DISCRETE SIMULATION:
FUNDAMENTALS
AND
MICROCOMPUTER SUPPORT

Contents

1 Introduction
  1.1 Introduction to Modeling ........................................ 1
  1.2 The Simulation Process ........................................... 5
    1.2.1 Problem Identification ...................................... 5
    1.2.2 Problem Definition ......................................... 5
    1.2.3 Model Formulation .......................................... 6
    1.2.4 Data Acquisition ........................................... 6
    1.2.5 The Computer Model ......................................... 6
    1.2.6 Verification and Validation ................................ 6
    1.2.7 Inference and Analysis .................................... 7
  1.3 The Modeling Language .......................................... 8

2 Discrete Simulation Examples
  2.1 Statistical Variations and Histograms ......................... 10
  2.2 A Simple Monte Carlo Example ................................ 13
  2.3 A Simple Event-Driven Example ................................. 17
  2.4 Final Remarks .................................................. 20
  2.5 Exercises ...................................................... 21

3 Getting Started
  3.1 Overview of the Language Structure ............................ 24
  3.2 The First Program ............................................... 25
    3.2.1 Getting into the Editor .................................... 25
    3.2.2 The Editor ................................................ 27
    3.2.3 Executing a Program ....................................... 25
  3.3 General Language Syntax ......................................... 23
  3.4 Overview of Output ............................................. 31
    3.4.1 Output Control ............................................. 33

Department of Industrial Engineering
Department of Agricultural Engineering
Texas A&M University
College Station, Texas 77843

Copyright 1983
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>Error Trapping</td>
<td>34</td>
</tr>
<tr>
<td>3.4.3</td>
<td>The Help System</td>
<td>34</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Quitting the Session</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Exercises</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>Generation of Random Variates</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Uniform Random Numbers</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Supporting Language Structures</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Discrete Random Variates</td>
<td>42</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Supporting Language Structures</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Continuous Random Variates</td>
<td>49</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Supporting Language Structures</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Statistical Analysis for Input Data</td>
<td>55</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Parameter Estimation</td>
<td>56</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Estimation of Distributions</td>
<td>58</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Goodness-of-Fit Test</td>
<td>59</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Testing for Randomness</td>
<td>67</td>
</tr>
<tr>
<td>4.5</td>
<td>Exercises</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>Fundamentals of Simulation Modeling</td>
<td>72</td>
</tr>
<tr>
<td>5.1</td>
<td>The Definition Phase</td>
<td>73</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Supporting Language Structures</td>
<td>73</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Example</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>The Logic Flow Development Phase</td>
<td>80</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Supporting Language Structures</td>
<td>81</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Examples Continued</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>The Control Phase</td>
<td>88</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Supporting Language Structures</td>
<td>89</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Customized Output</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>Statistics Gathering</td>
<td>93</td>
</tr>
<tr>
<td>5.5</td>
<td>Internal Operating Principles</td>
<td>98</td>
</tr>
<tr>
<td>5.5.1</td>
<td>The RELEASE Block Exception</td>
<td>101</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Language Support: EventChain</td>
<td>101</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Language Support: Interrupt Menu</td>
<td>104</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Language Support: The View Option of Interrupt</td>
<td>104</td>
</tr>
<tr>
<td>5.6</td>
<td>Modeling Ideas</td>
<td>105</td>
</tr>
<tr>
<td>5.7</td>
<td>Exercises</td>
<td>109</td>
</tr>
<tr>
<td>6</td>
<td>Statistical Analysis of Output</td>
<td>115</td>
</tr>
<tr>
<td>6.1</td>
<td>Terminating Simulations</td>
<td>116</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Supporting Language Structures</td>
<td>121</td>
</tr>
<tr>
<td>6.2</td>
<td>Steady-State Simulations</td>
<td>124</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Replicates</td>
<td>129</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Batch Means</td>
<td>131</td>
</tr>
<tr>
<td>6.2.3</td>
<td>The Regenerative Method</td>
<td>135</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Supporting Language Structures</td>
<td>138</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Comparisons</td>
<td>139</td>
</tr>
<tr>
<td>6.3</td>
<td>Exercises</td>
<td>140</td>
</tr>
<tr>
<td>7</td>
<td>Complex Simulation Models</td>
<td>143</td>
</tr>
<tr>
<td>7.1</td>
<td>Preempting Resources</td>
<td>143</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Example</td>
<td>145</td>
</tr>
<tr>
<td>7.2</td>
<td>Combining and Splitting Entities</td>
<td>147</td>
</tr>
<tr>
<td>7.2.1</td>
<td>The COPY and COMBINE Blocks</td>
<td>147</td>
</tr>
<tr>
<td>7.2.2</td>
<td>The BUNDLE and UNBUNDLE Blocks</td>
<td>153</td>
</tr>
<tr>
<td>7.3</td>
<td>Linking and Unlinking Entities</td>
<td>159</td>
</tr>
<tr>
<td>7.4</td>
<td>Advanced Language Features</td>
<td>163</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Advanced Data Structures</td>
<td>163</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Advanced Programming Statements</td>
<td>167</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Example</td>
<td>170</td>
</tr>
<tr>
<td>7.5</td>
<td>Gates</td>
<td>173</td>
</tr>
<tr>
<td>7.5.1</td>
<td>One Lane Traffic Light</td>
<td>174</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Traffic Light with Turn Lane</td>
<td>175</td>
</tr>
<tr>
<td>7.6</td>
<td>Searching Lists for Entities</td>
<td>178</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Example: Strict FIFO</td>
<td>182</td>
</tr>
<tr>
<td>7.7</td>
<td>Creating and Scheduling Entities</td>
<td>184</td>
</tr>
<tr>
<td>7.8</td>
<td>Exercises</td>
<td>186</td>
</tr>
<tr>
<td>8</td>
<td>An Overview of GPSS and SIMAN</td>
<td>193</td>
</tr>
<tr>
<td>8.1</td>
<td>Overview of GPSS</td>
<td>193</td>
</tr>
<tr>
<td>8.1.1</td>
<td>GPSS Language Structure</td>
<td>194</td>
</tr>
<tr>
<td>8.1.2</td>
<td>A GPSS Example and Basic Blocks</td>
<td>195</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Other GPSS Blocks</td>
<td>199</td>
</tr>
<tr>
<td>8.1.4</td>
<td>A Second GPSS Example</td>
<td>201</td>
</tr>
<tr>
<td>8.1.5</td>
<td>GPSS Tables</td>
<td>205</td>
</tr>
<tr>
<td>8.2</td>
<td>Overview of SIMAN</td>
<td>209</td>
</tr>
<tr>
<td>8.2.1</td>
<td>SIMAN Language Structure (Experimental Frame)</td>
<td>210</td>
</tr>
</tbody>
</table>
8.2.2 SIMAN Language Structure (Model Frame) .... 211
8.2.3 SIMAN Output Processor ....................... 213
8.2.4 A SIMAN Example .............................. 213
8.2.5 SIMAN Tables .................................. 219
8.3 Exercises ........................................ 222
8.4 Language Cross References ...................... 222

A Description of the Editor ............................ 227
A.1 Menus ........................................... 227
A.1.1 File Menu ..................................... 227
A.1.2 System Menu .................................... 229
A.1.3 Run Menu ...................................... 229
A.1.4 Options Menu ................................... 230
A.2 Function Keys .................................... 231

B Description of the MOR/DS Language ............ 234
B.1 Nature of MOR/DS Programs ........................ 235
B.2 Basic Language Elements ......................... 237
B.2.1 Symbols .......................................... 237
B.2.2 Comments ....................................... 237
B.2.3 Numerical Values ............................... 237
B.2.4 Strings .......................................... 237
B.2.5 System Constants ............................... 238
B.2.6 Identifiers ...................................... 238
B.2.7 System Variables ................................ 239
B.2.8 Expressions .................................... 240
B.2.9 Functions ....................................... 242
B.2.10 System Functions .............................. 242
B.2.11 Types .......................................... 244
B.2.12 Iterative Constructs ........................... 244
B.2.13 Conditional Expressions ...................... 245
B.3 MOR/DS Program Structure ...................... 245
B.4 The DEFINITION Segment ......................... 246
B.4.1 The ALIAS Feature .............................. 246
B.4.2 Declarations .................................... 247
B.4.3 Real Variables .................................. 249
B.4.4 The ARRAY .................................... 249
B.4.5 The STRING .................................... 250
B.4.6 In-Line Functions .............................. 250

B.4.7 TEXTFILE ...................................... 252
B.4.8 ATTRIBUTES .................................... 252
B.4.9 System Predefined Attributes .................. 253
B.4.10 ENTITY .......................................... 253
B.4.11 LABELS ......................................... 254
B.4.12 The Simulation Constructs ...................... 254

B.5 Control Segment .................................. 258
B.6 Logic Segment .................................... 260
B.6.1 Statement Classification ......................... 260
B.6.2 Simulation Block Statements ..................... 268
B.6.3 Future Events List ............................. 278

C Miscellaneous Tables .................................. 285

Index .................................................. 291
List of Figures

2.1 A histogram obtained from rolling a die 60 times ............ 11
2.2 Dial to be used in building a histogram .................. 12
2.3 Histogram obtained from spinning the dial ............... 14

4.1 Cumulative probability distribution function ............. 43
4.2 Transformation to obtain the random variate $C$ .......... 44
4.3 Transformation to obtain the random variate $M$ .......... 45
4.4 Inverse transformation for the exponentially distributed random variable $T$ ............ 51
4.5 Mapping used to generate the random variable $X$ Value .... 53
4.6 An empirical cumulative probability distribution function .... 59
4.7 An empirical cumulative probability distribution function .... 60
4.8 The determination of the K-S statistic ................... 66

6.1 Histogram from 30 simulations .......................... 119
6.2 Daily costs versus time .................. 126
6.3 Running average of cost versus time .................. 127
6.4 Typical realization showing number of machines in repair room versus time ............ 136

List of Tables

2.1 Experimental results from rolling a die 60 times ............ 11
2.2 Experimental data from 20 spins of the dial .................. 12
2.3 Data for histogram from spinning the dial 20 times .......... 14
2.4 Procedure for determining the interest of a potential customer and the microwave model an interested customer buys .... 5
2.5 Simulated behavior of 20 customers .................. 26
2.6 Procedure for determining the quality of arriving widgets and the machining time for those widgets that are actually processed 18
2.7 Simulated production process for 106 minutes ............ 20

3.1 Summary of editor commands ........................... 36

4.1 Data from a normally distributed random variable ............ 57

6.1 Data from 30 simulations of the one pump service station .... 118
6.2 Individual data points from a simulation run of length 37 days from the machine repair example .................. 134
6.3 Comparisons among the various steady-state confidence interval estimation techniques for the machine repair problem .... 140

8.1 GPSS blocks ........................................ 205
8.2 GPSS system variables ................................ 209
8.3 SIMAN blocks ........................................ 220
8.4 SIMAN system variables ................................ 221
8.5 Statements used to control simulations .................. 223
8.6 Statements used for definitions ........................ 224
8.7 Simulation statement comparison ........................ 225
8.8 Logic statement comparison .......................... 226

A.1 Summary of editor commands ........................... 233
LIST OF TABLES

B.1 System variables .......................................... 239
B.2 MOR/DS DEFINITION structures and parameter syntax .. 255
B.3 MOR/DS CONTROL parameters ............................ 258
B.4 MOR/DS LOGIC modeling blocks and parameter syntax .. 279

C.1 Normal distribution ......................................... 286
C.2 Student's t-distribution ...................................... 287
C.3 Chi-square distribution ...................................... 288
C.4 Critical values for the Kolmogorov-Smirnov test ........ 289
C.5 ASCII characters ............................................. 290

Preface

This text is intended for a one-semester introductory course in computer simulation at the undergraduate or first year graduate level. We assume that the reader has a basic knowledge of statistics and computer programming, as well as some familiarity with microcomputers.

Simulation is a way to study random processes. Learning how to model with discrete simulation is a demanding challenge because the subject is complex and both the modeling process and simulation output analysis require a reasonable working knowledge of probability, statistics, and programming. Experience has shown that these topics are difficult to master, and, unfortunately, most students who take simulation are uncomfortable with these topics. To circumvent these problems, this textbook proceeds in a careful manner through the concepts of modeling random processes, performing statistical analysis, and relating these topics to discrete simulation. The MOR/DS simulation language and user environment were developed specifically to simplify the interface between the student and the microcomputer by providing an easy to use system. The language itself was developed to provide modeling power without sacrificing ease of use. Right from the start, the student will use the computer to learn about statistics, and later statistics will be used as a tool in simulation analysis. Computer-related problems are addressed before the more difficult issues of simulation programming are encountered.

A word about MOR/DS. This name results from discrete simulation (DS) being one part of a complete operations research and management science software system developed by the authors. MOR signifies microcomputer support for operations research and management science; the two letters at the end of the name indicate which of the components is active (the choices are DS for discrete simulation, LP for linear programming, NL for nonlinear programming, and DP for dynamic programming). All systems operate through the same text editor which offers pull-down menus, complete editor functions, and immediate access to DOS commands that are not provided by the menus. On-line help and context-sensitive error messages are also pro-
vided. Students can learn this system very quickly and focus their attention on modeling, without concern for the operating system.

The MOR/DS system comes on a single diskette suitable for an IBM personal computer or other fully compatible microcomputer. MS-DOS 2.1 or greater and 640K random access memory are needed. Included on the disk is a file named README.DOC containing update information since the textbook went to press. This file can be loaded into the editor for reading.

Chapter 1 introduces the basic concepts of modeling and discusses the general simulation process. Chapter 2 provides a collection of examples to motivate the student’s study of random processes and to illustrate what simulation offers as a means for analyzing probabilistic real-world problems. Chapter 3 introduces the MOR/DS programming environment and gives an overview of its simulation programming language. Several examples are provided that use MOR/DS as the mechanism for carrying out statistical experiments that offer hands-on experience. Since the purpose of the book is to teach the concepts of simulation and to use the language only as a reinforcing agent, this chapter is small and can be covered quickly. Only the information needed for using the language as a learning aid is covered in Chapter 3. Chapter 4 presents methods for generating random variates, as well as a review of the common statistical tools needed in simulation analysis.

Chapter 5 presents the basic concepts and tools for simulating simple systems. Also presented are several simple examples and their MOR/DS models. This chapter provides the real beginnings of simulation modeling by discussing discrete block-oriented techniques. Chapter 6 provides a comprehensive introductory treatment of the methods and issues involved in the statistical analysis of a simulation model’s output, and Chapter 7 is intended to provide an in-depth study of simulation modeling.

The authors believe that simulation modeling requires a rather extensive “bag of tricks,” and these tricks amount to solutions to particular modeling problems. MOR/DS affords capabilities common to a number of well-known simulation languages as well as other features not used in other languages. The syntax of the language was designed to simplify the use of these capabilities. Therefore, we present several example problems and ask the student to develop his or her own solutions to additional problems as exercises to form the “bag of tricks,” thus providing each student with the capability of applying simulation modeling to situations outside the classroom. While a number of language-dependent factors do creep into the tricks, we believe that once this process is understood, most any language can be used (some more easily than others). This is why two chapters (5 and 7) are devoted to modeling in MOR/DS. Anyone who diligently works through the examples and exercises will be well prepared to attack new problems using MOR/DS or any other language. Chapter 8 addresses just this issue by briefly considering GPSS and SIMAN, two of the most popular commercial simulation languages. The appendices provide the formal specifications of the editor and the MOR/DS language, as well as some useful statistical tables.
Acknowledgements

We are indebted to many of our colleagues for their invaluable assistance and professional support. Although space prohibits listing all those who have helped, we would like to explicitly express thanks and appreciation to Kam Bennett (Industrial Engineering, Texas A&M University), Bart Childs (Computer Science, Texas A&M University), John Fowler (Industrial Engineering, Texas A&M University), Lee Lowery (Civil Engineering, Texas A&M University), Paul Rubin (Michigan State University) and Bob Shannon (Industrial Engineering, Texas A&M University) for using preliminary versions of this textbook and software. They made many suggestions for improving both the text and the MOR/DS system. Bart Childs also provided support and assistance in using \LaTeX\ to typeset this textbook. We also wish to thank our wives and families for the patience and loving support they gave us throughout the duration of this exciting project. Finally, we acknowledge our thanks through the words of the psalmist, "O give thanks to the Lord, for He is good; for His steadfast love endures for ever!" (Psalms 107:1, RSV)

Chapter 1

Introduction

This is a book about modeling with discrete simulation. Most readers of this text have encountered the subject of modeling in courses on statistics, operations research and management science, or computer programming. In most of these experiences, the problem (system) being modeled was highly structured, and the techniques used were narrowly focused (e.g., linear programming). Textbook problems that you solved usually provided clearly defined objectives for the model such as to maximize profit. However, the experiences you will have as you progress through this text will broaden both your concept of modeling and your awareness of the myriad issues that can arise during a modeling effort. Put another way, simulation modeling is less structured than most other types of modeling, and the objectives are less clearly defined. We begin with a brief discussion of the topic of modeling in general to establish the need and viability of modeling. Then we describe simulation and how it fits into the general scheme of modeling.

1.1 Introduction to Modeling

Before we can discuss modeling, we need to address the subject of a system, since a model is a stand-in or substitute for a system. The system is the real object of our inquiry. Unfortunately, it is very difficult to give a precise, formal definition to the term "system" without making reference to terms equally difficult to define. A system is usually something physical, it has many parts, and these parts interact. Examples of a system are a building project, a manufacturing process, an airport, an inventory and distribution component of a retailing system, and a traffic system. Most
systems of interest are complicated to study because of the interaction of the parts that constitute the system, the overall size of the system, the effects of uncertainty (randomness), and the difficulty of isolating the system to be studied from its encompassing surroundings.

Systems studies are needed when designing new systems, modifying existing systems, and attempting to improve the operation of systems. It should be obvious that we can do a better job in any of these applications of systems design, modification, or control if we have a good understanding of how the system operates and how it responds to changes in the parameters that define the system. Unfortunately, it is not possible or even wise to study or experiment with the system itself. The costs or consequences of such experimentation may be prohibitive, or the system may not exist at the time of the analysis.

Consider the system consisting of an automobile assembly line. We are interested in changing the suppliers of the sheet metal components that are assembled in the plant. Clearly, the quality of the incoming parts, the delivery rates (or when the parts enter the plant), and other aspects of the supply process will have a marked impact on the operation of the assembly plant. How can we evaluate these impacts? We certainly cannot try out different suppliers and experiment with different operation rules in the plant for each supplier. Modeling offers an alternative to experimenting with actual physical systems.

The model bridges the gap between the system and analysis; it is a stand-in or surrogate for the real thing. A well-conceived and properly tested model provides the test bed for quality systems analysis. However, we must stress that when we are using a model to study a system we are at best making inferences concerning the real system. We must be cautious about what we conclude, and that the quality of these conclusions is in direct proportion to the quality of the model used in the analysis.

The objective is to formulate a model that incorporates sufficient reality and detail so that inferences made relative to the model can be extrapolated back to the system. It is equally important, however, to avoid unnecessary detail that causes the model to be too cumbersome and unwieldy for analysis.

For our purposes, a system can be viewed as a function, or transformation, that converts given inputs into specified outputs. In most circumstances, the transformation process is dynamic; that is, the process evolves over time. Except for very simple cases, the transformation process is complex and difficult to capture and explain.

1.1. INTRODUCTION TO MODELING

Differing purposes for systems give rise to two kinds of models, called descriptive and prescriptive models. A descriptive model describes the specific transformation process that converts the inputs into the outputs. A prescriptive model goes further to describe a policy or specific set of operating rules to use for a given set of inputs to meet some predefined goal. As an example, consider a standard linear program for determining which products to produce in what quantities so as to maximize total profit, subject to resource availability constraints. The linear programming problem is both prescriptive and descriptive. By specifying the optimal production quantities, it is prescriptive, and by specifying the profit function and constraint functions, it is descriptive, because the profit value and resource consumptions can all be determined given a value for each product (i.e., activity level). The constraints coefficients provide rates of resource consumption so we can clearly see the impact of each product on resources. Computations must be made to measure these quantities, and even more computations must be carried out to find the optimal policy.

Most modeling activities lead to building descriptive models. Subsequent use of these models can lead to the development of prescriptive models or to the use of the descriptive model for prescriptive purposes. A descriptive model serves as the experimental surrogate for the actual system.

Descriptive mathematical models can be constructed or developed using a variety of techniques and approaches, two of which are most pertinent to this book. The first approach (called direct modeling) focuses directly on methods for computing the quantities of interest; that is, the outputs are modeled as functions of the inputs, controls, and mechanisms. This approach usually requires sophisticated mathematical methods for both formulation and computations. For example, the model may consist of systems of equations (linear and/or nonlinear), differential equations, etc. When such a model is used, a great deal of computation may be required to determine the quantities of interest even though the model was built using analytical methods.

The second fundamental approach for descriptive modeling is called simulation, an indirect or experimental method. Simulation captures the operational characteristics of the system and derives estimates of the quantities of interest through a process of statistical experimentation. The model is logical rather than mathematical, and it is designed to simulate the actual operation of the system over time. The desired quantities are determined through statistical sampling and estimation. Whereas direct models use mathematical constructs to show how desired quantities are computed, the
indirect or simulation model describes the logical behavior of the system over time and how to measure observations of the desired quantities.

Both direct and simulation approaches require extensive computations, but the natures of the computations are radically different. Simulation is, by its very nature, empirical and hence requires multiple repetitions to create meaningful and accurate estimates. Direct models must make use of often-times sophisticated numerical methods such as equation solvers and numerical integrators. Simulation analysts employ special computer languages and systems to simplify the modeling and computational problems. Direct methods more frequently require special purpose computer programs to be developed as part of the modeling effort. Simulation languages are specially designed for simulation but are relatively general in nature to cover a wide range of simulation models. Consequently, simulation programs are slow and require substantial amounts of computer time. Specially crafted programs to support direct models are usually fast and efficient, but require substantial time and expertise to develop.

Direct models have the advantage that they compute the desired quantities directly rather than through experimentation, as do simulation models. Their disadvantage lies in the level of mathematical sophistication required for modeling and computation. Loss of reality and loss of flexibility. Indeed, even minor changes in the assumptions can completely render a direct model invalid. The logical nature of a simulation model makes it possible to model complex systems relatively easily. Such models are robust and easy to modify. Principal disadvantages stem from the model's empirical and statistical nature. Much computer time is required to obtain accurate results, and a knowledge of the statistical aspects of simulation is mandatory for a proper model design and for proper interpretation of the results.

While direct methods and simulation are at opposite ends of the descriptive modeling spectrum, they can and should share a meaningful coexistence. Quite frequently, special purpose direct models can be constructed as a consequence of insights gained through experimentation with the simulation. These models are more efficient and faster than the entire simulation and consequently are better able to serve as the basis for prescriptive models. The simulation model, however, is always at hand to verify solutions by providing a closer depiction of reality. The two should work together.

Most readers of this book have had some exposure to the basic methods of operations research and management science, as well as experience in formulating direct mathematical models from "word" problems. This book will expose you to the broader and less structured world of modeling and analysis. You will learn how to express the logic of the system in the discrete simulation language MCR/DS. You will encounter a variety of statistical tools needed to provide the data for the model, to verify the model, and to interpret the results.

1.2 The Simulation Process

The model building process can be broken down into several interrelated steps.

1.2.1 Problem Identification

Although it may seem somewhat redundant to point out explicitly the need to identify the problem, this step is quite important. Imbedded in the problem identification is the motivation and purpose for the study. Usually there are many problematic issues that merit study, and decisions must be made as to which are the most important. Factors such as how critical the problem is, anticipated cost savings, complexity of the system, and qualifications of the staff help to narrow the choices.

1.2.2 Problem Definition

The problem definition step formalizes the scope of the problem and the purpose of the study/model. The scope determines the boundaries of what is to be included in or excluded from the system to be modeled. The purpose of the study determines the questions to be answered as well as the extent of detail necessary to investigate properly different alternatives and their consequences. As mentioned before, there is a trade-off between detail (model size and complexity) and tractability. A common mistake is initially to attempt to put enough detail in the model so that it can be used for multiple purposes. For example, high-level aggregate planning models need not be too detailed because broad policy issues are studied, whereas low-level operational planning requires much detail. The same model cannot adequately and efficiently serve both needs. However, careful planning during problem definition can lead to a structured model that can be extended later in the analysis. Too much detail should be avoided early in the study.
1.2.3 Model Formulation

Once the modeler decides on the scope and purpose of the problem, he or she can build the model. A simulation model has three components: data and simulation structures, simulation control information, and process logic. The measures of effectiveness and other variables of interest are in the first category, as are model parameters. Any parameters for which specific values are unknown must be identified. At this point, a prototype computer program might be developed to determine the feasibility of the general approach.

1.2.4 Data Acquisition

Data must be collected to estimate, in a statistical sense, any unknown parameters found in the problem identification and model formulation steps. Proper design of the statistical procedures is necessary to insure maximum utilization of the data. If possible, it is desirable to obtain accuracy estimations for the parameters so that sensitivity studies may be done later in the analysis process.

1.2.5 The Computer Model

The major part of the modeling process involves expressing the logic that describes the functionality of the system. Typically, the full logic design is drawn from the prototype described above through the use of a simulation language such as FORTRAN or SIMSCRIPT.

1.2.6 Verification and Validation

Two issues must be addressed before the model can be used safely. First, the simulation program must be verified to ensure that it correctly does what was meant to do. This verification focuses on the computations and logic in the simulation. Second, the statistics of the simulation program must be compared with those of the system to make sure the model adequately mimics the operation of the system. Since there is considerable room for error in a large study, verification and validation can be time consuming and difficult. However, both steps must be done well because the ultimate goal is to make inferences about the system based upon inferences about the model. If the model is not reliable or if it does not accurately represent the system, inferences drawn from it will be misleading and meaningless.

1.2.7 Inference and Analysis

Once the simulation model is deemed to be correct, both in terms of its internal operation and the system it represents, it can be used to study the behavior of the actual system. Reasons for this study vary, but most center on characterizing the behavior of the system as input parameters are changed or as structural modifications are made. As an example of the first case, suppose the model describes the checkout counters (cashiers) at a supermarket. Input parameters may include the number of cashiers to use, the length of time required to process customers, and characteristics of the customers (number in the store, length of time spent in the store, quantity of items to purchase, etc.). The study would typically involve determining the consequences that result from different combinations of parameter values. We might change the rate of service (which could result from training personnel or using different equipment) or the probability law governing the service process (which could result from modifying equipment). We might want to observe the length of the queues at each counter, the waiting times of the customers, the utilization of the cashiers, the cost of the system, and perhaps much more. Managers or designers can make better decisions if the consequences of their decisions are measurable and understood.

After such a study is complete, the simulation model is ready to serve as the experimental device for measuring the characteristics of the system decided upon by management. A great deal can be inferred about the system by exercising the simulation model. Two very important issues govern the analysis phase of the effort. First, how do we decide which parameter values to try (i.e., which simulation runs do we make)? Second, how do we measure and analyze each of the simulation runs? Because a simulation model tracks the performance of the system over time, a number of issues come up concerning run length, stabilization of the system, and proper statistical analysis of the results.

The decision as to which runs to make is important because this process can be time consuming and expensive. For example, suppose two parameters are going to be varied; parameter 1 has three possible values and parameter 2 has ten possible values. Further, let us assume each simulation run takes 20 minutes. To consider all possible combinations of runs would require $3 \times 10 = 30$ runs and 600 minutes, which would yield a great deal of output to analyze. Judicious choices must be made to minimize the number of values for each parameter and, for that matter, the number of parameters to
vary. An important branch of statistics called experimental design provides
a systematic approach to dealing with these types of decisions.

In summary, a number of interesting and difficult questions arise when
one actually designs a model, runs the simulation, and interprets the results.
This textbook provides the necessary information to enable the student to
answer some of these questions.

1.3 The Modeling Language

Fundamental to simulation modeling are general purpose discrete simu-
lation languages. To support the teaching of simulation, we have developed
an event-driven simulation programming language called MOR/DS. Our lan-
guage is easy to use and provides sufficient power and flexibility to support
the variety of topics that can be covered in a first course in simulation.
These include elementary modeling, sophisticated modeling, statistical anal-
ysis, and various combinations of these topics.

MOR/DS is a self-contained modeling and analysis environment. The
heart of the system is a full-featured text editor that provides not only the
facilities for building and modifying models, but also the capabilities for
running simulation models. Commands (editing, file manipulation, and sim-
ulation) are accessed with a unified collection of function keys and pulldown
menus. The integration of the editor, compiler, and run-time components
into a single system facilitates modeling, debugging, and analysis. The mod-
eler can build a model, debug it, and execute it without ever having to access
operating system commands or even leave the editor. Beginning students are
not intimidated or confused by this environment and quickly get over the
fear of dealing with a new, unknown system. More advanced users appreciate
the unified framework because they are free to concentrate on modeling
and analysis instead of system level concerns.

The MOR/DS language provides both simulation block commands and
an almost complete Pascal-like programming structure. In this manner,
routine simulation tasks are readily carried out with “blocks,” while more
algorithmic nonsimulation tasks are directly and simply performed with pro-
gramming statements. Sufficient programming capabilities are available to
support the creative desires of the modeler. The language uses named pa-
rameters for simulation blocks; thus, the meaning of a parameter is clear
because the name is part of the command.

The compiler and run-time components of MOR/DS perform all of the
tasks necessary to execute a model. Because the system is integrated, com-
piler errors leave the cursor in the vicinity of the error; this, along with the
error messages reported by the compiler, greatly simplifies error diagnosis
and correction. The run-time system provides both a real-time animation
feature and a user interrupt capability. Animation makes it possible to mon-
itor the dynamic behavior of the system being modeled; several of its capa-
bilities are under user control. The interrupt system displays a menu with a
wide range of commands including quit, print results, and view entity lists,
among others. Together, these animation and interrupt features enhance the
learning process as well as facilitate model validation and interpretation.
Chapter 2

Discrete Simulation Examples

It is often necessary to make decisions about complex processes that would benefit from quantitative analyses. However, many processes contain enough complexities so that rigorous mathematical evaluation is impossible or, at least, very difficult. This problem can often be solved with properly designed simulations and their careful analyses, which can provide the quantitative guidelines sought by decision makers. This chapter presents several simple examples intended to provide an understanding of the power and usefulness of simulation and the basic operations that must be performed to simulate a process. Since understanding the concept of randomness is so essential for understanding simulation, this chapter begins with a discussion of randomness. The remainder of the chapter introduces the mechanical details of simulation and some of the major conceptual issues that must be answered in performing a well-designed simulation study of a system.

2.1 Statistical Variations and Histograms

To reinforce the idea of randomness and the statistical variation of experiments, the reader should consider the results of taking a die and rolling it 60 times. Such an experiment was done with the results (i.e., the number of dots on the upward face) being recorded in Table 2.1. Assuming the die was fair (i.e., each of the six sides is equally likely to show), we would expect each of the six faces to occur 10 times; however, that did not happen in the experiment. Although the list of numbers in Table 2.1 describes the results of the experiment, it is often convenient to present a visual representation of the experimental results. For this purpose, a graph similar to a bar chart is used. In such a graph, there is a bar above each number ranging from one to six, where the height of the bar represents the number of occurrences as observed from the experiment. Such a chart, shown in Figure 2.1, is called a histogram.

<table>
<thead>
<tr>
<th>Dots on up face</th>
<th>one</th>
<th>two</th>
<th>three</th>
<th>four</th>
<th>five</th>
<th>six</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of occurrences</td>
<td>7</td>
<td>11</td>
<td>9</td>
<td>14</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 2.1: A histogram obtained from rolling a die 60 times

Histograms serve two functions. They provide a quick visual representation of the experimental results, and they represent the probability law governing the experiment when the probabilities of the various outcomes are


2.2. A Simple Monte Carlo Example

To illustrate histograms further, we will perform one more experiment. Consider a free spinning pointer on a dial that is marked zero to one as in Figure 2.2 (actually, the dial in the figure represents the interval $[0, \pi]$ that includes zero but not one). The experiment is to (randomly) spin the dial 20 times and record the number to which the arrow points each time (assuming the circumference is continuously labelled). Table 2.2 contains the results of this experiment. To draw a histogram from this experiment, the interval $[0,1]$ is divided into subintervals or "classes." As an example, we divide the interval $[0,1]$ into four equal classes that yield Table 2.3 and finally the histogram in Figure 2.3. If we had the perseverance, we could redo the "spinning pointer" experiment and perform it 1000 times. Because more data points are available, the interval $[0,1]$ can be divided into 100 equal subintervals instead of only 4. Since the spinning of the pointer was (supposedly) completely random, we expect each subinterval to have approximately the same number of occurrences (except for the usual statistical variation). The numbers that are generated by this experiment are called uniform random variates between zero and one. They are called uniform because equal intervals have equal probabilities of occurring.

We conclude this chapter with two examples to further illustrate the basic concepts of simulation, again using physical methods for creating the random variates. We will take up the question of how to generate random numbers with a computer in Chapter 4.

Table 2.2: Experimental data from 20 spins of the dial

| 0.10 | 0.22 | 0.24 | 0.42 | 0.37 |
| 0.06 | 0.72 | 0.91 | 0.14 | 0.36 |
| 0.88 | 0.48 | 0.52 | 0.87 | 0.71 |
| 0.51 | 0.60 | 0.94 | 0.58 | 0.99 |

Figure 2.2: Dial to be used in building a histogram

---

model, and 25 percent purchase the deluxe model. The plain model yields a profit of $30, the standard model yields a profit of $60, and the deluxe model yields a profit of $75.

The salesperson wishes to determine the average profit per customer. We assume that the salesperson does not have a mathematical background and therefore is unable to calculate the exact expected profit per customer. Hence, an alternative to a mathematical computation for estimating the profit must be used.) One approach for estimating the average profit would be to keep records of all the customers who talk to the salesperson and, based on the data, calculate an estimate for the expected profit per customer. However, there is an easier and less time-consuming method: the process can be simulated.

<table>
<thead>
<tr>
<th>Class range</th>
<th>0 - 0.24</th>
<th>0.25 - 0.49</th>
<th>0.50 - 0.74</th>
<th>0.75 - 0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of occurrences</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.3: Data for histogram from spinning the dial 20 times

![Histogram](image)

Figure 2.3: Histogram obtained from spinning the dial

The basic concept in simulation is to generate random outcomes (for example, we might toss a coin or roll dice to generate outcomes) and then associate appropriate physical behavior with the resultant (random) outcomes. To simulate the microwave oven buying decision, a (fair) coin is tossed with a head representing a customer and a tail representing a browser; thus, there is a 50-50 chance that an individual entering the department is a customer. To simulate the type of microwave that is bought by an interested customer, two fair coins are tossed. Two tails (which occur 25 percent of the time) represent buying the plain model, a head and a tail (which occur 50 percent of the time) represent buying the standard model, and two heads (which occur 25 percent of the time) represent buying the deluxe model. Table 2.4 summarizes the relationship between the random coin tossing results and the physical outcomes being modeled. Table 2.5 shows the results of repeating this process 20 times to simulate 20 customers and their buying decisions.

<table>
<thead>
<tr>
<th>Random value</th>
<th>Simulated outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tail-tail</td>
<td>Buy plain</td>
</tr>
<tr>
<td>Head</td>
<td>Customer</td>
</tr>
<tr>
<td>Tail-head</td>
<td>Buy standard</td>
</tr>
<tr>
<td>Head-tail</td>
<td>Buy standard</td>
</tr>
<tr>
<td>Head-head</td>
<td>Buy deluxe</td>
</tr>
</tbody>
</table>

Table 2.4: Procedure for determining the interest of a potential customer and the microwave model an interested customer buys

If we take the final cumulative profit and divide it by the number of customers, the estimate for the profit per customer is calculated to be $34.50. Two facts are immediately obvious. First, the number of interested customers was 12 out of 20, but because there is a 50-50 chance that any given customer will be interested, we expect only 10 out of 20. Furthermore, utilizing some basic probability rules, a person knowledgeable in probability would determine that the theoretical expected profit per customer is only $28.125. Thus, it is seen that the simulation does not provide the exact theoretical values sought.

The simulation is, in fact, just a statistical experiment. This point cannot be over-emphasized. The results of a simulation involving random numbers...
must be interpreted statistically. In future chapters, some basic statistical concepts will be given to aid in the proper analysis of simulation results. For now it is important to realize that the simulation is simply a statistical experiment performed so that the expense and time needed to perform and/or to observe the actual process can be avoided. For instance, this experimental evaluation might require less than an hour to accomplish, whereas obtaining an estimate of the theoretical profit by observing the actual process for 20 customers might take days.

- Suggestion: do Exercises 2.2 and 2.3.

### Table 2.5: Simulated behavior of 20 customers

<table>
<thead>
<tr>
<th>Customer number</th>
<th>Coin toss</th>
<th>Interested?</th>
<th>Two-coin toss</th>
<th>Profit</th>
<th>Cumulative profit</th>
<th>Average profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>yes</td>
<td>TH</td>
<td>60</td>
<td>60</td>
<td>30.00</td>
</tr>
<tr>
<td>3</td>
<td>H</td>
<td>yes</td>
<td>HH</td>
<td>75</td>
<td>135</td>
<td>45.00</td>
</tr>
<tr>
<td>4</td>
<td>H</td>
<td>yes</td>
<td>TH</td>
<td>60</td>
<td>195</td>
<td>48.75</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>195</td>
<td>39.00</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>195</td>
<td>32.50</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
<td>yes</td>
<td>TT</td>
<td>30</td>
<td>225</td>
<td>32.14</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>225</td>
<td>28.75</td>
</tr>
<tr>
<td>9</td>
<td>H</td>
<td>yes</td>
<td>HT</td>
<td>60</td>
<td>285</td>
<td>11.67</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
<td>yes</td>
<td>HH</td>
<td>75</td>
<td>360</td>
<td>16.00</td>
</tr>
<tr>
<td>11</td>
<td>H</td>
<td>yes</td>
<td>TT</td>
<td>30</td>
<td>390</td>
<td>35.45</td>
</tr>
<tr>
<td>12</td>
<td>H</td>
<td>yes</td>
<td>TT</td>
<td>30</td>
<td>420</td>
<td>35.06</td>
</tr>
<tr>
<td>13</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>420</td>
<td>32.31</td>
</tr>
<tr>
<td>14</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>420</td>
<td>30.00</td>
</tr>
<tr>
<td>15</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>420</td>
<td>28.00</td>
</tr>
<tr>
<td>16</td>
<td>H</td>
<td>yes</td>
<td>TH</td>
<td>60</td>
<td>480</td>
<td>20.00</td>
</tr>
<tr>
<td>17</td>
<td>H</td>
<td>yes</td>
<td>HT</td>
<td>60</td>
<td>540</td>
<td>21.75</td>
</tr>
<tr>
<td>18</td>
<td>T</td>
<td>no</td>
<td>-</td>
<td>0</td>
<td>540</td>
<td>20.00</td>
</tr>
<tr>
<td>19</td>
<td>H</td>
<td>yes</td>
<td>HH</td>
<td>75</td>
<td>615</td>
<td>32.37</td>
</tr>
<tr>
<td>20</td>
<td>H</td>
<td>yes</td>
<td>HH</td>
<td>75</td>
<td>690</td>
<td>34.50</td>
</tr>
</tbody>
</table>

### 2.3 A Simple Event-Driven Example

When the timing of activities is an important component of a simulation study, a decision must be made as to how to treat time. The two common methods for dealing with time are (1) to increment time a fixed amount and (2) to increment time only when something will change the system. In both cases the simulation must maintain an internal clock, and thus the two methods refer to the procedure by which the internal clock is updated. In event-driven simulations (the second method), the clock is updated whenever an event occurs that might change the system. This concept is illustrated in the following example.

Consider a simplified manufacturing process where widgets are processed through a production center. One widget arrives at the facility every 12 minutes; however, because of poor quality control, 25 percent of these widgets are defective and must be scrapped. The remaining 75 percent of the widgets are sent to a work station where they are processed one at a time through a single machine. The chemical composition of the widgets varies; thus, the processing time required on the machine varies. Twenty-five percent of the widgets require 17 minutes of machining time, 50 percent of the widgets require 16 minutes, 12.5 percent of the widgets require 15 minutes, and 12.5 percent require 14 minutes. While a widget is on the machine, other widgets that arrive at the work station are placed in a special holding area until the widget being processed is completed and removed. The next widget is then placed on the machine and the process is repeated. Widgets in the holding area are said to be in a queue (waiting line) waiting for the machine to become available. To characterize this process in traditional queueing terminology, we would say that we are dealing with a single server queueing system (the machining process operates on widgets one at a time) having a deterministic arrival process, random service times, and utilizing a first-in-first-out selection scheme (FIFO queueing discipline) for the next item in the queue to be processed.

We wish to simulate this production center in order to answer the question—“What is the long-run average number of widgets in the facility?” For this system, two random processes must be simulated (see Table 2.6). The occurrence of the arriving widgets and their status (defective or nondefective) will be decided by tossing two fair coins: two tails indicate scrap (25 percent of the outcomes) and any other combination indicates a good widget. The second random process, namely the machining time, will be decided by tossing three fair coins: three tails (with a chance of 1 in 8 or 12.5 percent)
will indicate 14 minutes, three heads (again, 1 chance in 8) will indicate 15 minutes, the combinations THT, TTH, HTT, THH (with a chance of 4 in 8 or 50 percent) will indicate 16 minutes, and EHT, HTH (with a chance of 2 in 8 or 25 percent) will indicate 17 minutes of machining time.

The widget example is more complicated to simulate than the salesperson example because of the queue in front of the work station. We must keep track of the widget arrival times and processing completion times. The conceptual approach for the simulation is to maintain a “simulated clock” and record the physical status of the system that is being modeled each time a change to the system occurs. This method is called “next event simulation,” which is more efficient for this type of system than the systematic time-step approach.

Table 2.6 Procedure for determining the quality of arriving widgets and the machining time for those widgets that are actually processed

<table>
<thead>
<tr>
<th>Random value</th>
<th>Simulated outcome</th>
<th>Random value</th>
<th>Simulated outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-T-T</td>
<td>14 minutes</td>
<td>T-T-H</td>
<td>15 minutes</td>
</tr>
<tr>
<td>H-H-H</td>
<td>15 minutes</td>
<td>H-H-T</td>
<td>15 minutes</td>
</tr>
<tr>
<td>T-H-H</td>
<td>16 minutes</td>
<td>T-T-H</td>
<td>16 minutes</td>
</tr>
<tr>
<td>H-T-H</td>
<td>16 minutes</td>
<td>T-H-H</td>
<td>16 minutes</td>
</tr>
<tr>
<td>T-H-T</td>
<td>17 minutes</td>
<td>H-H-T</td>
<td>17 minutes</td>
</tr>
<tr>
<td>H-T-H</td>
<td>17 minutes</td>
<td>H-T-H</td>
<td>17 minutes</td>
</tr>
</tbody>
</table>

Table 2.7 displays the steps of the simulation. The simulation (Table 2.7) begins at time zero with the arrival of the first widget (see row 1, column 1). Two coins are tossed to determine if the arriving widget will be scrapped or processed. Since the coin toss results in two tails, the widget is bad and does not enter the production facility (see row 1, columns 4 through 5). A widget arrives every 12 minutes: therefore, the system (i.e., work station) remains empty at least until the next arrival at time 12.

When the next widget arrives, two coins are again tossed to determine the quality of the widget. Since a tail-head occurred and the widget is considered good, it enters the production facility (as reflected in row 2, column 6), and the machining process begins. It is now necessary to determine how long the widget will be on the machine. The result of tossing three coins (row 2, column 7), is a 16-minute machining process. Since the current clock is 12, the 16 minutes are added to the current time and the widget is scheduled to finish the machining process and leave the facility at time 28.

All of the activities that occur at time 12 have been completed and now the time of the next event (i.e., next change to the system) must be obtained. Looking ahead, there are two possible events that might cause the system to change: the next arrival will occur at time 24 and the widget currently on the machine will finish at time 28. Since the next arrival occurs before the machining process finishes, the clock is advanced to time 24 and the next widget arrives.

The third widget and associated activities are displayed in the third row of Table 2.7. The widget is considered acceptable for machining after the two coin toss; however, this time the machine is busy so the widget must wait for the processing widget to finish. Note that the number in the system (row 3, column 6) is updated to 2, but a service time is not determined until the widget is actually placed on the machine.

The last column in the table is used to keep track of the average number of widgets in the system. The time-averaged number of widgets is determined by summing the number of widgets within the facility multiplied by the length of time each widget spent in the facility and then dividing by the total length of time. The last column of Table 2.7 accumulates this sum, which will be divided at the end of the simulation by the total time to obtain the average number. At time 24, one item has been in the system for 12 minutes so the last column contains a 12.

To continue the simulation, we again look for the time that the next change in the system will occur. The two possibilities are at time 36 when the next widget arrives and at time 28 when the current widget leaves the machine. Since the minimum of those two times is 28 minutes, the clock is advanced to time 28. The current widget now leaves the system and the next widget is placed on the machine. Three coins are tossed to determine the processing time, and since they come up head-head-head, we assign 15 minutes of machining time to this widget. The 15 minutes of machining time is added to the current clock time of 28 minutes, and we record in column 8 that the widget will leave the facility at time 43. The final column of Table 2.7 is updated by observing that the system has contained 2 widgets during the previous 4 minutes, so we add 8 units to the previous sum of 12.
2.5 Exercises

2.1. Using two dice to generate random observations, draw a histogram from the results obtained from 10 rolls. Then draw a histogram from 50 rolls. Which experiment would you expect to give a better estimate for the probability of rolling a 7? Compare what you expected with what your experiments did.

2.2. Using two coins, simulate 20 customers from the example of Section 2.2 and estimate the average profit per customer. Compare your estimate with the estimate from Table 2.5.

2.3. A door-to-door salesman sells pots and pans. He only gets in 50 percent of the houses that he visits. Of the houses that he enters, 1/6 of the householders are still not interested in purchasing anything, 1/3 of them end up placing a $60 order, and 1/3 of them end up placing a $100 order. Estimate the average sales receipts per house visit by simulating 25
2.4. Using three coins, simulate the production process of Section 2.3. Stop the simulation as soon as the simulated clock is greater than or equal to 100 and estimate the average number of widgets in the system. Compare your estimate with the estimate from Table 2.7.

2.5. Simulate the production process of Section 2.3, but assume that there are several machines available for the machining process. In other words, if a widget is acceptable (not destined for the scrap pile) and goes to the facility for machining, there are enough machines available so that the widget never has to wait to be machined. Thus, any number of widgets can be machined at the same time. Estimate the average number of widgets in the facility over a 100-minute period.

2.6. Simulate the production process of Section 2.3, but assume that arrivals are random. Specifically, the time between arriving widgets will equal 12 minutes half the time, 11 minutes one-third of the time, and 10 minutes one-sixth of the time. Use a die along with the coins to simulate this system. Estimate the number of widgets in the facility over a 100-minute period.

2.7. The manager of a service station next to a busy highway is concerned about congestion. The service station has two gas pumps and plenty of room for customers to wait if the pumps are busy. Data indicate that a car arrives every 3 minutes. Of the customers that arrive, 25 percent of them take 5 minutes 45 seconds in front of the pump and spend $10, 50 percent of them take 5 minutes 50 seconds in front of the pump and spend $10.50, and 25 percent of them take 5 minutes 55 seconds in front of the pump and spend $11. Simulate the arriving and servicing of cars and estimate the average receipts per hour and the average number of customers in the service station.

2.8. Consider the service station of Exercise 2.7 except that the time between arrivals is not fixed. Assume that the time between arriving cars is 2.5 minutes with probability 0.25, 3 minutes with probability 0.50, and 3.5 minutes with probability 0.25. Using simulation, estimate the average receipts per hour and the average number of customers in the service station.

2.9. A computer store sells two types of computers: the MicroSpecial and MicroSuperSpecial. People arrive at the computer store randomly throughout the day, such that the time between arriving customers is 40 minutes with probability 0.5 and 50 minutes with probability 0.3. The store is small and there is only one clerk who helps in selection. (The clerk handles customers one at a time in a FIFO manner.) Twenty-five percent of the customers who enter the store end up buying nothing and using exactly 15 minutes of the clerk's time. Fifty percent of the customers who enter the store end up buying a MicroSpecial (yielding a profit of $225) and taking a random amount of the clerk’s time, which is approximated by a discrete uniform distribution between 31 and 36 minutes. Twenty-five percent of the customers who enter the store end up buying a MicroSuperSpecial (yielding a profit of $700) and taking a random amount of the clerk’s time, which is approximated by a discrete uniform distribution between 51 and 56 minutes. The store is open only 4 hours on Saturday. The policy is to close the door after the 4 hours but continue to serve whoever is in the store when the doors close. Simulate the activities for a Saturday and answer the following questions. (The discrete uniform distribution can be simulated by rolling a die.)

(a) The store opens at 10:00 A.M. and closes the door at 2:00 P.M. Based upon your simulation experiment, at what time is the clerk free to go home?

(b) What is your estimate for the the total profit on Saturday?

(c) If a random time was picked between 10:00 A.M. and 2:00 P.M., what would be the expected number of customers in the store?

(d) Simulate the process a second time, and determine how much your answers change.
Chapter 3

Getting Started

This chapter provides an introduction for the MOR/DS modeling language as well as a description of how to setup and use the microcomputer environment provided by the software that accompanies this book. From the user’s standpoint, the system is a text editor program that provides a command that causes the model currently in the editor to be executed. The editor serves both as the tool used to enter the model into computer memory and as the simulation program itself. The MOR/DS system was designed to minimize the uncertainties and hassles of using a computer. You do not need to learn command names and computer jargon to use the editor. As you begin this chapter, remember the following points: (1) the ESC key gets you out of the menu system, (2) the F1 key will give you help regarding the rules of the MOR/DS language, (3) the F1 key together with the control key gives you a help screen for the editor, (4) the F9 key is the fast way to quit the system (provided all menus are inactive), and (5) experiment (in most cases, the worst that can happen is a beep!).

In order to get you involved immediately in the development of simulation models, some preliminary information and structure must be learned. From this rudimentary base, you can develop simple simulation models to aid in learning the concepts of simulation.

This chapter presents a step-by-step execution of a simulation program and an overview of the editor. By following the example, you should obtain an appreciation of the capabilities and use of the MOR/DS system.

In this text we follow some typing conventions to enable you to distinguish quickly between system words and user-supplied names. All system keywords will be typed in boldface (e.g., Exp and Sqrt), whereas user-supplied names use a typewriter font (e.g., myVariable). A “hook” arrow (→) indicates the use of the return key, and underlined names refer to menu names or menu item names.

The MOR/DS system comes on a single diskette suitable for an IBM personal computer or other fully compatible microcomputer. MS-DOS 2.1 or greater is required, and a minimum of 640K random access memory is needed. The MOR/DS diskettes contain DOS system files, so you must provide your own system disk. (The system disk contains some hidden files and the DOS file COMMAND.COM.)

3.1 Overview of the Language Structure

Each simulation program developed in MOR/DS must start with the word PROGRAM and conclude with the word END followed by a period. Furthermore, every program has three basic components: the DEFINITION segment, the CONTROL segment, and the LOGIC segment. These three segments must always be defined and must appear in this specific order. The DEFINITION segment is used to define system parameters such as resource capacities and queueing selection disciplines. (An example of a resource capacity is the number of machines used in the example of Section 2.3. In that example, the resource capacity was one because the machine could only process one widget at a time. An illustration of the queueing selection discipline is the FIFO rule used in that same example.) User-defined data such as constants, arrays, and functions also appear in the DEFINITION segment. The CONTROL segment is used to define the simulation run length characteristics and other control information for output and animation. The LOGIC segment contains the flow of logic governing the simulation and is the executable portion of the program.

3.2 The First Program

This chapter is intended to develop only the minimum structures necessary to proceed with the study of simulation using a language-oriented approach. These structures have been placed in a model example file called START.DSE, which will be discussed shortly. (START.DSE is found on the diskettes.)

---

1 We remind you that the copyright law prohibits the copying of the MOR/DS disk for use on more than one microcomputer at a time.
3.2. THE FIRST PROGRAM

3.2.2 The Editor

The system editor operates on standard ASCII text files. The top line of the editor is reserved to remind the user of the meaning of some of the function keys. The second line of the editor indicates the current position of the cursor and the file from which the current model was read. Pulldown menus provide a variety of file-handling, block editing, and executing commands. To activate the menu system and access the menu commands, press F10. After the F10 key is pressed, the following will be displayed on the screen:

```
| File  | System | Run   | Options | <ESC to exit> |
```

These are the headers for the four available menus. The appropriate header is selected by moving the highlighted bar with the cursor keys and pressing ↙. The header may also be selected by pressing the key corresponding to the capitalized letter in the header name. The corresponding pulldown menu is displayed and the top bar is highlighted. Again, selections may be made either by pressing the capitalized letter or by moving the highlighted bar with the cursor keys and pressing ↙. The ESC key deactivates the menu system one level at a time.

The editor is quite powerful and easy to use once you become familiar with its operation. The menus eliminate the need to remember command names. Editing commands other than the menu items are given in Table 3.1 at the end of this chapter. The most common keys are the cursor movement (arrow) keys and the "end," "home," "pg up," and "pg down" keys on the number pad. For example, "pg up" scrolls the screen up one full page, "pg down" scrolls the screen down one full page, "<ctrl>pg up" moves the cursor to the top of the file (the symbol <ctrl> before a letter or key indicates that the control key is held down while the letter or key is pressed), and "<ctrl>pg down" moves the cursor to the bottom of the file. A black line can be inserted by going to the end of the current line (by pressing the "end" key) and then pressing the return key. The current line is deleted by the command <ctrl>L. (Either upper- or lower-case letters may be used for commands.)
3.3 General Language Syntax

In this section, we will give some general rules for writing simulations. The parameters for blocks and definitions are enclosed in braces. If a number contains a decimal point, there must be a digit to the left of the decimal point. Thus 0.4 is a legal number, but .4 is not permissible. It should be noted that all language statements must end with a semicolon. Multiple statements per line may be used, or statements can extend over more than one line.

The structure of the simulation model is specified by a collection of statements called blocks. Blocks have parameters, some of which are required, and some of which are optional and use default values. The block provides the name of the command, while the parameter(s) provide the specific information required by the command. (As with all statements, block statements end with a semicolon). Blocks are activated by system entities as they flow through the simulation model. The general block syntax is:

\[
<\text{optional label}:> \text{ BLOCK NAME} \{ \text{Keyword = expression} \}
\]

The simulation language syntax is a freeform structure; the block and definition parameters can be listed in any order because they are designated by keywords. (The syntax provides in aid in remembering when order is important: brackets are used when order is not important, and parentheses are used when order is important.) Since most of the parameters are easy to remember when developing a simulation model, you will soon discover that the keyword approach of MOR/DS is very easy to use and creates very readable programs. For example in the HISTOGRAM definition statement of START.DSE, the three parameters needed are the number of histogram cells, the minimum value for the histogram, and the maximum value; thus, the parameter names are Cells, MinValue, and MaxValue. In defining a histogram it becomes relatively easy to remember that these three parameters are needed, but their specific order is a matter of choice and the freeform style helps in that regard. The histogram may be named whatever the user desires; in this case it is named myData. A global variable, myNumber, is also used in the START.DSE model; hence, it is defined in the DEFINITION segment and is given an initial value of 0 since all variables used in the LOGIC segment must be predefined.

The CONTROL segment has several possible parameters which are defined in Chapter 5. Many simulations are controlled using just one param-
CHAPTER 3. GETTING STARTED

3.4. OVERVIEW OF OUTPUT

The following is the output that results from a run of the program stored in the file START.DSE.

MOR/DS 1.00

Date: 8/31/86

Time: 12:52:14

A:\START.DSE


3.4. OVERVIEW OF OUTPUT

- - - - - - - - - - Global Variables - - - - - - - - - -

**MYPNUMBER** = $6.00

Compile time: 0.28 (Secs.)
Run time : 2.36 (Secs.)
Total memory available : 213546
Memory used by model : 1304
Maximum dynamic memory used : 208
Total memory used : 1512

The system printout consists of several parts, beginning with a listing of the simulation model. Once the model has been executed, a relisting occurs, each line is given a sequential number (in square brackets), and the block entity activities are displayed (that is, the total number of entity entries into each statement and the current number residing in each statement are displayed). The ARRIVE block at the printout time of 100 time units has a total count of 31 and a current count of 1. This means that 30 entities have been generated and have left this block, with 1 entity currently waiting for its interarrival time to expire before passing to the next statement in the model. (Because this model uses random numbers, the output shown here may not agree with yours.)

In models more complicated than our beginning example, additional output will consist of the statistical summaries for resources and associated queues (discussed later). Histogram data are displayed in categories for the number of cells defined for the histogram, plus an underflow cell and an overflow cell. These data are printed as the number of entities falling within each category. In the output, categories are represented by their upper limit values. The mean and standard deviation of the histogram entries, along with the minimum and maximum entries, are also displayed.

### 3.4.1 Output Control

The Options menu contains an option for output control. The option is labeled **Output** and it controls the device for the output. The three possibilities are to send the output to the screen (the default option), send the output to the printer, or send the output to a file. If a file is desired,
the output will be stored as an ASCII text file on a disk. (The name of the output file is the same as the file submitted to MOR/DS with the additional extension .OUT). To illustrate the use of output control, let us send the output to the printer. First, make sure the printer is connected to the computer and is turned on. Then, type the sequence F10-0-0 (F10 for menu: selection, 0 for the Options menu, and 0 for the Output option). The screen should now display a window requesting further information. Type P for the printer selection, press ESC twice to exit from the menu system, and press the F4 key to execute the program. After a run, return the output to the screen by repeating the same commands except type S instead of P when the window appears requesting the output device option.

3.4.2 Error Trapping

To illustrate the error trapping capabilities of the system, first make sure you are back in the editor (by pressing ← if the output is still on the screen). Position the cursor in front of the word ARRIVE and type an extra t (thus misspelling the word). Now, execute the system again (using F4) and notice the compiler halts and displays an error message. Press any key to return to the editor and notice where the cursor is positioned. MOR/DS attempts to place the cursor immediately following an error.

3.4.3 The Help System

To illustrate the user help interface of the MOR/DS language, press the F1 key. A menu of help categories is listed on the screen. Any of the words in all upper-case letters are words for which there is help information. To access the help file for any of these keywords, type the keyword in the blank space and hit the return key. For example, to see the list of statements possible in the LOGIC segment, type LOGIC. To exit from the help system, simply press the ESC key. To obtain a summary of the various editor commands, press <ctrl>F1.

3.4.4 Quitting the Session

A MOR/DS session is terminated by the F9 key or through the menu system by the F10-F-0 sequence. After pressing the F9 key, the system asks if you want to save the current model before the session is terminated; respond with Y for yes. A Y response will automatically save your changes over your original file on the disk. If you would like to save the program under a new file name, the saveAs option in the File menu must be used.

3.5 Exercises

The following exercises are designed to assist you in becoming comfortable with MOR/DS and to encourage experimentation. Before starting these exercises, load the MOR/DS system and then load the START.DSE file.

3.1. Before executing the program in START.DSE, plot the histogram that the program should generate. Execute the program and determine if your plot was correct. If there is a difference between your plot and the program's data, explain why. Run the program a second time. Now is there a difference?

3.2. Delete the first semicolon in the program START.DSE and replace the first program line in the LOGIC segment with the line

```
ARIVE { Tim = 2*2.25; }
```

You have now created at least three errors in the program. Execute this program and correct the syntax as errors are discovered by the compiler. (If you get confused, press F1 and type ARRIVE.)

3.3. Change the original program in START.DSE so that an arrival occurs every 10 time units.

3.4. Change the original program in START.DSE to simulate 50 customers arriving to a toll booth where the time between each customer's arrival is 7.5 minutes.

3.5. Save the resultant model from Exercise 3.4 under the name STARTM.DSE. (Hint: Try the key sequence F10-F-A.)
Chapter 4

Generation of Random Variates

The examples of Chapter 2 simulated random outcomes by having the modeler physically toss coins. A computer, by contrast, must simulate random outcomes by generating numbers to give the appearance of randomness. This chapter deals with the problems involved in programming a computer to simulate randomness. The goal of this chapter is to introduce the tools necessary for statistically describing and analyzing a physical situation for modeling purposes. By making slight changes in the program file START.DSE, sample simulations can be executed to help solidify the concepts introduced in this chapter. These changes are pointed out throughout this chapter.

4.1 Uniform Random Numbers

Most mathematical table books contain several pages of random numbers. Such random numbers have three properties: (1) the numbers are between zero and one. (2) the probability of selecting a number in the interval \((a, b)\) is equal to \((b - a)\), where \(0 \leq a < b \leq 1\), and (3) the numbers are statistically independent. The concept of statistical independence simply means that if several numbers are chosen, knowledge of the value of one number does not provide information that will help in predicting the value of another number. Independence, therefore, rules out the possibility of trends or cyclic patterns occurring within a sequence of random numbers. In terms of probability, these random numbers are sequences of independent, identi-
cally distributed random variates having a continuous uniform distribution between zero and one.

Technically, random numbers can refer to the observations of random variables having any arbitrary probability distribution; however, usually the term random numbers refers to the observations of uniform zero-one random variables. For observations of random variables governed by a distribution other than the uniform zero-one, the more general term random variates is used. Thus, the term random numbers refers to numbers that statistically reproduce observations of zero-one uniform random variables, and the term random variates refers to numbers that statistically reproduce observations of arbitrarily distributed random variables.

As will be shown later, uniform zero-one random numbers can be used to generate (at least theoretically) random variates from any given distribution function. The first task we must do is to use a computer to generate random numbers like those generated from the ”spinning pointer” experiment in Table 2.2 and those contained in tables. Because a computer can perform only very specific steps, random numbers generated by a computer are not truly random but are more properly called “pseudo random numbers” (although the “pseudo” prefix is usually dropped). Numbers generated by a computer program or subprogram are, therefore, called “random” if statistical tests cannot determine the difference between computer-generated and truly random sequences of numbers. (Some of these statistical tests will be discussed in Section 4.4.)

All methods of random number generation begin with an initial value called the initial random number seed. Each time a random number is desired, the random number seed is used to produce the next random number and the seed is transformed into another number that becomes the new seed. Thus, the random number seed is continually changed as the random numbers are generated. A simulation program must always furnish an initial random number seed. If the same initial random number seed is used whenever the simulation program is run, the exact same string of random numbers will be generated. If a different string of random numbers is desired, then a different initial random number seed must be furnished. Most simulation programs allow the user to change the initial seed if so desired.

One of the most popular methods of generating random numbers by a computer is the congruential method. In this method the random number seed is always a positive integer, and the random number associated with the seed is the value of the seed divided by the largest possible integer that can be stored in the computer. To obtain the new seed, the old seed is multiplied by a large constant and a second constant is added to the product. The resulting integer might then be too large (i.e., larger than the largest possible integer for the computer size). If it is too large, it is divided by the largest possible integer and the remainder is the new seed. If it is not too large for the computer word size, it becomes the new seed. To express this mathematically, let a and b be two fixed integers, and let L denote the largest possible (signed) integer that the computer can store. Let \( S_{old} \) be a random seed and let \( S_{new} \) be the next seed to be determined. The random number associated with the seed is

\[
R_{old} = \frac{S_{old}}{L},
\]

and the next seed is

\[
S_{new} = (aS_{old} + b) \mod L.
\]

For example, for a 16-bit microcomputer, \( L = 32767 \) and we might set:

\[ a = 1217, \quad b = 0, \quad \text{and let the initial random number seed be } S_0 = 23. \]

For this situation, the random number sequence is generated by the following calculations:

\[
S_1 = 1217 \times 23 = 27991
\]

\[
R_1 = \frac{27991}{32767} = 0.85424 \quad \text{(first random number)}
\]

\[
S_2 = (1217 \times 27991) \mod 32767
\]

\[
= 34065047 \mod 32767 = 20134
\]

\[
R_2 = \frac{20134}{32767} = 0.61446 \quad \text{(second random number)}
\]

There are several “rules of thumb” used to determine values for the constants \( a \) and \( b \) that will produce good pseudo random number strings. However,
most high-level languages contain their own random number generators; although you need not be concerned with programming these methods, a conceptual understanding of random number generation is beneficial. It is especially important to remember the role of the initial random number seed. If the initial seed is the same for every run of a given simulation program, the output will be the same.

4.1.1 Supporting Language Structures

The most productive method for learning simulation is to do it! Whenever we present a new simulation concept, we will present also the necessary blocks, keywords, and accompanying language syntax so that the concept can be tried, practiced, and reinforced by utilizing MOR/DS. In the following paragraphs, we will develop the language syntax for understanding and changing the START.DSE initial model for random number generation studies. (Remember: Numbers that contain a decimal point must have a digit to the left of the decimal.)

Logic Statements. The ARRIVE block has four parameters that may be used in defining the time between arrivals. The general form is

\[
\text{ARRIVE} \{ \text{Time} = x_1, \text{Limit} = x_2, \text{Offset} = x_3, \text{Quantity} = x_4 \};
\]

where Time is the time between arrivals, Limit is the maximum number of entity groups to be generated by the block, Offset is an additional delay before the ARRIVE block starts operation (i.e., the first entity arrives at \(\text{Time} + \text{Offset}\)), and Quantity is the number of individuals in each arrival group (i.e., the arrival batch size). Time is the only keyword required; the other three keywords are optional. If no Limit is given, there is no upper bound on the number of entities that can arrive through the block. If no Offset is given, the offset is set to zero. If no Quantity is given, exactly one entity is generated at each arrival time. For example,

\[
\text{ARRIVE} \{ \text{Time} = 10, \text{Limit} = 4, \text{Offset} = 69, \text{Quantity} = 2 \};
\]

will cause two entities to arrive at times 70, 86, 90, and 100 if the simulation program does not terminate before time 100 for other reasons.

The last statement in a program is END; however, this is not actually an executable statement since it serves only to identify termination for the compiler. Furthermore, it is important to keep in mind that the simulation is driven by the flow of entities. It does not make sense for an entity to ever activate the END. Once an entity is no longer needed, it should enter a DEPART block. If control of the simulation is governed by the constant StopTime, then the syntax to use is

\[
\text{DEPART} \{ \};
\]

System Functions and Parameters. There are many built-in functions (called system functions) within the simulation language. Some of the system functions need parameters and some do not. The system function Random needs no parameters and is used to generate a uniform random number between zero and one. For example,

\[
\text{ARRIVE} \{ \text{Time} = \text{Random} \};
\]

will cause entities to arrive randomly, where the time between each individual arrival is a uniform random number between zero and one.

To generate random numbers, the simulation language sets an initial random number seed so that every time the system is loaded and executed, the same string of random numbers is generated. The optional parameter Randomize can be used in the CONTROL segment of the program to generate a random initial random number seed. Normally, the Randomize parameter is set to Off, in which case the same initial random number seed is used whenever the program is executed. However, if Randomize is set to On, the initial random number seed is randomized (actually, it is determined by a function dependent on the internal clock of the computer). Note that whenever Randomize is not explicitly listed in the CONTROL segment, it is automatically set to Off. If every simulation should use the same sequence of random numbers different from the default sequence, the statement

\[
\text{Seed} = z2;
\]

is used in the CONTROL segment, where \(z2\) is an integer between 1 and 32767. The effect of the Seed statement is to set the initial random number seed equal to the number on the right-hand side of the equality. For example, if the control portion of your program is

\[
\text{CONTROL}
\]

\[
\text{StopTime} = 1000; \text{Randomize} = \text{On};
\]

then every time the simulation program is executed, different random number sequences and, hence, different results are obtained. However, if the control portion of your program is

\[
\text{CONTROL}
\]

\[
\text{StopTime} = 1000; \text{Randomize} = \text{Off};
\]

then every time the simulation program is executed, the same random number sequences will be generated, and hence, the same results will be obtained.
CONTROL
StopTime = 1000; Seed = 29;
then the simulation continues until 1000 time units have elapsed. Every time
the program is executed, the sequence of random numbers will be identical.

Let us return to the program stored in the disk file START.DSE. We
shall change two lines of the program to illustrate uniform zero-one random
numbers. The HISTOGRAM definition statement and the ARRIVE block
are changed but the other lines of the program remain the same, as is shown
in the program below.

PROGRAM
DEFINITION
  myNumber = 0;
  myData : HISTOGRAM = { Cells = 4, MinValue = 0,
                         MaxValue = 1 };

CONTROL
  StopTime = 100;
LOGIC
  ARRIVE { Time = Random };
  myData = ClockTime - myNumber ;
  myNumber = ClockTime ;
  DEPART { };
END.

When this program is executed, the histogram called myData will contain
uniformly generated discrete numbers between 0 and 1.
•  Suggestion: do Exercises 4.1 – 4.8.

4.2 Discrete Random Variates

Consider developing a simulation model of the microwave salesperson
example of Section 2.2. An immediate problem arises because the example
involves two random variables, neither of which is a continuous uniform zero-
one random variable. The first random variable represents the decision as to
whether or not a customer will end up buying a microwave. Mathematically,
we can denote the customer’s interest by the random variable C and give its
probability mass function as

\[ P(C = 0) = 0.5 \]
\[ P(C = 1) = 0.5 \]

where the random variable being zero \( C = 0 \) represents a customer who
is not interested in purchasing a microwave, and the random variable being
one \( C = 1 \) represents the customer who purchases a microwave.

There are two mathematical methods for representing the probability law
of a discrete random variable: (1) the probability mass function, which gives
the probability that the random variable will take on a specified value and (2)
the cumulative probability distribution function, which gives the probability
of the random variable taking on a value less than or equal to the specified
value. The graph of the cumulative probability distribution function for \( C \)
is given in Figure 4.1.

![Figure 4.1: Cumulative probability distribution function for the random variable C](image-url)

The cumulative probability distribution function permits an easy trans-
formation of the uniform zero-one random number into a discrete random
variate arising from another probability law. The transformation is obtained
simply by letting the uniform zero-one random number occur along the ordi-
nate (y-axis) and using the cumulative function to form the inverse mapping back to the abscissa (x-axis). For the "customer interest" random variable \( C \), the following rule is obtained: If the uniform zero-one random number is less than 0.5, then \( C = 0 \), i.e., the customer is not interested in a purchase; otherwise, if the uniform zero-one random number is greater than 0.5, then \( C = 1 \), i.e., the customer is interested. This transformation is represented schematically in Figure 4.2 for one possible value of the random number.

\[
\begin{align*}
\text{Random number} & : 0 \rightarrow 0.5 \\
\text{Distributed random variate} & : 0 \rightarrow 1
\end{align*}
\]

Figure 4.2: Transformation to obtain the random variate \( C \)

The second random variable that must be simulated in the microwave dispenser example is the profit resulting from the purchase of a microwave. Let the "model" random variable be denoted by \( M \) with probability mass function given by

\[
\begin{align*}
P(M = 30) &= 0.25 \\
P(M = 60) &= 0.50 \\
P(M = 75) &= 0.25
\end{align*}
\]

In this case, \( M = 30 \) is the profit from the purchase of a basic model, \( M = 60 \) the profit from a standard model, and \( M = 75 \) the profit from a deluxe model. Again, it is easiest to simulate this random variable if its cumulative probability distribution function is first determined. The jump points of the cumulative function are given by

\[
\begin{align*}
P(M \leq 30) &= 0.25 \\
P(M \leq 60) &= 0.75 \\
P(M \leq 75) &= 1.0
\end{align*}
\]

The inverse mapping that gives the transformation from the uniform zero-one random number to the random variate representing the profit obtained from the purchase of a microwave is illustrated in Figure 4.3. The results are: If the uniform zero-one random number is less than 0.25, the plain model is purchased; if it is greater than or equal to 0.25 and less than 0.75, the standard model is purchased; and if it is greater than or equal to 0.75, the deluxe model is purchased.

\[
\begin{align*}
\text{random number} & : 0 \rightarrow 0.75 \\
\text{distributed random variate} & : 0 \rightarrow 75
\end{align*}
\]

Figure 4.3: Transformation to obtain the random variate \( M \)
CHAPTER 4. GENERATION OF RANDOM VARIATES

The rather intuitive approach for obtaining the random variates \( C \) and \( M \) as shown in Figures 4.2 and 4.3 needs to be formalized for use for continuous random variables. Specifically, Figures 4.2 and 4.3 indicate that the random variates were generated by using an inverse of the cumulative probability distribution function. A mathematical justification for this is now given.

The following property (proven in most introductory mathematical probability and statistics books) leads to the so-called inverse transformation method of generating arbitrarily distributed random variates.

Property 4.1. Let \( X \) be a continuous random variable with cumulative probability distribution function \( F \); that is,

\[
P\{X \leq a\} = F(a) \quad \text{for} \quad -\infty < a < \infty.
\]

Then \( F(X) \) is a random variable with a continuous uniform distribution between zero and one. Furthermore, if \( R \) is a random variable with a continuous uniform distribution between zero and one, and \( F^{-1} \) is the inverse of the cumulative probability distribution function, with \( X \) defined by

\[
X = F^{-1}(R),
\]

then \( X \) is a random variable with distribution function \( F \).

From the above property, it is seen that the function that relates the uniform zero-one random variate with another random variable is simply the inverse of the cumulative probability distribution function. We utilize his property in the next section.

1.2.1 Supporting Language Structures

User Definitions. A mechanism is needed for defining the discrete function that represents the inverse of the cumulative probability distribution function for a discrete random variable. Such a mechanism will allow for the generation of arbitrarily distributed discrete random variables. The DEFINITION segment of the simulation language contains provisions for user-defined functions through the DISTRIBUTION statement. The statement has the form

\[
\text{myName : DISTRIBUTION(Discrete)} = (\text{sequence of ordered pairs});
\]

where myName is the user-defined name that represents the inverse of the distribution function being used to generate a random variable. The definition type DISTRIBUTION has the modifier Discrete, indicating that the function being defined is discrete. The parameter list within the parentheses is a sequence of ordered pairs defining the function. Each ordered pair has the form \((zx, yy)\), where \(zx\) is the cumulative probability and \(yy\) is any real value. Each ordered pair must be separated by a comma. (Note also that the successive values of \(zx\) should be increasing, as should the successive values of \(yy\), if the function is to be used for random variate generation. This structure can be used for any function; to emphasize that it need not be a distribution, the keyword MAPPING can be used.) For example, the function for the random variable \( C \) as in Figure 4.2 would be defined as follows:

\[
\text{cFunction : DISTRIBUTION(Discrete)} = ((0.5, 0), (1.1));
\]

Note: that parentheses are used in the DISTRIBUTION definition on the right-hand side of the equality, because the order within the list is important.

System Functions. There is an alternate method to generate the random variable \( C \) because of its special structure. A discrete uniform random variable is a random variable that has an equal probability of taking on any of the values within its range. For example, tossing a fair coin or rolling a fair die would yield discrete uniform random variables. Because discrete uniform random variables are often used for modeling, a system function has been defined to generate these variates. Its syntax has the form

\[
\text{Duniform}(zx, yy)
\]

where the integer \(zx\) is the first value, the integer \(yy\) is the final value, and every integer between \(zx\) and \(yy\) (inclusive) is a possible value having an equal probability of occurring. To illustrate, consider the following two example programs.

PROGRAM "Example 1"
DEFINITION
myDistribution: DISTRIBUTION(Discrete) = ((0.25, 0.30),
(0.50, 0.31), (0.75, 0.32), (1.0, 0.33));
CONTROL
Start-Time = 200;
LOGIC
ARRIVE \{ Time = myDistribution(Random) \};
DEPART \{
\};
END.
4.3 Continuous Random Variates

The assignment statement is illustrated by the following program, which simulates the profit obtained from 20 customers at the microwave center.

```plaintext
PROGRAM "Simulation of 20 customers in microwave store"
DEFINITION
    totalProfit = C;
    cValue = 0;
    aPurchase : DISTRIBUTION(Discrete) = ((0.25,30),
                                        (0.75,60),(1.0,75));
END.

CONTROL
    StepTime = 20;
END.
LOGIC
    ARI V E { Time = Duniform(1,20) }
    DEPART { }
END.
```

Each time the keyword Random is encountered, a new uniform zero-one random number is generated. When used in the assignment statement within the above program, the random number becomes the independent variable of the user defined function aPurchase. The variable cValue indicates whether or not the arriving customer is interested in making a purchase. When cValue = 0, the arriving customer is uninterested in purchasing a microwave; however, the simulation program still computes the value of aPurchase but since it is multiplied by zero, its value is ignored.

* Suggestion: do Exercises 4.4 - 4.7.

4.3 Continuous Random Variates

Although many physical systems can be modeled using discrete random variables, there are many systems that are more appropriately modeled using continuous random variables. For example, the times between arrivals of customers to a teller window in a bank would be best described by a continuous value instead of an integer.
One of the most commonly encountered continuous random variables is the exponentially distributed random variable. Specifically, if $T$ is an exponentially distributed random variable, then its cumulative distribution function is given by

$$F(a) = 1 - e^{-a/\theta} \quad \text{for } a \geq 0,$$

where $\theta$ is a positive scalar with $E[T] = \theta$.

The inverse transformation method (Property 4.1) can be used to generate exponentially distributed random variates from uniform zero-one random numbers. To see this, let $R$ be a uniform zero-one random number and $T$ be an exponentially distributed random variate. By Property 4.1, we have the following:

$$R = 1 - e^{-T/\theta},$$

$$e^{-T/\theta} = 1 - R,$$

$$\ln(e^{-T/\theta}) = \ln(1 - R),$$

$$\frac{T}{\theta} = \ln(1 - R),$$

$$T = -\theta \ln(1 - R).$$

One final simplification can be made by observing that if $R$ is a uniform zero-one random number, then $(1 - R)$ must also be a uniform zero-one random number; therefore,

$$T = -\theta \ln(R).$$

Figure 4.4 shows a graph of the transformation from $R$ to $T$.

The exponential distribution is convenient to use because the expression for its cumulative distribution function is easy to invert. However, many distributions are not so “well behaved.” A normally distributed random variable (its probability density function has the familiar bell-shaped curve) is an example of a distribution whose cumulative probability distribution function cannot be written in closed form, much less inverted. One method of generating normally distributed random variables is to take advantage of the central limit theorem. This well-known theorem indicates that if a string of $n$ uniform zero-one random numbers is generated, say $R_1, \ldots, R_n$, the random variable $Z$ has approximately a normal distribution with mean zero and a standard deviation of one, where $Z$ is defined by

$$Z = \sqrt{12n \left( \frac{R_1 + \cdots + R_n}{n} - \frac{1}{2} \right)}.$$

Experience has shown that a sample of 12 is sufficient for the resulting value of $Z$ to be adequately close to normal. (This method is given for historical reasons only; the method discussed next is much better.)

A second (and more efficient) method of generating normally distributed random variates is based on a derivation by Box and Muller. They established that if $R_1$ and $R_2$ are two independent uniform zero-one random numbers, then $Z_1$ and $Z_2$ are two independent normally distributed random variates with mean zero and standard deviation of one, where

$$Z_1 = \sqrt{-2 \ln(R_1)} \cos(2\pi R_2),$$

$$Z_2 = \sqrt{-2 \ln(R_1)} \sin(2\pi R_2).$$

There are several other techniques for generating nonuniform random variates when the inverse of the cumulative distribution function cannot

be obtained in a closed form. It is not necessary to know these techniques, however, since either the advanced techniques are buried in the simulation language being used (and thus are transparent to the user) or else an approximation method (discussed below) can be used quite adequately.

4.3.1 Supporting Language Structures

User Definitions. The function that allows for the mapping of the uniform zero-one random number into a general continuous random variate can be defined by the user in a manner analogous to the discrete case. The approximating function (which is the inverse of the cumulative probability distribution function) must be a piecewise, linear, continuous function, as defined in the DEFINITION segment by the statement

\[
\text{myName : DISTRIBUTION(Continuous) = ( sequence of ordered pairs );}
\]

where again an ordered pair has the form \((zx, yy)\), with \(zx\) a probability and \(yy\) a real number. (Although the DISTRIBUTION statement is used here to generate random variates, it is not restricted to such use; \(zx\) need not necessarily refer to probabilities, but can be any real number.) MOR/DS interpolates between the ordered pairs to obtain the continuous function. When the user-defined distribution is to be an inverse of a cumulative probability distribution function, the first ordered pair should have zero for its first component; and the last ordered pair should have one for its first component. The statement

\[
\text{xFunction : DISTRIBUTION(Continuous) = ( (0,1), (0.25,1.5), (0.75,3.5), (1.0,4) );}
\]
defines a function that can be used to generate the random variable \(X\), whose cumulative distribution function is shown in Figure 4.5. As with the discrete case, the DISTRIBUTION statement can be used to generate functions other than those used to generate random variates. The restriction that the \(yy\) values of the ordered pairs need to be increasing is important only for random variate generation. In the statement call xFunction(z), if \(z\) is less than \(zx\) of the first ordered pair \((zx, yy)\) from the DISTRIBUTION definition, the function returns the first \(yy\) value; likewise, if \(z\) is greater than the last \(zx\) value, the function returns the last \(yy\) value.

4.3. CONTINUOUS RANDOM VARIATES

\[
\text{Random number} \rightarrow
\]

Distributed random variate

Figure 4.5: Mapping used to generate the random variable X

System Functions. There are several commonly used distributions for continuous random variables. The first is the continuous uniform random variable. The continuous uniform random variable between two values is one in which intervals of equal length have equal probabilities of occurring so long as the lengths are within the range of the random variable. A system function is available that generates a continuous uniform random variable and is activated by

\[
\text{Cuniform}(zx, yy)
\]

where \(zx\) and \(yy\) are the left and right end points, respectively, of the distribution. In other words the two statements

\[
\text{Cuniform(5, 25.8)}
\]
\[
\text{uFunction(Random)}
\]

will yield equivalent results if uFunction was defined in the DEFINITION segment with the statement

\[
\text{uFunction : DISTRIBUTION(Continuous) = ( (0.5), (1.0,25.8) );}
\]
A second common continuous random variable is defined by the exponential distribution. The syntax for the system function that generates an exponentially distributed random variable is:

\[ \text{Expd}(\bar{x}) \]

where \( \bar{x} \) is the mean value of the random variable (namely, \( \bar{x} \) is the estimate of \( \theta \) as in Equation 4.1).

The normal distribution is a third common random variable with its own system defined generator in MOR/DS. Its syntax is

\[ \text{Normal}(\bar{x},y) \]

where \( \bar{x} \) is the mean value of the random variable and \( y \) is its standard deviation. Other random variate generators that are defined as system functions are Weibull(\( \bar{x},y \)), Gamma(\( \bar{x},y \)), Beta(\( \bar{x},y \)), Erlang(\( \bar{x},k \)), Triangular(\( \bar{x},y,\bar{z} \)), Poisson(\( \bar{x} \)), and Binomial(\( k,y \)). The parameters for the Weibull, Gamma, and Beta functions are shape and scale parameters (see Appendix B. subsection B.2.10). The Erlang’s parameters are the mean and \( y \); \( \text{Erlang}(x,k) \) refers to the sum of \( k \) independent exponential random variables each with mean \( x/k \). The Triangular function has three parameters: the minimum, mode, and maximum values. The Poisson’s parameter is the mean; the Binomial’s parameters are the number of trials and the probability of a success.

To illustrate the use of these system functions, we consider again the microwave salesperson example. Instead of being interested in the profits from 0 customers, suppose we are interested in the profit per day. We assume that the time between customer arrivals has been determined to be exponentially distributed with a mean of a half hour. We now wish to estimate the total profit for an 8-hour day. The following model simulates ten days of sales.

```
PROGRAM "Simulation of ten days of sales"
DEFINITION
  totalProfit = 0;
  cValue = 0;
  aPurchase : DISTRIBUTION(Discrete) = \{ (0.25,30),
  \( (0.75,60), (1.0,75) \);
```

4.4. STATISTICAL ANALYSIS FOR INPUT DATA

**CONTROL**

\[ \text{StopTime} = 10*8; \]
\[ \text{eight hour days} \]

**LOGIC**

\[ \text{ARRIVE} \{ \text{Time} = \text{Expd}(0.5) \}; \]
\[ \text{cValue} = \text{Duniform}(0,1); \]
\[ \text{totalProfit} = \text{totalProfit} + \text{cValue} * \text{aPurchase}(\text{Random}); \]

**DEPART** \{ \};

END.

The only differences between this program and the previous example are that the parameter \text{StopTime} in the CONTROL segment now reflects the ten days, and the \text{ARRIVE} block is changed to reflect the proper random times between customer arrivals.

A full list of system functions is given in Appendix B.2.10. These functions include the standard mathematical functions (like Abs, Sin, etc.) as well as the random variate generators.
- **Suggestion:** to Exercises 4.8 - 4.10.

4.4 Statistical Analysis for Input Data

It is now easy to simulate an arrival process if the probability distribution function of the times between arrivals is known. However, the analyst responsible for a simulation study is often the one responsible for obtaining the estimates for the relevant probability laws, so undertaking a simulation study requires some degree of statistical analysis. (As mentioned previously, the analysis of the output also involves statistics but those aspects will be covered in the next chapter.) There are many excellent texts on statistics; it is not our intent to do more than present a cursory review of a few statistical techniques and indicate how they can be used in simulation analysis. The techniques covered here should be viewed as the minimum statistics required for a simulation study, and the reader is encouraged to review a more thorough treatment in any of the excellent statistical texts available.
4.4.1 Parameter Estimation

It is often possible to make a reasonable hypothesis concerning the underlying probability distribution of interest and, thus, identify the appropriate type of distribution. The remaining major question is the value(s) of the distribution's parameter(s). In an arrival process, for example, if the customer interarrival times are "purely random" and independent of each other, the resulting process yields exponentially distributed interarrival times. Only one parameter, the average interarrival time, would need to be estimated. Or, for another situation, it might be possible to use the central limit theorem and assume that the distribution of interest is a normal distribution that requires estimates for only the mean and standard deviation. 

Let us assume that we are interested in estimating a parameter, say \( \theta \), from a known distribution. (Testing the validity of the assumption regarding the distribution is discussed in Section 4.4.3.) Estimates are usually based on the observation of a random sample, where a random sample of size \( n \) is defined to be a collection of \( n \) independent random variables all from the same distribution function. (Note that in practice when you observe a random sample, you may consider the observations of \( n \) different but identically distributed random variables or, equivalently, you may consider \( n \) independent observations of the same random variable.) Thus, we shall have a collection of \( n \) random variables (or \( n \) independent repetitions of the same random variable), say 

\[
X_1, X_2, \ldots, X_n 
\]

and then an estimator for \( \theta \) would be some function of this random sample. For example, two common estimators are the sample mean, \( \bar{X} \), and the sample variance, \( S^2 \), given by

\[
\bar{X} = \frac{X_1 + X_2 + \cdots + X_n}{n}, \tag{4.2}
\]

\[
S^2 = \frac{(X_1^2 + X_2^2 + \cdots + X_n^2) - n\bar{X}^2}{n - 1}.
\]

(The sample standard deviation is the square root of the sample variance.) Such functions are called statistics, where a statistic is defined to be a function of a random sample that does not depend on any unknown values. For example, assume that \( \theta \) represents the mean value of a distribution (as in the exponential), then we could let \( \bar{X} \) to obtain an estimator for \( \theta \). Thus,

the estimator \( \hat{\theta} \) is in fact a random variable and has a mean and variance. If it is confusing to think of the estimator as a random variable, consider ten people each taking a different random sample (i.e., ten people each observing a sequence of \( n \) independent random observations from the same distribution) and then each taking the arithmetic average of their own sample. It should be clear that each person will most likely have a different value for their individual average, \( \bar{X} \); thus, the estimator \( \hat{\theta} \) is a random variable.

It is often true that the mean value of an estimator is equal to the parameter being estimated. Such an estimator is called unbiased; that is, \( \theta \) is unbiased if \( E[\hat{\theta}] = \theta \). The sample mean (namely, \( \bar{X} \) of Equation 4.2) is always an unbiased estimator for the mean of any distribution (having finite mean) and the sample variance (namely, \( S^2 \) of Equation 4.2) is always an unbiased estimator for the variance of any distribution (having finite variance).

<table>
<thead>
<tr>
<th>Table 4.1: Experimental data from a normally distributed random variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>98</td>
</tr>
<tr>
<td>85</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>95</td>
</tr>
</tbody>
</table>

To illustrate the estimation of the mean and variance, suppose we need to simulate a normally distributed random variable whose mean and variance are to be determined from the data contained in Table 4.1 (assumed to be from a random sample). We have the following calculations:

\[
\sum_{i=1}^{30} x_i = 2,939 \quad \sum_{i=1}^{30} x_i^2 = 292,033
\]

\[
\bar{x} = 97.97 \quad s^2 = 141.69
\]
Thus, if arrivals were to occur with this interarrival time, we would write

\[
\text{ARRIVE \{Time = 0 Max Normal(97.97, 11.9)\}};
\]

or, equivalently,

\[
\text{ARRIVE \{Time = 0 Max Normal(97.97, Sqrt(141.69))\}};
\]

(Note that it is permissible in the modeling syntax to utilize expressions that need to be evaluated. We also used the square root function defined as Sqrt and the binary operator Max that takes the maximum of two numbers; thus, a Max b yields the larger of a and b. The reason for using Max is that it is possible for the normal distribution to generate negative numbers, which would be unacceptable in an ARRIVE block.)

- **Suggestion:** do Exercise 4.11.

### 4.4.2 Estimation of Distributions

When there is no a priori reason for using a particular probability distribution function, then a histogram provides an easy means of specifying the relevant probability law. As discussed in Section 2.1, a histogram records the number of observations for various possible values (or ranges of values) of a random variable. From the histogram, an empirical cumulative probability distribution function can be obtained. First, the numbers in the histogram are converted to probabilities to obtain a probability mass function, then the probabilities are accumulated to obtain the cumulative distribution function. For example, the histogram in Figure 2.1 would be converted to the distribution function as shown in Figure 4.6. Thus, if we are unwilling to assume that the die is fair, we could use the sample histogram as the process distribution. This is frequently the approach taken in simulation studies because it avoids the need to hypothesize a specific distribution, but still replicates the observed data. Thus, a distribution could be defined in the DEFINITION segment by the following statement:

\[
\text{aBiasedDie : DISTRIBUTION(Discrete) = ( (0.1167,1), (0.3,2), (0.45,3), (0.6833,4), (0.85,5), (1,6) );}
\]

and then random variates would be generated in the LOGIC segment using \text{aBiasedDie(Random)}.

![Figure 4.3: The empirical cumulative probability distribution function based on the histogram of Figure 2.1](image)

For a continuous random variable, the empirical cumulative probability distribution function can be formed by aggregating the data into classes as in Table 2.3 or by using each individual point. The maximum number of ordered pairs that can be used in a MOR DS distribution definition is 20. If there are more than 20 points, the data must be aggregated. When all the data points are used, it is easiest to first order the points from the minimum to the maximum. If there are n data points, the empirical cumulative probability distribution function increases by \( \frac{1}{n} \) at every data point. Figure 4.7 illustrates such a distribution function using the data in Table 2.3.

- **Suggestion:** do Exercises 4.12 and 4.17.

### 4.4.3 Goodness-of-Fit Test

Sometimes, there is reason to suggest that the distribution function of interest is of a certain type. Given the suggestion, it is then necessary to verify the suggestion using data. In other words, we hypothesize that
4.4. **STATISTICAL ANALYSIS FOR INPUT DATA**

We again return to the example in Section 2.1 involving rolling a die. We hypothesize that the distribution for the die is a discrete uniform distribution from one to six (namely, it is a fair die). Mathematically, the hypothesis is written as

\[
H_0 : f(k) = \begin{cases} 
\frac{1}{6} & \text{for } k = 1, \ldots, 6, \\
0 & \text{otherwise.}
\end{cases}
\]

The stated hypothesis of interest is called the *null hypothesis* and is denoted by \(H_0\). If we end up rejecting \(H_0\), then \(\alpha\) (which is usually set to be five percent or ten percent) denotes the probability of an error. If we end up accepting \(H_0\), then the probability of error depends on the true underlying distribution function. The Type II error is a complicated function, depending on all possible alternative distributions. For our purposes, it is not important to understand any quantitative descriptions of the Type II error except to know that its maximum value is equal to \(1 - \alpha\). Since we normally make the probability of a Type I error small, the maximum probability of a Type II error is large. When the hypothesis is accepted, we do not have much statistical assurance that a correct decision is made, but we assume it is true for lack of anything better. (As the amount of data used in a test increases, the Type II error decreases; thus, the size of the sample is often chosen to satisfy requirements of a Type I acceptable error. Details for choosing the sample size are omitted, however, since these are outside the scope of this textbook.)

Statistical hypothesis tests are usually based on some established statistic (like the sample mean defined by Equation 4.2) that is calculated from the data (called the *test statistic*) and then compared with some value (called the *critical value*) usually obtained from some statistical tables. The general procedure for most hypothesis tests is as follows:

1. State the null hypothesis explicitly.
2. Decide on which statistic to use (discussed in the paragraphs below).
3. Decide on the desired value for \(\alpha\), called the significance level of the test, and then determine the critical value for the test (usually by looking it up in a table book).
4. Collect the data.
5. Calculate the test statistic.
6. Reject or fail to reject the null hypothesis after comparing the test statistic with the critical value of the statistic.

There are two common statistical tests for hypotheses involving equality of distributions: the chi-square goodness-of-fit test and the Kolmogorov-Smirnov test.

The Chi-Square Test

The chi-square test is performed on aggregated data. In other words, the range for the random variable is divided into \( k \) intervals, and the data consist of the number of observations that fall within each interval. Thus, the data are the same as results from a histogram. The test compares the actual number of observations within each interval with the (theoretically) expected number of observations, and if the difference is "too" large, the hypothesis is rejected.

To state the test more precisely, let \( o_i \) denote the observed number of data points that fall within the \( i^{th} \) interval and let \( e_i \) denote the expected number of data points that should fall within the \( i^{th} \) interval. The test statistic is then given by

\[
\chi^2_{k-1} = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i},
\]

(4.3)

where \( k \) is the number of intervals. The intervals do not have to be of equal length or of equal probability, but they should be large enough so that the expected number of data points in each interval is at least 5; that is, \( e_i \geq 5 \) or \( i = 1, \ldots, k \).

The chi-square random variable has one parameter, its degrees of freedom. A chi-square random variable with \( m \) degrees of freedom is denoted by \( \chi^2_m \). Appendix Table C.3 contains tables of probabilities associated with chi-square random variables of various degrees of freedom. For the goodness-of-fit test involving a distribution whose parameters are assumed known, the degrees of freedom equal the number of intervals minus one. Typically, the value for \( \alpha \) (i.e., the probability of an error when the null hypothesis is rejected) is taken to be five percent or ten percent. The critical value for \( \chi^2_{k-1} \) s found by looking in the appropriate table under \( k - 1 \) degrees of freedom or the specified \( \alpha \). (Sometimes, tables do not list the statistic under \( \alpha \) but instead list it under the cumulative probability, which equals \( 1 - \alpha \). In other words, if the table is for the distribution function \( F \) of random variable \( X \), usually the table will give \( x \) where \( \alpha = P(X > x) = 1 - F(x) \).)

If the test statistic is greater than or equal to the critical value, the null hypothesis is rejected (because the large test statistic showed that the process had more variation than would be expected under the null hypothesis). If the test statistic is less than the critical value, there is not enough variation to reject the null hypothesis, and the hypothesized distribution is assumed to be the correct one.

Again, the data of Table 2.1 are used to illustrate the chi-square goodness-of-fit test. The null hypothesis is the one stated above—that a fair die was used in obtaining the data. Using a significance level (i.e., value for \( \alpha \)) of five percent along with six intervals yields (from Appendix Table C.3) a critical value of \( \chi^2_{critical} = 11.1 \). The calculations for the test statistic are

\[
\chi^2_{test} = \frac{(7 - 10)^2}{10} + \frac{(11 - 10)^2}{10} + \frac{(9 - 10)^2}{10} - \frac{(14 - 10)^2}{10} - \frac{(10 - 10)^2}{10} - \frac{(9 - 10)^2}{10}
\]

\[
= 2.8.
\]

Therefore, since \( 2.8 < 11.1 \), we cannot reject the hypothesis. The hypothesis that the die was fair is accepted.

The use of the chi-square test for a continuous distribution is illustrated using the "spinning dial" experiment along with the data contained in Table 2.3. The hypothesis is that the pointer produces a uniformly distributed random variable. Recognizing that the cumulative distribution for a uniform zero-one random variable is a simple linear function from zero to one, the null hypothesis is written as

\[
H_0: F(s) = \begin{cases} 
0 & \text{for } s < 0, \\
\text{s for } 0 \leq s < 1, \\
1 & \text{for } s \geq 1.
\end{cases}
\]

(4.4)

In our experiment, there were 20 data points, so if four equal sized intervals are used, each interval would be expected to contain 5 data points. Since the test requires the expected number to be at least 5, it is permissible to use the four intervals shown in the histogram. Using a significance level of five percent along with four intervals yields a critical value of \( \chi^2_{critical} = 7.815 \). The calculations for the test statistic are

\[
\chi^2_{test} = \frac{(6 - 5)^2}{5} + \frac{(4 - 5)^2}{5} + \frac{(6 - 5)^2}{5} + \frac{(14 - 5)^2}{5}
\]

\[
= 0.8.
\]
Because $\chi^2_{test} = 0.8 < 7.815$, we cannot reject the hypothesis, and we accept the hypothesis that the value of the pointer has a continuous uniform zero-one distribution.

Occasionally we might want to hypothesize a distribution, but the relevant parameters (e.g., mean, variance, etc.) are not known. For example, suppose we wish to assume that the data in Table 4.1 are from a normal distribution. Our assumption now needs to be verified statistically, so the null hypothesis is formed as

$$H_0 : f(\cdot) \text{ is a normal probability density function.}$$

Before intervals can be decided, the parameters for the normal distribution need to be estimated. The same data used for the chi-square statistic are used to estimate all unknown parameters, but each parameter estimated causes a loss of one degree of freedom in the test statistic. In other words, the chi-square statistic is calculated just as above once the parameters have been estimated; however, the degrees of freedom equal one less than the number of intervals minus the number of parameters estimated from the data.

The data from Table 4.1 yield a mean of 97.97 and a standard deviation of 11.9 (from Equation 4.2). Let the number of intervals equal four, where the intervals are: $(-\infty, 87.97)$, $(87.97, 97.97)$, $(97.97, 107.97)$, and $(107.97, \infty)$. This grouping meets the criterion of at least 5 expected observations per interval. (The determination of the intervals is arbitrary as long as it meets the criterion, and as long as it is done before the data are collected so it is not influenced by the data.) Thus, the critical value for the test at a five percent significance level is $\chi^2_{critical} = 3.841$. From the standard normal tables (see Appendix Table C.1), we have the following for a random variable $X$ that is normal with mean 97.97 and standard deviation 11.9:

$$P\{X < 87.97\} = 0.2005$$
$$P\{87.97 \leq X < 97.97\} = 0.2995$$
$$P\{97.97 \leq X < 107.97\} = 0.2995$$
$$P\{X \geq 107.97\} = 0.2005$$

4.4. Statistical Analysis for Input Data

The value for the expected number of data points in each interval equals the probability of the random variable occurring in the interval times the size of the random sample. This yields

$$e_1 = e_4 = 30 \times 0.2005 = 6.015,$$
$$e_2 = e_3 = 30 \times 0.2995 = 8.985.$$ 

Since each value is greater than or equal to five, the intervals are legitimate for this test. Therefore,

$$\chi^2_{test} = \frac{(5 - 6.015)^2}{5.015} + \frac{(11 - 8.985)^2}{8.985} + \frac{(9 - 8.985)^2}{8.985} + \frac{(5 - 6.015)^2}{5.015} = 0.754.$$ 

The hypothesis that the data are normally distributed is accepted since $\chi^2_{test} < 3.841$. However, again we emphasize that such a test does not "prove" the hypothesis is true; it indicates only that with the available data there is no sufficient evidence to reject the hypothesis.

The test for normality in the above example used only one degree of freedom. This is bothersome because the greater the degrees of freedom the "stronger" the test. (A test is said to be stronger than another if the Type II error for the first test is less than the Type II error for the second.) However, there is a tradeoff in the chi-square test because it is an approximate test, with the condition $e_i \geq 5$ being the limit of its approximation. The test is more accurate if the intervals are large enough so that the expected number of observations is larger than the minimum of five. However, a chi-square goodness-of-fit test with few degrees of freedom is somewhat weak, and the user should not put much confidence in the test results if the null hypothesis was not rejected. If these statements seem more qualitative than quantitative, you are right. At times, statistics is closer to art than to mathematics!

- Suggestion: do Exercises 4.14 and 4.15.

The Kolmogorov-Smirnov Test

One disadvantage of the chi-square test is that it cannot be used with small data sets because of the minimum number of expected observations required per interval. For continuous random variables, the Kolmogorov-Smirnov (K-S) test is an alternative approach that can be used with small data sets as well as with large ones. The statistic is based on a comparison:
between the hypothesized cumulative probability distribution function and the empirical cumulative probability distribution function. More precisely, let \( F_0 \) denote the hypothesized cumulative probability distribution function, and let \( \hat{F}_n \) denote the empirical cumulative probability distribution function based on a sample size of \( n \). Then let \( D_n \) be a random variable denoting the maximum difference in absolute value between the two distributions; that is,

\[
D_n = \max_x |F_0(x) - \hat{F}_n(x)|.
\]

(4.5)

The critical value to be used for rejecting or accepting the hypothesis is found in the appropriate tables according to the significance level and sample size.

4.4. Statistical Analysis for Input Data

Data set (Equation 4.4). The critical value (from Appendix Table C.4) for the statistic is \( D_{\text{critical}} = 0.294 \) for \( \alpha = 0.05 \). Using the graph in Figure 4.8, it is seen that \( D_{\text{test}} = 0.10 \). It should be noted that the supremum (actually, the maximum) difference occurs either at the left endpoint or the right endpoint of each line segment so both ends need to be checked in calculating \( D_{\text{test}} \). Since \( D_{\text{test}} < D_{\text{critical}} \), the hypothesis is not rejected.

- Suggestion: do Exercise 4.16.

4.4.4 Testing for Randomness

A final statistical test that is sometimes helpful is the runs test, which is a test for randomness. Usually data are assumed to be random, but if the analyst did not ensure a random collection, a test for randomness would be appropriate. It also might be worthwhile at times to check the random number generator of a simulation system. Even though the random number generator might have been previously tested, using a different initial random number seed could produce a sequence of numbers that is not acceptable.

Both the chi-square goodness-of-fit test and the K-S test can be used to test the hypothesis that the random numbers are from a uniform zero-one distribution (Equation 4.4 gives the uniform zero-one hypothesis); however, it is also important that each number be independent from all other numbers.

Conceptually, the runs test determines randomness based on the number of times a sequence of numbers alternates between high values and low values. For example, consider the following sequence: 0.3, 0.58, 0.86, 0.24, 0.78, 0.91, 0.62, 0.41, 0.63, 0.96, 0.45, 0.32, 0.2, 0.51, 0.79, 0.45, 0.77, 0.22, 0.83, 0.65, 0.28, 0.03. The median value of this sequence is 0.5. (To determine the median of a data set, order the data from lowest to highest value. If the number of data points is an odd number, the middle value is the median; if the number of data points is an even number, then the median is the number halfway between the two middle values.) The data are now converted to minuses and pluses depending on whether the data are less than the median or greater than the median value, respectively. Thus, the above sequence becomes

\[-/++/++/++/+-+----++/+-+/-++/--- .
\]

The runs for this sequence are indicated by the "/", where a run is defined to be one or more of the same symbols that are contiguous. The quantity \( N \) denotes the number of data points below the median, and the quantity \( R \) denotes the number of runs. For the above sequence, \( N = 11 \) and \( R = 13 \).
If $R$ is extremely large, the numbers alternate too much, which implies that high values usually follow low values. Therefore, we would be forced to conclude that there are dependencies among the numbers. If the number of runs is too small, we are forced to conclude that low numbers usually follow low numbers and high numbers usually follow high numbers and again conclude dependencies must exist. A quantitative measure of “too high” or “too low” can be obtained by the fact that $R$ approximately follows a normal distribution when $N > 10$, with

$$E[R] = N + 1$$

$$\text{var}(R) = \frac{N(N - 1)}{2N - 1}.$$  

The statistic used is the (two-tailed) $z$ statistic based on the standard normal tables. If a significance level of $\alpha$ is used, the critical value is the $z$ statistic from the standard normal tables, using $\frac{\alpha}{2}$ because of the two-tailed test (that is, errors can occur if the $z$ statistic is too large or too small). The test statistic is $(R - E[R])/\sqrt{\text{var}(R)}$; namely,

$$(R - N - 1)/\sqrt{\frac{N(N - 1)}{2N - 1}}.$$  (4.6)

For the above sequence, $z_{\text{critical}} = 1.96$ for $\alpha = 0.05$, and

$$z_{\text{test}} = \frac{13 - 10}{\sqrt{\frac{110}{21}}} = 1.31.$$  

The sequence is accepted as random since $1.31 < 1.96$.

The runs statistic is based on the number of points above the median being equal to the number of points below the median. If this is not the case, delete the median value so that the sequence satisfies this assumption. For example, if the sequence is 7, 6, 7, 3, 8, the median is 7 but there are two values below and only one value above. Therefore, we would delete the first 7 so that the remaining numbers satisfy the assumption of an equal number above and below the median, which now equals 6.5.

- **Suggestion: do Exercise 4.17.**

### 4.5 Exercises

4.1. Before running the program listed at the end of Section 4.1, draw a histogram typical of what you would expect the program to produce.

4.5. **EXERCISES**

Run the program twice as i: appears in the text. Run the program twice using the Randomize = on statement. Run the program twice using a fixed initial random number seed. Compare the output from the six runs and comment on any differences.

4.2. Generate an arrival stream such that entities arrive at times 300, 400, and 500.

4.3. Generate an arrival stream such that batches of three entities each arrive at times 300, 400, 500, 700, 750, and 800. (Hint: A program can contain more than one ARRIVE block.)

4.4. Write a program to simulate rolling a single die 1000 times using the DISTRIBUTION statement. Generate a histogram with six cells and compare your results with the results from the START DSE program. (Note that you can use StopTime = 1000, and ARRIVE (Time = 1), and then insert values directly into the histogram with an assignment statement, such as myHistogram = myDistribution(Random).)

4.5. Estimate the probabilities of rolling a 7 with two dice by simulating the roll of two independent dice via the Uniform(1,6) function. Make several runs of different lengths (i.e., runs with a varying number of simulated rolls) and comment on the accuracy of the estimates obtained from the different runs.

4.6. Boxes are packaged with a varying number of radios: 25 percent of the time a box contains 3 radios, 60 percent of the time a box contains 4 radios, and 15 percent of the time a box contains 5 radios. Generate an arrival stream of entities where the individual radios are the entities. Assume that one box is packaged every 4 minutes, and estimate the expected number of radios packaged every hour.

4.7. When a female boll weevil lays eggs, 75 percent of the time a single egg is laid in a single cotton flower bud; 20 percent of the time two eggs are laid within a single bud; 4 percent of the time three eggs are laid within a single bud; and 1 percent of the time four eggs are laid in a single bud. Whenever one or more eggs are laid in a flower bud, that bud is absized (dropped off) by the plant. Use simulation to estimate how many flower buds will be lost due to one female boll weevil if the weevil lays 1000 eggs. (Let “time” units be eggs and let entities be dropped buds.)
4.8. Run the program at the end of Section 4.3 ten times to obtain an estimate of the daily profit from the microwave sales. From the ten runs, draw a histogram showing the variability in the daily profit.

4.9. The program at the end of Section 4.3 assumes exponentially distributed interarrival times. Change the program to reflect uniformly distributed interarrival times with the same mean as before (i.e., use uniform zero-one random numbers) and run the program 10 times. Draw a histogram for daily profits and comment on the difference between the exponential and uniform assumptions.

4.10. Given the sequence of three uniform random numbers, 0.15, 0.57, 0.74, generate (by hand) the corresponding random sequences from the specified distributions:
   (a) From the exponential with a mean of 4 time units
   (b) From the discrete uniform distribution varying between 1 and 4
   (c) From the discrete random variable, with the probability it is equal to 1, 2, 3, 4 being 0.3, 0.2, 0.1, 0.4, respectively.

4.11. Let be a normally distributed random variable with mean 100 and standard deviation 10. Using the system function Normal(100,10), generate 10 random variables and compute an estimate for the mean and variance. Compare your estimates to the actual mean and variance values of 100. Repeat the calculations with 1000 variates. (Hint: Use an assignment statement for your calculations.)

4.12. Simulate rolling a fair die ten times. Use the results to build an empirical cumulative probability distribution function. Simulate rolling a die 1000 times, except use the probabilities for the die from your empirical distribution function. Compare your results with the results of Exercise 4.4. Comment on any discrepancies.

4.13. Simulate 10 observations of a normally distributed random variable with mean 100 and standard deviation 10. (Since we have not yet introduced the PRINT statement, you can simply run the program 10 times and record the values of the 10 generated random variables.) Determine an empirical distribution function, and then generate 1000 observations for a random variable from this empirical distribution function. Calculate the sample mean from the 1000 observations. Comment on why your calculated numbers are likely to be different from the original mean of 100 used to generate the first 10 numbers.

4.14. Let be a random variable such that
   \[ P(X = 1) = 0.15 \]
   \[ P(X = 2) = 0.20 \]
   \[ P(X = 3) = 0.25 \]
   \[ P(X = 4) = 0.25 \]
   \[ P(X = 5) = 0.15 \].

Generate 50 observations of this random variable and then perform a chi-square goodness-of-fit test where the null hypothesis is the distribution used to generate the observations. Then perform a chi-square goodness-of-fit test using an \( \alpha \) error of 5 percent, where the null hypothesis is the discrete uniform distribution from 1 to 5 (in other words, perform a test for which you know that the null hypothesis is false). Repeat the above, except use 1000 observations. Comment on your results. (Assume you have no \textit{a priori} knowledge of the mean.)

4.15. Generate 100 observations of an exponentially distributed random variable with a mean of 5. Use a chi-square goodness-of-fit test (\( \alpha = 0.05 \)) to determine if you can statistically accept the fact that the generated data came from an exponential distribution. (Assume you have no \textit{a priori} knowledge of the mean.)

4.16. Generate 10 observations of an exponentially distributed random variable with a mean of 5. Use the Kolmogorov-Smirnov test to determine if the simulator generated the variates as you requested (\( \alpha = 0.05 \)).

4.17. Statistically test if the program is generating independent uniform zero-one random numbers using an \( \alpha \) of 5 percent. Perform the test twice: the first time use the default initial random number seed, the second time use an initial random number seed of your choosing. (This problem should only be done by students who have had previous programming experience, because it uses the If/Then/Else structure, which is not discussed until the next chapter.)
Chapter 5

Fundamentals of Simulation Modeling

This chapter provides an overview of the concepts involved in building and analyzing discrete simulation models. We begin with a discussion of the model building process and introduce the language syntax needed for constructing simple computer simulations and collecting statistical data generated from the simulation. Next, the concepts needed for understanding the internal operating procedures for block-oriented languages are discussed. The chapter closes with a presentation of several programming ideas useful in modeling.

Developing a simulation program for modeling a particular system can be segmented into three main tasks. The first task is to identify the parameters characterizing the system to be modeled and to define the relevant physical aspects of the system. The second task is to construct the necessary logic that actually allows the model to mimic the system behavior. The final task is the bookkeeping or control function, in which the decisions are recorded as to how the overall model is to be controlled (when to start and stop the program), and how the simulation results are to be reported. In this chapter, we discuss each of these aspects of model-building and illustrate these concepts with some simple models. The next chapter deals with the analysis of results, and Chapter 7 deals with more complex models. (The distinction between a simple model and a complex model is the number of concepts needed for programming the models: GATEs, user-controlled CHAINS, and general programming structures such as the REPEAT, WHILE, and DOCASES statements are dealt with in Chapter 7.)

5.1. THE DEFINITION PHASE

5.1 The Definition Phase

The first task of program development is always the definition phase. It is important to be specific in defining the quantities of interest and in defining those aspects of the system that must be included in the model. It is at this initial point that communication among all participants in the modeling study is extremely important. The goals of the study should have already been explicitly stated and the various random phenomena identified so that data can be collected to estimate all appropriate probability laws. In the program definition phase, the estimated probability laws are defined.

Most discrete simulation studies involve the flow of entities (customers, machine parts, tools, etc.) through the system. Usually the entities flow or proceed through various resources or facilities within the system, and, in fact, often one of the purposes of a simulation study is to determine the effect of limited resources or finite capacity on the system. Associated with a resource is usually a queue for entities that cannot currently be handled by the resource. For example, the study of a bank might involve analyzing the flow of customers, where each teller window open to the public might be considered a resource. A teller can handle only one customer at a time; if a teller is busy with one customer, then other customers who ask for the service of that teller must wait in the queue associated with that teller until the teller is free. The rules by which entities go from the queue to the resource are also important and must be specified. Most banks, for example, would use a FIFO (first-in first-out) rule; however, other situations, like customers shopping for dated goods (e.g., milk or photographic film), might call for a LIFO (last-in first-out) rule. More complicated queueing disciplines are also common in other circumstances such as computer systems.

The goals of a simulation study are often closely connected to specific resources. Normally, there are general system properties that need to be addressed, but also there are specific resource properties for which information is needed. Such things as flow times through a resource (or a series of resources) and queue lengths usually are desired quantities. The specific statistics needed to fulfill the simulation goals would be specified during the definition phase of model development.

5.1.1 Supporting Language Structures

Again, we emphasize that the block types described here and the activities of model building discussed are general concepts that apply to modeling
in almost any block-oriented simulation language. Although the program implementation is through the MOR/DS language, these concepts must be learned for any discrete simulation model-building activity, regardless of the language utilized to develop the resulting simulation model.

As has been pointed out, the MOR/DS language structure is designed to reflect the natural model-building steps by separating the program into a DEFINITION segment, a LOGIC segment, and a CONTROL segment. The DEFINITION segment appears first and is used for defining global variables, arrays, functions, and the language structures for resources, queues, histograms, statistics, gates, and entity (local) attributes. The RESOURCE, QUEUE, ATTRIBUTES, and LABELS statements (when used) are part of the DEFINITION segment of the program. They are described below so they can be used to build some simple models.

The RESOURCE statement is the means by which the user defines the physical resources of the simulation. The statement has the form

```
myName : RESOURCE = { Capacity = xx, Queue = yy }; 
```

where both keywords, Capacity and Queue, are optional. (Note: that the brackets are required in MOR/DS commands even if no keywords are used.) Capacity is the maximum number of individual entities that can utilize the resource at one time. If the keyword Capacity is not present, then the default capacity of one is used. A default queue of unlimited capacity and a FIFO order discipline is automatically assigned to the resource. This default queue has no name and is unique to the resource. Unless other options are specified in the RESOURCE definition through the Queue option, waiting customers will be kept in this default waiting line.

There are several system variables associated with each resource accessible by the user so that the resource status can be determined dynamically during program execution. We will discuss only three of them here; the others are given in Appendix B. The variable Capacity[name] contains the capacity of the resource identified by name. The variable Available[name] contains the current number of available resource units. The variable Used[name] contains the current number of units being used.

If any discipline other than FIFO is needed or if a finite queue capacity is required by a resource, then a user-defined queue is necessary and is defined by statements of the form:

```
qName : QUEUE = { Capacity = xx, Disciplene = yy, Histogram = hh, StatOnOff = On/Off }; 
```

where none of the keywords are required. (The brackets are required even if no keywords are used.) Capacity specifies the maximum number of entities that may be kept in the queue at any one time, with a default of 32,000 if no number is specified. The Discipline is used for determining the order in which entities line up in the queue. When resource space becomes free, entities from the front of the list are selected for the next utilization of the resource. The queue discipline must be one of FIFO, LIFO, CHOICE(expression). The FIFO priority scheme is the default value. The CHOICE priority involves a form of priorities based on the evaluation of expression, lower values having higher priorities. It is often the case that expression simply refers to an attribute variable; however, it is permissible to use an expression for the CHOICE parameter. For example, if priority1 and priority2 are defined as attribute variables (see ATTRIBUTES on the next page), then CHOICE(priority1+2*priority2) is proper. Priorities can also be used for preempting service of a facility, which is discussed in Chapter 7. A histogram of the time entities spend in the queue can be collected automatically by the use of the Histogram option, where the keyword is set equal to the name of a defined histogram. The histogram must have been previously defined in the DEFINITION segment using a HISTOGRAM statement. (The histogram only includes flow times for entities that have completely passed through the queue, whereas the default statistics also include data from entities that currently reside in the queue. Therefore, if the queue is not empty when the system stops, the mean from the histogram may not equal the mean given by the normal queue statistics.) The keyword StatOnOff is used to turn off the listing of the statistics information associated with the QUEUE. The keyword is optional, and if it is not given it is assumed to be StatOnOff = On.

Consider, for example, the queue definition:

```
qEl : QUEUE = { Capacity = 2 }; 
```

This statement specifies that the queue called qEl has a capacity of two, will use the default FIFO discipline, and will have its relevant statistics printed. The capacity of two implies that whenever two entities are waiting in qEl, no other entities can join that queue. The two waiting entities are kept in the order of their arrival at the queue since the default FIFO discipline is used. Keep in mind that these statements are only definitions; they do not determine how entities obtain and release resource units, or how and why they are queued and subsequently taken out of the queue. The queues defined by the QUEUE definition statement are related to specific resources
through the Queue option of the RESOURCE definition statement. The system variables Capacity[name], Available[name], and Used[name] are associated with each queue and are available to the user with the same meaning as the analogous resource variables. Obtaining information for the default queue associated with a specific resource requires slightly different notation because no special name has been associated with this queue. The rule is Capacity[resourceName:Queue]. (The system variables Available and Used are analogous to Capacity.)

The ATTRIBUTES statement establishes those variables (and their names) associated with each entity created by the ARRIVE block. In other words, each entity created in the simulation model will have a local variables list attached to it, as defined inside the brackets of the ATTRIBUTES statement and given the initial values specified. (All variables in MOR/DS, except certain dummy variables, must appear in the DEFINITION segment and must be given an initial value.) No label or name is needed for the attribute definition statement because every entity carries the same attribute names, but those attributes have values unique to that entity. An example is

```csharp
ATTRIBUTES = { dollars = 0, date = 0, pri = 0 }; 
```

where dollars, date, and pri are to be assigned values unique to each entity. For example, pri might contain the assigned priority for the entity and thus would be identified by using CHOICE(pri) in the QUEUE definition. The attribute list can be complex in the sense that arrays as well as variables can be used. The ARRAY statement is covered in Chapter 7.

Situations often arise where it is convenient to use some form of a branch statement within the LOGIC segment of the program. In such a case, labels are needed to identify the destination for the branch. All labels used must be first defined in the DEFINITION segment through the use of a LABELS statement. The syntax is

```csharp
LABELS = { label1, label2, label3 }; 
```

where as many labels as desired may be used in the statement and it is permissible to use more than one LABELS statement.

We encountered examples in previous chapters of other statements that belong in the DEFINITION segment of the program. The HISTOGRAM and DISTRIBUTION statements were defined previously, so they will not be repeated here. Global real variables are defined in the DEFINITION segment by simply writing an assignment statement setting them equal to their initial value. Two cautions might be helpful at this point to avoid compiler errors:

1. Do not begin a fraction with a decimal, 0.56 is permitted but .56 will cause an error message.
2. No name can be used on the right-hand-side of a definition if it has not been defined and initialized.

Consider the following:

```csharp
DEFINITION
  chair = 10;
  room : RESOURCE = { Capacity = chairs };
```

where chairs is a global variable and room is the name of a resource. The above use of chairs in the RESOURCE statement is permissible because chairs has already been defined before its utilization; however, an error would occur if the order of the two statements had been reversed.

Variables and labels follow the same naming convention. They must begin with a letter followed by up to 14 alphanumeric characters and may then be followed by a period and a numerical value. For example, the following are all permissible names: x, xName, xName123, xName.3, xName5again3.23.

### 5.1.2 Example

To illustrate the concepts discussed in this chapter, two example problems are formulated, and programs to model these examples are developed as we proceed through the chapter.

#### Widget Processing Example

We return to the widget example described in Section 2.3. As you recall, a widget arrives to the processing center every 12 minutes. Seventy-five percent of the widgets are suitable for processing, and those pass one-at-a-time through a processing machine taking 1.5, 15, 16, or 17 minutes with probabilities of 12.5%, 12.5%, 50%, and 25%, respectively. The processing machine is the only resource for this situation, and since there is unlimited waiting room and service is on a first-in first-out basis, we can use the default queue. The DEFINITION segment of the simulation program will therefore need to contain the RESOURCE definition statement. The only other statement required is a DISTRIBUTION statement to create the distribution needed for generating the service times. The DEFINITION segment is:
5.1. The Definition Phase

Three other jobs waiting. Thus the queue for jobs waiting to use the computer has a capacity of three; however, the queue for research jobs has a capacity of one. The operating procedure for the system is essentially a first-in first-out service discipline. However, if only one terminal is being used (exactly one student is on the system), then students are allowed to bypass their research project colleagues (professors) and use the idle terminal.

Due to complaints from both the student users and those involved in research projects, the department head is considering an upgrade of the system. The department head would like to order enough terminals (together with a corresponding upgrade in CPU) to promise that 80% of the time any user would be able to complete his or her job in two hours. Our goal is to design a simulation to advise the department head how many terminals are needed to meet this goal. The first task is to write a simulation program that mimics the current computer system.

We present the DEFINITION segment for that program and discuss the various statements in the segment. The remaining segments will be presented and discussed later in this chapter as the various aspects of program development are covered.

---

An academic department at State University has a minicomputer system with two remote terminals. These two remote terminals are adequate for most student programs. However, there are some faculty research projects that require the dedicated use of the entire minicomputer, which implies that both remote terminals are in use by the same job. After observing the arrival pattern of student jobs, it is determined that the time between arrivals is random and is governed by the exponential distribution with an average of three student users arriving every hour. (Because of the dedication of the students, the arrival pattern stays the same 24 hours a day.) The time between the arrival of requests to run the research jobs is a continuous uniform random variable ranging from 30 minutes to 90 minutes. Analysis of the student-usage time requirements indicates that their usage time can be grouped into three categories: 25% of the time the jobs last 20 minutes, 62.5% of the time the jobs last 40 minutes, and 12.5% of the time the jobs last one hour. Research project run-times are more variable and appear to be exponentially distributed with a mean time of 45 minutes.

Individuals (either students or faculty) do not wait if they arrive while a research program is either in progress or waiting to use the computer; furthermore, arriving students with jobs will not wait if there are already
5.2 The Logic Flow Development Phase

Discrete simulations typically trace the movement of entities through the system. The influence of resources on the system is through their utilization by the entities as they seize and eventually release them. For example, as customers queue in front of a bank teller, we would say from the teller's point of view that the teller serves the various customers one at a time according to the time demands of the different customers. From the customer's (i.e., entity's) point of view, we would say that the entity seizes the teller (i.e., the resource) as the teller becomes available. Block-oriented simulations operate from the entity point of view; thus, the utilization of resource units is through seize and release procedures. In actual practice, entities seize and release the resource. When the resource is released, a number of resource units is freed for possible allocation to other entities. Conceptually, the resource becomes active at this point in attempting to find waiting entities to utilize fully the idle resource capacity. This is accomplished by searching the resource queue list from front to back for the first entity that can fit into the available space. Then, a second entity is looked for, and the process repeats until the resource is fully utilized or the resource queue list has been completely searched.

After entities have seized a resource, there are several options available depending on the dynamics of the system being simulated. The most common activity is simply for the entity to wait until the requested service is complete.

For many simulations, the logic development phase of the program involves the proper ordering of arriving entities, their seizing and releasing of resources, and their waiting for services. The most difficult part of this aspect, especially for experienced programmers, is reorienting the entity point of view. Discrete simulations are not based strictly on the sequential execution of statements, but on the simultaneous movement of entities through the system (see Section 3.5). Computations of those quantities needed for output analysis must be integrated within the flow of the entities.

5.2.1 Supporting Language Structures

Entities enter the simulation through an ARRIVE block (see Section 4.1.1) that creates the physical system entity for a single customer and schedules its time of arrival into the system. A model can contain more than one ARRIVE block, making it possible to have multiple input streams of entities, each with different characteristics (but identical attribute lists). Entities request a resource with a SEIZE block, which has syntax

```
SEIZE { Name = name, Units = xz, Recess = label1, Qecess = label2 ;
```

where only the first keyword, Name, is required. Name identifies the particular resource (as named in the DEFINITION segment) being seized, and Units is the number of units of the resource that the entity requests.
If the keyword Units is omitted, then it is automatically set to one. (For example, assume that a resource has been defined that represents a parking lot with spaces for 100 cars. The entities for the model refer to both cars and trucks, but when a truck comes into the parking lot, it takes up two spaces. Therefore, the RESOURCE statement will set Capacity = 100 and the SEIZE statement would have Units = 1 for a car and Units = 2 for a truck.) The keyword Rexcess, if present, specifies a statement label to which the requesting entity is transferred if the resource is busy. In this case, no waiting line in front of the resource is allowed to form by that SEIZE block. The Rexcess option allows entities to do things other than simply join the queue associated with the resource. The Qecess keyword, if present, specifies where the entity should go if the named queue (named in the RESOURCE definition statement) is already filled to capacity. It is considered a modeling error, found during program execution, if the queue capacity is exceeded but no alternative branch location is specified. Most uses of the SEIZE block do not require the keyword Rexcess, and thus a queue associated with the resource is usually formed. Assuming the absence of a Rexcess label, entities that are unsuccessful in their attempt to seize the specified number of resource units are placed in a resource queue.

As mentioned in the previous section, every resource has associated with it a default queue. If a queue structure other than an infinite capacity FIFO waiting discipline is desired, the alternate queue must be defined with the QUEUE definition statement.

When an entity is finished with a resource, it must explicitly give back the units being used. The entity does this through the RELEASE block, which has the form

\[ \text{RELEASE \{ Name = name, Units = zz \}}; \]

where Name is required and both parameters have the same meaning as in the SEIZE block. The default number of units requested is one.

An entity that has seized more than one unit of the resource can release the seized units in phases with more than one RELEASE block. The ultimate number of units released by the entity must eventually equal the number seized or this capacity difference will be lost to later entities.

After an entity releases resource units, the resource attempts to fill its available space from entities in the queue. This process is initiated by the RELEASE block. The ordering of the entities within the queue is according to the queueing discipline specified in the QUEUE statement (or according to the FIFO rule if the default queue was used). If an entity requires more resource units than are available, the resource will bypass that entity and continue to search the queue for an entity that demands fewer units. Other possible resource-filling procedures are illustrated in Section 5.6 and Chapter 7.

The most common situation that occurs while an entity has control of a resource is for that entity to merely wait until some service by the resource is complete (like a teller processing the customer’s deposit). This service time is simulated by the WAIT block, which has the form

\[ \text{WAIT \{ Time = zz \}}; \]

where Time is required and specifies the length of time that the entity must wait until it can proceed to the next statement. The keyword Time can be set equal to a number, an expression, or a function. Examples of legitimate statements are

\[ \text{WAIT \{ Time = 10 \}}; \]
\[ \text{WAIT \{ Time = 10 + 3 * Expd(Random) \}}; \]
\[ \text{WAIT \{ Time = Sqrt(x.Value(Random)) \}}; \]

In the last statement, x.Value is the name of a function defined in the DEFINITION segment through the use of a DISTRIBUTION statement.

To illustrate these statements, we present a program to simulate cars arriving at a service station that has only one gas pump and room for only two other cars waiting. Cars arrive randomly, with the time between arrivals being an exponentially distributed random variable having a mean of five minutes. Each car takes exactly five minutes at the pump. Cars that arrive while there are three cars already there (one at the pump and two waiting) do not enter the system; they simply travel to the next service station. A program to simulate this system is as follows:

\[ \text{PROGRAM \ "first service station example"} \]
\[ \text{DEFINITION} \]
\[ \text{waiting \ : \ QUEUE \{ Capacity = 2 \}}; \]
\[ \text{pump \ : \ RESOURCE \{ Capacity = 1, Queue = waiting \}}; \]
\[ \text{LABELS \{ noEntry \}}; \]
\[ \text{CONTROL} \]
\[ \text{StopTime = 100;} \]
\[ \text{LOGIC} \]
\[ \text{ARRIVE \{ Time = Expd(5) \}}; \]
\[ \text{SEIZE \{ Name = pump, Units = 1, Qecess = noEntry \}}; \]
3.2. THE LOGIC FLOW DEVELOPMENT PHASE

which must be either true or false. (Boolean expressions can also be used in building more complex mathematical expressions. Our convention for boolean expressions is that a true statement is equivalent to 1, and a false statement is equivalent to 0.) A more general form of the IF statement is

IF boolean expression THEN GOTO true_label
ELSE GOTO false_label;

For this case, if the logical expression is false the entity is sent to the statement identified by false_label. (Note that the statement ends after the ELSE portion and thus the semicolon appears after the ELSE portion.)

The IF statement can actually be used to conditionally execute any statement or even a sequence of statements. Thus, a more general form of the IF statement syntax is

IF boolean expression THEN statement 1
ELSE statement 2;

where, as before, the ELSE portion can be omitted. The GOTO label is just one possibility for the statement. Two other examples are:

IF Random <= 0.5 THEN WAIT {Time = Expd(2.0)}
ELSE WAIT {Time = 4.0};

IF Duniform(1,3) <= 3 THEN dollars = dollars + 5;

When the logical flow of the entity is not changed by the IF statement, then after executing the associated true or false boolean expression, the entity continues its flow through to the next statement in sequence.

The modeler might want to perform more than a single statement on one of the THEN or ELSE branches of the IF statement. This can be accomplished, but the language compiler must be told to expect more than a single statement. Taking the lead from the Pascal language, we allow multiple statements in all our language constructs as long as they are enclosed by a begin and end pair. For example, consider

IF Random <= 0.5 THEN
begin
  WAIT {Time = 5.0};
  data = Duniform(1,3);
end
ELSE GOTO false_label;
Here if the uniform random number is less than or equal to 0.5, then the WAIT block and assignment statement are executed and the entity proceeds to the statement after the IF statement; otherwise, the entity is branched to the statement labeled false.label. Note that no semicolon follows the end, for to do so would tell the compiler that no ELSE clause is coming; that is, a semicolon cannot precede an ELSE.

5.2.2 Examples Continued

We return to the examples of Section 5.1.2. The LOGIC segment of the example programs are constructed using the blocks defined in this section. Again, we build on the previous work using the easy example first.

Widget Processing Example

The LOGIC segment is mainly the description of the flow of entities (widgets) through the system. As widgets arrive, 25% of them will immediately depart while the remaining 75% must be processed. With the LOGIC segment included, the program becomes:

PROGRAM "Widget example -- time units are minutes"

DEFINITION

processor : RESOURCE = { Capacity = 1 } ;
service : DISTRIBUTION(Discrete) = ( (0.125,14),(0.25,5),
(0.75,16),(1.0,17) );
LABELS = { bad } ;

CONTROL

StopTime = 60*24*5; "Simulates five days"

LOGIC

ARRIVE { Time = 12 } ;
IF Random < 0.25 THEN GOTO bad ;
SEIZE { Name = processor, Units = 1 } ;
WAIT { Time = service(Random) } ;
RELEASE { Name = processor, Units = 1 } ;
DEPART { } ;
bad:
DEPART { } ;
END.

In the IF statement, we used the GOTO block when we could have used the DEPART block directly as will be seen in the next example. The use of the GOTO block illustrates the use of a statement label (namely, the last DEPART block is labeled bad with a colon appearing immediately after the label). The reason that two DEPART blocks are used is to keep the counts of the good and bad widgets separate from the counts of the bad widgets. When we are ready to look at the output, the block counts of the separate DEPART blocks will provide the number of good widgets and the number of bad widgets.

Computer System Example

The flow of entities in this example is significantly more complex than for the widgets. The following is the full program for this example.

PROGRAM

"student and research jobs on two terminals"
"the unit of time is minutes"

DEFINITION

studentQ : QUEUE = { Capacity = 3 } ;
researchQ : RESOURCE = { Capacity = 1 } ;
computer : RESOURCE = { Capacity = 2, Queue = studentQ } ;
jobTime : DISTRIBUTION(Discrete) = ( (0.25,20), (0.875,40),
(1.0,60) ) ;
sJobHist : HISTOGRAM = { Cells = 10, MinValue = 20,
MaxValue = 120 } ;
resHist : HISTOGRAM = { Cells = 10, MinValue = 20,
MaxValue = 120 } ;

ATTRIBUTES = { type = 1, startTime = 0 } ;
LABELS = {quit1, quit2, finished} ;

CONTROL

StopTime = 60*24*5; "Simulates five days"

LOGIC

ARRIVE { Time = Expd(20) } ; "3 per hour"
startTime = ClockTime ;
IF Used[researchQ] > 0 THEN DEPART { } ;
SEIZE { Name = computer, Units = type } ;
Qexcess = finished
WAIT { Time = jobTime(Random) } ;
RELEASE { Name = computer, Units = type } ;
sJobHist = ClockTime - startTime ;
finished: DEPART { } ;
CHAPTER 5. FUNDAMENTALS OF SIMULATION MODELING

3. THE CONTROL PHASE

"begin flow of research jobs"
ARRIVE { Time = Cuniform(30,90) ;
type = 2; startTime = ClockTime ;
SEIZE { Name = researchQ, Rexcess = quit2 };
SEIZE { Name = computer, Units = type, Qexcess = quit1 };
WAIT { Time = Expd(45) };
RELEASE { Name = computer, Units = type };
quit1: RELEASE { Name = researchQ };
quit2: resHist = ClockTime - startTime;
DEPART { };
END.

The RESOURCE labeled researchQ was used to limit the number of research jobs waiting to one. The keyword Rexcess in the SEIZE block for researchQ indicates that research jobs that are unable to obtain researchQ do not wait, but depart from the system. Note also that the IF statement used the DEPART block directly instead of branching to a DEPART block, as was done in the widget example. This is the recommended practice (i.e., to use IF instead of GOTO statements) because models are less fragmented and disjointed; hence, they are easier to read. The IF statement also uses the system-defined variable Used[researchQ], which contains the number of units in the resource named researchQ.

5.3 The Control Phase

Decisions regarding the length of the simulation run and the amount of output are the final steps in program development. These decisions are more difficult than they might seem at first. The reason for the difficulty is that there are often no quantitative guides for making these decisions. For example, it is quite common for simulation studies to produce massive amounts of output, most of which are never utilized. Too much output tends to confuse the analysis instead of clarifying the stated goal.

A simulation is a statistical experiment and, thus, there should always be more than one run, just as a statistical experiment would never be performed to collect just one data point. (The statistical issues dealing with estimations and run lengths are discussed in the next chapter.) Very often a simulation is performed to determine the long-run behavior of the system under study. For example, a bank might be considering switching from four to ten electronic teller machines (a 24-hour teller service popular in many states) and is interested in their utilization. Therefore, a simulation is written to estimate the system characteristics for ten such electronic tellers. If the resulting simulation run-time is too short, then the output might simply reflect a bias due to the initial switch over: from four to ten tellers, instead of the actual long-run behavior.

There are now two related decisions to be made. One decision is the length of the simulation run to reflect long-term system behavior, and the second decision is the number of replications that need to be made to determine the inherent statistical variation in the final estimates. The simulation run length decision is probably best made after observing a graph of the output. The visual impression of the output is usually much better and easier to use than any quantitative measure.

Initial conditions usually bias a simulation study of long-run behavior and, therefore, most simulation languages allow for a reinitialization of the collected statistics after a given period of time while leaving the current entities at their observed locations. Again the decision as to the length of the initial period is difficult, because no easy quantitative guides for such decisions are available. The best method is to make a trial run and observe a plot of the output to acquire a "feel" for the system behavior.

5.3.1 Supporting Language Structures

The CONTROL segment involves directives that set several system parameters to control the program execution. In MOR/DS, we have located the CONTROL segment of the program between the DEFINITION and the LOGIC segments. Technically, in the simulation language design, we could have placed this segment anywhere within the program structure. However, once the logic flow of the program has been written, most of the changes to the program occur in the CONTROL and DEFINITION segments such as capacity changes and run lengths. Thus, we have chosen to place these segments adjacent. The program in the file START.DSE used the keyword StopTime to determine the length of the simulation run. Two other keywords, ResetTime and PrintTime can be used for initialization and print increments. If the statement: ResetTime = value is used, then all statistics are reset to zero at the specified time. The statement PrintTime = increment establishes a sequence of times (in simulation time units) when printouts of the system statistics will occur. For example, suppose we had
CONTROL
StopTime = 1050; ResetTime = 100; PrintTime = 300;
then the simulation would last 1050 time units, and at time 100 all statistics (like block counts, average numbers, histograms) would be reset to zero. Furthermore, at (simulated) times 300, 600, and 900 there would be a printout of the current state of the system as well as the final printout at time 1050. The final printout would include statistics collected over the last 950 time units of the simulation. It should be noted that the ResetTime and PrintTime are actually minimum times since the system controller checks the simulation clock only at event times. In other words, if ResetTime = 100, then the first event that occurs on or after time 100 will cause the simulation statistics to be restarted. The same event timing lag can occur for the statistics printout. However, the StopTime gives the exact time at which the program execution stops, even if an event is not scheduled at this time. This timing variation can be controlled by creating an artificial entity whose flow actions occur at the exact time desired.

Another method of controlling the simulation run length is based on departure counts. In other words, we might decide to run the simulation until 1000 entities have been processed through the system. For departure count control, the statement: StopCount = *value* is used. If departure counts are to be used, then the DEPART block (in the LOGIC segment) must be modified to reflect the units each departure contributes towards this total count. The syntax is

DEPART { Quantity = *xx*);

where Quantity is optional and the default is zero units. If Quantity is given, then it indicates that departures through this block are to be used with the parameter StopCount in determining when to stop the simulation. The StopCount is initially set to the specified value, then whenever an entity encounters a DEPART block, the system parameter StopCount is decremented by Quantity. When StopCount becomes zero or negative, the simulation is terminated. More than one DEPART block can contribute to the StopCount. To parallel the options for the time control of printout and statistics resets, the statements ResetCount = *value* and PrintCount = *value* are used in an analogous manner to the time controls.

The modeler can obtain a visual output of the simulation by utilizing the watch animation option. This activates a run time display of the current contents of each resource and associated queue. The purpose of the watch option is to help the modeler gain insight into how the system is functioning over time. To utilize the animation option, the user must provide a WATCHLIST statement to define the resource names and their queues to be watched. In the computer system example, the CONTROL segment can be rewritten as

CONTROL
StopTime = 60*24*5;
WATCHLIST = { computer, student0, research0 };

This control segment will cause the animation feature to be activated so that the dynamics of the simulation can be observed during execution.

A helpful feature for WATCHLIST is that the user can choose the desired symbols to be displayed on the screen during the animation process by appending, with a colon separator, a single character to each name or the ASCII number (0-255) of the character to be shown (see Table C.5). Thus, we again rewrite the CONTROL segment as

CONTROL
StopTime = 60*24*5;
WATCHLIST = { computer: 'c', student0: 's', research0: 82 };

It is also possible to give individual entities specific characters for the WATCHLIST animation feature. Each entity automatically has a system defined attribute variable named WatchChar. If WatchChar is blank (the default character), then when the entity joins a queue or resource, the character printed by the animation feature is as defined in the WATCHLIST statement. If, however, the WatchChar contains a character, then that character will be displayed during animation. The program on page 94 contains an example where the letter R is displayed for research jobs based on the assignment WatchChar = 'R'.

The debugging potential of the WATCHLIST is enhanced by the control possible over the animation. The keyword WatchStatus can be set to On, Off or SingleStep (default is On). Thus, if WatchStatus = Off is placed in the CONTROL segment, then no animation will be displayed even if the WATCHLIST is present. The keyword WatchDelay can be used to slow (or speed up) the animation. The default is WatchDelay = 300, which means that a 300 microsecond delay is added during each animation cycle. During
the execution of the program, the ESC key can be pressed to interrupt the program. When the program is interrupted, the user can change the WatchStep and/or WatchDelay.

After a program has been written and successfully verified, it may not be necessary to print out the list of block counts after each run. (The block counts are very useful during debugging and you should get into the habit of checking those numbers to determine if they are reasonable.) If it is desired not to list the block counts, then the statement

```
BlockListing = Off;
```

should be used in the CONTROL segment. Other listings that can be turned off are the listing involving the statistics (use StatListing = Off;) and the listing of the global variables (use VarListing = Off;). If you desire to suppress the listing of the program itself, simply write Listing = Off (followed by a semicolon) before any statement and the listing will be turned off at that point. If you desire to begin the listing again several statements later, then write Listing = On (followed by a semicolon) and the listing will be turned back on.

### 5.3.2 Customized Output

The LOGIC segment provides two statements for creating user-defined output. They are the entity-activated blocks PRINT and PRINTLN, which are similar to the Pascal write statements. Examples of these are

```
PRINT { 'The clock time is ', ClockTime: 5: 0 }; 
PRINTLN { 'The global variable is ', myNumber: 5: 1 }; 
```

where the operational difference between the first and second statements is that the second statement will generate a line feed and carriage return after the line is printed and the first statement will not. The number after the first colon is the number of spaces that the variable will occupy on the output line, and the second number is the number of digits to the right of the decimal that will be printed out. Note that if a nonzero decimal unit is used, the printed decimal in the number will take up one of the specified spaces. The PRINT and PRINTLN blocks are very useful in model verification (debugging) and are helpful in specialized data output. As many fields as desired can be included in a print statement, each being separated by a comma.

### 5.4 Statistics Gathering

An additional help in output control, especially during debugging, is the ability to direct printed output to a specified location. This is through a data structure called a TEXTFILE. The TEXTFILE is declared in the DEFINITION segment by

```
Id : TEXTFILE = { Name = MsDosName, Status = Write }; 
```

where Name, Status, and Write are required. MsDosName is any valid MS-DOS file specification enclosed in single quotes. In particular the printer is specified by 'PRN' and the microcomputer screen (console) is specified by 'CON'. Thus, if the user desires to have the standard output directed to the screen, but some special printouts directed to the printer the procedure is to insert the following in the DEFINITION segment

```
f : TEXTFILE = { Name = 'PRN', Status = Write }; 
```

Then, when the PRINT or PRINTLN is used, the first part of the declaration contains f, as in

```
PRINTLN { f, ClockTime:4:1 }; 
```

The program at the end of the next section has an example of this.

Three other statements have been defined for the purposes of controlling output during the execution phase of the simulation. These three statements all can appear in the LOGIC segment of the program. They are EJECTPAGE, CLEARSCREEN, and INTERRUPT. The function of the first two commands should be self-evident. The INTERRUPT is a method to stop the execution of the program temporarily. When an entity passes through the INTERRUPT command, it causes execution to pause, and the same interrupt menu is displayed that occurs when the ESC key is pressed during execution.

- Suggestion: do Exercises 5.2 - 5.4.

### 5.4 Statistics Gathering

It is often desirable to estimate flow times between two points for an entity. The purpose of the HISTOGRAM definitions, called sJobHist and resHist, in the previous program is to collect flow time statistics. To use the histograms, it was necessary to define the entity attribute startTime and to use two assignment statements. In order to free the user from the need to calculate these statistics, a STATISTICS definition can be used to
gather flow time statistics, and the necessary attributes and assignments are generated automatically. The form of the statement is

\[ \text{statName} \text{ : STATISTICS = \{ Histogram = histName \};} \]

where the keyword \text{Histogram} is optional. If \text{Histogram} is not present then only the mean flow time is reported. If a histogram of the flow times is desired, then a previously defined histogram is indicated.

The specific times desired for the statistics gathering are indicated through the use of the \text{STARTSTAT} and \text{ENDSTAT} statements. These statements have the form

\begin{verbatim}
STARTSTAT \{ Name = statName \};
some statements between
ENDSTAT \{ Name = statName \};
\end{verbatim}

where \text{statName} is the name from a previously defined \text{STATISTICS} statement. The \text{STARTSTAT} block indicates the point at which the flow time starts and the \text{ENDSTAT} block indicates the point at which the flow time stops. The following program is the same as on page 87 except that the \text{STATISTICS} statement is used and some features discussed in the previous section are included.

```
CONTROL
StopCount = 1;
BlockListing = Off;
WATCHLIST = \{ computer, studentQ, researchQ \};
"begin flow of normal student jobs"
ARRIVE \{ Time = Expd(20) \}; "3 per hour"
WatchChar = 'S';
IF Used[researchQ] > 0 THEN
begin
NumLeave = NumLeave + 1;
DEPART \{ \};
end;
```

```
LOGIC
STARTSTAT \{ Name = jobStat \};
SEIZE \{ Name = computer, Units = type, Qexcess = finished \};
WAIT \{ Time = jobTime(Random) \};
RELEASE \{ Name = computer, Units = type \};
finished:
ENDSTAT \{ Name = jobStat \};
DEPART \{ \};
"begin flow of research jobs"
ARRIVE \{ Time = Cuniform(30,90) \}; type = 2; WatchChar = 'R';
STARTSTAT \{ Name = resStat \};
SEIZE \{ Name = researchQ, Hexcess = quit2 \};
SEIZE \{ Name = computer, Units = type, Qexcess = quit1 \};
WAIT \{ Time = Expd(45) \};
RELEASE \{ Name = computer, Units = type \};
RELEASE \{ Name = researchQ \};
ENDSTAT \{ Name = resStat \};
DEPART \{ \};
```

```
DEFINITION
"student and research jobs on two terminals"
studentQ : QUEUE = \{ Capacity = 3 \};
if : TEXTFILE = \{ Name = 'PRN', Status = Write \};
researchQ : RESOURCE = \{ Capacity = 1 \};
computer : RESOURCE = \{ Capacity = 2, Queue = studentQ \};
jobTime : DISTRIBUTION(Discrete) = ( [0.25,20], (0.875,40],
 (1.6,60) );
sJobHist : HISTOGRAM = \{ Cells = 10, MinValue = 20,
 MaxValue = 120 \};
resHist : HISTOGRAM = \{ Cells = 10, MinValue = 20,
 MaxValue = 120 \};
jobStat : STATISTICS = \{ Histogram = sJobHist \};
resStat : STATISTICS = \{ Histogram = resHist \};
ATTRIBUTES = \{ type = 1 \};
NumLeave = 0;
```

END.
The last three lines of code above the final END are placed there to control the simulation. The desire is to send printed output to the printer that gives the final number of students that did not wait for a computer terminal because of the presence of research jobs. The TEXTFILE declaration in the DEFINITION segment provides access to the printer. Since the desire is to run the simulation for five days, it is necessary to activate the PRINTLN statement only after the five days have been simulated and then stop the simulation run. This can all be accomplished by using StopCount in the CONTROL segment and then using the Quantity parameter in the final DEPART block.

The output from this model will consist of a printed message on the printer, some animation during the execution of the program, and the following displayed on the screen:

```
MDR/DS 1.00
Date: 2/18/83 Time: 14:28:51
A:\EXAMPLE.5
MODEL DESCRIPTION
 listing = off

Number of students that do not wait 250

--- Simulated time 7200.00

Departures Remaining: 0

--- Flow Statistics ---

JOBSTAT
Current Number Number Average Std Dev
Number Completed Zeros Time Time
0 129 2 46.523 21.786

Histogram: SJOBHIST
Mean Standard Smallest Largest Total
Value Deviation Value Value Number
46.523 21.786 0.000 119.854 129

Number Upper Limit: Number Upper Limit: Number Upper Limit
1 30.00 54 40.00 7 50.00
18 60.00 7 70.00 6 80.00
6 90.00 4 100.00 0 110.00
2 120.00

Number too Small: 24 Number too Large: 0

RESSTAT
Current Number Number Average Std Dev
Number Completed Zeros Time Time
1 119 47 38.799 48.181

Histogram: RESHIST
Mean Standard Smallest Largest Total
Value Deviation Value Value Number
38.799 48.181 0.000 230.243 119

Number Upper Limit: Number Upper Limit: Number Upper Limit
5 30.00 8 40.00 8 50.00
5 60.00 6 70.00 7 80.00
2 90.00 3 100.00 6 110.00
1 120.00

Number too Small: 60 Number too Large: 5

--- Resources ---

RESEARCHQ
Capacity = 1 Current Units = 1
Total Number = 76 Maximum Entities = 1
Maximum Average Average Std Dev Average
Units Util. Units Units Time
1 0.641 0.64 0.48 61.56
Default Queue Statistics
No entries.
### 5.5 Internal Operating Principles

The purpose of this section is to provide a brief synopsis of the inner workings of MOR/DS. (Other block oriented languages operate in a similar fashion.) The reason for doing this is quite simple: upon first encounter, discrete simulation programs (models) are difficult to comprehend, because while they look similar to regular computer programs they do not behave, in general, like regular programs. Consequently, until the basic operation of block-oriented simulation systems is understood, there is an element of mystery that may cause the student difficulty with building models.

Let us begin with a brief and simplified discussion of the operation of a standard computer program. Internal to the underlying system is an instruction counter that indicates the next instruction to execute. After each instruction is processed, this instruction counter is incremented. The cycle of “get instruction, execute instruction, and increment counter” is repeated until some termination criterion is met. The instructions that are executed are usually calls into a library of run-time routines established by the compiler; these eventually lead to low level instructions understood by the computer (microprocessor) itself. In fact, the instruction counter is maintained by the microprocessor.

MOR/DS makes use of this same conceptual structure except for two critical differences: each entity maintains its own instruction counter, and only active entities can cause the execution of a command (that is a program statement or a simulation block). Furthermore, MOR/DS provides its own controller, and calls to its event-routine library perform the necessary commands.

MOR/DS maintains both active and inactive entities. While the external behavior suggests that several entities are active concurrently (this is relative to simulated time), the fact is that exactly one entity is actually active (this is relative to real time). It is the operation of the MOR/DS controller, through the active entity and the inactive entities, that gives a simulation program its distinctive characteristics.

The inactive entities are stored on either a special list called The Future Event Chain or lists such as queue lists. Each entity has a time at which it is scheduled to become active, and this time may be established in a number of ways. The two most obvious methods are by an ARRIVE block or by a WAIT block, each of which calculates the scheduled time by adding a user-specified (by Time = value; value to the current simulation time (i.e., ClockTime). Of course, other less straightforward mechanisms can do this as well. When an event is scheduled, the time is stored in an internal entity attribute and the entity is placed onto the Future Event Chain. The members of Future Event Chain are sorted by increasing time and then by FIFO; the entity with the earliest scheduled time is at the front of this list.

At any point during the operation of the simulation, the active entity is executing commands while the Future Event Chain keeps track of other entities and when they are scheduled to occur. At some point, the active entity encounters a delay command (which means an event time is calculated and the entity is stored in the Future Event Chain), a blocking command (as in an attempt to seize a resource and the entity is placed on a queue.

---

**COMPUTER**

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Current Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total Number</th>
<th>Maximum Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maximum Average</th>
<th>Average</th>
<th>Std Dev Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Units Util.</td>
<td>Units Util.</td>
<td>Units Time</td>
</tr>
<tr>
<td>2</td>
<td>0.725</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>52.73</td>
</tr>
</tbody>
</table>

**STUDENTQ**

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Current Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maximum Total</th>
<th>Average</th>
<th>Std Dev Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Units Number</td>
<td>Number</td>
<td>Number Time</td>
</tr>
<tr>
<td>3</td>
<td>87</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td></td>
<td>29.11</td>
</tr>
</tbody>
</table>

---

**NUM.LEAVE = 250.00**

**Compile time:** 1.82 (Secs.)

**Run time:** 10.75 (Secs.)

| Total memory available | 215542 |
| Memory used by model   | 8258   |
| Maximum dynamic memory used | 1556   |
| Total memory used      | 9854   |

---

- **Suggestion:** do Exercises 5.1 and 5.5.
5.5. INTERNAL OPERATING PRINCIPLES

5.5.1 The RELEASE Block Exception

When an entity has been selected as the active entity, it is processed through as many model statements as possible without losing active control to another entity. This concept is maintained throughout the block logic of MOR/DS except for one particular block operation. When an entity releases a resource (the RELEASE block), the controller is then charged with the responsibility of releasing entities from the resource queue so that they can obtain the needed resource units and then continue with their processing. The resource-filling rule attempts to utilize all of the available resource units; this occurs at the expense of bypassing an entity which cannot fit into the available space and selecting one that arrived at a later time, or has a lower priority depending on the queue discipline, to fill that space. To accomplish this resource-filling task, the controller selects the entities to receive the released resource capacity and places them on the front of the Future Event Chain. Thus, they will be processed immediately following the active entity’s completion. A logic dilemma within MOR/DS occurs at this point in the processing. If by chance the active entity would attempt to SEIZE the resource just released, it would precede the entities waiting in the front of the Future Event Chain and, therefore, obtain the resource again. This condition would cause the MOR/DS system to attempt to retrace the previous sequence of the entities on the resource queue and cause possible logic problems. To avoid this problem, the RELEASE block operation includes forcing the active entity to release its active status via a zero time delay, which in turn allows the waiting entities to obtain their resource units from the released resource before the causative entity re-initiates processing.

5.5.2 Language Support: EventChain

The language aspects that are helpful in learning about the internal system structure are: WATCHLIST with EventChain, INTERRUPT, and the View option of the Interrupt menu. The WATCHLIST structure has been used earlier in the chapter. One particularly useful aspect of the WATCHLIST feature is the ability to include the Future Event Chain as one of the lists to be observed. This is accomplished by including the Future Event Chain’s name, EventChain, in the WATCHLIST. Then with the single step feature in particular, you can observe the entities as they are moved about on the various system lists. Recall that a WAIT block will place the active entity on the Future Event Chain, whereas an entity at a SEIZE block that
cannot obtain the resource units needed is queued on the resource queue list. The following simple program was developed specifically to illustrate how the WATCHLIST and EventChain can be used to help understand the system mechanics. In the program we use the system entity attribute called EntityNumber, which is the system-maintained sequential number of each entity.

PROGRAM

"model for illustrating the EventChain operation"
"this is model EVENTCHN.DSE on distribution disk"

DEFINITION

res : RESOURCE = { Capacity = 1 };

CONTROL

StopTime = 100; Seed = 12345;

WATCHLIST = { res, EventChain:'A' };

WatchStatus = SingleStep;

LOGIC

ARRIVE : Time = Expd(3) ;

WatchChar = ((EntityNumber-1) Mod 26) + 97;

"ASCII 97 is the character 'a'

SEIZE { Name = res, Units = 1 };

WAIT { Time = Expd(5) };

RELEASE { Name = res, Units = 1 };

DEPART [ ];

END.

This program places a lower case letter into WatchChar so that the entities are labeled in alphabetical sequence. The arriving entity is also on the Future Event Chain but is represented by a capital A. By single stepping through a few activities using this model, you will be able to see how entities are placed on the Future Event Chain and eventually work to the front of the chain as the clock time approaches their due out time. (Recall in the SingleStep mode that you must press a key for every operation to occur.)

The following "frames" are a sequence of the watch screen displays for the above model. The first frame occurs at time 15.41 just after the arrival of entity A and its placement on the resource queue. The capital A entity is the next arrival from the ARRIVE block. The next event to occur is that entity A will release the resource and depart from the system at time 15.51.

5.5. INTERNAL OPERATING PRINCIPLES

FRAME 1: Time = 15.41

| Resource Queue | b
def
| RES | a

EventChain | a

FRAME 2: Time = 15.51

| Resource Queue | c
def
| RES | b

EventChain | b

Frame 2 illustrates the special mechanics for handling resource releases. At time 15.51, entity a has completed its service time and is in the process of releasing the resource to entity b. However, before a leaves the system, entity b is placed on the front of the Future Event Chain and a is forced, via a zero time delay, to transfer control over to entity b. This results in the current status displayed by Frame 2.

FRAME 3: Time = 15.51

| Resource Queue | c
def
| RES | b

EventChain | a

FRAME 4: Time = 15.51

| Resource Queue | c
def
| RES | b

EventChain | a

In Frame 3, entity b has obtained the resource, encountered the WAIT block, drawn its random wait time, and been replaced on the event chain. In this case entity b is due out after the next arriving entity represented by A. Still at time 15.51, entity a regains active entity status and is removed from the system producing the last frame in our example.
5.5.3 Language Support: Interrupt Menu

During the operation of a simulation, the modeller may desire to interrupt the normal flow for a variety of reasons. By pressing the ESC key, the Interrupt menu is activated. From this menu the user may select one of several actions. These include terminating the current run, printing the current statistics information, changing the WatchDelay time, activating or turning off the Watch option, and viewing the entities on the various lists denoted by WATCHLIST. The Interrupt menu can be activated under model control by using the INTERRUPT statement. When an entity executes this statement, the Interrupt menu is immediately activated for user action. There are many situations, where being able to observe the system status, statistics, chains, and entities can help in isolating a modeling or analysis problem. The INTERRUPT statement was incorporated for exactly these purposes. One of the more useful aspects of the interrupt is the View option.

5.5.4 Language Support: The View Option of Interrupt

The final language support construct which is useful in studying the operation of MOR/DS is the entity view option of the Interrupt menu. The view option allows the modeller to observe the entities that are on the various lists denoted by WATCHLIST. The view mechanism displays only the constructs (queues, resources, chains, and gates) that have been designated in the WATCHLIST set. The entity characteristics that are displayed are four internal attributes (entity number, WatchChar symbol, the time of the last or next scheduled event for the entity, and the key value for sorting the entities on priority-based queues), and up to six user-defined entity attribute values. For the example model of Section 5.5.2, the following would be displayed at the time that the system has the status displayed in Frame 1.

Event Chain
1 a 15.506 0.00 |
----
7 23.012 0.00 |

There are no user-defined entity attributes used in this example; therefore, no entries are listed for the attribute values. Note that entity 7 appears on both the res and the EventChain lists. Entity 7 has control of the single resource unit of res and is currently waiting out its service time (on the Future Event Chain) before releasing the resource unit and leaving the system. Entity 7 on the event chain is the next scheduled arrival to the system. It is scheduled to arrive at time 23.012 and at that time will be assigned the WatchChar value of g. Note that while observing the animation feature, this future arrival is represented by an a. This character comes from the default character assignment of the WATCHLIST statement. If entities have been assigned a WatchChar value, then this symbol is displayed to represent the entity; otherwise, the character chosen in WATCHLIST is used. If neither of these situations dictates the character representation, then the system defaults are displayed.

5.6 Modeling Ideas

In this section, a few common situations are described, and the suggested ideas for modeling them are given.

Variable arrival rates. Assume that we have a Poisson arrival process in which arrivals occur at a rate of 5 per hour between midnight and 8:00 a.m., 10 per hour between 8:00 a.m. and 8:00 p.m., and 8 per hour between 8:00 p.m. and midnight. One procedure for modeling these different arrival rates is to have a different ARRIVE block for each rate and then discard arrivals occurring at the wrong time. Thus, we have

"units of time are hours"

```
ARRIVE { Time = Expd(1/5) };
IF (ClockTime MOD 24) < 8 THEN GOTO label 1
ELSE DEPART { };
ARRIVE { Time = Expd(1/10) };
IF ((ClockTime MOD 24) >= 8) AND ((ClockTime MOD 24) < 24)
THEN GOTO label 1 ELSE DEPART { };
```
ARRIVE { Time = Expd(1/8) ;
IF (ClockTime MOD 24) >= 20 THEN GOTO label1.
ELSE DEPART { 

(The MOD operator is a binary operator where a MOD b is the remainder of a/b; thus, 5.2 MOD 24 is 5.2 and 25.3 MOD 24 is 1.3.)

The use of multiple ARRIVE blocks is only proper theoretically when the interarrival times are exponentially distributed because of the exponential distribution’s memoryless property. If a distribution other than the exponential is used, then more information regarding the probability distribution of the first arrival immediately after the rate change is needed. However, care should also be taken to not build into the simulation more detail than the data warrants. That is, if data are not collected to investigate specifically the first interarrival time after a rate change, then it is wasted effort to model it.

An alternative approach for approximating this process is to define a function representing the arrival rate. Thus, in the DEFINITION segment of the program we have

arrRate : DISTRIBUTION(Discrete) = ( (8.0, 2), (20,0.1), (24,0.125) );

In the LOGIC segment we have

ARRIVE { Time = Expd(arrRate(ClockTime MOD 24)) ;

The second approach is actually just an approximation because the parameter for the next arrival depends on the time of the current arrival; thus, the arrival rate would change at the first arrival after 8:00 A.M., rather than precisely at 8:00 A.M. Therefore, this second method should not be used when the number of arrivals is small relative to the length of the time interval.

FIFO Ordering at a Resource. A small parking area has room for two cars. However, if a truck comes, it takes up both parking spaces. Vehicles arrive according to some random process (20% of the vehicles are trucks) and stay for an exponential length of time, with a mean stay of 3 minutes. Assume in the following code segment that the user-defined arrival distribution myDist was defined in the DEFINITION segment.

5.5. MODELING IDEAS

DEFINITION
parking : RESOURCE = { Capacity = 2 };
...
ATTRIBUTES = { spaces=1 };

CONTROL
StopTime = 100;

LOGIC
ARRIVE { Time = myDist(Random) ;
IF Random < 0.20 THEN spaces = 2;
SEIZE { Name = parking, Units = spaces };
...

As you consider the above fragment of logic, assume that a car (spaces = 1) first arrived, then a truck (spaces = 2) next arrived, and thirdly a car arrived before any of the vehicles left. In the normal resource filling procedure of most block-oriented languages, including MOR/DS, the third vehicle to arrive will seize the resource ahead of the second vehicle because there is room in the resource for it. One of the easiest mechanisms to use when a strict FIFO ordering discipline is desired is to create a fictitious “door” through which all entities must pass. The following code segment illustrates this concept.

DEFINITION
door : RESOURCE = { Capacity = 1 };
...
ATTRIBUTES = { spaces=1 };

CONTROL
StopTime = 100;

LOGIC
ARRIVE { Time = myDist(Random) ;
IF Random < 0.20 THEN spaces = 2;
SEIZE { Name = door };
SEIZE { Name = parking, Units = spaces };
RELEASE { Name = door };
WAIT { Time = Expd(3) };
RELEASE { Name = parking, Units = spaces };
...

(It is sometimes tempting not to define the attribute variable spaces and instead use a random variable in the SEIZE statement. However, this will
chapter 5. fundamentals of simulation modeling

5.7. exercises

job is started it will run to completion, that is, the higher priority jobs do not have preemptive capabilities.

DEFINITION

"time units are minutes"

ATTRIBUTES = { pri: 1 };

queue : QUEUE = { Discipline = CHOICE(pri) };

server : RESOURCE = { Capacity = 1, Queue = theQ };

CONTROL

StopTime = 100;

LOGIC

ARRIVE { Time = Expd(12) };

IF Random <= 0.1 THEN pri = -1;

SEIZE { Name = server };

WAIT { Time = 16 };

RELEASE { Name = server };

DEPART { };

END.

• Suggestion: do Exercises 5.6 - 5.10.

5.7. Exercises

5.1. Add or complete the needed statements to the following model to accomplish the five requests (the basic time unit is minutes).

(a) Waiting time at block label \( \text{WT1} \) is to be a continuous uniform random variable varying from 10 to 20 minutes.

(b) The waiting time distribution at block label \( \text{WT2} \) is to be developed from the following ten data points obtained by observing the actual waiting times: 8, 18, 6, 10, 8, 18, 8, 10, 8, 18.

(c) Collect a histogram of the time in the system for all customers. Use 10 minute categories ranging from 20 to 100 minutes.

(d) The simulation run length is to be the minimum of 500 minutes or 10 completed customers, whichever occurs first.

(e) Order the customers in the queue waiting for resource \( \text{ResourceA} \) according to a priority scheme based on a random assignment from an exponential distribution with a mean number of 100.

Operator as a Resource. A gas station attendant is responsible for two pumps. A car arrives at the gas station every eight minutes. It takes the attendant about 5 minutes (uniformly distributed between 3 and 7 minutes) to service the car at the pump. When the attendant is finished, the car owner goes inside (during which time the car stays in front of the pump) and pays for the gas, which takes an additional 2 minutes. As long as the attendant is with one car, he cannot be with the other car.

Ordinarily, the gas station would be thought of as a resource with a capacity of two; however, the station must be modeled as two resources because the attendant is also a resource. The following partial model illustrates these concepts.

DEFINITION

attendant : RESOURCE = { Capacity = 1 };
pumps : RESOURCE = { Capacity = 2 };

CONTROL

StopTime = 100;

LOGIC

ARRIVE { Time = 8 };

SEIZE { Name = pumps };

SEIZE { Name = attendant };

WAIT { Time = Cuniform(3, 7) };

RELEASE { Name = attendant };

WAIT { Time = 2 };

RELEASE { Name = pumps };

...
PROGRAM "for problem number 1"

DEFINITION
   aTable : RESOURCE = { Capacity = 2 };
   Waiter : RESOURCE = { Capacity = 1 };
   LABELS = { WT1, WT2 };

CONTROL

LOGIC
   ARRIVE { Time = Expd(10) };
   SEIZE { Name = Waiter, Units = : };
   WT1: WAIT { Time = : };
   RELEASE { Name = Waiter, Units = 1 };
   SEIZE { Name = aTable, Units = 1 };
   WT2: WAIT { Time = : };
   RELEASE { Name = aTable, Units = 1 };
   DEPART { : };

END.

5.4. A barbe shop has two barber chairs and an additional four chairs for waiting customers. Service is on a first-come first-serve basis, no reservations. Customers will always wait if they can get a chair, but if no chairs for waiting customers are available (i.e., when six customers are in the shop), then an arriving customer will not wait. The two barbers cut hair at a fairly constant rate. It takes them from 30 to 45 minutes, uniformly distributed, for each customer. If only one customer is in the shop, then only one barber is busy; the barbers do not help each other while cutting hair. The shop is open 10 hours each day. The times between arrivals are exponentially distributed, with a mean depending on the time of day. During the first two hours of the business day, the mean interarrival time is 30 minutes; during the next three hours, the mean is 15 minutes; during the next three hours, the mean is 20 minutes; and during the last two hours, the mean is 12 minutes. Estimate the number of customers lost each day. (Hint: See the "Variable Arrival Rate" discussion in Section 5.6.)

5.5. The computer system at a certain manufacturing company is composed of five units: the CPU (central processing unit), hard disk drive, tape drive, high-speed line printer, and plotter. A job enters the computer system by "arriving" at the CPU. The job then spends an average of 3.2 seconds in the CPU. When it is finished with the CPU, the job leaves the system with probability 0.1, goes to the disk drive with probability 0.8, or goes to the tape drive with probability 0.1. A job that needs disk drive operations spends an average of 0.8 seconds operating the disk drive. When it is finished with the disk, it then uses the printer with probability 0.5, uses the plotter with probability 0.1, returns for more CPU time with probability 0.3, or leaves the system with probability 0.1. A job that returns to the CPU acts like a new job: that is, it spends an average of 3.2 seconds in the CPU again and then proceeds as before. Jobs that use the tape drive, the printer, or the plotter leave the system after they are finished with those units. The average length of time spent for the tape drive, printing, and plotting operations is 1.8 seconds, 7.3 seconds, or 32.2 seconds respectively. The arrival rate of jobs at the CPU unit is 10 per minute, and the times between jobs are exponentially distributed random variables. All job times or the individual units are also exponentially distributed random variables, and each unit operates as a single server (one-at-a-time) with a FIFO ordering policy and a buffer that can be considered infinite. Let the
run length be long enough for 100 jobs to be completed and estimate the average length of time a job spends in the system. Also estimate the fraction of jobs that use more than 5 seconds cumulative CPU time. To obtain an understanding of the dynamics of this system, include all the resources in a WatchList. Use the statement

\[
\text{WatchChar} = \left(\frac{\text{EntityNumber} - 1}{26}\right) \, +97;
\]

as soon as an arrival occurs (see the program in Section 5.5.2).

5.6. A paint shop has two painting units. The painting units can either work on independent jobs or on the same job (depending on the type of job). Jobs arrive at the shop in a Poisson arrival process, with a mean rate of 10 jobs per hour. One in three jobs requires that both painting units be used simultaneously, and two out of three jobs use only one of the units. The service time per job is exponentially distributed, independent of the number of painting units demanded, with a mean service time of 7.5 minutes. A strict FIFO queueing discipline is used; that is, if two one-unit jobs arrive sequentially, they may be served simultaneously; however, if a two-unit job arrives between two one-unit jobs, then the second one-unit job must wait until the two-unit job is finished. (Hint: in order to implement the strict FIFO discipline, a second facility of one-unit capacity must be defined that can act as a "doorway" to the paint shop. All jobs should demand one unit of service from the "doorway" facility.) Use a 5-hour initialization period and then simulate the process for 25 hours. Estimate the average number of jobs in the system. What is the average number of two-unit jobs in the system? How do these averages change if the standard resource-filling procedure is used?

5.7. A machine shop has two machines, "a" and "b," serviced by a single operator. The operator must load jobs on each machine when they are to be processed. Each machine can only work on one part at a time. It takes 4 to 5 minutes (continuous uniform) to set up a job on machine "a," and then the processing time is exponentially distributed, with a mean of 10 minutes. It takes a fixed time of 6 minutes to set up a job on machine "b," and then the processing time is a discrete random variable taking between 4 and 8 minutes. The operator must also unload each job; this takes exactly 2 minutes for jobs on machine "a" and 3 minutes for jobs on machine "b." Jobs arrive at the shop with a mean interarrival time of 10 minutes, exponentially distributed. One-third of the jobs use machine "a," and the remainder use machine "b." The operator, obviously, can work on setting up or unloading only one job at a time. The machines run independently of the operator once they have been loaded with a job, and they stop and wait for the job to be unloaded when it is complete. Estimate the average length of time a job spends in the system. What is the difference in the mean flow times for jobs that use machine "a" and for jobs that use machine "b"? The shop is open to receive jobs for eight hours, and all jobs remaining in the shop at the end of the day are processed immediately so that the shop is always empty at the start of each day. (Hint: You will most likely define three resources: one for the operator and one for each machine. You must be careful as to the order in which the resources are seized. If the order is wrong, the system will have unbounded queues.)

5.8. Parts arrive at a work station that is staffed by two workers. The arrivals are Poisson with a mean rate of fifteen per hour. The processing of each part consists of two steps. A special tool and a worker is required for the first step. There is only one tool, and it must be shared by both workers. The second step only requires a worker (no tool), and is performed as soon as the first step is complete. The length of time required by the first step is a continuous uniform random variable ranging from 3 to 5 minutes. The length of time needed for the second step is an exponentially distributed random variable with a mean of 5 minutes.

As parts come into the system, they are given a priority value that is equally likely between 1 and 6. The parts are worked on according to the priority value with, for example, a part of priority 1 being worked on before a part of priority 2. There is only room in the work station for 15 parts to wait, and any parts that arrive when all 15 spots are taken (no preemption) are sent to an alternate work station (the alternate work station is no part of this model).

Obtain the average time in the queue and the average time in the system for parts. Determine the difference in the system flow times for parts of priority 1 and parts of priority 6. Also, obtain the number of parts that were sent to the alternate work station. What fraction of those parts lost were of priority 1 and what fraction were of priority 6?
The simulation should be terminated when either 8 simulated hours have passed or when 50 parts have been completed, whichever occurs first. (Hint: See the “Priorities” discussion in Section 5.6.)

5.9. Consider an M/M/1 queueing system. (That is, a single-server system with no limit on the number of customers who wait, and the interarrival and service times are exponentially distributed.) Let the mean interarrival time be 1.2 minutes and the mean service time be 1.5 minutes. Model this system for 100 completed customers. What is the mean and standard deviation for the time spent in the system (time in queue plus time in service) for these 100 customers? What is the effect on the time in the system if two identical servers are in parallel?

5.10. Consider an M/M/2 queue (see previous problem for terminology) where the servers are in parallel working on a single queue but the service times have different means. One of the servers uses an old machine with a mean time of 1.3 minutes, whereas the other server uses a new machine which has a mean service time of 1 minute. The interarrival times are exponential with a mean time of 0.5 minutes. Model this system simulating 100 completed customers and determine the mean and standard deviation for the number of customers in the system. Customers will select the faster server if they have a choice. (Hint: use two resources, one of capacity 2 for the servers and another of capacity 1 for the fast machine. Then with the R/Excess option for the fast machine, the excess can be routed to the slower machine.)

Chapter 6

Statistical Analysis of Output

Simulation modeling is a very powerful tool for analysis that can be learned by most technically inclined individuals and, thus, its use is widespread. The typical approach is to learn the basic simulation concepts and a simulation language, but pay very little attention to the proper analysis of the output. The result is that many simulation studies are performed improperly. Simulation programs do not serve much purpose without a proper interpretation of the output. In fact, an improper understanding of the results of a simulation run can be detrimental in that the analysis could lead to an incorrect decision. Therefore, now that we can build simple models, it is appropriate that the analysis of the output be discussed before advanced modeling techniques become so complex that programming mechanics overwhelm the output analysis concepts.

The reason that the output analysis of simulation is often given only a cursory review is that output analysis is more difficult than simple programming. Furthermore, there remain unanswered questions in the area of output analysis. In this chapter, we discuss enough of the concepts to equip the analyst with the minimum set of tools. We caution that there is much more to this area than we are able to cover at the level of this introductory text. Many of the concepts presented here are based on the text by Averill Law and W. David Kelton, which we recommend for further reading.

Simulation output can be divided into two categories: terminating simulations and steady-state simulations. A terminating simulation is one in which there is a fixed time reference of interest. An example of a terminating simulation is Exercise 5.2, which deals with a newspaper stand that only operates for four hours every day; thus the simulation terminates after four hours of simulated clock time. Steady-state simulations do not have a fixed length of time; they deal with questions of long-run behavior. If we assume that the widgets from the example of Section 2.3 are produced 24 hours a day seven days a week, then questions dealing with the long-run behavior of the process would be of interest. Analyses of terminating and steady-state simulations are considered separately.

5.1 Terminating Simulations

Consider a small service station with only one gasoline pump located long a major road leading into a large metropolitan area. It is open from 6:00 A.M. to 10:00 P.M. every day, with a rush hour occurring from 4:00 P.M. to 6:00 P.M. (Access to the station is difficult for people going into the city; therefore, a morning rush hour does not cause congestion at the gasoline pump.) Because of space limitations, there is room for only four customers in the station; in other words, if one customer is at the pump, a maximum of three vehicles can be waiting. The owner of the station is concerned that so many customers are lost because people find the station full; therefore, simulation study is desired to determine how many customers are lost due to the limited capacity. If the number lost is significant, the owner will consider purchasing a second pump. All interarrival times are observed to be exponentially distributed random variables, with an average of 6 arrivals per hour from 6:00 A.M. to 4:00 P.M., 20 per hour from 4:00 P.M. to 6:00 P.M., and 6 per hour from 6:00 P.M. to 10:00 P.M. The time each customer spends in front of the pump is also exponentially distributed, with an average time of 4 minutes.

This service station example illustrates a situation suitable for a terminating simulation study. (A steady-state analysis does not make sense since the process is constantly being "restarted" every morning.) In our analysis, we will consider the fact that information from this example, as well as the more dependent simulated output. Observe that the command Randomize On is used in the CONTROL segment of the program so that each execution of the program is equivalent to a different statistical experiment. The simulation model of the situation is as follows:

PROGRAM
"service station example with one pump"
"system time unit is minutes"

DEFINITION
theQueue : QUEUE = { Capacity = 3 } ;
thePump : RESOURCE = { Capacity = 1, Queue = theQueue } ;
LABELS = { ok, excess } ;
numDone = 0 ;

CONTROL
StopTime = 60*6 ; "simulates 16 hours"
Randomize = On ; "set seed randomly"

LOGIC
"non-rush hour arrivals, 6 per hr."
ARRIVE ; Time = Expd(10) ;
GOTO ok ;
"rush hour additional arrivals, 14 per hr."
ARRIVE ; Time = Expd(60/14), OffSet = 10*60 ;
IF ClockTime > (12*60) THEN
DEPART { } ;
ok : SEIZE { Name = thePump, Qexcess = excess } ;
WAIT { Time = Expd(4) } ;
RELEASE { Name = thePump } ;
DEPART { } ;
excess : numDone = numDone + 1 ;
DEPART { } ;

END.

The first step in determining the feasibility of obtaining a second pump is to use the above program to estimate the expected number of daily customers lost under the current configuration of one pump. The number of lost customers is contained in the variable numDone, computed at the block labeled excess. The program was executed three times, yielding the number of lost customers in a day as 25, 10, and 12, respectively. With variations of this magnitude, it is clear that care must be taken in estimating the mean number of customers lost. Furthermore, after an estimate is made, it is important to know the degree of confidence that we can place in the estimate.
Therefore, instead of using a single value for an estimate, it is more appropriate to give a confidence interval. For example, a 95 percent confidence interval for the expected number of customers lost each day is an interval which we can say with 95 percent confidence contains the true mean. That is, we expect to be wrong only 5 percent of the time. (The choice of using 95 percent confidence instead of 97.2 percent or 88.3 percent confidence is, for the most part, one of historical precedence.)

To illustrate the building of confidence intervals, the simulation of the one pump service station was executed 30 times and the daily total number of lost customers was recorded in Table 6.1. The histogram of the data is in Figure 6.1. (It might appear that it is redundant to plot the histogram since the table gives all the information; however, we recommend that you make a practice of plotting your data in some fashion because quantitative information cannot replace a visual impression for producing an intuitive feel for the simulated process.)

<table>
<thead>
<tr>
<th>#</th>
<th>25</th>
<th>10</th>
<th>12</th>
<th>16</th>
<th>31</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>10</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>13</td>
<td>3</td>
<td>16</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>23</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Data from 30 simulations of the one pump service station

The usual techniques for constructing confidence intervals assume that the data are observations of independent, normally distributed random variables. Under such an assumption, the 1 - α confidence interval for a mean value is given by

\[
(\overline{X} - t_{(n-1,\frac{\alpha}{2})} \frac{S}{\sqrt{n}}, \overline{X} + t_{(n-1,\frac{\alpha}{2})} \frac{S}{\sqrt{n}})
\]

(6.1)

where \(n\) is the number of data points, \(\overline{X}\) and \(S\) are the sample mean and standard deviation using the \(n\) data points (see Equation 4.2), and \(t_{(n-1,\frac{\alpha}{2})}\) is a critical value based on the Student-t probability distribution (see Table 3.2 in the Appendix). The \(n-1\) subscript in the \(t\)-value refers to the degrees of freedom, and \(\frac{\alpha}{2}\) refers to the amount of "right-hand" error we are willing to risk. To understand why the \(1 - \alpha\) confidence interval uses a critical value associated with \(\frac{\alpha}{2}\), consider a 95 percent confidence interval; such a confidence interval has an associated risk of 5 percent error. Since the error is divided evenly on both sides of the interval, there is a 2.5 percent chance of error to the left of the interval and a 2.5 percent chance of error to the right.

![Histogram from 30 simulations of the one pump service station using intervals of five units](image)

The sample mean and standard deviation for the data obtained from the 30 simulations are \(\overline{X}_{30} = 12.9\) and \(S_{30} = 6.7\), respectively. The critical \(t\)-value for a 95 percent confidence interval with 29 degrees of freedom is 2.045. (It should be observed that the Student-\(t\) distribution limits to the normal distribution as the degrees of freedom increase.) Equation 6.1 thus
yields
\[ 12.9 \pm 2.045 \times \frac{6.7}{\sqrt{30}} = (10.4, 15.4). \]
That is, we are 95 percent confident that the expected number of customers lost each day is between 10.4 and 15.4.

We need to consider the two assumptions under which Equation 6.1 theoretically holds. The assumption of independent observations is adequately satisfied as long as each replicate execution of the simulation program begins with a different initial random number seed. This is accomplished by including the `Randomize` = `On` statement in the `CONTROL` segment to obtain independent observations. The normality assumption is troublesome and can cause errors in the confidence interval. This is especially true for the service station example since the histogram of Figure 6.1 does not give the appearance of a symmetric bell shaped curve.

Regardless of the actual underlying distribution (assuming a finite mean and variance), we can always take advantage of the central limit theorem and transform the output into normally distributed random variables. The central limit theorem states that sample means, when properly normed, become normally distributed as the sample size increases. To take advantage of this, we group the 30 data points from Table 6.1 into five groups of six points each. For example, let each row of Table 6.1 represent a different group. For each group, we calculate the mean of that group and treat the resulting five values as our data. Thus, for the one pump service station example, the five data points (average value for each row of Table 6.1) are 17.1, 21.3, 10.2, 12, and 12.1. The sample mean and standard deviation for these data are \( \bar{x}_5 = 12.9 \) and \( s_5 = 2.4 \). Applying Equation 6.1 on this data set yields
\[ 12.9 \pm 2.776 \times \frac{2.4}{\sqrt{5}} = (9.9, 15.9), \]
for the 95 percent confidence interval. Note that the critical \( t \)-value is based on 4 degrees of freedom since our sample size is 5. Obvious questions now arise as to how big the grouping and the optimum overall sample size should be. Unfortunately, there are no good quantitative answers to such questions. Qualitatively, evidence suggests that as a distribution becomes more skewed (deviates from symmetry), the confidence interval based on the normal assumption becomes less accurate. However, specific quantitative guidelines that are suitable for all situations are not available.

Three common measures estimated with a simulation are mean values, variances, and probabilities. Equation 6.1 gives the confidence interval used for means. For completeness, we now present confidence intervals for the other quantities.

The 1 - \( \alpha \) confidence interval for the variance is
\[ \left( \frac{(n-1)S^2_n}{\chi^2_{(n-1), 1-\frac{\alpha}{2}}} \right)^{1/2}, \left( \frac{(n-1)S^2_n}{\chi^2_{(n-1), 1-\frac{1-\alpha}{2}}} \right)^{1/2}, \] (6.2)
where \( \chi^2_{n, \alpha} \) is the chi-square statistic (see Table C.1) using \( n \) degrees of freedom with a "right-hand" error of \( \alpha \). Notice that the larger \( \chi^2 \) statistic is used in the left-hand limit of the interval, whereas the smaller statistic is used in the right-hand limit.

A confidence interval for proportions is used when tail probabilities are desired. For example, a manager may be more concerned about costs becoming too high than about mean values. For the purpose of constructing a confidence interval for proportions, let \( \hat{p} \) be the estimator for a proportion based on a sample size of \( n \). (For example, suppose we desire to estimate the probability that the daily cost will be greater than $5,000. Let \( p \) denote the value of the probability that the daily cost is in excess of $5,000, and let \( \hat{p}_{50} \) be the estimator for \( p \) based on 50 runs. Then \( \hat{p}_{50} \) is computed by dividing the number of times that the cost was actually above $5000 by the total number of runs, i.e., 30.) The 1 - \( \alpha \) confidence interval for proportions should only be used when \( n \) is large and is given by
\[ \left( \hat{p}_n - z_{\alpha} \sqrt{\frac{\hat{p}_n(1 - \hat{p}_n)}{n}}, \hat{p}_n + z_{\alpha} \sqrt{\frac{\hat{p}_n(1 - \hat{p}_n)}{n}} \right), \] (6.3)
where \( z_{\alpha} \) represents the statistic from the standardized normal table (see Table C.1) with a right-hand error of \( \alpha \).

### 6.1.1 Supporting Language Structures

It is possible to simply rerun a program several times to obtain the necessary replicates for a simulation. However, manually rerunning a program and recording the relevant values by hand becomes tedious. Fortunately, a feature is available in the `CONTROL` segment that allows program control of replicate runs. Provided within the scope of the `CONTROL` segment is an optional logic segment, called `EXITLOGIC`, which allows the full complement of `MOR/DS` programming statements. `EXITLOGIC`, if present, is executed at the `termination` of a simulation run. It is actually a subsegment...
in the CONTROL segment that provides a mechanism to perform special tasks, to close down a run, and to structure and control multiple runs of the same model. The beginning of the EXITLOGIC section is identified by the keyword EXITLOGIC and terminated by END.

With the exception of SETGATE (discussed in the next chapter), no simulation commands are permitted within EXITLOGIC. However, all other programming statements are permitted. There are three special statements especially designed for control purposes. These are:

1. CONTINUE, which resumes the current run as though the ESC key had been pressed or the INTERRUPT statement executed,

2. RESET, which continues the same simulation run but clears all non-protected statistics variables and leaves ClockTime as is,

3. CLEAR, which deletes all entities, clears all non-protected statistics variables, and resets ClockTime to zero.

In all three cases, user variables and the system variables StopTime, StopCount, ResetTime, and ResetCount remain at their current values.

When a program is first executed, global variables are initialized in the DEFINITION segment; however, when the EXITLOGIC is executed, the initialization in the DEFINITION segment is not repeated. It is the user's responsibility to ensure that all user and system variables are properly set for subsequent runs of the same model. A common error is to forget to reset the status for gates.

Two statements are available in both the EXITLOGIC and the LOGIC segments to cause the execution to stop: HALT and ABORT. A HALT in the LOGIC segment causes the current run to terminate with control being passed to EXITLOGIC. A HALT in EXITLOGIC suspends its operations, and control is passed to the LOGIC segment provided that either CLEAR, CONTINUE, or RESET was encountered prior to the HALT. The ABORT command causes immediate termination of the entire simulation program. If ABORT is executed in the LOGIC segment, control does not pass back to EXITLOGIC; instead, the program is immediately terminated, and normal statistical output is displayed before the run is closed down.

When activated, EXITLOGIC begins execution at the first statement and operates in the same fashion as an ordinary program; there are no active or inactive entities.

In most multi-run simulations, EXITLOGIC must perform three distinct functions. First, the current run must be closed out by tabulating global statistics and printing out summary results. Second, all relevant initializations must be performed for the next run to occur. Third, a final close out section is needed to perform any necessary final tabulations or printouts.

The following program illustrates the typical EXITLOGIC structure. Notice the user variable runCount is used to control the overall system. Also observe that print statements are included in the EXITLOGIC. There is no default output after the EXITLOGIC, so any new information obtained must be accessed by print statements. The final values of global and attribute variables that are automatically printed at the end of a run are the values at the end of the LOGIC segment, not those at the end of the EXITLOGIC.

**PROGRAM**

```
"service station example with one pump"
"max of 4 in system: system time unit is minutes"

```

**DEFINITION**

```
theQueue : QUEUE = { Capacity = 3 } ;
thePump : RESOURCE = { Capacity = 1, Queue = theQueue } ;
f
: TEXTFILE = { Name = 'PRN', Status = Write } ;
"make sure printer is on!"
LABELS = { ok, excess } ;
numGone = 0 ; meanGone = 0 ;
numSq = 0 ; runCount = 0 ;
stdGone = 0 ; numRuns = 5 ;
```

**CONTROL**

```
StopTime = 60*16 ; "simulates 16 hours"
EXITLOGIC
PRNTLN { f, 'number lost: ', numGone } ;
runCount = runCount + 1 ;
meanGone = meanGone + numGone ;
numSq = numSq + numGone^2 ;
numGone = 0 ;
IF runCount < numRuns THEN CLEAR
  "thru where numCount = numRuns"
ELSE begin
  meanGone = meanGone/numRuns ;
  stdGone =
  Sqrt((numSq - numRuns*Sqr(meanGone))/ (numRuns-1)) ;
  PRNTLN { f, 'mean is ', meanGone, 5:2, ;
```

```
6.2 Steady-State Simulations

A large manufacturing company has hundreds of a certain type of machine that it uses in its manufacturing process. These machines are critical to the manufacturing process, so the company maintains a repair center that is open 24 hours a day, seven days a week. At any time, day or night, there is one repairman who takes care of fixing machines. Each machine that is in the repair center costs the company $75 per hour in lost production due to its unavailability. It has been determined that the arrival of failed machines to the repair center can be approximated by a Poisson arrival process with a mean rate of 4 per day, i.e., independent and exponentially distributed interarrival times with a mean of 1/4 day between arrivals. (We have assumed that the company has such a large number of these machines in use that the failure of a few of the machines does not significantly affect the arrival rate resulting from the failure of any of the remaining machines.) The time it takes to repair a machine is highly variable due to the many different things that can go wrong, and, in fact, the repair times are also described by an exponentially distributed random variable with a mean time of 4 hours.

The repair center thus operates as a single server, infinite capacity, FIFO queueing system. The company would like to determine if it would be economically advantageous, in the long run, to add another repairman per shift so that two machines could be under repair at the same time when the need arises. Each repairman costs the company $30 per hour, including salary and overhead.

The following program simulates the repair shop:

```
PROGRAM  "repair center with one repairman
          time units are days"

DEFINITION
numberDays = 52;
numberRepairmen = 1;
dailyCost = 24*30*numberRepairmen;
totalCost = 0;  "used to accumulate daily costs"
avgCost = 0;
ATTRIBUTES = { timeDown = 0 };
repairman : RESOURCE = { Capacity = numberRepairmen };

CONTROL
StopTime = numberDays;

LOGIC  "first arrive block simulates arrival of failed machines"
ARRIVE { Time = Expd(1/4) };
timeDown = ClockTime;
SEIZE { Name = repairman, Units = 1 };
WAIT { Time = Expd(4/24) };
RELEASE { Name = repairman, Units = 1 };
dailyCost = dailyCost + 75*24 * (ClockTime - timeDown);
DEPART { };

ARRIVE { Time = 1 } ;  "keeps track of days"
totalCost = totalCost + dailyCost;
addrCost = totalCost / ClockTime;
PRINT { 'Day ', ClockTime :4 } ;
PRINTLN { ' daily cost is ', dailyCost :7.2, 
          ' average cost is ', avgCost :3.2 } ;
dailyCost = 24*30*numberRepairmen;  "reset this value"
DEPART { } ;
```

END.
The program contains a global variable labeled `avgCost` that maintains a running average of the cost to the company due to the machines being in the repair center. The last 7 statements of the above program were added to allow for a daily printout of each day's cost (the global variable `dailyCost`) due to machines being in the repair center, and the running average of those costs. The high degree of variability in daily costs is illustrated in Figure 6.2, in which daily costs for three simulations are shown. Figure 6.3 shows the plot of the running averages of those costs. The variability between simulations for this repair facility is illustrated by comparing the three curves in Figure 6.3.

In our example, we are interested in a long-range policy, and it is the long-run average cost per day on which the decision will be based. Consequently, we need to look at the behavior of this quantity over time. (The first question that actually needs to be asked when a steady-state analysis is contemplated is “Do steady-state values exist?” It is easy to construct situations in which no steady-state exists. For example, if a slow server is trying to handle fast arrivals, the backlog might continually increase and never reach steady-state. A rigorous answer to the “existence” question is beyond the scope of this textbook. We shall simply assume throughout this section that the processes under consideration do reach steady-state behavior.) Notice that the graphs of `avgCost` in Figure 6.3 initially increase rapidly, and then they approach the straight line drawn on the graph. Extrapolating beyond the shown graph, the plot of `avgCost` would eventually cease to change. Its limiting value with respect to time is called its steady-state value. The straight line in the figure...
is the theoretical steady-state value (which in this case is the cost per day, incurred by the machines being in the repair room).

One of the primary purposes of a simulation is to estimate steady-state values of performance measures, including means, variances, and in some cases, probabilities. But, because values from simulations are statistical estimates, we also would like to determine the accuracy (i.e., confidence limits) of the estimates. While both obtaining an estimate and determining its accuracy present challenges, it is usually easier to derive the estimates than it is to determine their confidence limits. If a steady-state value exists, the time-averaged value from any single simulation will limit to the steady-state value as the run length increases to infinity. The obvious practical difficulty is to know when to stop the simulation (since few of us have time to wait for infinity). Since the above simulation was stopped short of infinity, we do not expect the simulation’s final average cost to be exactly equal to the true long-run value; therefore, it is necessary to determine how much confidence we have in the estimate. To illustrate the problem, assume that the simulations illustrated in Figure 6.3 were stopped after 52 days. Taking any one of the simulations in the figure, the analyst would possibly conclude that the simulation is reasonably close to steady-state because the graphs indicate that the major fluctuations have ceased; however, none of the three examples have yet reached the asymptotic value. Of course, this particular example was chosen because we could easily determine the theoretically correct value, and with this knowledge we have the ability to measure the accuracy of our estimators. Without the theoretical values (we would not normally simulate if the theoretical values were known), we must turn to statistical methods, which usually involve confidence limits.

Before discussing specific procedures for obtaining confidence intervals, let us consider some of the underlying concepts. For illustrative purposes, our context will be the determination of an estimate for mean cost. It should be remembered that constructing a confidence interval for a parameter requires an estimator and the variance of the estimator.

Let \( \mu \) denote the long-run mean, and let \( \sigma^2 \) be the long-run variance of the daily cost. Further, let \( C_1, C_2, \ldots \) be the sequence of daily costs obtained by the simulation on days 1, 2, \ldots, respectively. Suppose \( n \) such observations have been obtained by a run of the simulation and let

\[
\overline{C}(n) = \frac{1}{n} \sum_{i=1}^{n} C_i.
\]

6.2. STEADY-STATE SIMULATIONS

Figure 6.3 plots \( \overline{C}(n) \) versus \( n \) for three different realizations, where \( n \) is the number of days. In theory,

\[
\lim_{n \to \infty} \overline{C}(n) = \mu \text{cost}
\]

which means that if we let \( n \) grow indefinitely, the estimate (6.4) converges to the true value of \( \mu \text{cost} \). The intuitive meaning of Equation 6.5 is that we can take a single simulation, run it a long time, and then use the time-averaged value for an estimate of the long-run expected value\(^2\) (see Section 6.2.4 for a discussion of computations). In similar fashion, the long-run sample variance, namely

\[
S^2(n) = \frac{\sum_{i=1}^{n} C_i^2 - n(\overline{C}(n))^2}{n-1},
\]

converges to \( \sigma^2 \) as \( n \) limits to \( \infty \). Consequently, we can control the quality of our estimates of the population mean and variance by the run lengths of the simulation.

The quality of the estimator, \( \overline{C}(n) \), of \( \mu \text{cost} \) depends upon the variance of \( \overline{C}(n) \). (Actually, we need its distribution too, but as we shall see, difficulties will arise with just its variance.) You might recall that if we assume \( C_1, \ldots, C_n \) are independent and identically distributed, an unbiased estimator for the variance of \( \overline{C}(n) \) is \( S^2(n)/n \). However, in a simulation study, the sequential random variables \( C_1, \ldots, C_n \) are often highly dependent, which introduces a (possibly large) bias into \( S^2(n)/n \) as an estimator for the variance of \( \overline{C}(n) \). Therefore, the sample variance (Equation 6.6) of the sequentially generated data should not be used with Equation 5.1 to build a confidence interval for \( \mu \text{cost} \). This fact should be stressed: when \( n \) is large, \( \overline{C}(n) \) is a valid estimator for \( \mu \text{cost} \), \( S^2(n) \) is a valid estimator for \( \sigma^2 \), but \( S^2(n)/n \) is not a valid estimator for the variance of \( \overline{C}(n) \).

6.2.1 Replicates

The most straightforward method of obtaining confidence intervals is to use several replicates of the simulation run (using different initial random number seeds), just as was done for the terminating simulation analyses. What this does provide is a way to create truly random samples by sequential

\(^2\)Technically, the equation should read "equal with probability one," but such fine points are not necessary for our purposes.

\(^3\)When a process has the property that the limiting time-averaged value and the value averaged over the state space are the same, the process is said to have the ergodic property.
6.2. STEADY-STATE SIMULATIONS

Time is usually at a premium, which causes a serious obstacle to use of the replication technique. With these considerations in mind, we investigate alternative methods for obtaining confidence intervals for mean values.

- Suggestion: do Exercises 6.6 - 6.8.

6.2.2 Batch Means

Let us review again the context of our estimation problem. We make a long simulation run (preferably containing a start-up period) obtaining \(m\) data points, \(C_1, \ldots, C_m\). We desire to estimate \(\mu_{\text{cost}}\), which is the long-run mean value; that is,

\[
\mu_{\text{cost}} = E[C_m]
\]

where \(m\) is assumed large. The most common estimator for \(\mu_{\text{cost}}\) is based on Equation 6.5, which indicates that the sample mean (Equation 4.2) is a good estimator. However, in most simulations \(C_i\) and \(C_{i+1}\) are highly dependent random variables, which is the reason that estimating the standard deviation of the estimator for \(\mu_{\text{cost}}\) is difficult.

As mentioned previously, steady-state results are independent of the initial conditions. This fact implies that if two costs are sufficiently far apart, they are essentially independent: that is, \(C_i\) and \(C_{i+k}\) can be considered independent if \(k\) is large enough. This leads to one of the most popular forms of obtaining confidence intervals: the method of batch means. The concept is that the data will be grouped into batches, and then the mean of each batch will be used for the estimators. If the batches are large enough, the batch means can be considered independent because, intuitively, they are far apart. Since the batch means are considered independent, they can be used to estimate not only the overall mean, but also the standard deviation. Specifically, let us take the \(m\) data points and group them into \(n\) batches, each batch containing \(\ell = m/n\) data points, and let \(\bar{C}_j\) denote the mean of the \(j^{th}\) batch; namely,

\[
\bar{C}_1 = \frac{C_1 + \cdots + C_\ell}{\ell} \\
\bar{C}_2 = \frac{C_{\ell+1} + \cdots + C_{2\ell}}{\ell} \\
\vdots
\]

\(^4\)To be technically correct in this paragraph, we need to discuss concepts and assumptions related to the ergodic property, but this is beyond the scope of this textbook. Here we only intend to give an intuitive feel for these principles.
where \( m = nl \). As \( t \) gets large, \( \overline{C}_j \) and \( \overline{C}_{j+1} \) become independent. Using the definitions in Equation 4.1, we have the following:

\[
\overline{X}_{\text{batch}} = \frac{\overline{C}_1 + \cdots + \overline{C}_n}{n} = \frac{C_1 + \cdots + C_m}{m},
\]

\[
S^2_{\text{batch}} = \frac{\overline{C}_1^2 + \cdots + \overline{C}_n^2 - n\overline{X}^2_{\text{batch}}}{n-1}.
\] (6.8)

The confidence interval is, thus, given by

\[
(\overline{X}_{\text{batch}} - t_{(n-1, \frac{\alpha}{2})} \frac{S_{\text{batch}}}{\sqrt{n}}, \overline{X}_{\text{batch}} + t_{(n-1, \frac{\alpha}{2})} \frac{S_{\text{batch}}}{\sqrt{n}}),
\] (6.9)

where \( t_{(n-1, \frac{\alpha}{2})} \) has the same meaning as it does in Equation 6.1.

The two major assumptions made in the development of the confidence interval of Equation 6.9 are: (1) the random variables \( \overline{C}_1, \ldots, \overline{C}_n \) are distributed according to a normal distribution, and (2) the random variables are independent. As long as the batch size is large, the central limit theorem permits the assumption of normality (a rule of thumb for using the central limit theorem is to have a sample size of at least 30). The independence assumption is more difficult to verify. One test for independence of the sequence is to estimate the correlation between adjacent values, namely, the correlation between \( \overline{C}_i \) and \( \overline{C}_{i+1} \), called the lag 1 correlation coefficient. (A zero correlation between adjacent values does not guarantee independence, but a nonzero value guarantees lack of independence.)

The correlation coefficient for random variables \( X \) and \( Y \) is given by

\[
\rho = \frac{E((X - E[X])(Y - E[Y]))}{\sqrt{\text{var}(X)\text{var}(Y)}}.
\]

The most common estimator for the lag 1 correlation coefficient for the sequence \( \overline{C}_1, \ldots, \overline{C}_n \) is given by

\[
\hat{\rho} = \frac{\sum_{i=1}^{n-1}[(\overline{C}_i - \overline{X}_{\text{batch}})(\overline{C}_{i+1} - \overline{X}_{\text{batch}})]}{\sum_{i=1}^{n-1}[(\overline{C}_i - \overline{X}_{\text{batch}})^2]},
\] (6.10)

where \( \overline{X}_{\text{batch}} \) is defined in Equation 6.8. A slightly easier formula (again using Equation 6.8) for computing this estimator is

\[
\hat{\rho} = \frac{\sum_{i=1}^{n-1} C_i \overline{C}_{i+1} - n \overline{X}^2_{\text{batch}} + \overline{X}_{\text{batch}}(C_1 + \overline{C}_n - \overline{X}_{\text{batch}})}{(n-1)S^2_{\text{batch}}}.
\] (6.11)

A rule of thumb for having confidence in the results obtained from the method of batch means is that the absolute value of the lag 1 correlation coefficient should be less than 0.05. However, a major problem with the estimator of Equation 6.10 is that it is highly variable. For example, the program on page 125 was modified to simulate the process for 2,002 days using a start-up period of 2 days. Using the last 2,000 days, the method of batch means was used to estimate a confidence interval where the size of each batch was 50 days. To determine the appropriateness of the method, the lag 1 correlation coefficient was estimated as \( \hat{\rho} = -0.04 \). If the -0.04 number is an accurate estimate for \( \rho \), then the one simulation run is all that is needed for the confidence interval. Therefore, to determine the accuracy of the \( \hat{\rho} = -0.04 \) estimate, the simulation was run five more times, each time being a run of 2,002 days. The five values for \( \hat{\rho} \) were 0.21, 0.25, -0.54, -0.13, and 0.07. The conclusion from this example is that the estimator for \( \rho \) is too variable to be reliable when only 40 batches are used; thus, more batches are needed in order to have a reliable estimator.

The difficulty with choosing a batch size is that if the batch size is large enough to obtain a small lag 1 correlation coefficient, then the number of batches will be too small to produce a reliable estimate; however, if the number of batches is large enough to produce a reliable estimate, then the batch size will be too small to maintain independence between batch means.

When the modeler has the freedom to choose the run length based on values dynamically determined within the simulation, he or she may use the textbook by Law and Kelton (mentioned at the beginning of this chapter) that contains a suggested approach for confidence intervals. The authors' procedure is to extend the run length until the absolute value of the lag 1 correlation coefficient determined from 400 batches is less than 0.4. In other words, for a given run length, the batch size necessary to produce 400 batches is used, and the lag 1 correlation coefficient is calculated based on those 400 batches. If the absolute value of the coefficient is greater than 0.4, one can infer that the batches are too small, and the run length is increased. Once the absolute value of the coefficient is less than 0.4, the run length is fixed and the batch size is increased by a factor of ten. The resulting 40 batches are then used to obtain the confidence interval. The justification for this approach is that when 400 batches produce a coefficient of less than 0.4 and then the batches are made 10 times larger, the resulting 40 batches should produce a coefficient of less than 0.05.

More commonly, the run length of the simulation cannot be dynamically determined, but must be fixed at the start of the simulation. A good "quick
and dirty estimate of confidence intervals may be made using a short restart period, dividing the remaining run length into five to ten batches, and then using those batches to estimate the confidence interval\(^3\). The idea of using a small number of batches is theoretically flawed since it is difficult to justify using the Central Limit Theorem; in practice, however, it does seem to perform adequately.

### Table 6.2: Individual data points from a simulation run of length 37 days from the machine repair example

<table>
<thead>
<tr>
<th>Week</th>
<th>Daily Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2929</td>
</tr>
<tr>
<td>1</td>
<td>2558 3958 1772 6096 3451 1094 1296</td>
</tr>
<tr>
<td>2</td>
<td>6338 11105 7774 3710 1062 1931 5406</td>
</tr>
<tr>
<td>3</td>
<td>2270 2182 4365 3065 7111 7570 11107</td>
</tr>
<tr>
<td>4</td>
<td>4276 4024 2340 2345 1665 4042 3345</td>
</tr>
<tr>
<td>5</td>
<td>1728 2376 3050 1476 1490 1201 2305</td>
</tr>
</tbody>
</table>

Table 6.2 contains data to be used in a numerical example. (We use a run length that is too small for accuracy, but it is better for illustrating the computations. Do not infer from this example that estimates should be made based on such short runs.) This example uses an initialization period of two days and a batch size of seven days that yields five batches. We calculate the average for each row in Table 6.2, which gives the five batch means as: \( \bar{z}_1 = 3.0179 \), \( \bar{z}_2 = 5.3324 \), \( \bar{z}_3 = 3.3514 \), \( \bar{z}_4 = 3.1483 \), and \( \bar{z}_5 = 1.9467 \). The sum of the five values is 18,826.6, and the sum of their squares is 80,202,043.44. The confidence interval calculations are

\[
\bar{z}_{\text{batch}} = \frac{18,826.6}{5} = 3,765.32
\]

\[
\hat{s}_{\text{batch}} = \sqrt{\frac{80,202,043.44 - 5 \times 3,765.32^2}{4}} = 2,328,567.4
\]

which yields a 95 percent confidence interval of\(^6\)

\[
(3765.3 - 2.776 \times \frac{1525.9}{\sqrt{5}}, 3765.3 + 2.776 \times \frac{1525.9}{\sqrt{5}}); \ (1870.9, 5659.7).
\]

Thus, we see that a run of this small size yields an estimate with a large degree of variability. With 95 percent confidence, we would say that the maximum error is 1,894.4. Let us consider what would have happened if we had improperly used the individual 35 daily values (i.e., a batch size of one) to obtain the confidence interval. In that case, we would have estimated the maximum error to be 878.5, which greatly understates the inherent variation of the 3,765.3 estimated value of the average daily cost.

- **Suggestion:** do Exercise 6.9.

### 6.2.3 The Regenerative Method\(^5\)

Steady-state simulations usually involve regenerative processes; that is, processes for which there are points in time in which the process probabilistically repeats, or regenerates, itself. For example, consider Figure 6.4, which contains a typical realization showing the number of machines in the repair room as a function of time. The times labeled \( R_1, R_2, \ldots \) are called regenerative points because the process regenerates itself at those points in time. In other words, when the system is at time \( R_1 \), the probability law governing the future evolution of the process is exactly the same as at time \( R_2 \). Furthermore, the intervals of time between regenerative points are called regenerative cycles. The key property that makes regenerative processes helpful is that the sequence of regenerative cycles forms an independent, identically distributed process. Thus, the conceptual approach of the regenerative method is to take advantage of the fact that what happens during the successive regenerative cycles can be considered as independent events.

The regenerative method is illustrated by applying it to the previous machine repair example problem, which determined the long-run cost per unit time. Let the costs incurred during the sequential regenerative cycles be denoted by \( \bar{z}_1, \bar{z}_2, \ldots \), and let the lengths of the cycles be \( \bar{T}_1, \bar{T}_2, \ldots \). In other words, if we use the notation in Figure 6.4, \( \bar{z}_1 \) is the sum of the costs incurred by the customer who arrives at time \( R_1 \) and all customers arriving after time

---


\(^6\)This subsection involves slightly more complicated expressions and can be skipped.
6.2. STEADY-STATE SIMULATIONS

The main difficulty with using the regenerative method is that the dependency between the numerator and denominator of each ratio results in a complicated expression for the variance of $\mu_{\text{cost}}$ in (6.12). Let $S_{\text{cost}}^2$ be the sample variance from the data $\bar{C}_1, \ldots, \bar{C}_m$, and let $S_{\text{time}}^2$ be the sample variance from the data $\bar{T}_1, \ldots, \bar{T}_m$. Finally, let $S_{\text{cov}}$ be the estimate of the covariance between the costs and the times; that is,

$$S_{\text{cov}} = \frac{\sum_{k=1}^m \bar{C}_k \bar{T}_k - m \bar{X}_{\text{cost}} \bar{X}_{\text{time}}}{m - 1},$$

where $m$ is the number of regenerative cycles. The variance estimate for the estimator $\mu_{\text{cost}}$ is now given as

$$S_{\text{ratio}}^2 = S_{\text{cost}}^2 - 2 \mu_{\text{cost}} S_{\text{cov}} + \mu_{\text{cost}}^2 S_{\text{time}}^2.$$  \hspace{1cm} (6.14)

The central limit theorem can now be invoked for regenerative processes, yielding a confidence interval given by

$$\left(\mu_{\text{cost}} - \frac{t_{(m-1, \frac{\alpha}{2})} S_{\text{ratio}}}{\sqrt{m}}, \mu_{\text{cost}} + \frac{t_{(m-1, \frac{\alpha}{2})} S_{\text{ratio}}}{\sqrt{m}}\right).$$  \hspace{1cm} (6.15)

The $t_{(m-1, \frac{\alpha}{2})}$ value again comes from tables giving the Student-t distribution, using a confidence level of $1 - \alpha$. Since the central limit theorem needs to be invoked, the value of $m$ should be at least 30 or 40 so that the standard normal tables can be used instead of the t-tables.

The example chosen to illustrate the regenerative method was applied to cost per unit time; however, it might also be desirable to estimate the cost per customer. The regenerative method is easily adapted to a per customer value by simply changing the variable of the denominator. If the quantity $\bar{T}_i$ is the number of machines arriving during the $i^{th}$ regenerative cycle instead of the total time per cycle, then the procedure outlined by Equations 6.12 - 6.15 yields the expected average cost per customer.

The appropriateness of the regenerative method does not depend on anything like the correlation coefficient estimate since the method itself guarantees suitable independence. However, not only is the regenerative method more difficult to apply, but also its use depends on identifying suitable regenerative points. In a busy and complex system, the regenerative cycles might be so long that the run length would need to be too long to be practical. However, if suitable regenerative points can be identified, the regenerative method is a good method to use to assure theoretically correct results.
6.2.4 Supporting Language Structures

Most simulations provide (automatically or under user selection) standard statistical summaries of the usage of resources, queues, chains, and so on; in fact, the necessary computations are part of the internal mechanism for any block statement referencing these constructs. While languages may differ regarding the types of information they provide, most of them at least compute means and variances. The issue we wish to address here is how to compute such quantities, focusing attention on means and variances.

Because the data generated by a simulation is dynamic (that is, they depend upon time) it is necessary to use a technique called time-weighted averages. To illustrate, consider a sequence of arrival times and service times for machines coming to a repair facility: at time 9 minutes, the first arrival occurs; at time 55 minutes, the second arrival occurs; at time 64 minutes, the third arrival occurs; at time 71 minutes, the first arrival leaves; at time 83 minutes, the second arrival leaves; and at time 94 minutes, the third arrival leaves. The average number of items in the system over the 94-minute interval is the time-weighted average of the number of items present. The calculations for this would be

\[
\frac{1 \times (56 - 9) + 2 \times (64 - 56) + 3 \times (71 - 64) + 2 \times (83 - 71) + 1 \times (94 - 83))}{94} = 1.266.
\]

An estimate for the variance would be

\[
\frac{1^2 \times (56 - 9) + 2^2 \times (64 - 56) + 3^2 \times (71 - 64) + 2^2 \times (83 - 71) + 1^2 \times (94 - 83))}{94} - 1.266^2 = 0.535.
\]

Note that there is 1 machine in the facility from time 9 to time 56, 2 in the system from time 56 to time 64, and so on. What we are doing is looking at the number in the system at instants in time where a change occurs. At each such time, we update the cumulative sum of the product of number \times time, where time is the period since the last update. Once we have finished this calculation, we divide the cumulative sum by the total elapsed time to obtain the time-weighted average. (Of course, this type of calculation for obtaining estimates should only be used after long runs.)

The data structure VARSTATISTICS (for variable statistics) is designed to be helpful in obtaining time-weighted averages for specified variables. The syntax for its definition in the DEFINITION segment is

\[
\text{name : VARSTATISTICS = \{ Value = \text{xx, Protect = On'/Off};
\]

where \text{Value} is required and gives the initial value of the variable. If Protect is omitted it is set to \text{Off} and then the variable is not protected (i.e., it is cleared) during the \text{RESET} or \text{CLEAR} command within the \text{EXITLOGIC}. Whenever the \text{VARSTATISTIC} is used within the \text{LOGIC} segment, its time-weighted average is computed, assuming that there has been no change in its value since the last time an entity passed through its assignment block.

We illustrate the use of VARSTATISTICS by estimating the probability that a simple single-server system is empty. The variable \text{empty} (of type VARSTATISTICS) will be used as an indicator variable: \text{empty} = 1 when there are no entities in the system, and \text{empty} = 0 when at least one entity is in the system. MOR/DS automatically performs the bookkeeping for time-weighted averages each time the value of \text{empty} is updated. The mean value of \text{empty} will then be an estimate for the long-run probability that the system is empty.

\[
\text{PROGRAM "example for the use of VARSTATISTICS"}
\text{DEFINITION}
\text{machine : RESOURCE = \{ \};
\text{empty : VARSTATISTICS = \{ Value = 1 \};
\text{CONTROL}
\text{STOP Time = 10000;}
\text{LOGIC}
\text{ARRIVE \{ Time = Expd(10) \};
\text{empty = 0;}
\text{SEIZE \{ Name = machine \};
\text{WAIT \{ Time = Expd(9) \};
\text{RELEASE \{ Name = machine \};
\text{IF Used[machine] = 0 THEN empty = 1;
\text{DEPART \{;
\text{END.}
\]

- Suggestion: do Exercise 6.10.

6.2.5 Comparisons

To illustrate the various approaches for determining confidence intervals for steady-state simulations, Table 6.3 presents confidence intervals based on 30 runs of the example problem. Due to the ease of implementation, our recommendation is to use the method of batch means. When the run length is fixed, it appears adequate to use a small initialization period and
then fix the batch size so as to have five batches. To emphasize again the inappropriateness of naively treating all data as a random sample (i.e., using a batch size of one), the maximum error observed using the 2,000 daily costs as a random sample understates the true variability of the results by approximately 50 percent.

One advantage of the batch mean approach and the regenerative method over the replicate approach is that a single run is used to obtain the confidence interval. Making 30 runs, as in our example, is extremely time consuming. However, since 30 runs were made for the replicate method, the error reported in Table 6.3 is the average value from all 30 runs. The last column in the table gives an indication of the variability of the error estimator among the 30 runs. The error for the replicate method is the smallest, but it must be remembered that the error for the replicate approach is based on 30 runs, whereas the error for the other methods is based on a single run.

Table 6.3: Comparisons among the various steady-state confidence interval estimation techniques for the machine repair problem

<table>
<thead>
<tr>
<th>Method</th>
<th>Mean estimate</th>
<th>Max. error</th>
<th>Std. dev. of max. error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replicates (2-day restart)</td>
<td>4,320.9</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>Batch means size=50 days</td>
<td>4,320.9</td>
<td>436</td>
<td>119</td>
</tr>
<tr>
<td>Batch means size=400 days</td>
<td>4,320.9</td>
<td>399</td>
<td>175</td>
</tr>
<tr>
<td>Regenerative</td>
<td>4,326.7</td>
<td>453</td>
<td>222</td>
</tr>
</tbody>
</table>

6.3 Exercies

6.1. Consider the service station example of Section 6.1 and assume that the profit per customer can be approximated by multiplying the service time at the pump in minutes by $0.50. Collect data from 40 independent observations of the one pump station and give a 95 percent confidence interval for the expected profit per day under the following two conditions:

(a) Assume that the daily profit is normally distributed.

(b) Assume that the means from groups of size 5 are approximately normally distributed.

6.2. The manager of the service station (cf Section 6.1) has determined that a second pump can be leased for $10 per day. (The second pump reduces the waiting space by one unit.) Should the manager take advantage of the lease offer? How confident are you of your recommendation?

6.3. Give a 95 percent confidence interval for the daily revenue of the corner newspaper stand of Exercise 5.2.

6.4. Give a 95 percent confidence interval for the daily number of customers lost by the barber shop in Exercise 5.4.

6.5. Estimate the distribution of time that customers must spend in the shop in Exercise 5.4.

6.6. Consider an M/M/1 queuing system in steady-state. (That is, a single-server queue with exponential time between arrivals and exponential service times.) The arrival rate to the system is 30 arrivals per hour, and the average time each customer spends in the server is 1 minute. Since this is a simple system, various measures can be derived analytically. For example, the mean waiting time in the system is 2 minutes per customer, the mean number of customers in the system is 1, and the probability that there are n customers in the system at an arbitrary point in time is 1/2^n.

Develop a model to simulate this system and obtain a 95 percent confidence interval for the mean waiting time, mean number in the system, and the probability that n customers are present for n = 0, · · · , 5.

(a) Use five replicate runs with a run length of 900 completed customers.
(b) Use the batch means approach with a single run-length of 4,500 completed customers using five batches. (Hint: use the VARSTATISTICS construct.)

7. Using the method of replications, make a recommendation as to whether or not a second repairman should be hired in the example of Section 6.2.

8. Use the method of replications and give a 95 percent confidence interval for the average number of occupied parking spaces at steady-state in Situation 3 of Section 5.5. What is the average number of cars and trucks turned away each day?

9. Assuming a steady-state analysis is appropriate, use the method of replications to obtain 95 percent confidence intervals in answering the following questions regarding Situation 4 of Section 5.5. What is the average length of time a job waits until it first accesses the CPU? What is the average length of time a job spends in the system? What is the average number of jobs within the system at an arbitrary point in time? What is the main bottleneck for the computer system?

10. Repeat the analysis for Exercises 6.6 - 6.8 except use the method of batch means.

11. Repeat the analysis for Exercises 6.6 - 6.8 except use the regenerative method.

Chapter 7

Complex Simulation Models

The definition and block types defined in Chapter 5 allow for developing a wide variety of models. However, as situations become more complex, more advanced programming capabilities are needed. In this chapter, we present additional concepts and programming structures useful for modeling complex processes. The advanced block types introduced are common to most block-oriented languages. The additional programming statements are not available in most simulation languages, but they are included in MOR/DS because of the flexibility they permit. The approach taken in this chapter is to present the advanced statements by way of examples. Thus, the introduction of new data structures and program statements are illustrated with detailed examples. For a concise description of all types and structures, see Appendix B.

7.1 Preempting Resources

Many situations arise in which one entity preempts service from another entity. For example, it is common within large computer systems to define priorities so that jobs within one priority class have the capability of forcing other jobs to terminate their service prematurely so that the CPU becomes immediately available. The MOR/DS simulation language allows entities in control of a resource to be preempted and replaced with another entity. The preempting action is based on the relative values of the system entity attribute, PreemptLevel, with lower numbers being of higher priority value. Entities may be preempted if their PreemptLevel values are greater than the preempting entity's PreemptLevel value. This preempting activity is
CHAPTER 7. COMPLEX SIMULATION MODEL

7.1. PREEMPTING RESOURCES

There are two keys to keep in mind when using this block. The first is that entities that are preempted are placed in front of the queue (in the absence of the PreemptedTo parameter) with no check being made on the queue capacity limits. Therefore, it is possible to have more entities in a queue than specified by the Capacity parameter. Also, entities that arrive after the preemption occurs might be placed in front of the preempted entity if there is available capacity and if a queue discipline (priority scheme) is used that allows such a placement. The second consideration to keep in mind regarding the PREEMPT block is that an entity can only be preempted if its PreemptLevel has a higher numerical value than the entity entering the PREEMPT block and if the entity is in a WAIT block. For example, an entity that has seized one resource and then been placed on a queue for a second resource cannot be preempted from the first resource as long as it remains on a queue and not in a WAIT block.

The default options for the PREEMPT block are one unit requested and the Resume option set to ON. The Name parameter is required. The resource being preempted must, of course, have been defined via a RESOURCE definition statement. The following example illustrates a situation where the PREEMPT block is a useful tool for modeling.

7.1.1 Example

We will illustrate the PREEMPT block by using an example involving changing shifts. The concept is that the first shift is capable of more capacity than the second shift. When a change occurs from the first to the second shift, the reduction in capacity is modeled by preempting entities within any resources that can no longer be serviced due to the reduction in capacity. The situation is as follows.

Parts arrive at a work station according to an exponential distribution with a mean of 10 minutes between arrivals. The work station is manned by two different classes (A and B) of workers. The parts require two operations. The first operation uses a class A worker and takes between 3 and 5 minutes (uniformly distributed) to complete. Parts must be cooled for a minimum of 2.5 minutes before the second operation can begin. The second operation requires 2 class B workers and takes between 2.5 and 4.5 minutes (uniformly distributed). The work station has two shifts, each lasting 12 hours. There are 2 class A workers and 4 class B workers on the first shift and 1 class A worker and 2 class B workers on the second shift. The resulting program to model this situation is as follows.
7.2. Combining and Splitting Entities

It is often necessary to model packaging operations or production and assembly processes consisting of various order-splitting and recombining scenarios. The MOR/DS language supports five commands related to copying and recombining of entities. These are: COPY, COMBINE, GATHER, BUNDLE, and UNBUNDLE. This section discusses the use and syntax of these commands.

7.2.1 The COPY and COMBINE Blocks

The COPY command allows the duplication of the activating entity, and the specification of the block label identifying the statement where these copies are to enter the model. This block has two parameters: Quantity is the number of duplicates to be made and is a required parameter, and CopiesTo is an optional destination for the copies. Thus, if a COPY block is used to make five copies of a specific entity (i.e., Quantity = 5), then
actually there will be six such entities in the system, the original plus the five copies. Copies are sent to the next sequential block if the CopiesTo parameter is not used. The block syntax is:

COPY { Quantity = expression, CopiesTo = label };

It should be noted that even though the COPY block duplicates the entering entity’s attributes (i.e., both system and user-defined attributes), only the original entity has control of any resources that it may have seized (via the SEIZE or PREEMPT block) prior to the COPY operation and other constructs such as entries into statistics collection devices, etc. To reiterate this important point, a copy of an entity cannot release a resource that the original entity seized. The COPY block does not duplicate the system information related to seized entities. This point is of particular importance when recombining entity copies.

The COMBINE block groups entities into a single survivor entity. The (required) Quantity parameter of this command specifies the total number of entities that are to be reduced into the single (survivor) entity. The surviving entity is always the last entity to arrive at the COMBINE block and, therefore, is not necessarily the original (copied) entity. An optional Decrement parameter allows the modeler to specify the number of units that the entering entity decrements from the original Quantity value for the current entity. The default value for this parameter is one unit.

The COMBINE block operates as follows. The first entity of a group establishes the initial Quantity value. Then a running count of the current value is decremented by each entity that enters the COMBINE, including the initial entity. The decrease in the running count is by one per entity, unless the Decrement keyword is used in the parameter list. When this running count becomes less than or equal to zero, the entity that causes the zero threshold to be crossed is allowed to pass through the COMBINE block. Thus, only the last entity of the group survives the COMBINE block; all preceding entities are destroyed. The next entity to enter the COMBINE block after the survivor has passed through the block activates a new Quantity count and the process is repeated.

An option of the COMBINE block allows entities to be combined based on a grouping or matching value of a specified entity attribute. That is, the modeler may want to combine entities whose attribute, say quantity, has matching values. The COMBINE block keyword UseAttribute specifies which user-defined entity attribute is to be used for the comparison. A running Quantity counter is kept, therefore, for each distinct value of the grouping attribute for entities that enter the COMBINE block. This option, while affording the modeler a wide latitude of control over the entity combining operation, requires considerable care to insure its proper operation. If the UseAttribute option is not specified, then entities are combined as if they were all members of the same group, that is indiscriminately.

The COMBINE block syntax is:

COMBINE { Quantity = expression, Decrement = expression, UseAttribute = user-attribute };

Example

Consider a packaging operation where quart cans of oil are to be grouped together into cases for shipment. Individual quart cans arrive randomly at the packaging center such that the time between arrivals is uniformly distributed (continuous) between 6 and 15 seconds. All items are of the exact same type, grade, and brand of oil and are to be packaged into 24-unit cases. Assuming that the packaging operation itself takes essentially no time per case, how many cases are completed in one hour? A complete model to answer this question is given below.

PROGRAM "Simple 24 unit packaging operation"

DEFINITION

CONTROL

   StopTime = 60; "time unit is one minute"

LOGIC

   ARRIVE { Time = Cuniform(0.1,0.25) };
   COMBINE { Quantity = 24 }; "default Decrement is 1 unit"
   DEPART {};

END.

To embellish this model slightly, assume that there are two distinct types of units to be packaged into separate cases. There are two different containers of the same motor oil, a brand name container for 10W30 and a generic container for 10W30. These differ: units arrive intermingled to the packer who sorts them while placing them into the proper containers. Assuming the same arrival rate as before with 1/3 of the items being the generic brand, how many cases of each type are produced in one hour? The program below answers this question. Notice that there are two DEPART blocks so that separate counts can be maintained on the two different types of departures.
7.2. COMBINING AND SPLITTING ENTITIES

Under the assumption that only one subcomponent is required for each of the sub-processes and that these are made-to-order components (that is, the subcomponent for an order must be recombined back with that specific order), this process can be modeled as follows. We assume that the processing times are all approximated by a random variable with a triangular distribution whose parameters depend on the process and that each processing center has only one server/machine available.

PROGRAM "Example Model for a COPY/COMBINE Process"
DEFINITION
initProc : RESOURCE = { Capacity = 1 };
leftProc : RESOURCE = { Capacity = 1 };
rightProc : RESOURCE = { Capacity = 1 };
finalProc : RESOURCE = { Capacity = 1 };
orderNumber = 0;
ATTRIBUTES = [ thisOrder = 0 ];
LABELS = [ leftComponent, rightComponent, finalAssembly ];

CONTROL
StopTime = 100;

LOGIC
ARRIVE { Time = Expd(3) };
orderNumber = orderNumber + 1; "global count"
thisOrder = orderNumber; "count placed on entity for combining"
SEIZE { Name = initProc };
WAIT { Time = Triangular(0.5,2,3.5) };
RELEASE { Name = initProc };
COPY { Quantity = 1, CopiesTo = rightComponent };

leftComponent:
SEIZE { Name = leftProc };
WAIT { Time = Triangular(1,4,8) };
RELEASE { Name = leftProc };
GOTO finalAssembly;

rightComponent:
SEIZE { Name = rightProc };
WAIT { Time = Triangular(0.5,6) };

A typical use of the COPY block in conjunction with the COMBINE block is for modeling production systems where an order is received and processed, then separated into two or more parts that undergo concurrent but independent subcomponent processing, and finally these subcomponents are combined to produce a completed product. The COPY block is used to separate the order so that the subcomponents can be processed independently. Then the COMBINE is used to assemble these components into a composite product. To illustrate this concept with an example model, consider a production process that has a flow diagram as depicted below.
7.2. COMBINING AND SPLITTING ENTITIES

In this model, the COPY block creates an entity that represents the processing that must occur on the right branch while the original entity represents the processing that occurs on the left branch. Both the active entity that caused the copy and the newly created entity contain a unique number in their attribute variable thisOrder so that during the COMBINE operation the original entity can only be combined with its own copy. The COMBINE block takes advantage of the number stored in thisOrder through the use of the UseAttribute parameter.

This production process model assumed that only one unit was needed for each subcomponent. There are situations when multiple subcomponents will be needed. For example, let this assembly process represent a stereo equipment installation center for automobiles. If the leftComponent represents the installation of a receiver and the rightComponent represents the installation of the speakers, then the normal quantities to be installed would be one receiver and two speakers. For this situation, the above model would only need minor changes. The COPY block parameter Quantity would be changed to 2 units and the COMBINE parameter Quantity would not be changed to 3 units. This model operates under the assumption that there is an individual who installs the receiver and a different individual who installs the speakers.

Another similar situation is a variation of this model is a production process where 2 units of the left component are to be assembled with 3 units of the right component. All left components are identical, as are all right components; hence, any two left components, regardless of their starting order, can be combined with any 3 components from the right line. To incorporate these changes, the parameter Quantity of the COPY block is set to 3, generating 3 right-side components per order, and a second COPY block is added immediately following the first COPY to generate another entity for the left-side component (Quantity = 1) with the CopiesTo destination set to leftComponent. Then, for each order, there would be the correct number of items of each component processed. All that remains then is to assemble them into the final product. Note that the assembly operation merely works on the first-arriving components, since there is no distinction between parts, and they need not be assembled with order integrity. The assembling operation can be accomplished by first developing two counters (in place of the single count orderNumber), cntL and cntR, placed immediately after the corresponding component processing RELEASE block in the leftComponent and rightComponent program segments, respectively. The proper value to assign to each item then can be computed for the left side components as

\[
\text{thisOrder} = \text{Ceiling}(\text{cntL}/2);
\]

and the right side components as

\[
\text{thisOrder} = \text{Ceiling}(\text{cntR}/3);
\]

These statements assign equal values to pairs of entities on the left side and equal values to every three entities on the right side. These statements are inserted following the cntL and cntR updates, and, in conjunction with a corresponding combine Quantity increase to 5, solve this particular problem. The UseAttribute parameter is used just as the previous example program illustrates. (The UseAttribute parameter is needed to ensure that all items combined are not all left or right items.)

7.2.2 The BUNDLE and UNBUNDLE Blocks

The COPY and COMBINE operations are permanent entity actions in the sense that COPY creates new independent entities, and COMBINE destroys all but the last entity of the combine group. There are instances when one would like to temporarily group entities and later break out the original entities for further processing. For example, consider a production process that manufactures items (or collects items from stock) as they arrive for processing. These items are collected together temporarily for shipping to the next processing station. Then the items are again treated as distinct orders for further processing. The temporary grouping of items as a shipment group, transportation pallet, etc., is a common situation encountered in modeling. The MOR/DS language supports this temporary grouping of entities with the BUNDLE and UNBUNDLE blocks.

As entities enter the BUNDLE block, they are grouped together and a single entity (called a surrogate) is created to represent each group. The
surrogate leaves the block when the last entity of the group arrives. The
BUNDLE command has essentially the same parameter options as does the
COMBINE block. The BUNDLE and COMBINE blocks operate similarly except
that entities bundled together can later be broken out again via the
UNBUNDLE block. This is distinct from the operation of a COMBINE
and later a COPY to recreate the entities, since the COPY creates multiple
copies of identical entities; whereas, the UNBUNDLE restores the original
entities with their respective and most likely distinct attribute values. Bun-
dled entities also do not lose control or reference to any resources that
they have previously seized and as yet not released. One additional option of
the BUNDLE block is the choice of the attribute values to be assigned to
the surrogate entity representing the bundle. This entity can either use the
attribute values of the first or last entering entity into the BUNDLE block.
The Surrogate option allows this choice, the default being the last entity
into the bundle. The BUNDLE block syntax is:

\[
\text{BUNDLE \{ Quantity = expression, Decrement = expression, } \\
\text{ UseAttribute = attribute\_name, Surrogate = First/Last \};}
\]

The UNBUNDLE statement has no parameters since it merely unlinks
the bundled entities from a temporary linked-list with their destination upon
remergence into the active blocks being the statement following the UN-
BUNDLE block. The surrogate entity is destroyed in the UNBUNDLE
block.

An example illustrating the operation of the BUNDLE/UNBUNDLE
blocks follows. This model merely collects 6 identical items, from a classi-
fication of three types, into a bundle. Then the surrogate entity requests
a resource (transport vehicle) and waits a given period of time representing
the time for the transport vehicle to arrive, collect, and deliver the bundle
to its destination. The individual entities are then broken out again and
counted by item classification.

**PROGRAM**

"Simple shipping-grouping example"

**DEFINITION**

\[
\text{ATTRIBUTES = \{ type = 0 \};}
\]

\[
\text{waitQ : QUEUE = \{\};}
\]

\[
\text{vehicle : RESOURCE = \{ Capacity = 1, Queue = waitQ \};}
\]

**CONTROL**

\[
\text{StopCount = 60; Seed = 1331;}
\]

**7.2. COMBINING AND SPLITTING ENTITIES**

**LOGIC**

```
ARRIVE \{ Time = Expd(5) \};
type = Uniform(1,3);
BUNDLE \{ Quantity = 6, UseAttribute = type \};
"bundle completed and call for transportation"
SEIZE \{ Name = vehicle \};
WAIT \{ Time = Expd(30) \};
RELEASE \{ Name = vehicle \};
"bundle arrived, break up bundle and count by type"
UNBUNDLE \{ \};
IF type = 1 THEN DEPART \{ Quantity = 1 \};
IF type = 2 THEN DEPART \{ Quantity = 1 \};
IF type = 3 THEN DEPART \{ Quantity = 1 \};
END.
```

The block counts from the execution of this simulation model are dis-
played below. From this information it is apparent that 10 bundles of 6
items each were created and transported during the simulation. Statements
4 through 6 indicate that only ten surrogate entities were transported,
hence, 7C entities total have been processed in the UNBUNDLE block
(statement 7) with the ten surrogates being destroyed and the 60 unbundled
entities leaving the block to be counted by type. There were 4 bundles of
type 1 (24 departures, statement 8), and three bundles each of the other two
types.

<table>
<thead>
<tr>
<th>Stat</th>
<th>Line</th>
<th>Label</th>
<th>Name</th>
<th>Total</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>ARRIVE</td>
<td>71</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>ASSIGNMENT</td>
<td>70</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>BUNDLE</td>
<td>70</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>SEIZE</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>WAIT</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>RELEASE</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>UNBUNDLE</td>
<td>70</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>DEPART</td>
<td>24</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>DEPART</td>
<td>18</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>DEPART</td>
<td>13</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 7. COMPLEX SIMULATION MODEL.

The last of the entity grouping statements is the GATHER block. This block holds entities, in designated attribute variable value classes if specified, until the proper number of entities has arrived at the block. Then all these entities are released simultaneous with the last arrival. This block serves to delay and hence coordinate groups of entities. Its syntax is

\[
\text{GATHER} = \{ \text{Quantity} = \text{expression}, \text{UseAttribute} = \text{attribute}_{\text{name}}, \text{Decrement} = \text{expression} \};
\]

Example

Our last example of this section illustrates the COPY, GATHER, and PREEMPT blocks. Consider a full service gas station with four multiple purpose pumps and a single service attendant. The company policy is for the attendant to wash each customer’s windows, and check the tires, water, oil, belts, etc. This takes the attendant an exponentially distributed amount of time with a mean of 3 minutes per customer (longer times are due to services required to correct any problems found during these inspections). Customers arrive at the station with a mean interarrival time of 3 minutes exponentially distributed. If they are not able to obtain one of the four gas pumps immediately, they leave and either return at a later date (outside the scope of our analysis) or they obtain service elsewhere. The station owner realizes that his business will suffer if customers waiting at the pumps are not attended promptly. He has determined that the attendant should service customers in their order of arrival, except that he should always be greeted as soon as possible and the gas filling operation started. The automatic cutoff mechanism will stop the filling operation at the requested units of gas. The attendant, therefore, is to interrupt the washing and checking services and meet any new arrivals as soon as possible. The greeting and startup operation take only 1 to 2 minutes (according to a continuous uniform distribution), while it then takes another 3 to 5 minutes (also continuous uniform) to complete the gas filling operation without the attendant being present. The service operation (washing, oil check, etc.) can take place while the gas filling operation is being completed. The attendant also has a final responsibility, that of taking payment for the gas and any other items needed. The payment operation is not interruptible and takes a mean time of one minute per customer exponentially distributed. Customer payments are serviced in their order of completing the gas filling operation and this process also has preemptive priority over the general service process. Thus, once the gas filling operation is complete, the attendant will temporarily halt checking the oil (if necessary) to take the customer’s payment.

The owner would like to model this operation to ascertain the utilization of the service attendant and the average delays encountered by customers waiting for the attendant services. The simulation model follows.

PROGRAM "Service station attendant model"
DEFINITION

\[
\begin{align*}
\text{attnQ} & : \text{QUEUE} = \{ \text{Discipline} = \text{Choice(PreemptLevel)} \}; \\
\text{attn} & : \text{RESOURCE} = \{ \text{Capacity} = 1, \text{Queue} = \text{attnQ} \}; \\
\text{pumps} & : \text{RESOURCE} = \{ \text{Capacity} = 4 \}; \\
& \text{ATTRIBUTES} = \{ \text{PreemptLevel} = -1, \text{cust} = 0 \}; \\
\text{LABELS} & = \{ \text{Thru}, \text{Service}, \text{Lost} \}; \\
\text{custCount} & = 0;
\end{align*}
\]

CONTROL

\[
\begin{align*}
\text{StopTime} & = 8*60; \\
\text{WatchList} & = \{ \text{attnQ:}’Q’, \text{attn:}’A’, \text{pumps:}’P’ \};
\end{align*}
\]

LOGIC

\[
\begin{align*}
& \text{ARRIVE} \{ \text{Time} = \text{Expd}(3) \}; \\
& \text{custCount} = \text{custCount} + 1; \\
& \text{cust} = \text{custCount}; \\
& \text{SEIZE} \{ \text{Name} = \text{pumps}, \text{Rexcess} = \text{Lost} \}; \\
& \text{PREEMPT} \{ \text{Name} = \text{attn} \}; \\
& \text{WAIT} \{ \text{Time} = \text{Cuniform}(1,2) \}; \\
& \text{RELEASE} \{ \text{Name} = \text{attn} \}; \\
& \text{COPY} \{ \text{Quantity} = 1, \text{CopiesTo} = \text{Service} \}; \\
& \text{WAIT} \{ \text{Time} = \text{Cuniform}(3,5) \}; \\
& \text{PREEMPT} \{ \text{Name} = \text{attn} \}; \\
& \text{WAIT} \{ \text{Time} = \text{Expd}(1) \}; \\
& \text{RELEASE} \{ \text{Name} = \text{attn} \}; \\
\end{align*}
\]

Thru:

\[
\begin{align*}
& \text{GATHER} \{ \text{Quantity} = 2, \text{UseAttribute} = \text{cust} \}; \\
& \text{IF Preemp-Level} = 0 \text{ THEN DEPART} \{ \}; \\
& \text{RELEASE} \{ \text{Name} = \text{pumps} \}; \\
& \text{DEPART} \{ \};
\end{align*}
\]

Lost:

\[
\begin{align*}
& \text{DEPART} \{ \};
\end{align*}
\]

Service:

\[
\begin{align*}
& \text{PreemptLevel} = 0;
\end{align*}
\]
7.3 Linking and Unlinking Entities

The internal structures used in block oriented languages for queues and for keeping track of the future events that must be processed are linked lists (also called chains). When entities try to seize a resource but cannot, they are placed on a linked list, although the placement is transparent to the user. However, there are times when the user needs to control the placement of entities on linked lists instead of letting it happen automatically. The LINK and UNLINK blocks are designed to allow the user to control the placement of entities on linked lists. The lists used for the linking and unlinking are called CHAINS. A CHAIN is defined in the DEFINITION segment exactly like a QUEUE. Functionally, a CHAIN and a QUEUE are the same, except that a LINK may not be used with a QUEUE and entities on a CHAIN cannot be released automatically by a RESOURCE. (The reason to use two different names is to protect the programmer from accidently linking an entity to a QUEUE and then thinking that a RESOURCE will have access to that entity and could remove it during a release operation.) The only way to remove a linked entity from a CHAIN is via the UNLINK block.

The syntax for the LINK block is

\[
\text{LINK} \{ \text{Name} = \text{c\#ame}, \text{Cexcess = nyl\#abel} \};
\]

where Name is required and indicates the name of the chain on which the link is to occur. The optional branching label designated by Cexcess indicates the location to which the entity will branch if the chain is full. The named chain must have been defined in the DEFINITION segment. The syntax for the UNLINK block is

\[
\text{UNLINK} \{ \text{Name} = \text{c\#ame}, \text{Quantity = expression, Destination = label, Condition = bool\#Exp} \};
\]

where Name, Quantity, and Destination are required. Quantity indicates the number of entities (if they exist on the chain) to be unlinked from the chain identified by Name. Each of the entities that are unlinked are sent to the location identified by Destination. If the boolean expression in Condition evaluates to false (value of zero), then unlinking will not occur; otherwise, unlinking occurs. Entity attributes referenced in the Condition expression refer to entities on the chain, not the entity activating the unlinking operation. For example, if an entity attribute type has been defined and we would like to unlink all entities in myChain that have type = 1, then we write...
7.3. LINKING AND UNLINKING ENTITIES

CONTROL
StopCount = 20;
WATCHLIST = { BackLog:'B', OnHand:'I' };

LOGIC

"initialize inventory level"
ARRIVE { Time = 0, Quantity = InitialInventoryLevel, Limit = 1 }
LINK { Name = OnHand };

ARRIVE { Time = Expd(MeanTime) }; "Customers"
IF Used[OnHand] > 0 THEN
begin
    UNLINK { Name = OnHand, Quantity = 1,
        Destination = ClearItem };
    InvPosition = InvPosition - 1;
    IF InvPosition <= ReOrderPoint THEN
        COPY { Quantity = 1, CopiesTo = PlaceOrder };
    DEPART { Quantity = 1 }
end
ELSE
begin
    InvPosition = InvPosition - 1;
    IF InvPosition <= ReOrderPoint THEN
        COPY { Quantity = 1, CopiesTo = PlaceOrder };
    LINK { Name = BackLog };
end;

PlaceOrder:
InvPosition = InvPosition + Q;
WAIT { Time = Normal(MeanLeadTime, StdLeadTime) MAX 0 };
COPY { Quantity = 0, CopiesTo = Shelve };
DEPART { };

Shelve:
IF Used[BackLog] > 0 THEN
begin
    UNLINK { Name = BackLog, Quantity = 1,
        Destination = FreeBackOrder };

Example

We consider a continuous review inventory policy where a reorder point
and order quantity have been established. The situation is as follows. A
company has decided to observe continuously their inventory position. (In-
ventory position refers to the amount of inventory on hand and the amount
of inventory on order that has not yet arrived.) Whenever the inventory pos-
tion falls below the reorder point, an order for more inventory will be placed.
Because of a transportation factor and processing time, there is a lead time
that represents the time between when the order was placed and the actual
time at which the order arrives. The lead time can be approximated by a
normal distribution with a mean of 5 days and standard deviation of 1 day.
The order quantity is always the same, and it has been decided to set the
reorder point equal to the average amount of demand during the mean lead
time interval. Demand for the inventoried items occurs one-at-a-time, and
the time between demands is exponentially distributed with a mean time of 5
days. Demand for items for which there is no inventory results in backorders
that are supplied as soon as an order is received.

The resulting simulation program is displayed below. In the program,
the system variable Used[chain.name] is utilized, which contains the current
number of entities on the chain identified by chain.name.

PROGRAM "Inventory Example"

DEFINITION
BackLog : CHAIN = { };
OnHand : CHAIN = { };
Q = 5; "order quantity"
MeanLeadTime = 5; StdLeadTime = 1;
MeanTime = 5;
ReOrderPoint = MeanLeadTime / MeanTime;
InitialInventoryLevel = ReOrderPoint + Q;
InvPosition = InitialInventoryLevel;
LABELS = {PlaceOrder, ClearItem, Shelve, FreeBackOrder};
CHAPTER 7. COMPLEX SIMULATION MODEL

DEPART { }
end
ELSE LINK { Name = OnHand; }

ClearItem: DEPART { }
FreeBackOrder: DEPART { Quantity = 1; }

END.

This model contains two CHAINS: one for inventoried items and one for backorders. The first ARRIVE block generates entities representing items to be inventoried, and thus the model begins with the maximum amount of inventory, which is equivalent to the arrival of an order and no demand occurring during the lead time. Since the initial situation is to have maximum inventory with no pending backorders, the initial items are immediately placed on the CHAIN (by the first LINK block), representing the “on-hand” inventory. The second ARRIVE block generates entities that represent customers demanding the items in inventory. There are two possibilities when an item is asked for: either it is in stock (i.e., Used[OnHand] > 0) or it must be backordered (i.e., Used[OnHand] = 0). If the item is in stock, the UNLINK block gets the item from OnHand and then sends the inventoried item out of the store (via the DEPART block at the line labelled clearItem). If the inventory position is less than the reorder point, then an order for additional items is placed. The placing of an order is accomplished by creating an entity through the COPY block that can be thought of as being the truck that will deliver the order. This “truck” entity is sent to the statement labelled as placeOrder, at which time the inventory position is updated and a WAIT block is entered that represents the lead time delay. When the truck arrives, the entities are created (again via the COPY block) and immediately sent to the statement labelled Shelf, where the entities are used to fill backorders if there are any or are linked to OnHand if they are to be inventoried. Backorders are filled by unlinking entities from the BackLog chain and sending them out of the store with a DEPART block through the statement labelled FreeBackOrder. The final step in understanding the above model is to observe what happens when demand for an item occurs while nothing is inventoried. The section of code immediately after the ELSE statement represents the occurrence of demand that must be backordered. As before, the arrival of demand affects the inventory position, so it is possible that a stock replenishment order will be placed at this time. Also, the entity representing unsatisfied demand encounters a LINK block and is placed on the CHAIN BackLog, which holds the entity until items arrive at inventory that can satisfy this demand.

- Suggestion: do Exercises 7.7 - 7.9.

7.4 Advanced Language Features

Before discussing additional simulation blocks common to most block-oriented systems, we present some special programming features. These advanced features will be familiar to students who have been exposed to other programming languages. They have been included in MOR/DS because they permit great flexibility and result in more readable models.

7.4.1 Advanced Data Structures

There are some data structures available that should make many programming chores easier. In the following, $Id$ represents the user-assigned name for the structure being defined.

The SEQUENCE

An MOR/DS sequence is a finite collection of real numbers, defined by

\[ Id : SEQUENCE = < NumList >; \]

There are two ways to represent NumList: the regular method and a shorthand version. The regular method is to represent NumList as a list of expressions separated by commas.

Examples of legal SEQUENCE declarations are:

A : SEQUENCE = < 1, 2, 3, 4, 5, 6, 7, 8 > ;
B : SEQUENCE = < 1, 1.25, 1.5, 1.75, 2 > ;
C : SEQUENCE = < 1, 1.3, 1.6, 1.9 > ;
D : SEQUENCE = < 5, Sqr(5), 2/2 > ;

Note that the last example is the sequence <5,25,10>.

The shorthand version is when the elements of the sequence are equidistant from each other, in which case two dots represent the progression. The general shorthand notation takes the form:

\[ Id : SEQUENCE = < Left . Right : Mesh > ; \]
where \( \text{Left} \), \( \text{Right} \), and \( \text{Mesh} \) are expressions denoting the left endpoint, the right end point, and the mesh size, respectively. The \( \text{Mesh} \) is optional, and a default of 1.0 is used if \( \text{Mesh} \) is not specified. The number \( x \) is contained in the sequence \( <a,b:d> \) if, and only if, \( a \leq x \leq b \) and \( x = a + i \cdot d \) for some nonnegative integer \( i \).

For example, the declarations

\[
\begin{align*}
\text{A} & : \text{SEQUENCE} = <1..8>; \\
\text{B} & : \text{SEQUENCE} = <1..2:0.25>; \\
\text{C} & : \text{SEQUENCE} = <1..2:0.3>;
\end{align*}
\]

are equivalent to the same named sequences defined above.

It is also possible to mix these two styles. Thus, we could also write the following equivalent definition for \( \text{A} \):

\[
\text{A: SEQUENCE} = <1..4,5,6..8>;
\]

Sequences are evaluated at compilation time and thus cannot be redefined during execution. If a SEQUENCE is needed that will be dynamic during the simulation, then the INTERVAL data structure must be used.

The INTERVAL

In MORS/DS, an interval can either be discrete or continuous. The discrete interval is equivalent to a sequence except that it is not evaluated until execution. The general syntax for the discrete interval is similar to the sequence:

\[
\text{Id : INTERVAL} = [\text{Left} .. \text{Right} : \text{Mesh}];
\]

For example, the declarations

\[
\begin{align*}
\text{A} & : \text{INTERVAL} = [1 .. 8]; \\
\text{B} & : \text{INTERVAL} = [1 .. 2 : 0.25]; \\
\text{C} & : \text{INTERVAL} = [1 .. 2 : 0.3];
\end{align*}
\]

are equivalent to the same named sequences defined above. A major advantage of the discrete INTERVAL over the SEQUENCE is that the endpoints can be expressions whose values may change as the program is executed.

7.4. ADVANCED LANGUAGE FEATURES

The continuous interval is denoted as a closed interval with syntax

\[
\text{Id} : \text{INTERVAL} = [\text{Left} .. \text{Right}];
\]

Thus, the comma designates a continuous interval and two dots designates the discrete interval. The main use of the continuous interval is in test conditions. For example, the boolean expression

\[
\text{IF} (x >= 5.2) \text{ AND } (x <= 8.3) \text{ THEN} \cdot \cdot\cdot
\]

can be replaced by the shorter expression

\[
\text{IF} \ x \ \text{IN} \ [5.2, 8.3] \ \text{THEN} \cdot \cdot \cdot
\]

Notice that the sequence operator \( \text{IN} \) is utilized in the above statement. The operator has the form \( \text{value IN sequence} \) and returns the value \( \text{True} \) if \( \text{value} \) is contained within the designated sequence.

Real Variables

The syntax to define simple numerical variables is

\[
\text{Id} = \text{Expr};
\]

\( \text{Id} \) is a (possibly subscripted) identifier and \( \text{Expr} \) is any expression. A simple identifier is a string between 1 and 15 characters beginning with a letter. A subscripted identifier is a simple identifier followed by a period followed by a numerical value. Real variable declarations do not use the color reserved word format of other declarations. Examples of some legitimate real variable definitions are

\[
\begin{align*}
x & = 10; \\
y.1 & = x; \\
y.2 & = x + y.1;
\end{align*}
\]

Note that a variable can be used as soon as it is declared but not before. Thus, if the order of the above statements were reversed, an error message would occur during program compilation.
The ARRAY

MOR/DS provides a general type of array structure for manipulating tabular data. These arrays are more general than the arrays provided in most programming languages. The general syntax is

\[ \text{Id : ARRAY[ DimList ] = ArrayCon;} \]

\( \text{DimList} \) is a list of (up to 3) dimension specifications separated by commas and \( \text{ArrayCon} \) is the specification of the contents of the array. \( \text{DimList} \) and \( \text{ArrayCon} \) will be discussed in turn.

Each dimension of an array has a specification of the domain of the index values allowed; a domain must be a sequence. Consider the table \( T \)

\[
\begin{array}{c|ccc}
\text{Id} & 12 & 24 & 26 \\
3 & 1 & 2 & 3 \\
5 & 4 & 5 & 6 \\
\end{array}
\]

The first dimension has domain \(<3,5>\) and the second dimension has domain \(<12, 24, 26>\). This table would be represented as an array defined by \( T : \text{ARRAY[<3,5>,<12, 24, 26>] = ((1, 2, 3), (4, 5, 6))} \);

Then, \( T[3,24] = 2 \) and \( T[5,12] = 4 \).

The \( ArrayCon \) specifies the values for each element of an array. In the above example the right-hand side illustrates how to form an \( ArrayCon \) for a two dimensional array. The elements of an array are stored row by row with the highest (i.e., last) dimension varying the fastest. Each row receives its values as a list enclosed in parentheses. In the above example, the values for row labeled 3 are \((1, 2, 3)\) and for row labeled 5 are \((4, 5, 6)\).

Other legitimate definitions are

\[
\begin{align*}
A : \text{ARRAY[<1,4>] = (1, 2, 3, 4);} \\
B : \text{ARRAY[<1, 2>, <1, 2>, <1, 3>] =} \\
&\{ (111, 112, 113), (121, 122, 123) \}, \\
&\{ (211, 212, 213), (221, 222, 223) \} \\
\end{align*}
\]


A quick way to initialize an array where each element is to be assigned the same value is to use the \texttt{ALL(value)} initializer. Thus, the statement

\[
C : \text{ARRAY[<1,2>,<1..5>] = ALL(10);}
\]

creates the array \( C \), where \( C \) is a \( 2 \times 5 \) dimensioned array with each element equal to 10.

7.4. ADVANCED LANGUAGE FEATURES

7.4.2 Advanced Programming Statements

Statements are either simple or compound statements. Simple statements are statements that contain no other statement. A compound statement consists of a sequence of statements prefixed with the reserved word \texttt{begin} and suffixed by the reserved word \texttt{end}. The compiler treats a compound statement as a single statement. The primary use of compound statements is to enable the execution of more than one statement in commands expecting a single statement (see the \texttt{IF} statement in the previous chapter and the \texttt{WHILE} statement below). They may also be used anywhere a statement is allowed simply for the purposes of enhancing readability. A compound statement need not contain any statements: \texttt{begin end} is a legitimate compound statement.

We first present repetitive statements that are a convenient way to execute statements more than once. MOR/DS supports three of these statements: \texttt{WHILE}, \texttt{REPEAT}, and \texttt{FOR} statements. The first two are called indefinite loop statements because they use a test condition to determine when to terminate the process of repetition. The \texttt{FOR} statement is a definite loop since it repeats a fixed number of times. Therefore, care needs to be exercised to avoid infinite loops when using the \texttt{WHILE} and \texttt{REPEAT} statements.

The WHILE Statement

The syntax for the \texttt{WHILE} statement is

\[
\text{WHILE Boolean Expr DO Stmt;}
\]

\texttt{Boolean Expr} is a boolean expression and \texttt{Stmt} is a single program statement (simple or compound).

Notice that the test condition is performed prior to the execution of the statement; this means that the \texttt{WHILE} statement need not be executed at all. Consider the following example.

\[
i = 1;
\]

\[
\text{WHILE } i <= 5 \text{ DO}
\]

\[
\begin{align*}
&\text{begin}
&\text{PRINTLN}\{ A[i]:8.2\};
&i = i + 1;
&\end{align*}
\]

\[
\text{end:}
\]
Everytime an entity enters the above WHILE block, the value contained in \( A[1], A[2], A[3], A[4], \) and \( A[5] \) will be printed. Of course, for the above lines of code to execute, the variable \( i \) and the array \( A \) would have to be defined in the DEFINITION segment of the program.

The REPEAT Statement

The syntax for the REPEAT statement is:

\[
\text{REPEAT} \\
\text{Stmt1;}
\]

\[
\text{Stmt2;}
\]

\[
\ldots
\]

\[
\text{Stmtn;}
\]

\[
\text{UNTIL BoolExpr;}
\]

\( \text{Stmt1, Stmt2, \ldots, Stmtn are statements and BoolExpr is a boolean expression used to control the loop. Since the test condition is last, the sequence of statements in the construct is executed at least once. The following example of the REPEAT block has the same effect as the previous example of the WHILE.} \]

\[
\begin{align*}
&i = 1; \\
&\text{REPEAT} \\
&\quad \text{PRINTLN \{ A[i:3:2] \}}; \\
&\quad i = i+1; \\
&\text{UNTIL } i > 5;
\end{align*}
\]

The key difference between the WHILE and REPEAT statements is that the REPEAT – UNTIL structure always causes at least one execution since the boolean expression is evaluated after the statements; whereas for the WHILE – DO structure the boolean expression is evaluated first.

The FOR Statement

The syntax for the FOR statement is:

\[
\text{FOR \{AssignStmt, BoolExpr, IncrStmt\} DO EndStmt;}
\]

In the FOR – DO loop, an index variable (either a global or attribute variable defined in the DEFINITION segment) is initialized in AssignStmt, then if BoolExpr is true, DoStmt is executed; then the index variable is incremented according to IncrStmt and then BoolExpr is evaluated again. As long as the BoolExpr remains true, DoStmt is executed with IncrStmt being executed immediately after DoStmt. For example, the following is equivalent to the WHILE and the REPEAT examples above.

\[
\text{FOR (i=1, i<= 5, i=i+1) DO PRINTLN \{i:2\};}
\]

It is also possible to use a compound statement for the DO portion. The use of a compound statement would look something like the following:

\[
\text{FOR (AssignStmt, BoolExpr, IncrStmt) DO} \\
\text{begin} \\
\text{Stmt1;} \\
\text{Stmt2;} \\
\ldots \\
\text{Stmtn;} \\
\text{end;}
\]

The DOCASES Statement

A conditional statement selects for execution a single one of its component statements based upon some type of selection condition. The two conditional statements supported by MOR/DS are the IF and DOCASES statements. The IF statement was discussed previously and the DOCASES is discussed below.

The DOCASES statement is functionally similar to the computed GOTO statement in FORTRAN and BASIC and structurally similar to the CASE statement in Pascal. In FORTRAN and BASIC, an explicit transfer to a labelled statement is made, while in Pascal and MOR/DS the transfer is implicit in order to emphasize the complete structure of the construct. A specific statement from a collection of statements comprising the DOCASES structure is selected and executed. The syntax is
7.4. ADVANCED LANGUAGE FEATURES

DOCASES Expr OF
CASE value1: Stmt1;
CASE value2: Stmt2;
...
CASE valuen: Stmtn;
ELSE ElseStmt;
END;

For each j, valuej is a real number, an interval or a sequence, and Stmtj is a single (simple or compound) statement. The ELSE = ElseStmt clause is optional and is selected if none of the CASE values match the selection value, Expr. Good programming practice dictates using the ELSE clause to capture unexpected selector values (to catch errors in logic) or those values that do not warrant special treatment.

Example:

DOCASES JobType OF
CASE <1,3,5> : PRINTLN {'One, three or five'};
CARE 2:
begin
SEIZE { Name = Drill }; 
WAIT { Time = 5};
RELEASE { Name = Drill }; 
end; "end of case 2"
CASE 4 : PRINTLN {'Finally got a four!'};
ELSE
PRINTLN {'Unexpected JobType: ', JobType:1};
END; "end of doCASES"

7.4.3 Example

The ability to use arrays and general indexing of resources, queues, and other similar structures permits greater flexibility of the MOR/DS language. Indexing is accomplished by writing a label with a period followed by a number. For example, we could define the following three resources:

shop.1 : RESOURCE = { Capacity = 2 }; 
shop.2 : RESOURCE = { Capacity = 1 }; 
shop.3 : RESOURCE = { Capacity = 3 }; 

Specifically, a symbolic name is a string of between 1 and 15 characters beginning with a letter, and a label as used for a resource or queue is simply a symbolic name or a symbolic name followed by a period followed by a numerical value. Thus, if num is a previously defined variable and is equal to 2, then shop.num refers to the above defined RESOURCE with a capacity of 1 (i.e., shop.2).

To illustrate the concept of using arrays and attribute values for referencing, consider a job shop facility with three shops. There are four different kinds of production orders that are received by the shop. The difference in the kinds of orders is the sequence in which the jobs are processed through the three shops. These four different sequences are given in the following table where a zero indicates the end of the sequence.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Shop sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 1 2 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>1 2 3 2 1 0</td>
</tr>
<tr>
<td>3</td>
<td>2 1 2 3 1 0</td>
</tr>
<tr>
<td>4</td>
<td>3 1 2 0 0 0</td>
</tr>
</tbody>
</table>

For example, if a job of the first kind of production order arrives, it must first be processed through Shop 3, then Shop 1, then Shop 2, and lastly through Shop 1 again. The shops have capacities for simultaneously serving 2, 1, and 3 jobs, respectively. The service time for each job in each shop is approximated by a random variable with an Erlang type 4 distribution and with mean times that vary with the shop. These mean times are 5, 3.5, and 6.1 minutes for the three shops, respectively. Jobs arrive at the production facility one at a time with an exponentially distributed interarrival time with a mean of three minutes. The kind of production order associated with each arriving job is random with equal probability that it will be one of the four kinds.

We will keep the job sequence type in an entity attribute named kind and the job's current location within the sequence in an attribute named where. The array defining the sequence of shops associated with the different kinds of production orders will be kept in an array named seq, so that seq[1,1] = 3, seq[1,2] = 1, seq[1,3] = 2, etc., and thus for any entity, seq[kind,where] gives the entity's current location.
The program is as follows.

```plaintext
PROGRAM "a job shop example"
DEFINITION
  ATTRIBUTES = { where = 1, kind = 0 }
  shop.1 : RESOURCE = { Capacity = 2 }
  shop.2 : RESOURCE = { Capacity = 1 }
  shop.3 : RESOURCE = { Capacity = 3 }
  shopTime : ARRAY[ <1..3> ] = (5, 3.5, 6.1);
  seq : ARRAY[ <1..4>, <1..5> ] = ([3, 1, 2, 1, 0],
                  [1, 2, 3, 2, 1],
                  [2, 1, 2, 3, 1],
                  [3, 1, 2, 0, 0]);
  maxSteps = 5;
  LABELS = { thru }
CONTROL
  StopTime = 100;
LOGIC
  ARRIVE { Time = Expd(3) }
  kind = Duniform(1,4);
  WHILE where IN [1..maxSteps] DO
    begin
      IF seq[kind, where] = 0 THEN GOTO thru;
      SEIZE { Name = shop.seq[kind, where], Units = 1 }
      WAIT { Time = Erlang(shopTime[seq[kind, where]], 4) }
      RELEASE { Name = shop.seq[kind, where], Units = 1 }
      where = where + 1;
    end;
  thru: DEPART { };
END.

CAUTION: The BoolExpr in both of the REPEAT and WHILE statements can take on an arbitrary form and may use any combination of global, system, or attribute variables. Exercise care if any statement contained within the construct may delay or hold the active entity. For example,

```plaintext
steps = 1;
WHILE steps <= noSteps DO
  begin
    c = Routing[type, steps];
```
where the keyword Name is required and gives the name of the gate whose status is to be checked. If the keyword IfClosed is not present, then the entity will remain at the gate until it is opened. If the keyword IfClosed is present, then the entity will be sent to the statement identified by label if the gate is closed. The status of a gate may be changed by the SETGATE block, which has the syntax

```
SETGATE { Name = gName, Status = Open/Close/Invert }
```

where both keywords are required. The SETGATE block affects the status of the gate identified by gName according to Status.

An important concept on the usage of gates is how they operate. When the gate is closed, any entity that enters the TESTGATE block (without an IfClosed parameter) is placed on the gate's linked list according to the discipline specified in the GATE declaration in the DEFINITION segment. When the gate's status is changed to Open via a SETGATE, all blocked entities at the gate are immediately considered as to have passed through the gate. Subsequent changes in the GATE status will not rellock these entities.

We shall consider two examples: a simple one lane traffic light and a more complex two lane traffic light. We give the simple example first to ensure an understanding of the basic concept of gates.

### 7.5.1 One Lane Traffic Light

We simulate a one lane road in which traffic can only go in one direction. Cars arrive at the traffic light according to an exponential distribution with an average of 15 seconds between cars. Every 2 minutes the light changes color. We shall also demonstrate another convenient feature of MOR/DS, namely the ALIAS command. The ALIAS command is provided to enhance the readability of programs. The syntax is

```
ALIAS = { nameList }
```

where nameList is a list of elements of the form

```
newName = oldName
```

separated by commas. It is placed in the DEFINITION segment of the program to change the name of any keyword to names that the user would prefer. For example, in the traffic example it is more natural to think of red and green instead of Open and Close; therefore, we would use the ALIAS statement to change those keywords.

The following simulation will generate a histogram of the length of time that cars spend at the light. The first three logic segment blocks represent the cars that test the light, and when it is green they leave. The second three blocks represent the light controller. An arrival every two minutes inverts the light setting from green to red, or red to green.

**PROGRAM** "One lane traffic light example"

**DEFINITION**

```
ALIAS = { green = Open, red = Close };
flowHist : HISTOGRAM = { Cells = 10, MinValue = 0, MaxValue = 5 };
```

**CONTROL**

```
StopTime = 100;
```

**LOGIC**

```
ARRIVE { Time = Expd(15/60) };
TESTGATE { Name = light };
DEPART { };

"generate traffic light controller"
ARRIVE { Time = 2 };
SETGATE { Name = light, Status = Invert };
DEPART { };
```

**END.**

### 7.5.2 Traffic Light with Turn Lane

We now consider a more complex situation. The problem is that a four-way traffic light exists at the intersection of two roads. The roads are assumed to be only one lane in each direction. We will model one direction with a turn lane of very limited capacity. The problem is that if a vehicle cannot get into the turn lane space, it will stop in the main road and block traffic attempting to go straight. The same condition can exist on the main road if vehicles are backed up far enough to cut off the left turn lane entrance. In such a case, those wanting to turn cannot do so until the blockage is released. For simplicity, we are assuming no time delays for turning or responding to a line starting to move ahead.

We model the system by establishing the left turn lane as a resource and the straight ahead lane adjacent to it also as a resource. Then any
vehicle that arrives at the intersection and cannot get into its desired lane (i.e., resource) stops and blocks all future traffic until it can proceed as desired. The traffic lights are modeled as gates. Thus, due to the simplifying assumption of no time delays, this model has resources without the normal wait block between the seizure and release. The blocked vehicles in the main road are also controlled by a gate, but its closing is controlled by blocked vehicles. Every time the light controller changes a light, the blocked vehicle gate is opened to see if any vehicles can now proceed. The first one that cannot proceed then recloses the gate and waits with all the vehicles behind it. The light controller, as seen from our flow direction, merely repeats a sequence of changing the lights and waiting between changes. Once the whole light sequence has been performed, the controller starts over again. The sequence is as follows: both lights are red for 2 minutes, then the left turn lane has a green for 30 seconds, then both lanes are green for 36 seconds, then the left turn lane is red for 90 seconds, and finally both lights return to red and the sequence repeats.

Both cars and trucks use this road, the difference being that trucks take up twice as much room as cars. Ninety-five percent of the vehicles are cars and 5% are trucks. To indicate left turns, an entity attribute variable is defined and set equal to 1 for a left turn (20% of the time) and set equal to 2 for straight ahead traffic (80% of the time). The program is as follows.

```
PROGRAM "Traffic light with turn lane"
DEFINITION
  ALIAS = { green = Open, red = Close }
  light.1 : GATE = { } ; "left turn lane"
  light.2 : GATE = { } ; "straight lane"
  blkGate : GATE = { } ; "those that cannot enter lanes"
  lane.1 : RESOURCE = { Capacity = 3 } ; "lane.1 are spaces in left turn lane"
  lane.2 : RESOURCE = { Capacity = 3 } ; "lane.2 are spaces in straight lane"
  turnsLeft : DISTRIBUTION(Discrete) = ( 0.2, 1 ), 1.0, 2 );
  truck : DISTRIBUTION(Discrete) = 0.95, 1 ), 1.0, 2 );
  flowHist : HISTOGRAM = { Cells = 10, MinValue = 0,
                          MaxValue = 5 } ;
  flowTime : STATISTICS = { Histogram = flowHist } ;
  forever = 0 ; "0 equals false"

ATTRIBUTES = { direction = 0, size = 0 }
LABELS = { entry, bked };
CONTROL
  StopTime = 100;
LOGIC
  ARRIVE { Time = 0, Limit = 1 } ; "light controller"
  REPEAT
    SETGATE { Name = light.1, Status = red } ;
    SETGATE { Name = light.2, Status = red } ;
    SETGATE { Name = blkGate, Status = Open } ;
    WAIT { Time = 2 } ;
    SETGATE { Name = light.1, Status = green } ;
    SETGATE { Name = blkGate, Status = Open } ;
    WAIT { Time = 0.5 } ;
    SETGATE { Name = light.2, Status = green } ;
    SETGATE { Name = blkGate, Status = Open } ;
    WAIT { Time = 0.6 } ;
    SETGATE { Name = light.1, Status = red } ;
    WAIT { Time = 1.5 } ;
UNTIL forever; "Repeat loop is forever since 0 is false"

"vehicles arriving at corner"
  ARRIVE { Time = Expd(15/60) } ;
  direction = turnLeft(Random);
  size = truck(Random);
  STARTSTAT { Name = flowTime } ;
  TESTGATE { Name = blkGate } ;
  TESTGATE { Name = blkGate, EClosed = entry } ;
  "second TESTGATE needed because first one releases everything at once then it opens"
  SEIZE { Name = lane.direction, Units = size,
           Rexist = bked } ;
  TESTGATE { Name = light.direction } ;
  RELEASE { Name = lane.direction, Units = size } ;
  ENDSTAT { Name = flowTime } ;
  DEPART { } ;
```
blkid: SETGATE { Name = blkidGate, Status = Closed }; GOTO entry;
END.

An unusual feature of the above model is seen at the statements starting with the label entry. The difficulty is that entities that have technically passed the GATE blkidGate can still be blocked according to the physical situation. To accomplish the blocking at the GATE, two TESTGATE blocks are sequenced together both referencing blkidGate. Once the GATE is opened, all the blocked cars (entities) are then past the gate; however, if one of these vehicles gets blocked out of the lane, that vehicle will set the GATE to Close, thereby blocking subsequent vehicles. By making these entities test the GATE a second time before proceeding, they can again be blocked by the GATE being closed. The second TESTGATE transfers entities back to the first TESTGATE so that on the next light change, the process will repeat.

- Suggestion: do Exercises 7.10 and 7.11.

### 7.6 Searching Lists for Entities

One unusual feature of MOR/DS is its list searching and entity removal capability. These capabilities are supported by the language constructs: SEARCH and REMOVE. The SEARCH block allows an active entity to search almost any list structure (excluding the future events list) in the language for an entity that matches some criterion. The block returns either an entity pointer, the sequence in the list of the entity, or both. The specified entity can then be removed if desired by the REMOVE block. Knowledge of entity pointers provides a means by which the user can write (or read) directly into an attribute of the selected entity. These statements provide a great deal of power and flexibility into the language.

The entity pointer variable(s) used in these block commands is defined in the DEFINITION segment. The modeler defines and names as many entity variables as are needed in the DEFINITION segment similar to other variable definitions, except that entity pointer variables are not initialized by the modeler. The ENTITY type is used to define these variables. The syntax is:

\[ \text{myEntity : ENTITY} \]

Entity variables are actually pointers that point to where entity attributes are stored in memory. These attributes can be referenced by listing the entity variable name then enclosing the attribute name in square brackets similar to our array notation. Thus, myEntity[EntityNumber] gives the entity number (a system defined attribute) of the entity named myEntity. If each entity has an attribute list such as:

\[ \text{ATTRIBUTES = \{ one, two, three \};} \]

then myEntity[one] is the value of the attribute one associated with the entity myEntity. The value stored in the ENTITY variable myEntity is actually the memory address of the entity found by the SEARCH block. An ENTITY variable can be used to reference any number of entities, one at a time. The entity address, however, can only be set by the SEARCH block.

The SEARCH block has several parameters since the user may specify either an entity variable, or a list sequence number as the output, or both, and a list number range for the search can be defined. That is, the modeler may want to find the first entity in the third to fifth locations in the list that satisfies a given condition such as having a PreemptLevel value of 3. This type of flexibility necessarily requires a considerable amount of data regarding the search conditions. The SEARCH block syntax is:

\[ \text{SEARCH \{ Name = listName, Entity = entityVariable, Rank = globalVar, Begin = expression, End = expression, Condition = BooleanExpr \}} \]

where listName is the name of the list to be searched such as a queue, gate, or chain; entityVariable is an optional entity variable into which the specified entity pointer is to be placed; the NIL function is used to check if no entity satisfying the conditions was found; Rank indicates the global variable into which the index, or location within the list, for the selected entity is stored (optional parameter); Begin and End are (optional) keywords relating expressions to determine the first through last entity locations in the list that are to be searched (the default search is over the total list); and Condition specifies an optional boolean expression usually relating to the searched entity’s attributes that must be true for the entity to be selected. If no entity is found by the SEARCH block, then entityVariable will equal NIL and the Rank variable will be zero. The variable used for storing the entity rank should be a global variable. If this variable is an entity attribute, it will be stored on the entity of the list being searched not on the entity activating the search. Either the Entity or Rank (or possibly both) parameter must be used so that the result of the search will be known.
The SEARCH block is extremely useful when developing routines that operate on linked-lists. The following example program tests the sequencing aspects of a priority-based queueing process. Ten entities arrive and nine are forced onto a queue list ordered by a randomly assigned priority. Just before the initial entity leaves the resource, a listing of the entity sequence on the queue is printed by utilizing the SEARCH block in conjunction with a WHILE loop and a PRINT statement.

```
PROGRAM "Example to Illustrate the SEARCH Block"

DEFINITION

  ATTRIBUTES = { Pri = 0 };
  theQ : QUEUE = { Discipline = Choice(Pri) };
  res : RESOURCE = { Capacity = 1, Queue = theQ };
  cnt = 1; listR = 0;
  ent : ENTITY;

CONTROL

  StopCount = 1; Seed = 1331;

LOGIC

  ARRIVE { Time = 0, Limit = 10 };
  Pri = Duniform(-10,10);
  SEIZE { Name = res, Units = 1 };
  WAIT { Time = 5 };
  cnt = 0;
  WHILE cnt <= Used[theQ] DO
    begin
      SEARCH { Name = theQ, Begin = cnt, End = cnt, 
               Rank = listR, Entity = ent };
      IF NOT NIL(ent) THEN
        PRINT { 'sequence: ', cnt:2, ' rank: ', listR:3, 
                ' Pri: ', ent[Pri]:5:0};
        cnt = cnt + 1;
      end;
      RELEASE { Name = res, Units = 1 };
    DEPART { Quantity = 1 };
  END.
```

These results illustrate both the Rank and Entity options of the SEARCH block. The NIL function was utilized to test if an entity pointer was actually found.

If we wanted the rank of the first entity on the list with a Pri value equal to 6, then we could omit the WHILE statement and just use a SEARCH of the form:

```
  SEARCH { Name = theQ, Rank = listR, Condition = (Pri=6) };
```

This statement would return in listR the value 5, which is the rank of the first entity on the list with attribute Pri equal to 6. Note that Condition is a boolean expression, in this case Pri = 6.

If no entity is found during the search operation, a zero value is returned to the Rank variable and a NIL pointer is returned in the ENTITY variable.

The REMOVE block works in conjunction with the SEARCH block to allow the removal from the linked list of the entity found by the search. The REMOVE block can operate with either the entity’s rank, placed in listR in the example, or the entity pointer variable, ent in this example. The syntax of the REMOVE block is:

```
  REMOVE { Name = listName, Rank = globalVariable, 
            Entity = entityVariable, Destination = label };
```

where listName is the name of the queue, chain, or gate list where the entity resides; Rank points to the entity’s sequential location in the list and is optional (Entity must be provided if Rank is not); Entity points to the entity variable containing the entity to be removed and is optional (Rank must be provided if Entity is not); and Destination indicates the statement label where the removed entity is to be sent (if Destination is not given, then the removed entity simply proceeds to the next block). The entity that activated the REMOVE always proceeds to the next block.

7.6. SEARCHING LISTS FOR ENTITIES

The queue list output from this simulation is:

```
sequence: 1 rank: 1 Pri: -8
sequence: 2 rank: 2 Pri: -3
sequence: 3 rank: 3 Pri: 5
sequence: 4 rank: 4 Pri: 5
sequence: 5 rank: 5 Pri: 6
sequence: 6 rank: 6 Pri: 7
sequence: 7 rank: 7 Pri: 7
sequence: 8 rank: 8 Pri: 9
sequence: 9 rank: 9 Pri: 10
```

The results illustrate both the Rank and Entity options of the SEARCH block. The NIL function was utilized to test if an entity pointer was actually found.

If we wanted the rank of the first entity on the list with a Pri value equal to 6, then we could omit the WHILE statement and just use a SEARCH of the form:

```
  SEARCH { Name = theQ, Rank = listR, Condition = (Pri=6) };
```

This statement would return in listR the value 5, which is the rank of the first entity on the list with attribute Pri equal to 6. Note that Condition is a boolean expression, in this case Pri = 6.

If no entity is found during the search operation, a zero value is returned to the Rank variable and a NIL pointer is returned in the ENTITY variable.

The REMOVE block works in conjunction with the SEARCH block to allow the removal from the linked list of the entity found by the search. The REMOVE block can operate with either the entity’s rank, placed in listR in the example, or the entity pointer variable, ent in this example. The syntax of the REMOVE block is:

```
  REMOVE { Name = listName, Rank = globalVariable, 
            Entity = entityVariable, Destination = label };
```

where listName is the name of the queue, chain, or gate list where the entity resides; Rank points to the entity’s sequential location in the list and is optional (Entity must be provided if Rank is not); Entity points to the entity variable containing the entity to be removed and is optional (Rank must be provided if Entity is not); and Destination indicates the statement label where the removed entity is to be sent (if Destination is not given, then the removed entity simply proceeds to the next block). The entity that activated the REMOVE always proceeds to the next block.
7.6. Searching Lists for Entities

7.6.1 Example: Strict FIFO

The method that MOR/DS uses in an attempt to fill available resource space is activated when a release occurs. The released units are added to the available units, and the resource fill rule is invoked. This process is to select the first entity in the resource queue list and, if it will fit, to transfer it to the appropriate SEIZE block. If the selected entity requires too many resource units, then it is bypassed, and the next entity in the list is considered. This process is repeated until either the resource is filled, or the list has been traversed.

One of the most straightforward filling rules is to follow a strict first-in-first-out (FIFO) rule. Thus, entities that reside in the waiting queue behind entities that cannot fit into the available space are blocked from the resource even if they can fit. We illustrate the difference between the normal FIFO filling rule and the strict FIFO rule with a simple example of a restaurant. Consider a restaurant with a capacity of 20 people and assume it currently has 18 people at the tables. There are two groups of 4 people waiting to be seated (i.e., 8 people in all are waiting). While the groups of four wait, another group of two arrives. The group of two will immediately be seated under the normal FIFO rules since there is room for them. However, under a strict FIFO ordering scheme, they would not be seated.

Most simulation languages utilize the nonstrict FIFO rule as followed by MOR/DS for filling unused resource capacity. Any other rule such as the strict FIFO rule must be handled directly by the modeler. The embedding of Pascal-like statements within MOR/DS provides for a robust programming language that allows for implementing general filling rules. Furthermore, the ENTITY type data structure also provides for increased programming capabilities.

We develop the strict FIFO filling rule below to illustrate the utilization of these entity blocks. (At the end of Chapter 5, we indicated a simpler method that can be used to implement the Strict FIFO rule by creating a second resource that, in effect, acts as a door; however, we feel the program below will be instructive in the use of the SEARCH and REMOVE blocks.) To develop a specialized resource filling rule, we completely control the linking and unlinking of entities that cannot directly obtain the requested resource. Entities requesting the resource first check the chain (waiting line) in front of the resource. If other items are waiting, then the new entities join the back of the chain. We name the resource and chain as res and que, respectively, and place the resource demand units in the entity attribute size. The following program implements the strict FIFO rule.

PROGRAM "Example of resource filling versatility
strict first-in-first-out (FIFO) order"

DEFINITION
res : RESOURCE = { Capacity = 4 };
que : CHAIN = { };
cust : ENTITY;
ATTRIBUTES = { size = 0 };
LABELS = { join, out, full };
cumulative = 0;

CONTROL
StopTime = 200;

LOGIC
ARRIVE { Time = 5 };
size = Duniform(1.4);

join: SEIZE { Name = res, Units = size, Rexcess = full };
WAIT { Time = 12 };
RELEASE { Name = res, Units = size };
"modeler controlled filling rule"
cumulative = 0;

WHILE cumulative <= Available[res] DO
begin
IF Used[que] = 0 THEN
DEPART { }; "que is empty"
SEARCH { Name = que, Begin = 1, End = 1,
Entity = cust };
IF cust.size > (Available[res] - cumulative) THEN
DEPART { }; "strict FIFO rule"
REMOVE { Name = que, Entity = cust,
Destination = join };
cumulative = cumulative + cust.size;
end;
7.7 Creating and Scheduling Entities

The concept is to disengage the standard filling rule by not allowing entities on the resource queue and, therefore, no automatic filling will take place since the resource is not aware of any queued entities. (Of course, if all entities require the same number of units of the resource, i.e., the attribute size is the same for all entities, then the normal FIFO rule is equivalent to the strict FIFO rule, and the above program would not be needed to implement strict FIFO.) If the entity cannot get into the resource, then the Recess destination option of the seize block is used. If the resource places them in its normal queue, then it would retrieve them in the standard fashion. The Recess destination is merely a link to our chain, que, with the label join. This logic accomplishes linking all queue entities in our chain list que. Now, when an entity releases resource units, the resource will not be filled by the normal filling rule since no entities reside in the resource related queue. Therefore, we must initiate our own filing rule utilizing the releasing entity as the active vehicle. After the release statement, we compute the resource space available and unlink all entities at the front of que such that their unit demands, size, sum to less than or equal to the available resource units. When this sum exceeds the available space, the unlinking process stopped. This is accomplished by a set of statements controlled with a WHILE statement. This sequence of statements first checks if the queue is empty and quits if it is. Otherwise, the units demand for the first entity on the chain is obtained by the SEARCH statement. If the units demanded by the first entity, cust[size], are more than the available units, then the process is complete. If the entity will fit, then the entity is removed from the chain and sent to seize the resource. Note, however, that the removed entity was not yet seized the resource units allocated to it, and the availability counts kept by the active entity in the global variable cumulative. The availability is updated, and the process repeated. Note also that the ENTITY variable act was used to read the size request of the inactive entities on the chain.

two steps via the CREATE and SCHEDULE blocks. This set of blocks thus allows the modeler to generate his or her own arrival block type structure. The CREATE establishes an entity data structure and places the entity address into a defined entity variable name so that the entity's attribute values can be set by the entity that activates the CREATE block command. The SCHEDULE block places the entity on the future events list along with any associated delay time and also specifies where the entity is to enter the system once this time has transpired. The concept behind the CREATE block is to allow the user maximum flexibility when it is desired. The ARRIVE block creates an entity and establishes the time at which that entity will become active. The ARRIVE block is the only block that is not activated by an entity since it is the beginning of entities. The CREATE block is activated by an entity, and when it is activated it creates another (passive) entity. The SCHEDULE block is used to determine when the entity created by the CREATE block will become active. The syntax of these blocks is

```
CREATE { Entity = entityVariable; }
SCHEDULE { Entity = entityVariable, Time = expression,
     Destination = blockLabel; }
```

To illustrate the use of this block pair, consider the following example that creates five entities, loads their attributes from a prespecified array, schedules their entry into the system after a short delay, and then prints out the entity attributes to demonstrate the accomplishment of the designed tasks.

```
PROGRAM "Example Model of Create and Schedule"

DEFINITION
ATTRIBUTES = { attr : ARRAY[<1..3>] = ALL(0) ;
   labels = { out, icop } ;
   ent : ENTITY ;
    i = 0 ; count = 0 ; j = 0 ;
attval : ARRAY[<1..5>,<1..3>] = ( (1,2,3),
     (4,5,6),
     (7,8,9),
     (10,11,12),
     (13,14,15) ) ;

CONTROL
StepCount = 5 ; Seed = 1331 ;
```

```
7.8 Exercises

7.1. A manufacturing company is open 24 hours a day, seven days a week. Parts to be processed arrive (deterministically) every 5 minutes during the 8-hour day shift, every 6 minutes during the 8-hour evening shift, and every 12 minutes during the 8-hour night shift. Processing time for the parts is an exponentially distributed random variable with a mean of 30 minutes. The day shift has 6 workers, the evening shift has 5 workers, and the night shift has 3 workers. As soon as a shift is over, the workers leave their job, letting the next shift finish it when they have time. (Processing time of the job is not lost when the work is interrupted.) If a part arrives during the day shift, how long will it spend in the system? What about a part that arrives during the night shift?

7.2. A small airline company has one person that helps arriving customers. Customers arrive according to a Poisson process with a mean rate of 6 arrivals per hour. (Note that a Poisson process implies that customers arrive one-at-a-time with exponentially distributed interarrival times.) Ten percent of the customers are VIPs. Not only are the VIPs always served ahead of ordinary people, but a VIP is able to preempt the non-VIP from service. If a person is preempted, his/her service must start all over again. The length of time a customer spends getting help is distributed according to a continuous uniform distribution that varies between 8 and 11 minutes. Determine the average number of customers per hour that get preempted.

7.3. The fire department of a large city owns and operates a garage which performs maintenance and repair on all vehicles owned by the fire station. The garage has only one service bay and one mechanic which implies that service can be performed on only one vehicle at a time. All vehicles owned by the fire station have predetermined maintenance schedules. Thus at the start of each day (they only work eight hours per day), there are 1 to 3 (discrete uniform) vehicles waiting for maintenance work. The time required to service one vehicle varies between 1 to 3 hours (continuous uniform).

Of course, fire-station vehicles are in use 24 hours each day and breakdowns occur from time to time. Whenever a vehicle breakdown, it is taken immediately to the garage and considered an emergency repair job. (If a breakdown occurs in the evening or night, then the vehicle will be waiting at the start of the following morning when the shop opens.) Thus, if a regularly scheduled maintenance job is in progress, the emergency repair job takes precedence and preempts the maintenance job. Emergency repair jobs occur according to a Poisson process at a mean frequency of one every 48 hours. The service time for emergency jobs is distributed according to the Weibull distribution with a scale parameter of 2.75 hours and a shape parameter of 4.

Develop a simulation model of the city maintenance garage. Estimate the average length of time vehicles are out of service.

7.4. Surgeons arrive to a hospital according to a Poisson process at a rate of 5 per hour. They first check in at the registration desk where they give the receptionist information regarding the patient in which they are interested. The time it takes to give the (single) receptionist all
the information varies according to a continuous uniform distribution between 3 and 5 minutes. The receptionist then sends copies of the information via computer to three departments: X-ray, diagnostic lab, and the records room. Because of the high variability among patients, the time it takes each department to respond is distributed as an exponential random variable with means 10 minutes, 12 minutes, and 8 minutes, respectively. After each department has completed their report, it is sent back to the receptionist. Once the receptionist has collected all the reports, it takes 2 to 3 minutes (continuous uniform) to organize them and get them ready to be picked up. Estimate the mean time it takes to have a report ready for a surgeon and estimate the probability that it will take longer than 25 minutes.

7.5. Individual parts arrive to a workstation every 5 minutes. Twenty-five percent of the parts are Type I and 75% are Type II. The parts are placed in a box (part types are mixed within a box). Type I parts take exactly 2 minutes to be placed in the box and Type II parts take 3 minutes. When the box contains 24 parts, the entire box is moved to another location via a conveyor belt system. The length of this move can be approximated by a normal distribution of mean 2 hours and standard deviation 12 minutes. Once the box arrives, the parts are removed from the box and painted. (Assume that there are no resource limitations for the removal and painting operations; thus, no parts have to wait while other parts get painted.) The removal and painting operations take between 25 and 35 minutes (continuous uniform) for Type I parts and 50 to 55 minutes (continuous uniform) for Type II parts.

As long as a part is being processed within the system, the company loses money. For each hour that a Type I part spends in this system (from the time of initial arrival to the end of the paint job) the company estimates that it loses one dollar, and for each hour that a Type II part spends in this system the company estimates that it loses $1.20. What is the mean and standard deviation for the money lost per part?

7.6. Reconsider the above workstation problem except that the removal and painting of the parts is resource limited. There are ten robots that work in a FIFO manner (one robot per box). Thus a box will be in front of the robot until each part within the box has been removed and painted.

The company is considering changing their procedure slightly in an effort to save money. At the boxing center, it has been suggested that the parts be placed in boxes according to type so each box contains only Type I parts or only Type II parts. Since the robots could then be programmed differently, they will save 5 minutes in the unloading and painting operation. Furthermore, for an equivalent cost of $5.00 per hour, the boxes could be arranged so that the robots give priority (but not preemption) to Type II boxes (or Type I boxes if that is better). What would you suggest for this operation? (Hint: you might let the robot have capacity of 10*24 and then use a GATHER block to insure that all parts release the robot at the same time.)

7.7. Individual parts arrive according to a Poisson process with a mean rate of 10 per hour. The parts must be painted and their bases drilled. There are two painters and one driller to work on the parts. (Two painters cannot work on a single part, but a painter and a driller may work on the same part at the same time.) The painting operation takes an exponentially distributed length of time with a mean of 5 minutes, and the drilling takes between 3 and 4 minutes (continuous uniform). (There is plenty of room, so assume infinite capacity queues.) After being painted and drilled, the part is placed in a warehouse, which takes between 15 and 20 minutes. A truck comes and removes all parts from the warehouse every 24 hours. How big should we plan on making the warehouse? Also, define a global variable that will equal the average number of parts removed per truck. (The warehouse clearing aspect can be modeled using either chains or gates. Do the problem twice so that you will understand both methods.)

7.8. Modify the above problem as follows. Parts come in two sizes: 40% come in the big size and 60% come in the small size. The drilling operation takes the same time for both sizes (as in the previous problem), but the painting operation is exponentially distributed with a mean of 5 minutes for the small size and 8 minutes for the big size. The parts are stored in boxes in the warehouse without mixing sizes. There are 5 big parts to a box or 10 small parts. Although the truck comes once a day, there is a certain amount of randomness to the schedule that can be approximated by the normal distribution. The average time of arrival for the truck is at noon each day with a half-hour standard deviation. (Notice that this is not: equivalent to a normally distributed
7.9. A blood bank receives an average of 100 pints of blood at the start of every day. (The actual number of pints is random and can be described by a Poisson random variable.) Of the blood received, 40% is Type A, 30% is Type O, 20% is Type B, and 10% is Type AB. Requests for blood occur continuously throughout the day according to a non-stationary Poisson process. From 6 A.M. until 11 A.M. the demand averages 10 requests per hour; from 11 A.M. until 5 P.M. the demand averages 5 per hour; from 5 P.M. until 11 P.M. the demand averages 9 per hour; and from 11 P.M. until 6 A.M. the demand averages 2 per hour. The number of pints demanded with each request varies: one pint 10% of the time, two pints 10% of the time, three pints 40% of the time, four pints 30% of the time, and five pints 10% of the time. A request is either for Type A, Type O, Type B, or Type AB with probabilities 0.4, 0.3, 0.2, 0.1, respectively. All pints associated with one request are the same type. At the start of each day (at 6 A.M.), any blood that has been in the bank 21 days or more is discarded. What is the probability that a request will come in that cannot be completely filled? Does that probability depend on blood type?

7.10. Cars on a two lane highway must pass over a one-way bridge. Cars arrive at the bridge at an average rate of one every 30 seconds going east and one every 20 seconds going west. (The arrival process is Poisson.) It takes 20 seconds to cross the bridge. What is the probability an arriving car will have to wait at the bridge for more than one minute before crossing?

7.11. Cars arrive at a ferry according to a Poisson process with a mean rate of 30 per hour at each side of the river. A single ferry with a capacity for 25 cars is used. It takes between 15 and 20 minutes (continuous uniform) for the ferry to cross the river. As soon as the ferry reaches one side, it unloads its cars and then loads those waiting and crosses again. Thus, it crosses the river whether or not it has a full load. What is the average load for the ferry? What is the average length of time that it takes a car to cross the river? First, answer the questions assuming that the loading and unloading of cars takes no time. Second, answer the questions assuming that the loading and unloading takes 15 seconds per car. (A difficulty arises as follows: If 12 cars are to be unloaded and 12 cars are to be loaded, the operation takes 6 minutes, during which time more cars might have arrived and thus would be loaded.)

7.12. Augment the Job Shop Example Problem of Section 7.4.3 to sequence jobs at each shop according to the job classification "kind." That is, at each shop all jobs with kind equal to 1 have (non-preemptive) priority over all other jobs, jobs with kind equal to 2 have priority over all other jobs except for those with kind equal to one, etc. Collect the system flow time by job classification "kind."

7.13. A small rope-making company operates as a job shop with two different types of machines. The first type of machine produces the initial filament. There are four of these machines, each one producing filament in a different color: blue, red, yellow, and white. The second type of machine takes the basic filament and produces the rope. Two of these machines produce small rope and three of the machines produce large rope. Thus, if an order comes in for small red rope, the red filament machine must be used, and then either one of the two small rope machines could be used. (We assume that all rope produced is the same length.) After a statistical analysis of past orders, it has been determined that an order is equally likely to be any of the four colors; however, there is a difference in the proportions of small and large rope based on color. Twenty percent of the blue rope is large, 50% of the red rope is large, 60% of the yellow rope is large, and 70% of the white rope is large.

The company operates eight hours per day, five days a week. The paperwork on orders is processed throughout the day so that at the start of the next day a batch of orders is ready at the shop. The number of orders that are to be made during the day varies according to a Poisson process with an average number of 100 orders present at the start of each day.

Processing times on the machines are all deterministic. It takes 15 minutes per order of small rope on the filament machine and 20 minutes per order of large rope. For the rope machines, it takes 20 minutes per order for small rope on either of the two machines, and 30 minutes per order for large rope on one of the three machines.

The shop will stay open as long as it takes to complete the day's orders. Thus, whenever it takes longer to manufacture all the orders than 8
hours, the company must pay overtime. It estimates that it costs $50 per hour of overtime per machine that must be used. Analyze this shop and determine possible options that management should consider if they want to reduce overtime.

7.14. Reconsider the above rope producing company with the following additional complication. The filament machines are old and are periodically down. The time between failures for each machine can be described by an Erlang distribution with a mean of 100 operating hours and standard deviation of 50 operating hours. It takes a random length of time to fix a down machine where the distribution is Erlang with a mean repair time of one hour and a standard deviation of 15 minutes.

Chapter 8

An Overview of GPSS and SIMAN

There are two popular commercial languages, GPSS and SIMAN, whose simulation language structure is designed along principles similar to MOR/DS. Once the principles governing MOR/DS are understood, learning to use either GPSS or SIMAN should not be difficult. This chapter provides an overview of GPSS and SIMAN so that an interested student will be familiar with their structure when exposed to these languages.

8.1 Overview of GPSS

One of the original block-oriented simulation languages is GPSS (General Purpose Simulation System). This system was devised by Geoffrey Gordon \(^1\) during the early 1960s and evolved into a widely used language supported by IBM. After 1985, microcomputer implementations became available \(^2\).

GPSS has been the standard for discrete simulation languages since its inception, partly because of IBM’s prominent position in the computer manufacturing field, but also because the language is simple and robust. The block-oriented structure pioneered by GPSS greatly influenced the design


\(^2\) We know of two implementations: GPSS/PC by MINUTEMAN Software, Inc., P.O. Box 171, Stow, MA 01775, and GPSS/H by Wolverine Software Corp., 7630 Little River Turnpike, Annandale, VA 22003.
and structure of both MOR/DS and SIMAN, as well as that of other simulation languages.

8.1.1 GPSS Language Structure

GPSS is strictly a block language and does not contain any of the non-block statements found in MOR/DS. For example, GPSS does not have an assignment statement—in instead it provides a collection of blocks, each providing a specific assignment operation.

The general syntax for GPSS blocks is

```
BLOCKNAME Parameter1, Parameter2, ..., ParameterN
```

where BLOCKNAME names and denotes the function of the block. Unlike MOR/DS, GPSS block parameters must be provided in a specific order. Because there are no keywords for the parameters, the position of the parameter in the list determines its meaning. A comment may be added to the right of the last parameter on the block line. Unlike MOR/DS, GPSS does not permit mathematical expressions as block parameters. Expressions must be handled in an indirect manner by labeling them with a VARIABLE definition statement.

Entities in GPSS are called transactions, and as with MOR/DS, a GPSS model is controlled by the flow of these transactions (i.e., entities) through model blocks. A GPSS simulation model begins with a SIMULATE statement and concludes with an END statement. Transactions are created by the GENERATE block and are destroyed by the TERMINATE block (analogous to the ARRIVE and DEPART blocks of MOR/DS). Comment lines are created by placing an asterisk (*) in the first column of a line.

Most GPSS implementations provide a number of predefined constructs such as facilities and gates, in addition to global variables and transaction attributes. These predefined constructs need not be defined by the modeler. The modeler also cannot name attribute variables in GPSS. Attribute variables are called transaction parameters and have the predeclared names P1, P2, and so on. That is, a transaction parameter (attribute variable) is pre-defined and starts with the letter P followed by a numerical suffix. Twelve such variables are automatically provided as the default. If more are needed, there is a way to override this default setting. Similarly, global variables are pre-specified as X1, X2, and so on. User-supplied names are accepted for resources, queues, gates, and block statement labels. Such names consist of between three and five letters and digits, with the first three being letters. The values of all variables and the internal simulation clock time are integers.

GPSS supports two types of resources: facilities and storages. A FACILITY is a preemptable resource with capacity of one. A STORAGE is a non-preemptable resource that can have a capacity of more than one. The name and capacity for a STORAGE are specified by the user with the STORAGE statement. These two resource types have distinct sets of request and return commands. Storage units are requested with an ENTER block and returned with a LEAVE block. A facility is requested with a SEIZE block and released with a RELEASE block. A facility is preempted with a PREEMPT block and released with a RETURN block.

8.1.2 A GPSS Example and Basic Blocks

A simple example will serve to highlight the distinctions and similarities between these two languages. The problem is patterned after a production testing example presented by Thomas Schriber\(^1\).

Consider a production line that manufactures electronic equipment. There are two individuals who test the units after they have been assembled. The time it takes to accomplish this testing operation is random and varies uniformly from one to two hours. Equipment arrives at the testing station randomly, with a mean rate of one per hour (exponentially distributed time between arrivals). Twenty percent of the items are defective and must be sent to a single repairman to be reworked. The repair time is random and varies from three to six hours, uniformly distributed. After the item has been repaired, it is sent back to the assembly line testing station to be rechecked. We simulate five eight-hour shifts starting from an empty system.

Before giving the GPSS example, we present an equivalent MOR/DS model so that you may relate the GPSS model to something familiar. This MOR/DS model was coded in a format to facilitate direct comparison to the GPSS model that follows. You will notice the style is different from that used in our earlier models.

```
PROGRAM "MOR/DS example model"

DEFINITION
    tests : RESOURCE = { Capacity = 2 };
    fixer : RESOURCE = { Capacity = 1 };

CHAPTER 8. AN OVERVIEW OF GPSS AND SIMA

8.1. OVERVIEW OF GPSS

...
8.1. OVERVIEW OF GPSS

The GPSS TRANSFER block provides both random branching and direct branching. By random branching we mean that a specified proportion of the entering transactions are branched to the given block label in accordance with a random number. The TEST block provides conditional branching based on the truth value of a given boolean expression.

GPSS provides only transaction court control of simulation run lengths. The simulation run stops according to a count of transaction terminations. This number is specified as a parameter on the START block. As the simulation progresses, this number is decremented each time a transaction enters a TERMINATE block (the actual decrement size is specified on the TERMINATE block as a parameter, which may be zero). The run terminates when the count reaches or goes below zero. This control mechanism is virtually identical to the MOR/DS method using StopCount. If any other control mechanism is desired (for example, simulated time), the modeler must supply the necessary logic explicitly.

8.1.3 Other GPSS Blocks

It would be helpful at this point to discuss some of the other common blocks necessary for modeling in GPSS. Except for the general category of assignment statements, there is essentially a one-to-one correspondence between GPSS blocks and MOR/DS block statements.

Because GPSS does not provide a general purpose assignment statement, a variety of special purpose blocks are given instead. In contrast, an assignment statement in MOR/DS can be used to modify global variables, attribute variables, and histogram variables. The blocks of GPSS associated with assignments are: MARK, ASSIGN, SAVEVALUE, PRIORITY, and TABULATE.

1. The MARK block places the current clock time into a specified parameter of the transaction. (Example: MARK 1)

2. The ASSIGN block is used to assign numerical values to transaction parameters. (Example: ASSIGN 1, 72)

3. The SAVEVALUE block allows the modeler to assign values to the global variables (X's) of the system. (Example: SAVEVALUE 1, 72)

4. The PRIORITY block allows the modeler to change the transaction attribute PRIORITY. Transactions (entities) are processed FIFO within priority classes.
5. The **TABULATE** block is the method by which values are assigned to tables. (Example: **TABULATE 1, v1**)

A construct similar to an assignment block is the **VARIABLE** definition statement, used for naming expressions. Special names $V1$, $V2$, and so on are set aside for this purpose. For example, to use the expression $P2 + X4$ in a model, we would first define $V1$ as

```
1 VARIABLE P2 + X4
```

to define the expression. We can use this expression later by using the name $V1$.

The **LINK** and **UNLINK** blocks assign and remove transactions from predefined chains. The **LINK** block places the transaction onto the specified chain according to the queueing discipline given in the **LINK** statement.

The **SPLIT** block creates a specified number of duplicates of the entering transaction. These transactions can later be combined back into a single transaction with an **ASSEMBLE** block, or coordinated (but not combined into a single transaction) with the **GATHER** block. The **GATHER** block holds transactions until the specified number have reached the block, then all of the transactions are released to the next block.

The **LOGIC** block is used to set or change the status of a gate. The **GATE** block is used to block transactions if the gate status is closed. (The **SETGATE** and **TESTGATE** blocks in MOR/DS accomplish similar tasks.) An optional parameter allows the transaction to be transferred to an alternative block label if the gate is closed. The default operation is to block the transactions until the gate is opened (the **IfClosed** option is not included). Note that when the gate is opened through the action of a **LOGIC** block, all transactions currently blocked by the gate are freed from the blocking condition.

A run of this model produces the following output:

```
START-TIME EVD-TIME BLOCKS FACILITIES STORAGES FREE-KEMORY
0 2400 16 1 1 370860
```

---

This listing is an edited version of the output produced by GPSS/PC Version 1.1 by MINUTEMAN Software.

### 8.1.4 A Second GPSS Example

Consider a small manufacturing operation that produces and assembles equipment made up of two components. There are two separate shops for building the components and one scheduler/assembler to prepare the initial order and perform the final assembly. Job orders arrive at the plant according to a Poisson process at a rate of 12 per hour (i.e., exponential interarrival times with a mean of five minutes). Upon the order's arrival, the scheduler/assembler prepares the initial paperwork involving two work or-
8.1. OVERVIEW OF GPSS

SIMULATE

* time units in minutes

shop1 STORAGE 2  shop one
shop2 STORAGE 1  shop two
expd FUNCTION RN1,C14  expd. approximation
0.0/0.1.0.14/0.2/0.222/0.3/0.355/0.4/0.509/0.5/0.6/0.6/9.915
0.7/1.2/0.8.1.6/0.84/1.83/0.83/2.12/0.9.2.3/0.95/3.0/1.0,7
flwtime TABLE m1,20,10  10 cells of width 20

* GENERATE 5, FN$exp
  MARK 1  put clock time in P1
  SEIZE sched  seize the scheduler
  ADVANCE 4,1  wait is 4±1 minutes, uniform
  RELEASE sched
  SPLIT 1,two  one copy to "two"

* following is shop one
  QUEUE area1  begin statistics collection
  ENTER shop1, 1
  DEPART area1  end statistics collection

* wait for match to combine assembly
  QUEUE area2
  ASSEMBLE 2
  DEPART area2
  SEIZE sched
  ADVANCE 3, 2
  RELEASE sched
  TABULATE flwtime
  TERMINATE 1

* "the following is for shop two"
  QUEUE area3
  ENTER shop2, 1
  DEPART area3
  ADVANCE 5, FN$exp
  LEAVE shop2, 1
  TRANSFER , return
  START 50
  END

A run of this model produces the following output:

<table>
<thead>
<tr>
<th>START TIME</th>
<th>ENL TIME</th>
<th>BLOCKS</th>
<th>FACILITIES</th>
<th>STORAGES</th>
<th>FREE MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>950</td>
<td>25</td>
<td>1</td>
<td>2</td>
<td>365752</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOC</th>
<th>BLOCK TYPE</th>
<th>ENTRY COUNT</th>
<th>CURRENT COUNT</th>
<th>RETRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GENERATE</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>MARK</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>SEIZE</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ADVANCE</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>RELEASE</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>SPLIT</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>QUEUE</td>
<td>56</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>ENTER</td>
<td>56</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*This listing is an edited version of the output produced by GPSS/PC Version 1.1 by MINI/TEMAN software.*
8.1.5 GPSS Tables

Table 8.1 is designed to give an overview of the basic GPSS language features and a cross reference with MOR/DS⁶. Table 8.2 gives the system variables that can be used in GPSS. Two different naming conventions can be used in GPSS. A facility (or other structure) can be referenced either by a number or by a $ followed by a name. Thus, in Table 8.2, the designation F$ can represent either F1 or F$nam.

---

### Table 8.1: GPSS Blocks

<table>
<thead>
<tr>
<th>GPSS</th>
<th>MOR/DS equivalency and parameter explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADVANCE</td>
<td>WAIT parameter 1: mean time held in block parameter 2: half range for variate or function name (optional)</td>
</tr>
<tr>
<td>ASSEMBLE</td>
<td>COMBINE parameter 1: assembly count</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>parameter 1: assignment statement for transaction parameters, P parameter 2: number of parameter to be modified parameter 3: value to be assigned to parameter parameter 4: number of function modifier (optional)</td>
</tr>
<tr>
<td>DEPART</td>
<td>ENDSTAT parameter 1: name of queue collecting statistics parameter 2: number of units</td>
</tr>
<tr>
<td>ENTER</td>
<td>SEIZE parameter 1: name of STORAGE resource parameter 2: number of units requested</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>DISTRIBUTION or MAPPING label (written to left of block name) function name argument parameter 1: number of data points and type of interpolation parameter 2: data list written on lines after the FUNCTION line</td>
</tr>
</tbody>
</table>

---

⁶We remind the reader again that our purpose is to provide only an abridged summary. Modeling with GPSS requires a user's manual to give the necessary syntax for the specific implementation to be used.
### Table 8.1: continuation

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GATHER</strong></td>
<td>gather count</td>
</tr>
<tr>
<td><strong>GENERATE</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>mean time</td>
</tr>
<tr>
<td>parameter 2</td>
<td>half range for variate or function modifier (optional)</td>
</tr>
<tr>
<td>parameter 3</td>
<td>offset interval (optional)</td>
</tr>
<tr>
<td>parameter 4</td>
<td>limit count (optional)</td>
</tr>
<tr>
<td>parameter 5</td>
<td>priority level (optional)</td>
</tr>
<tr>
<td>parameter 6</td>
<td>number of parameters (optional)</td>
</tr>
<tr>
<td>parameter 7</td>
<td>type of parameters (optional)</td>
</tr>
<tr>
<td><strong>LEAVE</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>name of STORAGE resource</td>
</tr>
<tr>
<td>parameter 2</td>
<td>number of units returned</td>
</tr>
<tr>
<td><strong>LINK</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>name of chain</td>
</tr>
<tr>
<td>parameter 2</td>
<td>merge criteria (FIFO, LIFO, Pj)</td>
</tr>
<tr>
<td>parameter 3</td>
<td>alternate block (optional)</td>
</tr>
<tr>
<td><strong>LOGIC</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>Invert/Set/Reset</td>
</tr>
<tr>
<td>parameter 2</td>
<td>gate name</td>
</tr>
<tr>
<td><strong>MARK</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>assignment for clock time into transaction parameter</td>
</tr>
<tr>
<td>transaction parameter number (optional)</td>
<td></td>
</tr>
<tr>
<td>If no parameter is designated, then the internal MARK parameter is used.</td>
<td></td>
</tr>
<tr>
<td><strong>PREEMPT</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>the FACILITY name</td>
</tr>
<tr>
<td>parameter 2</td>
<td>priority (optional)</td>
</tr>
<tr>
<td>parameter 3</td>
<td>next block for preempted transaction (optional)</td>
</tr>
<tr>
<td>parameter 4</td>
<td>parameter number for remaining time (optional)</td>
</tr>
<tr>
<td>parameter 6</td>
<td>option to restart processing from back of queue (optional)</td>
</tr>
<tr>
<td><strong>PRIORITY</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>new priority value</td>
</tr>
<tr>
<td>parameter 2</td>
<td>option to cause a restart of the event chain</td>
</tr>
</tbody>
</table>

### Table 8.1: continuation

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QTABLE</strong></td>
<td>HISTOGRAM associated with STATISTICS (written to left of block) qtable label</td>
</tr>
<tr>
<td>parameter 1</td>
<td>name of flow time statistics block</td>
</tr>
<tr>
<td>parameter 2</td>
<td>starting point</td>
</tr>
<tr>
<td>parameter 3</td>
<td>width of interval</td>
</tr>
<tr>
<td>parameter 4</td>
<td>number of intervals</td>
</tr>
<tr>
<td><strong>QUEUE</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>name of queue collecting statistics</td>
</tr>
<tr>
<td>parameter 2</td>
<td>number of units</td>
</tr>
<tr>
<td><strong>RELEASE</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>the FACILITY name</td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>the name of preempted FACILITY</td>
</tr>
<tr>
<td><strong>SAVEVALUE</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>assignment statement for global variable, i</td>
</tr>
<tr>
<td>parameter 2</td>
<td>variable number (i)</td>
</tr>
<tr>
<td>parameter 3</td>
<td>value to be assigned</td>
</tr>
<tr>
<td>parameter 3</td>
<td>type (optional)</td>
</tr>
<tr>
<td><strong>SEIZE</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>the FACILITY name</td>
</tr>
<tr>
<td><strong>SPLIT</strong></td>
<td></td>
</tr>
<tr>
<td>parameter 1</td>
<td>number of transactions to be created</td>
</tr>
<tr>
<td>parameter 2</td>
<td>next block for the copies</td>
</tr>
<tr>
<td>parameter 3</td>
<td>serialization parameter (optional)</td>
</tr>
<tr>
<td>parameter 4</td>
<td>number of parameters for each copy (optional)</td>
</tr>
<tr>
<td><strong>STORAGE</strong></td>
<td></td>
</tr>
<tr>
<td>label</td>
<td>RESOURCE definition for a storage</td>
</tr>
<tr>
<td>parameter 1</td>
<td>(written to left of block) storage name</td>
</tr>
<tr>
<td>parameter 2</td>
<td>capacity</td>
</tr>
<tr>
<td><strong>TABLE</strong></td>
<td></td>
</tr>
<tr>
<td>label</td>
<td>HISTOGRAM (written to left of block) table name</td>
</tr>
<tr>
<td>parameter 1</td>
<td>name of variable to be tabulated</td>
</tr>
<tr>
<td>parameter 2</td>
<td>starting point</td>
</tr>
<tr>
<td>parameter 3</td>
<td>width of interval</td>
</tr>
<tr>
<td>parameter 4</td>
<td>number of intervals</td>
</tr>
</tbody>
</table>
3.2. OVERVIEW OF SIMAN

Table 8.2: GPSS system variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation or Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>value of system clock</td>
</tr>
<tr>
<td>CH$</td>
<td>current contents of user chain</td>
</tr>
<tr>
<td>F$</td>
<td>units in facility (0 or 1)</td>
</tr>
<tr>
<td>FN$</td>
<td>user defined function</td>
</tr>
<tr>
<td>LS$</td>
<td>logic switch status</td>
</tr>
<tr>
<td>M1</td>
<td>value of system clock minus mark time</td>
</tr>
<tr>
<td>N$</td>
<td>total transactions into block</td>
</tr>
<tr>
<td>W$</td>
<td>current number transactions waiting at block</td>
</tr>
<tr>
<td>Q$</td>
<td>number transactions in queue</td>
</tr>
<tr>
<td>RN1</td>
<td>random number between 0 and 995</td>
</tr>
<tr>
<td>R$</td>
<td>units of storage available</td>
</tr>
<tr>
<td>S$</td>
<td>units of storage in use</td>
</tr>
<tr>
<td>Xi</td>
<td>value of variable i</td>
</tr>
<tr>
<td>Pi</td>
<td>value of parameter i</td>
</tr>
</tbody>
</table>

8.2 Overview of SIMAN

A popular and relatively new block-oriented language is SIMAN\(^7\) (SIMulation ANalysis). The SIMAN language develops the block-oriented simulation approach initiated by GFSS into a more structured yet flexible implementation, and it allows for the incorporation of continuous simulation. Certain characteristics of manufacturing systems such as conveyors and transports are difficult to model directly in any general purpose simulation language. SIMAN includes a variety of manufacturing-oriented simulation constructs that greatly facilitates conveyor and transport modeling, as well as other similar modeling problems. In addition to special purpose blocks, sub-

routines can be written and compiled in FORTRAN and then linked directly into SIMAN. (Similar subroutine capability is possible in GPSS, depending on the implementation.)

Another capability not found in GPSS but incorporated into SIMAN is a segmented modeling concept. Bernard Zeigler and Tuncer Oren proposed organizing a simulation model into two sections: (1) the model and associated logic program, and (2) the experimental data program that drives the model. A SIMAN program has two such segments, called frames. The experimental frame contains all of the capacity definitions, distributions, queues, and other definition and statistical information. The model frame consists of the model logic, which includes references to the experimental data but not their specific values. Each frame requires its own compilation. Thus, to run a SIMAN program, two separate compilations and a linking step are needed to create a complete model.

SIMAN has extensive analysis tools to supplement the simulation system. These include a post-simulation animation module that performs a graphical playback of the simulation, and a graphical or icon-directed model development system. The playback animation system in particular is an excellent device for analyzing and understanding simulation behavior. Another particularly useful feature of SIMAN is the output processor. The output processor allows data to be graphed, histograms created, and autocorrelations calculated.

8.2.1 SIMAN Language Structure (Experimental Frame)

The SIMAN experimental frame consists of the constructs needed for defining and sizing variables, specifying the run length control and the output information desired, and establishing all distributional data for the model. This experimental frame is somewhat similar to the MOR/DS DEFINITION and CONTROL segments.

The experimental frame is a file that begins with the keyword BEGIN followed by a semicolon and terminates with END followed by a semicolon. The file name must have the (DOS) extension .EXP. (The model frame follows the same convention, except that it uses the extension .MOD for its file name.) All statements in SIMAN end with a semicolon. Anything that follows the semicolon on the same line is treated as a comment statement.

Blank lines are not permitted. The statement types generally included in the experimental frame are:

- PROJECT, model name, author name, month/date/year;
- DISCRETE, max. entities, number attributes, number queues;
- RESOURCES: number, name, capacity <:repeats>;
- REPPLICATE, number runs, start time, stop time;
- PARAMETERS: number, data list <:repeats>;
- DSTAT: number, system variable, title <:repeats>;
- TALLIES: number, title <:repeats>;
- COUNTERS: number, title, <:optional stop count> <:repeats>;

The lists that include number in the above structures are sequentially numbered. For example, if a model has two RESOURCES named tests and fixture, these are defined for the system as

RESOURCES: 1, tests, 2: 2, fixture, 1;

In addition, the modeler must specify the numerical data that is to be collected during the simulation run. This is accomplished with a DSTAT (discrete statistics) statement. The TALLIES and COUNTERS constructs also provide information printed out after the simulation is finished.

8.2.2 SIMAN Language Structure (Model Frame)

Entities in a SIMAN model frame are created by the CREATE block and disposed of by the DISPOSE block modifier (modifiers are discussed in the next paragraph). Entity attributes must be referenced by the (real valued) array variable called A, or by the (integer) variables called M, IS, and NS. The array variables are referenced by A(i), where i is an integer. Similarly, global variables are prespecified as the (real) array X or (integer) variable J. No other global variables are available. User-supplied names are supported for resources, queues, gates, and block statement labels. Such names consist of a maximum of eight characters.

The general syntax for SIMAN blocks is

\[ \text{label BLOCKNAME: Parameter1, \ldots, ParameterN: Modifier;} \]

where BLOCKNAME names and denotes the function of the block. A model frame statement line consists of four distinct fields. Columns 1-8 are
reserved for the label (whether or not one is used). Column 9 is a separator, Columns 10 and following are used for the block name, block parameters, and a block modifier (if present). The last field, following the semicolon, is treated as a comment. Note that SIMAN blocks must begin in Column 10. SIMAN block parameters must be provided in a specified order. That is, the position of the parameter in the list determines its meaning.

A block modifier (which is optional) can be one of four keywords: MARK, NEXT, DISPOSE, or DETACH. The MARK(i) modifier places the current clock time (TNOW) into the ith attribute (namely, A(i)), the NEXT(label) modifier transfers the entity to the block designated by label, and the DISPOSE modifier destroys the active entity after it has completed the block operation. Any one of these three modifiers can be appended to most blocks. The DETACH modifier is used to specify that an entity is to be linked unconditionally to the associated named queue. This modifier, therefore, can only be used with a QUEUE block.

SIMAN has no general programming statements in the language, and specifically, it does not provide direct assignment statements. Instead, SIMAN provides a variety of blocks specifically for assignment functions (ASSIGN, TALLY, COUNT, ALTER).

The resource structure of SIMAN is somewhat similar to MOR/DS, where a RESOURCE is defined in the experimental frame and then accessed by a SEIZE block, delayed by a DELAY block, and released by a RELEASE block. All SEIZE blocks in SIMAN must be preceded by a QUEUE block, which specifies the queueing structure under which blocked entities are handled while waiting for the requested resource to become available. The queue discipline information is given in a RANKING statement within the experimental frame, with FIFO being the default discipline. A key difference between SIMAN and MOR/DS lies in the use of queues. In MOR/DS, queues are associated with resources, while in SIMAN, queues are associated with seize blocks. Thus, each seize block in SIMAN has a distinct queue. When multiple seizes are used to request a resource, the SIMAN resource filling rule polls the queue associated with each seize block requesting that resource.

SIMAN supports a large number of distributed random number generators. User-defined empirical distributions are input usually using one of two different types: DP, and CP. DP is used for discrete probability distributions, and CP is used for continuous probability distributions. The data for these constructs are listed in the PARAMETERS statements of the experimental frame.

3.2. OVERVIEW OF SIMAN

8.2.3 SIMAN Output Processor

A SIMAN simulation run does very little statistical analysis. Instead, the modeler indicates what raw data is to be used for statistical analysis in the experimental frame. Once the run is complete, a host of post-processing capabilities are provided to actually perform the analysis on the collected data. TALLY and DSTAT information are readily output for this analysis. In the experimental frame, an optional output file number can be assigned to each variable, one file per variable. Then, during execution, these files are written by the SIMAN system. After the simulation run or runs have been completed, the output processor can be invoked as a separate processor. This system is interactive and provides commands to perform various analyses. The analyses include variable plotting, histogram development, and correlation analyses.

8.2.4 A SIMAN Example

We use the example from Section 8.1.2 as the first SIMAN example. Again, we remind the reader that our intent is to provide an overview and conceptual understanding of the language. A user's manual should be consulted for detailed syntax.

CONTENT OF THE EXPERIMENTAL FRAME:

BEGIN;
  PROJECT, inspect and repair, cdf, 7/14/1988;
  DISCRETE,100,1,2;
  RESOURCES:1,tests,2,2,fixer,1;
  TALLIES:1,flownine;
  REPLICATE,1,0,24000;
  DSTAT:1,NQ(1),queue tests :2,NQ(2),queue fixer:3,NR(1),tests util. :4,NR(2),fixer util. ;
  PARAMETERS:1,60:2,60,120:3,180,360;
END;
The output to be produced by this model is defined in the DSTAT statement. For this example, the requested output is the time-averaged number of transactions in the resources and the queues. These are denoted by the system variables NR and NQ for resources and queues, respectively. The PARAMETERS statement is used to define parameter sets that will be used in the model frame. Here, three data lists are defined: parameter set 1 contains a single element (60) which is used as the mean of an exponentially distributed random variable, and parameter sets 2 and 3 containing two numbers giving the endpoints for uniform distributions.

CONTENTS OF THE MODEL FRAME:

BEGIN;
CREATE:EX(1,1):MARK(1);
  bac QUEUE,1;
  SEIZE:tests;
  DELAY:UN(2,1);
  RELEASE:tests;
  BRANCH,1:
    WITH, 0.20,bad:
    ELSE,thru;
  ;
  bad QUEUE,2;
  SEIZE:fixer;
  DELAY:UN(3,1);
  RELEASE:fixer:NEXT(bac);
  ;
  thru TALLY:1,INT(1):DISPOSE;
  ;
END;

This model frame should be fairly easy to read, assuming you have a background in MOR/DS. The first statement generates entities with inter-arrival times being exponentially distributed and their time of arrival being "marked" on the first attribute via the MARK modifier. The mean of the exponential distribution used in the first statement is given by the first parameter set defined by the PARAMETER statement. The second "1" within the parentheses of EX indicates that the random numbers used to generate the exponential random variate come from the first random number stream. (A random number stream refers to the "stream" produced by the internal generator from a specific seed.)

All constructs that can cause entities to be blocked (i.e., to form a queue) must be preceded by a QUEUE statement. This informs the SIMAN system of the list in which the entities are to be stored and where to obtain the next entity when it may be taken from the queue (such as when a resource unit is freed). Unconditional branching, such as the MOR/DS GOTO, can be accomplished in SIMAN by merely designating the label of the next command with the NEXT modifier. Entities are destroyed with the modifier DISPOSE.

A unique feature of the SIMAN modeling language is the BRANCH statement, which is a mixture of entity duplicating and logical branching. In this example, we use the branching capabilities only. The BRANCH block creates a duplicate of the active entity for each logical test of the statement. If the test is true, a duplicate entity is sent to the designated label; otherwise, that copy is destroyed. After all copies have been handled, the original entity is also destroyed. Thus, no entity passes through the BRANCH to the next statement. Entities leave only by taking one of the branches. The first parameter of the BRANCH block specifies the maximum number of entity copies that can be sent out of the branch. For example, in a BRANCH statement that has five branches and a parameter value of 3, if the first three tests (branches) were true, no more tests would be made. The BRANCH in this example sends a copy to the statement with the label bad twenty percent of the time and a copy to the statement with the label thru eighty percent of the time.

A run of this model produces the following output:\[10]\:

Beginning execution of run number 1

\[10]\text{This listing is an edited version of the output produced by SIMAN Version 3.5, by Systems Modeling Corporation.}
8.2. OVERVIEW OF SIMAN

CONTENTS OF THE EXPERIMENTAL FRAME:

BEGIN;

PROJECT, two shop and combine, cdf, 7/14/1988;
DISCRETE,1000,3,6;
RESOURCES:1,shop1,2,2,shop2,1,3,schd,1;
TALLIES:1,flowtime;
REPLICATE,1,1;
DSTAT:1,NQ(1),queue schd :2,NQ(2),queue shop1:
  3,NQ(3),queue shop2 :4,NQ(4),fin. shop1:
  5,NQ(5),fin. shop2 :6,NQ(6),q. combine:
  7,NR(1),shop1 util. :8,NR(2),shop2 util.;
PARAMETERS:1,5;2,3,5;3,8,4,1,5;5,6;
COUNTERS:1,number completed,50;
END;

CONTENTS OF THE MODEL FRAME:

BEGIN;

CREATE:EX(1,1):MARK(1);
ASSIGN:X(2) = X(2) + 1;
ASSIGN:A(2) = X(2);
QUEUE,1;
SEIZE:schd;
DELAY:UN(2,1);
RELEASE:schd;
BRANCH,2:
  ALWAYS, two;
  ALWAYS, one;
  shop one
one QUEUE,2;
SEIZE:shop1;
DELAY:EX(3,1);

The second SIMAN example is from the model described in Section 8.1.4. The experiment and model frames for this simulation are as follows:
8.2. OVERVIEW OF SIMAN

Run ended at time  1.208E-05

Tally Variables

<table>
<thead>
<tr>
<th>Number</th>
<th>Identifier</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
<th>Numb of 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FLOWTIME</td>
<td>638.5693c</td>
<td>376.58020</td>
<td>13.11863</td>
<td>980.10910</td>
<td></td>
</tr>
</tbody>
</table>

Discrete Change Variables

<table>
<thead>
<tr>
<th>Number</th>
<th>Identifier</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
<th>Time Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>QUEUE SCHD</td>
<td>6.53</td>
<td>7.20</td>
<td>.00</td>
<td>24.00</td>
<td>1207.92</td>
</tr>
<tr>
<td>2</td>
<td>QUEUE SHOP1</td>
<td>.15</td>
<td>.48</td>
<td>.00</td>
<td>4.00</td>
<td>1207.92</td>
</tr>
<tr>
<td>3</td>
<td>QUEUE SHOP2</td>
<td>47.55</td>
<td>25.92</td>
<td>.00</td>
<td>33.00</td>
<td>1207.92</td>
</tr>
<tr>
<td>4</td>
<td>FIN. SHOP1</td>
<td>47.24</td>
<td>26.05</td>
<td>.00</td>
<td>34.00</td>
<td>1207.92</td>
</tr>
<tr>
<td>5</td>
<td>FIN. SHOP2</td>
<td>.01</td>
<td>.08</td>
<td>.00</td>
<td>1.00</td>
<td>1207.92</td>
</tr>
<tr>
<td>6</td>
<td>Q. COMbine</td>
<td>68.84</td>
<td>46.52</td>
<td>.00</td>
<td>137.00</td>
<td>1207.92</td>
</tr>
<tr>
<td>7</td>
<td>SHOP1 UTIL.</td>
<td>1.15</td>
<td>.75</td>
<td>.00</td>
<td>2.00</td>
<td>1207.92</td>
</tr>
<tr>
<td>8</td>
<td>SHOP2 UTIL.</td>
<td>.98</td>
<td>.13</td>
<td>.00</td>
<td>1.00</td>
<td>1207.92</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Number</th>
<th>Identifier</th>
<th>Count</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NUMBER COMPLETED</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

Run time :  5 Second(s)

Stop - Program terminated.

8.2.5 SIMAN Tables

Table 8.3 is designed to give an overview of the basic SIMAN language features and a cross reference with MOR/DS12. Table 8.4 gives the system variables that can be used in SIMAN.

11This listing is an edited version of the output produced by SIMAN Version 3.5, by Systems Modeling Corporation.

12We remind the reader again that our purpose is only to provide an overview. SIMAN models can be built most effectively with the aid of a user’s manual.
Table 8.3: SIMAN blocks

<table>
<thead>
<tr>
<th>SIMAN</th>
<th>MOR/DS Equivalency and Parameter Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>assignment statement using Capacity[ ]</td>
</tr>
<tr>
<td>parameter 1</td>
<td>resource name</td>
</tr>
<tr>
<td>parameter 2</td>
<td>change in capacity</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>assignment statement</td>
</tr>
<tr>
<td>parameter 1</td>
<td>A(i) or X(i)</td>
</tr>
<tr>
<td>BRANCH</td>
<td>COPY and IF</td>
</tr>
<tr>
<td>parameter 1</td>
<td>maximum number of branches taken</td>
</tr>
<tr>
<td>option 1</td>
<td>IF, boolean expr., label</td>
</tr>
<tr>
<td>option 2</td>
<td>WITH, fraction, label</td>
</tr>
<tr>
<td>option 3</td>
<td>ELSE, label</td>
</tr>
<tr>
<td>option 4</td>
<td>ALWAYS, label</td>
</tr>
<tr>
<td>COMBINE</td>
<td>COMBINE</td>
</tr>
<tr>
<td>parameter 1</td>
<td>number of entities to combine</td>
</tr>
<tr>
<td>parameter 2</td>
<td>indicates attributes for surrogate</td>
</tr>
<tr>
<td>COUNT</td>
<td>assignment statement</td>
</tr>
<tr>
<td>parameter 1</td>
<td>counter number</td>
</tr>
<tr>
<td>parameter 2</td>
<td>increment for counter</td>
</tr>
<tr>
<td>CREATE</td>
<td>ARRI VE</td>
</tr>
<tr>
<td>parameter 1</td>
<td>group size</td>
</tr>
<tr>
<td>parameter 2</td>
<td>interarrival time</td>
</tr>
<tr>
<td>DELAY</td>
<td>WAIT</td>
</tr>
<tr>
<td>parameter 1</td>
<td>time to wait</td>
</tr>
<tr>
<td>GROUP</td>
<td>BLOCK</td>
</tr>
<tr>
<td>parameter 1</td>
<td>number of entities to group</td>
</tr>
<tr>
<td>parameter 2</td>
<td>indicates attributes for surrogate</td>
</tr>
<tr>
<td>PREEMPT</td>
<td>PREEMPT</td>
</tr>
<tr>
<td>parameter 1</td>
<td>priority</td>
</tr>
<tr>
<td>parameter 2</td>
<td>resource</td>
</tr>
<tr>
<td>parameter 3</td>
<td>number of units requested</td>
</tr>
<tr>
<td>parameter 4</td>
<td>label to send preempted entity</td>
</tr>
<tr>
<td>QUEUE</td>
<td>to place on a queue or chain</td>
</tr>
<tr>
<td>parameter 1</td>
<td>queue number</td>
</tr>
<tr>
<td>parameter 2</td>
<td>capacity limit</td>
</tr>
<tr>
<td>parameter 3</td>
<td>label to send entity if full</td>
</tr>
</tbody>
</table>

Table 8.3: continuation

<table>
<thead>
<tr>
<th>RELEASE</th>
<th>RELEASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter 1</td>
<td>resource name</td>
</tr>
<tr>
<td>parameter 2</td>
<td>units</td>
</tr>
<tr>
<td>SEIZE</td>
<td>SEIZE</td>
</tr>
<tr>
<td>parameter 1</td>
<td>resource name</td>
</tr>
<tr>
<td>parameter 2</td>
<td>number of units requested</td>
</tr>
<tr>
<td>SIGNAL</td>
<td>SETGATE</td>
</tr>
<tr>
<td>parameter 1</td>
<td>signal number</td>
</tr>
<tr>
<td>SPLIT</td>
<td>UNBUNDLE</td>
</tr>
<tr>
<td>TALLY</td>
<td>ENDSTAT</td>
</tr>
<tr>
<td>parameter 1</td>
<td>tally number</td>
</tr>
<tr>
<td>parameter 2</td>
<td>variable to be tallied</td>
</tr>
<tr>
<td>WAIT</td>
<td>TESTGATE</td>
</tr>
<tr>
<td>parameter 1</td>
<td>signal number</td>
</tr>
</tbody>
</table>

Table 8.4: SIMAN system variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation or Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNOW</td>
<td>current simulation time</td>
</tr>
<tr>
<td>EX( , )</td>
<td>exponential random variate</td>
</tr>
<tr>
<td>UN( , )</td>
<td>uniform random variate</td>
</tr>
<tr>
<td>RA( )</td>
<td>uniform 0–1 random number</td>
</tr>
<tr>
<td>TR( , )</td>
<td>triangular random variate</td>
</tr>
<tr>
<td>RN( , )</td>
<td>normal random variate</td>
</tr>
<tr>
<td>DP( , )</td>
<td>discrete probability distribution</td>
</tr>
<tr>
<td>X(i)</td>
<td>global system variable i</td>
</tr>
<tr>
<td>A(i)</td>
<td>entity attribute variable i</td>
</tr>
<tr>
<td>INT(i)</td>
<td>equals TNOW - A(i)</td>
</tr>
<tr>
<td>BET(i)</td>
<td>equals TNOW - X(i)</td>
</tr>
<tr>
<td>NQ(i)</td>
<td>number entities in queue i</td>
</tr>
<tr>
<td>NR(i)</td>
<td>number entities in resource i</td>
</tr>
<tr>
<td>NC(i)</td>
<td>current count of counter i</td>
</tr>
<tr>
<td>MR(i)</td>
<td>capacity of resource i</td>
</tr>
</tbody>
</table>
8.3 Exercises

8.1. Develop a GPSS model for Exercise 5.4.

8.2. Develop a GPS3 model for Exercise 5.8. (Note that if you plan to use Student Version 1.1 of GPSS/PC, this problem may result in a program too large for the computer.)

8.3. Develop a SIMAN model for Exercise 5.4.

8.4. Develop a SIMAN model for Exercise 5.8.

8.4 Language Cross References

Tables 8.5 – 8.8 are cross references between the constructs of the three languages: MOR/DS, GPSS, and SIMAN. Control statement information is listed in Table 8.5. Statements related to defining constructs, capacities, distributions, and so on are given in Table 8.6. The simulation and logic statements that are used in the model logic development are found in Tables 8.7 and 8.8.

We stress that this chapter is by no means sufficient to learn the intricacies of GPSS and SIMAN. The chapter's purpose is to introduce these languages, taking advantage of their similarities with MOR/DS to facilitate the familiarization process. One should recognize at this point that there is a great deal of commonality among the discrete event-oriented languages. With a clear understanding of any one of the languages, movement to another language is not difficult. The construct terms and syntax must be learned, but the modeling concepts and approaches are fairly standard.

<table>
<thead>
<tr>
<th>Table 8.5: Statements used to control simulations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Statement Comparisons</td>
</tr>
<tr>
<td>MOR/DS       GPSS       SIMAN</td>
</tr>
<tr>
<td>StopTime     Replicate option</td>
</tr>
<tr>
<td>PrintTime    Replicate option</td>
</tr>
<tr>
<td>ResetTime    Replicate option</td>
</tr>
<tr>
<td>StopCount    Start option</td>
</tr>
<tr>
<td>PrintCount   Start option</td>
</tr>
<tr>
<td>ResetCount   Reset</td>
</tr>
<tr>
<td>Listing</td>
</tr>
<tr>
<td>BlockListing always</td>
</tr>
<tr>
<td>StatListing  always</td>
</tr>
<tr>
<td>VarListing   always</td>
</tr>
<tr>
<td>WatchList</td>
</tr>
<tr>
<td>WatchStatus  DSTAT</td>
</tr>
<tr>
<td>Seed         Seeds</td>
</tr>
<tr>
<td>Randomize</td>
</tr>
<tr>
<td>EXITLOGIC    Start option</td>
</tr>
<tr>
<td>Reset        Reset</td>
</tr>
<tr>
<td>Continue     Standard</td>
</tr>
<tr>
<td>Clear        Clear</td>
</tr>
</tbody>
</table>
### Table 8.6: Statements used for definitions

<table>
<thead>
<tr>
<th>Definition Statement Comparisons</th>
<th>MOR/DS</th>
<th>GPSS</th>
<th>SIMAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE</td>
<td>STORAGE/ FACILITY</td>
<td>RESOURCE</td>
<td></td>
</tr>
<tr>
<td>QUEUE</td>
<td>RANKINGS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAIN</td>
<td>CHAIN</td>
<td>RANKINGS</td>
<td></td>
</tr>
<tr>
<td>HISTOGRAM</td>
<td>TABLE/ QTABLE</td>
<td>TALLIES</td>
<td></td>
</tr>
<tr>
<td>DISTRIBUTION/ MAPPING</td>
<td>FUNCTION</td>
<td>TABLES</td>
<td></td>
</tr>
<tr>
<td>GATE</td>
<td>predefined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTES user named</td>
<td>PARAMETERS</td>
<td>A only</td>
<td></td>
</tr>
<tr>
<td>Variables user named</td>
<td>Variables</td>
<td>X array</td>
<td></td>
</tr>
<tr>
<td>LABELS</td>
<td>OUTPUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEXTFILE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entities</td>
<td>Transactions</td>
<td>Entities</td>
<td></td>
</tr>
<tr>
<td>ALIAS</td>
<td>Synonyms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATISTICS predefined</td>
<td>TALLIES</td>
<td>CONVEYORS</td>
<td></td>
</tr>
</tbody>
</table>

### Table 8.7: Simulation statement comparison

<table>
<thead>
<tr>
<th>Simulation Statements</th>
<th>MOR/DS</th>
<th>GPSS</th>
<th>SIMAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRIVE</td>
<td>GENERATE</td>
<td>CREATE</td>
<td></td>
</tr>
<tr>
<td>DEPART</td>
<td>TERMINATE</td>
<td>:DISPOSE</td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td>ADVANCE</td>
<td>DELAY</td>
<td></td>
</tr>
<tr>
<td>SEIZE</td>
<td>SEIZE/</td>
<td>SEIZE</td>
<td></td>
</tr>
<tr>
<td>RELEASE</td>
<td>RELEASE/</td>
<td>RELEASE</td>
<td></td>
</tr>
<tr>
<td>PREEMPT</td>
<td>PREEMPT</td>
<td>PREEMPT</td>
<td></td>
</tr>
<tr>
<td>RELEASE</td>
<td>RETURN</td>
<td>RELEASE</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>default</td>
<td>QUEUE</td>
<td></td>
</tr>
<tr>
<td>STARTSTAT</td>
<td>QUEUE</td>
<td>:MARK</td>
<td></td>
</tr>
<tr>
<td>ENDSSTAT</td>
<td>DEPART</td>
<td>TALLY</td>
<td></td>
</tr>
<tr>
<td>LINK</td>
<td>LINK</td>
<td>QUEUE:DETACH</td>
<td></td>
</tr>
<tr>
<td>UNLINK</td>
<td>UNLINK</td>
<td>REMOVE</td>
<td></td>
</tr>
<tr>
<td>COMBINE</td>
<td>ASSEMBLE</td>
<td>COMBINE</td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>SPLIT</td>
<td>BRANCH</td>
<td></td>
</tr>
<tr>
<td>GATHER</td>
<td>GATHER</td>
<td>MATCH</td>
<td></td>
</tr>
<tr>
<td>EUNBUNDLE</td>
<td>GROUP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETGATE</td>
<td>LOGIC</td>
<td>SIGNAL</td>
<td></td>
</tr>
<tr>
<td>TESTGATE</td>
<td>GATE</td>
<td>WAIT</td>
<td></td>
</tr>
</tbody>
</table>
# Appendix A

## Description of the Editor

The editor included with MOR/DS is part of a software system (of which MOR/DS is one piece) designed for use in Operations Research/Management Science courses. Since the system is self-contained, once you are in the editor you probably will not need to return to the operating system until all of your operations research work is finished. The "Getting Started" chapter describes the procedure for getting into the editor and provides a brief overview of it. We remind you that pressing the F1 key activates a help facility for the system being used (here discrete simulation). Pressing <ctrl>F1 activates a help system for the editor commands. The table at the end of that chapter (repeated on page 233) is a summary of the control keys necessary for writing the programs. In this appendix, we give descriptions of the various function keys and menu options.

## A.1 Menus

The menu system is activated by pressing the F10 function key. As stated in the "Getting Started" chapter, the menus can be selected either by using the cursor control keys or by pressing the key corresponding to the capitalized letter in the header or menu.

### A.1.1 File Menu

**Open** When Open is activated (F3 key), the system will request the name of a file to load into memory. If the name exists, the contents of that file will be loaded into the editor and the file name will be displayed.
on line 2 of the screen. If the file does not exist, permission is sought to continue. If y is selected, the file name is displayed (line 2 of the screen) and an empty screen awaits input. This editor file now has been defined, but it has no contents until the file is saved. An y response leaves the screen blank, as well as the file name on line 2 of the screen. Thus, you will be in the editor, but no file name is associated with the current editor contents.

Close When close is activated, the current contents of the editor will be saved. If no name is associated with the contents, a prompt for the file name will be issued. After the contents are saved, a prompt is issued for the name of a file to be loaded.

Save When save is activated (F2 key), the current contents of the editor will be saved. If the file being edited has a name, the current contents are written to that file. Otherwise, a file name prompt is issued and then the file is saved. In this latter case, if a file name is given that corresponds to an existing file, a prompt is issued to request permission to overwrite this file.

SaveAs When saveas is activated, the current contents can be saved to a file other than the one from which it was originally loaded. This command is useful when you want to create a new file that has the current editor contents but you do not want to alter the existing file. For example, suppose the file being edited is C:\MOD.EX and you want to make a few changes and save the result as A:\MOD2.EX. Execute F10-F-A and respond to the prompt with A:\MOD2.EX. MOD2.EX will be created on drive A in the master directory, and the current editor file will now be called A:\MOD2.EX. The C:\MOD.EX file is left undisturbed.

tename When the rename command is activated, a prompt will be given asking for the old file name. Then a second prompt will be given asking for the new name. This command will work even if the active editor file is the one to be renamed—but this is an undesirable practice.

Dir When the dir command is activated, a list of files in a given directory will be displayed. Before the files are displayed, a prompt requesting the desired "mask" appears. If the return key is pressed in response to the mask prompt, all the files in the File Path directory (see the Options menu) are listed. If a valid path name is given, that directory is listed. For example, to find and list all files in the C:\MicroOR directory having an extension .DSE, use the mask C:\MICROOR\*.DSE.

Dos The dos command makes it possible to execute DOS commands without leaving the editor. Two formats are provided for executing the external commands: the & prefix form and the non-prefix form. MS-DOS commands (for example, PRINT and TYPE) must use the & prefix. For example, to display the contents of C:\LP.LPE type

  &TYPE C:\LP.LPE

When the & prefix is used, MOR loads COMMAND.COM and lets it execute the command. If the non-prefix form is used, the complete path name and extension of the file must be given (but COMMAND.COM is not loaded, thereby saving memory and time). Any command can be executed in this manner, provided sufficient memory above the editor is available. NOTE: Do not execute the DOS call to Print from the editor. This call reserves memory for DOS to use while processing this command, and it will reserve a portion of memory above the editor. Consequently, memory will be fragmented after exiting the editor.

Quit The quit command (F9 key) exits the editor. If the active file has not been saved or if changes have occurred since the last save, a prompt to save the file is issued. A y response saves the file, an n response quits without saving the file, and ESC aborts the command and returns to the editor.

A.1.2 System Menu

The System menu is used to switch systems. A pointer is placed to the left of the system selected and remains there until changed via the system menu. A system is selected either by pressing the key corresponding to the capitalized letter or by scrolling the highlight bar until it is on the desired system and then pressing the return key. Pressing ESC exits the System menu, and pressing ESC a second time exits the full menu system, returning to the MOR editor.

A.1.3 Run Menu

When the Run menu is activated (F4 key), the currently selected system will operate on the contents of the editor.
A.1.4 Options Menu

The options available within this menu vary, depending on the particular system selected. MOR/LP has a Tableaux option, MOR/NL has an Iterations option, and MOR/DP has a Tables option. MOR/DS has no special options. All systems have the orPath, filePath, Output, and Show options.

OrPath The orPath option allows the user to select the path name for the active system. The drive and directory for the location of the SLAVE.EXE (the executable code) and the MORXX.LLP (help) files are set for the active system. This command would be used if the desired system has a path different from that of the editor.

FilePath The filePath selects the default path for all standard input and output. Thus, the drive and directory established in the filePath is the default for load and save commands as well as for output when a file is used as established under the next option.

Output The Output option allows the output device to be selected from the model. When used, a prompt is displayed that allows the user to specify a file, the printer, or the screen.

Show The Show option gives a summary of the memory usage.

Tableaux When the Tableaux option is activated, an additional menu is displayed that gives the following choices: None (no simplex tableaux given during output), All (all simplex tableaux given during output), and F & L (only the first and last tableaux are output).

Tables When the Tables option is activated, an additional menu is displayed that gives the following choices: All (all recursion tables are output), Optimal (only the values of the optimal return function and the optimal decisions are output in table form), and None (no tables are output).

Iterations When the Iterations option is activated, an additional menu is displayed that gives the following choices: N (no intermediate, detailed information is output) and Y (intermediate, detailed information regarding algorithm execution is output).

A.2 Function Keys

The function keys have been defined to duplicate the most commonly used commands; thus, they provide a quick alternative to the menu system.

F1 Displays help information for the active system, and <ctrl>F1 displays help information for the editor.

F2 Saves the active file. If the file already exists, the current contents of the editor will be written over the file. If no file name has yet been given to the current contents, the system will ask for a name. This command cannot change the name of the file once it has been named. (See the File menu for renaming.)

F3 Loads (opens) a file. The name of the file to be loaded is requested. If a path name is not part of the file name, the editor will look on the FilePath directory (see the Options menu). If there is a file already in the editor, the system will ask for permission to load the new file.

F4 Runs the current system. The current contents are executed according to the system chosen from the System menu.

F5 Activates the algorithm selection window. This command works only if the active system is either MOR/LP or MOR/NL, in which case a list of the appropriate algorithms is displayed. The active algorithm is designated by a pointer to the left of the selection. An algorithm is selected either by pressing the key corresponding to the capitalized letter or by scrolling the highlight bar until it is on the desired algorithm and then pressing the return key. Pressing ESC exits the menu, leaving the current choice active. Selecting the algorithm or pressing the right or left scroll keys also closes the window.

F6 Sets find and replace controls. Choices are U — ignore case, G — global (start at front of file), and R — repeat. Any combination is permitted: UGR repeats search, starting at the top, and ignores case; UR repeats and ignores case but starts from the current cursor location.

F7 Marks the beginning of a block. The character: under the cursor will become the first character in the block. (Same as <alt>B.)

F8 Marks the end of a block. The character under the cursor will become the last character in the block. (Same as <alt>E.)
F9 Exits and returns to DOS system control. If the active file has not been saved or if changes have occurred since the last save, a prompt to save the file is issued. Answering Y saves the file, N does not save the file, and ESC cancels the F9 command.

F10 Activates the pulldown menu bar. The menu heading will appear on the second line of the screen display with the File heading highlighted. ESC deactivates the menu system and restores line 2 of the screen.

<table>
<thead>
<tr>
<th>CURSOR MOVEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>→↑↓←</td>
</tr>
<tr>
<td>&lt;ctrl&gt;→</td>
</tr>
<tr>
<td>&lt;ctrl&gt;↔</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>&lt;ctrl&gt;end</td>
</tr>
<tr>
<td>home</td>
</tr>
<tr>
<td>&lt;ctrl&gt;home</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>pg up</td>
</tr>
<tr>
<td>pg dn</td>
</tr>
<tr>
<td>&lt;ctrl&gt;pg up</td>
</tr>
<tr>
<td>&lt;ctrl&gt;pg dn</td>
</tr>
<tr>
<td>Right, up, down, and left movement</td>
</tr>
<tr>
<td>Moves right one word</td>
</tr>
<tr>
<td>Moves left one word</td>
</tr>
<tr>
<td>Jumps to end of line</td>
</tr>
<tr>
<td>Jumps to bottom of screen</td>
</tr>
<tr>
<td>Jumps to beginning of line</td>
</tr>
<tr>
<td>Jumps to top of screen</td>
</tr>
<tr>
<td>Starts new line</td>
</tr>
<tr>
<td>Pages up one full screen</td>
</tr>
<tr>
<td>Pages down one full screen</td>
</tr>
<tr>
<td>Jumps to top of file</td>
</tr>
<tr>
<td>Jumps to end of file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLOCK COMMANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alt&gt; B or F7</td>
</tr>
<tr>
<td>&lt;alt&gt; E or F8</td>
</tr>
<tr>
<td>&lt;alt&gt; H</td>
</tr>
<tr>
<td>&lt;alt&gt; C</td>
</tr>
<tr>
<td>&lt;alt&gt; N</td>
</tr>
<tr>
<td>&lt;alt&gt; P</td>
</tr>
<tr>
<td>&lt;alt&gt; D</td>
</tr>
<tr>
<td>&lt;alt&gt; W</td>
</tr>
<tr>
<td>&lt;alt&gt; R</td>
</tr>
<tr>
<td>Marks beginning of a block</td>
</tr>
<tr>
<td>Marks end of a block</td>
</tr>
<tr>
<td>Hides or shows marked block</td>
</tr>
<tr>
<td>Copies block to current cursor location</td>
</tr>
<tr>
<td>Moves block to current cursor location</td>
</tr>
<tr>
<td>Prints block</td>
</tr>
<tr>
<td>Deletes marked block</td>
</tr>
<tr>
<td>Writes block to file (request for name of file appears)</td>
</tr>
<tr>
<td>Reads file block (request for name of file appears)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAIN OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL</td>
</tr>
<tr>
<td>ESC</td>
</tr>
<tr>
<td>&lt;ctrl&gt;E</td>
</tr>
<tr>
<td>&lt;ctrl&gt;L</td>
</tr>
<tr>
<td>&lt;ctrl&gt;W</td>
</tr>
<tr>
<td>&lt;ctrl&gt;F</td>
</tr>
<tr>
<td>&lt;ctrl&gt;R</td>
</tr>
<tr>
<td>Deletes character behind cursor</td>
</tr>
<tr>
<td>Aborts command (after a find or find and replace)</td>
</tr>
<tr>
<td>Deletes to end of line</td>
</tr>
<tr>
<td>Deletes line</td>
</tr>
<tr>
<td>Deletes word</td>
</tr>
<tr>
<td>Finds pattern</td>
</tr>
<tr>
<td>Finds and replaces</td>
</tr>
</tbody>
</table>
Appendix B

Description of the MOR/DS Language

The MOR/DS (microcomputer support for operations research and management science—discrete simulation) language was designed to be logically compatible with GPSS and SIMAN, but with a structure and features to make it easier to use and in many ways more powerful than either of these languages. The system design was, at the very start, oriented to students who are encountering simulation modeling and programming for the first time. However, the capabilities built into the language make MOR/DS a viable analysis tool for advanced modelers as well as for beginning students.

The system has several distinctive features intended to make the language easy to use, self-documenting, and powerful. A help system is provided, and it is accessed by pressing the F1 key. When the F1 key is pressed, a menu of help categories is listed on the screen. Any of the words in all upper-case letters are words for which there is help information. For example, to see the list of statements possible in the LOGIC segment, type LOGIC. To exit from the help system, press the ESC key. To obtain a summary of the editor commands, press <ctrl>F1.

The language is a highly structured programming language, somewhat patterned after Pascal. All data structures used in a model are named by the user with simple or numerically subscripted identifiers. The same syntax rule applies to global variables and to entity attributes. Arithmetic expressions can be as complex as needed, and the language allows boolean (logical) constructs to be embedded within arithmetical expressions. Expressions may be used as the subscripts for subscripted variables; this and the versatile nam-

ing conventions eliminate the need for the indirect addressing capabilities found in most simulation languages.

MOR/DS provides Pascal-like IF-THEN-ELSE, WHILE, REPEAT-UNTIL statements, a generalization of the CASE statement (called DO-CASES), and PRINT statements. Compound statements (a sequence of statements preceded by begin and terminated with end) are supported to facilitate structured programming. These statements significantly improve program readability and comprehension because there is little need to use transfer (GOTO) statements. The versatility of the print statement gives the user control over what information is printed, its format, and where it is printed; there is no need for the typical internal trace mechanism.

A real-time animation capability provides a simple and direct mechanism for obtaining a visual impression of the dynamics of the simulation. Finally, the simulator directly reads the editor data structure during the compilation process. When syntax errors occur, a diagnostic message is displayed, and the cursor is positioned near the point of the error. The diagnostic, the position of the cursor, and the on-line help system greatly enhance error resolution. The simulation language syntax is a free-form structure where the block and definition parameters can be listed in any order since they are designated by keywords.

Several printing conventions are followed when example MOR/DS programs are listed. Parameters and system variables will be designated as boldface words with the first letter capitalized (e.g., Time), block names and definition types will be boldface and all capitals (e.g., ARRIVE), and user-defined names will appear in typewriter font (e.g., myNumber). The parameters for blocks and definitions are enclosed in braces. (As a general rule, braces are used when order is not important; parentheses are used when order is important.) If a number contains a decimal point, there must be a digit to the left of the decimal point. Thus .4 is a legal number, but .4 is not permissible. It should be noted that all language statements must end with a semicolon. Thus, multiple statements per line may be used, or a single statement can extend over several lines.

B.1 Nature of MOR/DS Programs

An MOR/DS program is similar in appearance to programs written in languages such as Pascal, C, BASIC, and FORTRAN in that statements are processed in sequential order except for transfer of control. However, there is
a very crucial difference. Typical MOR/DS programs tend to consist of logically separate groups of statements—somewhat like multiple, concurrently running programs within a single program. Furthermore, these groups of statements are usually not activated by the same stream of entities. This brings us to the fundamental concept that distinguishes simulation programming from other types of programming.

Entities “flow” through a group of program statements, and each statement is activated by the entity that visits the statement. Such an entity is called the active entity; other entities that may be associated with the statement are called passive or secondary entities. For example, suppose one kind of entity represents customers lined up outside a playhouse waiting for it to open. These customers are “blocked” at a gate waiting for the door to open. A second entity, the manager, comes and opens the door. At this point, the manager is the active entity associated with an “open-the-door” command, and all of the customers are passive or secondary entities. Therefore, each entity in the system owns a specific statement, and any one statement may be owned by more than one entity. It is helpful to think of multiple copies of the program statements—one copy per entity in the system. Therefore, multiple programs are operating concurrently. This concept of concurrence and entities owning sections of the program is what distinguishes simulation programming from ordinary programming. In an ordinary program, an instruction pointer is used to designate the statement being executed (or to be executed next); in MOR/DS, this information (as well as other system information) is carried by each entity in the system. The simulator controller knows what to do next by checking the “next statement” information on the active entity.

While an MOR/DS program behaves as though it consists of concurrently operating sections (in fact, this is true for the simulation clock), in reality only one entity is actually active at any point in time. Each activity scheduled to occur in the future is placed on a “future event chain” and remains there until the controller needs to process another activity. At this point, the next entity in this list is removed, and it becomes the active entity. The controller dispatches the entity to the appropriate program statement and this statement is activated. This entity either passes through a series of statements without consuming (simulated) time, or it is blocked or delayed. If the latter occurs, an event is scheduled to take place sometime in the future, and the entity is placed back onto the future event chain.

3.2. BASIC LANGUAGE ELEMENTS

B.2 Basic Language Elements

B.2.1 Symbols

Letters: a to z, A to Z and _ (underscore)
Digits: 0 to 9
Special: + - * / , . () [] {} < > = ; , "
Two-character symbols: <> <= >= .

With the exception of character strings (discussed below), MOR/DS converts all letters to upper case.

B.2.2 Comments

Any sequence of characters between successive double quotes (""") is a comment and is ignored by the compiler. Judicious use of comments clarifies the meaning of programs. Comments can be placed anywhere in a program. Good programming practice dictates the liberal use of comments so that two months after the program is written, it can still be read.

B.2.3 Numerical Values

All numerical values are real numbers having up to 11 significant digits of accuracy. The valid range is $10^{-38}$ to $10^{38}$. Each real number occupies 6 bytes of memory. A number is specified as either a sequence of digits (whole number) or a sequence of digits, a decimal point, and a sequence of digits. A number cannot begin with a decimal point; thus, 0.5 is legal whereas .5 produces an error message. Numbers may be preceded by + or - to designate the sign of the number; unsigned numbers are positive. MOR/DS does not currently support scientific notation (“E” format).

B.2.4 Strings

A string is a sequence of characters (except ‘’ and ‘"’ enclosed within single quotes (’)). A string can be up to 80 characters in length. Case is respected in strings, unlike in symbolic names where all letters are converted to upper case.
B.2.5 System Constants

The following list contains the constants that are predefined by the system.

\[ \begin{align*}
\pi &= 3.14159265 \\
\pm \text{PosInf} &= 32000 \\
\text{PosInf} &= 10^{30} \\
\text{NegInf} &= -\text{PosInf}
\end{align*} \]

B.2.6 Identifiers

Identifiers are used to denote variables, in-line functions, and statement labels. An identifier consists of a symbolic name, or a symbolic name followed by a period followed by a numerical value. The former is a simple identifier, while the latter is a subscripted identifier.

Example

\[ \begin{align*}
x & \quad \text{simple variable} \\
y.1, y.5, y.6 & \quad \text{subscripted variables}
\end{align*} \]

A symbolic name is a sequence of between 1 and 15 characters beginning with a letter. The remaining characters may be any combination of letters, digits, and the underscore character.

In MOR/DS, an identifier actually denotes a family of one or more instances of the same type of data; the numeric subscript is a mechanism for referencing the members of a family. In the above examples, \( x \) denotes a single-member family, while \( y \) denotes a family consisting of at least three members.

The numerical subscript is a powerful device because any expression may be used to designate its value. There is one caution: since real arithmetic is used, there is the danger that the expression may not produce the precise value used to define the subscript originally.

All identifiers are declared and initialized in the DEFINITION segment of the program.

Scope of Identifiers

All variables are either global to the program or entity attributes. In the latter case, each entity carries the identifier with it.

B.2.7 System Variables

There are several predefined system variables that can be accessed by the user to determine the current status of the system. Of the predefined variables, only three of them may be changed by the user; the other variables are read only. The three read/write variables are: PreemptLevel, WatchChar, and Capacity. The list of system variables is given in Table B.1.

<table>
<thead>
<tr>
<th>Table B.1: System Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attribute Variables (see Section B.4.9)</strong></td>
</tr>
<tr>
<td>EntityNumber</td>
</tr>
<tr>
<td>PreemptLevel</td>
</tr>
<tr>
<td>ResumeTime</td>
</tr>
<tr>
<td>WatchChar</td>
</tr>
<tr>
<td><strong>System Variables</strong></td>
</tr>
<tr>
<td>ClockTime</td>
</tr>
<tr>
<td>DepartCount</td>
</tr>
<tr>
<td><strong>System Arrays (Functions)</strong></td>
</tr>
<tr>
<td>Available[name]</td>
</tr>
<tr>
<td>BlockCurrent[label]</td>
</tr>
<tr>
<td>BlockTotal[label]</td>
</tr>
<tr>
<td>Capacity[name]</td>
</tr>
<tr>
<td>CumNumber[name]</td>
</tr>
<tr>
<td>CumProductNT[name]</td>
</tr>
<tr>
<td>CumProductVT[name]</td>
</tr>
<tr>
<td>CumValue[name]</td>
</tr>
<tr>
<td>LastTime[name]</td>
</tr>
<tr>
<td>MaxNumber[name]</td>
</tr>
<tr>
<td>MaxUsed[name]</td>
</tr>
<tr>
<td>MaxValue[name]</td>
</tr>
<tr>
<td>RsrcNumber[name]</td>
</tr>
<tr>
<td>Used[name]</td>
</tr>
<tr>
<td><strong>Future Event Chain</strong></td>
</tr>
<tr>
<td>EventName</td>
</tr>
<tr>
<td>FutureEventChain</td>
</tr>
</tbody>
</table>
B.2.8 Expressions

An expression is a statement of rules for computing a numerical value. Expressions consist of operators and operands (function calls, variables, and numbers). A unary operator has a single operand; convention places the operand on the right side of the operator. A binary operator has both a left and right operand with the operator placed between its operands.

Operators fall into 5 categories, denoted by their order precedence, highest first:

1. Unary minus (change sign), unary plus (ignored), and logical (NOT)
2. Exponentiation (~)
3. Multiplying operators: *, /, MOD, AND, MAX, MIN
4. Adding operators: +, -, OR, XOR
5. Relational operators: =, », <<, >=, >, IN, NOTIN.

Expressions are evaluated left to right, and expressions in parentheses are evaluated first, beginning with the inner-most level. The result of all operators is a real number. Relational operators and logical operators (AND, OR, XOR, and NOT) return a value of 1.0 if the result is true (or nonzero) and a value of 0.0 if the result is false (or zero).

A special kind of expression used in testing situations (i.e., true or false) is called a boolean expression. A boolean expression is an expression whose final result is determined by the action of a relational operator. Because boolean expressions evaluate to real numbers (1.0 or 0.0) they may be incorporated in arithmetical expressions provided they are enclosed by parentheses. For example, the assignment x = 6 + (2 < 3) is equivalent to x = 6 + 1, and thus 7 is assigned to x. However, x = 6 + 2 < 3 is not legal in MOR/DS because it reduces to the form x = 8 < 3 and an assignment statement cannot directly use a boolean expression. (Actually the compiler has a problem because it does not expect to see boolean expressions except inside parentheses.) The compiler allows boolean expressions without the parentheses only when such an expression is required, such as in a test condition (see the IF, WHILE, and REPEAT statements in the LOGIC segment description below).

B.2. BASIC LANGUAGE ELEMENTS

Unary Operators

When used as signs for numbers and expressions, the symbols + and - serve as unary operators. The - sign is interpreted as a change-the-sign command while + is ignored (unsigned numbers are positive). NOT is a logical operator and it inverts or reverses the truth value of its operand.

Exponentiation

The symbol ~ denotes exponentiation and is the only right associative operator in the MOR/DS language. For example, 2~3~2 = 512, not 64.

Conventions: 0^n = 0 for all n, n^0 = 1 for all n not zero, x^n is valid if x is positive and n is any real number, or if x is negative and n is an integer. If x is negative and n is not an integer, a run-time error occurs.

Multiplying Operators

<table>
<thead>
<tr>
<th></th>
<th>numerical multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>numerical division</td>
</tr>
<tr>
<td>MOD</td>
<td>remainder</td>
</tr>
<tr>
<td>AND</td>
<td>logical and</td>
</tr>
<tr>
<td>MAX</td>
<td>larger of the operands</td>
</tr>
<tr>
<td>MIN</td>
<td>smaller of the operands</td>
</tr>
</tbody>
</table>

Adding Operators

<table>
<thead>
<tr>
<th></th>
<th>numerical addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>numerical subtraction</td>
</tr>
<tr>
<td>OR</td>
<td>logical or</td>
</tr>
<tr>
<td>XOR</td>
<td>logical exclusive or</td>
</tr>
</tbody>
</table>

Relational Operators

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>IN</td>
<td>set inclusion</td>
</tr>
<tr>
<td>NOTIN</td>
<td>not set inclusion</td>
</tr>
</tbody>
</table>
The IN and NOTIN operators take a real number or expression as the left
operand and a sequence or interval as the right operand. If the number is
in the sequence or interval, IN is true and NOTIN is false; otherwise IN is
false and NOTIN is true.

B.2.9 Functions

A function is similar to an operator in the sense that it may take 1 or
more numerical arguments. The syntax for using functions is

\[ \text{Identifier}(\text{ArgList}); \]

where \text{ArgList} is a list of numbers or expressions separated by commas. Pre-
defined functions are called system or intrinsic functions, while user-defined
functions are called in-line functions. The identifiers used in the argument
list may not be subscripted, and must be predefined. These identifiers are
temporary variables and are unknown to the system after the function de-
definitions (see B.4.6).

B.2.10 System Functions

In addition to user-defined functions, there are system (predefined) func-
tions that are always available to the modeler. These functions are as follows:

**Trigonometric functions** (\(x\) in radians)
- \(\text{Sin}(x)\)
- \(\text{Cos}(x)\)
- \(\text{Tan}(x)\)
- \(\text{ArcTan}(x)\)

**Exponential functions**
- \(\text{Exp}(x) = e^x\)
- \(\text{Ln}(x) = \log_e(x)\)
- \(\text{Log10}(x) = \log_{10}(x)\)
- \(\text{Sqr}(x) = x^2\)
- \(\text{Sqrt}(x) = \sqrt{x}\)

**Other functions**
- \(\text{Abs}(x)\) absolute value of \(x\)
  \(\text{Abs}(1.2) = 1.2, \text{Abs}(-1.2) = 1.2\)
- \(\text{Floor}(x)\) greatest integer \(\leq x\)
  \(\text{Floor}(1.8) = 1, \text{Floor}(-1.8) = -2\)
- \(\text{Ceiling}(x)\) least integer \(\geq x\)
  \(\text{Ceiling}(1.2) = 2, \text{Ceiling}(-1.2) = -1\)

**System Random Variate Generators**

The following functions produce random numbers according to the indi-
cated distributions.

- **Random** Continuous uniform distribution between 0 and 1.
- **Duniform(a,b)** Discrete uniform distribution between the integers \(a\) and \(b\) (inclusive), with \(a < b\).
- **Binomial(a,b)** Binomial distribution with the number of trials equal
to \(a\) and the probability of success equal to \(b\). The value of \(a\) must be
  a positive integer and \(b\) a real number between 0 and 1.
- **Poisson(a)** Poisson distribution with mean \(a\), with \(a > 0\).
- **Cuniform(a,b)** Continuous uniform distribution between \(a\) and \(b\),
  with \(a < b\).
- **Expd(a)** Exponential distribution with mean \(a\), with \(a > 0\).
- **Normal(a,b)** Normal distribution with mean \(a\) and standard devia-
tion \(b\), with \(a\) any value and \(b > 0\).
- **Beta(a,b)** Beta distribution with shape parameters \(a\) and \(b\). The
  pdf has the form \(f(x) = k x^{a-1}(1-x)^{b-1}\), where \(k\) is a normalizing
  constant, \(a, b > 0\) and \(0 \leq x \leq 1\). (Note that the mean is \(a/(a+b)\).)
- **Erlang(a,b)** Erlang distribution with mean \(a\) and type \(b\) (i.e., equi-
  valent to the sum of \(b\) independent exponentials each with mean \(a/b\)).
  The Erlang has a standard deviation of \(a/\sqrt{b}\), with \(a > 0\) and \(b\) a
  positive integer.
- **Gamma(a,b)** Gamma distribution with scale parameter \(a\) and shape
  parameter \(b\), with \(a, b > 0\). If the mean, \(\mu\), and variance, \(\sigma^2\), are
  known, then \(a = \sigma^2/\mu\) and \(b = \mu^2/\sigma^2\).
- Triangular \((a, b, c)\) Triangular distribution with lower limit \(a\), the mode \(b\), and the upper limit \(c\), with \(a < b < c\).
- Weibull \((a, b)\) Weibull distribution with scale parameter \(a\) and shape parameter \(b\). The cumulative distribution has the form \(F(x) = \exp\left(-\frac{x}{a}\right)^b\) for \(x > 0\), with \(a, b > 0\). (Note that the mean is \(a\Gamma(1+1/b)\)).

B.2.11 Types

All identifiers except for statement labels refer to specific data structures designated by one of the following types:

- string
- real
- array
- attributes
- sequence
- interval
- in-line function
- label
- textfile
- resource
- queue
- gate
- statistics
- chain
- histogram
- distributions
- entity

A formal definition of the above types and how they are used is provided in the documentation of the DEFINITION segment below.

B.2.12 Iterative Constructs

MOR/DS provides a special-purpose iterative construct for performing indexed summations and products, and for finding the largest or smallest element and the index of the largest or smallest element of a collection of indexed quantities. The general syntax is

\[
\text{OpName}(\text{Expr} : \text{Id IN Domain WITH Cond})
\]

\(\text{OpName}\) is one of SUM, PRODUCT, MAX, MIN, ARGMAX, or ARGMIN; \(\text{Id}\) is a temporary (unsubscripted) variable; \(\text{Domain}\) is a sequence or interval; and \(\text{Cond}\) is a boolean expression. The \(\text{WITH-Cond}\) clause is optional. For example, the first of the following constructs yields the sum of the squares of the first ten integers and the second yields \(a^{10}/10!\).

\[
\begin{align*}
\text{SUM}\{ \text{i}^2 : \text{i IN [1 .. 10]} \} \\
\text{PRODUCT}\{ \text{a/i} : \text{i IN [0 .. 10]} \text{ WITH } \text{i <> 0} \}
\end{align*}
\]

Note that the temporary variable \(\text{i}\) has no meaning outside the single statement, and it is an error to use a previously defined variable for the temporary variable.

B.2.13 Conditional Expressions

The keyword USE can be utilized for conditional expressions. This is particularly useful for functions that have different functional forms over different sections of their domain. For example, the cumulative distribution function for the uniform zero-one random number would be

\[
\begin{align*}
F(x) &= \text{USE 0 IF } x \leq 0 \text{ ELSE } \\
& \quad \text{USE } x \text{ IF } x \in [0, 1] \text{ ELSE } \\
& \quad \text{USE 1;}
\end{align*}
\]

B.3 MOR/DS Program Structure

All programs have the following structure:

PROGRAM
  DEFINITION segment;
  CONTROL segment;
  LOGIC segment;
END.

The reserved word PROGRAM signifies the beginning of an MOR/DS program. The reserved word END followed by a period provides a distinctive means of identifying the end of the program. The DEFINITION segment is where all variables are declared, the CONTROL segment provides parameter values to control the execution of the program, and the LOGIC segment provides the executable program statements of the simulation.

MOR/DS uses a structured language concept that allows and requires the user to declare every variable used by the program. Furthermore, all constructs used by the simulation program, including simulation constructs (resources, queues, etc.), are treated the same way as numerical structures. Unlike most common simulation languages, no pre-defined structures exist...
B.4 The DEFINITION Segment

This section describes the declaration of variables, as well as declarations of aliases (alternative names for reserved words) and symbolic names.

B.4.1 The ALIAS Feature

This feature is provided primarily to enhance the readability of MOR/DS models. The syntax of an ALIAS list is

\[
\text{ALIAS} = \{ \text{NameList} \};
\]

where \text{NameList} is a list of elements in the form

\[
\text{NewName} = \text{OldName}
\]

with the elements separated by commas.

For example, to allow a RESOURCE to be called a DRILL and the WAIT clock name to be called DrillingTime, we would use

\[
\text{ALIAS} = \{ \text{DRILL} = \text{RESOURCE}, \text{DrillingTime} = \text{WAIT} \};
\]

Thus, whenever the keyword RESOURCE is used, the "new" keyword DRILL can be used, and instead of WAIT, DrillingTime can be used. The names RESOURCE and WAIT remain valid keywords.

B.4.2 Declarations

A declaration has the following general syntax:

\[
\text{Id} : \text{Type} = \text{value};
\]

\text{Id} is an identifier, \text{Type} is a valid MOR/DS type, and \text{value} is an appropriate specification of the necessary information to initialize the variable. Declarations are separated by semicolons. The syntax for reals, in-line functions, attributes, labels, and entities are different.

The SEQUENCE

An MOR/DS sequence is a finite collection of real numbers defined by

\[
\text{Id} : \text{SEQUENCE} = < \text{NumList} >;
\]

There are two ways to represent \text{NumList}: the regular method and a shorthand version. The regular method is to represent \text{NumList} as a list of expressions separated by commas.

Examples of legal SEQUENCE declarations are

\[
\begin{align*}
\text{A} & : \text{SEQUENCE} = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle; \\
\text{B} & : \text{SEQUENCE} = \langle 1, 1.25, 1.5, 1.75, 2 \rangle; \\
\text{C} & : \text{SEQUENCE} = \langle 1, 1.3, 1.6, 1.9 \rangle; \\
\text{D} & : \text{SEQUENCE} = \langle 5, \text{Sqr}(5), 20/2 \rangle;
\end{align*}
\]

Note that the last example is the sequence \(\langle 5,25,10 \rangle\).

The shorthand version is where the elements of the sequence are equidistant from each other, in which case two dots represent the progression. The general shorthand notation takes the form

\[
\text{id} : \text{SEQUENCE} = \langle \text{Left} \ldots \text{Right} : \text{Mesh} \rangle;
\]

where \text{Left}, \text{Right}, and \text{Mesh} are expressions denoting the left end point, the right end point, and the mesh size, respectively. The \text{:Mesh} is optional, and a default of 1.0 is used if it is not specified. The number \(x\) is contained in the sequence \(\langle a;b d \rangle\) if, and only if, \(a \leq x \leq b\) and \(x = a + i\cdot d\) for some nonnegative integer \(i\).
For example, the declarations

\[
\begin{align*}
\mathbf{A} : \text{SEQUENCE} &= \langle 1 \ldots 8 \rangle; \\
\mathbf{B} : \text{SEQUENCE} &= \langle 1 \ldots 2 : 0.25 \rangle; \\
\mathbf{C} : \text{SEQUENCE} &= \langle 1 \ldots 2 : 0.3 \rangle;
\end{align*}
\]

are equivalent to the same named sequences defined above.

It is also possible to mix these two styles. Thus, we could also write the following equivalent definition for \(\mathbf{A}\):

\[
\mathbf{A} : \text{SEQUENCE} = \langle 1.4,5,6,8 \rangle;
\]

Sequences are evaluated at compilation time and thus cannot be redefined during execution. If a \text{SEQUENCE} is needed that will be dynamic during the simulation, then the \text{INTERVAL} data structure must be used.

The INTERVAL

In MOR/DS, an interval can either be discrete or continuous. The discrete interval is equivalent to a sequence except that it is not evaluated until execution. The general syntax for the discrete interval is similar to the sequence

\[
\text{Id} : \text{INTERVAL} = [\text{Left} .. \text{Right} : \text{Mesh}];
\]

For example, the declarations

\[
\begin{align*}
\mathbf{A} : \text{INTERVAL} &= [1 .. 8]; \\
\mathbf{B} : \text{INTERVAL} &= [1 .. 2 : 0.25]; \\
\mathbf{C} : \text{INTERVAL} &= [1 .. 2 : 0.3];
\end{align*}
\]

are equivalent to the same named sequences defined above.

The continuous interval is denoted as a closed interval with syntax

\[
\text{Id} : \text{INTERVAL} = [\text{Left}, \text{Right}];
\]

A major advantage of the \text{INTERVAL} is that the end points can be expressions whose values can change as the program is executed.

B.4. THE DEFINITION SEGMENT

B.4.3 Real Variables

The syntax to define simple numerical variables is

\[
\text{Id} = \text{Expr};
\]

\(\text{Id}\) is a (possibly subscripted) identifier and \(\text{Expr}\) is any expression. Real variable declarations do not use the colon reserved word format of other declarations. Examples of some legitimate real variable definitions are

\[
\begin{align*}
x &= 10; \\
y.1 &= z; \\
y.2 &= x + y.1;
\end{align*}
\]

Note that a variable can be used as soon as it is declared but not before. Thus, if the order of the above statements were reversed, an error message would occur during program compilation.

B.4.4 The ARRAY

MOR/DS provides a general type of array structure for manipulating tabular data. These arrays are more general than the arrays provided in most programming languages. The general syntax is

\[
\text{Id} : \text{ARRAY}[\text{DimList}] = \text{ArrayCon};
\]

\text{DimList} is a list of (up to 3) dimension specifications (a sequence) separated by commas and \(\text{ArrayCon}\) is the specification of values for the array.

\text{DimList} specifies the dimensions of an array by giving the sequence of index values allowed for each domain. Consider the table

\[
\begin{array}{c|ccc}
T & 12 & 24 & 26 \\
\hline
3 & 1 & 2 & 3 \\
5 & 4 & 5 & 6 \\
\end{array}
\]

The first dimension has domain \(<3,5>\) and the second dimension has domain \(<12, 24, 26>\). This table would be represented as an array defined by

\[
T : \text{ARRAY}[<3,5>, <12, 24, 26>] = \{(1, 2, 3), (4, 5, 6)\};
\]

Then, \(T[3,24] = 2\) and \(T[5,12] = 4\).

The \(\text{ArrayCon}\) specifies the values for each element of an array. In the above example the right-hand side illustrates how to form an \(\text{ArrayCon}\) for a
two-dimensional array. The elements of an array are stored row by row with the highest (i.e., last) dimension varying the fastest. Each row receives its values as a list enclosed in parentheses. In the example, the values for row 3 are \((1, 2, 3)\) and those for row 5 are \((4, 5, 6)\). Other legitimate definitions are

\[
\begin{align*}
A & : \text{ARRAY[<1..4>]} = (1, 2, 3, 4) ; \\
B & : \text{ARRAY[<1..2>, <1..2>, <1..3>]} = \\
& \quad \{ (111, 112, 113), (121, 122, 123) \\
& \quad \quad (211, 212, 213), (221, 222, 223) \} \\
\end{align*}
\]


A quick method to initialize all elements of an array to a given constant is the \text{ALL(val)} command, where \text{val} is any numerical constant. For example,

\[
E : \text{ARRAY[<1..2>, <1..2>, <1..3>]} = \text{ALL(16)} ;
\]

would assign the value of 16 to each of the twelve elements of \(E\).

### B.4.5 The STRING

A string is a sequence of no more than 80 ASCII characters, excluding the characters \('\) and \('\). The syntax is

\[
S : \text{STRING} = \text{'characters'} ;
\]

For example,

\[
S : \text{STRING} = \text{'I need understanding.'} ;
\]

### B.4.6 In-Line Functions

An in-line function is defined by the syntax

\[
\text{Id( ArgList )} = \text{Expr} ;
\]

\text{Id} is an identifier, \text{ArgList} is a (possibly empty) list of simple identifiers separated by commas, and \text{Expr} is a general arithmetic expression. The identifiers constituting \text{ArgList} are only temporary—they may be reused for subsequent purposes. However, any name used must not have been used for a prior global or attribute variable. In the following three examples, the first will result in an error during compilation.

#### B.4. THE DEFINITION SEGMENT

1. \(x = 10 ;\)
   \(f(x) = 14\times \text{Sin}(x/2) ;\) "This is illegal"
2. \(x = 10 ;\)
   \(f(x) = 14\times \text{Sin}(x/2) ;\) "This is legal"
3. \(g() = \text{Normal}(100,10) ;\) "This is legal"

The first example is illegal because at the time \(x\) is defined, the identifier \(x\) has already been used; the second example is correct and illustrates a proper reuse of the temporary \(x\). The parentheses are required when using \(g\) even though it has no argument; \(g\) is still a function. In-line functions may appear in any expression; all arguments are real.

The use of cumulative probability distribution functions for generating random variates is so common that a special structure has been developed to make their use easy. The key structure that is needed for generating arbitrarily distributed random variates is a mechanism for defining a function that represents the inverse of the cumulative probability distribution function for a discrete random variable. The statement has the form

\[
\text{myName : \text{DISTRIBUTION(class)} = ( sequence of ordered pairs ) ;}
\]

where \text{myName} is the user-defined name to represent the inverse of the distribution function being used to generate a random variable. A variable of type \text{DISTRIBUTION} is a function with a single argument. The definition \text{DISTRIBUTION} has a modifier \text{class}, which is either set to \text{Discrete} or \text{Continuous}. The parameter \text{list} within the brackets is a sequence of ordered pairs defining the function. Each ordered pair has the form \((xx,yy)\), where \(xx\) is the cumulative probability and \(yy\) is any real value. Each ordered pair must be separated by a comma. (Note also that the successive values of \(xx\) should be increasing, as well as the successive values of \(yy\), if the function is to be used for random variate generation.) For example,

\[
\text{myFunction : \text{DISTRIBUTION(Discrete)} = ( (0.5,0),(0.7,1),(1,2) ) ;}
\]

could be used to generate the numbers 0, 1, and 2 with probabilities 0.5, 0.2, and 0.3, respectively. Such a generation would be accomplished by the following statement written within the \text{LOGIC} segment of the program:

\[
x = \text{myFunction(Random)} ;
\]

If the definition statement is written as

\[
\text{myFunction : \text{DISTRIBUTION(Continuous)} = ( (0.5,0),(0.7,1),(1,2) ) ;}
\]
then the function is used to create a continuous random variable where linear interpolation is used between the points. The keyword MAPPING is synonymous with DISTRIBUTION (and thus can be used in place of it) and is more precise terminology since the function defined by the DISTRIBUTION statement need not satisfy the mathematical properties of a distribution. MOR/DS does no verification or checking on the list of data pairs. Consequently, unpredictable behavior will result if the xx values are decreasing.

B.4.7 TEXTFILE

The PRINT and PRINTLN statements, used in the LOGIC segment or EXITLOGIC subsegment, allow the user to generate output consisting of string and numerical values. These statements support the optional use of text files (i.e., files on either floppy or hard disks). A text file is declared by

\[ Id : TEXTFILE = \{ Name = MsDosName, Status = Write \}; \]

Name, Status, and Write are required reserved words, and MsDosName is a valid MSDOS file specification enclosed in single quotes. For example,

\[ f : TEXTFILE = \{ Name = 'MyOutput.out', Status = Write \}; \]

will cause output to be written to the disk file called MyOutput.out when used with a PRINT or PRINTLN statement described below. The compiler attempts to open a file at the time it is declared. If the file does not exist, it is created; if it exists, the.old contents are deleted, and it is prepared for new output from the simulation. The simulator automatically closes the file at the end of the run, or before issuing a run-time error. Therefore, user-defined output is captured even in the event of a fatal error. (Two standard MSDOS defined files are 'PRN' for the printer and 'CON' for the console.)

B.4.8 ATTRIBUTES

The variables carried with each entity are called attributes and can be of type REAL or of type ARRAY. The declarations are the same as for global variables except they occur in the attribute list as follows:

\[ ATTRIBUTES = \{ VarList \}; \]

B.4.9 System Predefined Attributes

In addition to user-defined attributes, MOR/DS provides the following pre-defined (system) entity attributes:

EntityNumber The sequential number given to the entity at the time of its creation (read only).

PreemptLevel The numerical priority value used by the PREEMPT block.

ResumeTime Residual WAIT time for a preempted entity (read only).

WatchChar The predefined name for the character to be used by the animation system when displaying entities. By default a blank is used. Any ASCII character may be used by including WatchChar = 'x' in the ATTRIBUTE list, where x is any ASCII character. An alternative is to use WatchChar = k, where k is an integer between 1 and 255 denoting a particular ASCII character. For example, WatchChar = 65 will result in the letter a being printed. This value can be changed at any time in the LOGIC segment.

B.4.10 ENTITY

A very powerful and unique feature of MOR/DS is the ability to look at and modify the attribute values for secondary or passive entities. This is made possible by entity variables, which are pointers to the data structures for passive entities. These pointers can only be set by the SEARCH block and CREATE block in the LOGIC segment. The declaration is

\[ Id : ENTITY; \]

with no initialization. For example

\[ Ent : ENTITY; \]
B.4. THE DEFINITION SEGMENT

Table B.2: MOR/DS DEFINITION structures and parameter syntax.
The keywords are optional unless labeled as required.
The default value is given in parentheses.

<table>
<thead>
<tr>
<th>ALIAS example</th>
<th>synonym statement for system words</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIAS = { Que = QUEUE, Stop = Halt };</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARRAY example</th>
<th>declaration for dimensioned variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>a : ARRAY [ &lt; 1, 2, 3 &gt; ] = (0, 0, 0);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATTRIBUTES example</th>
<th>variable and array definitions, naming, and initialization for all created entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTES = { cost = 0, Priority = 3, tseq : ARRAY [ &lt; 1, 2, 3 &gt; ] = ( 0, 0, 12 );</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAIN example</th>
<th>entity holding list to be used with LINK and UNLINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity example</td>
<td>max. no. entities in list simultaneously; (32000)</td>
</tr>
<tr>
<td>Discipline example</td>
<td>queueing discipline of waiting entities, FIFO, LIFO, or Choice attr_name; (FIFO)</td>
</tr>
<tr>
<td>Histogram example</td>
<td>flow time histogram name for queue statistics print, set to On or Off; (On)</td>
</tr>
<tr>
<td>StatOnOff example</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISTRIBUTION example</th>
<th>user-provided data pairs for function generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(type) data list</td>
<td>type is Continuous or Discrete interpolation list of (x, y) data pairs separated by commas</td>
</tr>
<tr>
<td>myDist : DISTRIBUTION(Discrete);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ENTITY example</th>
<th>defines a variable type to be an entity pointer used to reference attributes of an entity other than the current active entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>aEnt : ENTITY;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function example</th>
<th>user-defined inline functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument names are dummy variables and cannot be already defined</td>
<td></td>
</tr>
<tr>
<td>f(a, b) = b * Sin(a);</td>
<td></td>
</tr>
</tbody>
</table>
### Table B.2: continuation

**GATE**
- **Status**
  - logic switch that queues entities if closed
  - initial status is Open or Close; (Open)
- **Discipline**
  - queueing discipline of waiting entities,
  - FIFO, LIFO, or CHOICE(\textit{attr}.

**Histogram**
- **StatOnOff**
  - flow time histogram name
  - for queue statistics print, set to On or Off; (On)
  - \texttt{myGate} : \texttt{GATE} = \{ \textit{Status} = \text{Open},
  \text{Discipline} = \text{LIFO} \};

**HISTOGRAM**
- **Cells**
  - data collection device definition
  - number of cells or categories; Required
- **MinValue**
  - lower limit of histogram data; Required
- **MaxValue**
  - upper limit of histogram data; Required
- **Protect**
  - information is not cleared by Reset; (Off)
  - \texttt{hdata} : \texttt{HISTOGRAM} = \{ \textit{Cells} = 4,
  \text{MinValue} = 0, \text{MaxValue} = 16 \};
  - used as \texttt{hdata} = \text{value} ;

**INTERVAL**
- **example**
  - discrete interval (default mesh size = 1.0)
  - \texttt{range} : \text{INTERVAL} = \{ [1..9:3]
  \text{continuos interval is} \text{range} : \text{INTERVAL} = \{[1.9]}

**LABELS**
- **example**
  - all statement labels used in the \text{LOGIC} segment
  - \texttt{LABELS} = \{ \text{myLabel}, \text{yourLabel}, \text{here} \};

**MAPPING**
- **(type)**
  - user-provided data pairs for function generation
  - \texttt{type} is \text{Continuous} or \text{Discrete} interpolation
  - list of \((x, y)\) data pairs separated by commas
  - \texttt{myDist} : \text{MAPPING(Discrete)}
    = \{ \{0.1, 1\}, \{0.4, 3\}, \{1.0, 6\} \};

**QUEUE**
- **example**
  - entity holding list to be associated with a resource
  - through the resource definition statement.
  - max. no. entities in list simultaneously; (32000)
  - set equal to FIFO, LIFO, or
  - \texttt{CHOICE(expression)}; (FIFO)
  - flow time histogram name
  - for queue statistics print, set to On or Off; (On)
  - \texttt{ShopQ} : \texttt{QUEUE} = \{ \text{Discipline} = \text{CHOICE(pri)} \};

### B.4. THE DEFINITION SEGMENT

**Real**
- simple global variables can take any nonreserved name and can be subscribed using a period
  - \textit{x} = 1.23; \textit{y} = 0; \textit{xyz} = 2.3;

**RESOURCE**
- resource definition
  - capacity of resource; (1)
  - named queue to be used for entities that cannot seize the resource
  - \texttt{shop} : \texttt{RESOURCE} = \{ \text{Capacity} = 1 \};

**SEQUENCE**
- denotes a collection of discrete values
  - \texttt{s : SEQUENCE} = \{ \text{< 1, 2.34, 5 .. 10 >} \};

**STATISTICS**
- defines a flow time statistics collection device
  - name of histogram for distribution
  - information is not cleared by Reset; (Off)
  - \texttt{sysflow} : \texttt{STATISTICS} = \{ \text{Histogram} = \text{hdata} \};

**STRING**
- a list of characters enclosed in single quotes
  - \texttt{s : STRING} = \text{'Aabc def'}

**TEXTFILE**
- specifies a named (disk) file for \text{PRINT} and \text{PRINTLN} output
  - \text{Name} DOS file name (in single quotes); Required
  - may use 'Prn' for printer or 'Con' for console
  - \text{Status} \text{Write; Required}
    - (Read not implemented at this time)
  - \texttt{myFile} : \texttt{TEXTFILE} = \{ \text{\textit{Name}} = \text{‘A.out \text{.fil}',}
    \text{Status} = \text{Write} \};

**VARSTATISTICS**
- a real variable for which the system automatically tabulates time valued statistics
  - initial value; Required
  - \text{Value}
  - \texttt{x} : \text{VARSTATISTICS} = \{ \text{\textit{Histogram}} = \text{h,}
    \text{Value} = 0, \text{\textit{Protect}} = \text{On} \};
B.5 Control Segment

The **CONTROL** segment provides the mechanisms for controlling run length, resetting statistics for an initial run interval, periodic printing of results, controlling the random number generator, and setting up the animation system. Table B.3 provides a summary of all elements of the **CONTROL** segment. All statements within this segment must be terminated by a semicolon.

Table B.3: **MOR/DS CONTROL parameters**.

<table>
<thead>
<tr>
<th>BlockListing</th>
<th>If On (default), a summary for counts in each statement is printed; if set equal to Off, the summary counts are not printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear</td>
<td>Used within <strong>EXITLOGIC</strong> to clear system, including <strong>ClockTime</strong></td>
</tr>
<tr>
<td>Continue</td>
<td>Used within <strong>EXITLOGIC</strong> to continue simulation</td>
</tr>
<tr>
<td>PrintCount</td>
<td>System statistics print increment in departures</td>
</tr>
<tr>
<td>PrintTime</td>
<td>System statistics print increment in time units</td>
</tr>
<tr>
<td>Randomize</td>
<td>Randomizes the starting random number seed (On, Off)</td>
</tr>
<tr>
<td>Reset</td>
<td>Used within <strong>EXITLOGIC</strong> to reset statistics</td>
</tr>
<tr>
<td>ResetCount</td>
<td>Departure count at which system statistics are zeroed</td>
</tr>
<tr>
<td>ResetTime</td>
<td>Time at which statistics are all zeroed out</td>
</tr>
<tr>
<td>Seed</td>
<td>Starting random number seed, integer from 1 to 32767</td>
</tr>
<tr>
<td>StatError</td>
<td>Set equal to Off to disable certain fatal errors</td>
</tr>
<tr>
<td>StatListing</td>
<td>Statistics summary information, default is On</td>
</tr>
<tr>
<td>StopCount</td>
<td>Run length in departure counts</td>
</tr>
<tr>
<td>StopTime</td>
<td>Simulation run length in clock units</td>
</tr>
<tr>
<td>VarListing</td>
<td>Global variable summary information, default is On</td>
</tr>
<tr>
<td>WatchDelay</td>
<td>Microseconds delay for <strong>WatchList</strong> (default is 300)</td>
</tr>
<tr>
<td>WatchList</td>
<td>List of resource, queue, chain, and gate names for animation, optional symbols shown are by appending either a single character or an ASCII index number</td>
</tr>
<tr>
<td>example</td>
<td><strong>WatchList</strong> = { <strong>myRes</strong>: '^R', <strong>myQue</strong>: 254 }; controls the execution of the <strong>WatchList</strong></td>
</tr>
<tr>
<td>WatchStatus</td>
<td>Default is On</td>
</tr>
</tbody>
</table>

Provided within the scope of the **CONTROL** segment is an optional logic segment, called **EXITLOGIC**, which allows the full complement of MOR/DS programming statements. **EXITLOGIC**, if present, is executed at the termination of a simulation run. It is actually a subsegment within the **CONTROL** segment that provides a mechanism to perform special tasks, to close down a run, and to structure and control multiple runs of the same model. The beginning of the **EXITLOGIC** section is identified by the keyword **EXITLOGIC** and terminated by **END**.

With the exception of **SETGATE** (discussed in the next chapter), no simulation commands are permitted within **EXITLOGIC**. However, all other programming statements are permitted. There are three special statements especially designed for control purposes. These are:

1. **CONTINUE**, which resumes the current run as though the ESC key had been pressed or the **INTERRUPT** statement executed.
2. **RESET**, which continues the same simulation run but clears all non-protected statistics variables and leaves **ClockTime** as is.
3. **CLEAR**, which deletes all entities, clears all non-protected statistics variables, and resets **ClockTime** to zero.

In all three cases, user variables and the system variables **StopTime**, **StopCount**, **ResetTime**, and **ResetCount** remain at their current values.

**Caution**: When a program is first executed, global variables are initialized in the **DEFINITION** segment; however, when the **EXITLOGIC** is executed, the initialization in the **DEFINITION** segment is not repeated. It is the user's responsibility to ensure that all variables, user and system, are properly set for subsequent runs of the same model. A common error is to forget to reset the status for gates.

Two statements are available in both the **EXITLOGIC** and the **LOGIC** segment to cause the execution to stop: **HALT** and **ABORT**. A **HALT** in the **LOGIC** segment causes the current run to terminate with control being passed to **EXITLOGIC**. A **HALT** in **EXITLOGIC** suspends its operations, and control is passed to the **LOGIC** segment provided either **CLEAR**, **CONTINUE**, or **RESET** was encountered prior to the **HALT**. The **ABORT** command causes immediate termination of the entire simulation program. If **ABORT** is executed in the **LOGIC** segment, control does not pass back to **EXITLOGIC**; instead, the program is immediately terminated with normal statistical output being displayed before closing down the run.
When activated, EXITLOGIC begins execution at the first statement and operates in the same fashion as an ordinary program; there are no active or inactive entities.

B.6 Logic Segment

A statement in MOR/DS is a command having a specific meaning. A statement begins with a keyword identifying the command or a variable name if the command is an assignment. Statements may be prefixed by a label identifier that specifies the target statement for transfer commands.

Labels are ordinary identifiers. A label is declared in the DEFINITION segment and receives its value when it serves as the prefix for a statement. To aid in the program's readability, a colon must be placed immediately after a label. The compiler issues a fatal error message if unresolved labels exist after the program has been compiled.

Consider the following three lines of code:

```
GOTO inspect;
inspect : SEIZE { Name = Inspector };
IF NoOfDefects > 0 THEN GOTO inspect;
```

In the example, inspect is a label; its value is the statement number of this particular SEIZE. It is referenced by the two GOTO statements. Notice that the first GOTO makes a forward reference to inspect even though at this point the label inspect has not yet received its value. The second GOTO makes a back reference to inspect. Both the forward and backward references are permissible.

B.6.1 Statement Classification

Statements are either simple or compound statements. Simple statements are statements that contain no other statements. A compound statement consists of a sequence (possibly empty) of statements prefixed with the reserved word begin and suffixed by the reserved word end. The compiler treats a compound statement as a single statement. The primary use of compound statements is to enable the execution of more than one statement in commands expecting a single statement (see IF, WHILE, FOR, and DO-CASES below). A compound statement may be used anywhere a statement

is allowed simply for the purpose of enhancing readability. A compound statement need not contain any statements: begin end; is a legitimate compound statement.

The statements available in MOR/DS can be classified by functionality as follows:

1. Assignment statements
2. Transfer statements
3. Conditional statements
4. Repetitive statements
5. Print statements
6. Simulation command statements

Assignment Statement

The assignment statement is used to set the contents of a specific variable to a specified value. The syntax is

\[ Id = expression; \]

\( Id \) denotes a real variable, an array variable element, or a system variable. For example:

\[ A[5,7] = 2x + \sin(\frac{y}{3}); \]

Transfer Statements

Transfer statements refer to the unconditional transfer of the active entity to a statement other than the next sequential statement. The two types of transfer statements are GOTO and GOSUB.

The GOTO Statement. The syntax for the GOTO statement is

```
GOTO lab;
```

where \( lab \) is a label identifier.
The GOSUB and RETURN Statements. MOR/DS provides a BASIC-like subroutine facility that is accomplished via the GOSUB and RETURN statements.

GOSUB label;
causes the entity to execute the statement indicated by label. RETURN causes the entity to return to the statement immediately following the most recent GOSUB. An alternate form of the RETURN is

RETURN { Destination = label };
which overrides the default return location.

In the following example, a subroutine is set up to print the contents of the attribute array val. Such a subroutine simplifies the model because this printing process is to be used at several different places in the model.

ListVals: FOR (i=1, i < n, i=i+1) DO
  PRINT { val[i]:8, ' ' };
  PRINTLN { };
RETURN;

ARRIVE { Time = Cuniform(3,10) };
FOR (i=1, i < n, i=i+1) DO
  val[i] = Duniform(10,100);
GOSUB ListVals;

Conditional Statements

A conditional statement selects for execution a single one of its component statements based upon some type of selection condition. The two conditional statements supported by MOR/DS are the IF and DOCASES statements.

The IF Statement. There are two forms of the IF statement:

IF bExpr THEN Stmt1;
IF bExpr THEN TrueStmt ELSE FalseStmt;

where bExpr is a boolean expression and Stmt1, TrueStmt, and FalseStmt are statements (simple or compound). The ELSE-FalseStmt clause is optional, but must not be preceded by a semicolon. If the bExpr is true, then TrueStmt is executed; otherwise FalseStmt is executed (if it exists). In either case, the statement following the IF statement is executed next unless a transfer statement is contained in the selected statement.

Two examples of the IF usage are

IF x > 5 THEN
  begin
    SEIZE { Name = Waiter };
    WAIT { time = 5 };
    RELEASE { Name = Waiter };
  end
ELSE
  begin
    PRINTLN { 'Not enough jobs yet.' };
    GOTO GetMoreJobs;
  end;

IF x > 6 THEN
  IF y IN <0,1> THEN y=0
  ELSE y = 5;

Notice that in the second example, the IF x > 6 statement does not have an ELSE clause; the ELSE clause is associated with the IF y IN <0,1> statement.

The DOCASES Statement. The DOCASES statement is functionally similar to the computed GOTO statement in FORTRAN and BASIC and structurally similar to the CASE statement in Pascal. In FORTRAN and BASIC an explicit transfer to a labeled statement is made, while in Pascal and MOR/DS the transfer is implicit in order to emphasize the complete structure of the construct. A specific statement from a collection of statements comprising the DOCASES structure is selected and executed. The syntax is

DOCASES Expr OF
  CASE value1 : Stmt1;
  CASE value2 : Stmt2;
  .
  .

where Expr is an expression whose value is a list of values and Stmt1, Stmt2, . are statements.
CASE value1: Stmt1;
ELSE elseStmt;
END;

For each j, valuej is a real number, an interval, or a sequence, and Stmtj is a single (simple or compound) statement. The ELSE - elseStmt clause is optional and is selected if none of the CASE values match the selection value, Expr. Good programming practice dictates using the ELSE clause to capture unexpected selector values or those values that do not warrant special treatment.

An example of the DOCASES statement is

DOCASES JobType OF
   CASE <1,2,5> : PRINTLN {'0ne, three or 5'};
   CASE 2 : begin
      SEIZE { Name = Drill };
      WAIT { Time = 5 };
      RELEASE { Name = Drill };
      end; "case 2"
   CASE <4> : PRINTLN {'Finally gct a four!'};
   ELSE
      PRINTLN {'Unexpected JobType: ', JobType:4};
   END; "DOCASES"

Repetitive Statements

Repetitive statements provide a convenient way to execute statements more than once. MOR/DS supports three of these statements: the WHILE, REPEAT, and FOR statements. All three are called indefinite loop statements because they use a test condition to determine when to terminate the process of repetition rather than repeat a fixed number of times such as the FORTRAN Do and the BASIC and Pascal FOR statements. Therefore, care needs to be exercised to avoid infinite loops.

The WHILE Statement. The syntax for the WHILE statement is

WHILE BoolExpr DO Stmt;

BoolExpr is a boolean expression and Stmt is a single program statement (simple or compound). Note that the test condition is performed prior to the execution of the statement; this means that the statement need not be executed at all.

An example of the WHILE usage is

i=1;
WHILE i <=5 DO
      PRINTLN { A[i]:8:2 };
      i = i + 1;
end;

The REPEAT Statement. The syntax for the REPEAT statement is

REPEAT
   Stmt1;
   Stmt2;
   ...
   Stmtn;
UNTIL BoolExpr;

Stmt1, Stmt2, ..., Stmtn are statements and BoolExpr is a boolean expression used to control the loop. Since the test condition is last, the sequence of statements in the construct is executed at least once.

An example of the REPEAT is

i = 1;
REPEAT
      PRINTLN { A[i]:8:2 };
      i = i + 1;
UNTIL i > 5;

The key difference between the WHILE and REPEAT statements is that in the REPEAT - UNTIL structure the boolean expression is evaluated after the statements (always causing at least one execution), whereas in the WHILE - DO structure, the boolean expression is evaluated first.

CAUTION: The BoolExpr in both of the above statements can take on an arbitrary form and may use any combination of global, system, or attribute variables. However, exercise care if any statement contained within the construct can delay or hold the active entity. For example,
Steps = 1;
WHILE Steps <= NoSteps DO
begin
  c = routing[type, Steps];
  SEIZE { Name = Drill.c };
  WAIT { Time = ProcessingTime(c) };
  RELEASE { Name = Drill.c };
  Steps = Steps + 1;
end;

If c is a global variable, then it is likely that it will have changed value between the SEIZE and RELEASE blocks; if so, a fatal error may occur because the RELEASE may attempt to access the wrong element of the Drill family.

The FOR Statement. The syntax for the FOR statement is

    FOR (assignStmt, BoolExpr, incrStmt) DO doStmt;

In the FOR – DO loop, an index variable (defined in the DEFINITION segment) is initialized in assignStmt, then if BoolExpr is true, doStmt is executed; the index variable is then incremented according to incrStmt and then BoolExpr is evaluated again. As long as the BoolExpr remains true, doStmt is executed, and incrStmt is executed immediately after it. For example, the following is equivalent to the WHILE and the REPEAT examples above.

    FOR (i=1, i<= 5, i=i+1) DO PRINTLN {i:2};

Of course, it is also possible to use a compound statement for the DO portion. The use of a compound statement would look something like the following:

    FOR (assignStmt, BoolExpr, incrStmt) DO
    begin
      Stmt1;
      Stmt2;
    ...
    Stmtm;
    end;

The Print Statements

The following two statements provide a flexible mechanism for outputting specific information during the execution of a simulation model. Both statements perform the same basic function, which is to print the individual elements listed within the statement. However, PRINTLN terminates the line with a carriage return and linefeed and PRINT does not. The destination of the output is controlled in two ways. If the optional file identifier is present, output goes to the associated disk file; otherwise, the output is sent to the screen, printer, or OUTPUT.BOS file depending on the active choice made in the editor. The syntax forms are:

    PRINT or PRINTLN;
    PRINT {} or PRINTLN {};
    PRINT {f} or PRINTLN {f};
    PRINT {f, PrintList} or PRINTLN {f, PrintList};
    PRINT {PrintList} or PRINTLN {PrintList};

In these statements, f is a TEXTFILE variable, and PrintList is a sequence of print elements separated by commas. A print element is either a string or a real value. In the case of real values, an optional format suffix may be used to control the field width and number of digits to the right of the decimal point. Note that if a destination file variable (cf type TEXTFILE) is used, it must be the first entry in the element list.

A string is specified either by a variable of type STRING or a string literal (a sequence of characters enclosed by single quotes).

The optional numerical format suffix has the form :fExpr:dExpr where fExpr and dExpr are expressions denoting the field width and number of digits to the right of the decimal point, respectively. The dExpr part is optional, and if not specified, the default is 0. If the format suffix is not present the default is :8:2. Note that the field width must be at least as large as the number of decimal point digits + 1 to include the decimal point. If the numerical value to be printed cannot fit within the field, a series of +’s or -’s are printed depending on the sign of the value.

The CLEARSCREEN and EJECTPAGE Commands

The CLEARSCREEN command will erase the video screen. This command has no effect if the simulation output is directed to the printer or to a file. The EJECTPAGE command inserts a control code into the output
going either to a file or to the printer that causes the printer to eject a page. 
This command has no effect on the video screen.

B.6.2 Simulation Block Statements

The MOR/DS language provides a wide array of simulation block statements (or “blocks”), each performing a specific function or collection of functions. The term block is an allusion to the diagramic interpretation of 
simulation statements used in many common discrete simulation languages such as GPSS and SIMAN. These languages associate a distinct figure or 
diagram with each statement to facilitate the modeling and presentation of 
models. The help system and Tables B.1 through B.4 provide sufficient 
documentation on the variables, DEFINITION segment, simulation blocks, and 
other pertinent information for the modeler to use MOR/DS effectively and 
correctly. Unfortunately, situations arise where the modeler must know ex-
actly how MOR/DS operates. The following discussion provides information 
regarding the operation of each simulation block statement. This discussion 
begins with a brief summary of the internal structure and then the block 
statements are treated alphabetically. (Some blocks, such as COPY and 
GATHER, are similar in function and are treated in the same paragraph.)

A simulation block statement is like a procedure (subroutine) call in 
MOR/DS. Therefore, most block statements have one or more parameters 
(operands) that control the nature of the function to be performed. Some 
parameters are required and some are optional; each block has its own rules 
regarding its parameters and their purpose(s). The general syntax is

\[ \text{label} : \quad \text{BLOCKNAME} \{ \text{List of named parameters} \}; \]

The list of named parameters may be empty or may contain multiple entries 
separated by commas. The block label is optional and may be used on any 
block except for ARRIVE. A named parameter entry has the form

\[ \text{KEYWORD} = \text{associated value} \]

where the associated value is governed by the context.

The parameter entries may be placed in any order within the list. The 
set braces ({} and { }), used to delimit the parameters, were chosen to emph-
size that, as in a set, the order of the elements is not important. Naming 
the parameters requires more typing, but it makes the model more readable, 
in addition to allowing for the free-form structure of the parameter lists.

B.6. LOGIC SEGMENT

When an optional parameter is not specified, the compiler uses the corre-
sponding default. Table B.4 (at the end of this appendix) provides detailed 
descriptions of all simulation block statements.

Entity Data Structure

Each entity consists of system attributes, user-declared attributes, and 
a collection of pointers to lists. Table B.1 summarizes the accessible system 
attributes. Each entity contains pointers to the list of seized RESOURCES, 
the list of open STATISTICS structures, and the list of subroutine return 
addresses. In addition, the surrogate entity that emerges from a BUNDLE 
block contains a pointer to the list of entities it represents. A fatal error 
occurs when an entity enters a DEPART statement and has control over a 
RESOURCE or has open STATISTICS. (This restriction can be disabled 
by the CONTROL directive StatError = Off. This feature is intended for 
advanced users.)

The ARRIVE, CREATE, and SCHEDULE Block Statements

These block statements provide the only means for introducing entities 
into the system. ARRIVE is under system control, while CREATE and 
SCHEDULE are under user control. Each entity receives a chronologi-
cal identification number, called the EntityNumber. When an entity is created 
(either by ARRIVE or CREATE), its system attributes are assigned, as are 
the default values for user attributes.

ARRIVE is actually not a simulation block statement in the usual sense; 
rather, it is a generator. It is treated as a statement only in the sense that its 
location governs the entry point for the arriving entity. It is illegal to branch 
an entity into an ARRIVE block. Each ARRIVE block is initiated at the 
beginning of the simulation run. The Limit parameter specifies the number 
of groups created by the block (infinity if no Limit is given); the group 
size is specified by the Quantity parameter (1 if Quantity is not specified), 
and the interarrival time is specified by the expression in the required Time 
parameter. The first entity (or group of entities) is scheduled to arrive at 
time equal to zero plus Offset plus Time, where Offset is 0 if not specified. 
The next entity (or group) is scheduled for time ClockTime plus Time, 
provided the Limit count has not been met.

NOTE: So long as Limit has not been exceeded, the scheduled group of 
entities is contained in the current block count for the ARRIVE block.
CREATE is under the control of the active entity, and the created entity (groups of entities cannot be created) is placed into the entity pointer specified by the Entity parameter. The created entity is set up as in the ARRIVE block except its existence to the system is known only by the entity pointer. The modeler sets up the attributes and then passes the created entity to the SCHEDULE block. SCHEDULE schedules the entity specified by the Entity parameter to enter the system at the statement given by the Destination parameter at time specified by the Time parameter.

The CHAIN, GATE, QUEUE, and STATISTICS Structures

These four structures all share a similar statistical information block, details of which are listed in Table B.1. CHAINS, QUEUES and GATES are structurally identical but logically distinct. QUEUES are associated with RESOURCES, and access to and from them is determined by the SEIZE, RELEASE, and PREEMPT blocks. Modelers create ordered lists of entities by using the CHAIN structure and they control access by using LINK and UNLINK. GATES produce ordered lists of entities that are controlled by SETGATE and TESTGATE. Each STATISTICS structure consists of a statistical information block that is activated through STARTSTAT and deactivated by ENDSTAT.

While CHAIN, GATE, and QUEUE are associated with lists of entities being blocked, STATISTICS structures have no such physical association. The first three block types may trap active entities and hold them from progressing further—this is in addition to the statistical computations that always take place. Quite differently, STARTSTAT creates a pointer to a record (internal to the entity) that stores the current time as well as a pointer to the specific STATISTICS structure given by the Name parameter. Nothing else happens to the active entity and nothing is done to the actual STATISTICS structure. When the entity passes through ENDSTAT, the system checks to determine if the entity has gone through a corresponding STARTSTAT. If so, the elapsed time is used to update the statistical data maintained in the named structure, and the pointers mentioned above are cleared. It is an error to dispose (DEPART, COMBINE or UNBUNDLE) an entity that has not gone through an ENDSTAT if it had previously entered a STARTSTAT. (Putting StatError = Off in the CONTROL segment disables this fatal error.)

The COMBINE, GATHER, and BUNDLE Block Statements

These blocks provide the means to collect entities, but each performs a different function. All work under the same concept called grouping, which works as follows. If the UseAttribute parameter is used, a different group is formed for each distinct value observed. Each entering entity decrements the Quantity counter (let us assume this value is n, set by Quantity=n), either by 1 or by the value specified by the Decrement parameter. The group is closed when the count reaches zero, and a new group is established when a new UseAttribute value is encountered.

The COMBINE block deletes all but the last entity from the system; it is only the last entity that emerges from the block. GATHER simply collects n entities into a group and lets them all pass through once the nth member enters the block. BUNDLE works like GATHER except a new entity, called the surrogate entity, is created when the nth enters the group. Only surrogate entities emerge from the BUNDLE block. The surrogate is a complete copy of either the first or last entity in the group. In addition, the surrogate maintains a pointer to a list of all n entities it represents. The UNBUNDLE block is the only mechanism to recover the entities maintained by a surrogate.

CAUTION: Be sure you are aware of seized RESOURCES and opened STATISTICS that belong to the entities entering a COMBINE, GATHER, or BUNDLE block.

The COPY and UNBUNDLE Block Statements

The COPY block duplicates the active entity and routes the copies to the destination specified by the CopiesTo parameter; the number of copies is specified by the Quantity parameter. Each entity receives a chronological EntityNumber. The copies do not receive any of the list pointers of the active entity, and therefore are not in STATISTICS structure and are not in control of any RESOURCE. Not observing this last fact leads to subtle modeling errors, so be careful!

The UNBUNDLE block reintroduces the entities in the list kept by the active surrogate entity. The surrogate is destroyed, and its constituent entities are placed on the Future Event Chain pointing to the block immediately following the UNBUNDLE block. The UNBUNDLE block should be entered only by a surrogate entity (i.e., those created by a BUNDLE block). Be sure the surrogate entity has released any previously seized RE-
SOURCEs or has closed any previously opened STATISTICS blocks before it enters the UNBUNDLE block.

The DEPART Block Statement

The DEPART block causes the active entity to be deleted from the MOR/DS system. If specified, the Quantity parameter determines the integer value subtracted from the DepartCount. (The DepartCount is initially set in the CONTROL segment by the StopCount parameter.) If no parameter is specified, the DepartCount is not changed. If the departing entity has control of one or more RESOURCES or has any opened STATISTICS (if no StatError = Off has been set), a fatal error occurs.

The HALT, ABORT, and INTERRUPT Commands

HALT immediately terminates the current run and returns control to EXITLOGIC (if specified). ABORT terminates the entire system and overrides any EXITLOGIC commands. A HALT executed in EXITLOGIC terminates this instance of EXITLOGIC processing. If a CONTINUE, RESET, or CLEAR command was encountered, the next run will immediately begin; otherwise, the simulation will terminate. ABORT is always global in impact: the simulation terminates. INTERRUPT suspends the run and activates the interrupt menu. This statement offers program control of the ESC key. If e - continue is selected from the menu, the run continues with the statement immediately following the INTERRUPT.

The LINK and UNLINK Block Statements

An entity entering a LINK block is placed on the named CHAIN in accordance with the declared (in the DEFINITION segment declaration) discipline. An entity entering an UNLINK block causes entities in the named CHAIN to be routed to the required Destination. Quantity specifies the maximum number to unlink, and Condition provides a matching criterion. The boolean expression of Condition uses the attributes of the inactive entities on the CHAIN, not the attributes of the active entity.

The PREEMPT Block

The PREEMPT block is the only alternative to SEIZE for an entity to request a RESOURCE. The difference is that preemption allows an entering entity to displace (temporarily) one or more entities currently in control of all or part of the RESOURCE. The entity attribute PreemptLevel expresses a relative level of importance to the entities on the RESOURCE list and the preempting entity. Preemption is a rather complicated process and sometimes produces subtle and unexpected consequences. We therefore suggest experimenting with this facility until its exact behavior is understood. This discussion should guide you to the aspects that need study.

The preemption entity is permitted to displace (preempt) an entity on the RESOURCE list only if three conditions are true: (1) the preempting entity must have a smaller PreemptLevel than the resource controlling (other) entity; (2) the entity to be preempted must be residing in a WAIT block (it makes sense to preempt an entity only if it is actively being “processed” by the RESOURCE—and this is modeled by a time delay); and (3) there must be sufficient capacity (Available plus those units that can be preempted) for the entering entity. If not, the entering entity joins the resource QUEUE (with room checking just like a seize), or if NotPreemptedTo is specified the entity is routed out of the PREEMPT. If preemption does occur, the active (preempting) entity bypasses the QUEUE and immediately seizes the RESOURCE. Two issues remain to discuss: what happens to the preempted entities and exactly which entities get preempted?

The RESOURCE entity list is a FIFO ordering of those entities actively in control of RESOURCE units. This list is scanned, front to back, to check PreemptLevel and residence in a WAIT block. Each preemptable entity is preempted until the Available and freed capacity equals or exceeds the requested units. Each such preempted entity frees its RESOURCE units and then is either routed by the IfPreemptedTo parameter or is placed at the front of the resource QUEUE (ignoring queue discipline). The entity remains on the RESOURCE list but cannot be preempted twice (it cannot still be in its WAIT block). Each preempted entity has a residual time left in the WAIT, and this quantity is stored in the ResumeTime attribute. If Resume=On is specified, then the preempted entity resumes operation in the WAIT block, otherwise the Time parameter is reused to establish a new waiting time.

Caution: the preempted entity is placed at the front of the queue used by the RESOURCE, regardless of the discipline of the queue. However, the preempted entity has no special significance to the queue discipline, so if FIFO or CHOICE is used, it is possible for a preempted entity to be pushed back into the queue. Secondly, an entity that has seized or preempted a resource may subsequently seize or preempt the same resource
again before encountering a RELEASE. The internal list of entities controlling the resource maintains the total sum of requested units each entity has, but the position of the entity does not change, even with changes in the PreemptLevel. Of course, no entity can ever preempt itself!

The RELEASE Block

The RELEASE block actually performs two functions. First, it frees the specified number of units seized by the active entity and updates statistics. (Note that an entity may use a different number of RELEASEs to give back all its RESOURCE units than the number of SEIZEs it used to acquire the units.) Second, it invokes a loading algorithm to take entities from the resource QUEUE, if one exists, to fill the available space. The process is sequenced in such a manner that queue entities get the RESOURCE before any entity at a SEIZE or PREEMPT. All of the timing is controlled via the Future Event Chain. (Refer to The RESOURCE Structure description for technical details.)

Suppose Available = n after freeing the units of the active entity. The loading algorithm searches the resource QUEUE from the front, taking entities that will fit in the RESOURCE until Available = 0 or no more will fit. No optimization is performed, and only one pass is made through the QUEUE. Consequently, the RESOURCE may not be completely filled and one or more entities may be bypassed because they do not fit at the time they are scanned. Except for these bypassed entities, the ordering of the QUEUE is preserved. Notice that this scanning algorithm begins at the front of the QUEUE; preempted entities, if any, are sitting at the front and thus are given the first opportunity to enter the RESOURCE.

All entities taken from the QUEUE get placed on the Future Event Chain in order to get into the RESOURCE. However, preempted entities return to the WAIT block where they were when preempted. Other entities pass out of the SEIZE or PREEMPT block to the next (usually a WAIT) block.

The RESET, CLEAR, and CONTINUE Commands

When the RESET command is encountered, all statistical quantities are cleared (but entities are not cleared). It can be employed in the LOGIC segment (at most once per run) or in EXITLOGIC. The latter is preferred because more control is available there.

CLEAR and CONTINUE are available only in EXITLOGIC. The CLEAR command returns the system (but not global variables!) to the initial condition. All entities are deleted from the system and statistics-gathering mechanisms are reset. No global variables, including gates, are altered, nor are control variables reset (such as StopTime, StopCount, etc.). Furthermore, the system clock is not reset. CONTINUE allows tabulation of preliminary results (batch sampling) and then the simulation resumes where it left off. A Protect option is available for statistical constructs to prevent them from being reinitialized by the above commands.

The RESOURCE Structure

Each RESOURCE contains a block of statistics information, including Available, Used, and Capacity. This information is documented in Table B.1. In addition, the RESOURCE maintains a list of entities currently having control of the RESOURCE (from SEIZE and PREEMPT). This list is of particular concern to the modeler when working with preemption. Any entity that is preempted and has not been routed out of the RESOURCE by PreemptedTo is kept on this list and is displayed by the animation system. Such an entity has temporarily given up control of its requested units but is still counted as being associated with the RESOURCE. An entity is removed from this list only by passing through a RELEASE block or being preempted and routed out of the RESOURCE.

The SEIZE Block Statement

The SEIZE block provides the standard mechanism for an entity to gain control of a specified number of RESOURCE capacity units given by the Units parameter. Assume Units = n. If Available ≥ n, then the entity bypasses the resource QUEUE and is allocated n units of the RESOURCE; Used and Available are updated accordingly. On the other hand, if Available < n, the entity cannot seize the RESOURCE and must either wait on the resource QUEUE or be routed elsewhere. If Excess appears in the parameter list, the entity immediately leaves the SEIZE block and goes to the specified destination. Otherwise, the system attempts to place the entity on the QUEUE in accordance with the active QUEUE discipline. If there is no room in the QUEUE and a QExcess destination is provided, the system so routes the entity; otherwise, a fatal error message is displayed and the simulation run terminates. It is possible (and sometimes necessary) for
an entity to execute more than one SEIZE for a resource before releasing
it. The only hidden consequence of this process is that the entity does not
change its position on the RESOURCE's entity list. This fact comes into
play only if such an entity is preempted, because although it may change
its PreemptLevel after the first SEIZE, the entity may be unexpectedly
preempted because of its position on the list.

NOTE: The resource QUEUE statistics do not have zero-time entries
because a seizing entity does not pass through the QUEUE if Available ≥

The SEARCH and REMOVE Block Statements

Any list, such as RESOURCE, GATE, QUEUE, CHAIN, or the Future
Event Chair (EventChain), can be scanned by the SEARCH block,
and any entity can be removed from these constructs via the REMOVE
block. SEARCH scans the list identified by the Name parameter starting
at position Begin (1, if not specified) and terminating after position End
(the last entity if not specified). The first entity that satisfies the matching
criteria Condition parameter (the first entity encountered, if Condition
is not specified) is returned in the Entity parameter (if specified) and its
position is returned in the Rank parameter (if specified). The boolean
expression of Condition uses the attributes of the inactive entity (entities on
the linked list) rather than the attributes of the active entity. Rank must
be a global variable or an attribute—if the latter, the result goes into the
active entity. At least one of the parameters Rank or Entity must be
specified. If no matching entity is found or if the list is empty, the Rank is set
to 0 and the Entity is set to NIL. The purpose of SEARCH is to identify
a specific entity on a list or to alter the attributes of such an entity. In
most simulation systems, the only way to alter the attributes of an entity
is with one or more assignment statements; this, of course, requires the enti-


\[
\text{REMOVE takes the entity off the named list either by position, if Rank}
\text{is specified, or by the value of the ENTITY pointer. The deleted entity}
\text{is routed to the location specified by the required Destination parameter.}
\text{Unlike SEARCH, Rank and Entity parameters cannot both appear. The
preferred method for using REMOVE is to do so with an ENTITY variable,}
\text{and the following setup should be used:}
\]

\[
\text{IF NOT NIL(\text{ent}) THEN REMOVE \{Name = thing, entity=ent,}
\text{Destination = Loc\};}
\]

We recommend never using REMOVE without obtaining the ENTITY by
SEARCH, to avoid unintended, unexpected, and even disastrous results.
Furthermore, the REMOVE should immediately follow the SEARCH to
make sure no intervening event changes the list. Never place a delay- or
blocking-type statement between a SEARCH and a REMOVE because EN-
TITY variables are global, not attributes.

The STARTSTAT and ENDSTAT Block Statements

These blocks allow easy computation of flow times. An entity enter-
ing STARTSTAT marks an entering time in a special location on the entity
structure (the quantity actually registered is established in the STATISTICS
declaration). ENDSTAT computes the flow time for this entity. (See the
previous paragraph dealing with CHAIN, QUEUE, GATE, and STATIS-
TICS.) An error occurs if the STATISTICS variable encountered in END-
STAT was not used by a matching STARTSTAT.

The TESTGATE and SETGATE Block Statements

An entity entering the TESTGATE passes directly through the named
GATE if that gate is open, or it is blocked at the GATE list if the gate is
closed. The named GATE is opened, closed, or has its status inverted by
the SETGATE block. RESET, CONTINUE, and CLEAR statements do
not alter the status of any GATE, so be sure to establish the correct status
explicitly.

The WAIT Block Statement

The active entity is delayed at this block for the Time specified.
B.6.3 Future Events List

In this subsection, we describe the default mechanisms used in scheduling events onto the future events list.

Zero Time Delays

When an entity enters a \texttt{WAIT \{Time = 0\}} block, that entity is placed on the future events list behind any entities that are due out at the current clock time.

If an entity enters a \texttt{RELEASE} block, the units of the resource that have been released are returned to any preempted entities. If there remain units of the resource available after the preempted entities have reclaimed what they need, then the queue is investigated. If any entities are on the queue that can now get into the resource, they are placed on the future events list ahead of all other entities. The original entity that just passed through the \texttt{RELEASE} block is placed on the future events chain behind any entities that are due out at the current clock time; that is, before this entity proceeds, it goes through a zero time delay. This protects against an entity releasing a resource and then re-seizing it before any queued entities.

If an entity opens a gate, any entities that are released by the gate maintain their original blocking sequence and are placed ahead of all other entities on the future events list. The active entity (that opened the gate) proceeds to the next block (i.e., is not placed on the future events list).

If an entity releases or creates entities by passing through an \texttt{UNLINK}, \texttt{COPY}, \texttt{UNBUNDLE}, \texttt{REMOVE}, or \texttt{SCHEDULE} block, the new entities are placed on the future events list behind any entities that are due out at the current clock time. The active entity always proceeds directly to the next block (i.e., is not placed on the future events list).

---

Table B.4: \texttt{MOR/DS LOGIC} modeling blocks and parameter syntax.
The keywords are optional unless labeled as required.
The default value is given in parentheses.
\textit{All blocks except ARRIVE can have a branching label.}

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ABORT}</td>
<td>causes immediate termination of the simulation</td>
</tr>
<tr>
<td>\texttt{ARRIVE}</td>
<td>schedules entities into system</td>
</tr>
<tr>
<td>\texttt{Time}</td>
<td>interarrival time - expression; Required</td>
</tr>
<tr>
<td>\texttt{Offset}</td>
<td>delay time before arrive block becomes active; (0)</td>
</tr>
<tr>
<td>\texttt{Limit}</td>
<td>maximum number of batches; (\infty)</td>
</tr>
<tr>
<td>\texttt{Quantity}</td>
<td>number of entities in each arrival batch; (1)</td>
</tr>
<tr>
<td>\texttt{example}</td>
<td>\texttt{ARRIVE {Time = \texttt{DUniform}(2,12), Limit = 5}};</td>
</tr>
<tr>
<td>\texttt{BUNDLE}</td>
<td>temporarily forms a surrogate to represent a group of entities</td>
</tr>
<tr>
<td>\texttt{Quantity}</td>
<td>number to form one surrogate; Required</td>
</tr>
<tr>
<td>\texttt{Decrement}</td>
<td>amount of decrease in counter for forming bundle</td>
</tr>
<tr>
<td>\texttt{UseAttribute}</td>
<td>specified attribute used for grouping</td>
</tr>
<tr>
<td>\texttt{Surrogate}</td>
<td>set to First or Last, surrogate attributes; (Last)</td>
</tr>
<tr>
<td>\texttt{example}</td>
<td>multiple levels of temporary groupings allowed</td>
</tr>
<tr>
<td>\texttt{CLEARSCREEN}</td>
<td>clears the screen</td>
</tr>
<tr>
<td>\texttt{COMBINE}</td>
<td>a group of entities is replaced by 1 entity</td>
</tr>
<tr>
<td>\texttt{Quantity}</td>
<td>only the final entity leaves the block</td>
</tr>
<tr>
<td>\texttt{Decrement}</td>
<td>number of entities to combine into one; Required</td>
</tr>
<tr>
<td>\texttt{UseAttribute}</td>
<td>an expression that specifies the decrement to the counter initialized by Quantity; (1)</td>
</tr>
<tr>
<td>\texttt{example}</td>
<td>the user-defined entity attribute used for grouping</td>
</tr>
<tr>
<td>\texttt{COMBINE}</td>
<td>\texttt{COMBINE {Quantity = 3, UseAttribute = \texttt{v}}};</td>
</tr>
<tr>
<td>\texttt{COPY}</td>
<td>creates copies of the active entity into a copy group</td>
</tr>
<tr>
<td>\texttt{Quantity}</td>
<td>all resulting entities leave as independent entities</td>
</tr>
<tr>
<td>\texttt{CopiesTo}</td>
<td>the number of copies to be made; Required</td>
</tr>
<tr>
<td>\texttt{example}</td>
<td>destination for copies; Required</td>
</tr>
<tr>
<td>\texttt{CREATE}</td>
<td>\texttt{COPY { Quantity = 2, CopiesTo = \texttt{label}}};</td>
</tr>
<tr>
<td>\texttt{Entity}</td>
<td>creates an entity (activated by \texttt{SCHEDULE} block)</td>
</tr>
<tr>
<td>\texttt{example}</td>
<td>name of the \texttt{ENTITY} data type containing</td>
</tr>
<tr>
<td>\texttt{CREATE}</td>
<td>address of the created entity; Required</td>
</tr>
<tr>
<td>\texttt{example}</td>
<td>\texttt{CREATE { Entity = \texttt{myEntity}}};</td>
</tr>
</tbody>
</table>
**Table B.4: continuation**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEPART</strong></td>
<td>removes entities from system</td>
</tr>
<tr>
<td><strong>Quantity</strong></td>
<td>number to be subtracted from StopCount; (0)</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>DEPART { }; or DEPART { Quantity = 3 };</td>
</tr>
<tr>
<td><strong>DOCASES</strong></td>
<td>initial statement of a set of case statements</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>ELSE clause is optional; must end with END</td>
</tr>
<tr>
<td></td>
<td>DOCASES i OF</td>
</tr>
<tr>
<td></td>
<td>Case 1: ( x = x + 1 );</td>
</tr>
<tr>
<td></td>
<td>Case ( &lt;2,3&gt; ): ( x = x + 3 );</td>
</tr>
<tr>
<td></td>
<td>Case [4..7]: ( x = x - 7 );</td>
</tr>
<tr>
<td></td>
<td>ELSE ( x = 0 );</td>
</tr>
<tr>
<td></td>
<td>END;</td>
</tr>
<tr>
<td><strong>ENDSTAT</strong></td>
<td>enters flow time from STARTSTAT into statistics</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>statistics information name; Required</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>ENDSTAT { Name = systlow };</td>
</tr>
<tr>
<td><strong>EJECTPAGE</strong></td>
<td>pages on the printer</td>
</tr>
<tr>
<td><strong>FOR</strong></td>
<td>performs loop according to specified increment counters</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>FOR ( i = 0, i &lt;= 7, i = i + 2 ) DO</td>
</tr>
<tr>
<td></td>
<td>PRINT { i:3.0 }; (prints values 0, 2, 4, 6)</td>
</tr>
<tr>
<td><strong>GATHER</strong></td>
<td>entities wait for Quantity of entities before leaving</td>
</tr>
<tr>
<td><strong>Quantity</strong></td>
<td>number to group before all are released; Required</td>
</tr>
<tr>
<td><strong>UseAttribute</strong></td>
<td>specified attribute used for grouping</td>
</tr>
<tr>
<td><strong>Decrement</strong></td>
<td>specified decrease in counter for arriving entities; (1)</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>GATHER { Quantity = 3, UseAttribute = count, Decrement = count*2 };</td>
</tr>
<tr>
<td><strong>GOSUB</strong></td>
<td>send entity to subroutine at specified label</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>return of entity occurs by RETURN statement</td>
</tr>
<tr>
<td></td>
<td>GOSUB myLabel;</td>
</tr>
<tr>
<td><strong>GOTO</strong></td>
<td>unconditional branch to block label</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>GOTO inspect;</td>
</tr>
<tr>
<td><strong>HALT</strong></td>
<td>method for program termination of the simulation</td>
</tr>
<tr>
<td><strong>IF</strong></td>
<td>conditional branch statement; ELSE clause is optional</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>IF Random &lt;= 0.5 THEN ( x = x + 1 )</td>
</tr>
<tr>
<td></td>
<td>ELSE GOTO myLabel;</td>
</tr>
<tr>
<td><strong>INTERRUPT</strong></td>
<td>interrupts execution, causes the Interrupt menu</td>
</tr>
<tr>
<td></td>
<td>to be displayed as if ESC had been pressed</td>
</tr>
<tr>
<td><strong>LINK</strong></td>
<td>places entity on specified chain list</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>name of CHAIN for linking; Required</td>
</tr>
<tr>
<td><strong>Cexcess</strong></td>
<td>if chain is full then branch location</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>: LINK { Name = ShopC, Cexcess = myLabel };</td>
</tr>
<tr>
<td><strong>PREEMPT</strong></td>
<td>interrupts seized resource units</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>units are returned by a RELEASE block</td>
</tr>
<tr>
<td><strong>Units</strong></td>
<td>resource name; Required</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>number of units requested; (1)</td>
</tr>
<tr>
<td><strong>IfNoPreempt</strong></td>
<td>transfer label if resource units not available; (queue)</td>
</tr>
<tr>
<td><strong>PreemptedTo</strong></td>
<td>transfer label of preempted entities; (queue)</td>
</tr>
<tr>
<td><strong>Resume</strong></td>
<td>set to On or Off, if Off then preempted</td>
</tr>
<tr>
<td><strong>Qexcess</strong></td>
<td>entity restarts time when returned to resource; (On)</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>transfer label if space not available in queue</td>
</tr>
<tr>
<td></td>
<td>PREEMPT { Name = Shop, Units = 2,</td>
</tr>
<tr>
<td></td>
<td>IfNoPreempt = label };</td>
</tr>
<tr>
<td><strong>PRINT</strong></td>
<td>prints the argument list without a linefeed</td>
</tr>
<tr>
<td><strong>PRINTLN</strong></td>
<td>prints the argument list with a linefeed</td>
</tr>
<tr>
<td></td>
<td>if a variable of type TEXTFILE is first, prints to file;</td>
</tr>
<tr>
<td></td>
<td>otherwise, prints to selected output device</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>PRINT { i, 'x is', x:3.3, ' time is', ClockTime:5:0 };</td>
</tr>
<tr>
<td><strong>RELEASE</strong></td>
<td>returns held units to resource</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>resource name; Required</td>
</tr>
<tr>
<td><strong>Units</strong></td>
<td>number of units returned; (1)</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>RELEASE { Name = Shop, Units = 1 };</td>
</tr>
<tr>
<td><strong>REMOVE</strong></td>
<td>removes a specified entity from a queue, chain, or gate</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>name of list; Required</td>
</tr>
<tr>
<td><strong>Rank</strong></td>
<td>variable containing rank of entity to be removed</td>
</tr>
<tr>
<td><strong>Entity</strong></td>
<td>entity variable containing entity pointer</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>either Rank or Entity (not both) are required</td>
</tr>
<tr>
<td><strong>Destination</strong></td>
<td>the location for removed entity: (next block)</td>
</tr>
<tr>
<td></td>
<td>REMOVE { Name = myQue, Entity = aimt };</td>
</tr>
<tr>
<td><strong>REPEAT</strong></td>
<td>causes statements to be repeated until the condition</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>in the UNTIL portion is true</td>
</tr>
<tr>
<td></td>
<td>( x = 0; ) REPEAT ( z = z - 1 ); UNTIL ( x = 10 );</td>
</tr>
<tr>
<td><strong>RESET</strong></td>
<td>method for program control for resetting statistics</td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
<td>returns entities to statement following GOSUB unless alternative branch is used</td>
</tr>
<tr>
<td><strong>Destination example</strong></td>
<td>optional alternative return</td>
</tr>
<tr>
<td><strong>SCHEDULE</strong></td>
<td>establishes the time in which an entity created by CREATE block will become active</td>
</tr>
<tr>
<td><strong>Entity</strong></td>
<td>ENTITY variable containing address of entity to be scheduled; Required</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>the time at which entity will become active; Required</td>
</tr>
<tr>
<td><strong>Destination example</strong></td>
<td>location at which entity enters simulation; (next block) SCHEDULE {Entity = ent, Time = ClockTime+5};</td>
</tr>
<tr>
<td><strong>SEARCH</strong></td>
<td>scans the queue, chain, or gate for first entity satisfying given condition, returns entity pointer or rank</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>name of list to be searched; Required</td>
</tr>
<tr>
<td><strong>Condition</strong></td>
<td>boolean expression compared against list entities (true)</td>
</tr>
<tr>
<td><strong>Begin</strong></td>
<td>starting rank of scan; (1)</td>
</tr>
<tr>
<td><strong>End</strong></td>
<td>last possible rank to test; (current contents)</td>
</tr>
<tr>
<td><strong>Rank</strong></td>
<td>variable name for returning position in list (zero if not found)</td>
</tr>
<tr>
<td><strong>Entity</strong></td>
<td>entity typed variable name for returned pointer (Nil if not found)</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>either Rank or Entity (possibly both) are required</td>
</tr>
<tr>
<td><strong>SEIZE</strong></td>
<td>captures units from defined resource if available</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>resource name; Required</td>
</tr>
<tr>
<td><strong>Units</strong></td>
<td>number of units requested; (1)</td>
</tr>
<tr>
<td><strong>Rexcess</strong></td>
<td>transfer label if resource units not available (queue)</td>
</tr>
<tr>
<td><strong>Qexcess</strong></td>
<td>transfer label if space not available in queue</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>SEIZE {Name = Shop, Units = 2, Rexcess = label};</td>
</tr>
<tr>
<td><strong>SETGATE</strong></td>
<td>modifies named gate status</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>gate name; Required</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>(Open, Close, or Invert); Required</td>
</tr>
<tr>
<td><strong>example</strong></td>
<td>SETGATE {Name = myGate, Status = Open};</td>
</tr>
</tbody>
</table>

| **STARTSTAT** | tags entities with time of passing for statistics |
| **Name example** | statistics information name; Required |
| **STARTSTAT** | {Name = sysflow}; |
| **TESTGATE** | if named gate is closed, entity is queued until a SETGATE causes gate to be opened |
| **Name** | name of gate to be tested by entity; Required |
| **IfClosed example** | branch destination if gate is closed |
| **TESTGATE** | {Name = myGate, IfClosed = out}; |
| **UNBUNDLE** | replaces the surrogate entity by the group it represents, surrogate is destroyed |
| **example** | UNBUNDLE { }; |
| **UNLINK** | takes entities off the specified chain list |
| **Name example** | chain name; Required |
| **Quantity** | number of entities to unlink; Required |
| **Destination example** | the location for unlinked entities; (next block) boolean expression for unlinking entities; (true) |
| **Condition** | UNLINK : Name = Shop, Quantity = 1; |
| **WAIT** | causes entities to wait for specified time |
| **Time example** | time entity is held in block; Required |
| **WAIT** | {Time = Expd(5*Random)}; |
| **WHILE** | repeats statements as long as condition is true |
| **example** | x = 0; WHILE x < 10 DO x = x + 1; |
Appendix C

Miscellaneous Tables
### Table C.1. Normal distribution

<table>
<thead>
<tr>
<th>$Z$</th>
<th>0.03</th>
<th>0.04</th>
<th>0.05</th>
<th>0.06</th>
<th>0.07</th>
<th>0.08</th>
<th>0.09</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.5003</td>
<td>0.5040</td>
<td>0.5080</td>
<td>0.5120</td>
<td>0.5160</td>
<td>0.5199</td>
<td>0.5239</td>
</tr>
<tr>
<td>0.1</td>
<td>0.7593</td>
<td>0.7632</td>
<td>0.7671</td>
<td>0.7710</td>
<td>0.7749</td>
<td>0.7788</td>
<td>0.7827</td>
</tr>
<tr>
<td>0.2</td>
<td>0.6179</td>
<td>0.6217</td>
<td>0.6255</td>
<td>0.6293</td>
<td>0.6331</td>
<td>0.6368</td>
<td>0.6406</td>
</tr>
<tr>
<td>0.3</td>
<td>0.5584</td>
<td>0.5621</td>
<td>0.5658</td>
<td>0.5694</td>
<td>0.5730</td>
<td>0.5766</td>
<td>0.5802</td>
</tr>
<tr>
<td>0.4</td>
<td>0.5015</td>
<td>0.5050</td>
<td>0.5085</td>
<td>0.5120</td>
<td>0.5154</td>
<td>0.5188</td>
<td>0.5222</td>
</tr>
<tr>
<td>0.5</td>
<td>0.4392</td>
<td>0.4427</td>
<td>0.4461</td>
<td>0.4495</td>
<td>0.4529</td>
<td>0.4562</td>
<td>0.4595</td>
</tr>
<tr>
<td>0.6</td>
<td>0.3849</td>
<td>0.3883</td>
<td>0.3916</td>
<td>0.3949</td>
<td>0.3982</td>
<td>0.4014</td>
<td>0.4047</td>
</tr>
<tr>
<td>0.7</td>
<td>0.3226</td>
<td>0.3257</td>
<td>0.3288</td>
<td>0.3319</td>
<td>0.3350</td>
<td>0.3381</td>
<td>0.3412</td>
</tr>
<tr>
<td>0.8</td>
<td>0.2597</td>
<td>0.2626</td>
<td>0.2655</td>
<td>0.2684</td>
<td>0.2713</td>
<td>0.2742</td>
<td>0.2771</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1950</td>
<td>0.2015</td>
<td>0.2079</td>
<td>0.2143</td>
<td>0.2207</td>
<td>0.2271</td>
<td>0.2335</td>
</tr>
</tbody>
</table>

### Table C.2. Student's $t$-distribution

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\alpha = .10$</th>
<th>$\alpha = .05$</th>
<th>$\alpha = .025$</th>
<th>$\alpha = .01$</th>
<th>$\alpha = .005$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.073</td>
<td>3.144</td>
<td>3.276</td>
<td>3.356</td>
<td>3.437</td>
</tr>
<tr>
<td>2</td>
<td>1.889</td>
<td>2.920</td>
<td>3.031</td>
<td>3.143</td>
<td>3.240</td>
</tr>
<tr>
<td>3</td>
<td>1.633</td>
<td>2.533</td>
<td>2.845</td>
<td>3.071</td>
<td>3.291</td>
</tr>
<tr>
<td>4</td>
<td>1.533</td>
<td>2.332</td>
<td>2.718</td>
<td>2.966</td>
<td>3.144</td>
</tr>
<tr>
<td>5</td>
<td>1.475</td>
<td>2.105</td>
<td>2.576</td>
<td>2.764</td>
<td>2.962</td>
</tr>
<tr>
<td>6</td>
<td>1.425</td>
<td>2.015</td>
<td>2.478</td>
<td>2.648</td>
<td>2.831</td>
</tr>
<tr>
<td>7</td>
<td>1.381</td>
<td>1.934</td>
<td>2.390</td>
<td>2.552</td>
<td>2.684</td>
</tr>
<tr>
<td>8</td>
<td>1.345</td>
<td>1.860</td>
<td>2.306</td>
<td>2.454</td>
<td>2.602</td>
</tr>
<tr>
<td>9</td>
<td>1.317</td>
<td>1.796</td>
<td>2.225</td>
<td>2.365</td>
<td>2.520</td>
</tr>
<tr>
<td>10</td>
<td>1.293</td>
<td>1.740</td>
<td>2.150</td>
<td>2.288</td>
<td>2.440</td>
</tr>
<tr>
<td>11</td>
<td>1.272</td>
<td>1.691</td>
<td>2.081</td>
<td>2.218</td>
<td>2.364</td>
</tr>
<tr>
<td>12</td>
<td>1.253</td>
<td>1.645</td>
<td>2.019</td>
<td>2.152</td>
<td>2.289</td>
</tr>
<tr>
<td>13</td>
<td>1.236</td>
<td>1.602</td>
<td>1.963</td>
<td>2.091</td>
<td>2.216</td>
</tr>
<tr>
<td>14</td>
<td>1.221</td>
<td>1.561</td>
<td>1.919</td>
<td>2.034</td>
<td>2.145</td>
</tr>
<tr>
<td>15</td>
<td>1.208</td>
<td>1.524</td>
<td>1.878</td>
<td>1.983</td>
<td>2.076</td>
</tr>
<tr>
<td>16</td>
<td>1.197</td>
<td>1.490</td>
<td>1.842</td>
<td>1.935</td>
<td>2.010</td>
</tr>
<tr>
<td>17</td>
<td>1.187</td>
<td>1.459</td>
<td>1.811</td>
<td>1.890</td>
<td>1.944</td>
</tr>
<tr>
<td>18</td>
<td>1.178</td>
<td>1.430</td>
<td>1.782</td>
<td>1.848</td>
<td>1.880</td>
</tr>
<tr>
<td>19</td>
<td>1.170</td>
<td>1.403</td>
<td>1.755</td>
<td>1.808</td>
<td>1.817</td>
</tr>
<tr>
<td>20</td>
<td>1.162</td>
<td>1.378</td>
<td>1.730</td>
<td>1.770</td>
<td>1.755</td>
</tr>
<tr>
<td>21</td>
<td>1.155</td>
<td>1.355</td>
<td>1.706</td>
<td>1.734</td>
<td>1.694</td>
</tr>
<tr>
<td>22</td>
<td>1.148</td>
<td>1.333</td>
<td>1.684</td>
<td>1.700</td>
<td>1.654</td>
</tr>
<tr>
<td>23</td>
<td>1.142</td>
<td>1.312</td>
<td>1.663</td>
<td>1.668</td>
<td>1.615</td>
</tr>
<tr>
<td>24</td>
<td>1.136</td>
<td>1.292</td>
<td>1.644</td>
<td>1.636</td>
<td>1.577</td>
</tr>
<tr>
<td>25</td>
<td>1.131</td>
<td>1.273</td>
<td>1.627</td>
<td>1.615</td>
<td>1.540</td>
</tr>
<tr>
<td>26</td>
<td>1.126</td>
<td>1.255</td>
<td>1.611</td>
<td>1.596</td>
<td>1.504</td>
</tr>
<tr>
<td>27</td>
<td>1.121</td>
<td>1.238</td>
<td>1.596</td>
<td>1.579</td>
<td>1.468</td>
</tr>
<tr>
<td>28</td>
<td>1.116</td>
<td>1.222</td>
<td>1.582</td>
<td>1.563</td>
<td>1.433</td>
</tr>
<tr>
<td>29</td>
<td>1.111</td>
<td>1.206</td>
<td>1.569</td>
<td>1.548</td>
<td>1.400</td>
</tr>
<tr>
<td>30</td>
<td>1.107</td>
<td>1.191</td>
<td>1.557</td>
<td>1.534</td>
<td>1.368</td>
</tr>
<tr>
<td>40</td>
<td>1.083</td>
<td>1.159</td>
<td>1.495</td>
<td>1.476</td>
<td>1.291</td>
</tr>
<tr>
<td>60</td>
<td>1.051</td>
<td>1.125</td>
<td>1.440</td>
<td>1.427</td>
<td>1.240</td>
</tr>
<tr>
<td>120</td>
<td>1.024</td>
<td>1.099</td>
<td>1.394</td>
<td>1.384</td>
<td>1.205</td>
</tr>
<tr>
<td>$\infty$</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>


### Table C.3. Chi-square distribution

<table>
<thead>
<tr>
<th>n</th>
<th>$\alpha = .01$</th>
<th>$\alpha = .05$</th>
<th>$\alpha = .025$</th>
<th>$\alpha = .01$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00157</td>
<td>0.00962</td>
<td>0.00339</td>
<td>0.0158</td>
</tr>
<tr>
<td>2</td>
<td>0.0201</td>
<td>0.0566</td>
<td>0.0431</td>
<td>0.0614</td>
</tr>
<tr>
<td>3</td>
<td>0.115</td>
<td>0.216</td>
<td>0.352</td>
<td>0.584</td>
</tr>
<tr>
<td>4</td>
<td>0.534</td>
<td>1.210</td>
<td>2.111</td>
<td>3.433</td>
</tr>
<tr>
<td>5</td>
<td>0.872</td>
<td>1.240</td>
<td>1.640</td>
<td>2.200</td>
</tr>
<tr>
<td>6</td>
<td>1.24</td>
<td>1.690</td>
<td>2.170</td>
<td>2.830</td>
</tr>
<tr>
<td>7</td>
<td>1.65</td>
<td>2.180</td>
<td>2.730</td>
<td>3.490</td>
</tr>
<tr>
<td>8</td>
<td>2.09</td>
<td>2.700</td>
<td>3.330</td>
<td>4.170</td>
</tr>
<tr>
<td>9</td>
<td>2.56</td>
<td>3.250</td>
<td>3.940</td>
<td>4.870</td>
</tr>
<tr>
<td>10</td>
<td>3.05</td>
<td>3.820</td>
<td>4.570</td>
<td>5.580</td>
</tr>
<tr>
<td>11</td>
<td>3.57</td>
<td>4.400</td>
<td>5.220</td>
<td>6.300</td>
</tr>
<tr>
<td>12</td>
<td>4.11</td>
<td>5.010</td>
<td>5.850</td>
<td>7.040</td>
</tr>
<tr>
<td>13</td>
<td>4.66</td>
<td>5.630</td>
<td>6.570</td>
<td>7.790</td>
</tr>
<tr>
<td>14</td>
<td>5.23</td>
<td>6.260</td>
<td>7.260</td>
<td>8.550</td>
</tr>
<tr>
<td>15</td>
<td>5.81</td>
<td>6.910</td>
<td>7.960</td>
<td>9.310</td>
</tr>
<tr>
<td>16</td>
<td>6.41</td>
<td>7.560</td>
<td>8.670</td>
<td>10.100</td>
</tr>
<tr>
<td>17</td>
<td>7.01</td>
<td>8.230</td>
<td>9.350</td>
<td>10.900</td>
</tr>
<tr>
<td>18</td>
<td>7.63</td>
<td>8.910</td>
<td>10.100</td>
<td>11.700</td>
</tr>
<tr>
<td>19</td>
<td>8.26</td>
<td>9.590</td>
<td>10.800</td>
<td>12.500</td>
</tr>
<tr>
<td>20</td>
<td>8.90</td>
<td>10.300</td>
<td>11.600</td>
<td>13.200</td>
</tr>
<tr>
<td>21</td>
<td>9.54</td>
<td>11.000</td>
<td>12.300</td>
<td>14.000</td>
</tr>
<tr>
<td>22</td>
<td>10.2</td>
<td>11.700</td>
<td>13.100</td>
<td>14.800</td>
</tr>
<tr>
<td>23</td>
<td>10.9</td>
<td>12.400</td>
<td>13.800</td>
<td>15.500</td>
</tr>
<tr>
<td>24</td>
<td>11.5</td>
<td>13.100</td>
<td>14.600</td>
<td>16.300</td>
</tr>
<tr>
<td>25</td>
<td>12.2</td>
<td>13.800</td>
<td>15.400</td>
<td>17.100</td>
</tr>
<tr>
<td>26</td>
<td>12.9</td>
<td>14.600</td>
<td>16.200</td>
<td>17.900</td>
</tr>
<tr>
<td>27</td>
<td>13.6</td>
<td>15.300</td>
<td>16.900</td>
<td>18.700</td>
</tr>
<tr>
<td>28</td>
<td>14.3</td>
<td>16.000</td>
<td>17.700</td>
<td>19.500</td>
</tr>
<tr>
<td>29</td>
<td>15.0</td>
<td>16.800</td>
<td>18.500</td>
<td>20.300</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$P\{x^2 > t\} = \alpha$ where $x^2$ has $n$ degrees of freedom.

### Table C.4. Critical values for the Kolmogorov-Smirnov test

<table>
<thead>
<tr>
<th>n</th>
<th>$\alpha = .20$</th>
<th>$\alpha = .15$</th>
<th>$\alpha = .10$</th>
<th>$\alpha = .05$</th>
<th>$\alpha = .01$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.900</td>
<td>.925</td>
<td>.950</td>
<td>.975</td>
<td>.995</td>
</tr>
<tr>
<td>2</td>
<td>.684</td>
<td>.726</td>
<td>.776</td>
<td>.842</td>
<td>.929</td>
</tr>
<tr>
<td>3</td>
<td>.565</td>
<td>.597</td>
<td>.642</td>
<td>.708</td>
<td>.829</td>
</tr>
<tr>
<td>4</td>
<td>.494</td>
<td>.525</td>
<td>.564</td>
<td>.624</td>
<td>.734</td>
</tr>
<tr>
<td>5</td>
<td>.446</td>
<td>.474</td>
<td>.510</td>
<td>.563</td>
<td>.669</td>
</tr>
<tr>
<td>6</td>
<td>.410</td>
<td>.436</td>
<td>.470</td>
<td>.521</td>
<td>.618</td>
</tr>
<tr>
<td>7</td>
<td>.381</td>
<td>.405</td>
<td>.438</td>
<td>.486</td>
<td>.577</td>
</tr>
<tr>
<td>8</td>
<td>.358</td>
<td>.381</td>
<td>.411</td>
<td>.457</td>
<td>.543</td>
</tr>
<tr>
<td>9</td>
<td>.339</td>
<td>.360</td>
<td>.388</td>
<td>.432</td>
<td>.514</td>
</tr>
<tr>
<td>10</td>
<td>.322</td>
<td>.342</td>
<td>.368</td>
<td>.409</td>
<td>.486</td>
</tr>
<tr>
<td>11</td>
<td>.307</td>
<td>.326</td>
<td>.352</td>
<td>.391</td>
<td>.458</td>
</tr>
<tr>
<td>12</td>
<td>.295</td>
<td>.313</td>
<td>.338</td>
<td>.375</td>
<td>.430</td>
</tr>
<tr>
<td>13</td>
<td>.284</td>
<td>.302</td>
<td>.325</td>
<td>.361</td>
<td>.403</td>
</tr>
<tr>
<td>14</td>
<td>.274</td>
<td>.292</td>
<td>.314</td>
<td>.349</td>
<td>.418</td>
</tr>
<tr>
<td>15</td>
<td>.266</td>
<td>.283</td>
<td>.304</td>
<td>.338</td>
<td>.404</td>
</tr>
<tr>
<td>16</td>
<td>.258</td>
<td>.274</td>
<td>.295</td>
<td>.328</td>
<td>.391</td>
</tr>
<tr>
<td>17</td>
<td>.250</td>
<td>.266</td>
<td>.286</td>
<td>.318</td>
<td>.380</td>
</tr>
<tr>
<td>18</td>
<td>.244</td>
<td>.259</td>
<td>.278</td>
<td>.309</td>
<td>.370</td>
</tr>
<tr>
<td>19</td>
<td>.237</td>
<td>.252</td>
<td>.272</td>
<td>.301</td>
<td>.361</td>
</tr>
<tr>
<td>20</td>
<td>.231</td>
<td>.249</td>
<td>.264</td>
<td>.294</td>
<td>.352</td>
</tr>
<tr>
<td>21</td>
<td>.225</td>
<td>.242</td>
<td>.264</td>
<td>.287</td>
<td>.342</td>
</tr>
<tr>
<td>22</td>
<td>.219</td>
<td>.232</td>
<td>.256</td>
<td>.278</td>
<td>.333</td>
</tr>
<tr>
<td>23</td>
<td>.213</td>
<td>.226</td>
<td>.249</td>
<td>.270</td>
<td>.324</td>
</tr>
<tr>
<td>24</td>
<td>.207</td>
<td>.220</td>
<td>.243</td>
<td>.263</td>
<td>.315</td>
</tr>
<tr>
<td>25</td>
<td>.201</td>
<td>.214</td>
<td>.237</td>
<td>.256</td>
<td>.306</td>
</tr>
<tr>
<td>26</td>
<td>.195</td>
<td>.207</td>
<td>.229</td>
<td>.249</td>
<td>.297</td>
</tr>
<tr>
<td>27</td>
<td>.190</td>
<td>.201</td>
<td>.223</td>
<td>.242</td>
<td>.288</td>
</tr>
<tr>
<td>28</td>
<td>.185</td>
<td>.194</td>
<td>.216</td>
<td>.234</td>
<td>.279</td>
</tr>
<tr>
<td>29</td>
<td>.180</td>
<td>.188</td>
<td>.209</td>
<td>.227</td>
<td>.270</td>
</tr>
<tr>
<td>30</td>
<td>.175</td>
<td>.183</td>
<td>.202</td>
<td>.216</td>
<td>.260</td>
</tr>
</tbody>
</table>

Asymptotic Formula: \[ \frac{\sqrt{n}}{\sqrt{\alpha}} \]

(For $\alpha = .01$ and $\alpha = .05$, asymptotic formulas give values that are too high — by 1.5 percent for $n = 80$.)

---


# APPENDIX C. MISCELLANEOUS TABLES

## Table C.5. ASCII characters

<table>
<thead>
<tr>
<th>DEC</th>
<th>CHAR</th>
<th>DEC</th>
<th>CHAR</th>
<th>DEC</th>
<th>CHAR</th>
<th>DEC</th>
<th>CHAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>32</td>
<td>SPC</td>
<td>64</td>
<td>@</td>
<td>96</td>
<td>^</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>33</td>
<td>!</td>
<td>65</td>
<td>A</td>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>34</td>
<td>&quot;</td>
<td>66</td>
<td>B</td>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>35</td>
<td>#</td>
<td>67</td>
<td>C</td>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>36</td>
<td>$</td>
<td>68</td>
<td>D</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>37</td>
<td>%</td>
<td>69</td>
<td>E</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>38</td>
<td>&amp;</td>
<td>70</td>
<td>F</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>39</td>
<td>'</td>
<td>71</td>
<td>G</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>40</td>
<td>(</td>
<td>72</td>
<td>H</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>41</td>
<td>)</td>
<td>73</td>
<td>I</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
<td>42</td>
<td>J</td>
<td>74</td>
<td>K</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
<td>43</td>
<td>+</td>
<td>75</td>
<td>L</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
<td>44</td>
<td>,</td>
<td>76</td>
<td>M</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
<td>45</td>
<td>.</td>
<td>77</td>
<td>N</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
<td>46</td>
<td>/</td>
<td>78</td>
<td>O</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
<td>47</td>
<td>0</td>
<td>79</td>
<td>P</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
<td>48</td>
<td>1</td>
<td>80</td>
<td>Q</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>DC1</td>
<td>49</td>
<td>2</td>
<td>81</td>
<td>R</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>DC2</td>
<td>50</td>
<td>3</td>
<td>82</td>
<td>S</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>DC3</td>
<td>51</td>
<td>4</td>
<td>83</td>
<td>T</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>DC4</td>
<td>52</td>
<td>5</td>
<td>84</td>
<td>U</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>NAK</td>
<td>53</td>
<td>6</td>
<td>85</td>
<td>V</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>SYN</td>
<td>54</td>
<td>7</td>
<td>86</td>
<td>W</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>ETB</td>
<td>55</td>
<td>8</td>
<td>87</td>
<td>X</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>CAN</td>
<td>56</td>
<td>9</td>
<td>88</td>
<td>Y</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>EM</td>
<td>57</td>
<td>A</td>
<td>89</td>
<td>Z</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>SUB</td>
<td>58</td>
<td>B</td>
<td>90</td>
<td>[</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>ESC</td>
<td>59</td>
<td>C</td>
<td>91</td>
<td>]</td>
<td>123</td>
<td>]</td>
</tr>
<tr>
<td>28</td>
<td>FS</td>
<td>60</td>
<td>D</td>
<td>92</td>
<td>\</td>
<td>124</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>GS</td>
<td>61</td>
<td>E</td>
<td>93</td>
<td>]</td>
<td>125</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
<td>62</td>
<td>F</td>
<td>94</td>
<td>&quot;</td>
<td>126</td>
<td>&quot;</td>
</tr>
<tr>
<td>31</td>
<td>US</td>
<td>63</td>
<td>G</td>
<td>95</td>
<td>-</td>
<td>127</td>
<td>-</td>
</tr>
</tbody>
</table>

*The first 32 characters are special, nonprinting, characters. For example, the 10th character is a line feed and the 13th character is a carriage return.*

## index

- ABORT, 122, 259, 272 279
- Abs, 242
- active entities, 99
- ALIAS, 174, 246
- ALL, 166
- alt B key, 28, 36, 272
- alt C key, 28, 36, 272
- alt D key, 36, 233
- alt E key, 28, 36, 272
- alt H key, 36, 233
- alt M key, 36, 233
- alt P key, 28, 36, 233
- alt R key, 36, 233
- alt W key, 36, 233
- AND, 240, 241
- animation, 90
- ArcTan, 242
- ARGMAX, 244
- ARGMIN, 244
- ARRAY, 166, 249, 255
- ARRIVE, 30, 40, 233, 279
- ASCII, 27, 91, 253, 290
- assignment statements, 261
- attribute variables, 239
- ATTRIBUTES, 76, 252, 255
- Available, 74, 76, 239
- bar chart, 11
- batch means, 131
- begin (compound statements), 85, 260
- Begin (from Search), 179, 282
- Beta, 54, 243
- binary operator, 240
- Binomial, 48, 54, 243
- block statements, 29, 268
- BlockCurrent, 239
- BlockListing, 92, 258
- BlockTotal, 239
- boolean, 84, 240
- Box, G. E. P., 51
- BUNDLE, 154, 271, 279
- Capacity, 74, 76, 239
- Capacity (from Chain), 255
- Capacity (from Queue), 75, 256
- Capacity (from Resource), 74, 257
- CASE, 263
- Ceiling, 153, 242
- Cells (from Histogram), 29, 256
- central limit theorem, 120
- Cexcess (from Link), 159, 281
- CHAIN, 159, 255, 270
- chi-square, 62, 121, 288
- CHOICE (from Queue), 75
- CLEAR, 122, 139, 243, 258, 259
- CLEARSCREEN, 93, 267, 279
- ClockTime, 30, 239
- Close (from File menu), 228
- Closed (from Gate), 173, 256
- Closed (from SetGate), 174, 282
- COMBINE, 149, 271, 279
<table>
<thead>
<tr>
<th>INDEX</th>
<th>NDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>oments, 31, 237</td>
<td>editor, 26, 27, 227</td>
</tr>
<tr>
<td>compound statements, 260</td>
<td>EJECTPAGE, 93, 267</td>
</tr>
<tr>
<td>;ON (from Textfile), 93, 252, 257</td>
<td>ELSE (from Docases), 170, 264</td>
</tr>
<tr>
<td>;ON (from Search), 179, 282</td>
<td>ELSE (from If), 85, 262</td>
</tr>
<tr>
<td>;ON (from Unlink), 159, 283</td>
<td>ELSE (from Use), 245</td>
</tr>
<tr>
<td>conditional expressions, 245</td>
<td>empirical distribution function, 58</td>
</tr>
<tr>
<td>conditional statements, 262</td>
<td>end (compound statements), 85, 250</td>
</tr>
<tr>
<td>confidence interval for means, 118</td>
<td>End (from Search), 179, 282</td>
</tr>
<tr>
<td>confidence interval for proportions, 121</td>
<td>end key, 56, 233</td>
</tr>
<tr>
<td>confidence interval for variances, 121</td>
<td>ENDSTAT, 94, 277, 280</td>
</tr>
<tr>
<td>ongruential method, 33</td>
<td>entities, 30, 73, 80, 269</td>
</tr>
<tr>
<td>constants, 238</td>
<td>ENTITY, 178, 253, 255</td>
</tr>
<tr>
<td>CONTINUE, 122, 243. 258, 259</td>
<td>Entity (from Create), 185, 279</td>
</tr>
<tr>
<td>CONTINUOUS (from Distribution), 52, 251, 255</td>
<td>Entity (from Remove), 181, 281</td>
</tr>
<tr>
<td>continuious uniform, 53, 243</td>
<td>Entity (from Schedule), 185, 282</td>
</tr>
<tr>
<td>CONTROL, 25, 74, 89, 243, 245</td>
<td>Entity (from Search), 179, 282</td>
</tr>
<tr>
<td>CONTROL parameters, 258</td>
<td>entity pointer, 178</td>
</tr>
<tr>
<td>control key, 27</td>
<td>EntityNumber, 89, 102, 239</td>
</tr>
<tr>
<td>control phase, 88</td>
<td>egcdic, 129, 131</td>
</tr>
<tr>
<td>copiesTo (from Copy), 143, 279</td>
<td>Erlang, 54, 243</td>
</tr>
<tr>
<td>COPY, 148, 271, 279</td>
<td>errors, 34</td>
</tr>
<tr>
<td>correlation coefficient, 132</td>
<td>ESC, 2, 24, 27, 36, 92, 104, 233, 243, 245, 246</td>
</tr>
<tr>
<td>cor, 242</td>
<td>estimator, 21, 57, 129</td>
</tr>
<tr>
<td>covariance, 137</td>
<td>event-driven simulation, 17</td>
</tr>
<tr>
<td>CREATE, 185, 269. 279</td>
<td>EventChain, 101, 239</td>
</tr>
<tr>
<td>critical value, 61</td>
<td>EXITLOGIC, 121, 259</td>
</tr>
<tr>
<td>crl, 27</td>
<td>Exp, 242</td>
</tr>
<tr>
<td>crl E key, 36, 233</td>
<td>Expd, 54, 243</td>
</tr>
<tr>
<td>crl F key, 36, 233</td>
<td>experimental design, 8</td>
</tr>
<tr>
<td>crl F1 key, 34, 227, 231</td>
<td>exponential distribution, 50</td>
</tr>
<tr>
<td>crl L key, 36, 233</td>
<td>exponentiation, 240, 241</td>
</tr>
<tr>
<td>crl R key, 36, 233</td>
<td>expressions, 240</td>
</tr>
<tr>
<td>crl W key, 36, 233</td>
<td>F1 key, 24, 34, 36, 227, 231</td>
</tr>
<tr>
<td>SumNumber, 239</td>
<td>F2 key, 231</td>
</tr>
<tr>
<td>SumProductNT, 239</td>
<td>F3 key, 26, 231</td>
</tr>
<tr>
<td>SumProductVT, 239</td>
<td>F4 key, 28, 231</td>
</tr>
<tr>
<td>sumulative distribution function, 43</td>
<td>F5 key, 231</td>
</tr>
<tr>
<td>CumValue, 239</td>
<td>F6 key, 231</td>
</tr>
<tr>
<td>Cuniform, 53, 243</td>
<td>F7 key, 36, 231</td>
</tr>
<tr>
<td>cursor keys, 27</td>
<td>F8 key, 36, 231</td>
</tr>
<tr>
<td>decimal point, 29</td>
<td>F9 key, 24, 34, 232</td>
</tr>
<tr>
<td>declarations, 247</td>
<td>F10 key, 27, 232</td>
</tr>
<tr>
<td>Decrement (from Bundle), 154, 279</td>
<td>FIFO, 73, 106</td>
</tr>
<tr>
<td>Decrement (from Combine), 149, 279</td>
<td>FIFO (from Queue), 75</td>
</tr>
<tr>
<td>Decrement (from Gather), 156, 280</td>
<td>File menu, 35</td>
</tr>
<tr>
<td>DEFINITION, 25, 74, 89, 243</td>
<td>Filepath (from Options menu), 230</td>
</tr>
<tr>
<td>definition phase, 73</td>
<td>Floor, 242</td>
</tr>
<tr>
<td>DEFINITION structures 246, 255-257</td>
<td>FOR, 168, 266, 280</td>
</tr>
<tr>
<td>degrees of freedom, 62</td>
<td>Frac, 243</td>
</tr>
<tr>
<td>DEL key, 36, 233</td>
<td>functions, 242, 250</td>
</tr>
<tr>
<td>DEPART, 30, 41, 89, 243, 280</td>
<td>future events list, 89, 101, 178, 184, 236, 239</td>
</tr>
<tr>
<td>DepartCount, 239</td>
<td>Gamma, 54, 243</td>
</tr>
<tr>
<td>Destination (from Remove), 181, 281</td>
<td>GATE, 173, 356, 270</td>
</tr>
<tr>
<td>Destination (from Return), 282</td>
<td>GATHER, 156, 271, 280</td>
</tr>
<tr>
<td>Destination (from Schedule), 185, 282</td>
<td>global variables, 75</td>
</tr>
<tr>
<td>Destination (from Unlink), 159, 283</td>
<td>goodness-of-fit, 59</td>
</tr>
<tr>
<td>Dir (from File menu), 228</td>
<td>Gordon, G., 193</td>
</tr>
<tr>
<td>Discipline (from Chain), 255</td>
<td>GOSUB, 262, 280</td>
</tr>
<tr>
<td>Discipline (from Gate), 173, 256</td>
<td>GOTO, 84, 261, 280</td>
</tr>
<tr>
<td>Discipline (from Queue), 75, 256</td>
<td>GPSS, 193</td>
</tr>
<tr>
<td>Discrete (from Distribution), 47, 251, 255</td>
<td>GPSS/H, 193</td>
</tr>
<tr>
<td>discrete uniform, 47</td>
<td>GPSS/PC, 193</td>
</tr>
<tr>
<td>DISTRIBUTION, 46, 52, 89, 255</td>
<td>HALT, 122, 255, 272, 280</td>
</tr>
<tr>
<td>DO (from While), 167, 264</td>
<td>help system, 34, 234</td>
</tr>
<tr>
<td>DO (from For), 168, 266</td>
<td>histogram, 11, 13</td>
</tr>
<tr>
<td>DOCASES, 169, 253, 280</td>
<td>HISTOGRAM, 25, 93, 256</td>
</tr>
<tr>
<td>Dos (from File menu), 229</td>
<td>Histogram (from Chain), 255</td>
</tr>
<tr>
<td>DSE extension, 26</td>
<td>Histogram (from Gate), 173, 256</td>
</tr>
<tr>
<td>Duniform, 47, 243</td>
<td>Histogram (from Queue), 75, 256</td>
</tr>
<tr>
<td>identifiers, 238</td>
<td>Histogram (from Statistics), 94, 257</td>
</tr>
<tr>
<td>IF, 84, 262, 280</td>
<td>home key, 33, 233</td>
</tr>
<tr>
<td>INDEX</td>
<td>INDEX</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>replicates, 120, 129</td>
<td></td>
</tr>
<tr>
<td>RESET, 122, 139, 258, 259, 281</td>
<td></td>
</tr>
<tr>
<td>ResetCount, 90, 122, 258</td>
<td></td>
</tr>
<tr>
<td>ResetTime, 89, 122, 258</td>
<td></td>
</tr>
<tr>
<td>RESOURCE, 74, 257, 275</td>
<td></td>
</tr>
<tr>
<td>resource-filling rule, 101, 182</td>
<td></td>
</tr>
<tr>
<td>Resume (from Preempt), 144, 231</td>
<td></td>
</tr>
<tr>
<td>ResumeTime, 239, 253</td>
<td></td>
</tr>
<tr>
<td>RETURN, 262, 282</td>
<td></td>
</tr>
<tr>
<td>return key, 36, 233</td>
<td></td>
</tr>
<tr>
<td>Recess (from Seize), 82, 282</td>
<td></td>
</tr>
<tr>
<td>Run menu, 229</td>
<td></td>
</tr>
<tr>
<td>runs test, 67</td>
<td></td>
</tr>
<tr>
<td>sample mean, 56, 128</td>
<td></td>
</tr>
<tr>
<td>sample variance, 55, 129</td>
<td></td>
</tr>
<tr>
<td>Save (from File menu), 228</td>
<td></td>
</tr>
<tr>
<td>SaveAs (from File menu), 35, 228</td>
<td></td>
</tr>
<tr>
<td>SCHEDULE, 185, 259, 282</td>
<td></td>
</tr>
<tr>
<td>Schriber, T. J., 195</td>
<td></td>
</tr>
<tr>
<td>scientific notation, 237</td>
<td></td>
</tr>
<tr>
<td>scope, 238</td>
<td></td>
</tr>
<tr>
<td>screen, 34</td>
<td></td>
</tr>
<tr>
<td>SEARCH, 178, 179, 259, 282</td>
<td></td>
</tr>
<tr>
<td>Seed, 41, 258</td>
<td></td>
</tr>
<tr>
<td>SEIZE, 31, 275, 282</td>
<td></td>
</tr>
<tr>
<td>semicolon, 29</td>
<td></td>
</tr>
<tr>
<td>SEQUENCE, 163, 247, 257</td>
<td></td>
</tr>
<tr>
<td>SETGATE, 174, 277, 282</td>
<td></td>
</tr>
<tr>
<td>Shannon, R. E., 13</td>
<td></td>
</tr>
<tr>
<td>Show (from Options menu), 230</td>
<td></td>
</tr>
<tr>
<td>Sign, 243</td>
<td></td>
</tr>
<tr>
<td>significance level, 61</td>
<td></td>
</tr>
<tr>
<td>SIMAN, 209</td>
<td></td>
</tr>
<tr>
<td>simple statements, 260</td>
<td></td>
</tr>
<tr>
<td>Sin, 242</td>
<td></td>
</tr>
<tr>
<td>SingleStep, 91, 258</td>
<td></td>
</tr>
<tr>
<td>Sqr, 242</td>
<td></td>
</tr>
<tr>
<td>Sqrt, 58, 242</td>
<td></td>
</tr>
<tr>
<td>standard deviation, 56</td>
<td></td>
</tr>
<tr>
<td>start-up period, 130</td>
<td></td>
</tr>
<tr>
<td>START.DSE, 25, 31, 40</td>
<td></td>
</tr>
<tr>
<td>STARTSTAT, 94, 277, 283</td>
<td></td>
</tr>
<tr>
<td>StatError, 258</td>
<td></td>
</tr>
<tr>
<td>STATISTICS, 93, 257, 270</td>
<td></td>
</tr>
<tr>
<td>StatListing, 92, 258</td>
<td></td>
</tr>
<tr>
<td>StatOnOff (from Chain), 255</td>
<td></td>
</tr>
<tr>
<td>StatOnOff (from Gate), 173, 256</td>
<td></td>
</tr>
<tr>
<td>StatOnOff (from Queue), 75, 256</td>
<td></td>
</tr>
<tr>
<td>Status (from Gate), 173, 256</td>
<td></td>
</tr>
<tr>
<td>Status (from SetGate), 174, 282</td>
<td></td>
</tr>
<tr>
<td>Status (from Textfile), 93, 252, 257</td>
<td></td>
</tr>
<tr>
<td>steady-state mean, 128</td>
<td></td>
</tr>
<tr>
<td>steady-state simulation, 116, 124</td>
<td></td>
</tr>
<tr>
<td>steady-state variance, 128</td>
<td></td>
</tr>
<tr>
<td>StopCount, 90, 122, 253</td>
<td></td>
</tr>
<tr>
<td>StopTime, 41, 89, 122, 258</td>
<td></td>
</tr>
<tr>
<td>STRING, 250, 257</td>
<td></td>
</tr>
<tr>
<td>strings, 237</td>
<td></td>
</tr>
<tr>
<td>Student-t statistic, 118, 286</td>
<td></td>
</tr>
<tr>
<td>subscripted identifier, 238</td>
<td></td>
</tr>
<tr>
<td>SUM, 244</td>
<td></td>
</tr>
<tr>
<td>surrogate entity, 153</td>
<td></td>
</tr>
<tr>
<td>Surrogate (from Bundle), 154, 279</td>
<td></td>
</tr>
<tr>
<td>symbolic name, 171, 233</td>
<td></td>
</tr>
<tr>
<td>symbols, 237</td>
<td></td>
</tr>
<tr>
<td>system arrays, 239</td>
<td></td>
</tr>
<tr>
<td>system constants, 238</td>
<td></td>
</tr>
<tr>
<td>system functions, 41, 47, 53, 242</td>
<td></td>
</tr>
<tr>
<td>System menu, 229</td>
<td></td>
</tr>
<tr>
<td>system variables, 239</td>
<td></td>
</tr>
<tr>
<td>t statistic, 118, 132, 137, 286</td>
<td></td>
</tr>
<tr>
<td>Tableaux (from Options menu), 230</td>
<td></td>
</tr>
<tr>
<td>Tables (from Options menu), 230</td>
<td></td>
</tr>
<tr>
<td>Tar, 242</td>
<td></td>
</tr>
<tr>
<td>terminating simulation, 116</td>
<td></td>
</tr>
<tr>
<td>test statistic, 61</td>
<td></td>
</tr>
<tr>
<td>TESTGATE, 173, 277, 283</td>
<td></td>
</tr>
<tr>
<td>text files, 27, 34</td>
<td></td>
</tr>
<tr>
<td>TEXTFILE, 93, 252, 257</td>
<td></td>
</tr>
<tr>
<td>THEN, 84, 262</td>
<td></td>
</tr>
<tr>
<td>Time (from Arrive), 30, 40, 279</td>
<td></td>
</tr>
<tr>
<td>Time (from Schedule), 185, 282</td>
<td></td>
</tr>
<tr>
<td>Time (from Wait), 83, 283</td>
<td></td>
</tr>
<tr>
<td>Triangular, 54, 244</td>
<td></td>
</tr>
<tr>
<td>type I error, 60</td>
<td></td>
</tr>
<tr>
<td>type II error, 60</td>
<td></td>
</tr>
<tr>
<td>types, 244</td>
<td></td>
</tr>
<tr>
<td>unary operators, 240, 241</td>
<td></td>
</tr>
<tr>
<td>unbiased, 57, 129</td>
<td></td>
</tr>
<tr>
<td>UNBUNDLE, 271, 283</td>
<td></td>
</tr>
<tr>
<td>uniform distribution, 38</td>
<td></td>
</tr>
<tr>
<td>Units (from Preempt), 144, 281</td>
<td></td>
</tr>
<tr>
<td>Units (from Release), 82, 281</td>
<td></td>
</tr>
<tr>
<td>Units (from Seize), 81, 282</td>
<td></td>
</tr>
<tr>
<td>UNLINK, 159, 159, 53, 283</td>
<td></td>
</tr>
<tr>
<td>UNTIL, 168, 265</td>
<td></td>
</tr>
<tr>
<td>USE, 245</td>
<td></td>
</tr>
<tr>
<td>UseAttribute (from Bundle), 154, 279</td>
<td></td>
</tr>
<tr>
<td>UseAttribute (from Combine), 149, 279</td>
<td></td>
</tr>
<tr>
<td>UseAttribute (from Gather), 156, 280</td>
<td></td>
</tr>
<tr>
<td>Used, 74, 76, 239</td>
<td></td>
</tr>
<tr>
<td>Value (from VarStatistics), 138, 257</td>
<td></td>
</tr>
<tr>
<td>variable arrival rates, 105</td>
<td></td>
</tr>
<tr>
<td>variance, 56, 129</td>
<td></td>
</tr>
<tr>
<td>VarListing, 92, 258</td>
<td></td>
</tr>
<tr>
<td>VARSTATISTICS, 138, 257</td>
<td></td>
</tr>
<tr>
<td>verification, 6</td>
<td></td>
</tr>
<tr>
<td>View, 101</td>
<td></td>
</tr>
<tr>
<td>View option, 104</td>
<td></td>
</tr>
<tr>
<td>WAIT, 83, 145, 277, 283</td>
<td></td>
</tr>
<tr>
<td>WatchChar, 91, 239, 253</td>
<td></td>
</tr>
<tr>
<td>WatchDelay, 91, 258</td>
<td></td>
</tr>
<tr>
<td>WATCHLIST, 91, 258</td>
<td></td>
</tr>
<tr>
<td>WatchStatus, 91, 258</td>
<td></td>
</tr>
<tr>
<td>Weibull, 54, 244</td>
<td></td>
</tr>
<tr>
<td>WHILE, 53, 167, 172, 283</td>
<td></td>
</tr>
<tr>
<td>WITH, 244</td>
<td></td>
</tr>
<tr>
<td>Write (from Textfile), 93, 252, 257</td>
<td></td>
</tr>
<tr>
<td>XOR, 240, 241</td>
<td></td>
</tr>
<tr>
<td>z statistic, 68, 121, 286</td>
<td></td>
</tr>
<tr>
<td>Zeigler, B. O., 210</td>
<td></td>
</tr>
<tr>
<td>zero-time delay, 100, 101, 278</td>
<td></td>
</tr>
</tbody>
</table>