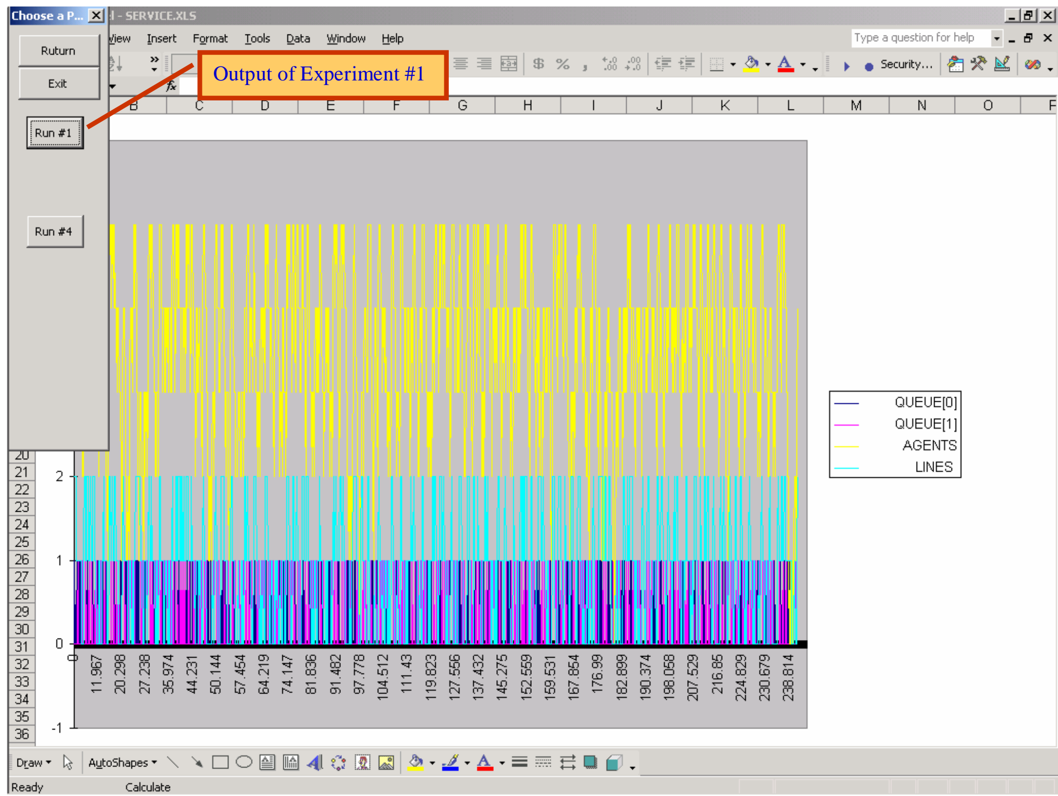


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

When the user hits a button, the corresponding worksheet containing the output plot will be activated. At most 10 plots are allowed in this example.



As an example, the following plot is from experiment number 1.



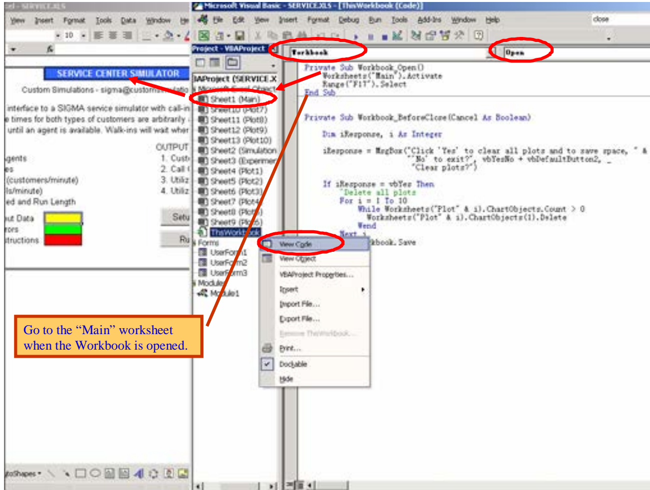
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Step 7. Miscellaneous details

Set default starting worksheet to the Main worksheet: Rather than have Excel start where the last session ended, it is better to direct the user to the “Main” worksheet when the spreadsheet is opened. This done by the code in the ThisWorkbook object in the Project Explorer.

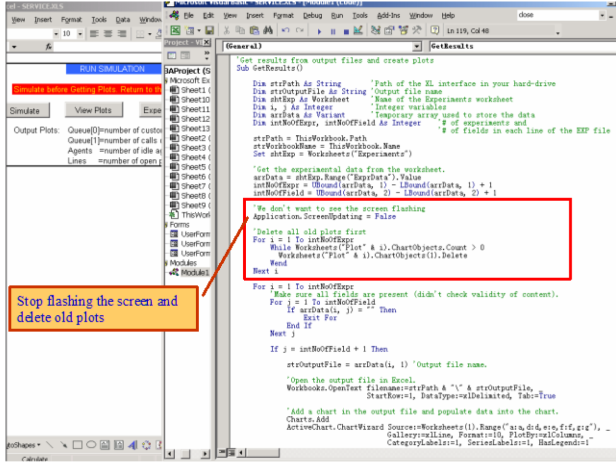
```
Private Sub Workbook_Open()
    Worksheets("Main").Activate
    Range("A1").Select
End Sub
```

The sub-routine Workbook.Open() is the first sub-routine to execute before the workbook is seen on the screen; therefore, we can add code into this sub-routine to activate the “Main” worksheet. Right click the ThisWorkbook and select “View Code.”



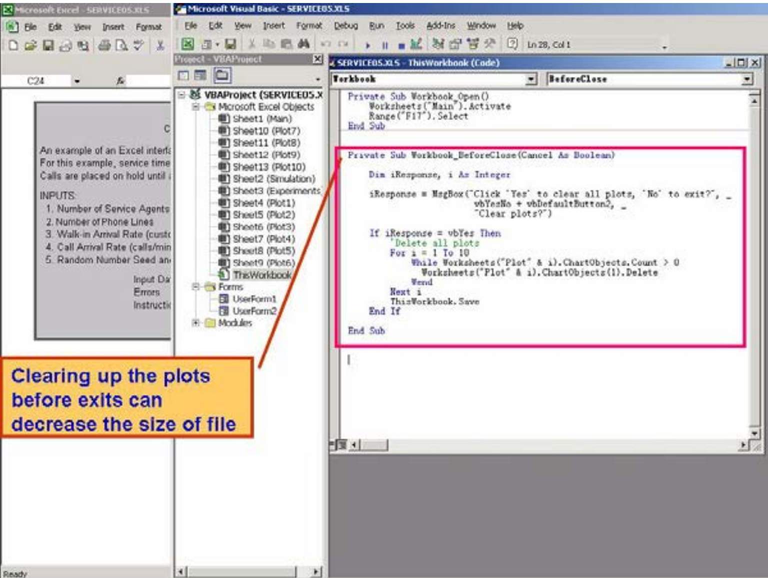
Stop screen flashing by using Application.ScreenUpdating = False: If you run 10 experiments, there will be 10 output files and `GetResults` will open them one by one cause screen flashes. To stop the screen from flashing repeatedly, include the following code.

```
'We don't want to see the screen flashing
Application.ScreenUpdating = False
```

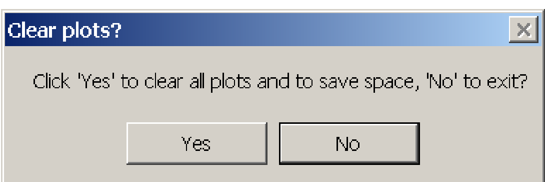


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

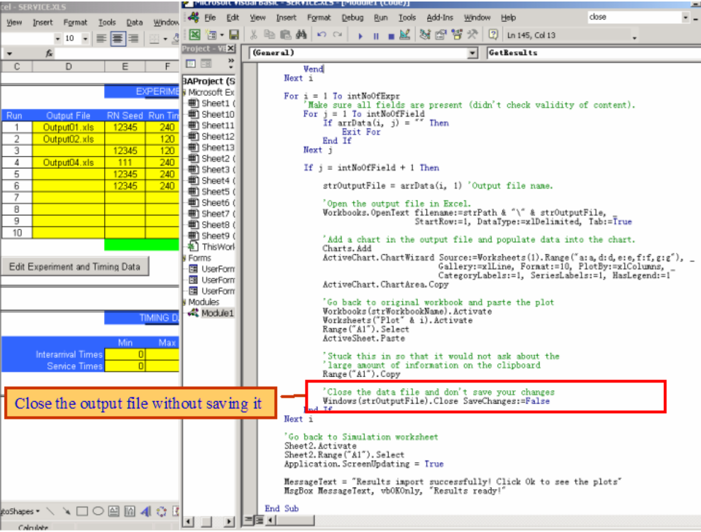
***Clear plots when exiting to save space*:** Since the plots could contain an enormous of data, it is recommended that you clear them when closing the spreadsheet. This code is also found under the ThisWorkbook object.



and displays the following messagebox to the user.



Close the output without saving it: Like in the `GetResults` module, we want to avoid the message asking if you want to save your output plots for each one of the worksheets created for each run of the experiment.



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Using this example as a template and the source of module codes, you should now be able to create professional looking dynamic spreadsheets run simulations that run SIGMA-generated simulators. Again: you should click on the code and press F8 repededly to step through this code and use help (F1) for more information to get acquainted with VBA.

11.5 Replacing the SIGMA Pseudo-Random Number Generator

SIGMA uses the pseudo-random number generator function, `lcg()`, provided in the `SIGMALIB.LIB` file on your disk. Like all pseudo-random number generators, `lcg()` is not perfect. You might want to replace this generator in your SIGMA-generated source code with one of your own. Fortunately, this is easy to do. Simply add your function your model and change the definition of `RND` as `lcg()` in the `SIGMALIB.H` file. A detailed example is given in Appendix B.18.

11.6 Exercises

11.6.1 Translations

Translate any of the models in Chapter 5 into C and run them.

11.6.2 Batched experiments

Create an experiment file for a batched experiment with any model you translated and compiled in the previous exercise.

11.6.3 Spreadsheet interface

Create a spreadsheet interface for the experiment in the previous exercise.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

12

Advanced Programming Techniques

Chapter 12 contains numerous advanced techniques that allow you to get the most out of your simulation programs. Techniques include how to start a run with simultaneously executing vertices, how to stop an event graph model on any general condition, how to use arrays of any dimension, how to run several replications of the same model in parallel, how to reduce the number of vertices in a model without changing the model's behavior, and how to use pre-emptive execution to eliminate the need to schedule some events.

12.1 General Starting and Ending Conditions

If several vertices are to be simultaneously executed or scheduled at the start of a run, create a start-run vertex (called, say, `RUN`) as vertex 1. Create edges from this vertex that schedule all the other vertices unconditionally with zero delay times and correct execution priorities.

To stop a run on any general condition, simply create a `STOP` vertex as the counted event. Add appropriate edges entering this vertex, conditioned by the desired halting criterion. In SIGMA, set the ending condition in the Run Options dialog box to Stop on Event, locate the `STOP` vertex in the drop down list, and click on it. The Iterations box should have a count of 1.

12.2 Simultaneous Parallel Replications

To assess the variability of a simulation output series, it may be desirable to run several independent replications of the same model. Parallel replication has a number of advantages, such as making the detection of initialization bias much easier.

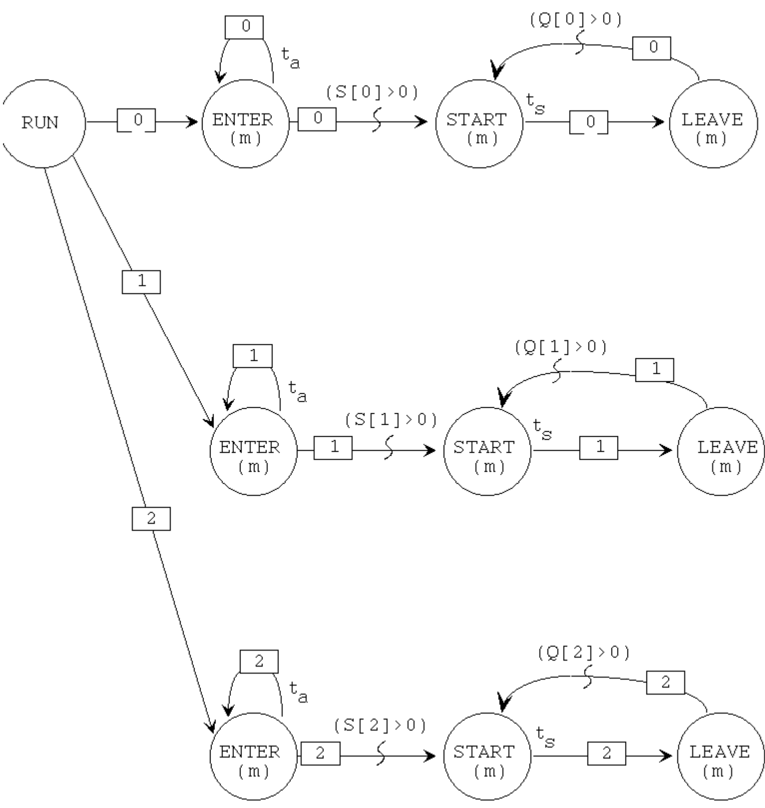
With event graphs, it is very easy to run these replications simultaneously. Simultaneous replication of a model is another application of event parameters and edge attributes. It might be useful here to think of parallel replicated simulations as an array of identical event graphs that are to be run at the same time.

To set up parallel replications, simply add two state variables to the model: one variable to identify the replication to which a particular event occurrence belongs and the other variable to tell how many simultaneous replications are desired. The replication index is globally added to the end of all edge attribute and event parameter strings. Also, every one of the original state variables gets an added array dimension for this index.

Consider the model of the single server queue (`CARWASH.MOD`) in Figure 2.6. An event graph for running three simultaneous replications of this model is given in Figure 12.1. The problem with this model is that, in addition to being messy, each time another replicate is desired a new sub-graph has to be appended.

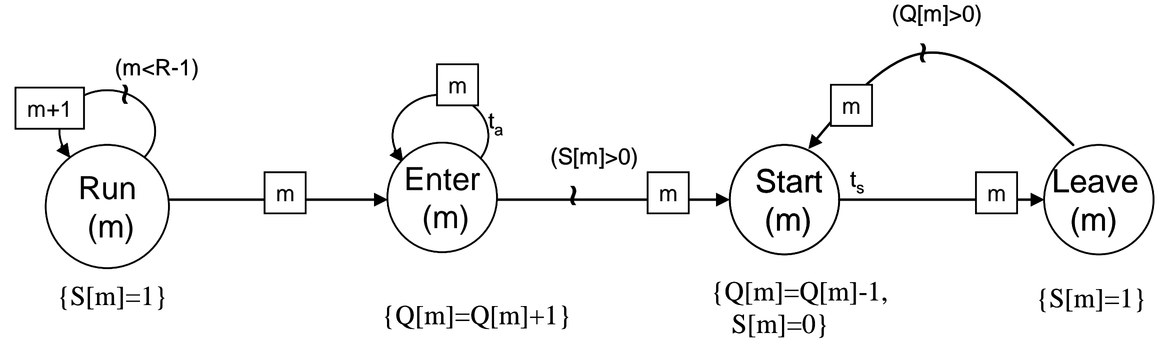
©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 12.1: Three Simultaneous Parallel Replications of a Queue



We can solve this easily by using parameterized events; define `R` as the desired number of replicates and `M` as an index to identify the replicate to which an event vertex belongs. The resulting parameterized event graph now looks like that in Figure 12.2.

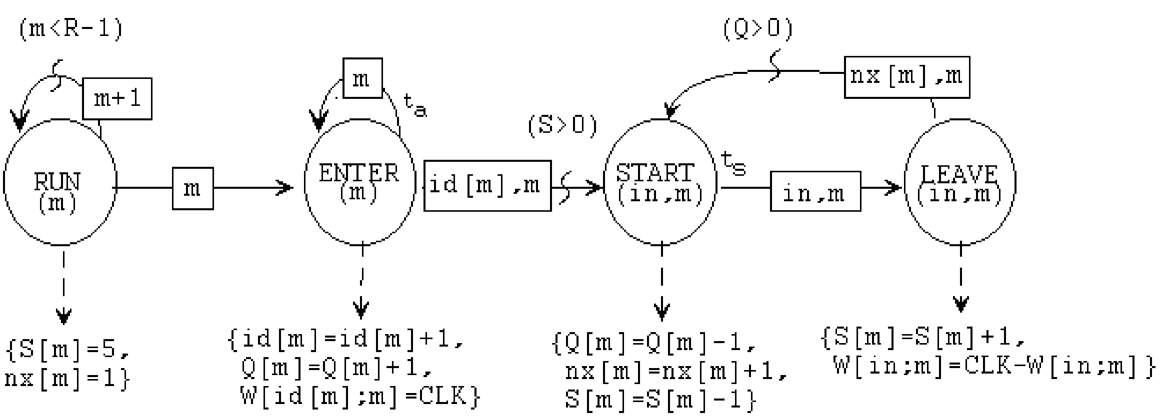
Figure 12.2: Simultaneous Parallel Replications of a Queue



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Finally, as a more complicated example, consider BANK2.MOD discussed previously, where we kept track of transient entities. Our queueing simulation event graph is parameterized for R simultaneous replications in Figure 12.3**.** The event graphs in Figures 12.2 and 12.3 are complicated and good tests of your comprehension. Fortunately, event graphs for these models look almost the same as the simple model in Figure 2.4. In fact, these models were created from our basic queueing model by using an ASCII editor to globally add the parameter, M, to all edge attribute and vertex parameter strings and also to make M an additional state variable dimension. The only other change needed was to set up a loop on the initial RUN vertex to start each replicate.

Figure 12.3: Simultaneous Parallel Replications of a Queue with Five Servers Collecting Customer Waiting Times



Notice that if the same arrival rate is used for all three replications, one ENTER event could be used to generate arrivals for all three replications. This would automatically implement the variance reduction technique of common random number streams discussed in Chapter 9**.** If our replicates represented small differences in some of the input parameter values, this technique could be used to estimate the gradient of the simulation response in a "single run" using finite differences between the averaged output series.

An implementation of parallel replications in SIGMA is given in the model, REPLCATE.MOD. Here we set up our basic model, CARWASH.MOD, for simultaneous parallel replication. The SIGMA graph for this model looks identical to that for CARWASH.MOD in Chapter 3, with the single exception of the "do loop" for the RUN vertex. In the RUN vertex, each replication is STARTed and indexed by the value of M passed as a vertex parameter.

12.3 Event Graph Reduction

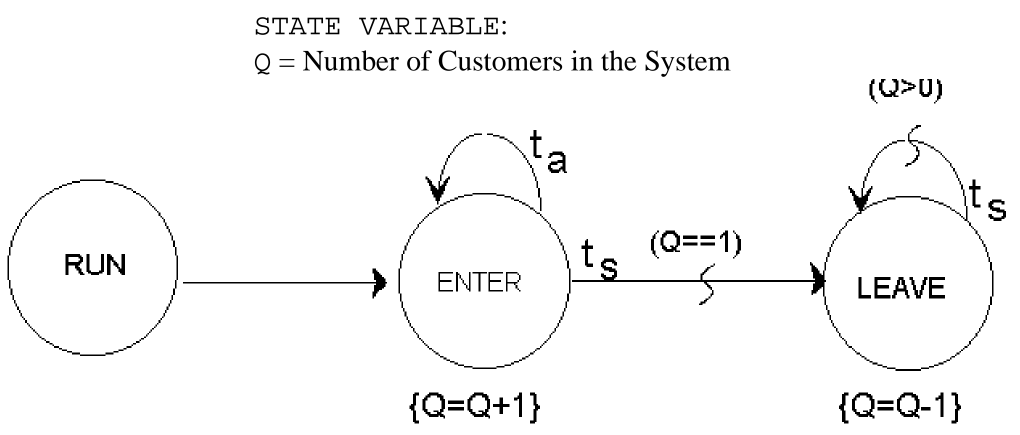
It is sometimes possible to reduce the number of vertices in an event graph without changing the behavior of the model. Simulations with few events will often run faster than models with more events, although this is not always the case. An event graph with fewer vertices may lead to a more efficient simulation program; however, the graph may be more difficult to understand. Generally, models with a large number of very simple vertices will be easier to debug or modify than models with fewer but more complicated vertices.

It is possible, however, to increase the efficiency of a simulation without increasing the complexity. This is done by using pre-emptive vertex execution, a scheduling delay time for an edge given by an asterisk (*). Recall that a delay time of * means that the scheduled destination vertex is executed immediately, without being placed on the future events list. It pre-empts any other events that might be scheduled at the same time. This often results in a significant reduction in simulation execution time.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

We will use our basic single server model to illustrate event graph reduction. The START vertex can be eliminated if we redefine Q (queue) to be the number of customers in the system, including any customers in service, rather than the length of the waiting queue. The resulting graph is shown in Figure 12.4.

Figure 12.4: Event Graph Without a START Event



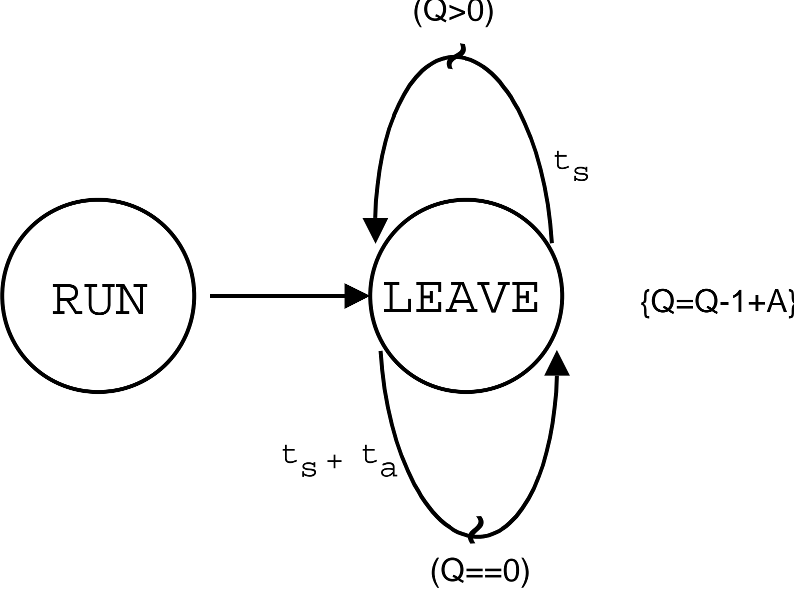
To get the same run time efficiency, the START vertex can be effectively eliminated without changing the graph except to make the delay times on the edge from ENTER to START and the edge from LEAVE to START asterisks (*) rather than zero. This would cause the execution of the START event to occur without ever scheduling it on the list of future events.

It is possible to eliminate all of the vertices in our model except the RUN vertex (which is executed only once anyway) and the LEAVE vertex. This would require that we generate the (random) number of arrivals to the queue during the time that a single customer is served. We will call this number of arrivals during a service interval, A. In some simple cases, a value for A can be generated very easily (e.g., if the times between customer arrivals are exponentially distributed, A will have a Poisson distribution). The single-event model for our queue is shown in Figure 12.5.

Figure 12.5: Single Vertex Model (RUN and LEAVE)

STATE VARIABLES:
Q = Number of Customers in the System
A = Number of Arrivals During a Service

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



While the model in Figure 12.5 might run very fast, it is difficult to enrich; therefore, it is not as useful as some of our earlier models for this system.

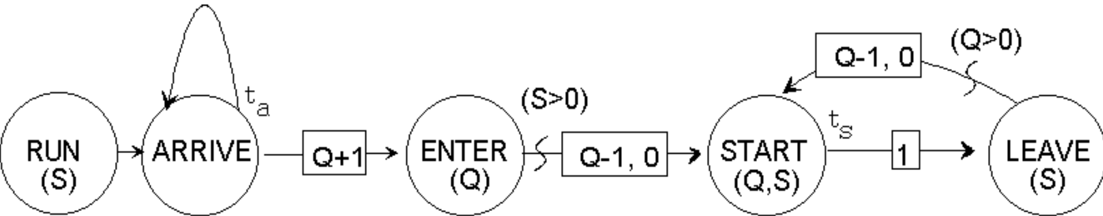
It is possible to use edge attributes to assign values to vertex parameters rather than using state changes. At first glance, this might seem like a complicated way to do things, but it is useful. Sometimes edge attributes make the model clearer; sometimes they do not. We will illustrate this technique with our single server queue, where the status of the server, S, and length of the waiting line, Q, will become parameters of the START and LEAVE vertices. The values of these parameters are passed as edge attributes. When an entering customer finds the server idle (S>0), the customer will begin service, making the server busy; the value 0 is passed as an attribute of the edge from ENTER to START into the START parameter, S. The result is the same as having the state change S=0 in the START vertex. Similarly, when the server finishes service, its status will change back to idle (1). This is done by passing the edge attribute value, 1, along the edge from START to LEAVE into the parameter, S, of the LEAVE vertex. This is shown in the event graph in Figure 12.6 (NOSTATEQ.MOD).

Figure 12.6: Event Graph That Uses Edge Attributes to Assign Values to Vertex Parameters

STATE VARIABLES:

Q = Number of Customers Waiting in Line, initial value set to zero, by default.

S = Status of Server (0=busy/1=idle), initially set equal to 1=idle.



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

We can do this since there is no chance of the state variable, S, changing during the time intervals represented by the delay times of the edges from ENTER to START and from START to LEAVE. If we had more than one server in this system (S initially greater than one), we would not be able to do this since S might be changed by more than one instance of an event vertex.

12.4 Using Arrays of Arbitrary Dimension

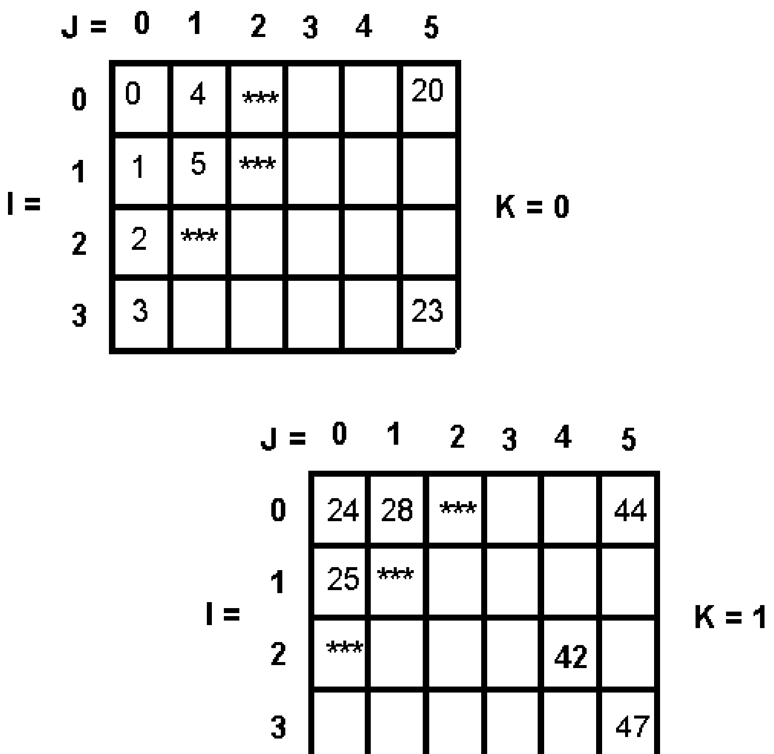
Four-dimensional arrays will be sufficient to meet most simulation modeling needs. However, if you find it necessary to have higher dimensional arrays, the method discussed here is easily implemented. Assume that one conceptually is working with a two-dimensional (*N* by *M*) array, *A*. That is, the element in the *i*th "row" and *j*th "column" of this array is the value of the variable $A_{i,j}$. The equation, $L = A_{i,j}$, written in SIGMA is `L = A[N*J+I]`. Recall that we start indexing arrays with zero, not with one.

This indexing scheme can be extended to more than just two dimensions. For example, element, $B_{i,j,k}$, of an *N* by *M* by *L* array would be the element, `B[N*M*K+N*J+I]`, of a single dimensional array, `B[.]`. The logical structure for a three-dimensional array of size (4,6,2) is shown in Figure 12.7. Each element of the single dimensional array, `B[.]`, is shown in a cell.

A more sophisticated array indexing scheme uses "ragged" arrays, which sometimes saves memory. Here the total number of elements in all rows strictly less than `I` is the value of the variable `M[I]`. The `i,j`th element of the ragged array, *A*, is given by the nested arrays `A[M[I]+J]`. When an element is added to the array, both the `M` and `A` arrays are updated (resulting in some additional computational overhead). Ragged arrays can be useful when the rows of a matrix are of unequal length (for example, the `i,j`th element is some characteristic of the `i`th customer in the `j`th queue). The trade-off in using a ragged array is favorable when the rows (or columns) of the array are of significantly differing lengths.

Figure 12.7: Representing a 4 by 6 by 2 Dimensional Array asElements [24*K + 4*J + I] of a One-Dimensional Array. For example: Element [2,4,1] is [24*1 + 4*4 + 2] = [42].

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



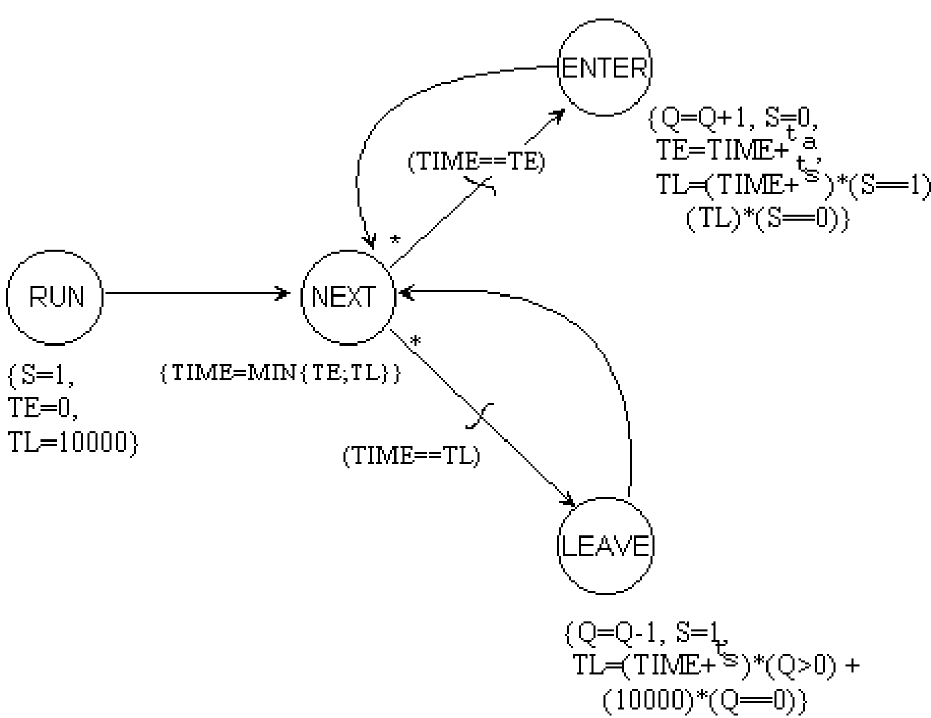
12.5 Eliminating Event Scheduling

By using pre-emptive vertex execution (delay times = *) it is sometimes possible to run a model without scheduling events. The model, NOLIST.MOD, does this for a single server queue. The graph for this model is given in Figure 12.8.

In this model, the times of the next events are computed as state changes and the simulated virtual time (normally kept by SIGMA in the variable, CLK) is updated in the vertex, NEXT. The simulation generated from this model runs very fast but is difficult to modify. For readers interested in the more theoretical aspects of event graph modeling, the graph for this version of a single-server queueing system is related to the geometric dual of the event graph of our earlier model of a single server queue, CARWASH.MOD.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Figure 12.8: Event Graph Without Scheduled Events



12.6 Exercise

12.7.1 Multi-Dimensional Arrays

Write an expression that will translate a point (v, w, x, y, z) in a five-dimensional array P with dimensions I by J by K by L by M into a one-dimensional array q.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Appendix A

Event Execution Sequence

In SIGMA, a mathematical expression can be used almost anywhere a variable can be used. This includes vertex state changes, edge delay times, and edge attribute values. In fact, SIGMA allows the execution order of simultaneous events to be state dependent through the use of mathematical expressions to model dynamic edge priorities. This gives SIGMA additional modeling power and allows rather complex applications where state-dependent event sequencing is important. In SIGMA the order of execution for an event vertex is as follows:

1. Set the vector of parameters for this vertex to the values of the edge attribute expressions placed on the future events list when this vertex was scheduled.
2. Make the state changes specified for this vertex.
3. Evaluate the edge conditions for *every* edge exiting this vertex.
4. For each exiting edge with a true edge condition:
   a. If the edge is a vertex scheduling or pending edge,
      i. Evaluate each of the edge attribute expressions.
      ii. Compute the edge delay time.
      iii. If the delay is equal to a *, immediately execute the destination vertex and go to the next event; otherwise, compute the edge execution priority and schedule the destination vertex for this edge onto the future events list.
   b. If the edge is a vertex cancelling edge,
      i. If the edge attribute is equal to a *, cancel all scheduled vertices that are the same type as the destination vertex, regardless of the values of their attributes.
      ii. If the edge attributes are left blank, cancel only the next scheduled vertex that is the same type as the destination vertex, regardless of the values of its attributes.
      iii. If the edge attributes are not blank and not equal to a *, cancel any vertices that are of the same type as the destination vertex with the exact same attribute values as this cancelling edge.

Most of the time you will never have to think about execution order; however, you should be aware of it. The intuitive notion of an "event" would be a sub-graph with zero-delay times on all of the edges in this sub-graph. Event graphs are merely pictures that decompose events into manageable logical components.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Appendix B

Reading SIGMA-Generated C Programs

This appendix is intended to help persons not familiar with the C programming language understand the C simulation programs generated by SIGMA; it is not intended to be a comprehensive introduction to C. You should look at the C simulation, `CARWASH.C` to find examples of C program elements discussed here. Consult a good reference on C (Kernighan and Ritchie, 1988) for correct definitions and details on the C language. SIGMA-generated C programs also run under C++, which includes objects with the C language as a subset.

There are several powerful features of C that are not used in the simulations generated by SIGMA. These include structures, pointers, "do-while", etc. Learning C or C++ is strongly encouraged as it will allow you to greatly improve the efficiency of your simulations. The library of simulation functions, `SIGMALIB.LIB`, makes more extensive use of the C language than do the simulations generated by SIGMA.

B.1 Functions

C programs have a simple and elegant structure. C programs are composed of *functions*, that is, subprograms that return a single value when called by another function. Functions are identified by function names followed by parentheses, `()`. The parentheses following a function name contain the parameters for the function; these are the names of any variables whose values the function will need. Every C program has a master function called, `main()`, where execution starts. The `main()` function typically does little else except initialize variables and call other functions. The functions in a C program should be defined at the top of the program or in a file (called a header file) that will be included in the program before it is compiled.

B.2 Parameters

Parameters for functions have only their values passed to the function, not the variables themselves. Therefore, the values of the parameters passed to a function are not changed in the program that calls the function (unlike some implementations of FORTRAN).

B.3 Comments

Comments to increase understanding can be included anywhere in the code; they are ignored by the compiler. Comments begin with the delimiter, `/*`, and end with the delimiter `*/`; they cannot be nested. In C++, the delimiter `//` denotes a comment that ends at the end of the line. The code generated by SIGMA has extensive commenting with most comments being the descriptions you entered when defining the variables, vertices, and edges in your event graph model.

B.4 Variables

All variables and functions must be *declared* by giving them a *type* and a *name*. Variable and function names are sequences of letters or digits starting with an underscore, _, or letter. These identifiers may be of any length; however, some compilers only distinguish as few as the first six characters. Case is important, e.g., the variable, `TimeOfDay`, is entirely different for the variable, `timeofday`. The code you created in your SIGMA model as well as SIGMA key words will appear in the C translation in upper case letters. All other C code generated by SIGMA is in lower case. This way you can tell instantly the C code you created that is specific to your simulation from the generic C code needed to run your program.

> **Important:** *Code that you created that is specific to your* SIGMA *simulation will appear in upper-case characters. Generic C code is in lower-case.*

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Variable definitions consist of the variable type followed by one or more variable names. Examples of C variable type definitions used by SIGMA include:

```
int i, J;     /* i and J are defined to be short integers */
long k;       /*k is defined to be a long integer (with more
              significant digits)*/
float g,f[6]; /* the variable, g, and array, f with 6 elements are
              defined to be real (floating point) numbers with
              decimals */
double h;     /*the variable, h, is defined to be a double-precision
              floating point number */
char target[10], /* target and date
    date[100];  are defined to be strings of characters of the
              specified lengths, the size of the string should allow
              for the final null string termination character, '/0'.
              Indices of n element array is 0 to n-1.  */
int *p;       /* the variable p is a pointer to the address of an
              integer */
```

The last example is a pointer that will have as its value the memory address of an integer variable. To find the value of the memory address of a variable, place & in front of the variable name.

Using only upper-case characters for SIGMA expressions to avoid confusion with lower-case reserved words used by ANSl Standard C is not fool-proof. For example, it is acceptable in SIGMA to name an event vertex EXIT or exit. However, if you use a Microsoft C (Version 5.1) compiler, these words have special meaning. This seems to cause no problems if you use other compilers. Of course, these other compilers also have functions added to their libraries, and you must avoid their function names.

B.5 Statements

Simple statements in C can be made as in most other languages, with a variable being set equal (=) to an expression. C statements are grouped into related blocks of code delimited by braces, {}. These blocks of code determine the ranges of variable definitions as well as the range of iterations (loops) and sequencing (branching) statements. A statement ends with a semicolon (;) but is otherwise free-form, starting and ending in any column.

The C arithmetic operators used by SIGMA include addition (+), subtraction (-), multiplication (*), and division (/). C has a short-hand notation that is popular but redundant. In this notation, the statement X=X+7 might be written X+=7 and the statement X=X+1 might be written simply as X++. This short-hand is irritating to people new to C, but in general people like it once it is familiar. The power operator (^) used in SIGMA does not exist in C; the function, pow(), is used instead. Relational operators for comparing two expressions include greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equal to (==), and not equal to (!=). Boolean logical operators in C are "&&" (and) and "¦¦" (or).

As in SIGMA, there are "side effects" to using functions to test edge conditions. For compound edge conditions using Boolean "and" operators, many C compilers will evaluate only until the first false condition is found. SIGMA, on the other hand, will evaluate all conditions before *and*ing. For example if you have an edge condition (Q>0&PUT{LINE;FST}) in SIGMA, the PUT function will always be executed. This condition translates to C as (Q>0&&PUT(LINE,FST)), and the PUT function may only be executed if Q>0, depending on which C compiler you use.

C has bit operators which are not used by SIGMA. Note that in SIGMA, Boolean "and" and "or" operators are & and | respectively, which are bitwise operators in C. In C, ^ denotes the "exclusive or" bitwise operator.

B.6 Directives

Lines that start with a pound sign (#) are called directives. Some examples of directives are:

```
#include sigmalib.h       /*insert the code in the header file, sigmalib.h,
                          in your default directory*/
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
#include <stdio.h>           /*insert the file stdio.h from the include
                             subdirectory of your C compiler*/

#define TRUE 1 *             /define the constant TRUE to be equal to the
                             integer 1*/

#if defined MSDOS            /*if there is an environment variable called MSDOS
                             defined execute the code until an #endif directive
                             is reached*/

#define SIN(a) (sin(a))      /*define the upper case function SIN(a) to be
                             equivalent to the standard C function sin(a) with
                             the same argument*/
```

Substitutions indicated by directives are executed in the first phase of compilation. Directives statements do not end in semicolons.

B.7 Sequencing or Branching

The sequence of statement execution can be controlled in several ways. The most common is the `if-else` construct which has the following format:

```
if(condition)
   {
   /* execute this block of code if the condition is true (not equal to
   zero)*/
   comments, statements and function calls
   }
else
   {
   /* execute this block of code if the condition is false (equal to
zero)*/
   block of code
   }
```

It is common to nest `if-else` statements within other `if` and `else` blocks of code. Another common flow control statement is the switch-case with the following format:

```
switch(expression)
   {
   case 1:
      block of code
      break;
   case 2:
      block of code
      break;
   ...
   case constant_expression:
      block of code
      break;
   default:
      block of code
      break;
   }
```

The block of code following the `case` equal to the `switch` argument is executed until a `break` is hit. For example, if the expression in the `switch` evaluates to the integer 6, only the code following `case 6:` is executed. The key word, `break`, prevents the execution of statements following the next case. C also has the `goto` statement, but this is very rarely used.

B.8 Iteration (looping)

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

To iterate a block of code several times, C provides the `for` and the `do-while` constructs. The `for` construct looks like this

```
for(starting_expression; ending_condition; looping _expression)
   {
   block of code
   }
```

This will execute the block of code if the condition expression is true, beginning with the execution of the starting expression and executing the looping expression each time until the condition expression is false. For example, the statement:

```
for(index=1;  index<6; index=index+1)
   {block of code}
```

will execute the block of code starting with index equal to 1 and increase the index by 1 for each loop as long as the index is less than 6. (Loop is not executed with index equal to 6).

The forms of while and do-while statements are

```
while(expression) {block of code}
```

and

```
do {block of code} while (expression)
```

In each case if the expression is true, the block of code continues to be executed (unless a break is hit). In the `while` format, the expression is tested before the block of code is executed, and in the `do-while` format the expression is executed after the block of code is executed. In each case, a `break;` will exit the loop and a `continue;` will pass control to the next iteration.

B.9 Organization

The typical main C program skeleton is as follows:

```
directives (lines beginning with #).
global variable declarations
function definitions
main()
   {
   local variable declarations, comments, statements,
   and function calls
...
   }
```

Typical C functions have the following skeletons

```
function_return_type  function name (list of parameters)
   parameter declarations
   {
   declarations, comments, statements and function calls
   }
```

The default type of value returned by a function is an integer. (A "void" type function does not return any value.)

B.10 Input and Output

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

In C, output is commonly done with a standard C function such as,

```
printf("format control string", list of variables);
```

This will cause the variables in the list to be printed in the default output device (typically the screen). The format control string dictates how the variables are to be printed.

The control string may contain ordinary characters and conversion specifications (preceded by a percent sign, `%`). Common conversion specifications are:

- `%s` - print as a character string
- `%f` - print as a floating point number
- `%d` - print as an decimal integer

Optional field specifications can also be included: `%20.10s` indicates that the field should be at least 20 characters wide, and 10 characters should be printed with right justification. `%-20.10s` will also print 10 characters in a field at least 20 characters wide but with left justification. Printing control characters can also be placed in the format control string, such as `\n` (end of line) and `\t` (tab). To illustrate, if `val` is a floating point number currently at 6.57643, `i` is an integer with a value of 3, and `name` is a string with value "profit", the statement

```
printf("The %s for quarter, %d, = $%-5.2f\n", name,i,val);
```

should result in the line

```
The profit for quarter, 3 = $6.57
```

being printed on the default output device.

Data is typically input into a C program using a standard library function such as

```
scanf("format string", pointers to variable addresses);
```

The format conversion string is similar to the `printf` statement. Since function arguments in C are passed by value not reference, it is important that the values of pointers to the addresses of variables being read in are given and not the names of the variables themselves;

```
scanf("%d",i);
```

would not read in a value for the variable, `i`; however,

```
scanf(%d, &i);
```

would**.** Placing the character `&` in front of a variable name indicates that we mean a pointer to the address of the variable. Examples of the `printf` and `scanf` functions can be seen when prompts for input are given in the simulation, `CARWASH.C`, on your SIGMA disk.

B.11 Referencing Disk Files

Files on your disk can be read from and written to using file pointers with standard C library functions. You need to define the special data type `FILE` and assign pointers to files of this type, e.g.,

```
FILE *i_file, *o_file;   /*pointers to files */
```

Files also need to be opened and closed using the `fopen` and `fclose` statements like,

```
i_file = fopen("input.dat","r");       /*open input.dat to read*/
o_file = fopen("output.dat","w");      /*opens output.dat to write*/
```

and

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
fclose(i_file);
fclose(o_file);
```

An example of a function that reads data from a disk file, `ROUTES.DAT`, and echoes to the screen is the following `INPUT` function:

```
/*** READING ROUTES FROM DISK ***/
void INPUT()
{
FILE *fp;  /*declare a file pointer for DISK input*/
int i, route[18];     /*index for reading data and the route*/

/*open the DISK file in read-only mode*/
fp = fopen("routes.dat","r");
/* loop to read 17 values from the disk file*/
   for(i=0; i<17; i++)
   {
   fscanf(fp,"%d",&route[i]);  /* read the data */
   printf("route[%d]=%d\n",i,route[i]);/*echo to screen*/
   }
fclose(fp); /*close the DISK file*/
}
```

It is strongly recommended that you replace the SIGMA `DISK{}` function with fscanf calls unless you are using some of the very special features of the DISK function.

B.12 Redefining Standard Input or Output Devices

When a C program is executed, the default input and output devices (the screen and keyboard) can be changed to any devices you want by using the < and > conventions. This is best illustrated with an example. You have an executable C program, `TEST.EXE`, on your `A:` drive whose input usually comes from the keyboard and whose output usually goes to the screen. To run this program with these default input and output devices you simply enter, `a:test`.

Now suppose that you want to read the input from a disk file named `INPUT.DAT` on your `C:` drive and you want the output to be written to the file, `OUTPUT.DAT`, on your `C:` drive , you execute the program by entering

```
a:test      <c:input.dat      >c:output.dat
```

The < indicates where the keyboard input will be coming from and the > indicates where the screen output will be written. This is useful in batching runs of a simulation where different input data and output files are desired for each replication. Be careful about redirecting the input; if the program needs information not in the specified file, you may have to reboot to use the keyboard.

B.13 Arrays, Pointers, and Data Structures

C has several features that are not exploited by SIGMA, including data structures and pointers. Pointers are variables whose values are the addresses in memory where data is located. The type of pointer (`int`, `char`, etc.) tells you how many bytes in memory are in the block of memory to which the pointer is pointing. (Pointers to functions are used in the support library, `SIGMALIB.LIB`, but you do not need to know about these to read the SIGMA-Generated C code.) A pointer is identified by having an asterisk `(*)` before the variable name. The operator, `&`, gives the address in memory of an object. If your program has the following declarations:

```
int var;   /* var is some integer variable */
int *p;    /* p is a pointer to the address of an integer */
```

to have "`p` point to `a`", you can write,

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved.   Written permission required for reproduction, distribution, or quotation.

```
p = &var;  /* assigns the address of var to the pointer p */
```

and you can increase the value of var by 5 with either

```
var = var + 5;
```

or

```
*p = *p + 5
```

Pointers are particularly useful in simulations when they are used to point to structures.

This brief introduction is not intended to give you a complete background to the use of structures and pointers but merely to give you some indication of their power. Hopefully, you will be encouraged to learn more about structures and pointers and use them in your models. A data structure can be thought of as the complement of a data array. Where an array gives you the value of a single variable when you specify one or more indices, a structure gives you one or more values for variables when you specify a single index (in the form of a pointer). An array can be used to represent many-to-one relationship and structures to represent one-to-many relationship.

Data structures can be used in simulations as records of information that pertain to a particular entity, i.e., school records for a group of students. Setting up data structures is useful when you need to keep track of many characteristics of an entity in your simulation model. Suppose you are simulating a factory and you want to keep track of each job's type, current task, location, start-time, routing, and time the job will take on each of several machines. If you assign a unique job identification number to each job in the system, you might use several arrays like we did in the SIGMA queueing network models discussed earlier. For each job, you could keep its type, current task, *x-y* coordinates of its location, and the time it was started in a collection of arrays like `TYPE[JOB]`, `TASK[JOB]`, `LOCATION[JOB;2]`, and `START_TIME[JOB]`. You could also have numbers representing the machine needed for each processing task for each job type in a table represented by the nested arrays, `ROUTE[TASK[JOB];TYPE[JOB]]`. The two-dimensional array `ROUTE[]` is an example of a two-to-one relationship; you specify two values, `TASK` and `TYPE`, and get one value of the machine number.

What we really want is to access many items of information about a single job, a one-to-many relationship. This is what a data structure does. A data structure set up to contain the relevant information for each job might look like the following in C.

```
struct tag {
   int      type;
   int      task;
   float    x_location;
   float    y_location;
   float    start_time;
   int      *route;
   }job;
```

You can also use structures within other structures. To reference an element of a structure you use a period, e.g., `if( job.task==2)` would test if the current task of a specific job was equal to 2.

Then an array of pointers to job records could be set up as

```
struct tag *job;
```

If we want to assign to the variable, `t`, the time of job 8 and `x` and `y` to its location, we could use the statements

```
struct tag *job;
t=job[8]->type;
x=job[8]->x_location;
y=job[8]->y_location;
```

An general-purpose structure called an `Entity` is defined in `SIGMALIB.H`. This is an array of pointers to linked lists of arrays; it is used to model transient entities, queues, etc. Memory for new entities is dynamically allocated as they are created. Memory is freed when entities are removed from lists. The `Entity` structure is defined as follows in C:

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
/* The Entity data type is a double-linked list */
/* S_MAXLIST lists have S_MAXATRIB elements in each entry*/

typedef struct entity{
struct entity
   *prev, /* pointer to previous entity on list */
   *next; /* pointer to next entity on list */
float attrib[S_MAXATRIB]; /* attribute list for this entity */
} Entity;

/* Set up Arrays of Pointers to Linked Lists of attribs */
Entity *head[S_MAXLISTS], *tail[S_MAXLISTS];
```

You do not need to know about the `ENTITY` structure to use SIGMA, but it is useful if you understand it. This structure is a reasonable test of your comprehension of C.

B.14 Structure of SIGMA-Generated C Simulations

The C code generated by SIGMA has a very elementary structure. It also contains extensive comments. In fact, your descriptions of variables, vertices, and edges appear right in the C source code (in upper-case letters). You should look at a SIGMA-generated C simulation (e.g., `CARWASH.C`) while we discuss the different elements of the program.

The program starts with comments that tell the model run defaults that were in effect when the SIGMA event graph was translated to C. This gives you a quick look at the events and variables that are being traced during the run (you can, of course, easily change this in the C code). This is followed by the two `#include` directives that insert the function definitions and variable definitions into your code from two header files, with the code,

```
#include "sigmalib.h"
#include "sigmafns.h"
```

Do not change these files.

The program next lists the event functions and state variables for your model; as in SIGMA, these are global variables. If you want, you can add local variables to the functions for each event or you can create additional global variables, say, for statistics gathering. This is followed by the `main()` function in your program.

The `main()` program initializes the variables and files in your model by calling the `initialize()` function, which appears after the `main()` function. The `initialize()` function looks for an `*.exp` file with the inputs for experiments you wish to run (discussed later). If it does not find such a file, it prompts you for an output file, a random number seed, stopping conditions, and initial values of the attributes for your first vertex. (Values for these attributes were needed in the Run Options dialog box to initialize your run in SIGMA.)

The `main()` function next schedules the beginning and end of the simulation run. The stopping conditions for your C simulation will match the choice you made for stopping your SIGMA event graph that generated this model. You should look at the C code generated from event graphs that have run options that specify stopping after a specific time (Stop on Time) or after a specific number of occurrences of some event (Stop on Event). Next, the start of your run is scheduled and the event execution loop is begun. This loop looks like the following

```
while (!run_error)
  {
  /* Pull next event from event list */
  next_event = c_timing();
  event_count[next_event]++;
  /* Call appropriate event routine */
  switch ( next_event )
      {
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
            case 0:     run_end();
                        break;
            case 1:     RUN();
                        event_trace("RUN",event_count[next_event]);
                        break;
            case 2:     ENTER();
                        event_trace("ENTER",event_count[next_event]);
                        break;
            case 3:     START();
                        event_trace("START",event_count[next_event]);
                        break;
            case 4:     LEAVE();
                        event_trace("LEAVE",event_count[next_event]);
            }
        }
```

The event execution loop is a WHILE loop that runs as long as there is no run error. The run_error flag is set if there is an error during execution; otherwise, this loop will continue until a run error occurs or the run_end() function is executed. The event execution sequence follows the event execution outline given in Appendix A. The sequence is:

1. Find the event number for the next event and advance the simulation clock (called current_time) using the c_timing() library routine in SIGMALIB.C.
2. Increment the event_count for the next event by one.
3. Execute a switch on the next event causing the appropriate event function to be called. If the event is to be traced, the event_trace() function is also executed.
4. The while loop returns to find the next event unless the run_end() function has executed or the run_error flag has been set.

The event_trace() function simply prints out the traced variable values in the same format used by SIGMA. You should be careful not to trace too many *variables* as this might make your output line too long for your compiler. Also, you should be careful not to trace too many *vertices* since this also might make your output file too long. In running a C simulation, most of your execution time typically is used in writing data to the output file in the event_trace() function.

> **Hint**: *Replacing the* event_trace() *function with some data collection variables and arrays will speed up execution of your simulation considerably.*

Elements of the transfer[] array (called ENT[] in SIGMA) are used as temporary storage buffers for the values to be placed on lists. Note that in the SIGMALIB.H file, elements 0,1, and 2 of the transfer array are redefined to equal the event_time, event_type and event_priority respectively. Also, in SIGMALIB.H, the general list filing function, c_file(), is renamed schedule_event(). These name changes in the SIGMALIB.H file are intended to make the SIGMA-generated C simulations more readable; they have no other effect.

The remainder of your simulation is made up of functions for ending the run (run_end()) followed by functions for each of the events in your simulation. For each vertex in your SIGMA graph there is an event function. The event functions have the following structure:

1. The description you gave for the vertex in your event graph model.
2. Definitions of some local variables for testing edge conditions (and any local variables you might want to add).
3. Assignment of values (from the transfer array filled in from the future events list) to the parameters for this vertex.
4. Execution of the vertex state changes.
5. Testing of conditions for all exiting edges.
6. Schedule (or cancel) destination vertices for the edges with conditions that test true by:
   (a) Setting elements of the transfer[] array to the values of attribute expressions for the edge.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

(b) Setting the event type, event time, and execution priority for the destination vertex of this edge.
(c) Scheduling the event onto the list of future events using the `schedule_event()` function.

B.15 Compiling Simulations with Microsoft C

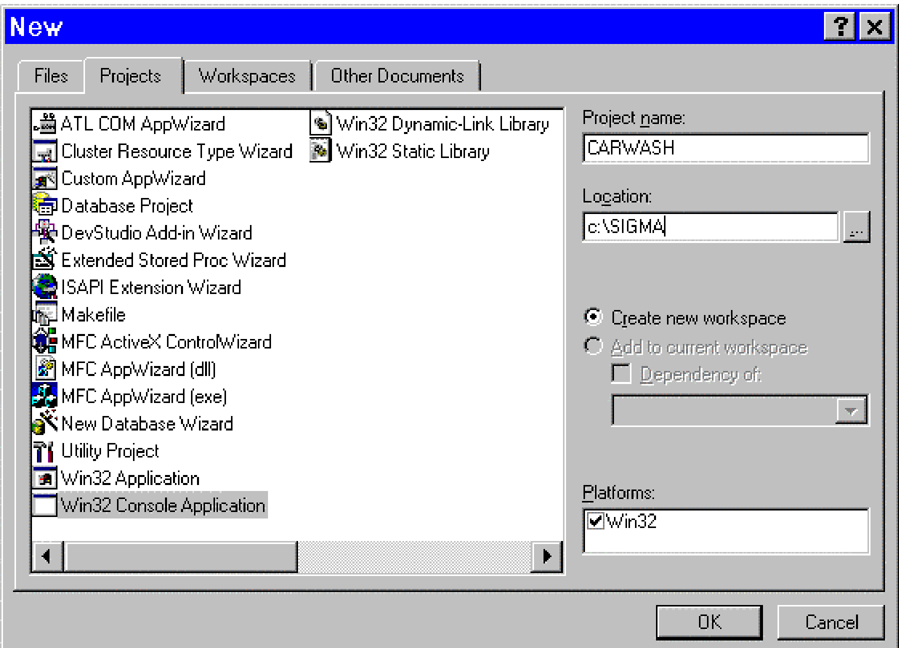
In Section 11.What follows are the details for a simple four-step procedure for compiling and running the automatically-generated C code for your SIGMA simulation. (You can skip these details and go to Section 11.2.3. and use the two-step procedure with the template in the subfolder, `CompileInCTemplate`.) This is followed by two steps for setting up an experimental design file. Your model then will run thousands of times faster, and multiple-run experiments can be batched into a single execution. Only Microsoft VisualC++(6.0) compiler (MSVC++) is used. The model, `CARWASH.MOD`, is our running example. You can then configure your model, design experiments, and run your simulation from a spreadsheet. (See the section on using an EXCEL interface for a SIGMA simulation.)

Microsoft VisualC++(6.0) compiler (MSVC++) is used. The model, `CARWASH.MOD`, is our running example. You can then configure your model, design experiments, and run your simulation from a spreadsheet. (See the section on using an EXCEL interface for a SIGMA simulation.)

NOTES: While SIGMA initializes all variables to zero, C does not. All values not otherwise computed in your model should be explicitly initialized to zero before generating C code. You should also uncheck the trace flags in all event vertex dialogs you do not want to appear in your output.

1) Create a Project: In MSVC++, you first need to build a *Project Workspace* before you can compile your simulation. Even if you have all of your code correct, MSVC++ will have no idea what to do with it until you define a workspace.

To start, open MSVC Developer's Studio and go to the "File" menu and select "New". You should see a dialog box listing the different objects you can start. Select "Project Workspace" and hit the [Enter] key. The following dialog box should appear.



We are going to create a "Console Application" which can be run alone, attached to a spreadsheet, or used to run a series of experiments. Select "Console Application".

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Give your project a name in the “Project name” field (this will be the name of the executable program you will be creating).

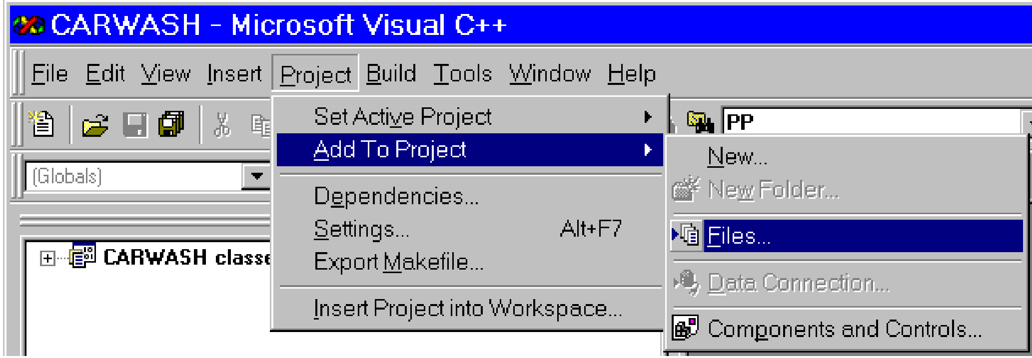
In the “Location” field, pick a directory on the hard drive (NOT your floppy disk, unless you want to die of old age waiting for it to compile). Note that MSVC will create a sub-folder in this directory with your project name. Click [OK].

In the next dialog make sure that "Empty Application" is checked and click [Finish] and confirm with [OK] in the confirmation dialog.

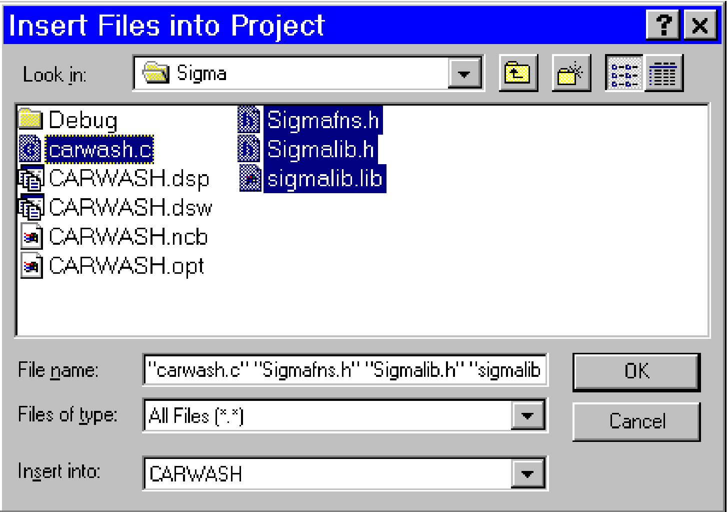
2) Add Files to Your Project: It is a good idea to copy all of the files you need, `SIGMALIB.H`, `SIGMAFNS.H`, and `SIGMALIB.LIB`, as well as the *`FILENAME.C`* that SIGMA has created into your project directory. This avoids the necessity of searching around for file and library paths.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

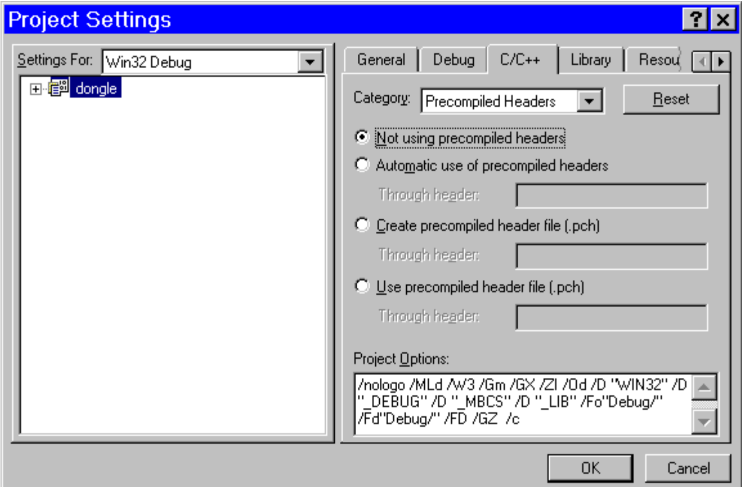
Select the "Project" menu command and select "Add Files". A dialog box like the following will pop up.



Under Files of type, select "All Files (*.*) and then (holding the Ctrl key) select your model (here, CARWASH.C) along with the three files SIGMAFNS.H, SIGMALIB.H, and SIGMALIB.LIB; press [OK].

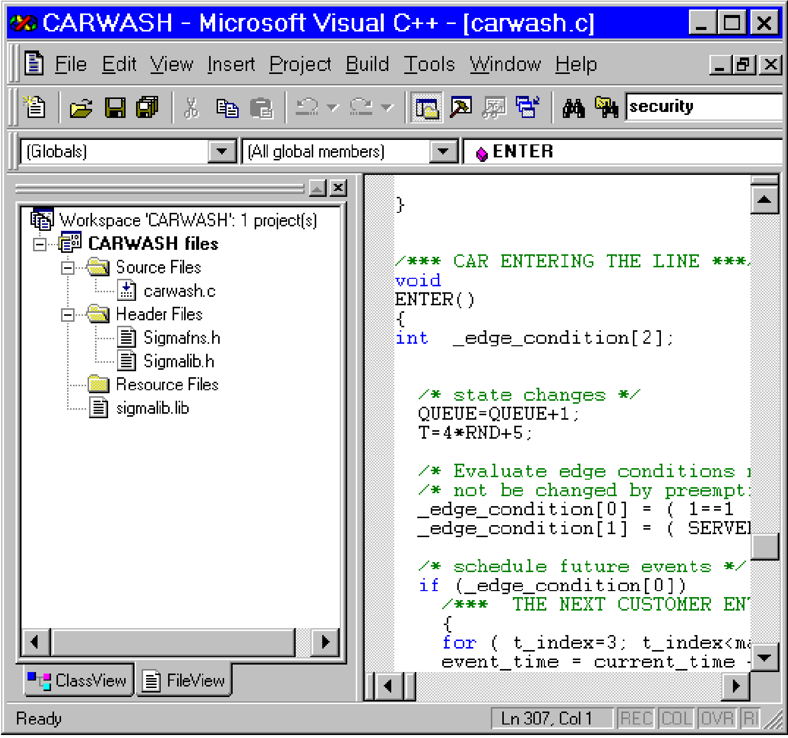


IMPORTANT: You need to turn *off* one of the features in Microsoft Developer Studio - the precompiled headers. Select the Project/Settings menu item and the C/C++ tab. Under Category: select Precompiled Headers and check "Not using precompiled Headers. The dialog should look like the following when you click[OK].

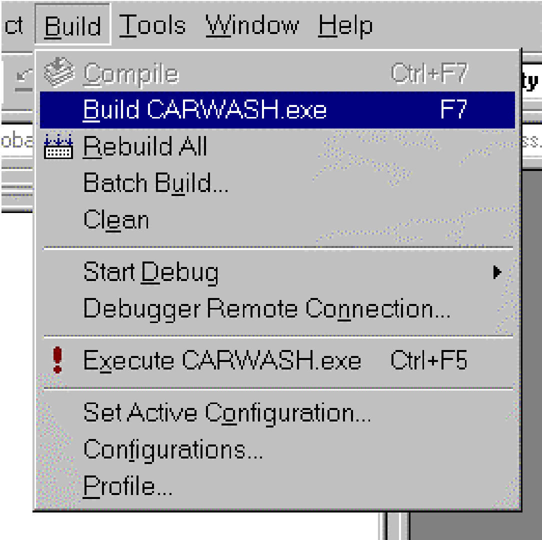


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

There should be a section of the Developer Studio window that looks like the following. Clicking on the "+" will expand the project tree and allow you to look at the different files in the project. Double-clicking on any file will open it up in an editing window so you can view/edit it.



3) Build Your Model: After you have finished all the editing necessary, you can now compile the program. To do this, go to the Build menu and select "Build CARWASH.EXE".



4) Check Program Logic: Did it work with no errors? If so, go to the experiment file material below. No? Don't get discouraged; few compile their programs properly the first time. Check the debug area (probably located at the bottom of the window) for error messages. "Warnings" can be ignored for now—it's the errors that keep the machine from compiling your program…actually, some warnings may be intentional (e.g. casting doubles as long, etc.). Double-clicking on the errors will either take you to the offending code or give you (a little) more information about

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

the cause. Selecting an error message and pressing F1 will bring up the Microsoft help file for the error - don't get too hopeful as these error messages may not be meaningful.

Edit your code as necessary and continue build attempts until it works. Using the debugger is really helpful in this part of the process, so don't be afraid to use it. When MSVC++ reports " 0 error(s)-n warning(s)", you're good to go.

Debugging Function Keys:	
F9:	Insert/delete breakpoint at cursor
F5:	Run to next breakpoint
F10:	Step over a block of code
F11:	Step into a block of code
Shift-F11:	Step out of a block of code

5) Run Your Model: Open the directory you are using, or use Windows Explorer's tools/find feature. You should see a bunch of files as well as your prize: (here: `CARWASH.EXE`). This is the file you can run. If, for instance, you have compiled a SIGMA model, the interface will ask for input and create an output file. This file (say, `CARWASH.OUT`) contains the event list for your run which can be read directly into an Excel spreadsheet for analysis.

6) Create Experiment Files: When you want to make multiple runs of your compiled simulation, it is usually a good idea to write an *experiment file*. This is covered in Section 11.3.

B.16 C Library Functions

The proprietary library for the functions that are needed to run your SIGMA simulations are in a file called `SIGMALIB.LIB`. It is illegal to link this library to a model without a Professional license. The functions in this library are grouped into different sets according to their purposes in the header `SIGMAFNS.H`. The library is compiled using Microsoft Visual C/C++ Version 6.0. Size limits on objects in the library are documented in `SIGMALIB.H`.

B.17 Incompatibilities between C and SIGMA

Power function: The SIGMA power operator `(^)` does not exist in C. (In C, ^ is one of several bitwise operators). Your C compiler probably has a power function in its library, called `pow(x,y)`, which you should use to replace `x^y` in your SIGMA generated C code.

DISK input: The indexed access of the SIGMA `DISK` function to specific items of data is not standard. You will probably want to use the standard C library input functions, scanf() and fscanf() that are discussed in this appendix in Sections B10 and B11 since they are much faster for reading input files sequentially than the SIGMA `DISK` Function. Comment lines and expressions are not evaluated like in SIGMA and must be removed.

CGET function: If you use the `CGET` function in SIGMA, then the translation to C looks like the following for, say, the condition, `(ENT[0]>30&&ENT[2]>=2&&X>=3)`

```
QUEUE=QUEUE-CGET((ENT[0]>30&&ENT[2]>=2&&X>=3),1);
```

To have this work in C you need to make the following 2 changes to your C code for each use of CGET:

1. First change the above line of code in the Sigma-generated C code to

```
QUEUE=QUEUE-CGET(pfCondition1,1);
```

(Here *pfCondition1* is any valid C function name you want.)

2. Then add function(s), returning the condition(s), immediately before main()

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
/* FUNCTIONS FOR CONDITIONAL GETs*/
int    pfCondition1();
int pfCondition1()
{
      return (ENT[0]>30&&ENT[2]>=2&&X>=3);
}

/*    MAIN PROGRAM      */
```

This function, called p*fCondition1()*, simply returns the condition as 1 (true) or 0 (false). This is an exact cut and paste of the condition from the original CGET to the return( ).

B.18 Replacing the Default Random Number Generator with a Multiple Stream Generator

As better, faster, or more flexible pseudo-random number generation codes are developed, they can be easily included in your SIGMA generated simulation engines. This section gives a detailed example for doing this. In this example, we will assume that you wish to use 6 different random number streams in your simulation model. See Chapter 9 for some reasons why you might be interested in using multiple streams in a simulation.

You first need to download the random number generator software you wish to use and add it to your C workspace. To illustrate, we will use RngStream.c and RngStream.h by Pierre L'Ecuyer available at the web site http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/c/. Both these files will need to be added to your C workspace.

Make four simple changes to your SIGMA-generated C code

1. Add the following line after `#include "sigmalib.h"`

```
#include "RngStream.h"
```

2. Define `i` and seed[ ] in `initialize(int argc, const char** argv)` by adding the following two lines after `char y_n[2] = 'p';/* yes/no for file overwrite*/`

```
int i;
unsigned long seed[6];
```

3.. Initialize functions in `RngStream.c` (also in `initialize( )`)

```
  /* PLACE CUSTOMIZED INITIALIZATIONS HERE */
 for (i=0;i<6;i++)
       seed[i]=rndsd;

 for (i=0;i<numstreams; i++){
       //Create stream i
       stream[i] = RngStream_CreateStream("");
       //RngStream_SetPackageSeed (unsigned long seed[6]);
       RngStream_SetPackageSeed(seed);

       //Reinitialize stream i to the beginning of its next substream
       //rngstream_resetnextsubstream(stream[i]);

       //Can generate antithetic variates (1=1-U, 0=U)
       //rngstream_setantithetic(stream[i], 0);

       //Increased precision (0=32 bit, 1=53 bits)
       //rngstream_increasedprecis(stream[i],1);
       }
```

4. In the `/* EVENT FUNCTIONS */` Replace `RND` with `RngStream_RandU01(stream[1])` (or other functions for as antithetic streams, etc.) everywhere in the event codes. For example, if Sigma generates the following line of code

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
    event_time = current_time + 3+5*RND;
```
replace it with
```
    event_time = current_time + 3+5*RngStream_RandU01(stream[1]);
```

Finally, add the following code to the bottom of sigmalib.h, where `numstreams` is the number of different streams you wish to use in your simulation.

```
//Multiple random number streams
#include "RngStream.h"
#define numstreams 6
RngStream stream[numstreams];
```

Now you compile and link these programs to create your simulation engine as before. If you use the compiling template provided for Microsoft Visual C/C++, just double-click on mysigmasimulation.dsw and press F7 to create an executable program of your SIGMA simulation engine. Your simulation engine will again be called MySigmasimulation.exe and is in the debug subfolder. You can now rename it anything you wish and include in a spreadsheet or web site – or double click it to run it.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Appendix C

Overview of Visual Basic for Applications[2]

This appendix will acquaint you with Visual Basic for Applications. Like all flavors of Visual Basic, VBA is large and dynamic. People are creating new and useful VBA objects and codes all the time. You do not "learn" VBA in the conventional sense, you become *acquainted* with it's fundamentals. Then you can use and share coded objects with a vast number of VBA users and programmers. The programs used for the tutorial are carwash.exe and carwash.xls both in the file carwash.zip.

After a quick introduction to VBA showing the basics for creating an Excel User Interface for your Model, the following topics are covered.

1.0 Introduction to VBA

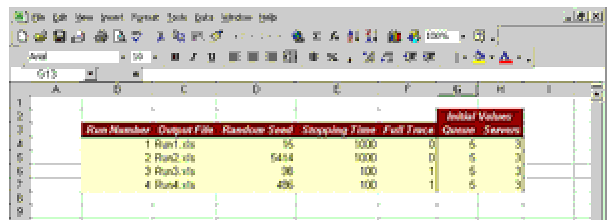
2.0 Event Driven Programming and Excel's Object Model

3.0 Variables

4.0 Procedures and Functions

5.0 Debugging and the Visual Basic Editor

Quick Overview: Creating an Excel User Interface for your Model



This mini tutorial will show you how to create a simple user interface for your model in Excel. I will be using CAEWASH.MOD for the examples, but you can use any model that you would like. You must have a compiled version of your model that functions and this tutorial is not going to cover creating DAT files, so you would be better off using a model that does not require one. To make things easier, we will start off with this Excel file (click for a larger view) that already has an area to input data that would normally go into a EXP file. Our interface will be very basic, users will enter data just like they would in a EXP file, and output will just go into some XLS files. In order to complete this tutorial, you must have CARWASH.EXE in the same directory as CARWASH.XLS and you must have WRITE/MODIFY access to that directory.

Creating the EXP file

Our first step will be to transform the users input in Excel into a EXP file (a plain text file). Lets go ahead an add a code module to our carwash.xls file (if you are not familiar with code modules, you

[2] This appendix was originally written by Arlen Khodadadi while he was a student at Berkeley.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

need to go back and read this). First thing we will do in our new module is declare a procedure name `CreateEXPFile` and declare some variables. Don't worry to much about what the variable declarations mean for now, we are just basically telling the computer we want to store some values that will be changing as our program runs. Enter in the following code:

```
Sub CreateEXPFile()

Dim strPath As String
Dim oFS As Object
Dim oExpFile As Object
Dim strExpFile As String
Dim shtMain As Worksheet
Dim rngEXPData As Range
Dim iRow As Integer

End Sub
```

These variables will be explained as they are used.

The FileSystemObject

In order to create an EXP file, we must be able to create a text file on the computers hard drive. Excel does not have this capabilities built in (well, it does, but its not very good at it). For this reason, we are going to use the ***FileSystemObject***. This object is not part of Excel, but rather it is part of the Windows Scripting Host. Anyway, through the magic of COM, we are able to access the FSO from VBA. The following code will create an instance of the FSO, and allow us to use it to create files later.

```
Set oFS = CreateObject("Scripting.FileSystemObject")
```

Now, in order to create a file, we need to know where to store it. We are going to ask Excel for the path of our current file and use that to create the full path name of our EXP file. Enter the following line into you module:

```
strPath = ThisWorkbook.Path
strExpFile = strPath & "\CARWASH.EXP"
```

All we are doing here is getting the Path property of the ThisWorkbook objects and using that to get a full path for our EXP file. You will notice in the second line that we can append to strings together with the `&` operator. So if **`strPath`** = "C:\Sigma", **`strExpFile`** would be "C:\Sigma\CARWASH.MOD". To create the file itself, we will use this code (the true passed to this method tells the FSO to overwrite the EXP file if it already exists).

```
Set oExpFile = oFS.CreateTextFile(strExpFile, True)
```

We have now asked the FSO to create a text file for us and store a reference to that file in our oExpFile variable. You may have noticed that some of these variable assignments require a `Set` before the variable name while others do not. When assigning a value to a simple variable (such as an integer or string) you do not need to use Set. When dealing with objects, however, you must always use Set or VBA will complain. This has to do with the fact that objects each implement the

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

assignment operator (the = sign) in a different method. Again, this is note something you have to fully grasp, but just be aware of this subtlety.

So now we have a file open and we can start writing to it.

Writing to Our EXP File

Eventhough in our contrived example we know exactly how many runs will be in our EXP file (4), we are going to pretend that the use could have entered as many as he or she wanted. In order to process the user input, we must know how to access it. The following lines do just that; the first line sets our **shtMain** variable to point to the `Sheet1` worksheet object. From now on in our code, if we want to access a property or method of `Sheet1`, we can simply use our variable **shtMain** instead. The second line of code sets our variable **rngEXPData** to a Range object on **shtMain**. A range object is simply a set of cells on a given worksheet. In this instance, our **rngEXPData** variable will reference cell B4 on Sheet1. While it is not necessary to have variables to reference cells like this (we could, for example, simply refer to `Worksheets("Sheet1").Range("B4")` the rest of the program), it is a good habit to get into. This way if we ever decide that we want to rename our sheet, or move the table to cell C6, we will only have to change one or two lines of code instead of hunting down every reference to `Worksheets("Sheet1").Range("B4")`.

```
Set shtMain = Worksheets("Sheet1")
Set rngEXPData = shtMain.Range("B4")
```

We can now actually put values in our EXP file. To achieve this, we will start entering the data that appears on the B4 line (the first line for data entry) and continue on to subsequent lines until we hit a blank line. The code to accomplish this is as follows:

```
iRow = 0
While (rngEXPData.Offset(iRow, 0).Value <> "")
    oExpFile.Write rngEXPData.Offset(iRow, 1) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 2) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 3) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 4) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 5) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 6) & vbNewLine
    iRow = iRow + 1
Wend
```

This code uses a ***While*** statement to check for empty cells. Basically a while statement checks its condition and if it is true, executes the code between the `While` statement and the `Wend`. After going through the code, the while statement tests its condition again and repeats the process until its condition is false (if the condition is false from the beginning, the code in-between`While` and `Wend` will never execute). In this example we are checking to see if `rngEXPData.Offset(iRow, 0).Value <> ""` is true. Now you now **rngEXPData** is a Range object, so you might be wondering what the ***Offset*** method of a range object does. As its name implies, the offset method simply returns a reference to the Range that is offset from itself by a certain number of rows and/or columns. The basic syntax is `RangeObject.Offset(RowOffset, ColOffset)`. For instance, `Range("A1").Offset(1,1)` would point to Range("B2").

Here our main while loop is offsetting**rngEXPData** by iRows and checking if this value is empty (two quotation marks such as `""` indicate an empty value). Everytime we go through the loop

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

(meaning that `rngEXPData.Offset(iRow, 0)` is not blank) we use the `Write` method of **`oExpFile`**. The Write method of a file object just writes its argument to the file. So whenever a line is not blank, we will write 6 columns of that line to our EXP file (we don't care about the run number as that does not go into our EXP file). These 6 columns correspond to columns C through H on our Excel sheet. We could have used another While loop to make the code a little cleaner and more efficient, but this method is easier to understand. Note that we are appending a single space to the end of the first 5 columns we insert into our EXP file. If we did not, VBA would simply write all 6 columns as one big word and our model would not be able to distinguish the different values in our file. Likewise, on the last line of output, we append the vbNewLine constant to our output. VBA defines a plethora of constants to simplify our lives, this particular constant is quite useful and tells the Write method to end the line after we put 6 columns of data on it.

The final line before Wend, `iRow = iRow + 1`, just increments **`iRow`** by one so that the next time the While loop checks its condition, it will be looking at a new line. This way, we keep moving down the page until we run into an empty line. Unlike C/C++, VBA does not have shortcut incremental operators such as iRow++ or iRow += 1.

The Wend line just tells us that our While loop is over.

Now that we are done outputting our file, we can get rid of our FSO object with this code:

```
oExpFile.Close
Set oExpFile = Nothing
Set oFS = Nothing
```

Don't get bogged down in the details, all we are doing here is telling VBA that we are done with the FileSystemObject we were using. As mentioned earlier, the FSO is not part of Excel, so it takes quiet a bit of computer memory for VBA to handle communication with this "out-of-process" component, which means that as long as Excel thinks we will need this object, it will allocate memory for it. So whenever you are done using the FSO or any other object not built into Excel, you should explicitly tell Excel you are done with it by setting the variables equal to nothing. Our CreateEXPFile procedure is now done and it should look something like this:

```
Sub CreateEXPFile()


Dim strPath As String
Dim oFS As Object
Dim oExpFile As Object
Dim strExpFile As String
Dim shtMain As Worksheet
Dim rngEXPData As Range
Dim iRow As Integer


Set oFS = CreateObject("Scripting.FileSystemObject")
strPath = ThisWorkbook.Path
strExpFile = strPath & "\CARWASH.EXP"
Set oExpFile = oFS.CreateTextFile(strExpFile, True)
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
Set shtMain = Worksheets("Sheet1")
Set rngEXPData = shtMain.Range("B4")

iRow = 0
While (rngEXPData.Offset(iRow, 0).Value <> "")
    oExpFile.Write rngEXPData.Offset(iRow, 1) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 2) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 3) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 4) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 5) & " "
    oExpFile.Write rngEXPData.Offset(iRow, 6) & vbNewLine
    iRow = iRow + 1
Wend

oExpFile.Close
Set oExpFile = Nothing
Set oFS = Nothing
End Sub
```

Executing Our Model

Once our EXP file is created, all we have to do is run our model. To do this, we will create another procedure, this time named ExecuteModel. The code for this procedure is much shorter and simpler:

```
Sub ExecuteModel()
Dim strCommand As String
Dim strPath As String
strPath = ThisWorkbook.Path
strCommand = strPath & "\CARWASH.EXE"
Shell strCommand, vbNormalFocus
End Sub
```

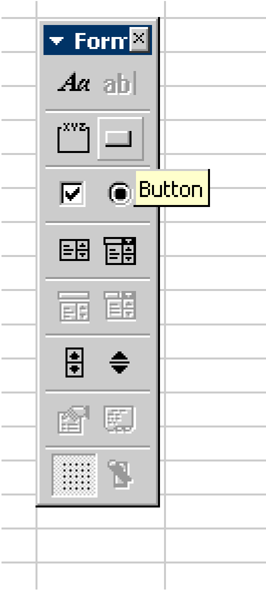
Like in our last module, we are taking our current files path (this is where we assume our model file is) and creating a full path for our compiled code (assumed to be CARWASH.EXE). The shell command is a VBA command to execute a DOS command. You will notice the use of another VBA constant for this command to tell Shell to open up this window to its normal size. The VBE should give you the other options for this argument after you type the comma after **`strCommand`**.

Starting Our Model

Now that most of the code is done, we are going to make one more simple procedure name `StartModel`. This procedure will simply call the other two and the code is as follows:

```
Sub StartModel()
CreateEXPFile
ExecuteModel
End Sub
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



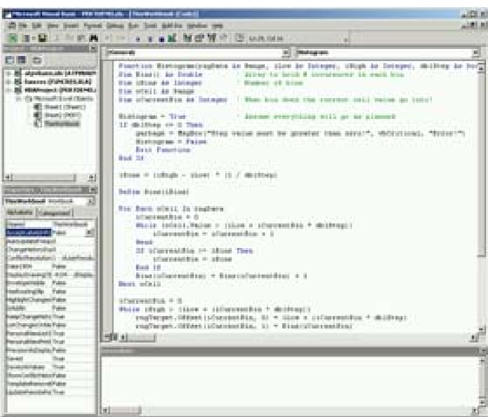
Now we just need a way for users to run this macro without having to go through the `Macros` menu. To do this, we are going to create a button on Sheet1 to run this macro. Open the forms toolbar by going to View -> Toolbars -> Forms and you should see a toolbar like the one to the right. The second to the top button on the right hand side will let you create a Button on your sheet. Click this button and then click and drag to draw a button on your form. As soon as you release your mouse button, Excel should ask you what macro you would like to link this button to. Select StartModel and close up the forms tool bar. Now save this file and you are all done!

1. Introduction to VBA for Excel

1.1 What is VBA?

Visual Basic for Applications (VBA) is Microsoft's common macro language for several different applications (including all Office applications as well as other software like Visio). VBA allows users to access the objects of its host application (in this case Excel) including sheets, charts, cells, etc. VBA (which is the language engine behind Visual Basic) allows users to create full blown applications that run on top of Excel.

1.2 The Visual Basic Editor (VBE)



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

VBA code is entered into Excel through the Visual Basic Editor (VBE). To access the VBE, either press ALT-F11 or go to `Tools -> Macro -> Visual Basic Editor`. As you can see from the adjoining picture (click for a larger view), the VBE is broken down into several areas (four by default); these areas include:

Project Explorer	Here you can explore the different excel objects in your project. These objects will be covered later and include workbooks, worksheets, and modules.
Properties Window	Here you can view and alter the properties of the currently selected object (in this example, the ThisWorkbook object is selected). All objects in Excel have both properties and methods, though we won't cover these items until later.
Code Window	This is the large window on the top right and is where most of the action will take place. Here is were you will input any VBA code you wish your application to run. As you can see, the VBE takes care of color coding your code to simplify programming. In this example, I was currently working on a function to create a histogram for a given set of data.
Immediate Window	You can enter VBA statements directly into this window and receive an "immediate" response. For example, try typing msgbox("This is a test") into this window and press enter

1.3 VBA Modules

You may have noticed that when you opened up your VBE window, the Code Window was blank, or not there at all. This will always be the case when you are working with a file that does not contain any VBA code yet. So, where do you put your code then? That is were ***Modules*** come into play.

Inserting a new module

To insert a new module sheet from the VBE, go to `Insert -> Module`. Your window should now look like this. By default, your new module will be name "Module1" and will go into a newly created folder (named, surprisingly enough, Modules). Typically, modules hold four types of elements:

Event Handlers: Code that reacts to user actions (like clicking on a button or changing a cell's value).

Subroutines: These bits of code perform some set action.

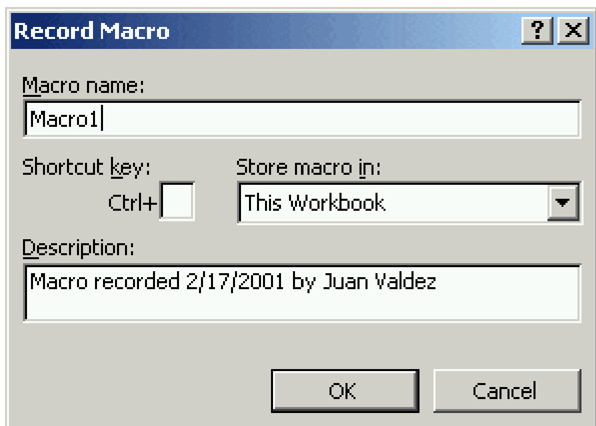
Functions: Like a subroutine, but a function can return a single value.

Declaration: A statement of information you provide to VBA. For instance, you can tell VBA that you will be using a variable name SUM to store an integer value.

These, and other, elements of VBA will be covered later.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

1.4 The Macro Recorder



One of the easiest ways to start learning VBA is to let Excel teach your through its ***Macro Recorder***. To start the macro record, go to `Tools -> Macro -> Record New Macro`. You should get a prompt similar to the one on the right, just go ahead and click `OK`. Now you should be returned to your normal Excel window, with the exception of a new toolbar with a stop button on it. Now, go to sheet1 and follow these steps:

1. Click on cell "B5" and type 1
2. In cells B6 to B9, type the numbers 2 through 5
3. Select cells B5 and B6 and make the font color red by using the font color tool button
4. Select cells B7 - B9 and make the cell fill color yellow with the fill color button
5. Hit the stop button on the new macro toolbar

Your Excel sheet should now look something like this. If you go back to the VBE, you will now see a Modules folder in the Project Explorer. Inside this folder, you will should see Module1, open it by double clicking on it. Your code should be similar to the code below:

```
Sub Macro1()

'

' Macro1 Macro

' Macro recorded 2/17/2001 by Juan Valdez

'



'
    Range("B5").Select
    ActiveCell.FormulaR1C1 = "1"
    Range("B6").Select
    ActiveCell.FormulaR1C1 = "2"
    Range("B7").Select
    ActiveCell.FormulaR1C1 = "3"
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
        Range("B8").Select
        ActiveCell.FormulaR1C1 = "4"
        Range("B9").Select
        ActiveCell.FormulaR1C1 = "5"
        Range("B5:B6").Select
        Selection.Font.ColorIndex = 3
        Range("B7:B9").Select
        With Selection.Interior
            .ColorIndex = 6
            .Pattern = xlSolid
        End With
    End Sub
```

Despite the simplicity of this code, there are quite a few important pieces of information to note.

The first line of the code, `Sub Macro1()`, tells us that we are defining a subprecedure (or simple, a procedure) named Macro1. VBA will allow you to name procedures anything you want, provided your name starts with a letter and is not the same as a VBA reserved word. The parenthesis that follow Macro1 indicate what type and number of arguments this procedure expects. In programs with more than one procedure, there has to be a way that one procedure can pass control to another procedure to allow the second procedure to accomplish its task. This is done by one procedure ***calling*** another. When a procedure (or function for that matter) calls another procedure (or function) it can pass this procedure a set of values. Inside these parenthesis is where you would tell VBA that a procedure (or function) expects to be passed some values when it is called; in our example, our procedure does not expect any parameters to be passed to it.

The next four or five lines of code (which are in green here) are comments. Comments in VBA start with a single apostrophe and continue until the end of the line. Comments do not have to take up a whole line, we can have some code followed by a comment on a single line. We can not, however, have a comment followed by code on the same line as the VBA compiler would not know that our comment ended until it reached the end of the line. Like in any other programming, it is a good idea to document your code through the use of comments.

We are not going to go into much detail about lines of code following the comments just yet. These lines use some of Excel's built in objects to accomplish the changes you just made to sheet1. As you learn more about VBA, you will realize that the Excel macro recorder actually does a terrible job of creating code; these same tasks could be performed much more efficiently in less than half the code. Still, the recorder is a very easy and fast way to find out some of the different properties of objects in Excel without having to use the help files.

Speaking of objects, you should note the method of referencing an object's properties (or methods) from the code above. For example, the line, `ActiveCell.FormulaR1C1 = "1"`, mean set the `FormulaR1C1` property of the `ActiveCell` object equal to 1. Notice how the object (`ActiveCell`) and the property (`FormulaR1C1`) are separated by a dot. This dot notation for accessing properties or member functions (method) is quite common and is used in other programming languages like C++, Java, and javascript. Again, don't worry too much about the specifics of this code (or objects for that matter), we will go into more depth later.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The last line of the code, `End Sub`, simple states that the code for our procedure is done. Every procedure or function you create will need a line like this to tell Excel when to stop executing your code.

Now, just to make sure your macro actually does work and that VBA is not just some conspiracy we concocted to make your life's miserable, go to sheet2 (which should be blank) of your workbook. Run your macro by going to `Tools -> Macro -> Macros` and double clicking on Macro1. After Excel does its thing, your sheet2 should like identical to your sheet1.

And now we are finally done with your introduction to VBA.

2.0 Event Driven Programming and Excel's Object Model

2.1 Event Driven Programming

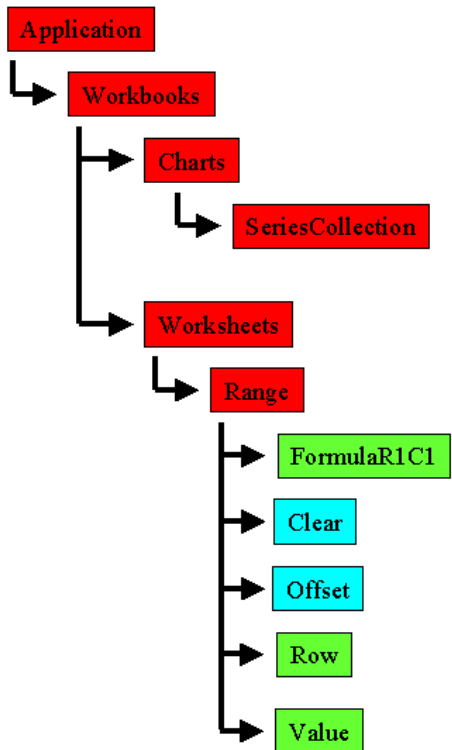
Most older programming languages (like C or Perl) are procedural in nature. That is, a program starts from the first line of code and runs straight to the end of the code (with optional branches in code execution). With procedural programming, the user reacts to the program, in other words, the program might prompt the user for a value and continue executing until it needs some more info from the user.

Event driven programming languages, like VBA, operate in the reverse direction. VBA programs react to certain user performed actions (or events) like changing the value of a cell or printing a worksheet. Code to react to a certain event is placed in the event's Event Handler. Once this event occurs, it is said to be "trapped" by the event handler and any code in the event handler is executed.

2.2 Object Oriented Programming

Now that you have some basic info about event driven programming, you might wonder where these events come from. As it turns out, VBA is an object aware language (not quite fully object oriented, but most of the way there). What this means is that most things your see in Excel are objects. You can think of objects as little "black boxes" that magically do what they are intended to do without you having to worry about how it does its job. For example, you don't have to know how a car works in order to drive one. Objects have properties and methods (or member functions in C++ terminology) associated with them. A property describes something about an object, in the car example your current speed would be a property of the car object. Most properties can be read and altered through your code, but some are read-only (for instance the model of your car is a property of your car, but not something you can change). Methods are ways you can communicate with an object and have it do something. In the car example hitting the gas pedal would be an example of a method. In this case, this method would, among other things, change the current speed property of this object. In a truly object oriented environment, you would have no idea how an object implements its methods, just like most people don't know (or don't care) what happens when the hit their car's gas pedal, all they care about is that once they hit this pedal, their car will speed up (hopefully).

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.



2.3 What Exactly is an Object Model?

So, now that we know about objects, where to we find them? Interestingly enough, most everything in Excel is already an object. Their are objects that represent your current workbook, the worksheet your are on, each cell on a worksheet, the menu bar, there is even an object for that stupid little paperclip that is always bugging you. Objects can even have properties that are objects of their own, forming a hierarchy of objects. Interfacing with these objects (through their properties and methods) is what VBA programming is all about. The figure on the left shows a very (very) small portion of Excel's object model (an object model basically depicts the hierarchy of all the different objects available). Everything in Excel actually stems from the Application object (an object that represents Excel itself). As you can see, a workbook can have worksheet and chart objects as some of its properties (again, this picture does not show all of the properties or methods associated with these objects). Some of the methods (in blue) and properties (in green) of the range object (which describes a set of cells on a worksheet) are shown also.

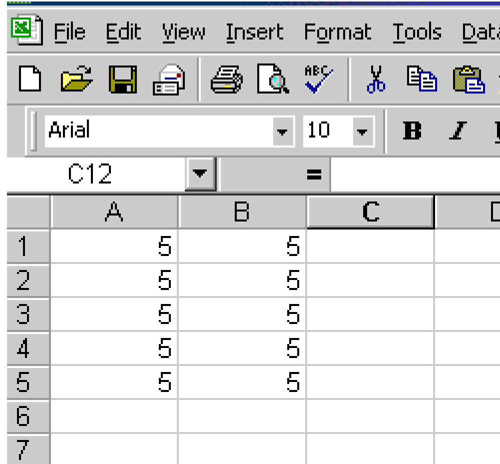
2.4 Accessing Properties and Methods of an Object

If you looked at your macro recorder generated code from the last section, you probably noticed a lot of code like `Range("B7:B9").Select`. This "dot" notation is actually how properties and methods of an object are used. The syntax is `Object.Method` or `Object.Property`, if a method requires some arguments, then the syntax is `Object.Method(arguments)`. To access an object that is a property of another object, simple use this syntax: `ParentObject.Object.Property`. We can use as many dots as necessary to traverse the object model. In this example, `Range("B7:B9").Select`, we are simply calling the `Select` method of the range of cells from B7:B9 (i.e. cells B7, B8, and B9). As is obvious from the name, this method simply selects the corresponding range of cells.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

To experiment, go insert the following code in a module of a new workbook:

```
Sub DoStuff()

Range("A1:B5").Value = "5"

End Sub
```



The first and last line of this code should be familiar to you, all we are doing is declaring a new procedure named DoStuff. The single line of code says take the range of cells from A1 to B5 and change their value property to the number 5. If you run this code on a blank sheet (you can either run your code from the `Tools -> Macro -> Macros` menu as before, or simply click on the run button while your cursor is inside your procedure in the VBE) You should something similar to the screen shot on the right.

Nothing to surprising, I hope. Now that we have seen how to change an objects property, lets try calling a method of an object. Go back to your DoStuff procedure and change your code to:

```
Sub DoStuff()

Range("A1:B5").Clear

End Sub
```

Running this code should effectively undo what your prior code did. Again, there is nothing tricky here, we are simply calling the Clear method of this object to erase whatever values this cells currently contain.

2.5 Collections

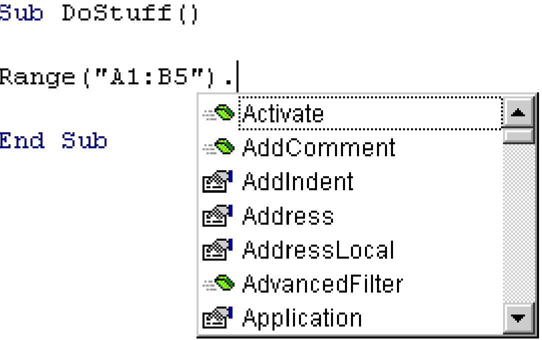
One more important note about accessing objects involves Collections. Collections are container objects, that is, they are a set of usually related data. For instance, the `Worksheets` collection contains all of the worksheets in the open workbook. Items in a collection can be referenced either by name or by index. For example, we can access sheet1 by using the `Worksheets("Sheet1")` syntax or we can access the first sheet in the collection with the `Worksheets(1)` syntax. It should be noted that collections are objects themselves and can contain other collections, objects, or just simple data. In this sense, a collection is VBA's implementation of an object-orinented array (much like the VECTOR object in Standard C++). Since collections are themselves objects, there is a difference between referencing a property of the collection itself, such as `Worksheets.Count`, as

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

opposed to referencing a property of an object in a collection, such as `Worksheets("Sheet2").Name`. In the former example, we are dealing with a collection object whereas the later example is dealing with a Worksheet object.

2.6 Help and VBE's Auto-completion

Describing Excel's object model in its entirety would take a book by itself (and such books do exist). So how do you know what properties and what methods exist for a given object (or even what objects exist)? This is one place where VBA's help and the VBE excel. Use the VBA help whenever you get stuck on anything, it is a great resource that fully details the Excel Object Model. If you press F1 while over a object name (such as Range("A1")), you will get a plethora of information about the Range object (click on the image to the left to see a sample of what VBA's help looks like in Excel 2000).



If you are on a machine with a newer version of Excel that is properly setup (i.e. any computer in the world other than the ones in the IEOR labs), you might have noticed that when you typed Range("A1:B5"), the VBE popped open a list box full of options (like the image to the right). This list box is part of the VBE's auto-complete feature. Whenever you type a "dot" after the name of a known

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

object into the VBE, it will give you a list of all of the objects properties and methods. The with the finger fondling something are properties and the items with what looks like a green brick flying through the air are methods. This way, if you don't know what properties or methods an object supports, you can simply type its name and a dot and the VBE will tell you what is available.

3.0 Variables

3.1 Variables

Just like in algebra, ***Variables*** in VBA are place-holders for data that can change its value during execution of your code. Declaring a variable in VBA tells the computer that you want a certain amount of memory reserved for storing these different values.

3.2 Data Types

The many different types of data one might want to store (like a number, a set of letters, or a date), are referred to as ***Data Types***. When we want to tell the computer that we are planning on using a variable, we declare the variable using the `Dim` command. The following line will create a variable (of Integer type) named **`intExamScore`**:

```
Dim intExamScore As Integer
```

Later on in our code, we can assign a value to this variable with the following command:

```
intExamScore = 5
```

We can now use **`intExamScore`** in other expressions just as if we were using the constant 5. For instance, we can pass **`intExamScore`** as a parameter to a function (or method) that expects an integer argument, or we can assign **`intExamScore`** to another variable (or property) that has an integer type.

Below are a few of the more common data types and their equivalents in C:

Data Type	Description	C Equivalent
Boolean	Holds either a `true` or `false` value	N/A
Date	Holds Dates from 1/1/100 to 12/31/9999	N/A
Integer	Whole numbers between -32,000 and 32,000	int
Long	Long integer, holds from about -2,000,000,000 to 2,000,000,000.	long
Single	Single precision number, can hold fractional numbers	float
Double	Double precision number, can hold much greater range of fractional numbers	double
String	A set of characters (like this sentence)	char[] or *char

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

3.3 Object Variables

This above list only includes simple variable types (i.e., these are not objects). In a more object oriented language (such as Java or C++), even data types are implemented as objects. So for example, if you had a string named **strName** the statement `strName.length` would tell you the length of your string. This is not the case in VBA, but VBA does allow you to create variables to store objects (actually these variables store references to the objects). This way you can create a variable to reference a sheet or a range of values. The following lines declare a variable that references Sheet1 and a variable that references cells A1:B2 of Sheet1:

```
Dim shtMain As Worksheet

Dim rngMyRange As Range

Set shtMain = Worksheets("Sheet1")

Set rngMyRange = Worksheets("Sheet1").Range("A1:B2")
```

The first two lines work just as our previous `Dim` statements, except this time we are using data types that are meant for object references. The third line of code assigns a reference to Sheet1 to the variable **shtMain**. You should notice a few things from this line of code. First of all, we are using the Worksheets collection, specifically, we are looking for the item named Sheet1 in this collection.

This assignment statement is also different from the one before in that we are using the keyword `Set` before the variable name. The reasoning behind this is a little complicated, but it involves the fact that **shtMain** and **rngMyRange** are variables that reference objects. As a VBA programmer, we have no clue how Excel implements the Worksheet or Range object, but rest assured, it is probably a really large and complicated beast. As this is the case with most objects you will deal with, it would be a real drag on performance if VBA had to make a complete copy of the Worksheet object every time you created a variable referencing it. So instead of making a separate copy, VBA simply assigns a "pointer" to this object to your variable (I use quotations around pointer so that you won't confuse this terminology with pointers in C or C++). This is why the variable rngMyRange is said to reference a range object. In reality, rngMyRange is just another name for `Worksheets("Sheet1").Range("A1:B2")`. This means that if we change **rngMyRange**, we are also changing the actual range object it points to. So when we use the line, `Set rngMyRange = Worksheets("Sheet1").Range("A1:B2")`, we are not actually setting **rngMyRange** equal to anything, instead what we are saying is, set **rngMyRange** to be a "pointer" to `Worksheets("Sheet1").Range("A1:B2")`. This is a subtle point, but VBA will give you an error is you try to simply assign a reference to an object without using the `Set` keyword.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

3.4 Naming Conventions

Data Type	Prefix
Boolean	bln
Date	dat
Integer	int
Long	lng
Single	sng
Double	dbl
String	str
Sheet	sht
Range	rng
Other Objects	o

VBA will allow you to name your variables anything you want, provided your name starts with a letter, has no spaces, and that it is not the same as a built in keyword (such as `If` or `Sub`). Over the years, though, a naming convention for VBA (and VB) has become pretty popular. This naming convention prefixes a three letter data type identifier to the beginning of variable names. You might have noticed that the variables we have used so far have had such prefixes (rng for Range variables, int for Integers, etc.). Variables names that are more than one word are usually written with the first letter of each word capitalized such as **`rngMyRange`** (this is the method I prefer) or with an underscore, such as **`rngMy_Range`**. The choice is your, just remember, VBA is not case sensitive, but it will capitalize all occurrences of a variable in the same way. While this convention is by no means required, it will ensure that you are never uncertain about a variables type while you are coding away. Whether or not you use this convention, you should use some convention and stick with it, this way your code will be much easier for you, or anyone else to follow. The table on the right shows some common prefixes used for different types.

3.5 The Variant

Unlike C++ or Java, VBA is not a strictly typed language. This means that you are not required to declare what type a variable will be. You might not even required to declare a variable (this depends on how you have your VBE set up). For example, the following code will work just fine:

```
Sub Test()
  Dim MyVariable
  MyVariable = 5
  MsgBox (MyVariable)
  MyVariable = "Hello"
  MsgBox (MyVariable)
  Set MyVariable = Worksheets("Sheet1")
End Sub
```


©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

The MsgBox() function simply pops up up a message with whatever argument you pass it. You should notice from this code that we never actually declare what type of variable MyVariable is, yet we are able to assign a integer (5) to it, a string ("Hello"), and even a reference to Sheet1. Internally, VBA treats this variable as a ***Variant***. Variants are variables that can take on any data type that you assign to them. This may make variants seem very desirable, but actually you should avoid using them at all in your code. Since VBA has no clue what you will store in a variant, it must allocate enough space for that variable to allow it to store the largest possible data type you might give it. This means that variants have a lot of overhead associated with them and they are very inefficient. Also, VBA can not do any error checking for your since it has no clue what you are planning on using **`MyVariable`** for. So if **`MyVariable`** was holding a string and we passed it to a function expecting an integer, VBA would have no way of warning us of our error until we actually tried to run our program.

Also, since VBA doesn't know what type of variable we are using, the VBE's built in auto-completeion will not work with variants. So in the above code, if we typed `MyVariable.` into the VBE (after the line `Set MyVariable = Worksheets("Sheet1")`), the VBE would not give us a list of a worksheet's properties and methods even though **`MyVariable`** is actually referencing a worksheet.

4.0 Procedures and Functions

4.1 What are Procedures and Functions

Two of the most important concepts in structured programming are ***Code Reuse*** and ***Encapsulation***. For example, lets say we are writing a program to calculate a final grade for students based on their exam and homework grades in a class. Somewhere in our code we are going to ask the user to enter in the homework grades for a specific student. Now, we are going to need to do this for each and every student in the class, but we do not want to write the same piece of code over and over again. What we would like to do is encapsulate this code in a way that we can use the same code over and over again while having it appear in our program only once. We can accomplish this by storing this one piece of code in a ***Procedure*** or a ***Function***.

Assume now that we have a way to enter student score into our program, we need some way of actually calculating a students final grade. Encapsulating this code into a function or procedure (and keeping it separate from the rest of our code) helps us to further breakdown our code into small, manageable pieces. This also also allows us to change this code without having to alter anything else. Lets say that we change our grading policy (like we now wish drop the lowest homework grade for each student), all we would have to do is change our one piece of encapsulated code and the rest of our program would not know or care. This is the same "black-box" concept as with Objects; we are trying to abstract our logic from our implementation. In other words, all we care about is that if we give our grade calculating function a set of homework and exam score, we will get back a final grade.

The difference between procedures and functions is that functions can return a single value. So, in our above example, we would declare the code to calculate a students grade as a function. This would allow us to call the function and store its return value (the students final grade) directly to a variable. On the other hand, we would probably define code to print a list of all the students in the class as a procedure since it really doesn't need to return any value.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

4.1 Declaring Procedures and Functions

We have already seen that the `Sub` keyword is used to define procedures, functions, surprisingly enough, use the `Function` keyword. Since functions return a value, we must declare the function itself as having a return type. Lets say for some reason we wanted a function that will add cells "A1" and "A2" on the current sheet and return that value, such as:

```
Function AddTwoCells() As Integer
Dim rngCell1 As Range
Dim rngCell2 As Range

Set rngCell1 = Range("A1")
Set rngCell2 = Range("A2")

AddTwoCells = rngCell1.Value + rngCell2.Value

End Function
```

As you can see this function, `AddTwoCells`, returns an integer value. There are a few other new things going on here so lets go over the code in detail. The first two lines (after the function declaration) just define two range variables. The next two lines set these variables to the cells A1 and A2. You should note that unlike before, we did not specify what sheet these ranges are on. Excel is smart enough to assume that we are talking about the current sheet. In this case, we want the function to just use the current sheet's values, but in general do not assume that you user is going to be on a certain sheet.
In actuality, Excel has been making such assumptions all along. Remember back when we discussed the Excel Object Model? We said that the top most object is the Application object and everything builds from there. Despite this, we don't ever say:

```
Set rngCell1 =
Application.Workbooks("Book1").Worksheets("Sheet1").Range("A1")
```

Even though this is the actual location of the range object in the Excel Object Model, we are not required to type this out every time we want to access a range. Instead Excel lets us use a couple default objects. For instance, we almost never have to specify the `Application` object, and when we don't specify a workbook object, Excel assumes we want to reference the `ActiveWorkbook` (an object that represents the workbook the user is working on). Like wise, when we don't specify a specific sheet, Excel assumes we wanted to use the `ActiveSheet` object.

Now the second to last line of code,

```
AddTwoCells = rngCell1.Value + rngCell2.Value,
```

displays how functions in VBA return values. In VBA, functions return values by assigning the return value to the function's name. Thus we are treating our function name as a variable and setting that variable equal to the value that we want our function to return. The last line of our code simply ends the function body much like `End Sub` ends a procedure's body.

We can't test our function just yet as we can not run a function by itself. Function have to return a value to something (the code that is calling the function), so we must create a procedure that will simply call this function and do something with its return value. The following procedure

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

does just this and creates a `MsgBox` displaying the value of our function.

```
Sub TestFunction()
Dim intAnswer As Integer

intAnswer = AddTwoCells
MsgBox (intAnswer)

End Sub
```

If we run this code with the values 1 and 3 in cells A1 and A2 of the current sheet, we should get a message box indicating that the value of **`intAnswer`** is 4. We could have also just used one line of code, `MsgBox (AddTwoCells)`, to display the output of `AddTwoCells`. Notice that when we called our function we did not include the parenthesis at the end of the function name. While we could have, it is not required since our function does not take any arguments (yet).

4.2 Arguments

While our last function worked as expected, it was quite useless and isn't exactly the kind of code that you will need to reuse. Chances are you will never need a function that does no more than simply add the values of cells A1 and A2. Lets pretend, however, that you need a function to give you the length of the hypotenuse of a right triangle given the length of the other two sides. Good old Pythagoras says that A^2 + B^2 = C^2 (where C is the length of the hypotenuse and A and B are the lengths of the other two sides). Lets also pretend that such a function might be useful to you (I don't know why, but we are pretending, right?). Now, since A and B are not going to be the same (or even known) every time this function is called, we are going to have to use ***Arguments***. Arguments allow you to make a function (or even a procedure) perform operations on values that are not known during programming time. In other words, a function can be called different times with different arguments and still perform the same operation. In this case, we want our function to solve for C in the above equations, given a value of A and B. So our code will now look like:

```
Function GetHypotenuse(dblSideOne As Double, dblSideTwo As
Double) As Double

GetHypotenuse = Sqr(dblSideOne ^ 2 + dblSideTwo ^ 2)

End Function
```

As you can see, adding arguments to a function is as easy as listing those arguments (along with their types) inside the parenthesis directly after the function's name. Argument declarations look identical to normal variable declarations except that the `Dim` keyword is missing. Here, we defined the two arguments, **`dblSideOne`** and **`dblSideTwo`**, as doubles, since triangle sides can have fractional lengths. When we are calling a function with arguments, we must always make sure that we pass the right number and type of arguments, and that we put the arguments in the same order as what the function is expecting. We also used the `Sqr()` function which gives us the square root of a number.

This code gets the job done, but now lets assume that we want the code to look for values for A and B on cells of our worksheet. We can now change the function to:

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
Function GetHypotenuse2(rngSideOne As Range, rngSideTwo As Range) As
Double
If Not (IsNumeric(rngSideOne.Value) And (rngSideOne.Value > 0)) Then
    MsgBox ("Error, first argument must be a positive number")
    Exit Function
End If
If Not (IsNumeric(rngSideTwo.Value) And (rngSideTwo.Value > 0)) Then
    MsgBox ("Error, second argument must be a positive number")
    Exit Function
End If
GetHypotenuse2 = Sqr((rngSideOne.Value) ^ 2 + (rngSideTwo) ^ 2)
End Function
```

This time, we are expecting arguments that are object references (here, they are references to Range objects). As you can see, using object references for arguments is no different then using simple variables types. The function also introduces some basic error checking and the use of the If statement. If statements work in VBA much the same as the do in other languages, except the syntax is quite different. In VBA, the syntax for an If statement is as follows:

If <condition> Then
<if body>
End If

Expression	Description
>	greater than
<	less than
<>	not equal to
>=	greater than or equal to
<=	less than or equal to
=	equal to
Not	Not condition

If our <condition> is true, then the lines of code in <if body> are executed. Conditional statements are the same as in any other language, and the comparison operators are given in the above table. Take care to note the differences between VBA and C for the "equal to" and "not equal to" operators.
Inside or If condition, we use the built in function IsNumeric() to determine if the contents of the specified ranges are numeric, and if so we check if their values are greater than zero. This should make sense, as we can not solve Pythagorean Theorem's if we do not have numeric (and positive) values. If either of the two arguments fail this test, we indicate as such with a message and then exit the function to ensure that no further action is taken by our function.
As mentioned earlier, the only difference between functions and procedures is that functions can return a value, as such creating a procedure that accepts arguments is identical to what we have already done (with the obvious use of Sub and End Sub as opposed to Function and End Function).

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved.   Written permission required for reproduction, distribution, or quotation.

4.3 Local Variables

Any variable that is declared inside a function or procedure is consider ***local*** to that function or procedures. In other words, this variable can not be accessed by any code outside of the procedure or function it is declared in. As such, local variables can have the same name as variables in other functions (or procedures) without any ambiguity between the variables. Take, for example, the following two procedures that each declare a variable named intX:

```
Sub Procedure1()
Dim intX As Integer
intX = 5
Procedure2
End Sub
Sub Procedure2()
Dim intX As Integer
intX = 8
MsgBox (intX)
End Sub
```

When `Procedure1` calls `Procedure2`, `Procedure2` is not aware of `Procedure1`'s **`intX`**. So the output of running `Procedure1` is a popup message indicating a value of "8".

5.0 Debugging and the Visual Basic Editor

5.1 Debugging Your Code

It would be great if all of our code worked the way we intended it to all the time. In reality, this rarely happens and our code might end up giving us unexpected results. Debugging our code gives us a good way to figure out what is going on. Since VBA is an interpreted language, it provides us with great debugging facilities. Lets assume we have the following code in our project:

```
Sub FillRange()
Dim rngFill As Range
Dim iRows As Integer
Dim iCols As Integer


Set rngFill = Range("A1")
For iRows = 0 To 9
    For iCols = 0 To 9
        rngFill.Offset(iRows,  iCols).Value  =  iCols  +  10  *
(iRows)
    Next
Next

End Sub
```

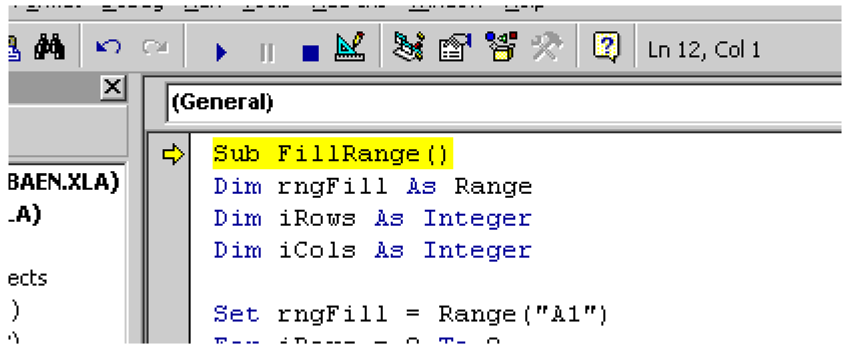
We haven't seen a for loop before, so lets review that first. The basic syntax for a for loop is:

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

```
For <index> = <start> To <end> Step <step>
  <for body>
Next
```

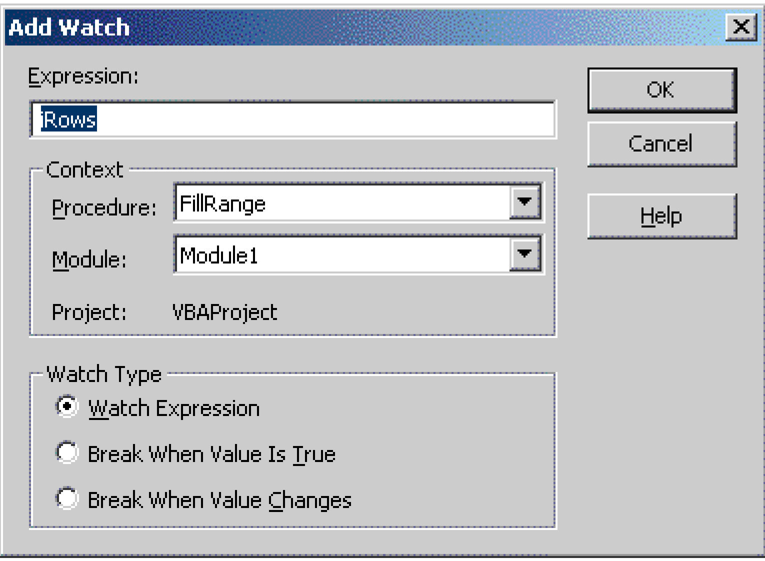
There is nothing too complicated going on. On the first time through the loop the variable `<index>` is set to `<start>` and `<for body>` is executed. The next time through, `<index>` is set incremented by the value of `<step>` and the body is executed again. This process continues until `<index>` equals <end>. In our example, we did not include the Step value (when it is omitted, VBA assumes a step value of one). We also nested one for loop inside of another, which is perfectly legal.

5.2 Stepping Through Your Code



Now you may already be able to tell what this code is going to do, if not, don't worry we are going to use VBA's debugging tools to "step" through the code. Once you have entered this code into the VBE, go to `Debug -> Step Into` (press F8) while your cursor is somewhere in the code. You should see something like the picture on the right. VBA will now begin to execute you code one line at a time, with the yellow highlighting indicating your current line. From now on, every time you press F8 the currently highlighted line will be executed. So press F8 two more times until the line `For iRows = 0 To 9` is selected.

Lets say we are interested in seeing what our for loop does to the variable **`iRows`**. On the current line, select the word iRows and go to `Debug -> Add Watch` or simply right click and select `Add Watch`. You should see something like the following:



©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

Go ahead and press `OK`. Now press F8 one more time and add another watch for **`iCols`**. When we add a variable to our "watch" list, we are telling the VBE that we want it to constantly check and display the value of this variable. You should notice that on the bottom of your VBE there will be a new window named `Watches`. Your VBE should look something like this. As you continue to step through your code with the F8 key, you will see that the Watch window constantly keeps you up to date on the values of **`iRows`** and **`iCols`**. If you just want to see the value of a variable while you are in debug mode, you can simply hold your mouse cursor over the variable name in code and the VBE will display its value.

If you want to see what is going on with your worksheet, simple switch back to excel while you are still in debug mode (i.e. there is still a highlighted line of code). While in this mode, you can easily switch back and forth between Excel and the VBE to see what your program is doing. To exit debug mode, either press F5 (to finish running your code) or press the blue stop button to stop execution of your code.

5.3 The Immediate Window

If, like me, you are too lazy to set up watches for different variables, you can accomplish the same task through the ***Immediate Window***. Lets step into our code again using F8 until we have completed a few loops of our inner for loop. While on of our lines is highlighted, we can go to the Immediate Window and type `?iRows`. You should get a response with the current value of **`iRows`**. Anytime you enter a command into the Immediate Window preceded by a question mark, you will get the return value of that command spit back to you in the Immediate Window. Try entering `TypeName(iRows)` into the window, you should get `Integer` as the response. The function `TypeName()` will return a string representing the type of the argument you give it.

The Immediate window is not limited to giving you values of commands. You can run practically any command in this window, including commands that change variable values. To run regular commands in this window, just type in the commands without a preceding question mark. Try, for instance, `iRows = 20`. Now press F8 a couple more times until you complete enough full loops of the inner loop for control to jump back out to the outer loop, you should notice that your code will not loop anymore as **`iRows`** is now 20, which is above the ending value of your outer loop. Check out your output on your Excel sheet as well, there should now be some values in row 21 of your sheet, indicating that **`iRows`** did indeed change its value.

In the debug menu there are also options for `Stepping Over` and `Stepping Out`. These commands work just like `Step Into` except that they will either step over function and procedure calls (i.e., the VBE will treat a function call as a single statement as opposed to stepping into the function and requiring you to step through each line of the function as well) or will step out of a function or procedure if you are already inside one.

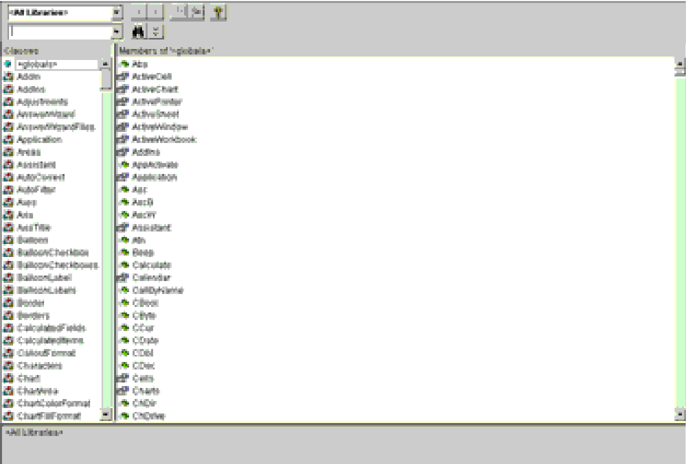
5.4 Breakpoints

What happens if you know that a specific part of your program is giving you trouble? You could use F8 and step through your entire program until you reached that part, but that seems like a pain in the butt. Instead, we can use breakpoints to indicate where we want our code to stop. If you go to a specific line of code and press F9 (or go to `Debug -> Toggle Breakpoint`), that line should turn burgundy and a dot should appear next to it on the left. Pressing F9 will remove the coloring and the dot (and clear the breakpoint). Once you have set a breakpoint at a certain line (you can have as

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.

many breakpoints as you wish), you can run your code as normal and it will automatically stop and put you in debug mode when your code reaches a breakpoint.

5.5 The Object Browser



Earlier we discussed the Excel Object Model and how complicated it was. The VBE provides an easy way to find objects and their properties or methods. When in the VBE press F2 or go to `View -> Object Browser`. You should now see a window like the one on the left (click on image for a larger view). From here you can search for an object by either scrolling up and down on the list to the left or by typing a name in the drop down box on top and pressing enter. If we scroll down on the left until we hit the `Workbook` object, we can see a list on the right of all its properties and methods. From here you can press F1 to get help on anything you need more info on.

©Copyright D.L. Schruben and L.W. Schruben – all rights reserved. Written permission required for reproduction, distribution, or quotation.