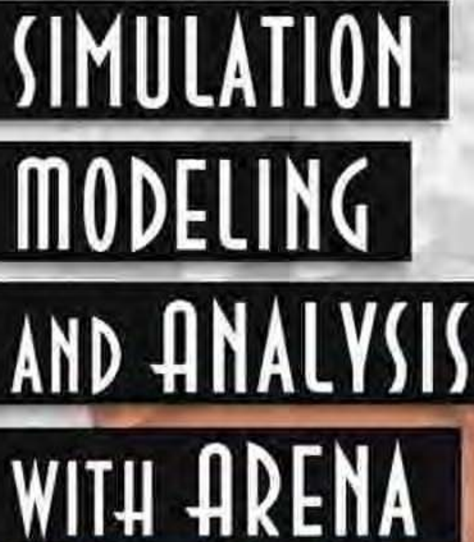


TAYFUR ALTIOK • BENJAMIN MELAMED

Simulation Modeling
and
Analysis with Arena

This page intentionally left blank

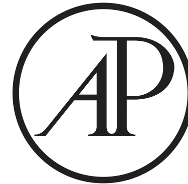
Simulation Modeling and Analysis with Arena

Tayfur Altiok
Rutgers University
Piscataway, New Jersey

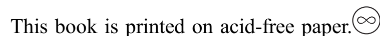
Benjamin Melamed
Rutgers University
Piscataway, New Jersey

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Academic Press is an imprint of Elsevier



Academic Press is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
525 B Street, Suite 1900, San Diego, California 92101-4495, USA
84 Theobald's Road, London WC1X 8RR, UK

This book is printed on acid-free paper. 

Copyright © 2007, Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request on-line via the Elsevier homepage (http://elsevier.com), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Application submitted.

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN 13: 978-0-12-370523-5
ISBN 10: 0-12-370523-1

For information on all Academic Press publications
visit our Web site at www.books.elsevier.com

Printed in the United States of America
07 08 09 10 9 8 7 6 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER BOOK AID International Sabre Foundation

Contents

Preface xvii
Acknowledgments xxi

Chapter 1
Introduction to Simulation Modeling

1.1 Systems and Models 1
1.2 Analytical Versus Simulation Modeling 2
1.3 Simulation Modeling and Analysis 4
1.4 Simulation Worldviews 4
1.5 Model Building 5
1.6 Simulation Costs and Risks 6
1.7 Example: A Production Control Problem 7
1.8 Project Report 8
Exercises 10

Chapter 2
Discrete Event Simulation

2.1 Elements of Discrete Event Simulation 11
2.2 Examples of DES Models 13
2.2.1 Single Machine 13

2.2.2 Single Machine with Failures 13
2.2.3 Single Machine with an Inspection Station and Associated Inventory 14

2.3 Monte Carlo Sampling and Histories 15
2.3.1 Example: Work Station Subject to Failures and Inventory Control 16

2.4 DES Languages 19

Exercises 20

Chapter 3
Elements of Probability and Statistics

3.1 Elementary Probability Theory 24
3.1.1 Probability Spaces 25
3.1.2 Conditional Probabilities 25
3.1.3 Dependence and Independence 26

3.2 Random Variables 27

3.3 Distribution Functions 27
3.3.1 Probability Mass Functions 28
3.3.2 Cumulative Distribution Functions 28
3.3.3 Probability Density Functions 28
3.3.4 Joint Distributions 29

3.4 Expectations 30

3.5 Moments 30

3.6 Correlations 32

3.7 Common Discrete Distributions 33
3.7.1 Generic Discrete Distribution 33
3.7.2 Bernoulli Distribution 34
3.7.3 Binomial Distribution 34
3.7.4 Geometric Distribution 35
3.7.5 Poisson Distribution 35

3.8 Common Continuous Distributions 36
3.8.1 Uniform Distribution 36
3.8.2 Step Distribution 37
3.8.3 Triangular Distribution 38
3.8.4 Exponential Distribution 39
3.8.5 Normal Distribution 40

3.8.6 Lognormal Distribution 41
3.8.7 Gamma Distribution 42
3.8.8 Student´s t Distribution 44
3.8.9 F Distribution 45
3.8.10 Beta Distribution 46
3.8.11 Weibull Distribution 47

3.9 Stochastic Processes 47
3.9.1 Iid Processes 48
3.9.2 Poisson Processes 48
3.9.3 Regenerative (Renewal) Processes 49
3.9.4 Markov Processes 49

3.10 Estimation 50

3.11 Hypothesis Testing 51

Exercises 52

Chapter 4
Random Number and Variate Generation

4.1 Variate and Process Generation 56

4.2 Variate Generation Using the Inverse Transform Method 57
4.2.1 Generation of Uniform Variates 58
4.2.2 Generation of Exponential Variates 58
4.2.3 Generation of Discrete Variates 59
4.2.4 Generation of Step Variates from Histograms 60

4.3 Process Generation 61
4.3.1 Iid Process Generation 61
4.3.2 Non-Iid Process Generation 61

Exercises 63

Chapter 5
Arena Basics

5.1 Arena Home Screen 66
5.1.1 *Menu* Bar 67
5.1.2 *Project* Bar 67
5.1.3 *Standard* Toolbar 68
5.1.4 *Draw* and *View* Bars 68
5.1.5 *Animate* and *Animate Transfer* Bars 68
5.1.6 *Run Interaction* Bar 69

5.1.7 *Integration* Bar 69
5.1.8 *Debug* Bar 69

5.2 Example: A Simple Workstation 69

5.3 Arena Data Storage Objects 74
5.3.1 Variables 75
5.3.2 Expressions 75
5.3.3 Attributes 75

5.4 Arena Output Statistics Collection 75
5.4.1 Statistics Collection via the *Statistic* Module 76
5.4.2 Statistics Collection via the *Record* Module 76

5.5 Arena Simulation and Output Reports 77

5.6 Example: Two Processes in Series 78

5.7 Example: A Hospital Emergency Room 84
5.7.1 Problem Statement 84
5.7.2 Arena Model 85
5.7.3 Emergency Room Segment 86
5.7.4 On-Call Doctor Segment 93
5.7.5 Statistics Collection 96
5.7.6 Simulation Output 97

5.8 Specifying Time-Dependent Parameters via a Schedule 100

Exercises 103

Chapter 6
Model Testing and Debugging Facilities

6.1 Facilities for Model Construction 107

6.2 Facilities for Model Checking 110

6.3 Facilities for Model Run Control 111
6.3.1 Run Modes 111
6.3.2 Mouse-Based Run Control 111
6.3.3 Keyboard-Based Run Control 112

6.4 Examples of Run Tracing 114
6.4.1 Example: Open-Ended Tracing 114
6.4.2 Example: Tracing Selected Blocks 116
6.4.3 Example: Tracing Selected Entities 117

6.5 Visualization and Animation 118
6.5.1 *Animate Connectors* Button 118
6.5.2 *Animate* Toolbar 118
6.5.3 *Animate Transfer* Toolbar 119

6.6 Arena Help Facilities 119
6.6.1 *Help* Menu 120
6.6.2 *Help* Button 120

Exercises 120

Chapter 7
Input Analysis

7.1 Data Collection 124

7.2 Data Analysis 125

7.3 Modeling Time Series Data 127
7.3.1 Method of Moments 128
7.3.2 Maximal Likelihood Estimation Method 129

7.4 Arena *Input Analyzer* 130

7.5 Goodness-of-Fit Tests for Distributions 134
7.5.1 Chi-Square Test 134
7.5.2 Kolmogorov-Smirnov (K-S) Test 137

7.6 Multimodal Distributions 137

Exercises 138

Chapter 8
Model Goodness: Verification and Validation

8.1 Model Verification via Inspection of Test Runs 142
8.1.1 Input Parameters and Output Statistics 142
8.1.2 Using a Debugger 143
8.1.3 Using Animation 143
8.1.4 Sanity Checks 143

8.2 Model Verification via Performance Analysis 143
8.2.1 Generic Workstation as a Queueing System 143
8.2.2 Queueing Processes and Parameters 144
8.2.3 Service Disciplines 145
8.2.4 Queueing Performance Measures 145

8.2.5 Regenerative Queueing Systems and Busy Cycles 146
8.2.6 Throughput 147
8.2.7 Little´s Formula 148
8.2.8 Steady-State Flow Conservation 148
8.2.9 PASTA Property 149

8.3 Examples of Model Verification 149
8.3.1 Model Verification in a Single Workstation 149
8.3.2 Model Verification in Tandem Workstations 153

8.4 Model Validation 161

Exercises 162

Chapter 9
Output Analysis

9.1 Terminating and Steady-State Simulation Models 166
9.1.1 Terminating Simulation Models 166
9.1.2 Steady-State Simulation Models 166

9.2 Statistics Collection from Replications 168
9.2.1 Statistics Collection Using Independent Replications 169
9.2.2 Statistics Collection Using Regeneration Points and Batch Means 170

9.3 Point Estimation 171
9.3.1 Point Estimation from Replications 171
9.3.2 Point Estimation in Arena 172

9.4 Confidence Interval Estimation 173
9.4.1 Confidence Intervals for Terminating Simulations 173
9.4.2 Confidence Intervals for Steady-State Simulations 176
9.4.3 Confidence Interval Estimation in Arena 176

9.5 Output Analysis via Standard Arena Output 177
9.5.1 Working Example: A Workstation with Two Types of Parts 177
9.5.2 Observation Collection 179
9.5.3 Output Summary 180
9.5.4 Statistics Summary: Multiple Replications 181

9.6 Output Analysis via the Arena *Output Analyzer* 182
9.6.1 Data Collection 183
9.6.2 Graphical Statistics 184
9.6.3 Batching Data for Independent Observations 185

9.6.4 Confidence Intervals for Means and Variances 186
9.6.5 Comparing Means and Variances 187
9.6.6 Point Estimates for Correlations 189

9.7 Parametric Analysis via the Arena *Process Analyzer* 190

Exercises 193

Chapter 10
Correlation Analysis

10.1 Correlation in Input Analysis 195

10.2 Correlation in Output Analysis 197

10.3 Autocorrelation Modeling with TES Processes 199

10.4 Introduction to TES Modeling 200
10.4.1 Background TES Processes 202
10.4.2 Foreground TES Processes 205
10.4.3 Inversion of Distribution Functions 211

10.5 Generation of TES Sequences 215
Generation of TES^+ Sequences 215
Generation of TES^- Sequences 216
Combining TES Generation Algorithms 216

10.6 Example: Correlation Analysis in Manufacturing Systems 219

Exercises 220

Chapter 11
Modeling Production Lines

11.1 Production Lines 223

11.2 Models of Production Lines 225

11.3 Example: A Packaging Line 225
11.3.1 An Arena Model 226
11.3.2 Manufacturing Process Modules 226
11.3.3 Model Blocking Using the *Hold* Module 227
11.3.4 Resources and Queues 229
11.3.5 Statistics Collection 230
11.3.6 Simulation Output Reports 231

11.4 Understanding System Behavior and Model Verification 237

11.5 Modeling Production Lines via Indexed Queues and Resources 239

11.6 An Alternative Method of Modeling Blocking 246

11.7 Modeling Machine Failures 247

11.8 Estimating Distributions of Sojourn Times 251

11.9 Batch Processing 253

11.10 Assembly Operations 256

11.11 Model Verification for Production Lines 258

Exercises 259

Chapter 12
Modeling Supply Chain Systems

12.1 Example: A Production/Inventory System 265
12.1.1 Problem Statement 265
12.1.2 Arena Model 266
12.1.3 Inventory Management Segment 267
12.1.4 Demand Management Segment 270
12.1.5 Statistics Collection 272
12.1.6 Simulation Output 273
12.1.7 Experimentation and Analysis 274

12.2 Example: A Multiproduct Production/Inventory System 276
12.2.1 Problem Statement 276
12.2.2 Arena Model 278
12.2.3 Inventory Management Segment 278
12.2.4 Demand Management Segment 284
12.2.5 Model Input Parameters and Statistics 290
12.2.6 Simulation Results 292

12.3 Example: A Multiechelon Supply Chain 293
12.3.1 Problem Statement 293
12.3.2 Arena Model 295
12.3.3 Inventory Management Segment for Retailer 295
12.3.4 Inventory Management Segment for Distribution Center 297

12.3.5 Inventory Management Segment for Output Buffer 299
12.3.6 Production/Inventory Management Segment for Input Buffer 303
12.3.7 Inventory Management Segment for Supplier 305
12.3.8 Statistics Collection 305
12.3.9 Simulation Results 306

Exercises 306

Chapter 13
Modeling Transportation Systems

13.1 *Advanced Transfer* Template Panel 314

13.2 *Animate Transfer* Toolbar 315

13.3 Example: A Bulk-Material Port 316
13.3.1 Ship Arrivals 317
13.3.2 Tug Boat Operations 320
13.3.3 Coal-Loading Operations 324
13.3.4 Tidal Window Modulation 328
13.3.5 Simulation Results 330

13.4 Example: A Toll Plaza 332
13.4.1 Arrivals Generation 334
13.4.2 Dispatching Cars to Tollbooths 336
13.4.3 Serving Cars at Tollbooths 340
13.4.4 Simulation Results for the Toll Plaza Model 344

13.5 Example: A Gear Manufacturing Job Shop 346
13.5.1 Gear Job Arrivals 349
13.5.2 Gear Transportation 351
13.5.3 Gear Processing 353
13.5.4 Simulation Results for the Gear Manufacturing Job Shop Model 358

13.6 Example: Sets Version of the Gear Manufacturing Job Shop Model 359

Exercises 365

Chapter 14
Modeling Computer Information Systems

14.1 Client/Server System Architectures 371
14.1.1 Message-Based Communications 372

14.1.2 Client Hosts 372
14.1.3 Server Hosts 373

14.2 Communications Networks 374

14.3 Two-Tier Client/Server Example: A Human Resources System 375
14.3.1 Client Nodes Segment 378
14.3.2 Communications Network Segment 378
14.3.3 Server Node Segment 380
14.3.4 Simulation Results 383

14.4 Three-Tier Client/Server Example: An Online Bookseller System 384
14.4.1 Request Arrivals and Transmission Network Segment 386
14.4.2 Transmission Network Segment 388
14.4.3 Server Nodes Segment 391
14.4.4 Simulation Results 399

Exercises 400

Appendix A
Frequently Used Arena Constructs

A.1 Frequently Used Arena Built-in Variables 405
A.1.1 Entity-Related Attributes and Variables 405
A.1.2 Simulation Time Variables 406
A.1.3 Expressions 406
A.1.4 General-Purpose Global Variables 406
A.1.5 Queue Variables 406
A.1.6 Resource Variables 406
A.1.7 Statistics Collection Variables 406
A.1.8 Transporter Variables 407
A.1.9 Miscellaneous Variables and Functions 407

A.2 Frequently Used Arena Modules 407
A.2.1 *Access* Module (*Advanced Transfer*) 407
A.2.2 *Assign* Module (*Basic Process*) 408
A.2.3 *Batch* Module (*Basic Process*) 408
A.2.4 *Create* Module (*Basic Process*) 408
A.2.5 *Decide* Module (*Basic Process*) 408
A.2.6 *Delay* Module (*Advanced Process*) 408
A.2.7 *Dispose* Module (*Basic Process*) 409
A.2.8 *Dropoff* Module (*Advanced Process*) 409
A.2.9 *Free* Module (*Advanced Transfer*) 409
A.2.10 *Halt* Module (*Advanced Transfer*) 409

A.2.11 *Hold* Module (*Advanced Process*) 410
A.2.12 *Match* Module (*Advanced Process*) 410
A.2.13 *PickStation* Module (*Advanced Transfer*) 410
A.2.14 *Pickup* Module (*Advanced Process*) 410
A.2.15 *Process* Module (*Basic Process*) 410
A.2.16 *ReadWrite* Module (*Advanced Process*) 411
A.2.17 *Record* Module (*Basic Process*) 411
A.2.18 *Release* Module (*Advanced Process*) 411
A.2.19 *Remove* Module (*Advanced Process*) 411
A.2.20 *Request* Module (*Advanced Transfer*) 411
A.2.21 *Route* Module (*Advanced Transfer*) 412
A.2.22 *Search* Module (*Advanced Process*) 412
A.2.23 *Seize* Module (*Advanced Process*) 412
A.2.24 *Separate* Module (*Basic Process*) 412
A.2.25 *Signal* Module (*Advanced Process*) 413
A.2.26 *Station* Module (*Advanced Transfer*) 413
A.2.27 *Store* Module (*Advanced Process*) 413
A.2.28 *Transport* Module (*Advanced Transfer*) 413
A.2.29 *Unstore* Module (*Advanced Process*) 413
A.2.30 *VBA* Block (*Blocks*) 414

Appendix B
VBA in Arena

B.1 Arena's Object Model 416

B.2 Arena's Type Library 416
B.2.1 Resolving Object Name Ambiguities 417
B.2.2 Obtaining Access to the *Application* Object 417

B.3 Arena VBA Events 417

B.4 Example: Using VBA in Arena 419
B.4.1 Changing Inventory Parameters Just Before a Simulation Run 419
B.4.2 Changing Inventory Parameters during a Simulation Run 421
B.4.3 Changing Customer Arrival Distributions Just before a Simulation Run 422
B.4.3 Writing Arena Data to Excel via VBA Code 424
B.4.4 Reading Arena Data from Excel via VBA Code 428

References 431
Index 435

This page intentionally left blank

Preface

Monte Carlo simulation (simulation, for short) is a powerful tool for modeling and analysis of complex systems. The vast majority of real-life systems are difficult or impossible to study via analytical models due to the paucity or lack of practically computable solution (closed-form or numerical). In contrast, a simulation model can almost always be constructed and run to generate system histories that yield useful statistical information on system operation and performance measures. Simulation helps the analyst understand how well a system performs under a given regime or a set of parameters. It can also be used iteratively in optimization studies to find the best or acceptable values of parameters, mainly for offline design problems. Indeed, the scope of simulation is now extraordinarily broad, including manufacturing environments (semiconductor, pharmaceutical, among others), supply chains (production/inventory systems, distribution networks), transportation systems (highways, airports, railways, and seaports), computer information systems (client/server systems, telecommunications networks), and others. Simulation modeling is used extensively in industry as a decision-support tool in numerous industrial problems, including estimation of facility capacities, testing for alternative methods of operation, product mix decisions, and alternative system architectures. Almost every major engineering project in the past 30 years benefited from some type of simulation modeling and analysis. Some notable examples include the trans-Alaska natural gas pipeline project, the British channel tunnel (Chunnel) project, and operations scheduling for the Suez Canal. Simulation will doubtless continue to play a major role in performance analysis studies of complex systems for a long time to come.

Until the 1980s, simulation was quite costly and time consuming in terms of both analyst time and computing resources (memory and run time). The advent of inexpensive personal computers, with powerful processors and graphics, has ushered in new capabilities that rendered simulation a particularly attractive and cost-effective approach to performance analysis of a vast variety of systems. Simulation users can now construct and test simulation models interactively, and take advantage of extensive visualization and animation features. The programming of simulation models has been simplified in a paradigm that combines visual programming (charts) and textual programming (statements) and debugging capabilities (e.g., dynamic information display and animation of simulation objects). Finally, input and output analysis capabilities make it easier to model and analyze complex systems, while attractive output reports remove the tedium of programming simulation results.

Arena is a general-purpose visual simulation environment that has evolved over many years and many versions. It first appeared as the block-oriented SIMAN simulation language, and was later enhanced by the addition of many functional modules, full visualization of model structure and parameters, improved input and output analysis tools, run control and animation facilities, and output reporting. Arena has been widely used in both industry and academia; this book on simulation modeling and analysis uses Arena as the working simulation environment. A training version of Arena is enclosed with the book, and all book examples and exercises were designed to fit its vendor-imposed limitations.

This work is planned as a textbook for an undergraduate simulation course or a graduate simulation course at an introductory level. It aims to combine both theoretical and practical aspects of Monte Carlo simulation in general, as well as the workings of the Arena simulation environment. However, the book is not structured as a user manual for Arena, and we strongly recommend that readers consult the Arena help facilities for additional details on Arena constructs. Accordingly, the book is composed of four parts, as follows:

PART I

Chapters 1 to 4 lay the foundations by first reviewing system—theoretic aspects of simulation, followed by its probabilistic and statistical underpinnings, including random number generation.

Chapter 1 is an introduction to simulation describing the philosophy, trade-offs, and conceptual stages of the simulation modeling enterprise.

Chapter 2 reviews system theoretic concepts that underlie simulation, mainly discrete event simulation (DES) and the associated concepts of state, events, simulation clock, and event list. The random elements of simulation are introduced in a detailed example illustrating sample histories and statistics.

Chapter 3 is a compendium of information on the elements of probability, statistics, and stochastic processes that are relevant to simulation modeling.

Chapter 4 is a brief review of practical random number and random variate generation. It also briefly discusses generation of stochastic processes, such as Markov processes, but defers the discussion of generating more versatile autocorrelated stochastic sequences to the more advanced Chapter 10.

PART II

Chapters 5 and 6 introduce Arena basics and its facilities, and illustrate them in simple examples.

Chapter 5 introduces basic elements of Arena, such as its graphical screen objects, modules, entities, storage objects (attributes, variables, and expressions), statistics collection, and output reporting. These concepts are introduced in examples arranged in increasing complexity.

Chapter 6 is a brief review of the Arena testing and debugging facilities, including run control and interaction, as well as debugging, which are illustrated in detailed examples.

PART III

Chapters 7 to 10 address simulation-related theory (input analysis, validation, output analysis, and correlation analysis). They further illustrate concepts and their application using Arena examples of moderate complexity.

Chapter 7 treats input analysis (mainly distribution fitting) and the corresponding *Input Analyzer* tool of Arena.

Chapter 8 discusses operational model verification via model inspection, and theory-based verification via performance analysis, using queueing theory (throughput, Little´s formula, and flow conservation principles). Model validation is also briefly discussed.

Chapter 9 treats output analysis (mainly replication design, estimation, and experimentation for both terminating and steady-state simulations) and the corresponding Arena *Output Analyzer* and *Process Analyzer* tools.

Chapter 10 introduces a new notion of correlation analysis that straddles input analysis and output analysis of autocorrelated stochastic processes. It discusses modeling and generation of autocorrelated sequences, and points out the negative consequences of ignoring temporal dependence in empirical data.

PART IV

Chapters 11 to 14 are applications oriented. They describe modeling of industrial applications in production, transportation, and information systems. These are illustrated in more elaborate Arena models that introduce advanced Arena constructs.

Chapter 11 addresses production lines, including finite buffers, machine failures, batch processing, and assembly operations.

Chapter 12 addresses supply chains, including production/inventory systems, multiproduct systems, and multiechelon systems.

Chapter 13 addresses transportation systems, including tollbooth operations, port operations, and transportation activities on the manufacturing shop floor.

Chapter 14 addresses computer information systems, including *n*-tier client/server systems and associated transmission networks.

PART V

The book includes two appendices that provide Arena programming information.

Appendix A provides condensed information on frequently used Arena programming constructs.

Appendix B contains a condensed introduction to VBA (Visual Basic for Applications) programming in Arena.

In using this work as a textbook for an undergraduate simulation course, we recommend sequencing the instruction as follows: Chapters 1 and 2, brief review of Chapters 3 and 4, and then Chapters 5, 6, 7, 11, 8, 9, 12, 13, and 14 in this order. We suggest covering Chapter 11 earlier in order to alternate the theoretical and practical portions of the book. Chapter 10 contains relatively advanced material, which is probably beyond the scope of most undergraduate classes. We recommend that it be covered after Chapter 9, and the revised instruction sequence can then be used in an advanced undergraduate or graduate course. In the latter case, Chapter 3 may be reviewed in detail, and Chapters 8 and 10 should be covered in some detail, since graduate students would be better prepared for this material. We strongly recommend that book exercises be assigned, since most of them are based on real-life systems and scenarios.

Acknowledgments

We gratefully acknowledge the help of numerous individuals. We are indebted to Randy Sadowski, Rockwell Software, for supporting this effort. We thank our students and former students Baris Balcioglu, Pooya Farahvash, Cigdem Gurgur, Abdullah Karaman, Unsal Ozdogru, Mustafa Rawat, Ozgecan Uluscu, and Wei Xiong for their help and patience. We further thank Mesut Gunduc of Microsoft Corporation, Andrew T. Zador of ATZ Consultants, and Joe Pirozzi of Soros Associates for introducing us to a world of challenging problems in computer information systems and transportation systems. We also thank Dr. David L. Jagerman for his longtime collaboration on mathematical issues, and the National Science Foundation for support in the areas of performance analysis of manufacturing and transportation systems, and correlation analysis.

Last but not least we are grateful to our respective spouses, Binnur Altiok and Shulamit Melamed, for their support and understanding of our hectic schedule in the course of writing this book.

Tayfur Altiok
Benjamin Melamed
June 2007

Chapter 1

Introduction to Simulation Modeling

Simulation modeling is a common paradigm for analyzing complex systems. In a nutshell, this paradigm creates a simplified representation of a system under study. The paradigm then proceeds to experiment with the system, guided by a prescribed set of goals, such as improved system design, cost–benefit analysis, sensitivity to design parameters, and so on. Experimentation consists of generating system histories and observing system behavior over time, as well as its statistics. Thus, the representation created (see Section 1.1) describes system structure, while the histories generated describe system behavior (see Section 1.5).

This book is concerned with simulation modeling of industrial systems. Included are manufacturing systems (e.g., production lines, inventory systems, job shops, etc.), supply chains, computer and communications systems (e.g., client-server systems, communications networks, etc.), and transportation systems (e.g., seaports, airports, etc.). The book addresses both theoretical topics and practical ones related to simulation modeling. Throughout the book, the Arena/SIMAN (see Kelton *et al*. 2000) simulation tool will be surveyed and used in hands-on examples of simulation modeling.

This chapter overviews the elements of simulation modeling and introduces basic concepts germane to such modeling.

1.1 SYSTEMS AND MODELS

Modeling is the enterprise of devising a simplified representation of a complex system with the goal of providing predictions of the system's performance measures (metrics) of interest. Such a simplified representation is called a *model*. A model is designed to capture certain behavioral aspects of the modeled system—those that are of interest to the *analyst/modeler*—in order to gain knowledge and insight into the system's behavior (Morris 1967).

Modeling calls for abstraction and simplification. In fact, if every facet of the system under study were to be reproduced in minute detail, then the model cost may approach that of the modeled system, thereby militating against creating a model in the first place.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

The modeler would simply use the "real" system or build an experimental one if it does not yet exist—an expensive and tedious proposition. Models are typically built precisely to avoid this unpalatable option. More specifically, while modeling is ultimately motivated by economic considerations, several motivational strands may be discerned:

- *Evaluating system performance under ordinary and unusual scenarios*. A model may be a necessity if the routine operation of the real-life system under study cannot be disrupted without severe consequences (e.g., attempting an upgrade of a production line in the midst of filling customer orders with tight deadlines). In other cases, the extreme scenario modeled is to be avoided at all costs (e.g., think of modeling a crash-avoiding maneuver of manned aircraft, or core meltdown in a nuclear reactor).
- *Predicting the performance of experimental system designs*. When the underlying system does not yet exist, model construction (and manipulation) is far cheaper (and safer) than building the real-life system or even its prototype. Horror stories appear periodically in the media on projects that were rushed to the implementation phase, without proper verification that their design is adequate, only to discover that the system was flawed to one degree or another (recall the case of the brand new airport with faulty luggage transport).
- *Ranking multiple designs and analyzing their tradeoffs*. This case is related to the previous one, except that the economic motivation is even greater. It often arises when the requisition of an expensive system (with detailed specifications) is awarded to the bidder with the best cost–benefit metrics.

Models can assume a variety of forms:

- A *physical model* is a simplified or scaled-down physical object (e.g., scale model of an airplane).
- A *mathematical* or *analytical model* is a set of equations or relations among mathematical variables (e.g., a set of equations describing the workflow on a factory floor).
- A *computer model* is just a program description of the system. A computer model with random elements and an underlying timeline is called a *Monte Carlo simulation model* (e.g., the operation of a manufacturing process over a period of time).

Monte Carlo simulation, or *simulation* for short, is the subject matter of this book. We shall be primarily concerned with simulation models of production, transportation, and computer information systems. Examples include production lines, inventory systems, tollbooths, port operations, and database systems.

1.2 ANALYTICAL VERSUS SIMULATION MODELING

A simulation model is implemented in a computer program. It is generally a relatively inexpensive modeling approach, commonly used as an alternative to analytical modeling. The tradeoff between analytical and simulation modeling lies in the nature of their "solutions," that is, the computation of their performance measures as follows:

1. An analytical model calls for the solution of a mathematical problem, and the derivation of mathematical formulas, or more generally, algorithmic procedures. The solution is then used to obtain performance measures of interest.

2. A simulation model calls for running (executing) a simulation program to produce sample histories. A set of s*tatistics* computed from these histories is then used to form performance measures of interest.

To compare and contrast both approaches, suppose that a production line is conceptually modeled as a queuing system. The analytical approach would create an analytical queuing system (represented by a set of equations) and proceed to solve them. The simulation approach would create a computer representation of the queuing system and run it to produce a sufficient number of sample histories. Performance measures, such as average work in the system, distribution of waiting times, and so on, would be constructed from the corresponding "solutions" as mathematical or simulation statistics, respectively.

The choice of an analytical approach versus simulation is governed by general tradeoffs. For instance, an analytical model is *preferable* to a simulation model when it has a solution, since its computation is normally much faster than that of its simulation-model counterpart. Unfortunately, complex systems rarely lend themselves to modeling via sufficiently detailed analytical models. Occasionally, though rarely, the numerical computation of an analytical solution is actually slower than a corresponding simulation. In the majority of cases, an analytical model with a tractable solution is unknown, and the modeler resorts to simulation.

When the underlying system is complex, a simulation model is normally *preferable*, for several reasons. First, in the unlikely event that an analytical model can be found, the modeler's time spent in deriving a solution may be excessive. Second, the modeler may judge that an attempt at an analytical solution is a poor bet, due to the apparent mathematical difficulties. Finally, the modeler may not even be able to formulate an analytical model with sufficient power to capture the system's behavioral aspects of interest. In contrast, simulation modeling can capture virtually any system, subject to any set of assumptions. It also enjoys the advantage of dispensing with the labor attendant to finding analytical solutions, since the modeler merely needs to construct and run a simulation program. Occasionally, however, the effort involved in constructing an elaborate simulation model is prohibitive in terms of human effort, or running the resultant program is prohibitive in terms of computer resources (CPU time and memory). In such cases, the modeler must settle for a simpler simulation model, or even an inferior analytical model.

Another way to contrast analytical and simulation models is via the classification of models into *descriptive* or *prescriptive* models. Descriptive models produce estimates for a set of performance measures corresponding to a specific set of input data. Simulation models are clearly descriptive and in this sense serve as performance analysis models. Prescriptive models are naturally geared toward design or optimization (seeking the optimal argument values of a prescribed objective function, subject to a set of constraints). Analytical models are prescriptive, whereas simulation is not. More specifically, analytical methods can serve as effective optimization tools, whereas simulation-based optimization usually calls for an exhaustive search for the optimum.

Overall, the versatility of simulation models and the feasibility of their solutions far outstrip those of analytical models. This ability to serve as an in vitro lab, in which competing system designs may be compared and contrasted and extreme-scenario performance may be safely evaluated, renders simulation modeling a highly practical tool that is widely employed by engineers in a broad range of application areas.

In particular, the complexity of industrial and service systems often forces the issue of selecting simulation as the modeling methodology of choice.

1.3 SIMULATION MODELING AND ANALYSIS

The advent of computers has greatly extended the applicability of practical simulation modeling. Since World War II, simulation has become an indispensable tool in many system-related activities. Simulation modeling has been applied to estimate performance metrics, to answer "what if" questions, and more recently, to train workers in the use of new systems. Examples follow.

- Estimating a set of productivity measures in production systems, inventory systems, manufacturing processes, materials handling, and logistics operations
- Designing and planning the capacity of computer systems and communication networks so as to minimize response times
- Conducting war games to train military personnel or to evaluate the efficacy of proposed military operations
- Evaluating and improving maritime port operations, such as container ports or bulk-material marine terminals (coal, oil, or minerals), aimed at finding ways of reducing vessel port times
- Improving health care operations, financial and banking operations, and transportation systems and airports, among many others

In addition, simulation is now used by a variety of technology workers, ranging from design engineers to plant operators and project managers. In particular, manufacturing-related activities as well as business process reengineering activities employ simulation to select design parameters, plan factory floor layout and equipment purchases, and even evaluate financial costs and return on investment (e.g., retooling, new installations, new products, and capital investment projects).

1.4 SIMULATION WORLDVIEWS

A *worldview* is a philosophy or paradigm. Every computer tool has two associated worldviews: a *developer worldview* and a *user worldview*. These two worldviews should be carefully distinguished. The first worldview pertains to the philosophy adopted by the creators of the simulation software tool (in our case, software designers and engineers). The second worldview pertains to the way the system is employed as a tool by end-users (in our case, analysts who create simulation models as code written in some *simulation language*). A system worldview may or may not coincide with an end-user worldview, but the latter includes the former.

The majority of modern computer simulation tools implement a system worldview, called the *discrete-event simulation* paradigm (see Khoshnevis 1994, Banks *et al*. 1999, Evans and Olsen 1998, and Law and Kelton 2000). In this system worldview, the simulation model possesses a *state* at any point in time. The state trajectory over time is abstracted as a piecewise-constant function, whose jumps (discontinuities) are triggered by discrete *events*. More simply, the simulation state remains unchanged unless a

simulation event occurs, at which point the model undergoes a state transition. The model evolution is governed by a *clock* and a chronologically ordered *event list*. Each event is implemented as a procedure (computer code) whose execution can change state variables and possibly schedule other events. A simulation run is started by placing an initial event in the event list, proceeds as an infinite loop that executes the current most imminent event (the one at the head of the event list), and ends when an event stops or the event list becomes empty. This beguilingly simple paradigm is extremely general and astonishingly versatile.

Early simulation languages employed a user worldview that coincided with the discrete-event paradigm. A more convenient, but more specialized, paradigm is the *transaction-driven* paradigm (commonly referred to as *process orientation*). In this popular paradigm, there are two kinds of entities: *transactions* and *resources*. A resource is a service-providing entity, typically stationary in space (e.g., a machine on the factory floor). A transaction is a mobile entity that moves among "geographical" locations (nodes). A transaction may experience delays while waiting for a resource due to contention (e.g., a product that moves among machines in the course of assembly). Transactions typically go through a life cycle: they get created, spend time at various locations, contend for resources, and eventually depart from the system. The computer code describing a transaction's life cycle is called a *process*.

Queuing elements figure prominently in this paradigm, since facilities typically contain resources and queues. Accordingly, performance measures of interest include statistics of delays in queues, the number of transactions in queues, utilization, uptimes and downtimes of resources subject to failure, and lost demand, among many others.

1.5 MODEL BUILDING

Modeling, including simulation modeling, is a complicated activity that combines art and science. Nevertheless, from a high-level standpoint, one can distinguish the following major steps:

1. *Problem analysis and information collection*. The first step in building a simulation model is to analyze the problem itself. Note that system modeling is rarely undertaken for its own sake. Rather, modeling is prompted by some system-oriented problem whose solution is the mission of the underlying project. In order to facilitate a solution, the analyst first gathers structural information that bears on the problem, and represents it conveniently. This activity includes the identification of input parameters, performance measures of interest, relationships among parameters and variables, rules governing the operation of system components, and so on. The information is then represented as logic flow diagrams, hierarchy trees, narrative, or any other convenient means of representation. Once sufficient information on the underlying system is gathered, the problem can be analyzed and a solution mapped out.
2. *Data collection*. Data collection is needed for estimating model input parameters. The analyst can formulate assumptions on the distributions of random variables in the model. When data are lacking, it may still be possible to designate parameter ranges, and simulate the model for all or some input parameters in those ranges. As will be discussed in item 5, data collection is also needed for

model validation. That is, data collected on system output statistics are compared to their model counterparts (predictions).

3. *Model construction*. Once the problem is fully studied and the requisite data collected, the analyst can proceed to construct a model and implement it as a computer program. The computer language employed may be a general-purpose language (e.g., C++, Visual Basic, FORTRAN) or a special-purpose simulation language or environment (e.g., Arena, Promodel, GPSS). See Section 2.4 for details.
4. *Model verification*. The purpose of model verification is to make sure that the model is correctly constructed. Differently stated, verification makes sure that the model conforms to its specification and does what it is supposed to do. Model verification is conducted largely by inspection, and consists of comparing model code to model specification. Any discrepancies found are reconciled by modifying either the code or the specification.
5. *Model validation*. Every model should be initially viewed as a mere proposal, subject to validation. Model validation examines the fit of the model to empirical data (measurements of the real-life system to be modeled). A good model fit means here that a set of important performance measures, predicted by the model, match or agree reasonably with their observed counterparts in the real-life system. Of course, this kind of validation is only possible if the real-life system or emulation thereof exists, and if the requisite measurements can actually be acquired. Any significant discrepancies would suggest that the proposed model is inadequate for project purposes, and that modifications are called for. In practice, it is common to go through multiple cycles of model construction, verification, validation, and modification.
6. *Designing and conducting simulation experiments*. Once the analyst judges a model to be valid, he or she may proceed to design a set of simulation experiments (runs) to estimate model performance and aid in solving the project's problem (often the problem is making system design decisions). The analyst selects a number of scenarios and runs the simulation to glean insights into its workings. To attain sufficient statistical reliability of scenario-related performance measures, each scenario is replicated (run multiple times, subject to different sequences of random numbers), and the results averaged to reduce statistical variability.
7. *Output analysis*. The estimated performance measures are subjected to a thorough logical and statistical analysis. A typical problem is one of identifying the best design among a number of competing alternatives. A statistical analysis would run statistical inference tests to determine whether one of the alternative designs enjoys superior performance measures, and so should be selected as the apparent best design.
8. *Final recommendations*. Finally, the analyst uses the output analysis to formulate the final recommendations for the underlying systems problem. This is usually part of a written report.

1.6 SIMULATION COSTS AND RISKS

Simulation modeling, while generally highly effective, is not free. The main costs incurred in simulation modeling, and the risks attendant to it, are listed here.

- *Modeling cost*. Like any other modeling paradigm, good simulation modeling is a prerequisite to efficacious solutions. However, modeling is frequently more art than science, and the acquisition of good modeling skills requires a great deal of practice and experience. Consequently, simulation modeling can be a lengthy and costly process. This cost element is, however, a facet of any type of modeling. As in any modeling enterprise, the analyst runs the risk of postulating an inaccurate or patently wrong model, whose invalidity failed to manifest itself at the validation stage. Another pitfall is a model that incorporates excessive detail. The right level of detail depends on the underlying problem. The art of modeling involves the construction of the least-detailed model that can do the job (producing adequate answers to questions of interest).
- *Coding cost*. Simulation modeling requires writing software. This activity can be error-prone and costly in terms of time and human labor (complex software projects are notorious for frequently failing to complete on time and within budget). In addition, the ever-present danger of incorrect coding calls for meticulous and costly verification.
- *Simulation runs*. Simulation modeling makes extensive use of statistics. The analyst should be careful to design the simulation experiments, so as to achieve adequate statistical reliability. This means that both the number of simulation runs (replications) and their length should be of adequate magnitude. Failing to do so is to risk the statistical reliability of the estimated performance measures. On the other hand, some simulation models may require enormous computing resources (memory space and CPU time). The modeler should be careful not to come up with a simulation model that requires prohibitive computing resources (clever modeling and clever code writing can help here).
- *Output analysis*. Simulation output must be analyzed and properly interpreted. Incorrect predictions, based on faulty statistical analysis, and improper understanding of system behavior are ever-present risks.

1.7 EXAMPLE: A PRODUCTION CONTROL PROBLEM

This section presents a simple production control problem as an example of the kind of systems amenable to simulation modeling. This example illustrates system definition and associated performance issues. A variant of this system will be discussed in Section 11.3, and its Arena model will be presented there.

Consider a packaging/warehousing process with the following steps:

1. The product is filled and sealed.
2. Sealed units are placed into boxes and stickers are placed on the boxes.
3. Boxes are transported to the warehouse to fulfill customer demand.

These steps can be combined into a single processing time, as depicted in the system schematic of Figure 1.1.

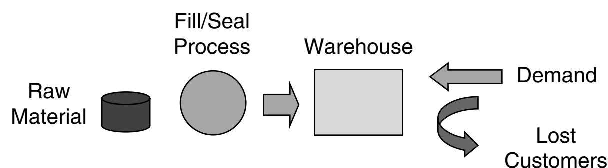


Figure 1.1 Pictorial representation of a packaging/warehousing system.

The system depicted in Figure 1.1 is subject to the following assumptions:

1. There is always sufficient raw material for the process never to *starve*.
2. Processing is carried out in batches, five units to a batch. Finished units are placed in the warehouse. Data collected indicate that unit-processing times are uniformly distributed between 10 and 20 minutes.
3. The process experiences *random failures*, which may occur at any point in time. Times between failures are exponentially distributed with a mean of 200 minutes. Data collection also showed that repair times are normally distributed, with a mean of 90 minutes and a standard deviation of 45 minutes.
4. The warehouse has a *capacity* (target level) of $R = 500$ units. Processing stops when the inventory in the warehouse reaches the target level. From this point on, the production process becomes *blocked* and remains inactive until the inventory level drops to the *reorder* point, which is assumed to be $r = 150$ units. The process restarts with a new batch as soon as the reorder level is down-crossed. This is a convenient policy when a resource needs to be distributed among various types of products. For instance, when our process becomes blocked, it may actually be assigned to another task or product that is not part of our model.
5. Data collection shows that interarrival times between successive customers are uniformly distributed between 3 and 7 hours, and that individual demand sizes are distributed uniformly between 50 and 100 units. On customer arrival, the inventory is immediately checked. If there is sufficient stock on hand, that demand is promptly satisfied; otherwise, customer demand is either partially satisfied and the rest is lost (that is, the unsatisfied portion represents lost business), or the entire demand is lost, depending on the availability of finished units and the loss policy employed.

This problem is also known as a *production/inventory problem*. We mention, however, that the present example is a gross simplification. More realistic production/inventory problems have additional wrinkles, including multiple types of products, production setups or startups, and so on. Some design and performance issues and associated performance measures of interest follow:

1. Can we improve the customer service level (percentage of customers whose demand is completely satisfied)?
2. Is the machinery underutilized or overutilized (machine utilization)?
3. Is the maintenance level adequate (downtime probabilities)?
4. What is the tradeoff between inventory level and customer service level?

The previous list is far from being exhaustive, and the actual requisite list will, of course, depend on the problem to be solved.

1.8 PROJECT REPORT

Once a system has been modeled, simulated, and analyzed, the study and its conclusions are often written up as a project report. A well-thought-out and properly written report is essential to the success of the study, since its value lies in its conclusions and their dissemination to technical and management personnel. The goal of this section is to aid the reader in structuring a generic report for a performance analysis study by outlining a generic skeleton.

A project report should be clearly and plainly written, so that nontechnical people as well as professionals can understand it. This is particularly true for project conclusions and recommendations, as these are likely to be read by management as part of the decision-making process. Although one can hardly overemphasize the importance of project reporting, the topic of proper reporting is rarely addressed explicitly in the published literature. In practice, analysts learn project-writing skills on the job, typically by example.

A generic simulation project report addresses the model building stages described in Section 1.5, and consists of (at least) the following sections:

- *Cover page*. Includes a project title, author names, date, and contact information in this order. Be sure to compose a descriptive but pithy project title.
- *Executive summary*. Provides a summary of the problem studied, model findings, and conclusions.
- *Table of contents*. Lists section headings, figures, and tables with the corresponding page numbers.
- *Introduction*. Sets up the scene with background information on the system under study (if any), the objectives of the project, and the problems to be solved. If appropriate, include a brief review of the relevant company, its location, and products and services.
- *System description*. Describes in detail the system to be studied, using prose, charts, and tables (see Section 1.7). Include all relevant details but no more (the rest of the book and especially Chapters 11–13 contain numerous examples).
- *Input analysis*. Describes empirical data collection and statistical data fitting (see Section 1.5 and Chapter 7). For example, the Arena Input Analyzer (Section 7.4) provides facilities for fitting distributions to empirical data and statistical tests. More advanced model fitting of stochastic processes to empirical data is described in Chapter 10.
- *Simulation model description.* Describes the modeling approach of the simulation model, and outlines its structure in terms of its main components, objects, and the operational logic. Be sure to decompose the description of a complex model into manageable-size submodel descriptions. Critical parts of the model should be described in some detail.
- *Verification and validation*. Provides supportive evidence for model goodness via model verification and validation (see Section 1.5) to justify its use in predicting the performance measures of the system under study (see Chapter 8). To this end, be sure to address at least the following two issues: (1) Does the model appear to run correctly and to provide the relevant statistics (verification)? (2) If the modeled system exists, how close are its statistics (e.g., mean waiting times, utilizations, throughputs, etc.) to the corresponding model estimates (validation)?
- *Output analysis*. Describes simulation model outputs, including run scenarios, number of replications, and the statistical analysis of simulation-produced observations (see Chapter 9). For example, the Arena Output Analyzer (Section 9.6) provides facilities for data graphing, statistical estimation, and statistical comparisons, while the Arena Process Analyzer (Section 9.7) facilitates parametric analysis of simulation models.
- *Simulation results*. Collects and displays summary statistics of multiple replicated scenarios. Arena's report facilities provide the requisite summary statistics (numerous reports are displayed in the sequel).
- *Suggested system modifications (if any)*. A common motivation for modeling an extant system is to come up with modifications in system parameters or configurations

that produce improvements, such as better performance and reduced costs. Be sure to discuss thoroughly the impact of suggested modifications by quantifying improvements and analyzing tradeoffs where relevant.

- *Conclusions and recommendations*. Summarizes study findings and furnishes a set of recommendations.
- *Appendices*. Contains any relevant material that might provide undesired digression in the report body.

Needless to say, the modeler/analyst should not hesitate to customize and modify the previous outlined skeleton as necessary.

EXERCISES

1. Consider an online registration system (e.g., course registration at a university or membership registration in a conference).
   a. List the main components of the system and its transactions.
   b. How would you define the state and events of each component of the registration system?
   c. Which performance measures might be of interest to registrants?
   d. Which performance measures might be of interest to the system administrator?
   e. What data would you collect?
2. The First New Brunswick Savings (FNBS) bank has a branch office with a number of tellers serving customers in the lobby, a teller serving the drive-in line, and a number of service managers serving customers with special requests. The lobby, drive-in, and service managers each have a separate single queue. Customers may join either of the queues (the lobby queue, the drive-in queue, or the service managers' queue). FNBS is interested in performance evaluation of their customer service operations.
   a. What are the random components in the system and their parameters?
   b. What performance measures would you recommend FNBS to consider?
   c. What data would you collect and why?
3. Consider the production/inventory system of Section 1.7. Suppose the system produces and stores multiple products.
   a. List the main components of the system and its transactions as depicted in Figure 1.1.
   b. What are the transactions and events of the system, in view of Figure 1.1?
   c. Which performance measures might be of interest to customers, and which to owners?
   d. What data would you collect and why?

Chapter 2

Discrete Event Simulation

The majority of modern computer simulation tools (simulators) implement a paradigm, called *discrete-event simulation* (*DES*). This paradigm is so general and powerful that it provides an implementation framework for most simulation languages, regardless of the user worldview supported by them. Because this paradigm is so pervasive, we will review and explain in this chapter its working in some detail.

2.1 ELEMENTS OF DISCRETE EVENT SIMULATION

In the DES paradigm, the simulation model possesses a *state* S (possibly vector-valued) at any point in time. A *system* state is a set of data that captures the salient variables of the system and allows us to describe system evolution over time. In a computer simulation program, the state is stored in one or more program variables that represent various data structures (e.g., the number of customers in a queue, or their exact sequence in the queue). Thus, the state can be defined in various ways, depending on particular modeling needs, and the requisite level of detail is incorporated into the model. As an example, consider a machine, fed by a raw-material storage of jobs. A "coarse" state of the system is the number jobs in the storage; note, however, that this state definition does not permit the computation of waiting times, because the identity of individual jobs is not maintained. On the other hand, the more "refined" state consisting of customer identities in a queue and associated data (such as customer arrival times) does permit the computation of waiting times. In practice, the state definition of a system should be determined based on its modeling needs, particularly the statistics to be computed.

The state trajectory over time, $S(t)$, is abstracted as a step function, whose jumps (discontinuities) are triggered by discrete *events*, which induce *state transactions* (changes in the system state) at particular points in time. Although computer implementation of events varies among DES simulators, they are all conceptually similar: An event is a data structure that always has a field containing its time of occurrence, and any number of other fields. Furthermore, the "occurrence" of an event in a DES simulator is implemented as the execution of a corresponding procedure (computer code) at the scheduled event occurrence time. When that procedure is run, we say that the event is *processed* or *executed*.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

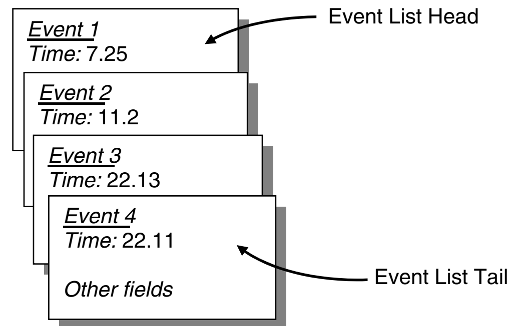


Figure 2.1 Structure of a DES event list.

The evolution of any DES model is governed by a *clock* and a chronologically ordered *event list*. That is, events are linked in the event list according to their scheduled order of occurrence (Figure 2.1). The event at the head of the list is called the *most imminent event* for obvious reasons. *Scheduling an event* means that the event is linked chronologically into the event list. The *occurrence* of an event means that the event is unlinked from the event list and executed. The execution of an event can change state variables and possibly schedule other events in the event list.

An essential feature of the DES paradigm is that "nothing" changes the state unless an event occurs, at which point the model typically undergoes a state transition. More precisely, every event execution can change the state (although on rare occasions the state remains intact), but every state change is effected by some event. Between events, the state of the DES is considered constant, even though the system is engaged in some activity. For example, consider a machine on the factory floor that packages beer cans into six-packs, such that the next six-pack is loaded for processing only when the previous one has been completely processed. Suppose that the state tracks the number of six-packs waiting to be processed at any given time. Then during the processing time of a six-pack, the DES state remains unchanged, even though the machine may, in fact, be processing six beer cans individually. The DES state will only be updated when the entire six-pack is processed; this state change will be triggered by a "six-pack completion" event. Note again that the definition of the state is up to the modeler, and that models can be refined by adding new types of events that introduce additional types of state transitions.

At the highest level of generalization, a DES simulator executes the following algorithm:

1. Set the simulation clock to an initial time (usually 0), and then generate one or more initial events and schedule them.
2. If the event list is empty, terminate the simulation run. Otherwise, find the most imminent event and unlink it from the event list.
3. Advance the simulation clock to the time of the most imminent event, and execute it (the event may stop the simulation).
4. Loop back to Step 2.

This beguilingly simple algorithm (essentially an infinite loop) is extremely general. Its complexity is hidden in the routines that implement event execution and the data structures used by them. The power and versatility of the DES simulation algorithm

stem from the fact that the DES paradigm naturally scales to collections of interacting subsystems: one can build hierarchies of increasingly complex systems from subsystem components. In addition, the processing of any event can be as intricate as desired. Thus, both large systems as well as complex ones can be represented in the DES paradigm. (For more details, see Fishman 1973, Banks et al. 1999, and Law and Kelton 2000.)

2.2 EXAMPLES OF DES MODELS

In this section the power and generality of DES models are illustrated through several examples of elementary systems. The examples illustrate how progressively complex DES models can be constructed from simpler ones, either by introducing new modeling wrinkles that increase component complexity, or by adding components to create larger DES models.

2.2.1 SINGLE MACHINE

Consider a failure-proof single machine on the shop floor, fed by a buffer. Arriving jobs that find the machine busy (processing another job) must await their turn in the buffer, and eventually are processed in their order of arrival. Such a service discipline is called FIFO (first in first out) or FCFS (first come first served), and the resulting system is called a queue or queueing system. (The word "queue" is derived from French and ultimately from a Latin word that means "tail," which explains its technical meaning as a "waiting line." Its quaint spelling renders it one of the most vowel-redundant words in the English language.) Suppose that job interarrival times and processing times are given (possibly random). A schematic description of the system is depicted in Figure 2.2.

To represent this system as a DES, define the state $S(t)$ to be the number of jobs in the system at time t. Thus, $S(t) = 5$ means that at time t, the machine is busy processing the first job and 4 more jobs are waiting in the buffer. There are two types of events: *arrivals* and *process completions*. Suppose that an arrival took place at time t, when there were $S(t) = n$ jobs in the system. Then the value of S jumps at time t from n to $n + 1$, and this transition is denoted by $n \to n + 1$. Similarly, a process completion is described by the transition $n \to n - 1$. Both transitions are implemented in the simulation program as part of the corresponding event processing.

2.2.2 SINGLE MACHINE WITH FAILURES

Consider the previous single machine on the shop floor, now subject to failures. In addition to arrival and service processes, we now also need to describe times to

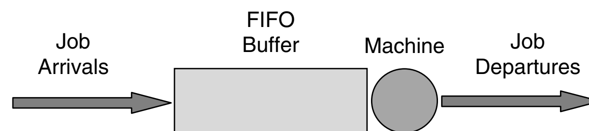


Figure 2.2 A single FIFO machine.

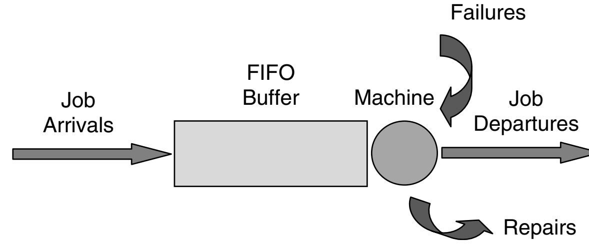


Figure 2.3 A single FIFO machine with failures.

failure as well as repair times. We assume that the machine fails only while processing a job, and that on repair completion, the job has to be reprocessed from scratch. A schematic description of the system is depicted in Figure 2.3.

The state $S(t)$ is a pair of variables, $S(t) = (N(t),\ V(t))$, where $N(t)$ is the number of jobs in the buffer, and $V(t)$ is the process status (idle, busy, or down), all at time t. In a simulation program, $V(t)$ is coded, say by integers, as follows: $0 =$ idle, $1 =$ busy, and $2 =$ down. Note that one job must reside at the machine, whenever its status is *busy* or *down*.

The events are *arrivals*, *process completions*, *machine failures*, and *machine repairs*. The corresponding state transitions follow:

- Job arrival: $(n,\ v) \rightarrow (n+1,\ v)$
- Service process completion: $(n,\ 1) \rightarrow \begin{cases} (0,\ 0), & \text{if } n = 1 \\ (n-1,\ 1), & \text{if } n > 1 \end{cases}$
- Failure arrival: $(n,\ 1) \rightarrow (n,\ 2)$
- Repair completion: $(n,\ 2) \rightarrow (n,\ 1)$

2.2.3 Single Machine with an Inspection Station and Associated Inventory

Consider the single machine on a shop floor, without failures. Jobs that finish processing go to an inspection station with its own buffer, where finished jobs are checked for defects. Jobs that pass inspection are stored in a finished inventory warehouse. However, jobs that fail inspection are routed back to the tail end of the machine's buffer for reprocessing. In addition to interarrival times and processing times, we need here a description of the inspection time as well as the inspection decision (pass/fail) mechanism (e.g., jobs fail with some probability, independently of each other). A schematic description of the system is depicted in Figure 2.4.

The state $S(t)$ is a triplet of variables, $S(t) = (N(t),\ I(t),\ K(t))$ where $N(t)$ is the number of items in the machine and its buffer, $I(t)$ is the number of items at the inspection station, and $K(t)$ is the storage content, all at time t. Events consist of *arrivals*, *process completions*, *inspection failure* (followed by routing to the tail end of the machine's buffer), and *inspection passing* (followed by storage in the warehouse). The corresponding state transitions follow:

- Job arrival: $(n,\ i,\ k) \rightarrow (n+1,\ i,\ k)$
- Process completion: $(n,\ i,\ k) \rightarrow (n-1,\ i+1,\ k)$
- Inspection completion: $(n,\ i,\ k) \rightarrow \begin{cases} (n+1, i-1, k), & \text{if job failed inspection} \\ (n,\ i-1, k+1), & \text{if job passed inspection} \end{cases}$

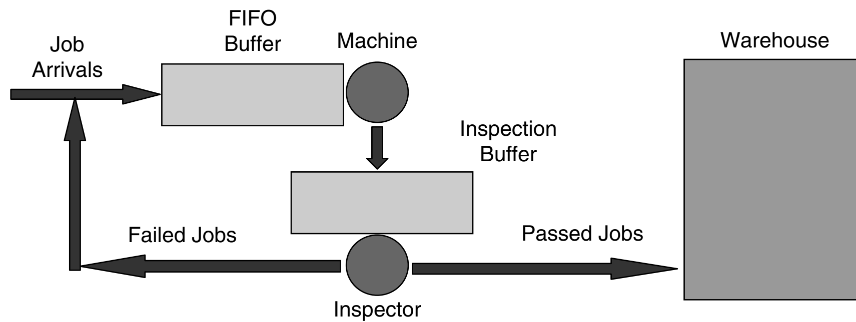


Figure 2.4 A single FIFO machine with inspection and storage.

2.3 MONTE CARLO SAMPLING AND HISTORIES

Monte Carlo simulation models incorporate randomness by sampling random values from specified distributions. The underlying algorithms and/or their code use *random number generators* (*RNG*) that produce uniformly distributed values ("equally likely") between 0 and 1; these values are then transformed to conform to a prescribed distribution. We add parenthetically that the full term is *pseudo RNG* to indicate that the numbers generated are not "truly" random (they can be reproduced algorithmically), but only random in a statistical sense; however, the prefix "pseudo" is routinely dropped for brevity. This general sampling procedure is referred to as Monte Carlo sampling. The name is attributed to von Neumann and Ulam for their work at Los Alamos National Laboratory (see Hammersly and Handscomb 1964), probably as an allusion to the famous casino at Monte Carlo and the relation between random number generation and casino gambling.

In particular, the random values sampled using RNGs are used (among other things) to schedule events at random times. For the most part, actual event times are determined by sampling an interevent time (e.g., interarrival times, times to failure, repair times, etc.) via an RNG, and then adding that value to the current clock time. More details are presented in Chapter 4.

DES runs use a statistical approach to evaluating system performance; in fact, simulation-based performance evaluation can be thought of as a statistical experiment. Accordingly, the requisite performance measures of the model under study are not computed exactly, but rather, they are estimated from a set of histories. A standard statistical procedure unfolds as follows:

1. The modeler performs multiple simulation runs of the model under study, using independent sequences of random numbers. Each run is called a *replication*.
2. One or more performance measures are computed from each replication. Examples include average waiting times in a queue, average WIP (work in process) levels, and downtime probabilities.
3. The performance values obtained are actually random and mutually independent, and together form a statistical sample. To obtain a more reliable estimate of the true value of each performance metric, the corresponding values are averaged and confidence intervals about them are constructed. This is discussed in Chapter 3.

2.3.1 Example: Work Station Subject to Failures and Inventory Control

This section presents a detailed example that illustrates the random nature of DES modeling and simulation runs, including the random state and its sample paths. Our goal is to study system behavior and estimate performance metrics of interest. To this end, consider the workstation depicted in Figure 2.5.

The system is comprised of a machine, which never starves (always has a job to work on), and a warehouse that stores finished products (jobs). In addition, the machine is subject to failures, and its status is maintained in the random variable $V(t)$, given by

$$V(t) = \begin{cases} 0, & \text{idle} \\ 1, & \text{busy} \\ 2, & \text{down} \end{cases}$$

Note that one job must reside at the machine, whenever its status is busy or down. The state $S(t)$ is a pair of variables, $S(t) = (V(t), K(t))$ where $V(t)$ is the status of the machine as described previously, and $K(t)$ is the finished-product level in the warehouse, all at time t. For example, the state $S(t) = (2, 3)$ indicates that at time t the machine is down (presumably being repaired), and the warehouse has an inventory of three finished product units. Customer orders (demand) arrive at the warehouse, and filled orders deplete the inventory by the ordered amount (orders that exceed the stock on hand are partially filled, the shortage simply goes unfilled, and no backorder is issued). The product unit processing time is 10 minutes. In this example, the machine does not operate independently, but rather is controlled by the warehouse as follows. Whenever the inventory level reaches or drops below $r = 2$ units (called the *reorder point*), the warehouse issues a replenishment request to the machine to bring the inventory up to the level of $R = 5$ units (called *target level* or *base-stock level*). In this case, the inventory level is said to down-cross the reorder point. At this point, the machine starts processing a sequence of jobs until the inventory level reaches the target value, R, at which point the machine suspends operation. Such a control regime is known as the (r, R) *continuous review* inventory control policy (or simply as the (r, R) policy), and the corresponding replenishment regime is referred to as a *pull system*. See Chapter 12 for detailed examples.

Sample History

Suppose that events occur in the DES model of the workstation above in the order shown in Figure 2.6, which graphs the stock on hand in the warehouse as a function of

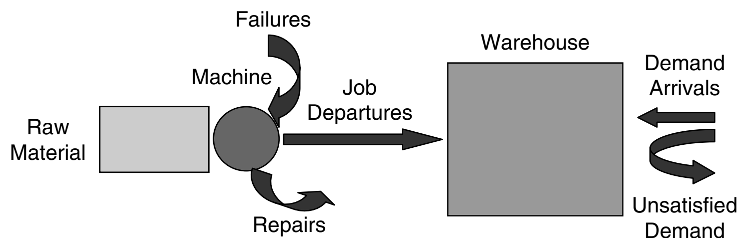


Figure 2.5 Workstation subject to failures and inventory control.

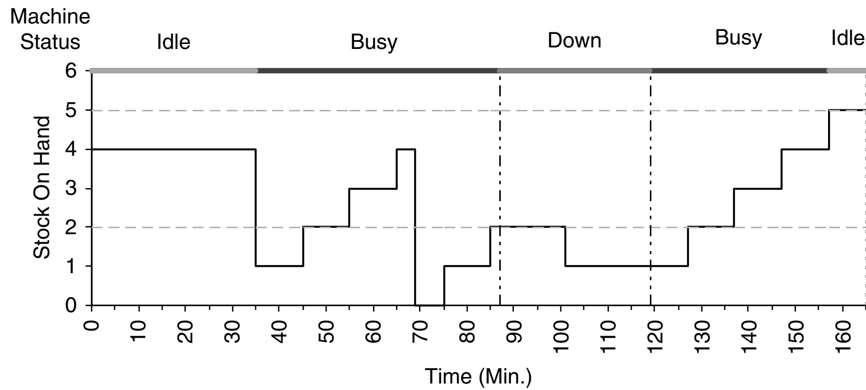


Figure 2.6 Operational history of the system of Figure 2.5.

time, and also tracks the status of the machine, $V(t)$, over time. (Note that Figure 2.6 depicts a sample history—one of many possible histories that may be generated by the simulated system.)

An examination of Figure 2.6 reveals that at time $t = 0$, the machine is idle and the warehouse contains four finished units, that is, $V(0) = 0$ and $K(0) = 4$. The first customer arrives at the warehouse at time $t = 35$ and demands three units. Since the stock on hand can satisfy this order, it is depleted by three units, resulting in $K(35) = 1$; at this point, the reorder point, r, is down-crossed, triggering a replenishment request at the machine that resumes the production of additional product in order to raise the inventory level to its target value, R. Note that the machine status changes concomitantly from *idle* to *busy*. During the next 30 minutes, no further demand arrives, and the inventory level climbs gradually as finished products arrive from the machine, until reaching a level of 4 at time $t = 65$.

At time $t = 69$, a second customer arrives and places a demand equal or larger than the stock on hand, thereby depleting the entire inventory. Since unsatisfied demand goes unfilled, we have $K(69) = 0$. If backorders were allowed, then we would keep track of the backorder size represented by the magnitude of the corresponding negative inventory.

At time $t = 75$, the unit that started processing at the machine at time $t = 65$ is finished and proceeds to the warehouse, so that $K(75) = 1$. Another unit is finished with processing at the machine at time $t = 85$.

At time $t = 87$, the machine fails and its repair begins (*down* state). The repair activity is completed at time $t = 119$ and the machine status changes to *busy*. While the machine is down, a customer arrives at time $t = 101$, and the associated demand decreases the stock on hand by one unit, so that $K(101) = 1$. At time $t = 119$, the repaired machine resumes processing of the unit whose processing was interrupted at the time of failure; that unit completes processing at time $t = 127$.

From time $t = 127$ to time $t = 157$ no customers arrive at the warehouse, and consequently the inventory reaches its target level, $R = 5$, at time $t = 157$, at which time the machine suspends production. The simulation run finally terminates at time $T = 165$.

Sample Statistics

Having generated a sample history of system operation, we can now proceed to compute associated statistics (performance measures).

Probability distribution of machine status. Consider the machine status over the time interval [0, T]. Let T_I be the total idle time over [0, T], T_B the total busy time over [0, T], and T_D the total downtime over [0, T]. The probability distribution of machine status is then estimated by the ratios of time spent in a state to total simulation time, namely,

$$\Pr\{\text{machine idle}\} = \frac{T_I}{T} = \frac{35 + (165 - 157)}{165} = 0.261,$$

$$\Pr\{\text{machine busy}\} = \frac{T_B}{T} = \frac{(87 - 35) + (157 - 119)}{165} = 0.545,$$

$$\Pr\{\text{machine down}\} = \frac{T_D}{T} = \frac{119 - 87}{165} = 0.194.$$

In particular, the probability that the machine is busy coincides with the server utilization (the fraction of time the machine is actually busy producing). Note that all the probabilities above are estimated by *time averages*, which here assume the form of the *fraction of time* spent by the machine in each state (the general form of time averages is discussed in Section 9.3). The logic underlying these definitions is simple. If an outside observer "looks" at the system at random, then the probability of finding the machine in a given state is *proportional* to the total time spent by the machine in that state. Of course, the ratios (proportions) above sum to unity, by definition.

Machine throughput. Consider the number of job completions C_T in the machine over the interval [0, T]. The *throughput* is a measure of effective processing rate, namely, the expected number of job completions (and, therefore, departures) per unit time, estimated by

$$\bar{o} = \frac{C_T}{T} = \frac{9}{165} = 0.0545.$$

Customer service level. Consider customers arriving at the warehouse with a demand for products. Let N_S be the number of customers whose demand is fully satisfied over the interval [0, T], and N_T the total number of customers that arrived over [0, T]. The customer service level, ξ, is the probability of fully satisfying the demand of an arrival at the warehouse. This performance measure is estimated by

$$\xi = \frac{N_s}{N_T} = \frac{2}{3} = 0.6667,$$

assuming that the demand of the customer arriving at $t = 69$ is not fully satisfied. Note that the ξ statistic is a *customer average*, which assumes here the form of the *relative frequency* of satisfied customers (the general form of customer averages is discussed in Section 9.3). Additionally, letting J_k be the unmet portion of the demand of customer k (possibly 0), the customer average of unmet demands is given by

$$\bar{J} = \frac{\sum_{k=1}^{M} J_k}{M},$$

where M is the total number of customers.

Table 2.1
Estimated distribution of finished products in the warehouse

k	0	1	2	3	4	5
$\Pr\{K = k\}$	0.036	0.279	0.218	0.121	0.297	0.048

Probability distribution of finished products in the warehouse. Consider the probability that the long-term number of finished units in the warehouse, K, is at some given level, k. These probabilities are estimated by the expression

$$\Pr\{k \text{ units in stock}\} = \frac{\text{total time spent with } k \text{ units in stock}}{\text{total time}},$$

and in particular, for $k = 0$,

$$\Pr\{\text{stockout}\} = \frac{75 - 69}{165} = 0.036.$$

Suppressing the time index, the estimated distribution is displayed in Table 2.1.

Summing the estimated probabilities above reveals that $\sum_{k=0}^{5} \Pr\{K = k\} = 0.999$ instead of $\sum_{k=0}^{5} \Pr\{K = k\} = 1$, due to round-off errors. Such slight numerical inaccuracies are a fact of life in computer-based computations.

Average inventory on hand. The average inventory level is estimated by

$$\bar{K} = \sum_{k=0}^{5} k \Pr\{K = k\} = 2.506,$$

which is a consequence of the general time average formula (see Section 9.3)

$$\bar{K} = \frac{\int_0^T K(t)\, dt}{T}.$$

2.4 DES LANGUAGES

A simulation model must be ultimately transcribed into computer code, using some programming language. A simulation language may be *general purpose* or *special purpose*. A general-purpose programming language, such as C++ or Visual Basic, provides no built-in simulation objects (such as a simulation clock or event list), and no simulation services (e.g., no clock updating or scheduling). Rather, the modeler must code these objects and routines from scratch; on rare occasions, however, the generality of such languages and the ability to code “anything we want” is actually advantageous. In contrast, a special-purpose simulation language implements a certain *simulation worldview*, and therefore *does* provide the corresponding simulation objects and services as built-in constructs. In addition, a good special-purpose language supports a variety of other simulation-related features, such as Monte Carlo sampling and

convenient reporting. While on rare occasions the built-in worldview may present difficulties in implementing unusual simulation constructs, the convenience of using built-in simulation services far outweighs the occasional disadvantage. The main reason is the reduced coding time: The modeler can concentrate on modeling, and rely on built-in constructs to facilitate the coding process and its debugging.

It is strongly recommended that the modeler give considerable thought to the selection of an appropriate simulation language. The main selection criterion is the degree to which the simulation language's worldview fits the type of simulation model to be coded. You should, of course, trade off the cost of learning a new (and better) simulation language and the convenience of using a possibly worse but familiar one. Generally, the cost paid in switching to a better simulation language is well worth it.

A broad variety of simulation languages are currently available in the marketplace to fit practically any conceivable worldview. Simulation languages can themselves be classified as general purpose and special purpose. Thus, *general-purpose simulation languages*, such as Arena/SIMAN (Kelton et al. 1998), PROMODEL (Benson 1996), GPSS (Schriber 1990), SLAM (Pritsker 1986), and MODSIM (Belanger et al. 1989), can be used to efficiently model virtually any system. Other simulation languages are specialized in one way or another to a particular modeling domain (e.g., telecommunications, manufacturing, etc.). The more specialized the language, the easier it is to use within its natural application domain, and the harder it is to use outside it. For example, COMNET (CACI 1988) is a special-purpose simulation language tailored to simulation modeling of communication networks.

EXERCISES

1. Consider a batch manufacturing process in which a machine processes jobs in batches of three units. The process starts only when there are three or more jobs in the buffer in front of the machine. Otherwise, the machine stays idle until the batch is completed. Assume that job interarrival times are uniformly distributed between 2 and 8 hours, and batch service times are uniformly distributed between 5 and 15 hours.

   Assuming the system is initially empty, simulate the system manually for three batch service completions and calculate the following statistics:

   - Average number of jobs in the buffer (excluding the batch being served)
   - Probability distribution of number of jobs in the buffer (excluding the batch being served)
   - Machine utilization
   - Average job waiting time (time in buffer)
   - Average job system time (total time in the system, including processing time)
   - System throughput (number of departing jobs per unit time)

   Approach: After each arrival, schedule the next interarrival time, and when each batch goes into service, schedule its service completion time. To obtain these quantities, use your calculator to generate a sequence of random numbers (these are equally likely between 0 and 1, and statistically independent of each other). Then transform these numbers as follows:

a. To generate the next random interarrival time, A, generate the next random number U from your calculator, and set $A = 2 + 6U$.
b. To generate the next random batch service time, B, generate the next random number U from your calculator, and set $B = 5 + 10U$.

If you cannot use random numbers from your calculator, use instead deterministic interarrival times, $A = 5$ (the average of 2 and 8), and deterministic service times $B = 10$ (the average of 5 and 15).

2. A manufacturing facility has a repair shop with two repairmen who repair failed machines on a first-fail-first-serve basis. They work together on the machine if there is one machine down (the repair still takes the same amount of time), and otherwise, each works on a separate machine. Thus, if there are more than two machines down, new failures simply wait for their turn to be repaired. Assume that machine failures arrive in a combined failure stream, so that we do not need to track machine identity. More specifically, assume that the times between machine failures are equally likely between 10 and 20 hours, and that repair times are equally likely between 5 and 55 hours for each machine.

   To manually simulate the manufacturing facility, use the approach of items (a) and (b) of Exercise 1 utilizing the following sequence of uniform random numbers (between 0 and 1), $U = \{0.2, 0.5, 0.9, 0.7, 0.8, 0.1, 0.5, 0.2, 0.7, 0.4, 0.3\}$ to generate times to failure and repair times. Simulate the manufacturing facility manually for five machine repair completions, and calculate the following statistics:

   - Fraction of time a machine is down
   - Average number of down machines waiting to be repaired
   - Fraction of time each repairman is busy (repairman utilization)
   - Fraction of time the repair facility is idle
   - Average time a failed machine waits until its repair starts
   - Throughput of the repair facility (number of repair completions per hour)

Note: These manual procedures will simulate the system for a short period of time. The tedium involved in simulating the system should make you realize that you need a computer program for long simulations or of even those of moderate complexity.

This page intentionally left blank

Chapter 3

Elements of Probability and Statistics

Many real-life systems exhibit behavior with random elements. Such systems encompass a vast array of application areas, such as the following:

1. Manufacturing
   - Random demand for product held in an inventory system
   - Random product processing time or transfer time
   - Random machine failures and repairs
2. Transportation
   - Random congestion on a highway
   - Random weather patterns
   - Random travel times between pairs of origination and destination points
3. Telecommunications
   - Random traffic arriving at a telecommunications network
   - Random transmission time (depending on available resources, such as buffer space and CPU)

Indeed, simulation modeling with random elements is often referred to as *Monte Carlo simulation*, presumably after its namesake casino at Monte Carlo on the Mediterranean. This apt term commemorates the link between randomness and gambling, going back to the French scientist Blaise Pascal in the 17th century.

Formally, modeling a random system as a discrete-event simulation simply means that randomness is introduced into events in two basic ways:

- Event occurrence times may be random.
- Event state transitions may be random.

For instance, random interarrival times at a manufacturing station exemplify the first case, while random destinations of product units emerging from an inspection station (possibly needing re-work with some probability) exemplify the second. Either way, probability and statistics are fundamental to simulation models and to understanding the underlying random phenomena in a real-life system under study. In particular, they play a key role in simulation-related input analysis and output analysis. Recall that input

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

analysis models random components by fitting a probabilistic model to empirical data generated by the system under study, or by postulating a model when empirical data is lacking or insufficient. Once input analysis is complete and simulation runs (replications) are generated, output analysis is then employed to verify and validate the simulation model, and to generate statistical predictions for performance measures of interest.

This chapter reviews the basic probabilistic and statistical concepts underlying Monte Carlo simulation. Additional material will be presented in Chapter 7 (on input analysis), Chapter 8 (on model verification and validation), Chapter 9 (on output analysis), and Chapter 10 (on correlation analysis). For further readings on probability, see, for example, Çinlar (1975), Ross (1993), Hoel et al. (1971a), Feller (1968), and Taylor and Karlin (1984).

3.1 ELEMENTARY PROBABILITY THEORY

Informally, probability is a measure of the uncertainty inherent in the occurrence of random phenomena, such as the following statements in future tense:

- It will rain tomorrow.
- I will win the lottery next week.
- The Fed will raise interest rates next month.

Probability is measured on a continuous scale spanning the interval [0, 1]. In particular, a probability of 0 means that it is certain that the phenomenon *will not* occur, while a probability of 1 means that it is certain that the phenomenon *will* occur. Probabilities lying strictly between 0 and 1 quantify any intermediate likelihood of occurrence.

The notion of "likelihood" has a practical operational meaning, linked intimately with statistics. Suppose we observe multiple "experiments" in the underlying system (replications), and each time we record whether or not some specified phenomenon, A, occurred. Suppose we observed n such experiments and found that in k of them the phenomenon A occurred (and therefore, in $n - k$ of them, it did not occur). The probability of A occurring, is then estimated by the frequency ratio

$$\hat{p}_A = \frac{k}{n},$$

which is indeed between 0 and 1. This is merely an estimate with a likely experimental error, but we hope that as the number of experiments n increases, the accuracy of $\hat{\boldsymbol{p}}_{\boldsymbol{A}}$ would improve. In practice, people often use the term "probability" loosely to refer to its estimate, because the true probability is unknown.

Probability estimates can be more complex than simply frequency ratios. For example, a probability estimate of a horse winning a race can indeed be computed as a ratio based on the horse's historical track record. However, the odds published by book makers are estimates based on the opinion of the betting public, which is itself based on many other complex factors, such as past races, weather, trainers, and possibly illegal inside information. All these are encapsulated into a measurable quantity and an observable statistic.

3.1.1 Probability Spaces

The theory of probability is an abstraction that formalizes these ideas. It introduces a set of postulates, including a probability calculus. Formally, a probability space is a triple of objects, (Ω, E, Pr), where:

- Ω is the *sample space*, corresponding to all possible "outcomes" of the random phenomenon under consideration. Although the sample space is an abstract concept, a *sample point* $\omega \in \Omega$ can be thought of as an "experiment" over the underlying (random) system.
- E is the *event set*, corresponding to permissible sets of "outcomes." Thus, an event $A \in E$ is a set of sample points, that is, $A \subset \Omega$. The empty set, ϕ, and the sample space, Ω, always belong to E. Furthermore, if Ω is countable (finite or infinite), then every subset of Ω belongs to E. In all other cases, we must impose technical conditions on the membership of events, which are beyond the scope of this book.
- Pr is a *probability measure*, which satisfies the following postulates:
  a. $0 \leq \Pr\{A\} \leq 1$ for all $A \in E$ (in particular, $\Pr\{\phi\} = 0$ and $\Pr\{\Omega\} = 1$).
  b. For any events $A, B \in E$, satisfying $A \cap B = \phi$ (disjoint events),

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}, \tag{3.1}$$

  which is a special case of the equality

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\}. \tag{3.2}$$

The postulates above are reasonable. The probability of "no outcome" is impossible and therefore always evaluates to the minimal value, 0. The probability of any of the "possible outcomes" occurring is a certainty, and therefore always evaluates to the maximal value, 1. Finally, if two events do not overlap, their probability of occurrence is the sum of their probabilities. Otherwise, the sum of their probabilities contains twice the probability of their intersection (instead of one such probability), so one superfluous probability of the intersection is subtracted.

Let $\Omega = \cup A_n$ be a partition of the sample space into mutually disjoint events $\{A_n\}$. Then for any event B, the *formula of total probability* is

$$\Pr\{B\} = \sum_n \Pr\{A_n \cap B\}. \tag{3.3}$$

3.1.2 Conditional Probabilities

The concept of *conditioning* plays a major role in probability theory. More precisely, if A and B are events, such that $\Pr\{B\} > 0$, then the probability of event A *conditioned* on event B, is denoted by $\Pr\{A|B\}$ and defined by

$$\Pr\{A|B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}}. \tag{3.4}$$

Equation 3.4 is alternatively referred to as the probability of event A *given* event B. The meaning of conditional probabilities can be explained as follows. Suppose we wish to consider the occurrence of event A, but only if we know that a prescribed event B has actually occurred. In a sense, we require the event B to become our new sample space,

and we look at the probability of event A only when it occurs concurrently with event B (the numerator of Eq. 3.4). We divide by the probability of B (the denominator of Eq. 3.4), to ensure that the maximal value of the conditional probability, $\Pr\{B|B\}$, is normalized to 1. Thus, a conditional probability cannot be defined via Eq. 3.4 when the given (conditioning) event, B, has zero probability of occurring; in this case, the conditional probability should be specified by other means.

The operational meaning of conditioning can be viewed as the modification of the probability of an event A by the added "information" that another event B has actually occurred. For example, if we throw a single die, then the odds of the outcome being 4 is one in six (probability 1/6). However, suppose that after the die is cast, we are not allowed to see the outcome, but are told that the outcome was even. This new information modifies the previous probability, $\Pr\{\text{outcome is } 4\} = 1/6$, to a new probability, $\Pr\{\text{outcome is } 4|\text{outcome is even}\} = 1/3$, since the odds of obtaining an even outcome (2, 4, or 6) is 1 in 3 (note that these events are disjoint, so the probabilities are additive). By the same token, if it were known that the outcome turned out to be odd, then $\Pr\{\text{outcome is } 4|\text{outcome is odd}\} = 0$. If, however, we were told that the outcome was a two-digit number (an impossible event), we would not be able to define the conditional probability.

3.1.3 Dependence and Independence

The concepts of *event independence* and *event dependence* are expressed in terms of conditional probabilities. A set of events, $A_i, i = 1, 2, \ldots, n$, are said to be (mutually) *independent*, provided that

$$\Pr\{A_1, A_2, \ldots, A_n\} = \prod_{i=1}^{n} \Pr\{A_i\}, \tag{3.5}$$

where the notation $\Pr\{A_1, A_2, \ldots, A_n\}$ is shorthand for $\Pr\{A_1 \cap A_2 \cap \ldots \cap A_n\}$. Otherwise, the events are said to be dependent. For two events, A and B, Eq. 3.5 can be written as

$$\Pr\{A \cap B\} = \Pr\{A\} \times \Pr\{B\}. \tag{3.6}$$

The meaning of independence (or dependence) becomes clearer when we divide (when permissible) both sides of the above equation by $\Pr\{A\}$, and separately by $\Pr\{B\}$. We then obtain the dual equations

$$\Pr\{A|B\} = \Pr\{A\} \text{or} \Pr\{B|A\} = \Pr\{B\}, \tag{3.7}$$

each of which is equivalent to Eq. 3.6. Thus, from Eq. 3.7, independence holds when the conditional and unconditional probabilities are equal. In other words, knowledge of one event does not modify the (unconditioned) probability of the other event. Presumably, this is so because the two events are "unrelated," and the occurrence of one does not affect the odds of the other. Conversely, two events are dependent, if knowledge of one event modifies the probability of the other.

Be careful not to confuse independent events ($\Pr\{A \cap B\} = \Pr\{A\} \times \Pr\{B\}$) with disjoint events ($\{A \cap B\} = \phi$). These are entirely different concepts, none of which necessarily implies the other.

3.2 RANDOM VARIABLES

Conducting an experiment can be thought of as sampling an observation at a sample point, subject to some underlying probability. For example, suppose we select at random a car on the assembly line for quality assurance. We can then make multiple measurements on the car, each revealing a different aspect of its quality, including possibly the following:

- Breaking distance at 65 miles per hour
- Extent of tire wear after 50,000 miles
- Crash test performance

The concept of *random variable* is the theoretical construct that captures aspects of sample points. In the simulation context, a random variable is also referred to as a *variate*. It should be pointed out that even though practitioners do not always refer explicitly to an underlying probability space, such a space is always assumed implicitly.

Omitting some technical conditions, which are beyond the scope of this book, a random variable X is a function

$$X: \Omega \to S, \tag{3.8}$$

where Ω is the underlying sample space, and S is called the *state space* of X, and consists of all possible values that X can assume. A particular value, $X(\omega) = x \in S$, realized by a random variable for a particular sample point, ω ("experiment outcome"), is called a *realization* of X. For example, a particular car in a road test plays the role of a sample point, ω, while its properties (breaking distance, tire wear, etc.) correspond to realizations of various random variables. Note carefully that the notion of a random variable is quite distinct from the notion of its realizations. To keep this distinction typographically clear, we shall always denote realizations by lower-case letters and random variables by upper-case letters.

A state space S can be quite general. It may be real valued or vector valued. In fact, it need not be numerical at all in order to capture qualitative aspects. For example, if the random variable X represents the status of a machine, the corresponding state space may be defined as the four status descriptors $S = \{\text{Idle}, \text{Busy}, \text{Down}, \text{Blocked}\}$.

Random variables are classified according to their associated state space. A state space is said to be *discrete* if it is countable, or *continuous*, if it is not (it can also be *mixed* with discrete and continuous components). For example, the status indicators $S = \{\text{Up}, \text{Down}\}$ for a machine form a discrete state space. However, the random variable that measures the time to failure of the machine has a continuous state space, since it can take values in some interval $S = [0, T_{\max}]$ of non-negative real numbers.

3.3 DISTRIBUTION FUNCTIONS

The probabilistic properties of random variables are characterized by their *distribution functions* (often abbreviated to *distributions*). These functions assume various forms, depending on the type of the associated random variable and the nature of its state space (numerical or not). In particular, a distribution function is continuous or discrete (or mixed) according to the type of its associated random variable.

3.3.1 Probability Mass Functions

Every discrete random variable X has an associated *probability mass function* (*pmf*), $p_X(x)$, defined by

$$p_X(x) = \Pr\{X = x\}, \quad x \in S. \tag{3.9}$$

Note that the notation $\{X = x\}$ above is a shorthand notation for the event $\{\omega: X(\omega) = x\}$. It should be pointed out that the technical definition of a random variable ensures that this set is actually an event (i.e., belongs to the underlying event set E). Thus, the pmf is always guaranteed to exist, and has the following properties:

$$0 \le p_X(x) \le 1, \quad x \in S,$$

and

$$\sum_{x \in S} p_X(x) = 1.$$

3.3.2 Cumulative Distribution Functions

Every real-valued random variable X (discrete or continuous) has an associated *cumulative distribution function* (*cdf*), $F_X(x)$, defined by

$$F_X(x) = \Pr\{X \le x\}, \; -\infty < x < \infty. \tag{3.10}$$

Note that the notation $\{X \le x\}$ is a shorthand notation for the event $\{\omega: X(\omega) \le x\}$. It should be pointed out that the technical definition of a random variable ensures that this set is actually an event (i.e., belongs to the underlying event set E). Thus, the cdf is always guaranteed to exist.

The cdf has the following properties:

(i) $0 \le F_X(x) \le 1, \; -\infty < x < \infty$.
(ii) $\lim_{x \to -\infty} F_X(x) = 0$ and $\lim_{x \to \infty} F_X(x) = 1$.
(iii) If $x_1 \le x_2$, then $F_X(x_1) \le F_X(x_2)$ (monotonicity).

Since $\{X \le x_1\}$ is contained in $\{X \le x_2\}$, this implies the formula

$$\Pr\{x_1 \le X \le x_2\} = F_X(x_2) - F_X(x_1), \quad \text{for any } x_1 \le x_2. \tag{3.11}$$

Property (iii) allows us to define the inverse distribution function, $F_X^{-1}(y)$, by

$$F_X^{-1}(y) = \min\{x: F_X(x) = y\}. \tag{3.12}$$

In words, since $F_X(x)$ may not be strictly increasing in x, $F_X^{-1}(y)$ is defined as the smallest value x, such that $F_X(x) = y$. The inverse distribution function is extensively used to generate realizations of random variables (see Chapter 4).

3.3.3 Probability Density Functions

If $F_X(x)$ is continuous and differentiable in x, then the associated *probability density function* (*pdf*), $f_X(x)$, is the derivative function

$$f_X(x) = \frac{d}{dx} F_X(x), \quad -\infty < x < \infty. \tag{3.13}$$

The pdf has the following properties for $-\infty < x < \infty$:

(i) $f_X(x) \geq 0$.

(ii) $F_X(x) = \int_{-\infty}^{x} f_X(x)\, dx$,

and in particular,

$$F_X(\infty) = \int_{-\infty}^{\infty} f_X(x)\, dx = 1.$$

Property (ii) implies the formula

$$\Pr\{x_1 \leq X \leq x_2\} = F_X(x_2) - F_X(x_1) = \int_{x_1}^{x_2} f_X(x)\, dx, \quad \text{for any } x_1 \leq x_2. \tag{3.14}$$

For a discrete random variable X, the associated pmf is sometimes referred to as a pdf as well. This identification is justified by the fact that a mathematical abstraction allows us, in fact, to define differencing as the discrete analog of differentiation. Indeed, for a discrete real-valued random variable X, we can write

$$F_X(x) = \sum_{y \leq x} f_X(y), \quad -\infty < x < \infty, \tag{3.15}$$

and each value, $f_X(x) = p_X(x)$, can be recovered by differencing in Eq. 3.15.

3.3.4 Joint Distributions

Let $X_1, X_2, \ldots, X_n$ be n real-valued random variables over a common probability space. The *joint cdf* of $X_1, X_2, \ldots, X_n$ is the function

$$F_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = \Pr\{X_1 \leq x_1, \ldots, X_n \leq x_n\}, \; -\infty < x_i < \infty, \quad i = 1, \ldots, n. \tag{3.16}$$

Similarly, the *joint pdf*, when it exists, is obtained by multiple partial differentiation,

$$f_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = \frac{\partial}{\partial x_1} \cdots \frac{\partial}{\partial x_n} F_{X_1,\ldots,X_n}(x_1, \ldots, x_n), \; -\infty < x_i < \infty, \quad i = 1, \ldots, n. \tag{3.17}$$

In this context, each cdf $F_{X_i}(x)$ and pdf $f_{X_i}(x)$ are commonly referred to as a *marginal distribution* and *marginal density*, respectively.

The random variables $X_1, X_2, \ldots, X_n$ are *mutually independent*, if

$$F_{X_1,\ldots,X_n}(x_1,\ \ldots,x_n) = \prod_{i=1}^{n} F_{X_i}(x_i),\ -\infty < x_i < \infty,\quad i = 1,\ \ldots,n \tag{3.18}$$

or equivalently

$$f_{X_1,\ldots,X_n}(x_1,\ \ldots,x_n) = \prod_{i=1}^{n} f_{X_i}(x_i),\ -\infty < x_i < \infty,\quad i = 1,\ \ldots,n, \tag{3.19}$$

provided that the densities exist. In other words, mutual independence is exhibited when joint distributions or densities factor out into their marginal components.

A set of random variables, $X_1, X_2,\ \ldots, X_n$, are said to be *iid* (*independently, identically distributed*), if they are mutually independent and each of them have the same marginal distribution.

3.4 EXPECTATIONS

The expectation of a random variable is a statistical operation that encapsulates the notion of "averaging." In other words, it assigns to a real-valued random variable, X, a number, $\mathrm{E}[X]$, called the *mean* or *expected value* or just the *expectation* of X. The expectation operation converts a random variable, X, to a deterministic scalar quantity, the mean value $\mathrm{E}[X]$, which can be thought of as a "central value" of X.

The mathematical definition of expectation varies according to the nature of the underlying state space. For a discrete random variable X with pmf $p_X(x)$, we define

$$\mathrm{E}[X] = \sum_{x \in S} x\ p_X(x), \tag{3.20}$$

and for a continuous random variable with pdf $f_X(x)$, we define

$$\mathrm{E}[X] = \int_{-\infty}^{\infty} x\ f_X(x)\ dx. \tag{3.21}$$

We mention that the expectations in Eqs. 3.20 and 3.21 are only defined when the corresponding sum or integral exist. In either case, the averaging action of the expectation yields a weighted sum or integral, where the weights are probabilities or densities.

Let X and Y be random variables, whose expectations exist, and let a and b be real numbers. Then,

$$\mathrm{E}[a\ X + b\ Y] = a\ \mathrm{E}[X] + b\ \mathrm{E}[Y]. \tag{3.22}$$

Equation 3.22 shows that expectation is linear.

3.5 MOMENTS

Moments are expectations of the powers of a random variable. They provide information on the underlying distribution, and are sometimes used as parameters of particular distribution functions. Mathematically, the k-th moment of X is given by

$$m_k = \mathrm{E}[X^k], \quad k = 1, 2, \ldots \tag{3.23}$$

Thus, for $k = 1, m_1 = \mathrm{E}[X]$ is just the mean. The second moment, $m_2 = \mathrm{E}[X^2]$, is used to define the *variance* of X

$$\mathrm{V}[X] = \mathrm{E}[X^2] - \mathrm{E}^2[X], \tag{3.24}$$

which measures the *variability* or *dispersion* of the random variable on the real line in units of X^2. Unlike expectation, the variance operation is not linear, since

$$\mathrm{V}[aX + bY] = a^2\ \mathrm{V}[X] + b^2\ \mathrm{V}[Y] + 2ab\ \mathrm{Cov}[X, Y], \tag{3.25}$$

where

$$\mathrm{Cov}[X, Y] = \mathrm{E}[XY] - \mathrm{E}[X]\ \mathrm{E}[Y] \tag{3.26}$$

is the *covariance* of X and Y. The covariance of two random variables is a measure of association, indicating how two random variables "vary together." This topic will be covered in greater detail in Section 3.6, where a more useful measure of association will be presented as a normalized covariance. For now, we just point out that (3.26) readily shows the nonlinearity of the covariance, since

$$\mathrm{Cov}[aX, bY] = ab\ \mathrm{Cov}[X, Y]. \tag{3.27}$$

An alternative measure of variability or dispersion is the *standard deviation* of X,

$$\sigma[X] = \sqrt{\mathrm{V}[X]} = \sqrt{\mathrm{E}[X^2] - \mathrm{E}^2[X]}, \tag{3.28}$$

which is expressed in units of X.

The *squared coefficient of variation* of X, is the statistic

$$\mathrm{c}^2[X] = \frac{\mathrm{V}[X]}{\mathrm{E}^2[X]}, \tag{3.29}$$

which is yet another measure of the variability or dispersion of X, this time normalized to a unitless quantity.

While the number of moments is infinite (though not all may exist), only the first few moments are considered in practice. In particular, the third moment influences the *skewness* (departure from symmetry) of the distribution of X via the *coefficient of skewness*

$$v[X] = \frac{\mathrm{E}[(X - \mathrm{E}[X])^3]}{\sigma^3[X]}, \tag{3.30}$$

which is negative, zero, or positive, according as the distribution is left-skewed, symmetric, or right-skewed, respectively. In a similar vein, the fourth moment influences the *kurtosis* of the distribution of X,

$$\mathrm{k}[X] = \frac{\mathrm{E}[(X - \mathrm{E}[X])^4]}{\sigma^4[X]} - 3, \tag{3.31}$$

which measures the degree of "fatness" of the distribution tail relative to a normal distribution with the same standard deviation (it is negative, zero, or positive, according as the distribution is less "fat," equally "fat," or more "fat," respectively).

Note that knowledge of a finite number of moments does not determine a distribution, except in special cases. Furthermore, in pathological cases, moments can be infinite or mathematically undefined, depending on the shape of the underlying distribution.

3.6 CORRELATIONS

Let X and Y be two real-valued random variables over a common probability space. It is sometimes necessary to obtain information on the nature of the *association* (probabilistic relation) between X and Y, beyond dependence or independence.

A useful measure of statistical association between X and Y is their *correlation coefficient* (often abbreviated to just *correlation*), defined by

$$\rho(X, Y) = \frac{\mathrm{E}[XY] - \mathrm{E}[X]\,\mathrm{E}[Y]}{\sigma[X]\,\sigma[Y]}, \tag{3.32}$$

which is well defined whenever the corresponding standard deviations exist and are finite. Note that the numerator of Eq. 3.32 is precisely $\mathrm{Cov}[X, Y]$. The division by the standard deviations normalizes the covariance into a correlation coefficient, so that it is invariant under scaling, that is,

$$\rho(a\,X, bY) = \rho(X, Y), \tag{3.33}$$

unlike its covariance counterpart (see Eq. 3.27).

The correlation coefficient has the following properties:

1. $-1 \leq \rho(X, Y) \leq 1$.
2. If X and Y are *independent* random variables, then X and Y are *uncorrelated*, that is $\rho(X, Y) = 0$. However, the converse is false, namely, X and Y may be *uncorrelated* and *dependent*, simultaneously.
3. If Y is a (deterministic) linear function of X, that is, $Y = aX + b$, then
   If $a > 0$, then $\rho(X, Y) = 1$.
   If $a < 0$, then $\rho(X, Y) = -1$.

Property (3) above provides a clue into the operational meaning of the correlation coefficient as a measure of *linear dependence* between X and Y. More specifically, $\rho(X, Y)$ measures the linear covariation of X and Y as described below.

First, if $\rho(X, Y) > 0$, then X and Y are *positively correlated* random variables in the sense that their realizations tend to behave as follows:

1. When $X(\omega)$ is a relatively *large* realization, then $Y(\omega)$ tends to be a comparatively *large* realization simultaneously.
2. When $X(\omega)$ is a relatively *small* realization, then $Y(\omega)$ tends to be a comparatively *small* realization simultaneously.
3. When multiple pairs $(X(\omega), Y(\omega))$ are plotted as a graph, the points tend to arrange themselves in a band of a positive slope. The higher the correlation, the narrower is the band, until it becomes a line with a positive slope for $\rho(X, Y) = 1$.

Second, if $\rho(X, Y) < 0$, then X and Y are *negatively correlated* random variables in the sense that their realizations tend to behave as follows:

1. When $X(\omega)$ is a relatively *large* realization, then $Y(\omega)$ tends to be a comparatively *small* realization simultaneously.
2. When $X(\omega)$ is a relatively *small* realization, then $Y(\omega)$ tends to be a comparatively *large* realization simultaneously.
3. When multiple pairs $(X(\omega), Y(\omega))$ are plotted as a graph, the points tend to arrange themselves in a band of a negative slope. The higher the correlation, the narrower is the band, until it becomes a line with a negative slope for $\rho(X, Y) = -1$.

Third, if $\rho(X, Y) = 0$, then X and Y are *uncorrelated* random variables in the sense that there is no apparent linear relation between realization pairs $X(\omega)$ and $Y(\omega)$. When multiple pairs $(X(\omega), Y(\omega))$ are plotted on as a graph, the points form a “blob” with no apparent “direction.”

Recall that correlation is a *weaker* concept than dependence, since it only measures linear dependence (X and Y may be related by another functional relation, e.g., quadratic). Still, linear dependence is a common instance of dependence, and is often taken as a *proxy* for dependence.

3.7 COMMON DISCRETE DISTRIBUTIONS

This section reviews the most commonly used discrete distributions and the underlying random experiment, and discusses their use in simulation modeling. For more information, see Bratley et al. (1987) or Law and Kelton (2000). We shall use indicator functions, defined for any set A by

$$1_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}. \tag{3.34}$$

We also routinely indicate that a random variable X has distribution D by the notation $X \sim \text{D}$.

3.7.1 Generic Discrete Distribution

A discrete random variable, X, corresponds to a trial (random experiment) with a countable (finite or infinite) number of distinct outcomes. Thus, its state space has the form $S = \{s_1, \ldots, s_i, \ldots\}$, where a generic state (realization), s_i, may be any symbol (it is common, however, to code the states by integers, when convenient). The generic *discrete distribution* is denoted by $\text{Disc}(\{(p_i, v_i): i = 1, 2, \ldots\})$,[1] where each parameter pair, (p_i, v_i), corresponds to $\Pr\{X = v_i\} = p_i$.

The pmf of $X \sim \text{Disc}(\{(p_i, v_i): i = 1, 2, \ldots\})$ is given by

[1] Note that while distribution names resemble those of Arena, the corresponding parameter definitions may differ from their Arena counterparts.

$$p_X(x) = \sum_{v_i \in S} 1_{\{v_i\}}(x)\, p_i = \begin{cases} p_i, & \text{if } x = v_i \text{ for some } i \\ 0, & \text{otherwise} \end{cases} \tag{3.35}$$

and for a real-valued state space, say $S = \{1, 2, \ldots\}$, the corresponding distribution function is given by

$$F_X(x) = \sum_{i=1}^{[x]} p_i = \begin{cases} 0, & \text{if } x < 1 \\ \sum_{i=1}^{k} p_i, & \text{if } k \le x < k+1 \end{cases} \tag{3.36}$$

where $[x]$ is the integral part of x.

The generic discrete distribution may be used to model a variety of situations, characterized by a discrete outcome. In fact, all other discrete distributions are simply useful specializations of the generic case.

3.7.2 Bernoulli Distribution

A *Bernoulli* random variable, X, corresponds to a trial with two possible outcomes: *success* or *failure*. Thus, its state space has the form $S = \{0, 1\}$, where state 0 codes for a *failure* realization and state 1 codes for a *success* realization. The Bernoulli distribution is denoted by Ber(p), where p represents the probability of success (and therefore, $1 - p$ is the probability of failure).

The pmf of $X \sim \text{Ber}(p)$ is

$$p_X(k) = \begin{cases} p, & k = 1 \\ 1 - p, & k = 0 \end{cases} \tag{3.37}$$

and the corresponding mean and variance are given by the formulas:

$$\text{E}[X] = p \tag{3.38}$$

and

$$\text{V}[X] = p(1 - p). \tag{3.39}$$

A Bernoulli random variable may be used to model whether a job departing from a machine is defective (failure) or not (success).

3.7.3 Binomial Distribution

A *binomial* random variable, $X = \sum_{k=1}^{n} X_k$, is the sum of n independent Bernoulli random variables, X_k, with a common success probability, p. Thus, its state space has the form $S = \{0, 1, \ldots, n\}$, and state k corresponds to a realization of k successes in n Bernoulli trials. The binomial distribution is denoted by B(n, p).

The pmf of $X \sim \text{B}(n, p)$ is

$$p_X(k) = \binom{n}{k} p^k (1 - p)^{n-k}, k = 0, 1, \ldots, n, \tag{3.40}$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, $n \geq k \geq 0$, and the corresponding mean and variance are given by the formulas

$$E[X] = np \tag{3.41}$$

and

$$V[X] = np(1-p). \tag{3.42}$$

A binomial random variable may be used to model the total number of defective items in a given batch. Such a binomial trial can be a much faster procedure than conducting multiple Bernoulli trials (for each item separately).

3.7.4 Geometric Distribution

A *geometric* random variable, X, is the number of Bernoulli trials to and including the first success. The geometric distribution is denoted by Ge(p), where p represents the probability of success (and therefore, $1 - p$ is the probability of failure). Since the number of trials is potentially unbounded, the state space becomes $S = \{1, 2, \ldots, k, \ldots\}$.

The pmf of $X \sim$ Ge(p) is

$$p_X(k) = (1-p)^{k-1}p, k = 1, 2, \ldots \tag{3.43}$$

and the corresponding mean and variance are given by the formulas

$$E[X] = \frac{1}{p} \tag{3.44}$$

and

$$V[X] = \frac{1-p}{p^2}. \tag{3.45}$$

A geometric random variable may be used to model the number of good product units, separating consecutive bad (defective) ones.

The geometric distribution is also widely used in mathematical models, because it often renders the analysis tractable. This tractability is due to the fact that the geometric distribution is the only discrete distribution with the so-called *memoryless property*, namely,

$$\Pr\{X > k + n | X > k\} = \Pr\{X > n\}, \quad \text{for all } k, n \geq 1. \tag{3.46}$$

This equation states that the probability that the remaining number of trials to the next success is independent of the number of trials elapsed since the previous success.

3.7.5 Poisson Distribution

A *Poisson* random variable, X, can be thought of as a generalization of a binomial random variable from discrete trials to continuous trials. It represents the total number

of successes as the limit of a sequence of binomial trials, in which n tends to infinity and p tends to 0, such that the product $np = \lambda$ is fixed and represents the rate of successes per time unit. The resulting Poisson random variable then represents the number of successes in a unit interval. Since the number of successes is potentially unbounded, the state space becomes $S = \{0, 1, \ldots, k, \ldots\}$. The Poisson distribution is denoted by Pois(λ).

The pmf of $X \sim$ Pois(λ) is

$$p_X(k) = \begin{cases} \frac{e^{-\lambda}\lambda^k}{k!}, & k = 0, 1, \ldots \\ 0, & \text{otherwise} \end{cases} \tag{3.47}$$

and the corresponding mean and variance are given by

$$\mathrm{E}[X] = \lambda \tag{3.48}$$

and

$$\mathrm{V}[X] = \lambda \tag{3.49}$$

A Poisson random variable is often used to model the number of random occurrences in a time interval. Examples include the number of machine failures in a time interval, number of customer demands in a time interval, and so on.

3.8 COMMON CONTINUOUS DISTRIBUTIONS

This section reviews the most commonly used continuous distributions and the underlying random experiment, and discusses their use in simulation modeling. For more information, see Bratley et al. (1987) or Law and Kelton (2000).

3.8.1 Uniform Distribution

A *uniform* random variable, X, assumes values in an interval $S = [a, b], b > a$, such that each value is equally likely. The uniform distribution is denoted by Unif(a, b), and is the simplest continuous distribution.

The pdf of $X \sim$ Unif(a, b) is

$$f_X(x) = \begin{cases} \frac{1}{b-a}, & \text{if } a \le x \le b \ , \\ 0, & \textit{otherwise} \end{cases} \tag{3.50}$$

and the cdf is

$$F_X(x) = \begin{cases} 0, & \text{if } x < a \\ \frac{x-a}{b-a}, & \text{if } a \le x \le b \\ 1, & \text{if } x > b. \end{cases} \tag{3.51}$$

The corresponding mean and variance are given by the formulas

$$\mathrm{E}[X] = \frac{a+b}{2} \tag{3.52}$$

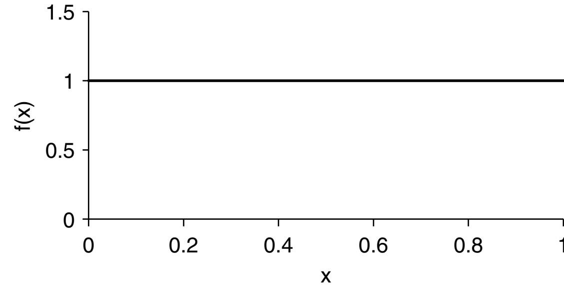


Figure 3.1 Density function of the Unif(0, 1) distribution.

and

$$\mathrm{V}[X] = \frac{(b-a)^2}{12}. \tag{3.53}$$

A graph of the pdf of a uniform distribution is depicted in Figure 3.1.

A uniform random variable is commonly employed in the absence of information on the underlying distribution being modeled.

3.8.2 Step Distribution

A *step* or *histogram* random variable, X, generalizes the uniform distribution in that it constitutes a *probabilistic mixture* of uniform random variables. The step distribution is denoted by $\mathrm{Cont}(\{(p_j, l_j, r_j): j = 1, 2, \ldots, J\})$, where the parameters have the following interpretation: $X \sim \mathrm{Unif}(l_j, r_j)$ with probability p_j, $j = 1, 2, \ldots, J$. Thus, the state space of X is the union of intervals,

$$S = \bigcup_{j=1}^{J} [l_j, r_j).$$

The pdf of $X \sim \mathrm{Cont}(\{(p_j, l_j, r_j): j = 1, 2, \ldots, J\})$ is given by

$$f_X(x) = \sum_{j=1}^{J} 1_{[l_j, r_j)}(x) \frac{p_j}{r_j - l_j} = \begin{cases} \frac{p_j}{r_j - l_j}, & \text{if } l_j \le x < r_j \\ 0, & \textit{otherwise} \end{cases} \tag{3.54}$$

Thus, the resulting pdf is a step function (mixture of uniform densities) as illustrated in by Figure 3.2, and the corresponding cdf is given by

$$F_X(x) = \begin{cases} 0, & \text{if } x < l_1 \\ \sum_{j=1}^{J} 1_{[l_j, r_j)}(x) \left[\sum_{i=1}^{j-1} p_i + (x - l_j) \frac{p_j}{r_j - l_j} \right], & \text{if } l_1 \le x < r_J \\ 1, & \text{if } x \ge r_J. \end{cases} \tag{3.55}$$

The corresponding mean and variance are given by the formulas

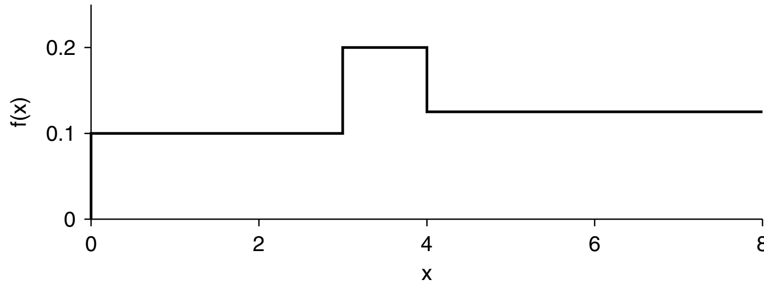


Figure 3.2 Density function of the Cont({(0.3, 0, 3), (0.2, 3, 4), (0.5, 4, 8)}) distribution.

$$E[X] = \sum_{j=1}^{J} p_j \frac{l_j + r_j}{2} \tag{3.56}$$

and

$$V[X] = \frac{1}{3}\sum_{j=1}^{J} p_j \left(l_j^2 + l_j r_j + r_j^2\right) - \frac{1}{4}\left(\sum_{j=1}^{J} p_j (r_j + l_j)\right)^2 \tag{3.57}$$

A step random variable is routinely used to model an empirical distribution, estimated by a histogram. Suppose the histogram has J cells. Then cell j coincides with the interval $[l_j, r_j)$, and the probability estimate (relative frequency) of the cell will be assigned as the value of the corresponding p_j.

3.8.3 Triangular Distribution

A *triangular* random variable, X, assumes values in an interval $S = [a, b]$, with the most "likely" value (the *mode*) being some point $c \in [a, b]$. The likelihood increases linearly in the subinterval $[a, c]$, and decreases linearly in the subinterval $[c, b]$, so that the density has a triangular shape (see Figure 3.3). The triangular distribution is denoted by Tria(a, c, b).

The pdf of $X \sim \text{Tria}(a, c, b)$ is

$$f_X(x) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)}, & \text{if } a \le x \le c \\ \frac{2(b-x)}{(b-a)(b-c)}, & \text{if } c \le x \le b \\ 0, & \textit{otherwise.} \end{cases} \tag{3.58}$$

The corresponding mean and variance are given by the formulas

$$E[X] = \frac{a + b + c}{3} \tag{3.59}$$

and

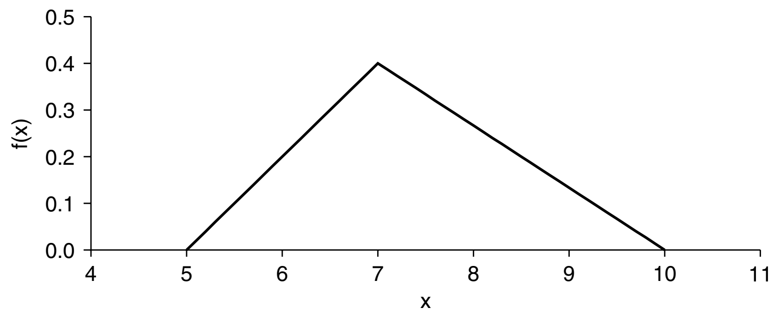


Figure 3.3 Density function of the Tria(5, 7, 10) distribution.

$$\mathrm{V}[X] = \frac{a^2 + b^2 + c^2 - ab - ac - bc}{18} \tag{3.60}$$

A triangular random variable is used when the underlying distribution is unknown, but it is reasonable to assume that the state space ranges from some minimal value, a, to some maximal value, b, with the most likely value being somewhere in between, at c. The choice of c then determines the skewness of the triangular distribution. The piecewise linear form of the pdf curve of Figure 3.3 is the simplest way to represent this kind of behavior.

3.8.4 Exponential Distribution

An *exponential* random variable, X, assumes values in the positive half-line $S = [0, \infty]$. The exponential distribution is denoted by Expo(λ), where λ is called the *rate* parameter.[2]

The pdf of $X \sim$ Expo(λ) is

$$f_X(x) = \lambda e^{-\lambda x}, \quad x \geq 0, \tag{3.61}$$

and the cdf is

$$F_X(x) = 1 - e^{-\lambda x}, \quad x \geq 0. \tag{3.62}$$

The corresponding mean and variance are given by the formulas

$$\mathrm{E}[X] = \frac{1}{\lambda} \tag{3.63}$$

and

$$\mathrm{V}[X] = \frac{1}{\lambda^2}. \tag{3.64}$$

A graph of the pdf of an exponential distribution is depicted in Figure 3.4.

[2] Note that in Arena, the corresponding parameter is the mean $1/\lambda$, rather than the rate λ.

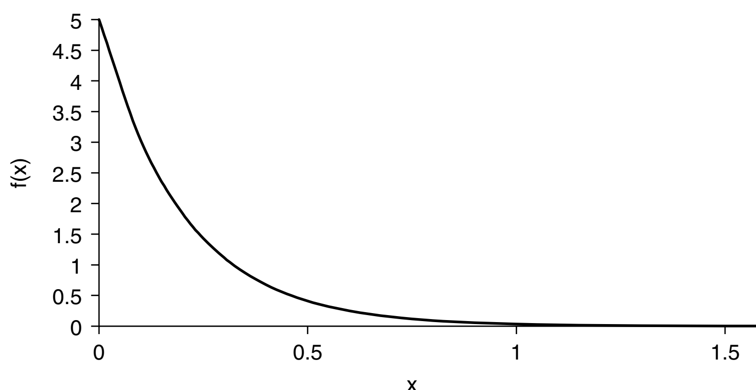


Figure 3.4 Density function of the Expo(5) distribution.

Exponential random variables are widely used to model "random" interarrival times in continuous time, especially when these are iid. Examples include customer interarrivals, times to failure, and so on.

The exponential distribution is also widely used in mathematical models, because it often renders the analysis tractable. This tractability is due to the fact that the exponential distribution is the only continuous distribution with the so-called *memoryless property*, namely,

$$\Pr\{X > s + t | X > s\} = \Pr\{X > t\}, \text{for all } s, t \geq 0. \tag{3.65}$$

The equation above states that the probability that the remaining time to the next arrival is independent of the time elapsed since the previous arrival. In fact, the exponential distribution constitutes a generalization of the geometric distribution to continuous time.

3.8.5 Normal Distribution

A *normal* random variable, X, can assume any value on the real line $S = (-\infty, \infty)$. The normal distribution is denoted by Norm(μ, σ^2), where μ is the mean (*scale* parameter) and σ^2 is the variance (*shape* parameter), and has the familiar bell shape (Figure 3.5), popularly known as the *bell curve*. In the technical literature, it is also known as the *gaussian* distribution, as a tribute to the mathematician Gauss. The special case Norm(0,1) is known as the *standard normal* distribution. Another transformation of normal random variables, implemented by Arena, results in the so-called *Johnson* distribution (see Kelton et. al. 1998).

The pdf of $X \sim$ Norm(μ, σ^2) is

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}}, \ -\infty < x \leq \infty. \tag{3.66}$$

The corresponding mean and variance are given by the formulas

$$\mathrm{E}[X] = \mu \tag{3.67}$$

and

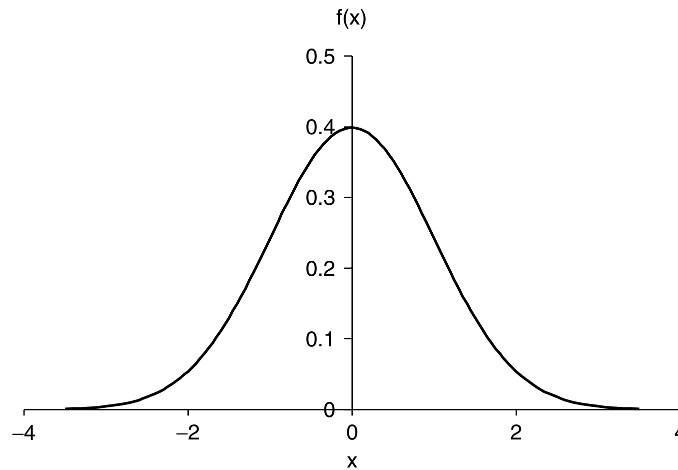


Figure 3.5 Density function of the Norm(0, 1) distribution.

$$\mathrm{V}[X] = \sigma^2. \tag{3.68}$$

A graph of the pdf of the standard normal distribution is depicted in Figure 3.5.

An important property of normal random variables is that they can always be standardized. This means that if $X \sim \mathrm{Norm}(\mu, \sigma^2)$, then

$$Z = \frac{X - \mu}{\sigma^2} \sim \mathrm{Norm}(0, 1)$$

is a standard normal random variable. Furthermore, if $X \sim \mathrm{Norm}(\mu_X, \sigma_X^2)$ and $Y \sim \mathrm{Norm}(\mu_Y, \sigma_Y^2)$ are independent normal variables, then

$$aX + bY \sim \mathrm{Norm}(a\,\mu_X + b\,\mu_Y, a^2\sigma_X^2 + b^2\sigma_Y^2),$$

which shows the linearity of normal distributions.

A normal random variable is used to model many random phenomena that can be expressed as sums of random variables, by virtue of the *central limit theorem*. This fundamental theorem asserts that the distribution of the sum approaches the normal distribution when the addends are iid (and in other cases as well).

The analyst should be careful in using normal distributions to model random phenomena, which cannot assume negative values (e.g., interarrival times). If the mean, μ, is large enough, then a negative value would be sampled relatively rarely, and may be simply ignored until further sampling yields a "legal" non-negative value. The analyst should be aware, however, that this procedure samples from a distribution that is no longer normal; rather, it is a normal distribution, conditioned on the outcome being non-negative.

3.8.6 Lognormal Distribution

A *lognormal* random variable, *X*, assumes values in the positive half-line $S = [0, \infty]$. The lognormal distribution is denoted by $\mathrm{Logn}(\mu, \sigma)$, where μ is a *scale* parameter and σ is a *shape* parameter.

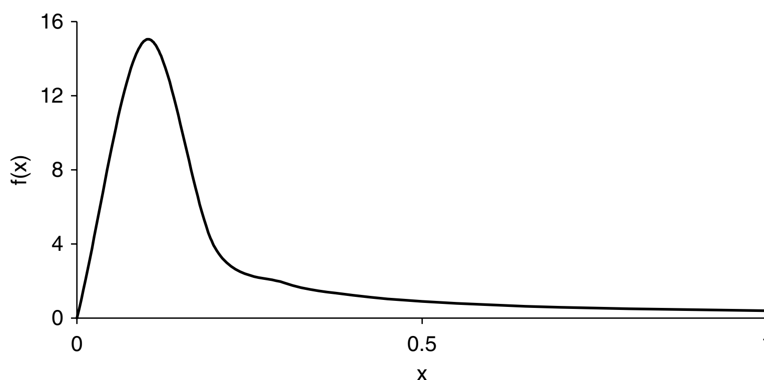


Figure 3.6 Density function of the Logn(0, 1) distribution.

The pdf of $X \sim \text{Logn}(\mu, \sigma)$ is

$$f_X(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\frac{(\ln x - \mu)^2}{2\sigma^2}}, \quad x \geq 0. \tag{3.69}$$

The corresponding mean and variance are given by the formulas

$$\text{E}[X] = e^{\mu + \sigma^2/2} \tag{3.70}$$

and

$$\text{V}[X] = e^{2\mu + \sigma^2}(e^{\sigma^2} - 1). \tag{3.71}$$

A graph of the pdf of a lognormal distribution is shown in Figure 3.6.

A lognormal random variable, X, can be represented as $X = e^Y$, where $Y \sim \text{Norm}(\mu, \sigma^2)$. It is always positive, and is often used in finance to model financial random processes.

3.8.7 Gamma Distribution

A *gamma* random variable, X, assumes values in the positive half-line $S = [0, \infty]$. The gamma distribution is denoted by *Gamm*(α, β), where $\alpha > 0$ is the *shape* parameter and $\beta > 0$ is the *scale* parameter.[3]

The pdf of $X \sim \text{Gamm}(\alpha, \beta)$ is

$$f_X(x) = \frac{x^{\alpha-1} e^{-x/\beta}}{\beta^\alpha \Gamma(\alpha)}, \quad x \geq 0, \tag{3.72}$$

where

$$\Gamma(\alpha) = \int_0^\infty y^{\alpha-1} e^{-y}\, dy \tag{3.73}$$

[3] Note that the gamma distribution in Arena has the parameters in reverse order.

is known as the *gamma function*. The corresponding mean and variance are given by the formulas

$$E[X] = \alpha\beta \tag{3.74}$$

and

$$V[X] = \alpha\beta^2. \tag{3.75}$$

Three graphs of the pdf of gamma distributions are depicted in Figure 3.7.

As the parameter names suggest, the gamma distribution is a parameterized family of distributions. A particular distribution can be selected with an appropriate choice of the shape and scale parameters. For example, for $\alpha = 1$ and $\beta = 1/\lambda$, we obtain the exponential distribution Expo(λ), since $\Gamma(1) = 1$. More generally, for integer $\alpha = k \geq 1$ and $\beta = 1/\lambda$, we obtain an *Erlang* distribution, denoted by Erl(k, λ),[4] and given by

$$f_X(x) = \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}, \quad x \geq 0. \tag{3.76}$$

The Erlang distribution is useful because an Erlang random variable can be represented as the sum of k iid exponential random variables, with a common rate, λ, and in particular, Erl($1, \lambda$) = Expo(λ). An Erlang random variable is useful in modeling multiple exponential "phases" with a common rate. For example, the model of a manufacturing subsystem, where products are serially processed without waiting in k processes with common processing rate λ, can be equivalently aggregated into one process with service distribution Erl(k, λ).

Another useful specialization is obtained for $\alpha = n/2$ (n even) and $\beta = 2$, which is called the *chi-square* distribution with n degrees of freedom, and denoted by $\chi^2(n)$. A $\chi^2(n)$ distributed random variable, X, can be represented as a sum

$$X = \sum_{i=1}^{n} Y_i^2$$

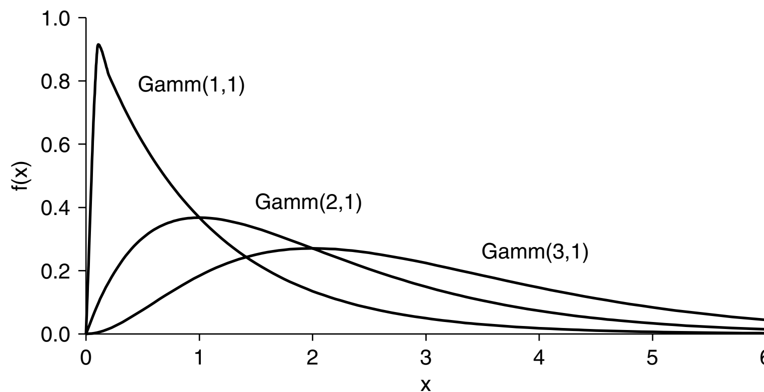


Figure 3.7 Density functions of the Gamm(1, 1), Gamm(2, 1), and Gamm(3, 1) distributions.

[4] Note that the Erlang distribution in Arena is represented with slightly different parameters.

of n independent squared standard normal random variables, Y_i. The class of *chi-square* distributed random variables has extensive applications in statistics.

3.8.8 Student's t Distribution

A *Student's t* random variable, X, (*t* random variable, for short) can assume any value on the real line $S = (-\infty, \infty)$. The t distribution is denoted by t(n), where the n parameter is the *number of degrees of freedom*.

The pdf of $X \sim \mathrm{t}(n),\ n > 2$, is

$$f_X(x) = \frac{\Gamma((n+1)/2)}{\sqrt{\pi n}\ \Gamma(n/2)} \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}, \quad -\infty \leq x \leq \infty \tag{3.77}$$

where Γ is the gamma function of Eq. 3.73. The corresponding mean and variance are given by the formulas

$$\mathrm{E}[X] = 0 \tag{3.79}$$

and

$$\mathrm{V}[X] = \frac{n}{n-2}. \tag{3.80}$$

A graph of the pdf of a Student's t distribution is depicted in Figure 3.8.

A t(n) distributed random variable X can be represented as

$$X = \frac{Z}{\sqrt{Y/n}}, \tag{3.81}$$

where $Z \sim \mathrm{Norm}(0, 1)$ is a standard normal random variable, $Y \sim \chi^2(n)$ is a chi-square random variable with n degrees of freedom, and Z and Y are independent. As can be seen in Figure 3.8, t(n) distributions have a functional form similar to that of the standard normal distribution, Norm(0, 1), but with "fatter" tails, which give rise to larger variances as indicated by Eq. 3.80. However, as the degrees-of-freedom parameter, n, tends to infinity, the t(n) distribution converges to Norm(0, 1).

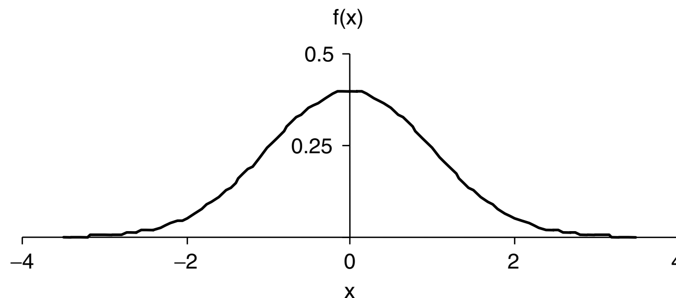


Figure 3.8 Density function of the t(10) distribution.

3.8.9 F Distribution

An F random variable, X, assumes values in the positive half-line $S = [0, \infty]$. The F distribution is denoted by $\mathrm{F}(n_1, n_2)$, where n_1 and n_2 are the *degrees of freedom* parameters.

The pdf of $X \sim \mathrm{F}(n_1, n_2)$ is

$$f_X(x) = \frac{\Gamma((n_1+n_2)/2)}{\Gamma(n_1/2)\Gamma(n_2/2)}\left[\frac{n_1}{n_2}\right]^{n_1/2}\frac{x^{(n_1/2)-1}}{\left[1+\frac{n_1}{n_2}x\right]^{(n_1+n_2)/2}},\quad 0 \le x \le \infty \tag{3.82}$$

where Γ is the gamma function of Eq. 3.73. The corresponding mean and variance are given by the formulas

$$\mathrm{E}[\boldsymbol{X}] = \frac{n_2}{n_2-2} \quad (\text{for } n_2 > 2) \tag{3.83}$$

and

$$\mathrm{V}[\boldsymbol{X}] = \frac{2n_2^2(n_1+n_2-2)}{n_1(n_2-4)(n_2-2)} \quad (\text{for } n_2 > 4). \tag{3.84}$$

An $\mathrm{F}(n_1, n_2)$ density is depicted in Figure 3.9.

An $\mathrm{F}(n_1, n_2)$ distributed random variable X can be represented as

$$X = \frac{V/n_1}{W/n_2}, \tag{3.85}$$

where $V \sim \chi^2(n_1)$ and $W \sim \chi^2(n_2)$ are independent chi-square random variables with the corresponding degrees of freedom. The $\mathrm{F}(n_1, n_2)$ distribution is skewed to the right, but it becomes less skewed as the degrees-of-freedom parameters, n_1 and n_2, increase in magnitude.

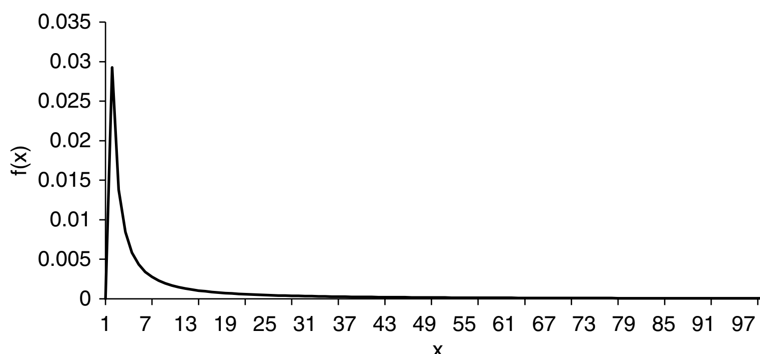


Figure 3.9 Density function of the F(1, 1) distribution.

3.8.10 Beta Distribution

A *beta* random variable, X, assumes values in the unit interval $S = [0, 1]$, although it may be scaled and shifted to any interval. The beta distribution is denoted by Beta(α, β), where $\alpha > 0$ and $\beta > 0$ are two *shape* parameters.[5]

The pdf of $X \sim$ Beta(α, β) is

$$f_X(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}, \quad 0 \leq x \leq 1, \tag{3.86}$$

where

$$B(\alpha, \beta) = \int_0^{\infty} y^{\alpha-1}(1-y)^{\beta-1} dy = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{3.87}$$

is known as the *beta* function, and is defined in terms of the gamma function of Eq. 3.73. The corresponding mean and variance are given, respectively, by the formulas

$$\mathrm{E}[X] = \frac{\alpha}{\alpha+\beta} \tag{3.88}$$

and

$$\mathrm{V}[X] = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}. \tag{3.89}$$

Three graphs of the pdf of beta distributions are depicted in Figure 3.10.

A beta random variable is often used in statistics to model an unknown probability, regarded as a random variable.

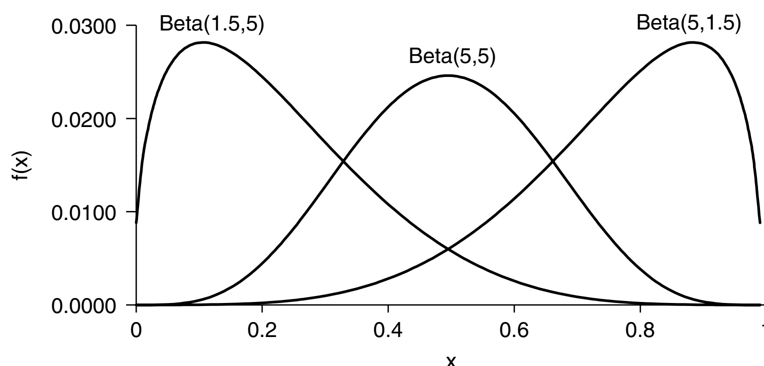


Figure 3.10 Density functions of the Beta(1.5, 5), Beta(5, 5), and Beta(5, 1.5) distributions.

[5] Note that the beta distribution in Arena has the parameters in reverse order.

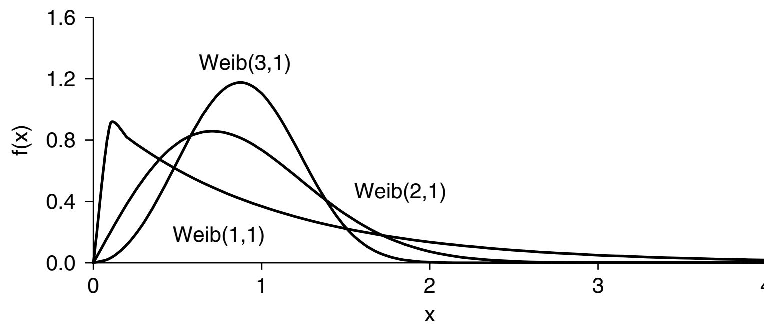


Figure 3.11 Density functions of the Weib(1, 1), Weib(2, 1), and Weib(3, 1) distributions.

3.8.11 Weibull Distribution

A *Weibull* random variable, X, assumes values in the positive half-line $S = [0, \infty]$. The Weibull distribution is denoted by Weib(α, β), where $\alpha > 0$ is the *shape* parameter and $\beta > 0$ is the *scale* parameter.[6]

The pdf of $X \sim$ Weib(α, β) is

$$f_X(x) = \frac{\alpha}{\beta^\alpha} x^{\alpha-1} e^{-(x/\beta)^\alpha}, \quad x \geq 0. \tag{3.90}$$

The corresponding mean and variance, respectively, are given by the formulas

$$\mathrm{E}[X] = \alpha \Gamma(1/\beta + 1) \tag{3.91}$$

and

$$\mathrm{V}[X] = \alpha^2 [\Gamma(2/\beta + 1) - \Gamma^2(1/\beta + 1)] \tag{3.92}$$

in terms of the gamma function of Eq. 3.73. Three graphs of the pdf of Weibull distributions are depicted in Figure 3.11.

The Weibull distribution is a parametric family of distributions. For $\alpha = 1$ and $\beta = 1/\lambda$, it becomes the exponential Expo(λ) distribution, while for $\alpha = 2$, it becomes the *Rayleigh* distribution (often used in artillery trajectory computations). Weibull random variables are often used in modeling the aging process of components in reliability analysis.

3.9 STOCHASTIC PROCESSES

A *stochastic process* is a time-indexed set of random variables, $\{X_t\}_{t \in T}$, with a common state space S, over a common probability space. The associated probability measure is called the *probability law* of the process. The time set, T, can be discrete or continuous, typically of the form $T = \{0, 1, \ldots, n, \ldots\}$ or $T = [0, \Theta]$, where Θ is either finite or infinite. For example, X_t may model the inventory level of a particular product in a warehouse at time t.

---

[6] Note that the Weibull distribution in Arena has the parameters in reverse order.

Stochastic processes are widely used to model random phenomena that evolve in time, such as arrival streams, service times, and routing decisions, to name but a few. In fact, simulation runs (replications) typically generate extensive realizations of multiple interacting stochastic processes. A realization of a stochastic process is also called a *sample path*, because it represents a possible history of its constituent time-indexed random variables. Most processes generated by simulation, but by no means all, are *stationary processes*, that is, their joint distributions (of any dimension) do *not* change in time.

The *autocorrelation function* of a stochastic process is the correlation coefficient of its lagged random variables,

$$\rho(\tau, \delta) = \frac{\mathrm{E}[X_\tau X_{\tau+\delta}] - \mathrm{E}[X_\tau]\mathrm{E}[X_{\tau+\delta}]}{\sigma[X_\tau]\sigma[X_{\tau+\delta}]}, \quad \tau \in T, \delta \geq 0. \tag{3.93}$$

For stationary processes, the autocorrelation function depends only on the first argument, τ. The autocorrelation function is often used as a convenient proxy for temporal dependence in stochastic processes.

The next few subsections discuss several stochastic processes, commonly used in simulation. Generation of their sample paths is discussed in Chapter 4. For further reading on stochastic processes, we recommend that the reader to refer to Ross (1993), and Taylor and Karlin (1984).

3.9.1 Iid Processes

Independent identically distributed (*iid*) processes have the simplest possible probability law, since all random variables indexed by its time set are mutually independent and share a common marginal distribution. This means that iid processes do not have *temporal dependence* (*time dependence*) in the sense that their "past" is always probabilistically irrelevant to their "future."

Iid processes are extensively used in simulation modeling, when justified by modeling considerations, or as a simplifying assumption in the absence of additional information. Typical examples are arrival processes, whose interarrival times are modeled as iid random variables or times to failure in a machine, which are often assumed to be iid.

3.9.2 Poisson Processes

A *Poisson process* $\{K_t\}_{t \geq 0}$ is a *counting process*, that is, it has state space $S = \{0, 1, \ldots\}$, continuous time set T, and nondecreasing sample paths; however, count increments may not exceed 1 (multiple simultaneous arrivals are not allowed). A random variable K_t from a Poisson process represents the (cumulative) *count* of some abstract "arrivals"; the last term actually connotes any phenomenon that can be declared to take place at discrete time points (e.g., job arrivals, failures, etc.). The distinguishing feature of any Poisson process is the *independent increment property*, which in its simplest form states that

$$\Pr\{K_{t+u} - K_t | K_s, s \leq t\} = \Pr\{K_{t+u} - K_t\}, \quad \text{for all } t, u \geq 0. \tag{3.94}$$

In words, a count increment in a future interval is independent of any past counts. It can be shown that this property alone forces the Poisson process to have a specific count increment distribution, and a specific interarrival distribution as follows:

1. Any count increment of the form $K_{t+u} - K_t$, over the interval $[t, t+u]$ of length u, has the Poisson distribution Pois(λu), for some $\lambda > 0$.
2. The interarrival times between successive arrivals are iid exponential with the aforementioned parameter, λ, that is, their distribution is Expo(λ).

In fact, conditions 1 and 2 are equivalent characterizations of the Poisson process. The parameter λ is the arrival rate of the Poisson process (expected number of arrivals per time unit).

The following operations on Poisson processes result in new Poisson processes (*closure* properties):

1. The *superposition* of independent Poisson processes (merging all their arrival points along the timeline) results in a new Poisson process. More specifically, if $\{K_t\}_{t \geq 0}$ and $\{L_t\}_{t \geq 0}$ are independent Poisson processes, with respective arrival rates λ_K and λ_L, then the superposition process, $\{K_t + L_t\}_{t \geq 0}$ is a Poisson process of rate $\lambda_K + \lambda_L$.
2. The *thinning* of a Poisson process (random deletion of its arrival points) results in a new Poisson process. More specifically, if $\{K_t\}_{t \geq 0}$ is a Poisson process of rate λ_K, from which arrivals are deleted according to independent Bernoulli trials with probability $1 - p$, then the thinned process, $\{L_t\}_{t \geq 0}$ is a Poisson process of rate $\lambda_L = p\lambda_K$.

The simplicity of Poisson processes and their closure properties render them a popular traffic model in network systems, because traffic merging and thinning of Poisson processes (by splitting such a stream into substreams) result in new Poisson processes. Moreover, Poisson processes have been widely used to model external arrivals to a variety of systems, where arriving customers make "independent arrival decisions." For example, telephone customers do not normally "coordinate" their phone calls, and customer demand arrivals are usually independent of each other. In these cases, the Poisson process assumption on the respective arrival processes may well be justified.

3.9.3 Regenerative (Renewal) Processes

A stochastic process $\{X_\tau : \tau \geq 0\}$ (discrete time or continuous time) is characterized as *regenerative* or *renewal* if it has (usually random) time points $T_1, T_2, \ldots$, such that the partial process histories $\{X_\tau : T_j \leq \tau < T_{j+1}\}$ over the intervals $[T_j, T_{j+1})$ are iid. In other words, the partial histories are independent statistical replicas of each other. For this reason, the time points $T_1, T_2, \ldots$ are referred to as *regeneration points* or *renewal points*, because they identify when the underlying process "renews" or "regenerates" itself statistically.

3.9.4 Markov Processes

Markov processes form the simplest class of dependent stochastic processes, with dependence extending only to the most "recent" past information. Formally,

$\{M_t\}_{t \geq 0}$ is a Markov process, if for all events $\{M_u \in A\}$, it satisfies the Markovian condition

$$\Pr\{M_u \in A | M_s : s \leq t\} = \Pr\{M_u \in A | M_t\} \text{ for all } 0 \leq t \leq u. \tag{3.95}$$

The probability law 3.95 is relatively simple. It stipulates that the probability of a future event probability $\Pr\{M_u \in A\}$ conditioned on past random variables $\{M_s : s \leq t\}$ (history) before u, equals the probability of the same future event, conditioned only on the most recent random variable, M_t. This means that in Markov processes, knowledge of information strictly predating some time t is immaterial for event probabilities after t.

The relative simplicity of Markov processes renders them popular models in analysis as well, without sacrificing the feature of temporal dependence. For example, discrete-time Markov processes with a discrete space S, known as *Markov chains*, are particularly simple. For a Markov chain, Eq. 3.95 becomes a matrix $Q = [q_{i,j}]$, called the *transition probability matrix*, where

$$q_{i,j} = \Pr\{M_{k+1} = j | M_k = i\} \text{ for any pair of states } i,j \in S. \tag{3.96}$$

The statistics of Markov chains can then be computed using matrix calculus.

Discrete-state Markov processes in continuous time are often classified as *jump processes*, because their sample paths have the form of step functions, whose discontinuities (jumps) correspond to state transitions. Markov jump processes have a simple structure that facilitates their generation:

1. Jumps are governed by transition probabilities similar to Eq. 3.96. The sequence of states visited by jumps is called the *jump chain*.
2. The time elapsed in state i is distributed exponentially with parameter λ_i, which depends only on state i but *not* on the state transitioned to.

As an example, consider an M/M/1 queue (iid exponential interarrival and service times), with the Markovian state being the number of customers in the system. The state jumps up and down following customer arrivals and service completions, respectively, and is otherwise constant.

A useful generalization of this kind of Markov jump processes is the class of *Markov renewal processes*. Here, the step-function form of sample paths is retained, as well as the Markovian structure of the jump chain. However, the times separating jumps can have a general (not necessarily exponential) distribution, and the transition probabilities of the time intervals separating successive jumps depend not only on the state jumped from, but also on the state jumped to. For more details, see Çinlar (1975).

3.10 ESTIMATION

An *estimator* is a random statistic, namely, a function of some observed random sample of data. A value of the estimator is called an *estimate* (usually the estimated quantity is some unknown parameter value). Note that an estimator is a random variable, while an estimate is one of its realizations. Good estimators are *unbiased*, that is, as the sample size grows to infinity, the expectations of such estimators (which are random variables) converge to the true parameter value, whatever it is.

Moments and related statistics are routinely estimated from sample data, using statistical estimators. Consider a (finite) sample $\vec{Y} = \{Y_1, Y_2, \ldots, Y_N\}$, where the random observations $\{Y_j,\ j = 1, \ldots, N\}$ have a common distribution with mean μ and variance σ^2. An unbiased estimator for the mean, μ, based on the sample $\vec{Y}$, is the *sample mean*

$$\bar{Y} = \frac{1}{N}\sum_{i=1}^{N} Y_i. \tag{3.97}$$

An unbiased estimator of the variance, σ^2, based on the sample $\vec{Y}$, is the *sample variance*

$$S_Y^2 = \frac{1}{N-1}\sum_{i=1}^{N} [Y_i - \bar{Y}]^2, \tag{3.98}$$

whereas the *sample standard deviation*, S_Y, is just the square root of S_Y^2.

For a continuous-time history, $\vec{X} = \{X_t: A \le t \le B\}$, the *sample time average* is

$$\overline{X} = \frac{1}{B-A}\int_A^B X_t\, dt. \tag{3.99}$$

For a sample of pairs $\vec{Z} = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_N, Y_N)\}$, with common joint distributions for all pairs (X_j, Y_j), $j = 1, \ldots, N$, a common estimator of the correlation coefficient, $\rho(X, Y)$, is the *sample correlation coefficient*

$$r(X, Y) = \frac{\sum_{i=1}^{N} (X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{N} (X_i - \overline{X})^2} \times \sqrt{\sum_{i=1}^{N} (Y_i - \overline{Y})^2}}. \tag{3.100}$$

Note that all the estimators above are *point estimators*, that is, they provide a scalar estimate of some unknown parameter. In addition, an estimate of an interval in which the unknown parameter lies can also be established. Specifically, let C be some estimator of an unknown parameter θ. A $(1 - \alpha)$ *confidence interval* for θ is a random interval of the form $[A, B]$, such that $\Pr\{A \le \theta \le B\} = 1 - \alpha$. In other words, a confidence interval $[A,\ B]$ contains the unknown parameter θ with probability $(1 - \alpha)$. Usually, the confidence interval is of the form $[A, B] = [C - D_1, C + D_2]$ for some random offsets, D_1 and D_2.

3.11 HYPOTHESIS TESTING

Hypothesis testing is statistical decision making. The modeler formulates two complementary hypotheses, called the *null hypothesis* (denoted by H_0) and the *alternative hypothesis* (denoted by H_1). A decision is traditionally applied to the null hypothesis, which is either *accepted* or *rejected*. Consequently, two types of errors are possible:

Type I: Rejecting H_0 erroneously

Type II: Accepting H_0 erroneously

The goal of hypothesis testing is to reject (or accept) H_0, such that if H_0 is in fact true, then the probability of erroneously rejecting it (type I error) does not exceed some prescribed probability, α, called the *confidence level* or *significance level*. The smaller is α, the higher is the confidence in a corresponding rejection decision.

For example, suppose we wish to compare the failure rates δ_1 and δ_2 of machines 1 and 2, respectively, at significance level $\alpha = 0.05$. The associated hypotheses follow:

$$\begin{cases} H_0: & \delta_1 \leq \delta_2 \\ H_1: & \delta_1 > \delta_2. \end{cases}$$

The modeler then forms a *test statistic*, T, from some observed sample data with a known distribution under the null hypothesis, H_0. In our case, the statistic might be the difference of two failure rate estimates based on some failure data. The state space S of T is then partitioned into two disjoint regions $S = R_0 \cup R_1$, where R_0 is the *acceptance region*, and R_1 is the *rejection region*, such that the probability of type I error does not exceed, say, $\alpha = 0.05$. In practice, the analyst computes a realization t of T and decides to accept or reject H_0, according as t fell in region R_0 or R_1, respectively. A *critical value*, c, which depends on the significance level and the test statistic, is often used to separate the acceptance and rejection regions. We have noted that intervals comprising individual regions are often constructed as confidence intervals at the corresponding confidence levels.

An alternative approach to hypothesis testing is to compute the *probability value* (commonly abbreviated to *p-value*) of the realization of the test statistic, t, where p is the smallest significance level,

$$\alpha_{\min} = p, \tag{3.101}$$

for which the computed test statistic, t, can be rejected (often p is computed as $p = \Pr\{T > t\}$). To understand this concept, note that in hypothesis testing, we first fix α (and therefore the critical values that define the acceptance and rejection regions), and then decide whether to accept or reject, depending on the region in which the test statistic t fell. We reject the null hypothesis when $p \leq \alpha$, and accept it when $p > \alpha$. Conversely, suppose we do not fix α before computing the test statistic t, but allow it to "float." We would like to know how small can α be made and still permit the null hypothesis to be rejected. This means that we seek the smallest α that satisfies $p \leq \alpha$. Clearly, the requisite $\alpha = \alpha_{\min}$ is given by Eq. 3.101.

The p-value contains a considerable amount of information on the *quality* of our test decision. Not only can we decide whether to reject the null hypothesis or accept it, but we can also obtain an idea on how "strongly" we reject or accept it. The smaller p is compared to α, the stronger is its rejection; conversely, the larger p is relative to α, the stronger is its acceptance. For this reason the p-value is also called the *observed level* of the test. For more information on statistical issues in estimation and hypothesis testing, see Hoel et al. (1971b) and Devore (1991).

EXERCISES

1. Let X be a random variable uniformly distributed over the interval [2, 6], and let Y be a random variable distributed according to Tria(2, 3, 7).

a. Write the pdf and cdf of each of these random variables.
b. Compute their means and variances.

2. Suppose that X is distributed according to Unif(10, B), for some $B > 10$.
   a. How does the squared coefficient of variation vary as B increases? Show this behavior in a graph.
   b. Repeat the same investigation for X distributed according to Tria(1, 4, B), and compare the corresponding graph to the previous one.
3. Let X be a discrete random variable with pmf

$$p_X(x) = \begin{cases} 0.2, & x = 1 \\ 0.3, & x = 2 \\ 0.5, & x = 4 \end{cases}$$

and let Y be a continuous random variable with pdf

$$f_Y(y) = \begin{cases} 0, & y < 0 \\ 0.2y, & 0 \le y \le 1 \\ 0.1 + 0.1y, & 1 < y \le 2 \\ 0.25 + 0.025y, & 2 < y \le 4 \\ 0, & y > 4. \end{cases}$$

   a. Compute the mean, variance, and the squared coefficient of variation of X.
   b. Show that $f_Y(y)$ above is a legitimate pdf, and compute its mean, variance, and squared coefficient of variation.

$X \setminus Y$	0	1	2	3
0	0	0.05	0.0625	0.02
1	0.125	0.25	0.02	0
2	0.03	0.05	0	0.125
3	0.1	0.0625	0.055	0.05

4. Consider random variables X and Y with a joint pmf, given in the following table.
   a. Compute $p_X(x)$ and $p_Y(x)$, namely, the marginal pmf of X and Y.
   b. Compute the mean, variance, and squared coefficient of variation of X.
   c. Are X and Y independent?
   d. Compute the conditional expectations $E[X|Y = 1]$ and $E[X|Y > 1]$.
   e. Compute the conditional expectation $E[X + 2Y|X > 1, Y < 2]$.
5. Let the joint pdf of two continuous random variables X and Y be given by

$$f_{X,Y}(x, y) = \left[\frac{a}{b}\right]^2 e^{-(a/b)y}, \quad 0 < x < y,$$

where a and b are two non-zero constants. Compute the following statistics in terms of a and b:

$$f_X(x),\ f_Y(y),\ E[X],\ E[Y],\ V[X],\ V[Y],\ c^2[X],\ c^2[Y],\ \mathrm{Cov}[X, Y],\ \rho(X, Y).$$

This page intentionally left blank

Chapter 4

Random Number and Variate Generation

Random numbers are used in simulation to sample realizations of random variables with prescribed distributions, as well as stochastic processes with prescribed probability laws (see Chapter 3). For instance, customer arrivals are often generated according to a Poisson process, namely, the interarrival times are iid exponential. (Equivalently, the number of arrivals obeys the appropriate Poisson distribution, but it is far more convenient to generate interarrival times in the simulation run.)

Random numbers are generally produced via a *random number generator* (*RNG*) procedure (*generator*, for short), used to generate iid numbers that are uniformly distributed between 0 and 1. These numbers are often further transformed so as to conform to a prescribed distribution (see Section 4.2). Although we use the term "random numbers" in the RNG term, it should be pointed out that the numbers generated are not truly random. For one thing, an RNG is a deterministic procedure whose generated number stream can always be recreated. An RNG is "random" in the sense that its random number sequence passes statistical tests for randomness, in this case the uniformity of the generated numbers and their mutual independence. For this reason, RNGs are sometimes referred to as *pseudo* RNGs, but for all practical purposes an RNG can be thought of as a stochastic sequence, $\{U_n\}$, of iid random variables, such that $U_n \sim \text{Unif}(0, 1)$.

In simulation, an RNG is implemented as a computer algorithm in some programming language, and is made available to the user via procedure calls or icons. A common family of RNG algorithms is the class of *congruential* generators (see Law and Kelton 2000, and Banks et al. 1999). A *linear congruential* generator is a recursive scheme with integer constituents of the form

$$x_i = [ax_{i-1} + c](\text{mod } m), \ i = 1, 2, \ldots \tag{4.1}$$

where the n (mod m) operation returns the remainder of n/m. The initial value, x_0, called the *seed* (or *initial seed*), must be chosen with care and is provided as part of the generator. Equation 4.1 produces integers in the range $\{0, 1, \ldots, m-1\}$. To convert the random number sequence $\{x_n\}$ to a standard RNG output $\{u_n\}$, in the range [0, 1), the transformation $u_i = x_i/m$ is used. From this construction, it should be clear that the

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

Table 4.1
Random numbers generated by a linear congruential RNG with $a = 5, c = 7, m = 8,$ and $x_0 = 3$

Step i	x_i	u_i
0	3	
1	6	0.750
2	5	0.625
3	0	0.000
4	7	0.875
5	2	0.250
6	1	0.125
7	4	0.500
8	3	0.375
9	6	0.750
10	5	0.625

number of distinct random numbers in the sequence $\{u_n\}$ is, in fact, finite. Therefore, the sequence will eventually repeat itself and generate identical subsequences, whose size is called the *period* (of the RNG). A good RNG should also have as long a period as possible, in addition to passing the aforementioned statistical tests for randomness. Note that repeating random numbers introduce (usually unwanted) dependence into the stream, but if the period is long enough, then that dependence may never be experienced in a run or become acceptably "weak." An example of a random number sequence is displayed in Table 4.1.

Table 4.1 shows that the numbers start repeating at step 8. Clearly, a period of 8 (the maximal period possible for $m = 8$) is obviously far too short for this RNG to be acceptable in practical simulation. Thus, the RNG of Table 4.1 is a poor one and should not be used in practice. Rather, the parameters a, c, m, and the initial seed, x_0, should be selected so as to maximize the period.

4.1 VARIATE AND PROCESS GENERATION

Suppose we have at our disposal an RNG; equivalently, we may assume the availability of an iid sequence of variates, $\{U_n\}$, such that $\{U_n\} \sim \text{Unif}(0, 1)$. Recall that the term "variate" refers to a random variable in simulation context.

The underlying RNG is used as raw material to generate a prescribed stochastic process (variate sequence), which model a random component of some simulated system (e.g., random arrivals, random services, random routing, etc.). In other words, we are given some probability law (see Chapter 3), and we wish to generate a variate sequence that conforms to that law.

The broad problem of variate generation can be classified into three cases. These are enumerated below in order of increasing complexity:

- The *single-variate* generation problem calls for generating a single variate, X, from a prescribed distribution, G.
- The *iid process* generation problem calls for generating an iid sequence of variates, $\{X_n\}$, with a common distribution, G.

- The *non-iid process* generation problem calls for generating a sequence of variates, $\{X_n\}$, from some general probability law L. The generated sequence may generally be correlated or dependent.

For more information, see Dagpunar (1988). We next address each generation problem in turn.

4.2 VARIATE GENERATION USING THE INVERSE TRANSFORM METHOD

The single-variate generation problem seeks to transform an iid uniform sequence of variates $\{U_n\}$ into an iid sequence of variates $\{X_n\}$, such that $X_n \sim G$. Furthermore, it is desirable that the generation method be general, namely, applicable to any requisite distribution, G.

The most general and widely used method is the *Inverse Transform* method [see Law and Kelton (2000), and Banks *et al.* (1999)]. It relies on the following properties of the cdf $F_X(x)$ of a variate X, and their relation to a uniform variate $U \sim \text{Unif}(0, 1)$.

Evaluating the distribution function $F_X(x)$ at the underlying variate X results in a variate U, given by

$$U = F_X(X), \tag{4.2}$$

whose cdf is uniform between 0 and 1.

Conversely, evaluating an inverse distribution function $F_X^{-1}(u)$ at a uniform variate $U \sim \text{Unif}(0, 1)$ results in a variate

$$X = F_X^{-1}(U) \tag{4.3}$$

with cdf $F_X(x)$. Note that since every cdf, $F_X(x)$, is nondecreasing (from 0 to 1), it follows that its inverse, $F_X^{-1}(u)$, is always well defined.

For a given cdf $F_X(x)$, Eq. 4.3 implicitly describes the generation of the underlying variate, X, via the Inverse Transform method as a two-step algorithm:

1. Use your favorite RNG to generate a realization u from a variate $U \sim \text{Unif}(0, 1)$.
2. Compute $x = F_X^{-1}(u)$ as a realization of X.

Figure 4.1 illustrates graphically an application of the Inverse Transform method.

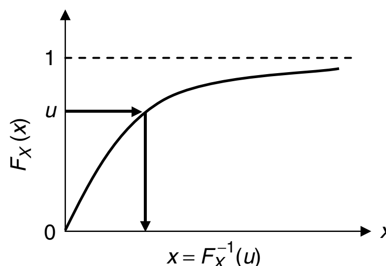


Figure 4.1 The Inverse Transform method.

We now proceed to demonstrate the use of the Inverse Transform method through several examples. We mention that Arena/SIMAN supports variate generation from selected distributions (see Section 7.4). We also point out that Arena samples values in an intelligent way. For example, when sampling in the context of interarrival or service times (which cannot be negative), Arena will discard any sampled negative values and will continue sampling until a non-negative value is obtained (see Kelton et al. 1998). Note, however, that this procedure can change the underlying distribution to a conditional one (in this case the conditioning is on non-negative values).

4.2.1 Generation of Uniform Variates

Suppose we wish to generate a realization x from the uniform distribution Unif (2, 10), for a realization $u = 0.65$ of the underlying RNG. Recalling the uniform cdf, given by Eq. (3.51), we write it as

$$u = \frac{x - a}{b - a},$$

where u is given and x is unknown. Solving the above for x to obtain the inverse cdf readily yields the formula

$$x = F_X^{-1}(u) = (b - a)u + a, \tag{4.4}$$

and substituting $a = 2,\ b = 10$, and $u = 0.65$ into the above results in the requisite value

$$x = (10 - 2)\,0.65 + 2 = 7.2.$$

4.2.2 Generation of Exponential Variates

Suppose we wish to generate a realization x from the exponential distribution Expo (0.5) with rate $\lambda = 0.5$ (mean 2), for a realization $u = 0.45$ of the underlying RNG. Recalling the cdf of the exponential distribution, given by Eq. 3.62, we write it as

$$u = 1 - e^{-\lambda x}$$

where u is given and x is unknown. Solving the above for x readily yields the formula

$$x = F_X^{-1}(u) = -\frac{1}{\lambda}\ln(1 - u), \tag{4.5}$$

and substituting $\lambda = 0.5$ and $u = 0.45$ into the above results in the requisite value

$$x = -2\ln(1 - 0.45) = 1.1957.$$

We mention that for practical simulation, Eq. 4.5 may be simplified into the equivalent formula

$$x = F_X^{-1}(u) = -\frac{1}{\lambda}\ln(u), \tag{4.6}$$

because if $U \sim \text{Unif}(0, 1)$, then also $1 - U = W \sim \text{Unif}(0, 1)$.

4.2.3 Generation of Discrete Variates

Suppose we wish to generate a realization x from a discrete distribution, whose pmf, $p_X(x)$, is specified in Table 4.2, for a realization $u = 0.45$ of the underlying RNG.

Suppose we code the state space by integers, say, $S = \{1, 2, \ldots\}$ as in our example. Recalling the discrete cdf, given by Eq. 3.36, we rewrite it as

$$u = \begin{cases} 0, & \text{if } x < 1 \\ \sum_{i=1}^{k} p_i, & \text{if } k \le x < k+1 \end{cases}$$

where k is the unique integer satisfying the equation above. Next, utilizing the general formula for an inverse cdf from Eq. 3.12, we deduce that the inverse cdf can be written as

$$x = F_X^{-1}(u) = \sum_{k \in S} k 1_{\Phi_k}(u), \text{ where } \Phi_k = \left[\sum_{n=1}^{k-1} p_n, \sum_{n=1}^{k} p_n \right],$$

or equivalently,

$$x = F_X^{-1}(u), \text{ for } \sum_{n=1}^{k-1} p_n \le u < \sum_{n=1}^{k} p_n. \tag{4.7}$$

In our case, (4.7) becomes

$$x = F_X^{-1}(u) = \begin{cases} 2, & \text{for } 0 \le u < 0.3 \\ 4, & \text{for } 0.3 \le u < 0.8 \\ 6, & \text{for } 0.8 \le u < 1 \end{cases}$$

and the corresponding graph is shown in Figure 4.2.

Table 4.2

Specification for a pmf for a discrete variate

x	$p_x(x)$
2	0.3
4	0.5
6	0.2

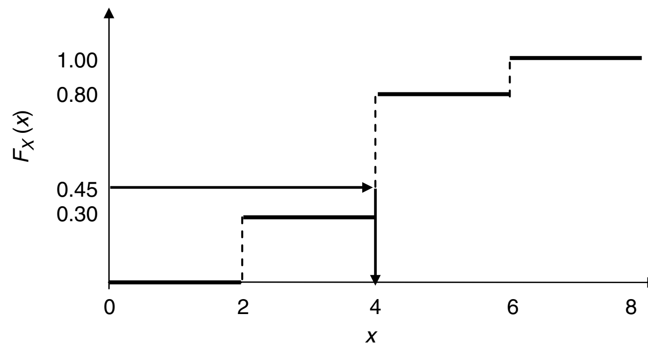


Figure 4.2 The Inverse Transform method for generating a discrete variate.

It follows that the requisite variate realization corresponding to $u = 0.45$ is $x = F_X^{-1}(0.45) = 4$, since $k = 2$ for this value of u.

4.2.4 Generation of Step Variates from Histograms

Suppose we wish to generate a realization x from a step distribution with $J = 4$ steps, whose pdf is specified in Table 4.3, for a realization $u = 0.5$ of the underlying RNG.

Recalling the step cdf, given by Eq. 3.55, we write it as

$$u = \begin{cases} 0, & \text{if } x < l_1 \\ \sum_{j=1}^{J} 1_{(l_j, r_j)}(x) \left[\sum_{i=1}^{j-1} p_i + (x - l_j) \frac{p_j}{r_j - l_j} \right], & \text{if } l_j \le x < r_j \\ 1, & \text{if } x \ge r_J. \end{cases} \tag{4.8}$$

Solving (4.8) for x yields the inverse step cdf,

$$x = F_X^{-1}(u) = \sum_{j=1}^{J} 1_{\Phi_j}(u) \left[l_j + \left(u - \sum_{i=1}^{j-1} p_i \right) \frac{r_j - l_j}{p_j} \right], \tag{4.9}$$

where

$$\Phi_j = \left[\sum_{n=1}^{j-1} p_n, \sum_{n=1}^{j} p_n \right].$$

The graph corresponding to Eq. 4.9 for this example is shown in Figure 4.3.

Table 4.3
Specification for a step pdf for a continuous variate

i	l_i	r_i	p_i
1	0	2	0.1
2	2	4	0.3
3	4	6	0.4
4	6	8	0.2

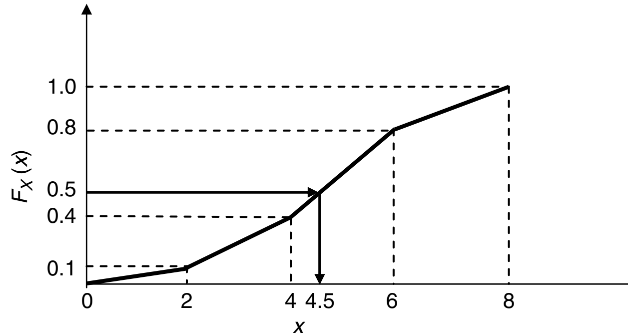


Figure 4.3 The Inverse Transform method for generating a variate with a step pdf.

It follows that the requisite variate realization corresponding to $u = 0.5$ is

$$x = F_X^{-1}(0.5) = l_3 + \left(u - \sum_{i=1}^{2} p_i\right) \frac{r_3 - l_3}{p_3} =$$
$$4 + (0.5 - [0.1 + 0.3]) \frac{6 - 4}{0.4} = 4.5.$$

4.3 PROCESS GENERATION

Generation of stochastic processes in simulation amounts to generating sequences of variates, subject to a particular probability law (see Section 3.9). Common cases are discussed in the following sections.

4.3.1 Iid Process Generation

Generation of iid processes is very common in simulation. Its implementation is rather straightforward: Simply generate repeatedly variates from a common distribution from successive RNG seeds, using the methods and formulas as illustrated above.

4.3.2 Non-Iid Process Generation

Generating non-iid processes that follow a prescribed probability law is the most difficult. The difficulty lies in reproducing key aspects of temporal dependence of the requisite non-iid stochastic process. More specifically, the problem is twofold.

To specify a probability law for a process, one must specify the temporal dependence of the process in the form of joint distributions of any dimension. Except for special cases (when dependence does *not* extend arbitrarily far into the past), this approach can be impractical.

When temporal dependence cannot be estimated in terms of joint distributions, one may elect to use a *statistical proxy* for temporal dependence, often in the form of autocorrelations. Even when the process to be modeled is stationary, the problem of devising a model that fits the observed (empirical) autocorrelations and marginal distribution, simultaneously, is nontrivial.

To illustrate the complexity inherent in the first problem, we shall outline the algorithm for generating a non-iid stochastic sequence, with a real valued state space, S.

Algorithm 4.1 Generation of a Non-Iid Stochastic Sequence

Input: a probability law L for a general random sequence $\{X_n\}_{n=0}^{\infty}$, specified by the probabilities

$$F_{X_0}(x_0) = \Pr\{X_0 \leq x_0\}, \quad x_0 \in S,$$
$$F_{X_1|X_0}(x_1|x_0) = \Pr\{X_1 = x_1|X_0 \leq x_0\}, \quad x_0, x_1 \in S,$$
$$F_{X_{n+1}|X_0,\ldots X_n}(x_{n+1}|x_0, \ldots, x_n) =$$
$$\Pr\{X_{n+1} \leq x_{n+1}|X_0 = x_0, \ldots, X_n = x_n\}, \quad x_0, \ldots, x_{n+1} \in S.$$

Output: A sample path (realization) $x_0, x_1, \ldots, x_n, \ldots$ from the probability law L.

1. Use the Inverse Transform method of Section 4.2 to generate $x_0 = F_{X_0}^{-1}(u_0)$ from the initial RNG seed, u_0.
2. Suppose that $x_0, \ldots, x_n$ have already been generated, using the RNG seeds $u_0, \ldots, u_n$. Use the Inverse Transform method of Section 4.2 to generate

$$x_{n+1} = F_{X_{n+1}|X_0,\ldots,X_n}^{-1}(u_{n+1}|x_0, \ldots, x_n),$$

   where u_{n+1} is the next RNG seed.
3. Continue analogously as necessary.

Clearly, Algorithm 4.1 requires an increasingly complex specification in view of the expanding conditioning on the n previous outcomes; this conditioning is suggestively called "memory," since to generate the next step, we need to "remember" the outcomes in all previous steps. Such extensive memory information is awkward, and in any event only rarely available.

We next illustrate two simple special cases of Algorithm 4.1: generation of discrete-state Markov chains in discrete time and in continuous time. Recall from Eq. 3.95 that Markov processes require limited "memory" of only one previous variate.

Algorithm 4.2 Generation of Discrete-Time, Discrete-State Markov Chains

Input: a probability law L for a discrete-time, discrete-state Markov chain $\{X_n\}_{n=0}^{\infty}$, specified by the probabilities

$$F_{X_0}(x_0) = \Pr\{X_0 \le x_0\}, \quad x_0 \in S,$$
$$F_{X_{n+1}|X_n}(x_{n+1}|x_n) = \Pr\{X_{n+1} \le x_{n+1}|X_n = x_n\}, \quad x_n,\ x_{n+1} \in S.$$

Output: A sample path (realization) $x_0, \ldots, x_n, \ldots$ from the Markov probability law L.

1. Use the Inverse Transform method of Section 4.2 to generate $x_0 = F_{X_0}^{-1}(u_0)$ from the initial RNG seed, u_0.
2. Suppose that $x_0, \ldots, x_n$ have already been generated, using the RNG seeds $u_0, \ldots, u_n$. Next, use the Inverse Transform method to generate

$$x_{n+1} = F_{X_{n+1}|X_n}^{-1}(u_{n+1}|x_n)$$

   where u_{n+1} is the next RNG seed.
3. Continue analogously as necessary.

The corresponding algorithm for a continuous-time, discrete-state Markov chain generates the jump chain exactly as in Algorithm 4.2. In addition, the jump times are also generated.

Algorithm 4.3 Generation of Continuous-Time, Discrete-State Markov Chains

Input: a probability law L for a continuous-time, discrete-state Markov chain $\{Y_t\}_{t=0}^{\infty}$, with a jump chain $\{X_n\}_{n=0}^{\infty}$ and jump times $\{T_n\}_{n=0}^{\infty}$, specified by the probabilities

$$F_{X_0}(x_0) = \Pr\{X_0 \le x_0\}, \quad x_0 \in S,$$
$$F_{X_{n+1}|X_n}(x_{n+1}|x_n) = \Pr\{X_{n+1} \le x_{n+1}|X_n = x_n\}, \quad x_n, x_{n+1} \in S,$$

and

$$F_{T_{n+1}-T_n|X_n}(\tau|x_n) = \Pr\{T_{n+1} - T_n \le \tau|X_n = x_n\} = 1 - \lambda_{x_n} e^{-\lambda_{x_n}\tau},$$
$$\tau \ge 0, \quad x_n, x_{n+1} \in S.$$

Output: A sample path (realization) of the jump chain, $y_0, y_1, \ldots, y_n, \ldots$, and jump times, $t_0, t_1, \ldots, t_n, \ldots$, from the Markov probability law L.

1. Set $t_0 = 0$, and use the Inverse Transform method to generate $x_0 = F_{X_0}^{-1}(u_0)$ from the initial RNG seed, u_0.
2. Suppose that $t_0, \ldots, t_n$ and $x_0, \ldots, x_n$ have already been generated, using the RNG seeds $u_0, \ldots, u_{2n+1}$. Then the current simulation clock is $T_n = t_n$, and the current jump chain state is $X_n = x_n$. The algorithm consumes two seeds per process jump as follows. First, use the Inverse Transform method of Section 4.2 to generate the next jump time realization,

$$t_{n+1} = t_n + F^{-1}_{T_{n+1}-T_n|X_n}(u_{2(n+1)}|x_n),$$

   where the interjump time is distributed exponentially according to Expo(λ_{x_n}) and $u_{2(n+1)}$ is the next RNG seed. Second, use the Inverse Transform method of Section 4.2 to generate the next state realization of the jump chain,

$$x_{n+1} = F^{-1}_{X_{n+1}|X_n}(u_{2(n+1)+1}|x_n),$$

   where $u_{2(n+1)+1}$ is the next RNG seed.
3. Continue analogously as necessary.

Finally, we mention here that methods exist for modeling temporal dependence using the autocorrelation function as a statistical proxy. This advanced topic is deferred, however, until Chapter 10.

EXERCISES

1. Construct the formula for a linear congruential generator with $m = 10{,}000$ and values of your choice for a and c. Set the last four digits of your Social Security Number as the initial seed, x_0 (if the leading digit of x_0 is 0, then change it to 9). Use this RNG to generate one complete period of random numbers. If the period turns out to be less than 1000, experiment with other values of a and c until you obtain an RNG with a period of at least 1000. Compute the mean, variance, and squared coefficient of variation of this sequence, as well as its histogram (use 10 cells).
2. Derive a formula for generating variates from the geometric distribution of mean 4, using the Inverse Transform method, and then apply it to obtain two samples: x_1 using $u_1 = 0.2$, and x_2 using $u = 0.7$. (*Hint*: Consult Section 3.7.4.)
3. Consider a random variable X with pdf

$$f_X(x) = \begin{cases} x/2, & 0 \le x < 1 \\ (-0.5x + 2)/3, & 1 \le x < 4 \\ 0, & x \ge 4. \end{cases}$$

a. Show that this is a legitimate pdf.
b. Derive a formula for generating variates from $f_X(x)$, using the Inverse Transform method, and then apply it to obtain two samples: x_1 using $u_1 = 0.4$, and x_2 using $u_2 = 0.9$.

4. Derive a formula for generating variates from the Weib(2, 0.5) distribution, and then apply it to obtain two samples: x_1 using $u_1 = 0.5$, and x_2 using $u_2 = 0.75$. (*Hint*: Consult Section 3.8.11.)
5. Inquiries arrive at a reception desk and are directed by the receptionist to one of three departments (Billing, Support, and Marketing). Measurements show that 30% of the calls are directed to Billing, 50% to Support, and the rest to Marketing. Furthermore, the sequence of departments to which inquiries are directed is iid.
   a. Devise an algorithm to direct inquires to the correct department, and code it in the language of your choice.
   b. Run the algorithm for 10,000 inquiries and write down the total calls per department and their relative frequencies.

Chapter 5

Arena Basics

The working simulation tool for the models in this book is *Arena*. Arena is a simulation environment consisting of module templates, built around *SIMAN* language constructs and other facilities, and augmented by a visual front end. This chapter provides an overview of Arena basics at an introductory level. For more detail, refer to Kelton et al. (2004). Because it may be hard to distinguish between Arena terms and generic terms, we shall always italicize any technical Arena terms throughout the book. Furthermore, we will adopt the notational convention that all characters in SIMAN constructs are always uppercase, while only the first character in Arena constructs is capitalized, and both are italicized.

SIMAN consists of two classes of objects: *blocks* and *elements*. More specifically, blocks are basic logic constructs that represent operations; for example, a *SEIZE* block models the seizing of a service facility by a transaction (referred to in Arena as "entity"), while a *RELEASE* block releases the facility for use by other transactions. *Elements* are objects that represent facilities, such as *RESOURCES* and *QUEUES*, or other components, such as *DSTATS* and *TALLIES*, used for statistics collection.

Arena's fundamental modeling components, called *modules*, are selected from template panels, such as *Basic Process*, *Advanced Process*, and *Advanced Transfer*,[1] and placed on a canvas in the course of model construction. A module is a high-level construct, composed of SIMAN blocks and/or elements. For example, a *Process* module models the processing of an entity, and internally consists of such blocks as *ASSIGN*, *QUEUE*, *SEIZE*, *DELAY*, and *RELEASE*. Arena also supports other modules, such as *Statistic*, *Variable*, and *Output* among many others. Frequently used Arena constructs (built-in variables and modules) are succinctly described in Appendix A.

Arena implements a programming paradigm that combines visual and textual programming. A typical Arena session involves the following activities:

1. Selecting module/block icons from a template panel, and placing them on a graphical model canvas (by drag and drop).

[1] Earlier versions of Arena have other panels, such as *Blocks*, *Elements*, *Common*, and *Support*. Later versions of Arena preserve these legacy panels in a template panel directory called *OldArena Templates*, and allow users to use them as necessary.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

2. Connecting modules graphically to indicate physical flow paths of transactions and/or logical flow paths of control.
3. Parameterization of modules or elements using a text editor.
4. Writing code fragments in modules using a text editor. Arena code is case-insensitive; that is, upper and lower case letters are interchangeable.

Arena has a *graphical user interface* (GUI) built around the SIMAN language. In fact, simulation models can be built using SIMAN constructs from the *Blocks* and *Elements* template panels alone, since Arena modules are just subprograms written in SIMAN. Still, Arena is far more convenient than SIMAN, because it provides many handy features, such as high-level modules for model building, statistics definition and collection, animation of simulation runs (histories), and output report generation. Model building tends to be particularly intuitive, since many modules represent actual sub-systems in the conceptual model or the real-life system under study. Complex models usually require both Arena modules and SIMAN blocks.

All in all, Arena provides a module-oriented simulation environment to model practically any scenario involving the flow of transactions through a set of processes. Furthermore, while the modeler constructs a model interactively in both graphical and textual modes, Arena is busy in the background transcribing the whole model into SIMAN. Since Arena generates correct SIMAN code and checks the model for syntactic errors (graphical and textual), a large amount of initial debugging takes place automatically.

5.1 ARENA HOME SCREEN

An Arena home screen is shown in Figure 5.1.

The reader is encouraged to browse the Arena home screen, and examine the objects to be mentioned in the sequel.

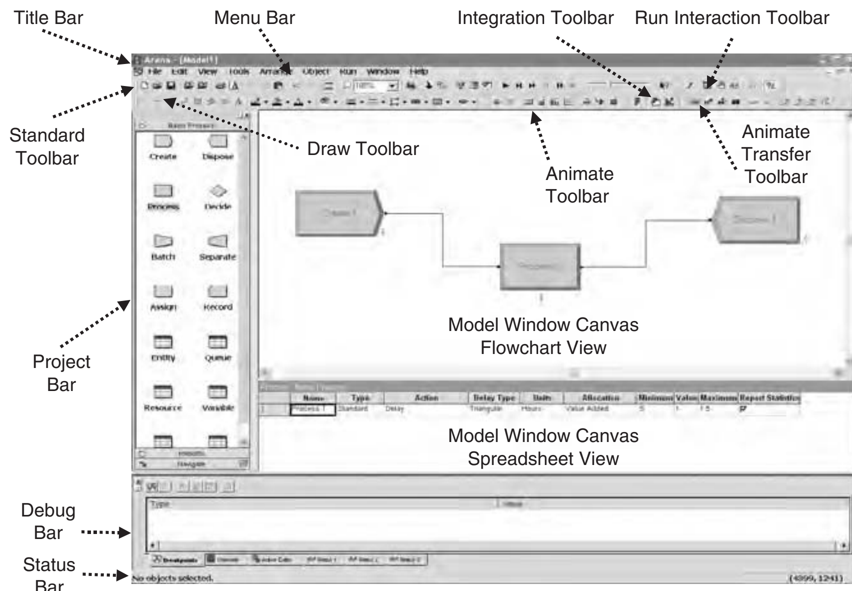


Figure 5.1 The Arena home screen.

The Arena home screen has a *Title* bar with the model name at its top. Below the *Title* bar is the Arena *Menu* bar, which consists of a set of general menus and Arena-specific menus. Below the *Menu* bar is a set of Arena toolbars that can be displayed and hidden by clicking the right mouse button on the background area of a specific toolbar. These toolbars consist of buttons that support model building and running; note that some toolbar buttons conveniently duplicate the functionality of certain menu options in the *Menu* bar.

The bulk of the home screen is allocated to a model window canvas consisting of a *flowchart* view and *spreadsheet v*iew. The user spawns modules and other objects, drags them into the flowchart view canvas, and progressively builds a model from component modules selected from the *Project* bar (on the left) with the aid of support functions. Selecting a module in the flowchart view pops up a spreadsheet view (below the flowchart view) that summarizes all module information and permits editing of the associated data.

In addition to the *Project* bar, the user can also pop up (or hide) two additional bars below the model window canvas. The *Debug* bar permits the user to track the evolution of simulation runs, while the *Status* bar (below it) displays instructions and feedback messages concerning user actions.

The *Menu* bar and the set of key Arena toolbars are described next. Consult the online help for complete descriptions of all Arena bars (menu bars, toolbars, and so on).

5.1.1 *MENU* BAR

The Arena *Menu* bar consists of a number of general menus—*File*, *Edit*, *View*, *Window*, and *Help*—which support quite generic functionality. It also has the following set of Arena-specific menus:

- The *Tools* menu provides access to simulation related tools and Arena parameters.
- The *Arrange* menu supports flowcharting and drawing operations.
- The *Object* menu supports module connections and submodel creation.
- The *Run* menu provides simulation run control. Its *Setup*. . . option opens a form that permits the user to enter information, such as project parameters (name, analyst, date, etc.), as well as replication parameters (length, warm-up period, etc.). It also has options that provide VCR-type functionality to run simulation replications and run control options to monitor entity motion, variable assignments, and so on.

5.1.2 *PROJECT* BAR

The *Project* bar lets the user access Arena template panels, where Arena modules, SIMAN blocks, and various other objects cohabit. Template panels can be attached to the *Project* bar by clicking the *Attach* button on the *Standard* toolbar. More specifically, when the *Attach* button is clicked, a dialog box pops up on the screen and shows the so-called *.tpo* files corresponding to each template panel. Choosing a *.tpo* file will attach its template panel to the *Project* bar. The Arena template panels available to users are as follows:

- The *Basic Process* template panel consists of a set of basic modules, such as *Create*, *Dispose*, *Process*, *Decide*, *Batch*, *Separate*, *Assign*, and *Record*.
- The *Advanced Process* template panel provides additional basic modules as well as more advanced ones, such as *Pickup*, *Dropoff*, and *Match*.
- The *Advanced Transfer* template panel consists of modules that support entity transfers in the model. These may be ordinary transfers or transfers using material-handling equipment.
- The *Reports* template panel supports report generation related to various components in a model, such as entities, resources, queues, and so on.
- The *Blocks* template panel contains the entire set of SIMAN blocks.
- The *Elements* template panel contains elements needed to declare model resources, queues, variables, attributes, and some statistics collection.

In addition to the Arena template panels above, the following Arena template panels from earlier versions are also supported:

- The *Common* template panel contains Arena modules such as *Arrive*, *Server*, *Depart*, *Inspect*, and so on, as well as element modules such as *Stats*, *Variables*, *Expressions*, and *Simulate*.
- The *Support* template panel contains a subset of frequently used SIMAN blocks.

The models we present in this book will mostly utilize modules from template panels of Arena 10.0 and later versions.

5.1.3 *STANDARD* TOOLBAR

The Arena *Standard* toolbar contains buttons that support model building. An important button in this bar is the *Connect* button, which supports visual programming. This button is used to connect Arena modules as well as SIMAN blocks, and the resulting diagram describes the flow of logical control. The *Time Patterns Editor* feature consists of three buttons that allow the modeler to schedule the availability of resources and their service rate. The *Standard* toolbar also provides VCR-style buttons to run an Arena model in interrupt mode to trace its evolution. More details on this VCR-like functionality are discussed in Section 6.3.1.

5.1.4 *DRAW* AND *VIEW* BARS

The *Draw* toolbar supports static drawing and coloring of Arena models. In a similar vein, the *View* toolbar (not shown in Figure 5.1) assists the user in viewing a model. Its buttons include *Zoom In*, *Zoom Out*, *View All*, and *View Previous*. These functions make it convenient to view large models at various levels of detail.

5.1.5 *ANIMATE* AND *ANIMATE TRANSFER* BARS

The *Animate* toolbar is used for animation (dynamic visualization) of Arena model objects during simulation runs. Animated objects include the simulation clock, queues, monitoring windows for variables, dynamic plots, and histogram functions. The

Animate Transfer toolbar is used to animate entity transfer activities, including materials handling (see Section 13.2 for more details).

5.1.6 *Run Interaction* Bar

The *Run Interaction* toolbar supports run control functions to monitor simulation runs, such as access to SIMAN code and model debugging facilities. It also supports model visualization, such as the *Animate Connectors* button that switches on and off entity traffic animation over module connections. Because of this toolbar's fundamental role in model testing, it will be revisited in Chapter 6.

5.1.7 *Integration* Bar

The *Integration* toolbar supports data transfer (import and export) to other applications. It also permits Visual Basic programming and design. A primer on Visual Basic for Arena may be found in Appendix B.

5.1.8 *Debug* Bar

The *Debug* toolbar supports debugging of Arena models by monitoring and controlling the execution of a simulation run. It consists of two subwindows. The left subwindow can be used in command mode to set breakpoints, assign variable values, observe watched variables, and trace entity flows among modules. The right subwindow has tabs for viewing future SIMAN events in the model, displaying attributes of the current active entity, and watching user-defined Arena expressions. For more information, see Chapter 6.

5.2 EXAMPLE: A SIMPLE WORKSTATION

We now proceed to illustrate the basic features of Arena through a simple example. Consider a single workstation consisting of a machine with an infinite buffer in front of it. Jobs arrive randomly and wait in the buffer while the machine is busy. Eventually they are processed by the machine and leave the system. Job interarrival times are exponentially distributed with a mean of 30 minutes, while job processing times are exponentially distributed with a mean of 24 minutes. This system is known in queueing theory as the *M/M/1* queue (Kleinrock 1975).

The steady-state behavior of this system has been well studied, and analytical formulas have been derived for its main performance measures, such as distribution of the number of jobs in the system and average waiting time in the buffer (see *ibid*.) This example will compare the simulation statistics to their theoretical counterparts to gauge the accuracy of simulation results. Specifically, we shall estimate by an Arena simulation the average job delay in the buffer, the average number of jobs in the buffer, and machine utilization.

Simulating the above workstation calls for the following actions:

1. Jobs are created, one at a time, according to the prescribed interarrival distribution. Arriving jobs are dispatched to the workstation.

2. If the machine is busy processing another job, then the arriving job is queued in the buffer.
3. When a job advances to the head of the buffer, it seizes the machine for processing once it becomes available, and holds it for a time period sampled from the prescribed processing-time distribution.
4. On process completion, the job departs the machine and is removed from the system (but not before its contribution to the statistics of the requisite performance measures are computed).

A simple approach to modeling the workstation under study is depicted in Figure 5.2.

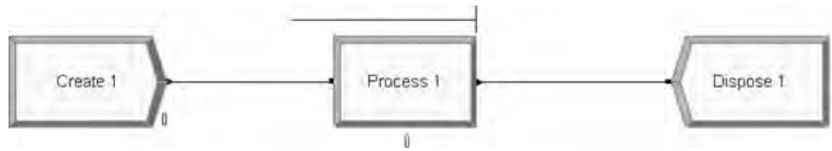


Figure 5.2 A simple Arena model of an *M/M/1* queue.

In this model, jobs (Arena entities) are created by the *Create* module *Create 1*. Jobs then proceed to be processed in the *Process* module *Process 1*, after which they enter the *Dispose* module, *Dispose 1*, for removal from the model. The graphic shaped like an elongated T (called a *T-bar*) above the module *Process 1* represents space for waiting jobs (here, the workstation's buffer). The interarrival specification of the *Create* module is shown in the dialog box of Figure 5.3.

Create
Name: Create 1
Entity Type: Entity 1
Time Between Arrivals
Type: Expression
Expression: EXPO(30)
Units: Minutes
Entities per Arrival: 1
Max Arrivals: Infinite
First Creation: 0.0
OK Cancel Help

Figure 5.3 Dialog box for a *Create* module.

The dialog box contains information on job interarrival time *(Time Between Arrivals* section), batch size (*Entities per Arrival* field), maximal number of job arrivals (*Max Arrivals* field), time of first job creation (*First Creation* field), and so on. The *Type* pull-down menu in the *Time Between Arrivals* section offers the following options:

- *Random* (exponential interarrival times with mean given in the *Value* field)
- *Schedule* (allows the user to create arrival schedules using the *Schedule* module from the *Basic Process* template panel)

- *Constant* (specifies fixed interarrival times)
- *Expression* (any type of interarrival time pattern specified by an Arena expression, including Arena distributions)

The job processing mechanism (including priorities) is specified in the *Process* module, whose dialog box is shown in Figure 5.4.

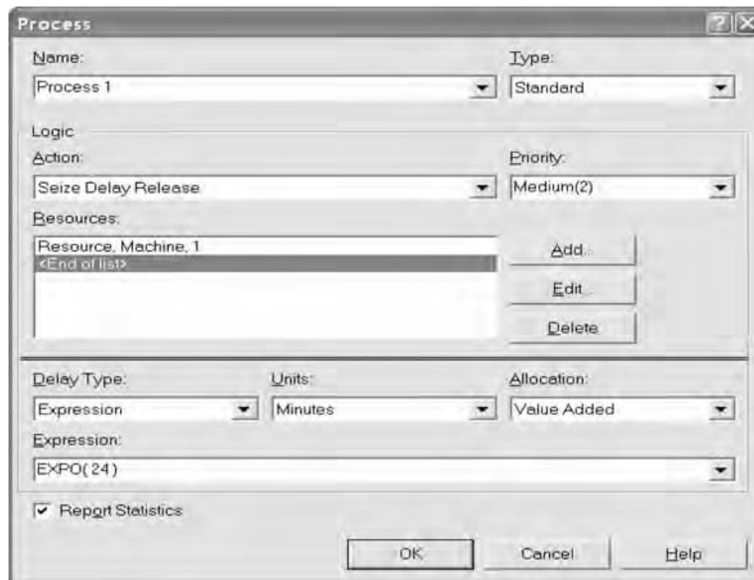


Figure 5.4 Dialog box for a *Process* module.

The *Action* field option, selected from the pull-down menu, is *Seize Delay Release*, which stands for a sequence of *SEIZE*, *DELAY*, and *RELEASE* SIMAN blocks. *SEIZE* and *RELEASE* blocks are used to model contention for a resource possessing a capacity (e.g., machines). When resource capacity is exhausted, the entities contending for the resource must wait until the resource is released. Thus the *SEIZE* block operates like a gate between entities and a resource. When the requisite quantity of resource becomes available, the gate opens and lets an entity seize the resource; otherwise, the gate bars the entity from seizing the resource until the requisite quantity becomes available. Note that the resource quantity seized should be an integer; otherwise Arena truncates it.

The processing (holding) time of a resource (*Machine* in our case) by an entity is specified via the *DELAY* block within the *Process* module. For instance, the dialog box of Figure 5.4 specifies that one unit of resource *Machine* be seized and held for an exponentially distributed time with a mean of 24 minutes. If multiple resources need to be seized simultaneously, all requisite resources are listed, and the holding time clock is started only after *all* requisite resources listed become available. When at least one of the listed resources is not available, arriving entities wait in a FIFO (First In First Out) queue, ordered further by priorities (specified in the *Priority* field of the dialog box). This discipline is called *FIFO within priority classes*, because higher-priority entities precede lower-priority ones, but arrivals within a given priority class are ordered FIFO.

The *Add* button in a *Process* module (see Figure 5.4) is used to specify resources and the resource quantities to be seized by an incoming entity. To this end, the *Add* button pops up a *Resources* dialog box as shown in Figure 5.5.

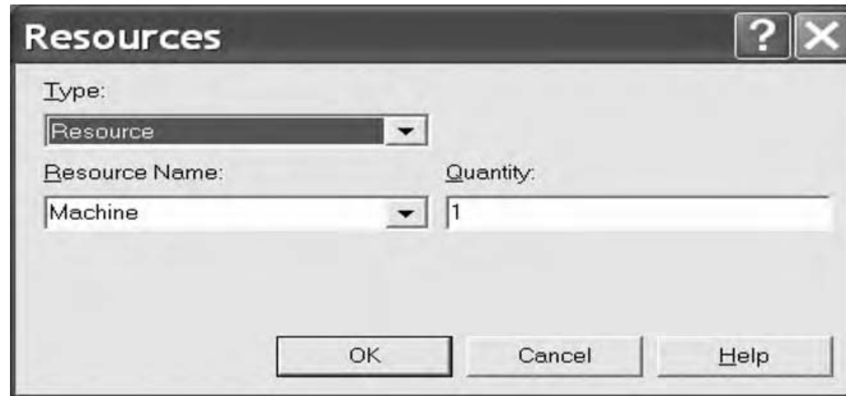


Figure 5.5 *Resources* dialog box for specifying a resource.

Note, however, that the capacity of resources introduced in a *Resources* dialog box is specified elsewhere, namely, in the *Resource* module spreadsheet (Figure 5.6).

Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Failures	Report Statistics
1	Machine	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑

Double-click here to add a new row.

Figure 5.6 *Resource* module spreadsheet for specifying resource capacities.

This spreadsheet is accessible in the *Basic Process* template of the *Project* bar, and is used to specify all resource capacities of an Arena model. The default capacity of resources is 1. More details on the *Resource* module spreadsheet are deferred to examples in the sequel.

The *Dispose* module implements an entity "sunset" mechanism, by simply discarding entities that enter it. As such, the *Dispose* module serves as a system "sink," thereby counteracting the *Create* module, which serves as a system "source." Note carefully that in the absence of properly placed *Dispose* modules, the number of entities created in the course of a simulation run can grow without bound, eventually exhausting available computer memory (or a prescribed limit) and terminating the run (very likely crashing the simulation program). A *Dispose* module is not necessary, however, when a fixed or bounded number of entities are created and circulate in the system indefinitely.

Whenever an Arena model is saved, the model is placed in a file with a *.doe* extension (e.g., *mymodel.doe*). Furthermore, whenever a model, such as *mymodel.doe*, is checked using the *Check Model* option in the *Run* menu or any run option in it, Arena automatically creates a number of files for internal use, including *mymodel.p* (program file), *mymodel.mdb* (Access database file), *mymodel.err* (errors file), *mymodel.opw* (model components file), and *mymodel.out* (SIMAN output report file). As these are internal Arena files, the modeler should not attempt to modify them.

Run control functionality is provided by the *Run* pull-down menu (see Figure 5.1). Selecting the *Setup...* option opens the *Run Setup* dialog box, which consists of multiple tabs, each with its own dialog box. In particular, the *Replication Parameters* tab permits the specification of the number of replications, replication length, and the warm-up period. (A warm-up period is a simulation of an initial interval, designed to "warm up" the system to a more representative state; this will be discussed in greater detail in Chapter 9 when we discuss output analysis.) In this example, the model will be simulated for 10,000 hours, with all other fields in the *Run Setup* dialog box retaining their default values.

The end result of a simulation run is a set of requisite statistics, such as mean waiting times, buffer size probabilities, and so on. These will be referred to as *run results*. Arena provides a considerable number of default statistics in a report, automatically generated at the end of a simulation run. Additional statistics can be obtained by adding statistics-collection modules to the model, such as *Record* (*Basic Process* template panel) and *Statistic* (*Advanced Process* template panel). However, when using SIMAN (the *Support* template panel in the *Old Arena Templates* folder), the user has to include additional blocks and elements in the model in order to affect statistics collection. The simpler statistics-collection facilities in Arena are one of the advantages of Arena over SIMAN. Subsequent chapters will provide additional information on statistics collection.

The run results of a single replication of the workstation model from Figure 5.2 are displayed in Figures 5.7 and 5.8.

3:33:44PM **Resources** September 1, 2005

A Simple Workstation Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 600,000.00 Time Units: Minutes

Machine

Usage	Value			
Total Number Seized	20,005.00			
Scheduled Utilization	0.8097			
Number Scheduled	1.0000	(Insufficient)	1.0000	1.0000
Number Busy	0.8097	0.014618405	0	1.0000
Instantaneous Utilization	0.8097	0.014618405	0	1.0000

Figure 5.7 Resource statistics from a single replication of the simple workstation model.

Figure 5.7 displays the *Resources* section, which includes resource utilization. The last three columns correspond to the half-width of the confidence interval (see Section 3.10), minimum observation, and maximum observation, respectively, of the corresponding statistics. The *(insufficient)* notation indicates that the number of observations is insufficient for adequate statistical confidence. Observe that the *Number Busy* item refers to the number of busy units of a resource, while the *Number Scheduled* item refers to resource capacity. The *Instantaneous Utilization* item pertains to utilization per

resource unit, namely, *Number Busy* divided by the *Number Scheduled*. We point out that in the long run, individual resource utilizations approach this number.

3:33:07PM **Queues** September 1, 2005

A Simple Workstation Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 600,000.00 Time Units: Minutes

Process 1.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	107.09	16.42523	0	807.06

Other	Average	Half Width	Minimum	Maximum
Number Waiting	3.5721	0.538999733	0	35.0000

Figure 5.8 Queue statistics from a single replication of the simple workstation model.

Figure 5.8 displays the *Queues* section, which consists of customer-oriented statistics (customer averages), such as mean waiting times in queues, as well as queue-oriented statistics, such as time averages of queue sizes (occupancies). Additional sections in an output report include *Frequencies* (probability estimates) and *User Specified* (any customized statistics). For more details on Arena statistics collection and output reporting, see Sections 5.4 and 5.5.

An examination of the run results (replication statistics) displayed in Figures 5.7 and 5.8 reveals that the machine utilization is about 81%. The average waiting time in the machine buffer is about 107 minutes, with a 95% confidence interval of 107.09 ± 16.42523 minutes, and a maximum of 807.06 minutes. The average number of jobs in the buffer is 3.57 jobs, with the maximal buffer length observed being 35 jobs.

Figure 5.7 can be used for preliminary verification of the workstation model. For example, we can use the classical formula $\rho = \lambda/\mu$ (Kleinrock 1975) of machine utilization to verify that the observed server utilization, ρ, is indeed the ratio of the job arrival rate ($\lambda = 1/30$) and the machine-processing rate ($\mu = 1/24$). Indeed, this ratio is 0.8, which is in close agreement with the run result, 0.8097. Note that this relation holds only approximately, due to the inherent variation in sampling simulation statistics. Other verification methods will be covered in some detail in Chapter 8.

5.3 ARENA DATA STORAGE OBJECTS

An important part of the model building process is assignment and storage of data supplied by the user (input parameters) or generated by the model during a simulation run (output observations). To this end, Arena provides three types of data storage objects: *variables*, *expressions*, and *attributes*. Variables and expressions can be introduced and initialized via the *Variable* and *Expression* spreadsheet modules, accessible from the *Basic Process* template and *Advanced Process* template, respectively, in the *Project* bar.

5.3.1 Variables

Variables are user-defined *global* data storage objects used to store and modify state information at run initialization or in the course of a run. Such (global) variables are visible everywhere in the model; namely, they can be accessed, examined, and modified from every component of the model. In an Arena program, variables are typically examined in *Decide* modules and modified in *Assign* modules. Unlike user-defined variables, certain predefined Arena (system) variables are read-only (i.e., they may only be examined to decide on a course of action or to collect statistics), and cannot be assigned a new value by the user; the system is solely responsible for changing these values. For instance, the variable *NQ*(*Machine_Q*) stores the current value of the number of entities in the queue called *Machine_Q*. Similarly, the variable *NR*(*Machine*) stores the number of busy units of the resource called *Machine*. Other important Arena variables are *TNOW*, which stores the simulation run's current time (*simulation clock*), and *TFIN*, which stores the simulation completion time. A list of all Arena variables may be found in the *Arena Variables Guide* or in the Arena *Help* facility (see Section 6.6). Recall that a selected partial list appears in Appendix A.

5.3.2 Expressions

Expressions can be viewed as specialized *variables* that store the value of an associated formula (expression). They are used as convenient shorthand to compute mathematical expressions that may recur in multiple parts of the model. Whenever an expression name is encountered in the model, it is promptly evaluated at that point in simulation time, and the computed value is substituted for the expression name. Variables of any kind (user defined or system defined) as well as attributes may be used in expressions.

5.3.3 Attributes

Attributes are data storage objects associated with entities. Unlike variables, which are global, attributes are local to entities in the sense that each instance of an entity has its own copy of attributes. For example, a customer's arrival time can be stored in a customer attribute to allow the computation of individual waiting times. When arrivals consist of multiple types of customer, the type of an arrival can also be stored in a customer entity's attribute to allow separate statistics collection for each customer type.

5.4 ARENA OUTPUT STATISTICS COLLECTION

As mentioned before, the end product of a simulation is a set of statistics that estimate performance measures of the system under study. Recall that such statistics can be classified into the two standard categories of time averages and customer averages (see, e.g., Section 2.3). More specifically, *time averages* are obtained by dividing the area under the performance function (e.g., number in the system, periods of busy and idle states, etc.) by the elapsed simulation time. *Customer average* statistics are averages of customer-related performance values (e.g., customer waiting times in queues).

Arena provides two basic mechanisms for collecting simulation output statistics: one via the *Statistic* module, and the other via the *Record* module. Time average statistics are collected in Arena via the *Statistic* module, while customer-average statistics must be collected via a *Record* module, and (optionally) specified in the *Statistic* module. Arena statistics collection mechanisms are described next in some detail.

5.4.1 Statistics Collection via the *Statistic* Module

Detailed statistics collection in Arena is typically specified in the *Statistic* module located in the *Advanced Process* template panel. Selecting the *Statistic* module opens a dialog box. The modeler can then define statistics as rows of information in the spreadsheet view that lists all user-defined statistics. For each statistic, the modeler specifies a name in the *Name* column, and selects the type of statistic from a drop-down list in the *Type* column. The options are as follows:

Time-Persistent statistics are simply time average statistics in Arena terminology. Typical *Time-Persistent* statistics are average queue lengths, server utilization, and various probabilities. Any user-defined probability or time average of an expression can be estimated using this option.

Tally statistics are customer averages, and have to be specified in a *Record* module (see Section 5.4.2) in order to initiate statistics collection. However, it is advisable to include the definition in the *Statistic* module as well, so that the entire set of statistics can be viewed in the same spreadsheet for modeling convenience.

Counter statistics are used to keep track of counts, and like the *Tally* option, have to be specified in a *Record* module (see Section 5.4.2) in order to initiate statistics collection.

Output statistics are obtained by evaluating an expression at the end of a simulation run. Expressions may involve Arena variables such as *DAVG(S)* (time average of the *Time-Persistent* statistic *S*), *TAVG(S)* (the average of Tally statistic *S*), *TFIN* (simulation completion time), *NR()*, *NQ()*, or any variable from the *Arena Variables Guide*.

Frequency statistics are used to produce frequency distributions of (random) expressions, such as Arena variables or resource states. This mechanism allows users to estimate steady-state probabilities of events, such as queue occupancy or resource states.

Note that all statistics defined in the *Statistic* module are reported automatically in the *User Specified* section of the Arena output report (see Section 5.5). Furthermore, *Queue* and *Resource Time-Persistent* statistics will be automatically computed and need not be defined in the *Statistic* module.

5.4.2 Statistics Collection via the *Record* Module

As mentioned in Section 5.2, the *Record* module is used to collect various statistics. Any statistics related to customer averages or customer observations, such as *Tally* and *Counter*, have to be specified in a *Record* module. Figure 5.9 displays a dialog box for a *Record* module, listing all types of statistics as options in the *Type* field.

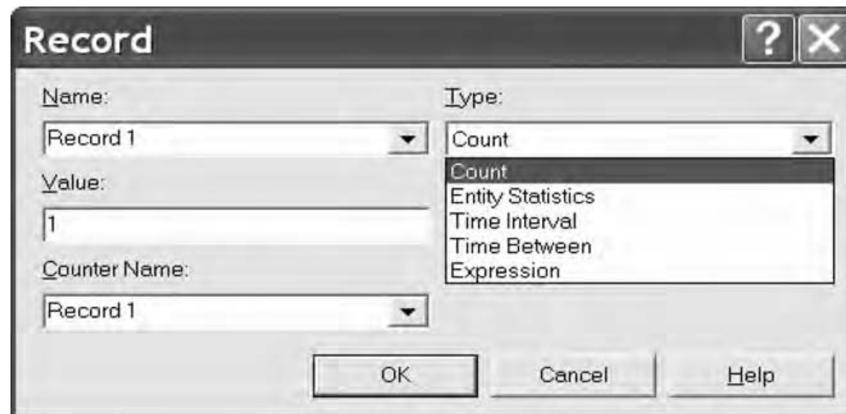


Figure 5.9 Types of statistics collected by the *Record* module.

These options are as follows:

1. The *Count* option maintains a count with a prescribed increment (positive or negative). The increment is quite general: it may be defined as any expression or function and may assume any real value. The corresponding counter is incremented whenever an entity enters the *Record* module.
2. The *Entity Statistics* option provides information on entities, such as time and costing/duration information.
3. The *Time Interval* option tallies the difference between the current time and the time stored in a prescribed attribute of the entering entity.
4. The *Time Between* option tallies the time interval between consecutive entries of entities in the *Record* module. These intervals correspond to interdeparture times from the module, and the reciprocal of the mean interdeparture times is the module's *throughput*.
5. Finally, the *Expression* option tallies an expression whose value is recomputed whenever an entity enters the *Record* module.

Note that except for the *Entity Statistics* option, all the options above are implemented in SIMAN via *COUNT* or *TALLY* blocks.

5.5 ARENA SIMULATION AND OUTPUT REPORTS

An Arena model run can be initiated and managed in two ways:

- Via the VCR-like buttons on the Arena *Standard* toolbar
- Via corresponding options in the *Run* pull-down menu bar

Refer to Section 6.3 for more details.

Standard Arena output reports provide summaries of simulation run statistics, as requested by the modeler implicitly or explicitly. Accordingly, Arena output reports fall into two categories:

Automatic reports. A number of Arena constructs, such as entities, queues, and resources, will automatically generate reports of summary statistics at the end of

a simulation run. Those statistics are implicitly specified by the modeler simply by dragging and dropping those modules into an Arena model, and no further action is required of the user.

User-specified reports. Reports on additional statistics can be obtained by explicitly specifying statistics collection via the *Statistic* module (*Advanced Process* template panel) and the *Record* module (*Basic Process* template panel). Recall that the *Statistic* module is specified in a spreadsheet view, while the *Record* module must be placed in the appropriate location in the model.

To request a formatted report, the user opens the *Reports* panel in the *Project* bar to display a list of report options that correspond to various types of statistics as follows:

- *Entities* reports automatically provide various entity counts.
- *Frequencies* reports provide time averages of expressions (including probabilities as a special case). The expression must be specified in a *Statistic* module with the *Frequency* option selected.
- *Processes* reports provide statistics associated with each *Process* module. These include incoming and outgoing entity counts, average service times, and average delays. Time-oriented statistics (e.g., delays) include the average, half-width of 95% confidence intervals, and minimal and maximal observed values. The half-width value is not displayed if the observations are correlated or if there are insufficient data (in which case Arena prints *(insufficient)* in the corresponding field).
- *Queues* reports provide statistics for each queue in the model, such as average queue delay and average queue size. Additional statistics include the half-width of the 95% confidence interval, and minimal and maximal observed values.
- *Resources* reports provide statistics for each resource in the model, such as utilization, average number of busy resource units, and number of times seized.
- *User Specified* reports are generated in response to explicit modeler requests for statistics collection in the *Statistic* module or *Record* modules (see Section 5.4). These include any *Time-Persistent*, *Tally*, *Count*, *Frequency*, and *Output* statistics.

Clicking on a report option in the *Reports* template panel of the *Project* bar prompts Arena to format the corresponding report and display it in a window. The top of that window contains VCR-like navigation buttons allowing the user to navigate report pages. Any number of options may be selected (sequentially), and the resultant report generated by Arena will consist of corresponding report sections. The *Toggle Group Tree* button at the top of the report window pops up a *Preview* panel to aid in navigating report sections.

5.6 EXAMPLE: TWO PROCESSES IN SERIES

This section presents a two-stage manufacturing model with two processes in series. Jobs arrive at an assembly workstation with exponentially distributed interarrival times of mean 5 hours. We assume that the assembly process has all the raw materials necessary to carry out the operation. The assembly time is uniformly distributed between 2 and 6 hours. After the process is completed, a quality control inspection is performed, and past data reveal that 15% of the jobs fail any given inspection and go back to the assembly operation for rework (jobs may end up going through multiple reworks until they pass inspection). Jobs that pass inspection go to the next stage, which

is a painting operation that takes 3 hours per job. We are interested in simulating the system for 100,000 hours to obtain process utilizations, average number of reworks per job, average job waiting times, and average job flow times (elapsed times experienced by job entities). Figure 5.10 depicts the corresponding Arena model and a snapshot of its state.

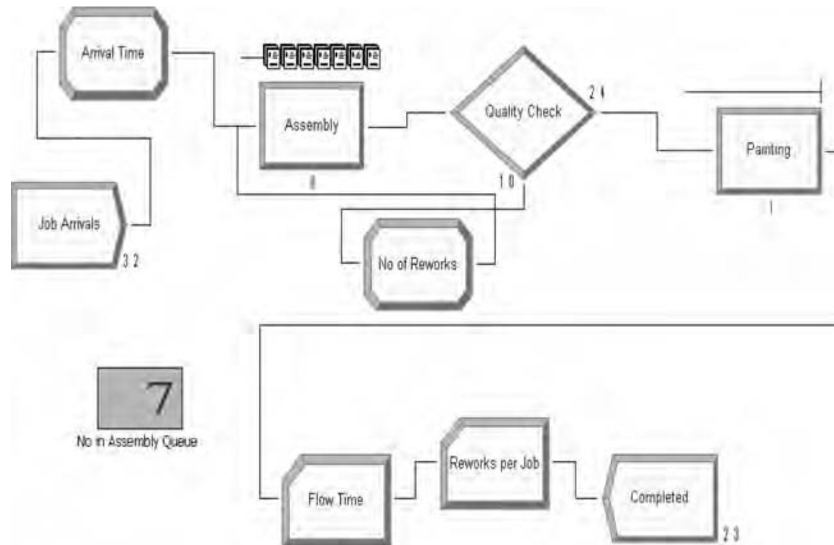


Figure 5.10 Arena model of two processes in series.

In this Arena model, entities represent jobs that are created by the *Create* module *Job Arrivals*, and enter the *Assign* module *Arrival Time*, where their arrival time is assigned to an attribute called *ArrTime*. Job entities then proceed to the *Process* module *Assembly*, where they undergo assembly. Next, assembled job entities go through inspection in the *Decide* module *Quality Check*, and those needing rework are routed to the *Assign* module *Number of Reworks* to record the number of reworks per job, and then back to module *Assembly.* Job entities that pass inspection proceed to be painted in the *Process* module *Painting*, which completes the manufacturing operation. Completed job entities go through *Record* modules *Flow Time* and *Reworks per Job* to collect statistics of interest. Finally, job entities enter the *Dispose* module *Completed*, where they are removed from the model.

The numbers at bottom right of the module icons display the number of entities that departed from the module. The icons above the module *Assembly* represent jobs waiting in the *Assembly* queue, called *Assembly.Queue*. The *Variable* window at bottom left of the figure displays the number of the aforementioned jobs. Job icons and data displayed on the module are *dynamic*, that is, they change as the simulation run unfolds in time. For example, this snapshot indicates that at the time it was taken, 28 jobs were created, 20 jobs were completed, 7 jobs were waiting to be assembled, and 1 job was being assembled.

We next proceed to explain the modules used in the model in more detail (note that the default module names have been replaced here by more expressive names relevant to the problem at hand).

The arrival time assignment is carried out in the *Assign* module *Arrival Time*, whose dialog boxes are shown in Figure 5.11.

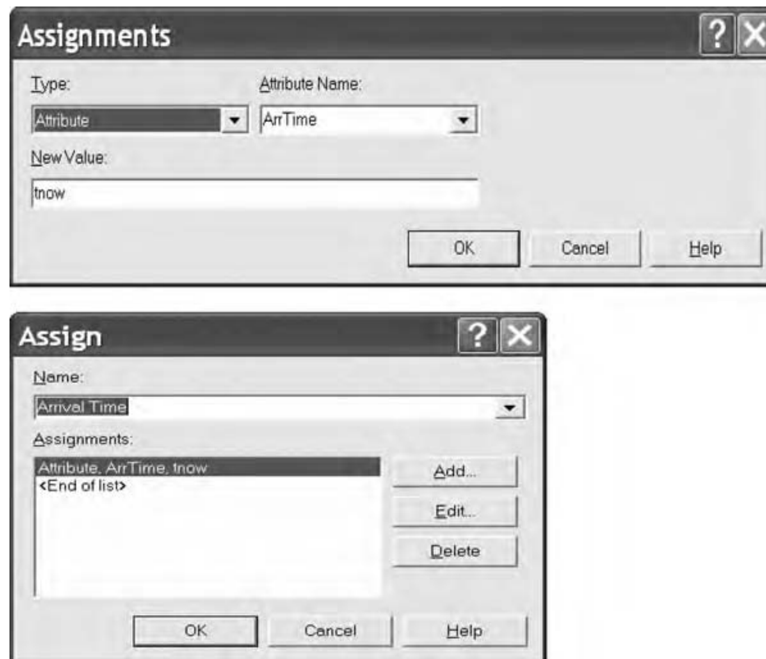


Figure 5.11 Dialog boxes for assigning an arrival time to a job entity's attribute.

Note that the bottom dialog box appears first on the screen, while clicking on an item in its *Assignments* field pops up the top dialog box for entering a value for the corresponding variable or attribute.

The module *Assembly* in Figure 5.10 is, in fact, a *Process* module that models the assembly operation with the appropriate job assembly time specification. Recall that the T-bar above the *Process* icon represents an animated *Queue* object for holding entities waiting for an opportune condition (e.g., to seize a resource). Note that the T-bar is displayed in a *Process* module only if the module contains a *Seize* option. The dialog box of the module *Assembly* in Figure 5.10 is displayed in Figure 5.12.

A *Decide* module, called *Quality Check*, follows the assembly operation, representing a quality control check. The *Decide* dialog box is shown in Figure 5.13. This module executes entity transfers, which may be probabilistic or based on the truth or falsity of some logical condition. Our example assumes that transfer times between processes are negligible, and therefore instantaneous. Here, we have a *two-way probabilistic branching* that splits the incoming stream of job entities into two outgoing streams:

With probability 0.85, an incoming job is deemed "good" (passed inspection), and is forwarded to the *Process* module, called *Painting*.

With probability 0.15 an incoming job is deemed "bad" (failed inspection), and is routed back to the *Process* module, called *Assembly*, for rework.

For convenience, the *Type* field provides separate options for simple two-way branching, which is the most common, as well as general *n*-way branching, which is more complex. In *n-way branching*, this module has multiple exits, exactly one

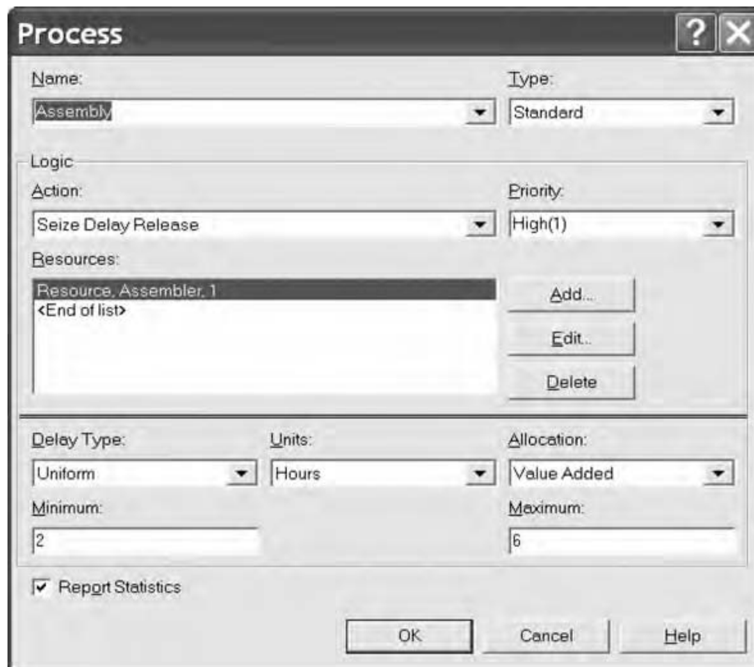


Figure 5.12 Dialog box of the *Process* module *Assembly.*

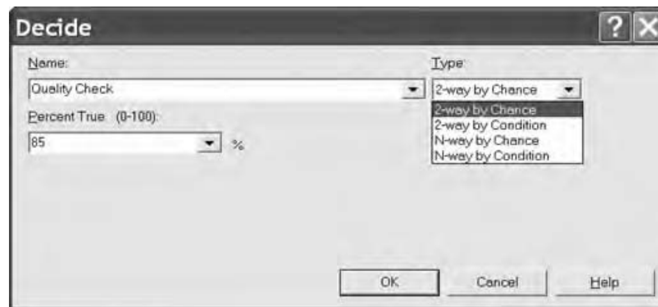


Figure 5.13 Dialog box of the *Decide* module *Quality Check* with a two-way probabilistic branching.

of which is designated the *else* branch and serves for default exits of entities from the module. More specifically, in *probabilistic branching*, a branch is taken with its associated probability, and the *else* branch automatically complements the exit probabilities, ensuring that they sum to 1. In *conditional branching*, the entity exits at the first branch whose condition evaluates to true, and if all conditions are false, the entity exits at the *else* branch.

Job entities that fail inspection enter the *Assign* module, called *No of Reworks*, where the job attribute, called *Total Reworks*, is incremented by the expression *Total Reworks + 1*. The dialog box for this module is shown in Figure 5.14.

Job entities departing from the *Painting* module enter two *Record* modules called *Flow Time* and *Reworks per Job*, to record their flow times and total number of reworks per job, respectively. In our example, the flow time is the difference between the job

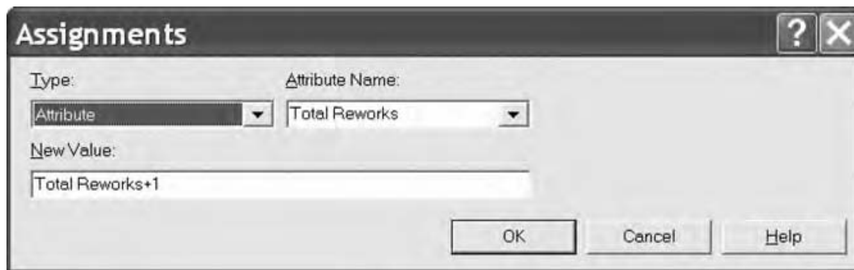


Figure 5.14 Dialog box of the *Assign* module *No of Reworks*.

departure time from the *Painting* module and the job's arrival time at module *Assembly*. Note that the job flow time includes any delays in queues as well as processing times. The average flow time is a customer average (*Time Interval* statistic), computed internally by a *TALLY* block, which can be verified by accessing the corresponding *.mod* file. The dialog box of the *Record* module is shown in Figure 5.15.

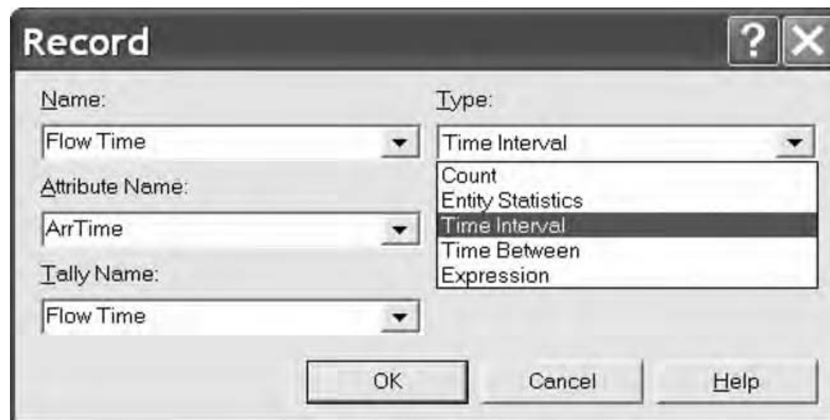


Figure 5.15 Dialog box of a *Record* module tallying job flow times.

In the next *Record* module, the expression consisting of the job entity's attribute *Total Reworks* is tallied in the *Record* module called *Reworks per Job*, as shown in the dialog box of Figure 5.16.

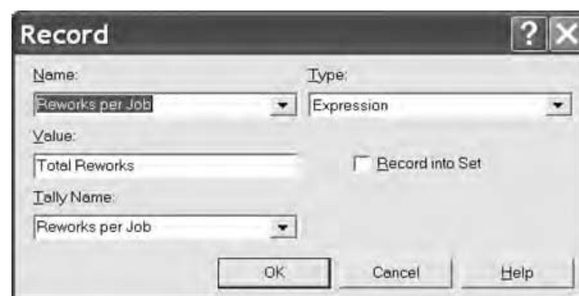
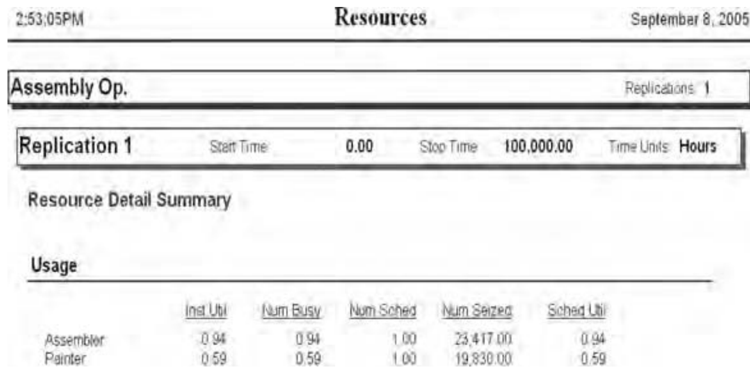


Figure 5.16 Dialog box of the *Record* module tallying the number of reworks per job.

Finally, job entities proceed to be disposed of in the *Dispose* module, called *Completed*. A single replication of the model was run for 100,000 hours, and the results are displayed in Figures 5.17, 5.18, and 5.19.



2:53:05PM **Resources** September 8, 2005

Assembly Op. Replications: 1

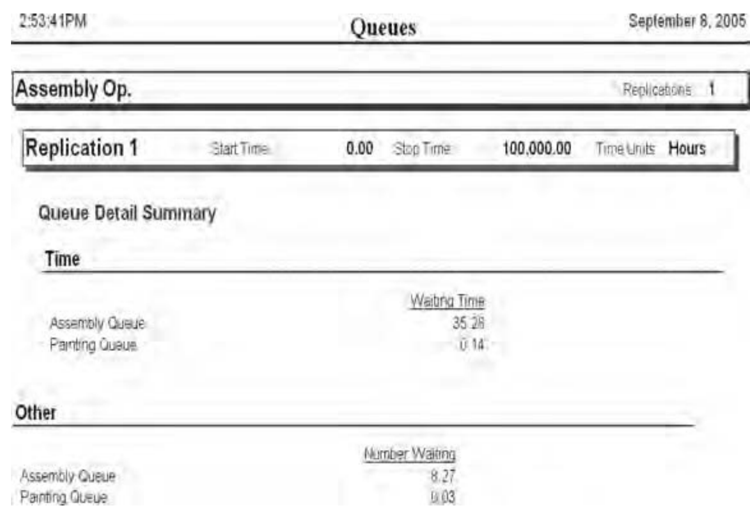
Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Resource Detail Summary

Usage

	Inst Util	Num Busy	Num Sched	Num Seized	Sched Util
Assembler	0.94	0.94	1.00	23,417.00	0.94
Painter	0.59	0.59	1.00	19,830.00	0.59

Figure 5.17 Resource statistics for the manufacturing model (two processes in series).



2:53:41PM **Queues** September 8, 2005

Assembly Op. Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Queue Detail Summary

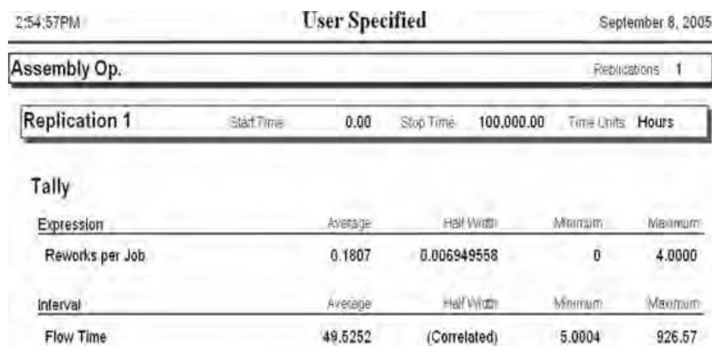
Time

	Waiting Time
Assembly Queue	35.28
Painting Queue	0.14

Other

	Number Waiting
Assembly Queue	8.27
Painting Queue	0.03

Figure 5.18 Queue statistics for the manufacturing model (two processes in series).



2:54:57PM **User Specified** September 8, 2005

Assembly Op. Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Tally

Expression	Average	Half Width	Minimum	Maximum
Reworks per Job	0.1807	0.006949558	0	4.0000

Interval	Average	Half Width	Minimum	Maximum
Flow Time	49.5252	(Correlated)	5.0004	926.57

Figure 5.19 Flow time statistics for the manufacturing model (two processes in series).

Figure 5.17 shows that the utilization estimates of the *Assembler* resource at the *Assembly* module and the *Painter* resource at the *Painting* module are 0.94 and 0.59, respectively. These estimates in Figure 5.17 correspond to a heavy traffic regime in the former and a medium traffic regime in the latter. These result in long average buffer delays and large average buffer occupancy in the assembly process, and in short average buffer delays and low average buffer occupancy in the painting process as shown in Figure 5.18. Finally, the *Tally* section of the *User Specified* output in Figure 5.19 displays not only averages, but also the corresponding 95% confidence interval half-widths, as well as the minimal and maximal observations for the number of reworks and flow time per job. Note that some jobs underwent as many as four reworks, while the average number of reworks is just 0.18, indicating a low level of reworks. The average flow time (49.5252 hours) is moderately longer than the sum of the average delays in the *Assembly* and *Painting* queues (35.42 hours) and the sum of average processing times there (7 hours), due to additional rework performed at module *Assembly*.

5.7 EXAMPLE: A HOSPITAL EMERGENCY ROOM

This section presents a more detailed model—in this case, of an emergency room in a small hospital—to further illustrate the power of simulation modeling.

5.7.1 Problem Statement

The emergency room of a small hospital operates around the clock. It is staffed by three receptionists at the reception office, and two doctors on the premises, assisted by two nurses. However, one additional doctor is on call at all times; this doctor is summoned when the patient workload up-crosses some threshold, and is dismissed when the number of patients to be examined goes down to zero, possibly to be summoned again later. Figure 5.20 depicts a diagram of patient sojourn in the emergency room system, from arrival to discharge.

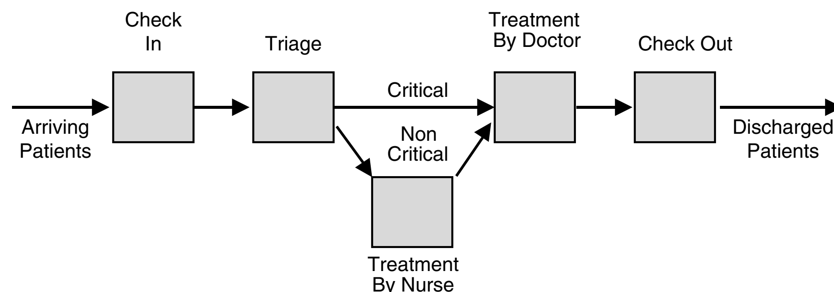


Figure 5.20 Patient sojourn in a hospital emergency room system.

Patients arrive at the emergency room according to a Poisson process with mean interarrival time of 10 minutes. An incoming patient is first checked into the emergency room by a receptionist at the reception office. Check-in time is uniform between 6 and

12 minutes. Since critically ill patients get treatment priority over noncritical ones, each patient first undergoes *triage* in the sense that a doctor determines the criticality level of the incoming patient in FIFO order. The triage time distribution is triangular with a minimum of 3 minutes, a maximum of 15 minutes, and a most likely value of 5 minutes. It has been observed that 40% of incoming patients arrive in critical condition, and such patients proceed directly to an adjacent treatment room, where they wait FIFO to be treated by a doctor. The treatment time of critical patients is uniform between 20 and 30 minutes. In contrast, patients deemed noncritical first wait to be called by a nurse who walks them to a treatment room some distance away. The time spent to reach the treatment room is uniform between 1 and 3 minutes and the treatment time by a nurse is uniform between 3 and 10 minutes. Once treated by a nurse, a noncritical patient waits FIFO for a doctor to approve the treatment, which takes a uniform time between 5 to 10 minutes. Recall that the queueing discipline of all patients awaiting doctor treatment is *FIFO within their priority classes*, that is, all patients wait FIFO for an available doctor, but critical patients are given priority over noncritical ones. Following treatment by a doctor, all patients are checked out FIFO at the reception office, which takes a uniform time between 10 and 20 minutes, following which the patients leave the emergency room.

The performance metrics of interest in this problem are as follows:

- Utilization of the emergency room staff by type (doctors, nurses, and receptionists)
- Distribution of the number of doctors present in the emergency room
- Average waiting time of incoming patients for triage
- Average patient sojourn time in the emergency room
- Average daily throughput (patients treated per day) of the emergency room

To estimate the requisite statistics, the hospital emergency room was simulated for a period of 1 year.

5.7.2 Arena Model

Having studied the problem statement, we now proceed to construct an Arena model of the system under study. Figure 5.21 depicts an Arena model of the emergency room system, where modules are labeled with their type to facilitate understanding.

The model is composed of two segments:

- *Emergency room segment*. This logic (top segment of Figure 5.21) keeps track of model entities (patients in our case), whose specification can be viewed and edited in the spreadsheet view of the *Entity* module from the *Basic Process* template panel. In this part of the model logic, a patient is generated and then moves through the emergency room processing, from admission to treatment to discharge.
- *On-call doctor segment*. This logic (bottom segment of Figure 5.21) controls the periodic summoning and dismissal of the extra doctor on call. This is achieved by a perpetually circulating single entity, called *administrator* in this model, which triggers the summoning and dismissal of the on-call doctor.

In addition, input and output data logic is interspersed in the two segments above. This logic consists of input/output modules (corresponding to variables, resources, statistics, etc.) that set input variables, compute statistics, and generate

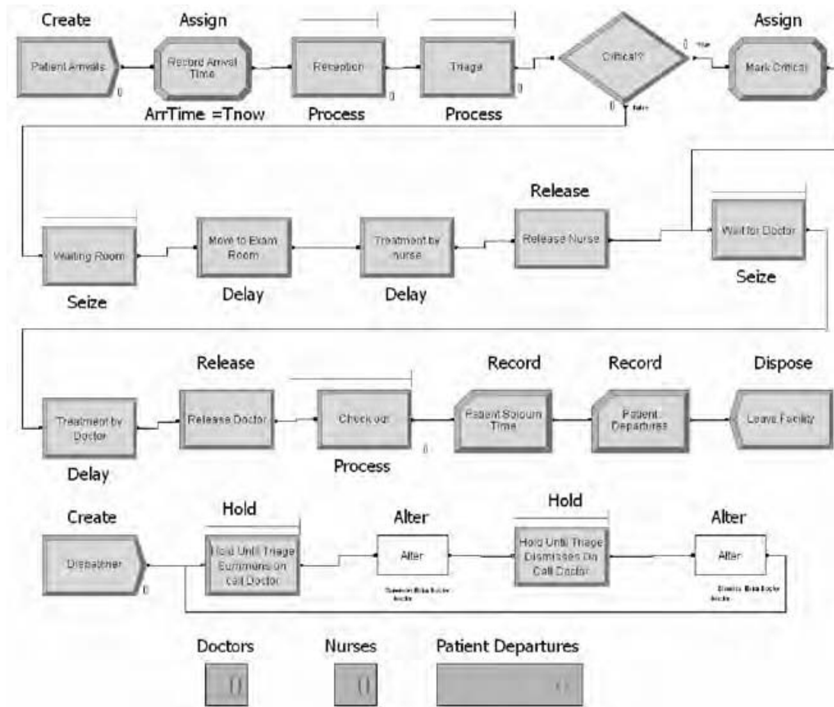


Figure 5.21 Arena model for the emergency room system.

summary reports. We assume that the emergency room is initially empty and the on-call doctor is not present.

We next proceed to examine the Arena model logic of Figure 5.21 in some detail, including the role of common modules from the *Basic Process* and *Advanced Process* template panels.

5.7.3 Emergency Room Segment

Starting at top left and moving to the right in the emergency room segment, the first module is the *Create* module, called *Patient Arrivals*, which generates incoming patient entities. Figure 5.22 displays the dialog box for this module, showing that patients arrive according to a Poisson process with exponential interarrival times of mean 10 minutes.

An incoming patient entity then enters the *Assign* module, called *Record Arrival Time*, where its arrival time is recorded in its *ArrTime* attribute. The value in this attribute will be carried by the patient entity throughout its sojourn in the emergency room system, and will be used later to compute its *sojourn time* (total time in the system).

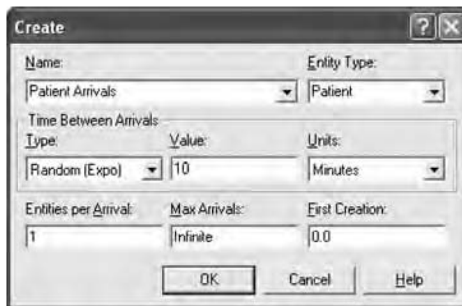


Figure 5.22 Dialog box of the *Create* module *Patient Arrivals*.

The patient entity then promptly attempts to check into the emergency room by entering the *Process* module, called *Reception*, whose dialog box is depicted in Figure 5.23.

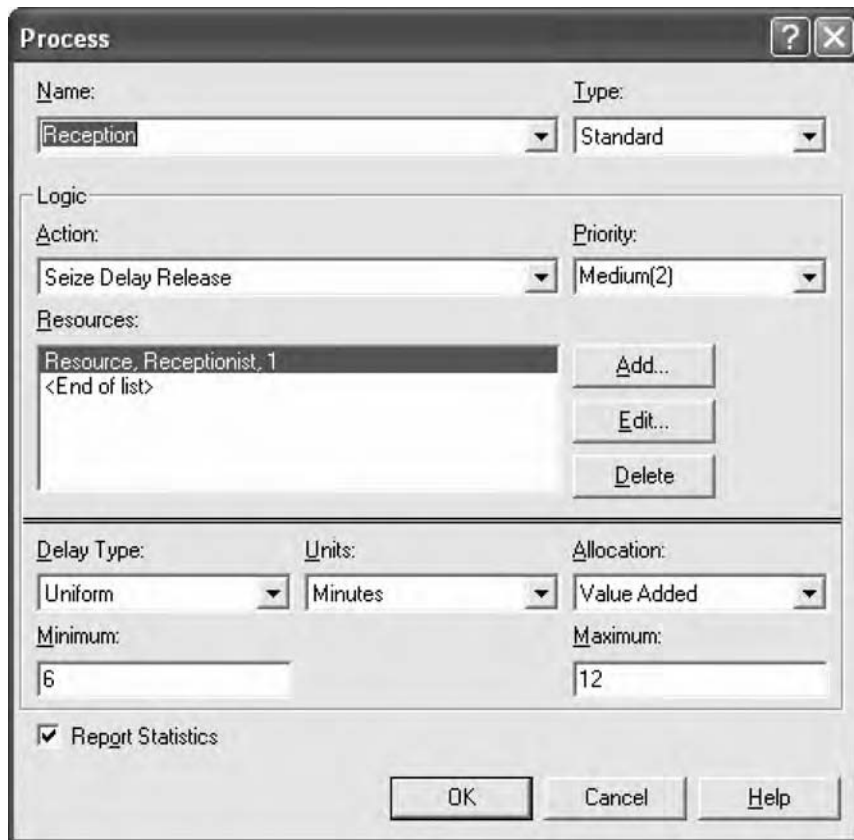


Figure 5.23 Dialog box of the *Process* module *Reception*.

At this juncture, the patient entity waits in line (if any) to seize a receptionist for a uniform check-in processing time between 6 and 12 minutes, after which the receptionist is released and becomes available to other patient entities. Note that the *Process* module uses the *Seize Delay Release* option in the *Action* field, since receptionists are modeled as

a resource, and the problem calls for computing expected waiting times and utilizations of the check-in operation. Once check-in is completed, the patient entity proceeds to the *Process* module, called *Triage*, to undergo a triage checkout by a doctor. Figure 5.24 shows that a patient entity waits for a doctor to become available, and then undergoes a random triage time, drawn from the triangular distribution between 3 and 15 minutes, with a most likely time of 5 minutes.

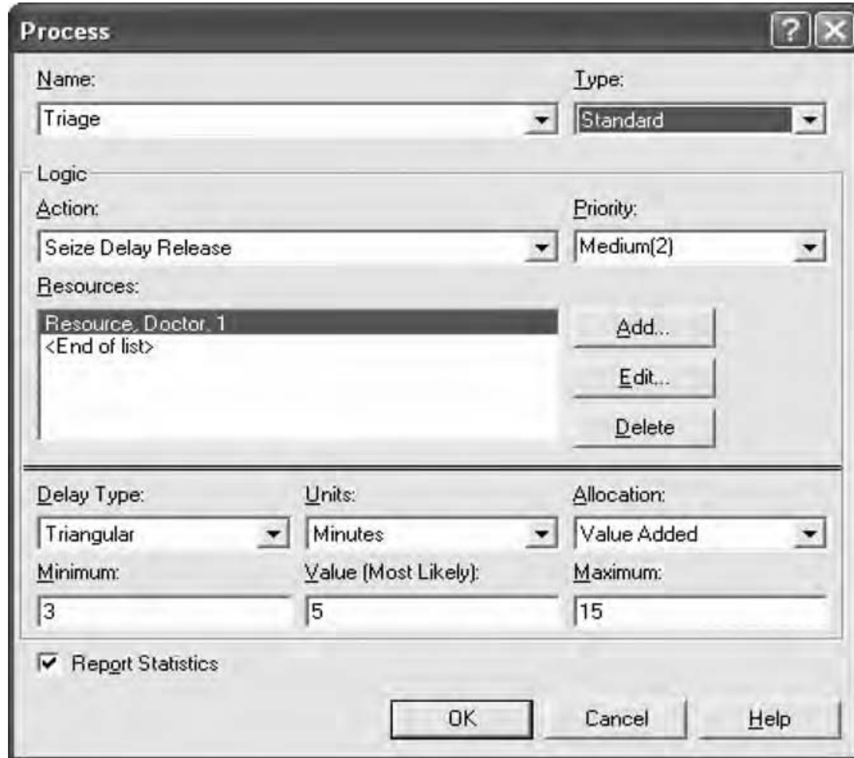


Figure 5.24 Dialog box of the *Process* module *Triage*.

After the triage delay is completed, the triage doctor is released and the patient entity proceeds to determine its level of criticality. To this end, it enters the *Decide* module, called *Critical?*, whose dialog box is depicted in Figure 5.25.

In Figure 5.25, the *2-Way by Chance* option specifies a two-way random branching based on the result of a random experiment, and the *Percent True* field indicates that 40% of the time the result is true (so 60% of the time the result is false). Accordingly, the corresponding branches emanating from the *Decide* module *Critical?* in Figure 5.21 are labeled *true* and *false*. Thus, a patient entity emerging from module *Critical?* takes either the *true* branch or the *false* branch as follows:

- The *true* branch indicates that the patient is entity deemed critical. Such a patient entity will proceed to be treated by a doctor. Recall that this applies to 40% of the patients.
- The *false* branch indicates that the patient entity is deemed noncritical. Such a patient entity will proceed to be treated by a nurse, and then will be inspected by a doctor, but at a lower priority than any critical patient entity.

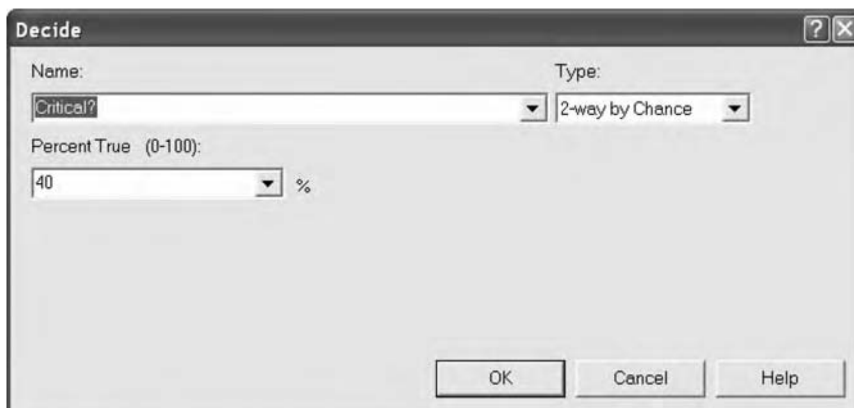


Figure 5.25 Dialog box of the *Decide* module *Critical?* for determining patient criticality.

The criticality level of patient entities is indicated in their *Criticality* attribute: a value of 1 codes for a critical patient, while a value of 0 codes for a noncritical patient. Accordingly, critical patient entities exiting module *Critical?* proceed to the *Assign* module, called *Mark Critical*, where their *Criticality* attribute is set to 1, as shown in the dialog box of Figure 5.26.

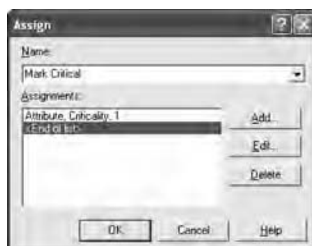


Figure 5.26 Dialog box of the *Assign* module *Mark Critical* for marking critical patients.

In contrast, noncritical patient entities are automatically marked as such, since the default value of the *Criticality* attribute is 0 (recall that this is the Arena convention for all attributes). Such patient entities exiting module *Critical?* proceed to the *Seize* module, called *Waiting Room*, whose dialog box is depicted in Figure 5.27.

In this module, noncritical patient entities wait FIFO in a queue, called *Waiting Room. Queue*, for a nurse until one becomes available. Once a nurse is seized, the patient entity passes through two *Delay* modules in succession: module *Move to Treatment Room* models the uniformly distributed time between 1 and 3 minutes that it takes the nurse to walk a (noncritical) patient to a treatment room, while module *Treatment by Nurse* models the uniformly distributed time between 3 and 10 minutes that it takes the nurse to treat a patient. Figure 5.28 depicts the dialog boxes of these two *Delay* modules.

Having completed its treatment, the noncritical patient entity releases the nurse by entering the *Release* module, called *Release Nurse*, whose dialog box is shown in Figure 5.29.

At this point the paths of critical and noncritical patient entities converge, and all patient entities, both critical and noncritical, attempt to enter the *Seize* module, called *Wait for Doctor*. Note that an individual *Seize* module has the same functionality as the *Seize*

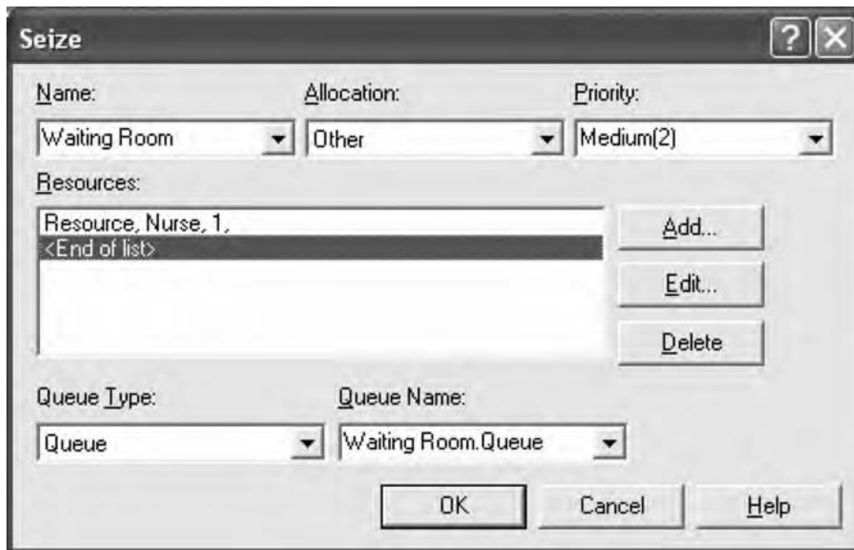


Figure 5.27 Dialog box of the *Seize* module *Waiting Room*.

Figure 5.28 Dialog boxes of the *Delay* modules *Move to Treatment Room* (left) and *Treatment by Nurse* (right).

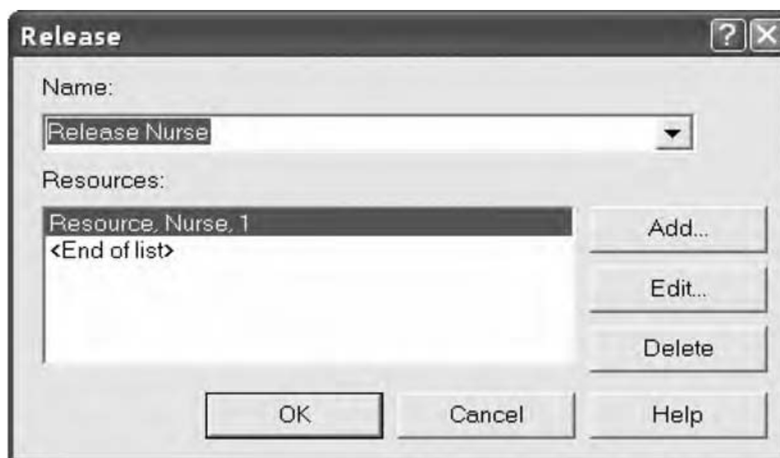


Figure 5.29 Dialog box of the *Release* module *Release Nurse*.

option in a *Process* module, but with the added flexibility that the modeler can insert extra logic between the *Seize* and *Delay* functionalities (this is impossible in a *Process* module). The dialog box of the *Wait for Doctor* module is shown in Figure 5.30.

Figure 5.30 Dialog box of the *Seize* module *Wait for Doctor*.

All patient entities wait in the queue, called *Wait for Doctor.Queue* to seize an available doctor, with critical patient entities receiving priority in treatment over noncritical ones. To this end, all patient entities queue up *FIFO within priority classes*, that is, all critical patient entities precede all noncritical ones, but each patient category is queued in the order of arrival. This is achieved by specifying the appropriate queueing discipline in the *Queue* module, whose spreadsheet view is shown in Figure 5.31.

Queue - Basic Process

	Name	Type	Attribute Name	Shared	Report Statistics
1	Check out.Queue	First In First Out	Attribute 1	☐	☑
2	Triage.Queue	First In First Out	Attribute 1	☐	☑
3	Reception.Queue	First In First Out	Attribute 1	☐	☑
4	Wait for Doctor.Queue	Highest Attribute Value	Criticality	☐	☑
5	Waiting Room.Queue	First In First Out	Attribute 1	☐	☑
6	Hold Until Triage Summons on call Doctor.Queue	First In First Out	Attribute 1	☐	☑
7	Hold Until Triage Dismisses On Call Doctor.Queue	First In First Out	Attribute 1	☐	☑
	Double-click here to add a new row.				

Figure 5.31 Spreadsheet view of the *Queue* module specifying queueing disciplines.

Each row in the spreadsheet specifies a queue in the Arena model, while columns *Type* and *Attribute Name* specify jointly the queueing discipline. Observe that all rows, except row 4, specify the ordinary FIFO discipline, while row 4 implicitly specifies the *FIFO within priority classes* discipline. More specifically, patient entities in *Wait for Doctor.Queue* queue up FIFO, but their queueing priority is determined by their *Criticality* attribute (the higher the value of *Criticality*, the higher the priority).

Once a doctor becomes available, the patient entity at the head of the line seizes that doctor and proceeds to the *Delay* module, called *Treatment by Doctor*, whose dialog box is depicted in Figure 5.32.

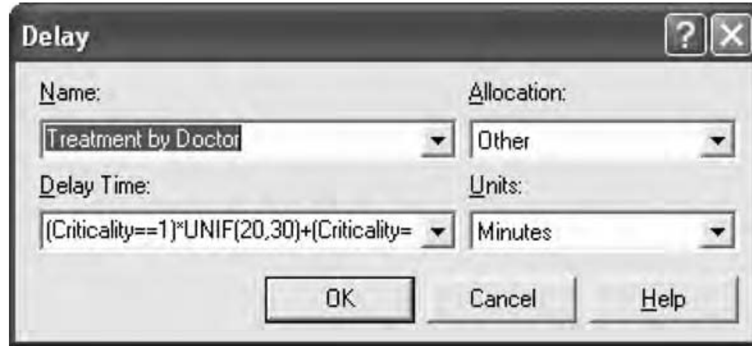


Figure 5.32 Dialog box of the *Delay* module *Treatment by Doctor*.

Recall that the treatment duration of a patient depends on its level of criticality, namely, on its *Criticality* attribute: for critical patients, the duration is uniform between 20 and 30 minutes, while for noncritical ones it is uniform between 5 and 10 minutes only. This dependence is captured in the *Delay Time* field of Figure 5.32 by the expression

$$(Criticality == 1)^{*}UNIF(20, 30) + (Criticality == 0)^{*}UNIF(5, 10).$$

Recall that *(Criticality==1)* and *(Criticality==0)* are logical expressions (predicates) that return 1 or 0 according to logical value of the expression in parentheses, which evaluates to true or false, respectively. Thus, for critical patients the expression above reduces to *UNIF(20,30)*, whereas for noncritical ones it reduces to *UNIF(5,10)*, which are the requisite distributions.

Following treatment by a doctor, all patient entities proceed to the *Release* module, called *Release Doctor*, where the doctor administering the treatment is released to other patients. Next, all patient entities are discharged from the emergency room. To this end, they enter the *Process* module, called *Check Out*, whose dialog box is shown in Figure 5.33. The checkout procedure requires a patient to seize a receptionist for a uniform time between 10 and 20 minutes, before releasing that receptionist.

Finally, patient entities enter two statistics-collecting *Record* modules, called *Patient Sojourn Time* and *Patient Departures*, respectively, whose dialog boxes are depicted in Figure 5.34.

The first *Record* module (left) tallies the total time that patient entities spend in the emergency room, from arrival to discharge (sojourn time)—a measure of patient satisfaction. This statistic is specified in the *Type* field by the *Time Interval* option, and is computed as *Tnow—ArrTime*, namely, the difference between the current time and the patient's arrival time as recorded in its *ArrTime* attribute. The second *Record* module (right) simply counts the number of patient entities discharged from the emergency room—a measure of emergency room productivity. This statistic is specified in the *Type* field by the *Count* option, and is computed by incrementing a counter variable, called *Patient Departures* (see the *Counter Name* field), whenever a patient entity enters this module. Note that the modeler can access the current value of any counter via the Arena variable *NC(counter_name)*.

Figure 5.33 Dialog box of the *Process* module *Check Out*.

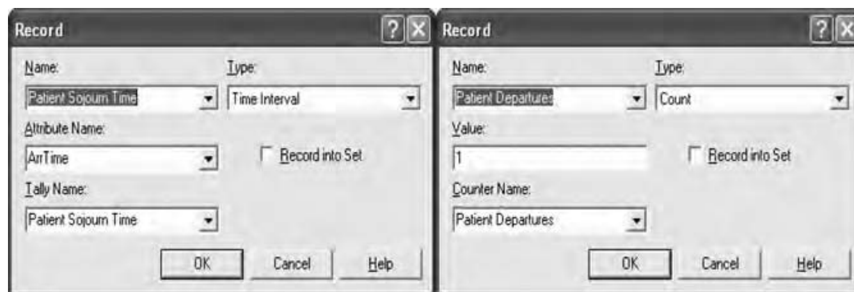


Figure 5.34 Dialog boxes of the *Record* modules *Patient Sojourn Time* (left) and *Patient Departures* (right).

At long last, patient entities enter the *Dispose* module, called *Leave Facility*, after which they are removed from the model.

5.7.4 On-Call Doctor Segment

The logic of this segment is controlled by a single circulating entity, dubbed *administrator*. The idea is to have the administrator modulate the number of doctors in the emergency room, depending on the number of patients in triage. Recall that emergency room doctors are a resource, so the administrator need only change this resource capacity as prevailing conditions change in the triage operation.

First, a single administrator entity is created at time 0, as shown in the dialog box of the *Create* module, called *Dispatcher*, in Figure 5.35.

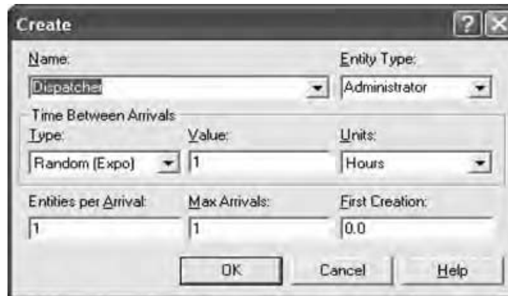


Figure 5.35 Dialog box of the *Create* module *Dispatcher*.

Note that this module simply creates precisely one entity of type *Administrator* at time 0. Consequently, the arrival stream specified by this module stops after the first entity is created, and therefore the interarrival time specification in the *Type* and *Value* fields is immaterial.

Since at this point the emergency room does not have the on-call doctor on duty, the administrator next watches for the condition that triggers summoning of the on-call doctor. To this end, the administrator enters the *Hold* module, called *Hold Until Triage Summons On-Call Doctor*, whose dialog box is shown in Figure 5.36.

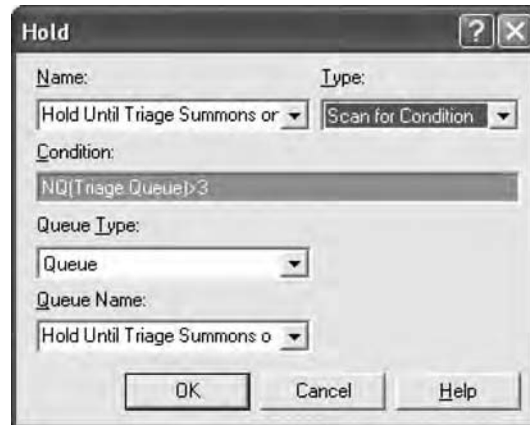


Figure 5.36 Dialog box of the *Hold* module *Hold Until Triage Summons On-Call Doctor*.

The administrator entity is held in this module (actually in a queue contained in this module as indicated by the *Queue Name* field) until the number of patients in triage exceeds three, signaling that the time has come to summon the on-call doctor. To watch for this condition, the *Scan for Condition* option is selected in the *Type* field, and the condition triggering the summoning of the on-call doctor is specified in the *Condition* field as

$$NQ\ (Triage.Queue) > 3,$$

where *NQ(Triage.Queue)* is the Arena variable that holds the current number of entities in the queue. As soon as this condition becomes true, the administrator entity proceeds

to perform the action of summoning the on-call doctor by entering the *Alter* module, labeled *Summon Extra Doctor*, whose dialog box is shown in Figure 5.37.

Figure 5.37 Dialog box of the *Alter* block *Summon Extra Doctor*.

Note that this module is called a *block* here, which is Arena's old term for *module*. To provide backward compatibility with older versions, Arena maintains a set of old blocks, which may be selected from the *Blocks* template panel, *Alter* included. The *Alter* block is used to alter model parameters. In our model, the administrator entity entering this block causes the *Doctor* resource pool to be incremented by 1, as evidenced by the expression in the *Resources* field above. This has the effect of increasing the available number of doctors in the emergency room by 1 (note that doctors do not have individual identities in our model).

The administrator entity next watches for the condition that triggers dismissal of the on-call doctor. To this end, it proceeds to the *Hold* module, called *Hold Until Triage Dismisses On-Call Doctor*, whose dialog box is depicted in Figure 5.38.

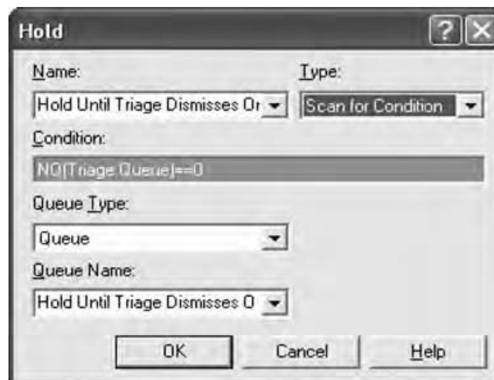


Figure 5.38 Dialog box of the *Hold* module *Hold Until Triage Dismisses On-Call Doctor*.

In a vein similar to the summoning action, the administrator entity is held in this module until the number of patient entities in triage drops to 0, signaling that the time has come to dismiss the on-call doctor, as evidenced by the *Condition* field expression.

$$NQ\ (Triage.Queue) == 0.$$

As soon as this condition becomes true, the administrator entity proceeds to perform the action of dismissing the on-call doctor by entering the *Alter* module, labeled *Dismiss Extra Doctor*, whose dialog box is shown in Figure 5.39.

Figure 5.39 Dialog box of the *Alter* block *Dismiss Extra Doctor*.

Here the effect of the administrator entity is to reduce the capacity of the *Doctor* resource by 1 (note that a resource capacity cannot be reduced to a negative value).

Finally, the administrator entity loops back to the *Hold* module, called *Hold Until Triage Summons On-Call Doctor*, to start the next cycle of summoning/dismissing the on-call doctor. Thereafter, the administrator entity will continue traversing this loop indefinitely throughout a simulation run.

5.7.5 Statistics Collection

In this model, statistics are collected using various methods as follows:

- Using *Record* modules
- Default collection of statistics by Arena
- Specifying statistics in the *Statistic* module

Statistics collection via *Record* modules was described in Section 5.4. Default collection of statistics is achieved in *Queue* and *Resource* spreadsheets by checking the last column (e.g., see Figure 5.31), and this is the default Arena setting. Finally, we illustrate statistics collection via the *Statistic* module, whose spreadsheet view is depicted in Figure 5.40.

Statistic - Advanced Process

	Name	Type	Expression	Report Label	Frequency Type	Expression	Output File	Categories
1	Distribution of Doctors	Frequency			Value	NR(Doctor)		4 rows
2	Daily Throughput	Output	NC(Patient Departures)/365	Daily Throughput				0 rows

Double-click here to add a new row.

Figure 5.40 Spreadsheet view of the *Statistic* module specifying frequency and output statistics.

This spreadsheet specifies collection of the distribution (*histogram*) of the number of doctors in the emergency room and the daily facility throughput.

5.7.6 Simulation Output

Figures 5.41 through 5.43 display reports of the results of a simulation run of length 525,600 minutes (1 year of emergency room operation).

Figure 5.41 displays statistics of patient sojourn time and patient flow through the emergency room. Here, the *Tally* section indicates that the tallied patient sojourn times, from patient arrival to patient discharge, last on average some 108 minutes, and the half-width of their 95% confidence interval is 2.3 minutes. However, the sojourn times have considerable variability as indicated by the minimal and maximal observed sojourn times. The *Counter* section records that the emergency room processed over 52,000 patients during its 1-year operation, while the *Output* section shows that the daily throughput was about 144 patients per day.

4:16:18PM **User Specified** September 28, 2005

Emergency Room Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 525,600.00 Time Units: Minutes

Tally

Interval	Average	Half Width	Minimum	Maximum
Patient Sojourn Time	108.07	2.31751	33.9229	510.04

Counter

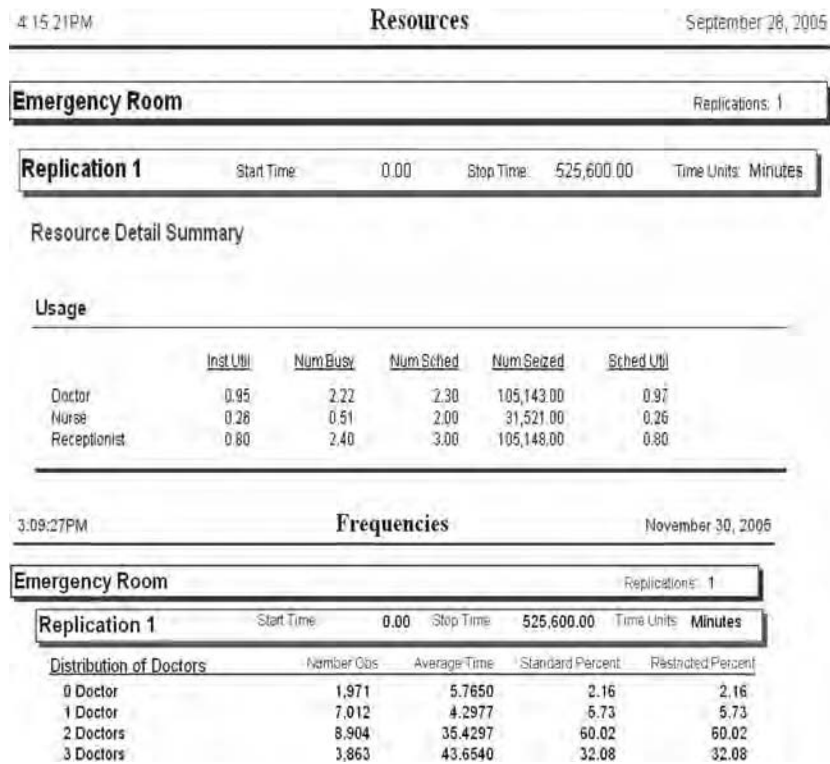
Count	Value
Patient Departures	52,563.00

Output

Output	Value
Daily Throughput	144.01

Figure 5.41 Statistics of patient sojourn time and patient flow in the emergency room model.

Figure 5.42 displays utilization statistics of human resources in the emergency room, which consist of doctors, nurses, and receptionists. Recall that the numbers of nurses and receptionists at the emergency room are fixed throughout the simulation horizon, whereas the number of doctors is variable due to the periodic summoning and dismissal of the doctor on call.



4:15:21PM **Resources** September 28, 2005

Emergency Room Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 525,600.00 Time Units: Minutes

Resource Detail Summary

Usage

	Inst Util	Num Busy	Num Sched	Num Seized	Sched Util
Doctor	0.95	2.22	2.30	105,143.00	0.97
Nurse	0.26	0.51	2.00	31,521.00	0.26
Receptionist	0.80	2.40	3.00	105,148.00	0.80

3:09:27PM **Frequencies** November 30, 2006

Emergency Room Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 525,600.00 Time Units: Minutes

Distribution of Doctors	Number Obs	Average Time	Standard Percent	Restricted Percent
0 Doctor	1,971	5.7650	2.16	2.16
1 Doctor	7,012	4.2977	5.73	5.73
2 Doctors	8,904	35.4297	60.02	60.02
3 Doctors	3,863	43.6540	32.08	32.08

Figure 5.42 Statistics of human resource utilization in the emergency room model.

The *Usage* section in the *Resources* segment displays utilization-related statistics of emergency room (human) resources, taking into account the fact that the number of available resources (in this case doctors) may vary. These utilization statistics are computed over the simulation horizon as follows:

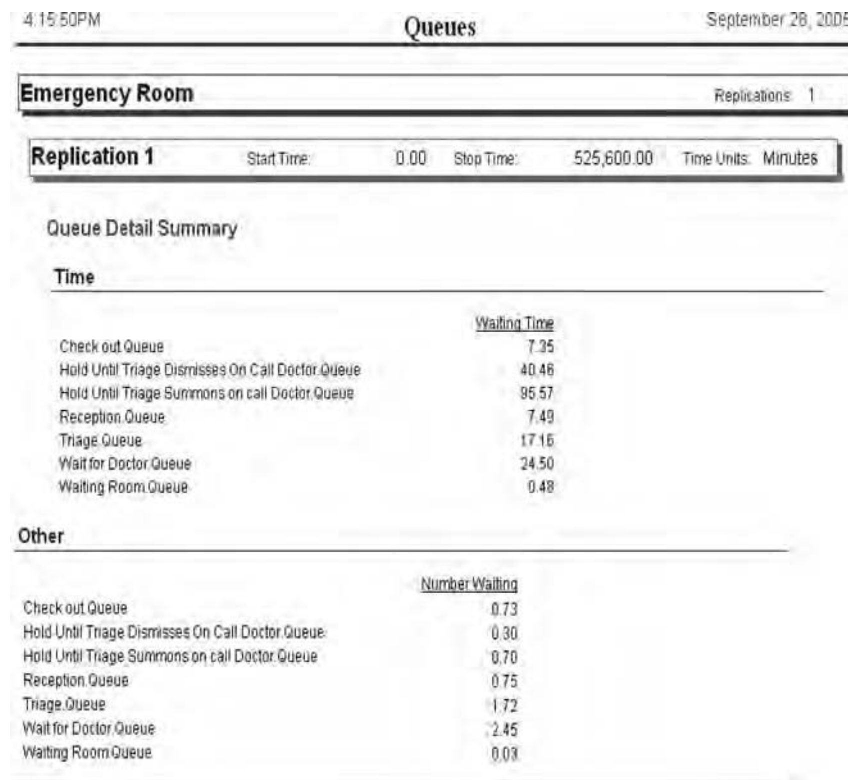
- The *Inst Util* column computes the time averages of instantaneous utilization. The *instantaneous utilization* of a resource is the fraction of busy resources to total available resources at any given time. For example, the instantaneous utilization of doctors is very high (95%), and would have been even higher had an on-call doctor not been available.
- The *Num Busy* column computes the time average of the number of busy resources.
- The *Num Sched* column computes the time average of the number of available resources.
- The *Num Seized* column computes the number of times a resource is seized.
- The *Sched Util* column computes the ratio of *Num Busy* to *Num Sched.*

The *Frequencies* segment displays distribution-related statistics of the random process of the number of busy doctors over time. These statistics are computed over the simulation horizon as follows:

- The *Distribution of Doctors* column lists all values (states) that can be assumed by the number of busy doctors.
- The *Number Obs* column tallies the observed frequency of each state listed.
- The *Average Time* column computes the average holding time in each state listed (i.e., average time spent in a state).
- The *Standard Percent* column computes the ratio of time spent in a state to the total simulation horizon and displays the ratio as a percentage. Note that these numbers provide an estimate of the probability distribution of the number of busy doctors.
- The *Restricted Percent* column is similar to the *Standard Percent* column except that some states may be excluded. Note that these numbers provide an estimate of the conditional probability distribution of the number of busy doctors, given that some states are excluded. Observe that in our case the two columns are identical since no exclusion was specified (in the *Statistic* module spreadsheet).

An examination of the *Frequencies* table clearly shows that the emergency room doctors are severely overworked! Indeed, this observation is consistent with the high doctor utilization in the *Usage* section.

Figure 5.43 displays waiting line statistics in the emergency room in terms of average waiting times and average number of patients in lines.



4:15:50PM **Queues** September 28, 2005

Emergency Room Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 525,600.00 Time Units: Minutes

Queue Detail Summary

Time

	Waiting Time
Check out Queue	7.35
Hold Until Triage Dismisses On Call Doctor Queue	40.46
Hold Until Triage Summons on call Doctor Queue	95.57
Reception Queue	7.49
Triage Queue	17.16
Wait for Doctor Queue	24.50
Waiting Room Queue	0.48

Other

	Number Waiting
Check out Queue	0.73
Hold Until Triage Dismisses On Call Doctor Queue	0.30
Hold Until Triage Summons on call Doctor Queue	0.70
Reception Queue	0.75
Triage Queue	1.72
Wait for Doctor Queue	2.45
Waiting Room Queue	0.03

Figure 5.43 Waiting lines statistics in the emergency room model.

These include two types of statistics:

- Patient waiting times at various stages of their sojourn in the emergency room.
- The on-call doctor's state (on duty and off duty), which are computed from waiting times of the administrator entity.

The *Time* section shows that the averages of waiting times in the reception queue and the checkout queue are comparable and relatively significant at around 7 minutes. Indeed the utilization of the receptionists is quite high at 80% (see Figure 5.42). However, the corresponding averages of patients waiting in those lines, shown in the *Other* section, are reasonable (under one person on average). In a similar vein, average patient waiting times in the triage queue and for treatment by a doctor are high (17 minutes and 25 minutes, respectively), and lead to higher average numbers of waiting patients in those lines. This is an expected consequence of the fact that doctors are overworked. By contrast, average waiting times for nurses in the waiting room is very low (less than a minute), as is the average number of such patients (just 0.03). This fact is borne out by the low utilization of nurses (26%) in Figure 5.42. Finally, the averages of times on duty and off duty of the on-call doctor show that the former is about half the latter. Thus, the on-call doctor is not heavily utilized.

5.8 SPECIFYING TIME-DEPENDENT PARAMETERS VIA A SCHEDULE

Our examples have so far assumed that random phenomena (e.g., arrivals, services, etc.) are modeled as variates from a fixed probability law that does not change in time, so that the underlying process is *stationary* (see Section 3.9). However, it is quite common in practice for the underlying probability law to vary in time, in which case the process is *nonstationary* (time dependent). For example, researchers debate whether recent indications of rising global temperatures are ordinary fluctuations of a stationary temperature process or an indication of a change in the underlying probability law corresponding to a nonstationary temperature process (global warming). On the other hand, many types of arrival events (e.g., customer arrivals in stores, banks, or factories) are known to be nonstationary. Typical examples include the "rush hour" phenomenon (temporary heavy traffic) or the "ebb hour" phenomenon (temporary light traffic). Thus, a bank operation may experience "rush hour" periods in the morning (customers stopping by on their way to work) and at lunch time (customers using part of their lunch time for banking), while mid-morning hours may become "ebb hour" periods. This pattern may recur day after day, with some random fluctuations. A common special case of nonstationarity is when the parameters of the underlying probability law change in time. For example, the arrival rate of a Poisson process may change in the course of a day, season, and so on, in accordance with "rush hour" or "ebb hour" phenomena. This section discusses Arena facilities that permit the modeling of time-dependent parameters of random processes (e.g., arrivals, service, and resource capacities), by varying such parameters over time via a schedule specification.

We first illustrate this facility by a time-dependent arrival specification. The Arena *Create* module provides a *Schedule* option in its *Time Between Arrivals* section, but only for exponential interarrival times, as depicted in Figure 5.44.

The *Schedule Name* field specifies that entity generation is governed by a *Schedule* module called *Schedule 1*. The spreadsheet view of the *Schedule 1* module is displayed in Figure 5.45.

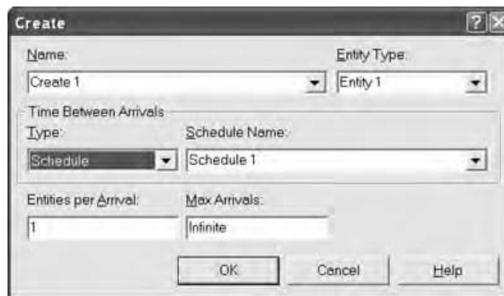


Figure 5.44 Dialog box of a *Create* module with the *Schedule* option.

Schedule - Basic Process

	Name	Format Type	Type	Time Units	Scale Factor	Durations
1	Schedule 1	Duration	Arrival	Hours	1.0	24 rows

Figure 5.45 Spreadsheet view of the *Schedule* module *Schedule 1*.

The *Arrival* option is selected in the *Type* column to indicate that the schedule is dedicated to an arrival process. The *Format Type* column declares whether the schedule is specified by durations or by a calendar, and the *Scale Factor* column may be used to scale all schedule magnitudes (the default is 1.0, meaning no scaling). Under the *Durations* heading is a button labeled *24 rows*, which specifies a time span consisting of 24 consecutive time slots (intervals). Clicking this button pops up a dialog box, depicted in Figure 5.46.

The graph of Figure 5.46 specifies the hourly arrival rates over a time span of 24 hours, depicted as bars over 24 time slots; these correspond to the 24 rows (or durations) alluded to in Figure 5.45. More specifically, the horizontal axis is a time axis divided into consecutive 1-hour time slots (intervals), while the vertical axis is a magnitude axis for the associated arrival rates. Using mouse clicks at appropriate points within the bars of the graph, the modeler can visually specify the requisite hourly arrival rates.

Note carefully that Arena selects the requisite arrival rate only when the most recent arrival "shows up" in the model. This means that if no arrival fell within a given time slot, then no arrivals would be generated with that slot's arrival rate! In other words, if an interarrival interval wholly contains a slot, then no arrivals from such a slot will be scheduled. This might create some modeling problems when the arrival rates vary significantly from slot to slot.

The modeler may interpret the actual time at will, since the first time slot (labeled by *Day 1 00:00:00*) is just a convention for the time origin. In our example, the first time slot in Figure 5.45 actually corresponds to the time interval 8:00 A.M. to 9:00 A.M.—a morning "rush hour" period with a high traffic rate. The hourly rate progressively ebbs towards midday, and it then picks up gradually in the afternoon, reaching its peak in the evening "rush hour" of 5:00 P.M. to 6:00 P.M. It then ebbs again during the night, bottoming out at 12:00 A.M. Finally, it rises again towards the morning "rush hour" of the next day (*Day 2 00:00:00*). This pattern of time-dependent arrival rates repeats on each subsequent day.

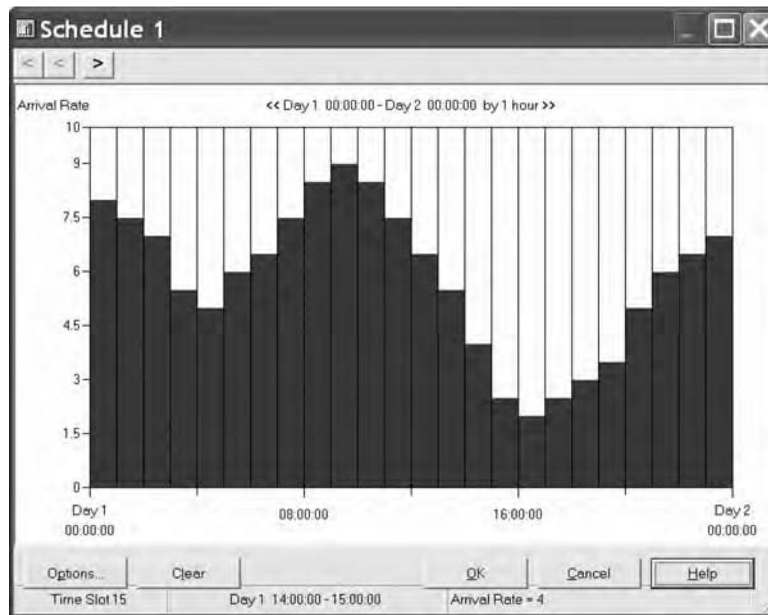


Figure 5.46 Dialog box for schedule *Schedule 1* specifying time-dependent arrival rates.

To customize the *Schedule* module, the modeler clicks the *Options*... button in Figure 5.46 to pop up the dialog box displayed in Figure 5.47. Here, the *X-axis* section has a *Time slot duration* field to specify the time unit (time slot width) on the horizontal axis, and a *Range* field to specify the number of time slots in the graph. The *Calendar speed* field is a user interface parameter that specifies the scrolling speed of the graph (using the scrolling buttons below the title bar in Figure 5.46), when the graph does not fit into its canvas. The *Y-axis* section has *Maximum* and

Figure 5.47 Dialog box of the *Options*... field of schedule *Schedule 1*.

Minimum field values for the vertical axis. The *Snap Spacing* field is a user interface parameter to specify the granularity of arrival rates (on the vertical axis) via button clicks as described above.

The replication length may precisely equal the time span of the graph in Figure 5.46, but in the majority of cases it is either shorter or longer than the specified time span. If the replication length is shorter, then the rest of the graph is simply ignored. Otherwise, the information in the *When at end of schedule* section specifies how to assign arrival rates beyond the time span of the schedule. Two options (radio buttons) are provided:

- The *Repeat from beginning* option simply cycles back and repeats the schedule from its beginning for each time span. In our example, the schedule in Figure 5.46 is a daily schedule, to be repeated for each simulated day.
- The *Remain at capacity* option provides a data field specifying a fixed arrival rate. The replication will follow the schedule for its first time span. However, once the simulation clock exceeds the time span, that fixed arrival rate will be in effect for the rest of the replication.

In a similar vein, the user can schedule time-dependent resource capacities. To this end, the user selects the *Based on Schedule* option in the spreadsheet view of the *Resource* module, and then proceeds to specify a resource capacity schedule analogously to the arrivals example above. What happens when resource capacity is decreased while the resource is in use? To handle such eventualities, Arena offers users three options in the *Schedule Rule* column of the *Resource* spreadsheet view when the *Based on Schedule* option is selected:

- The *Ignore* option starts the time duration of the schedule change immediately, but each "excess" resource is removed only after it is released by the entity currently seizing it.
- The *Wait* option waits to start the time duration of the schedule change until the last "excess" resource is released by the entity currently seizing it.
- The *Preempt* option starts the time duration of the schedule change immediately, but "interrupts" processing by promptly releasing "excess" resources and returning the seizing entities to their queues.

Incidentally, machine failures in Arena are implemented using these same options (see Section 11.7). Finally, a schedule specification can be similarly used in Arena to vary the value of any parameter over time by selecting the *Other* option in the *Type* column of the spreadsheet view of the *Schedule* module.

EXERCISES

1. *Press operation*. The press department of an automobile manufacturing facility runs two main operations, each with its own press machine: *front-plate press* operation and *rear-plate press* operation. These operations can be performed in any order, but both have to be performed for each arriving plate. Plates (jobs) arrive randomly and their interarrival times are exponentially distributed with mean 5 minutes. The service time in the *front-plate press* operation is distributed iid Unif(1, 5) minutes, and in the *rear-plate press* operation it is distributed iid Unif(2, 6) minutes. A plate joins the queue of the press operation with the least number of plates waiting at that time (since there is no sequencing requirement),

and on completion joins the queue of the other press operation after which it departs from the system. Finally, the press department is a three-shift facility running 24 hours a day.

a. Develop an Arena model of the press department, and simulate it for one year.
b. Estimate the following statistics:
   - Average time arriving plates spend in the press department
   - Utilization of the press machine in each operation
   - Average queue delay at each operation
   - Average time in the press department of those arriving plates that join first the *rear-plate press* operation, and then proceed to the *front-plate press* operation

2. *Electrolytic forming process*. An expensive custom-built product goes through two stages of operation. The first stage is an *electrolytic forming* process, served by two independently operating forming machines, where the product is built in a chemical operation that must conform to precise specifications. The second stage is a *plating operation* in which the product is silver plated. Customer orders arrive with interarrival times distributed iid Tria(3, 7, 14) hours, and join a queue in front of the forming process. The electrolytic forming processing time is distributed iid Unif(8, 12) hours. The silver-plating process also has a queue in front of it. Plating time is distributed iid Unif(4, 8) hours. The variability in the processing times is due to design variations of the incoming orders.

   The two processes do not perform perfectly. In fact, 15% of the jobs that emerge from the forming process and 12% of the jobs that emerge from the plating process are defective and have to be reworked. All defective jobs are sent to a single rework facility, where design modifications and corrections are performed manually. However, plating reworks have a lower priority than forming modifications. Plating rework times are distributed iid Unif(15, 24) hours, while forming reworks are distributed iid Unif(10, 20) hours. Jobs departing from the rework facility go back to the process they came from to redo the operation found defective. Jobs that successfully complete the plating process leave the facility. Note that a job may go back and forth between a process and the rework operation any number of times.

   a. Develop an Arena model of the electrolytic forming process, and simulate it for 1 year (24 hours of continuous operation).
   b. How busy are each of the two operations and the rework facility?
   c. What are the expected delays in process queues and the rework facility?
   d. What is the expected job flow time throughout the entire facility?
   e. Suggest a change in the system to reduce (even slightly) the expected job flow time. Run the modified model and compare the job flow statistics.

3. *Supermarket cashier management*. A supermarket is open 24/7 and operates in 3 shifts: first shift from 8:00 A.M. to 4:00 P.M., second shift from 4:00 P.M. to 12:00 A.M., and third shift from 12:00 A.M. to 8:00 A.M. Customers arrive according to a Poisson process with shift-dependent arrival rates, and their shopping times (excluding checkout) are iid but shift dependent. Consequently, the supermarket management assigns variable numbers of cashiers per shift. The arrival, shopping, and cashier parameters are displayed in the following table.

After shopping, customers queue up in a single line for checkout. Checkout times of customers are iid, but shopping-time dependent as follows: a customer's checkout

Shift Number	Arrival Rate (customers per minute)	Shopping Time Distribution (minutes)	Number of Cashiers
1	16.8	Unif(5, 15)	2
2	24.0	Unif(15, 40)	4
3	0.7	Unif(1, 5)	1

time is 2 minutes plus a random fraction of its shopping time, where the fraction is iid triangular between 20% and 30%, with a most likely value of 25%. Finally, a customer is assigned to the shift during which it departs from the supermarket (note that a customer may arrive during one shift and depart during a subsequent one).

a. Develop an Arena model of the supermarket, and simulate it for 30 days.
b. Compute the average, variance, and squared coefficient of variation of customer waiting times for a cashier (excluding checkout processing times).
c. What is the per-shift average sojourn time of customers in the supermarket?
d. What are the overall instant and scheduled cashier utilizations?

This page intentionally left blank

Chapter 6

Model Testing and Debugging Facilities

Recall that Arena is an object-oriented modeling tool with a graphical user interface (GUI), through which modeler and machine interact. More specifically, modeler (foreground) actions via the GUI prompt Arena to respond with (background) actions. For all practical purposes, each interaction is a pair of foreground/background actions, which can be thought of as taking place in lockstep in the sense that the previous pair must complete before the next pair begins processing.

A modeling and simulation tool ought to provide adequate facilities for model testing and debugging in order to help the modeler ensure that the simulation model under construction is free of syntactic and logical errors. Such modeling activities may be viewed as part of model verification (see Step 4 of Section 1.5), namely, checking that the simulation code correctly represents the conceptual model. Recall that only once the model is verified does the modeler proceed with model validation (see Step 5 of Section 1.5), namely, checking that the conceptual model (now properly represented by the simulation code) adequately captures reality.

Arena provides a rich set of facilities to support model testing and debugging throughout the life cycle of a simulation model. These facilities allow the user to access Arena objects, code, and data that underlie the model image on the screen (see Figure 5.1), such as model parameters and SIMAN code (blocks). It further allows the modeler to inspect model runs, either in visual animation mode or in textual command mode. This chapter reviews the Arena testing and debugging facilities supported by the *Run Interaction* toolbar. For more information, consult Kelton et al. (2004), Rockwell Software (2005), and the Arena help menus, discussed in Section 6.6.

6.1 FACILITIES FOR MODEL CONSTRUCTION

The most fundamental modeler/machine interaction in Arena is model construction. The modeler builds a model in the foreground by dragging and dropping module objects from template panels in the *Project* bar onto the model window canvas, and Arena reacts

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

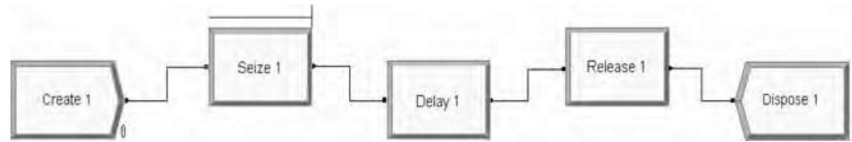


Figure 6.1 A simple Arena model of a single-server queue (file *test1.doe*).

in the background by constructing the associated SIMAN code. The model is saved in a file with the extension *.doe*. The automatically generated SIMAN code is placed in two files, both accessible via the SIMAN option of the pull-down *Run* menu:

- A model file (with the *.mod* extension) contains the model logic in the form of SIMAN blocks. It is well worth the effort to become familiar with SIMAN code, as this knowledge can significantly speed up model debugging. The extensive Arena *Help* menu further facilitates this task.
- An experiment file (with the *.exp* extension) lists all model elements, including project information, resources, queues, statistical outputs, and many others. It is important to examine this file and double-check model components, especially during the model construction process.

As an example, consider the simple model of Figure 6.1 depicting an Arena model called *test1*, which was accordingly placed by Arena in file *test1.doe*.

Model *test1* consists of a sequence of *Create*, *Seize*, *Delay*, *Release*, and *Dispose* modules in tandem, and has a resource called *Resource 1* with a queue called *Seize 1.Queue*. Panel 6.1 below displays the *test1.mod* file of the model *test1* in file *test1.doe*.

```
;       Model statements for module: Create 1
;
5$      CREATE,          1,HoursToBaseTime(0.0),
Entity 1:HoursToBaseTime(EXPO(1)):NEXT(6$);
6$      ASSIGN:          Create 1.NumberOut=Create 1.NumberOut + 1:
                         NEXT(0$);
;
;       Model statements for module: Seize 1
;
0$      QUEUE,           Seize 1.Queue;
        SEIZE,           2,Other:Resource 1,1:NEXT(10$);
10$     DELAY:           0.0,,VA:NEXT(2$);
;
;       Model statements for module: Delay 1
;
2$      DELAY:           EXPO( 0.8 ),,Other:NEXT(3$);
;
;       Model statements for module: Release 1
;
3$      RELEASE:         Resource 1,1:NEXT(4$);
;
```

```
;       Model statements for module: Dispose 1
;
4$      ASSIGN:         Dispose 1.NumberOut=Dispose 1.NumberOut + 1;
11$     DISPOSE:        Yes;
```

Panel 6.1 Contents of file *test1.mod*

An inspection of Panel 6.1 reveals that each line starts with an integer followed by a dollar sign. These are Arena-assigned labels for blocks, which are used to identify the blocks in SIMAN code for control transfer ("go-to") by a *NEXT* instruction. A walk-through of Panel 6.1 reveals the following sequence of SIMAN blocks that implement the mapping from Arena modules to SIMAN blocks.

1. The *Create1* module is mapped into two SIMAN blocks: *CREATE* and *ASSIGN*. The *CREATE* block models a random source of incoming entities with an exponential interarrival time distribution of mean 1. The *ASSIGN* block counts the total number of created entities.
2. The *Seize1* module is mapped into two SIMAN blocks: *QUEUE* and *SEIZE*. The *QUEUE* block, called *Seize 1.Queue*, holds incoming entities that contend for access to the *SEIZE* block for one unit of the resource called *Resource 1* (entities contend with priority 2).
3. The *Delay1* module is mapped into a SIMAN *DELAY* block that models an exponentially distributed service time with mean 0.8 time units for the entering entity.
4. The *Release1* module is mapped into a SIMAN *RELEASE* block that releases one unit of the *Resource 1* resource previously seized by the entering entity.
5. The *Dispose1* module is mapped into two SIMAN blocks: *ASSIGN* and *DISPOSE*. The *ASSIGN* block keeps track of the outgoing entity count, while the *DISPOSE* block simply disposes of the entering entity.

Note that Arena modules are mapped to a variable number of SIMAN blocks. For example, the *Create* module is mapped into two SIMAN blocks, while the *Release* module is mapped into just one. Observe that connections between modules are represented by the keyword *NEXT* and a parenthesized number followed by a $ sign. Recall that these numbers serve as block labels. As a matter of fact, the blocks in a *.mod* file are implicitly numbered starting at 1; the number 0 stands for the "environment" (the "complement" of the system). These numbers are not shown in the file, but they are useful in accessing the blocks from the *Run Controller* for debugging purposes, as will be shown later in this chapter.

Next, Panel 6.2 displays the *test1.exp* file of the model *test1* from file *test1.doe*.

```
PROJECT,      "Testing mod/exp files","Industrial Engineer", ,,No,
              Yes,Yes,Yes,No,No,No,No,No,No;
VARIABLES:    Dispose 1.NumberOut,CLEAR(Statistics),CATEGORY
              ("Exclude"):
              Create 1.NumberOut,CLEAR(Statistics),CATEGORY
              ("Exclude");
QUEUES:       Seize 1.Queue,FIFO,,AUTOSTATS(Yes,,);
PICTURES:
RESOURCES:    Resource 1, Capacity(1),,,COST(0.0,0.0,0.0),CATEGORY
              (Resources),,AUTOSTATS(Yes,,);
```

```
DSTATS:      NR(Resource 1),Resource 1 Utilization,"",
             DATABASE(,"Time Persistent","User Specified",
             "Resource 1 Utilization");
REPLICATE,   1,,,Yes,Yes,,,,24,Hours,No,No,,,Yes;
ENTITIES:    Entity 1,
             Picture.Report,0.0,0.0,0.0,0.0,0.0,0.0,AUTOSTATS
             (Yes,,);
```

Panel 6.2 Contents of file *test1.exp*.

A walk-through of Panel 6.2 reveals the following sequence of items:

1. The *PROJECT* item provides model identification.
2. The *VARIABLES* item lists each model variable, along with its initial value (default is 0).
3. The *QUEUES* item lists model queues, along with their queueing discipline (default is FIFO) and capacities (default is infinity).
4. The *PICTURES* item lists entity picture names.
5. The *RESOURCES* item lists model resources, along with their capacity, schedule, and failure/repair information.
6. The *DSTATS* item lists statistics specified with the *Time Persistent* option in the *Type* column of the *Statistic* spreadsheet, along with their attributes. The *test1.exp* file would also list statistics specified with the *Tally* option in the *Type* column of the *Statistic* spreadsheet under a *TALLIES* item for each of its counterparts in the *.mod* file; here, that list is empty, so the *TALLIES* item is omitted. A *FREQUENCIES* item would similarly be included for statistics specified with the *Frequency* option in the *Type* column of the *Statistic* spreadsheet, analogously to the *TALLIES* item.
7. The *REPLICATE* item lists the data from the *Replication Parameters* dialog box of the *Setup* option in the *Run* menu.
8. The *ENTITIES* item lists each entity name and the corresponding picture name.

It should now be clear to the reader that the *.mod* file and the *.exp* file reflect the information in the *.doe* file. Furthermore, a simulation model should be as compact as possible to increase its running efficiency: Since every module is mapped into a set of SIMAN blocks (statements), which are compiled to create an executable file, models should be "cleansed" of any unnecessary modules.

6.2 FACILITIES FOR MODEL CHECKING

Arena can be instructed to perform a number of checks on model logic to detect syntactic errors and animation errors. Most errors are detected at compile time, and the rest at run time. However, catching logical modeling errors is the responsibility of the modeler.

Model checking can be initiated in three ways:

- Selecting the *Check Model* option from the *Run* menu
- Clicking the *Check* button on the *Run Interaction* toolbar (see Section 6.3.1)
- Clicking the *Go* button in the *Standard* toolbar, which initiates a model check, and if successful, it starts a model run

When Arena detects an apparent error, it pops up a dialog box, titled *Errors/Warnings*, and generates a problem message, along with possible causes. This dialog box displays five buttons as follows:

- The *Previous* button pops up the previous error information.
- The *Next* button pops up the next error information.
- The *Find* button locates modules containing errors.
- The *Edit* button allows the modeler to correct errors by editing module fields.
- The *Close* button closes the dialog box.

The user then interacts with the dialog box until all errors (and usually warnings) are cleared.

6.3 FACILITIES FOR MODEL RUN CONTROL

Arena provides elaborate run-control facilities that support a wide variety of run-time interactions designed to monitor a simulation run and control the way it evolves over time. More specifically, the modeler can interrupt the run and make assignments to attributes and variables, or arrange for the run to stop whenever user-prescribed conditions are satisfied. Furthermore, the modeler can turn on visual animation and watch the model evolve in time, while monitoring variable values and entity movements in the model. Clearly, the use of these facilities is essential in verifying model logic from model behavior.

6.3.1 Run Modes

Arena supports VCR-like functionality for running a replication in various modes. This functionality is accessible via options in the *Run* menu or the corresponding buttons on the *Standard* toolbar. These buttons bear the familiar VCR-like icons as follows:

- The *Go* button initiates or resumes a replication.
- The *Step* button steps through the replication event by event. Each time it is clicked, the replication runs, processing the most imminent event (see Section 2.1), and then the replication is paused.
- The *Fast-Forward* button suspends model animation and runs the replication to completion.
- The *Pause* button pauses the replication. The modeler can resume execution by clicking on any of the buttons above.
- The *Start Ove*r button initializes a new replication.
- The *End* button terminates the replication.

6.3.2 Mouse-Based Run Control

The functionality of mouse-based run control is accessible either via the *Run Control* option of the *Run* menu, or the *Run Interaction* toolbar, shown in Figure 6.2.

Figure 6.2 Arena *Run Interaction* toolbar.

This toolbar consists of six buttons from left to right as follows:

1. The *Check* button performs model checking as described in Section 6.2.
2. The *Command* button opens a command window for textual interaction with the model. Command-line run control will be reviewed in Section 6.3.3.
3. The *Break* button opens a dialog box that allows the modeler to insert four types of break point: *Break on Time*, *Break on Condition*, *Break on Entity*, *Break on Module*, and *Break on Calendar Date Time*. Break-point parameters are specified in the appropriate fields of the dialog box. Whenever the run "hits" a break point, its execution is paused, allowing the modeler to inspect the state of the model. To resume execution, the modeler clicks any of the VCR-like buttons or menu options, as explained in Section 6.3.1.
4. The *Watch* button opens a window that permits the modeler to add, delete, or edit expressions so as to watch their dynamic values in the course of a run.
5. The *Break on Module* button sets or clears a break point in a selected canvas module. Whenever an entity enters the module or resumes executing its logic, the run is paused.
6. The *Animate Connectors* button turns on and off entity animation over model connectors.

The functionality of the *Run Interaction* toolbar can also be accessed from the *Run* menu.

6.3.3 Keyboard-Based Run Control

The *Command* button in the *Run Interaction* toolbar is very handy for keyboard-based interaction. It can also be accessed by clicking the *Run Control* option in the *Run* menu, and then selecting the *Command* option. Either action opens the *Command* window below the Arena model canvas as illustrated in Figure 6.3.

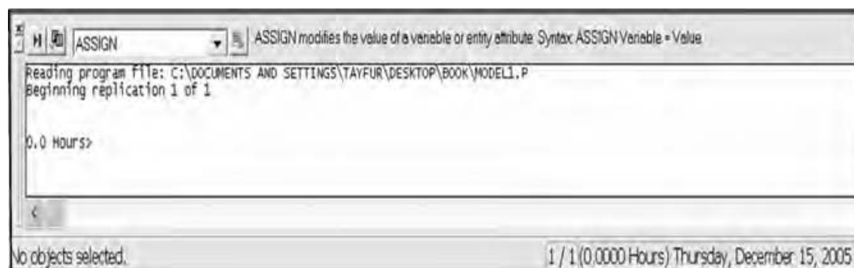


Figure 6.3 Arena *Command* window.

The top portion of the *Command* window displays the five graphical objects from left to right as follows:

1. The *Block Step* button is used for advancing the simulation one step (event) at a time.
2. The *Enable Block Trace* toggle button turns tracing on and off.
3. The next object is a pull-down menu allowing the user to select a debugging command.
4. The next button inserts the selected command at the bottom of the text panel of the *Command* window prefixed by a prompt that consists of the current simulation time followed by the greater than (>) character.
5. The last object is a text annotation describing the selected command and its syntax.

Once the *Command* window is opened, the modeler can also type in commands directly after the prompt. A list of frequently used commands follows:

1. *GO* starts or resumes a run.
2. *GO UNTIL* starts the run and stops it at the specified time. For instance,

   ```
   GO UNTIL 100
   ```

   starts a run and stops it at time $TNOW = 100$.
3. *END* terminates the run and produces a summary report.
4. *QUIT* does the same without producing a summary report.
5. *ASSIGN* allows the user to assign values to variables or entity attributes. For example, suppose the simulation is stopped at time 25.6. Then, the command

   ```
   25.6>   ASSIGN   Target Inventory = 250
   ```

   assigns the value of 250 to the variable *Target Inventory* at this point in time.
6. *SHOW* displays the current values of expressions, variables, and attributes. For instance, the commands

   ```
   SHOW NQ(Q1) + NQ(Q2)
   SHOW NQ(*)
   ```

   display, respectively, the sum of the number of entities in queues *Q1* and *Q2*, and the current length of each queue in the model. The asterisk (*) is the wild card character, and stands for "all."
7. *VIEW* displays a broad range of model data. For instance, the commands

   ```
   VIEW SOURCE 21..30
   VIEW ENTITY 5
   VIEW QUEUE Mac_q
   ```

   display, respectively, the SIMAN code of block numbers 21 through 30 (recall that SIMAN blocks are implicitly numbered from 1 onward), the attributes of entity 5 (recall that all entities are numbered consecutively from 1 at creation time), and the queue called *Mac_q*, along with all its entities and their attributes.
8. *SET* commands perform a number of important functions in the *Run Controller*, which include the following:
   a. *SET BREAK* sets break points at specified locations in the model. When an entity hits such a break point, Arena suspends the run and allows the modeler to "poke" into the model's state. The run can be resumed when the modeler types the appropriate command. For example,

```
SET BREAK 25
```

sets a break point in block number 25.

b. *SET WATCH* permits the modeler to monitor the value of a variable, attribute, or expression (in particular, whether a prescribed condition is satisfied). For example, the commands

```
SET WATCH NQ(Mac_q)
SET WATCH NR(Mac) > 2
```

suspend the run, respectively, whenever the size of queue *Mac_q* changes, or whenever the number of busy servers in resource *Mac* exceeds 2.

c. *SET TRACE* activates the SIMAN trace to monitor the flow of entities through blocks, and the assignments made in each block monitored. For example,

```
SET TRACE *
```

displays a complete history of entity movements and variable assignments in the course of a run from the time this command was issued until the run is suspended (see the example in Panel 6.3).

9. *CANCEL* cancels the *Run Controller* option that constitutes this command's argument. For instance,

```
CANCEL WATCH *
```

cancels all the *WATCH* requests issued at an earlier time. In a similar vein, the commands

```
CANCEL TRACE *
CANCEL BREAK 3
```

cancel, respectively, all prior trace requests, and the break point set on block 3.

6.4 EXAMPLES OF RUN TRACING

This section illustrates trace reports produced by Arena in response to modeler *TRACE* commands. The examples demonstrate that the modeler can acquire and review extremely detailed information on any simulation run. The resultant *TRACE* reports list the temporal sequence of Arena's internal events and the actions performed by each event at its time of occurrence. These actions include assignment of variables and attributes, seizing and releasing of resources, and generally, the movement of Arena entities. Recall that Arena always tries to "move" entities through blocks as far as they can go before being "blocked" from further forward movement (e.g., by a *DELAY* block). In fact, following each event, Arena attempts to push entities forward, until none can advance in the model, at which point the simulation clock is advanced to the most imminent event time (recall Section 2.1), and its processing triggers another round of entity movements.

6.4.1 EXAMPLE: OPEN-ENDED TRACING

Panel 6.3 displays a portion of an open-ended history trace of a run produced by issuing the command *SET TRACE ** at time 0.

```
Time: 0.0 Entity: 2
1 5$       CREATE
                 Entity Type set to Entity 1
                 Next creation scheduled at time 1.36794
                 Batch of 1 Entity 1 entities created
2 6$       ASSIGN
                 Create 1.NumberOut set to 1.0
3 2$       QUEUE
                 Entity 2 sent to next block
4          SEIZE
                 Tally Seize 1.Queue.WaitingTime recorded 0.0
                 Seized 1 unit(s) of resource Resource 1
5 10$      DELAY
                 Delayed by 0.0 until time 0.0
6 1$       DELAY
                 Delayed by 0.433114 until time 0.433114
Time: 0.433114 Entity: 2
7 4$       RELEASE
                 Resource 1 available increased by 1 to 1
8 0$       ASSIGN
                 Dispose 1.NumberOut set to 1.0
9 11$      DISPOSE
                 Disposing Entity 2
Time: 1.36794 Entity: 3
1 5$       CREATE
                 Entity Type set to Entity 1
                 Next creation scheduled at time 1.84734
                 Batch of 1 Entity 1 entities created
2 6$       ASSIGN
                 Create 1.NumberOut set to 2.0
3 2$       QUEUE
                 Entity 3 sent to next block
4          SEIZE
                 Tally Resource 1.Queue.WaitingTime recorded 0.0
                 Seized 1 unit(s) of resource Resource 1
5 10$      DELAY
                 Delayed by 0.0 until time 1.36794
6 1$       DELAY
                 Delayed by 1.36281 until time 2.73075
```

Panel 6.3 Trace produced by the command *SET TRACE* *.

An inspection of Panel 6.3 reveals that every line that contains a SIMAN block starts with an integer. That integer is an Arena-assigned block number used in *TRACE* reports. Finally, the actions occurring at each SIMAN block (on entity entry there) are listed in indented format.

A walk-through of Panel 6.3 should clarify how the *Run Controller* displays a temporal trace of a model's run history. Starting at the first line, the sequence of trace events unfolds as follows:

- At time 0.0, *Entity 2* (of type *ENTITY 1*) is created at the *CREATE* block (this type name was assigned as a default by Arena, but the modeler can, of course, change it). The next entity (*Entity 3*) arrival is scheduled to occur at time 1.36794. Note that entities are created singly (in batches of size 1).
- *Entity 2* next enters an *ASSIGN* block and triggers the assignment

$$CREATE1.NUMBEROUT = 1$$

  where *CREATE 1.NUMBEROUT* is an Arena variable that keeps track of the number of entities entering the model at block *CREATE 1*. Arena collects this statistic automatically in order to produce entity statistics when the run ends.
- *Entity 2* then proceeds through the *QUEUE*, *SEIZE*, and *DELAY* blocks belonging to the *Seize* module. It enters the queue called *Seize 1.Queue* at the *SEIZE* block, and having recorded its waiting time there, it immediately seizes the resource called *Resource 1*. (Note that if *Resource 1* were busy, then *Entity 2* would wait in *Seize 1. Queue* until *Resource 1* becomes free.) The *DELAY* block causes a delay of duration 0.
- The next *DELAY* block belongs to a *Delay* module and represents the elapsing of entity processing time. *Entity 2* is delayed for processing a duration of 0.433114 time units. Note carefully that the simulation clock (system variable *TNOW*) has been set to 0.0 until this *DELAY* block. To see why, note that because the *Resource 1* resource is initially idle, the just created *Entity 2* manages to "push its way" through the block sequence until it is "detained" at this *DELAY* block for processing by *Resource 1*.
- Next, the simulation clock is advanced to *TNOW* = 0.433114, and *Entity 2* completes its processing at *Resource 1* and releases this resource. It then enters an *ASSIGN* block where it increments by 1 the counter variable called *Dispose 1.NumberOut*.
- Finally, *Entity 2* enters the *DISPOSE* block in which it is removed from the model.
- The simulation clock is next advanced to *TNOW* = 1.36794 at which point the next arriving entity (*Entity 3*) is created in the *CREATE* block and enters the model. It then embarks on its own sojourn through the model.

6.4.2 Example: Tracing Selected Blocks

Panel 6.4 displays a portion of the history trace pertaining to blocks 1 through 4 only. This trace was produced by issuing the command *SET TRACE BLOCKS 1..4* at time 0.

```
0.0>set trace blocks 1..4
0.0>go until 4
Time: 0.0 Entity: 2
1 5$    CREATE
              Entity Type set to Entity 1
              Next creation scheduled at time 1.36794
              Batch of 1 Entity 1 entities created
2 6$    ASSIGN
              Create 1.NumberOut set to 1.0
```

```
3 1$    QUEUE
                Entity 2 sent to next block
4       SEIZE
                Tally Seize 1.Queue.WaitingTime recorded 0.0
                Seized 1 unit(s) of resource Resource 1
Time: 1.36794 Entity: 3
1 5$    CREATE
                Entity Type set to Entity 1
                Next creation scheduled at time 1.84734
                Batch of 1 Entity 1 entities created
2 6$    ASSIGN
                Create 1.NumberOut set to 2.0
3 1$    QUEUE
                Entity 3 sent to next block
4       SEIZE
                Tally Seize 1.Queue.WaitingTime recorded 0.0
                Seized 1 unit(s) of resource Resource 1
```

Panel 6.4 Trace produced by the command *SET TRACE BLOCKS 1..4*.

An examination of Panel 6.4 reveals that it starts with two user-issued commands (the first sets up a trace and the other specifies its duration), and the simulation time when they were issued. An inspection of the block numbers in Panel 6.4 verifies that only blocks 1, 2, 3, and 4 appear in the *TRACE* report; to wit, the first integer in each row is in the range 1 to 4 (this pattern continues in the extended trace). Thus, to produce this report, Arena has extracted only the relevant block information for inclusion in the report and discarded all other information. Such filtered traces are useful in focusing on the action in selected blocks, primarily for debugging purposes.

6.4.3 Example: Tracing Selected Entities

Panel 6.5 displays a portion of the run history pertaining to *Entity* 2 only. This trace was produced by issuing the command *SET TRACE ENTITY 2* at time 0.

```
0.0>set trace entity 2
*** Trace set on entity 2
0.0>go until 4
Time: 0.0 Entity: 2
1 5$     CREATE
                   Entity Type set to Entity 1
                   Next creation scheduled at time 1.36794
                   Batch of 1 Entity 1 entities created
2 6$     ASSIGN
                   Create 1.NumberOut set to 1.0
3 1$     QUEUE
                   Entity 2 sent to next block
```

```
                    Tally Resource 1.Queue.WaitingTime recorded 0.0
                    Seized 1 unit(s) of resource Resource 1
5 10$    DELAY
                    Delayed by 0.0 until time 0.0
6 0$     DELAY
                    Delayed by 0.541393 until time 0.541393
Time: 0.541393 Entity: 2
7 3$     RELEASE
                    Resource 1 available increased by 1 to 1
8 4$     ASSIGN
                    Dispose 1.NumberOut set to 1.0
9 11$    DISPOSE
                    Tally Entity 1.TotalTime recorded 0.541393
                    Disposing entity 2
Break at time: 4.0
```

Panel 6.5 Trace produced by the command *SET TRACE ENTITY 2*.

An examination of Panel 6.5 reveals that the second line is a confirmation of the *SET TRACE* command in the first line. An inspection of the entity numbers in Panel 6.5 verifies that only *Entity 2* appears there. Thus, to produce this report, Arena has extracted only the relevant entity information for inclusion in the report, and discarded all other information. Again, such filtered traces are useful in focusing on the history of a particular entity through the system, primarily for debugging purposes.

6.5 VISUALIZATION AND ANIMATION

Arena provides support for visualization of models and animation of simulation runs. These may be used to observe and monitor the temporal evolution of the model's state and statistics in the course of a run. Such observation and monitoring activities are used both to debug the model and to understand various phenomena occurring in it. We now proceed to describe the main animation facilities in Arena.

6.5.1 *Animate Connectors* Button

The *Run Interaction* toolbar includes the *Animate Connectors* button. When clicked, this button enables the motion of entities among modules. Entity objects are represented either by default pictures or user-assigned pictures. Tracking the motion of a target entity on the model canvas in the course of a run aids in verifying model logic.

6.5.2 *Animate* Toolbar

Figure 6.4 displays the *Animate* toolbar, which supports visualization and animation of dynamic data values over time.

This toolbar consists of nine buttons as follows (from left to right):

Figure 6.4 Arena *Animate* toolbar.

- The *Clock* button pops up a dialog box that permits the modeler to initialize a "clock" and start it running.
- The *Date* button pops up a dialog box that permits the modeler to initialize a calendar and start it running, showing the current date.
- The *Variable* button pops up a dialog box that permits the modeler to specify an expression and its format. The value of that expression is maintained up to date over the simulation run.
- The *Level* button pops up a dialog box that permits the modeler to specify and display the same information as the *Variable* button, but as a graphical analog "level." This is useful in tracking the value of a bounded expression in order to see at a glance the current relative magnitude. An example is resource utilization (bounded by 0 and 1), which is specified by typing *DAVG(Dstat ID)* in the *Expression* field of the dialog box, where *DAVG (Dstat ID)* stands for the average of the *Time-Persistent* statistic whose Arena number is *Dstat ID*. Similarly, the expression *TAVG(Tally ID)* is used to tally the average value collected by the *Tally* statistic whose Arena number is *Tally ID*.
- The *Histogram* button pops up a dialog box that permits the modeler to specify histogram collection on a specific expression over time during a simulation run.
- The *Plot* button pops up a dialog box that permits the modeler to specify a plot of the value of an expression over time during a simulation run.
- The *Queue* button pops up a dialog box that permits the modeler to specify a queue and display its size over time during a simulation run as a standard T-bar (see Section 5.2).
- The *Resource* button pops up a dialog box that permits the modeler to specify a user-defined resource and display its states over time during a simulation run. A resource can be identified on screen by its icon, which is provided by an Arena default, but may be graphically edited by the modeler.
- The *Global* button pops up a dialog box that permits the modeler to specify a finite-valued expression, and to display its value over time via icons during a simulation run.

6.5.3 *Animate Transfer* Toolbar

The *Animate Transfer* toolbar supports visualization and animation of various transportation devices, such as conveyors, forklifts, AGVs (automatic guided vehicles), and so on. This toolbar will be revisited and explained in Chapter 13.

6.6 ARENA HELP FACILITIES

Arena provides two online help facilities: the *Help* menu and the *Help* button. The main difference between these two facilities is in the way the modeler accesses the requisite information.

6.6.1 *Help* Menu

The *Help* menu is located on the Arena menu bar and contains an option called *Arena Help*, which opens a dialog box with three tabs:

- The *Contents* tab displays a table of contents of Arena help topics.
- The *Index* tab displays a list of Arena keywords in alphabetic order. It allows the user to enter a keyword into a text box to fetch the associated information.
- The *Search* tab allows the user to search for all occurrences of a word or phrase in Arena help files. The user enters search targets into a text box at the top of the tab to fetch the requisite information.

6.6.2 *Help* Button

The *Help* button is located on the Arena *Standard* toolbar and has an arrow and question mark icon. It may be used to obtain information on Arena objects depicted on the canvas. The user clicks the *Help* button and this action pops up a pointer with the same icon. The user then drags that pointer to the desired object, and clicks again on that object to retrieve the relevant information.

EXERCISES

1. *Production line*. Consider the following production line, consisting of two workstations in series: the first performs a *filling process* and the second performs a *capping process*. Both workstations have unlimited buffer space. Real-life filling systems are usually batch operations filling a number of containers simultaneously. However, to simplify modeling, here we will treat each job as a single entity. Job interarrival times at the first workstation are iid uniformly distributed between 1.5 and 5 minutes. Upon completion of the *filling process*, the job joins the buffer of the *capping process*. Job filling times are iid triangularly distributed with parameters 1, 3, and 5 minutes, while capping takes a fixed time of 3 minutes.
   a. Develop an Arena model for the production line and simulate it for 10,000 minutes.
   b. Estimate the following statistics:
      - Average number of jobs in each workstation buffer
      - Utilization of each workstation

   At the beginning of the simulation run, take the following actions:
   c. Print out and examine the Arena *.mod* and *.exp* files.
   d. Add *Variable* windows for
      - Current number of jobs in each queue
      - Output rate of the system (*Hint*: you may use an expression)
      - Total number of entities departing from the system
      - Average system time of departing entities

   e. Add a *Level* picture for the utilization of each workstation.
   f. Add a *Resource* picture for each workstation to monitor the idle and busy states.

g. Using the *Command* window, issue the appropriate command to stop the simulation at $TNOW = 500$. When the simulation stops, issue the appropriate commands to inspect entities in all model queues.
h. At $TNOW = 750$, issue the appropriate trace command to monitor the replication history until $TNOW = 800$. At that time, use the *END* command to display the simulation output.

2. *Message queue in a database server*. Consider a message queue in a database system where user requests (queries) arrive randomly. The message queue has a finite capacity of 64kb. Requests arrive iid according to an exponential interarrival-time distribution with a rate of 1 message per 2.5 seconds. Each message has an associated size (in kilobytes) distributed iid Tria(4, 12, 16). An arriving message will be lost if it cannot fit into the remaining capacity of the message queue. The queries are served on a smaller-size-first basis (which may not be too realistic since the long messages will tend to wait longer in the queue). The service time depends on the message size in such a way that it takes 1 second per 4 kilobyte in the message. (*Hint:* You can model the queue itself as a resource with a capacity of 64 units.)
   a. Develop an Arena model for the message queue system, and simulate it for 5000 seconds.
   b. Estimate the following statistics:
      - Average number of messages in the queue
      - Average delay per query in the queue
      - Utilization of the database server
      - Loss probability of a message upon arrival

   At the beginning of the simulation run, take the following actions:
   c. Print out and examine the Arena *.mod* and *.exp* files.
   d. Add *Variable* windows for
      - Current number of queries in the queue
      - Loss probability (you can use an expression)
      - Average system time of departing messages
   e. Add a *Level* picture for the utilization of the server.
   f. Add a *Resource* picture for the server to monitor the idle and busy states.
   g. Using the *Command* window, issue the appropriate command to stop the simulation at $TNOW = 1250$ seconds. When the simulation stops, issue the appropriate commands to inspect messages in the model queue.
   h. At $TNOW = 3000$ seconds, issue the appropriate trace command to monitor the replication history until $TNOW = 3100$. At that time, use the END command to display the simulation output.

3. *Electronic commerce site*. Consider an electronic commerce site that receives customer requests with iid exponential interarrival times of mean 10 minutes. The site serves the requests in such a way that at the end of every hour, all (new) requests are served in a batch service manner with iid service times uniformly distributed between 10 and 30 minutes. New arrivals during a service time have to wait for the next round of service that will start exactly 1 hour after the previous service started.
   a. Develop an Arena model for the e-commerce site, and simulate it for 5000 minutes. (*Hint*: You may use the same hint given in Exercise 2.)
   b. Estimate the following statistics:
      - Average number of requests waiting to be processed.

- Average response time of customer requests (the response time is defined as the time difference between the request arrival time and its process completion time).

At the beginning of the simulation run, take the following actions:

c. Print out and examine the Arena *.mod* and *.exp* files.

d. Add *Variable* windows for
   - Current number of requests waiting to be processed
   - Average customer response time (*Hint*: You can use the Arena variable *TAVG( )* to tally averages of response times.)

e. Add a *Resource* picture for the server to monitor its idle and busy states.

f. At $TNOW = 3000$ seconds, issue the appropriate trace command to monitor the replication history until $TNOW = 3100$. At that time, use the *END* command to display the simulation output.

Chapter 7

Input Analysis

Input data are key ingredients of simulation modeling. Such data are used to initialize simulation parameters and variables, or construct models of the random components of the system under study. For example, consider a group of milling machines on the shop floor, whose number is to be supplied as a parameter in an input file. You can define a parameter called, say, *No_of_Milling_Machines* in the simulation program, and set it to the group size as supplied by the input file. Other examples are target inventory levels, reorder points, and order quantities. On the other hand, arrival streams, service times, times to failure and repair times, and the like are random in nature (see Chapter 3), and are specified via their distributions or probability laws (e.g., Markovian transition probabilities). Such random components must be first modeled as one or more variates, and their values are generated via RNGs as described in Chapter 4.

The activity of modeling random components is called *input analysis*. From a methodological viewpoint, it is convenient to temporally decompose input analysis into a sequence of stages, each of which involves a particular modeling activity:

Stage 1. Data collection
Stage 2. Data analysis
Stage 3. Time series data modeling
Stage 4. Goodness-of-fit testing

The reader is reminded, however, that as in any modeling enterprise, this sequence does not necessarily unfold in a strict sequential order; in practice, it may involve multiple backtracking and loops of activities.

Data are the grist to the input analysis mill, and its sources can vary widely. If the system to be modeled already exists, then it can provide the requisite empirical data from field measurements. Otherwise, the analyst must rely on more tenuous data, including intuition, past experience with other systems, expert opinion, or an educated guess. In many real-life applications, expedient heuristics are routinely used. Some of these are mentioned in the following list:

- Random variables with negligible variability are simplified and modeled as deterministic quantities.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

- Unknown distributions are postulated to have a particular functional form that incorporates any available partial information. For example, if only the distribution range is known (or guessed at), the uniform distribution over that range is often assumed (every value in the range is equally likely). If, in addition, the mode of the distribution is known (or guessed at), then the corresponding triangular distribution is often assumed.
- Past experience can sometimes provide information on the functional form of distributions. For example, experience shows that a variety of interarrival times (customers, jobs, demand, and time to machine failure, to name a few) can be assumed to be iid exponentially distributed. The analyst is then only required to fit the rate (or mean) parameter of the exponential distribution to complete the requisite input analysis.

In this chapter we discuss the activities comprising the various stages of input analysis. The Arena *Input Analyzer* facilities that support these activities will also be described in some detail. For more information, consult Kelton et al. (2004), Rockwell Software (2005), and the Arena help menus, discussed in Section 6.6.

7.1 DATA COLLECTION

The data collection stage gathers observations of system characteristics over time. While essential to effective modeling, data collection is the first stage to incur modeling risk stemming either from the paucity of available data, or from irrelevant, outdated, or simply erroneous data. It is not difficult to realize that incorrect or insufficient data can easily result in inadequate models, which will almost surely lead to erroneous simulation predictions. At best, model inadequacy will become painfully clear when the model is validated against empirical performance measures of the system under study; at worst, such inadequacy will go undetected. Consequently, the analyst should exercise caution and patience in collecting adequate data, both qualitatively (data should be correct and relevant) and quantitatively (the sample size collected should be representative and large enough).

To illustrate data collection activities, consider modeling a painting workstation where jobs arrive at random, wait in a buffer until the sprayer is available, and having been sprayed, leave the workstation. Suppose that the spray nozzle can get clogged— an event that results in a stoppage during which the nozzle is cleaned or replaced. Suppose further that the metric of interest is the expected job delay in the buffer. The data collection activity in this simple case would consist of the following tasks:

1. *Collection of job interarrival times*. Clock times are recorded on job arrivals and consecutive differences are computed to form the requisite sequence of job interarrival times. If jobs arrive in batches, then the batch sizes per arrival event need to be recorded too. If jobs have sufficiently different arrival characteristics (depending on their type), then the analyst should partition the total arrival stream into substreams of different types, and data collection (of interarrival times and batch sizes) should be carried out separately for each type.
2. *Collection of painting times*. The processing time is the time it takes to spray a job. Since nozzle cleaning or replacement is modeled separately (see later), the painting time should exclude any downtime.
3. *Collection of times between nozzle clogging*. This random process is also known as *time to failure*. Observe that the nozzle clogging process takes place only during painting periods, and is suspended while the system is idle. Thus, the

observations of the effective time to failure should be computed as the time interval between two successive nozzle cloggings minus the total idle time in that interval (if any).

4. *Collection of nozzle cleaning/replacement times*. This random process is also known as *downtime* or *repair time*. Observations should be computed as the time interval from failure (stoppage) onset to the time the cleaning/replacement operation is complete.

It is important to realize that the analyst should only collect data to the extent that they serve project goals. In other words, data should be sufficient for generating the requisite performance statistics, but not more than that. For example, the nozzle-related data collection of items 3 and 4 in the previous list permit the analyst to model uptimes and downtimes separately, and then to generate separate simulation statistics for each. If these statistics, however, were of no interest, then an alternative data collection scheme would be limited to the times spent by jobs in the system from the start of spraying to completion time. These times would of course include nozzle cleaning/replacement times (if any), but such times could not be deduced from the collected data. The alternative data collection scheme would be easier and cheaper, since less data overall would be collected. Such a reduction in data collection should be employed, so long as the collected data meet the project goals.

Data collection of empirical performance measures (expected delays, utilizations, etc.) in the system under study is essential to *model validation*. Recall that validation checks the credibility of a model, by comparing selected performance measures predicted by the model to their empirical counterparts as measured in the field (see Section 1.5). Such empirical measurements should be routinely collected whenever possible with an eye to future validation. Clearly, validation is not possible if the system being modeled does not already exist. In such cases, the validity of a proposed model remains largely speculative.

7.2 DATA ANALYSIS

Once adequate data are collected, the analyst often performs a preliminary analysis of the data to assist in the next stage of model fitting to data (see Section 7.4). The analysis stage often involves the computation of various empirical statistics from the collected data, including

- Statistics related to moments (mean, standard deviation, coefficient of variation, etc.)
- Statistics related to distributions (histograms)
- Statistics related to temporal dependence (autocorrelations within an empirical time series, or cross-correlations among two or more distinct time series)

These statistics provide the analyst with information on the collected sample, and constitute the empirical statistics against which a proposed model will be evaluated for goodness-of-fit.

To illustrate the nature of data analysis, consider the sample of 100 repair time observations in Table 7.1, collected in a manufacturing process (consecutive observations are arranged by rows). Data analysis reveals that the sample minimum and maximum values are 10.3 minutes and 29.9 minutes, respectively. Recall, however, that the corresponding true population statistics may well differ from those of the

Table 7.1
Sample data of repair time observations

12.9	27.7	13.5	13.7	22.2
20.9	26.6	29.1	22.4	10.7
30.0	27.4	18.8	25.3	15.0
17.0	21.7	13.7	15.5	23.2
11.0	27.5	22.5	27.1	25.2
10.3	18.0	11.5	14.1	24.0
10.9	27.0	24.2	25.6	22.4
21.0	21.3	23.1	15.8	13.2
22.8	25.9	22.4	13.8	16.6
10.8	10.3	15.1	19.0	27.9
20.5	19.4	10.9	24.1	10.9
22.2	25.5	17.2	10.9	15.6
14.3	29.9	17.8	19.8	17.6
13.3	24.0	29.7	18.1	28.4
28.6	26.9	20.7	22.0	16.8
19.4	27.4	22.5	28.3	27.1
18.9	11.9	13.2	10.9	22.1
16.7	28.5	19.9	18.5	16.5
12.7	18.1	15.0	21.0	25.7
19.5	11.9	22.9	23.2	18.9

sample, and would usually vary from sample to sample. These statistics (minimum and maximum) play an important role in the choice of a particular (repair time) distribution. Data analysis further discloses that the sample mean is 19.8, the sample standard deviation is 5.76, and the squared sample coefficient of variation is 0.0846. These statistics suggest that repair times have low variability, a fact supported by inspection of Table 7.1.

Arena provides data analysis facilities via its *Input Analyzer* tool, whose main objective is to fit distributions to a given sample. The *Input Analyzer* is accessible from the *Tools* menu in the Arena home screen. After opening a new input dialog box (by selecting the *New* option in the *File* menu in the *Input Analyzer* window), raw input data can be selected from two suboptions in the *Data File* option of the *File* menu:

1. Existing data files can be opened via the *Use Existing* option.
2. New (synthetic) data files can be created using the *Generate New* option as iid samples from a user-prescribed distribution. This option is helpful in studying distributions and in comparing alternative distributions.

Once the subsequent *Input Analyzer* files have been created, they can be accessed in the usual way via the *Open* option in the *File* menu.

Returning to the repair time data of Table 7.1, and having opened the corresponding data file (called *Repair_Times.dft*), the *Input Analyzer* automatically creates a histogram from its sample data, and provides a summary of sample statistics, as shown in Figure 7.1.

The *Options* menu in the *Input Analyzer* menu bar allows the analyst to customize a histogram by specifying its number of intervals through the *Parameters* option and its *Histogram* option's dialog box. Once a distribution is fitted to the data (see next section), the same menu also allows the analyst to change the parameter values of the

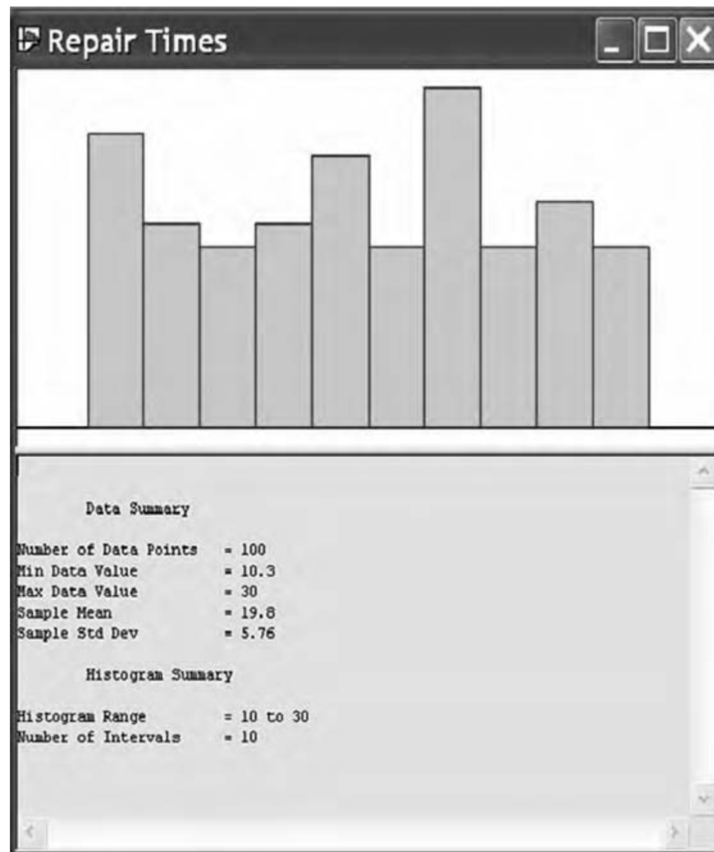


Figure 7.1 Histogram and summary statistics for the repair time data of Table 7.1.

fitted distribution. The *Window* menu grants the analyst access to input data (as a list of numbers) via the *Input Data* option.

7.3 MODELING TIME SERIES DATA

The focal point of input analysis is the data modeling stage. In this stage, a probabilistic model (stochastic process; see Section 3.9) is fitted to empirical time series data (pairs of time and corresponding observations) collected in Stage 1 (Section 7.1). Examples of empirical observations follow:

- An observed sequence of arrival times in a queue. Such arrival processes are often modeled as consisting of iid exponential interarrival times (i.e., a Poisson process; see Section 3.9.2).
- An observed sequence of times to failure and the corresponding repair times. The associated uptimes may be modeled as a Poisson process, and the downtimes as a renewal process (see Section 3.9.3) or as a dependent process (e.g., Markovian process; see Section 3.9.4).

Depending on the type of time series data to be modeled, this stage can be broadly classified into two categories:

1. *Independent observations* are modeled as a sequence of iid random variables (see Section 3.9.1). In this case, the analyst's task is to merely identify (fit) a "good" distribution and its parameters to the empirical data. Arena provides built-in facilities for fitting distributions to empirical data, and this topic will be discussed later on in Section 7.4.
2. *Dependent observations* are modeled as random processes with temporal dependence (see Section 3.9). In this case, the analyst's task is to identify (fit) a "good" probability law to empirical data. This is a far more difficult task than the previous one, and often requires advanced mathematics. Although Arena does not provide facilities for fitting dependent random processes, we will cover this advanced topic in Chapter 10.

We now turn to the subject of fitting a distribution to empirical data, and will focus on two main approaches to this problem:

1. The simplest approach is to construct a histogram from the empirical data (sample), and then normalize it to a step pdf (see Section 3.8.2) or a pmf (see Section 3.7.1), depending on the underlying state space. The obtained pdf or pmf is then declared to be the fitted distribution. The main advantage of this approach is that no assumptions are required on the functional form (shape) of the fitted distribution.
2. The previous approach may reveal (by inspection) that the histogram pdf has a particular functional form (e.g., decreasing, bell shape, etc.). In that case, the analyst may try to obtain a better fit by postulating a particular class of distributions having that shape, and then proceeding to estimate (fit) its parameters from the sample. Two common methods that implement this approach are the *method of moments* and the *maximum likelihood estimation* (*MLE*) *method*, to be described in the sequel. This approach can be further generalized to multiple functional forms by searching for the best fit among a number of postulated classes of distributions. The Arena *Input Analyzer* provides facilities for this generalized fitting approach.

We now proceed to describe in some detail the fitting methods of the approach outlined in 2 above, while the Arena facilities that support the generalized approach will be presented in Section 7.4.

7.3.1 Method of Moments

The *method of moments* fits the moments (see Section 3.5) of a candidate model to sample moments using appropriate empirical statistics as constraints on the candidate model parameters. More specifically, the analyst first decides on the class of distributions to be used in the fitting, and then deduces the requisite parameters from one or more moment equations in which these parameters are the unknowns.

As an example, consider a random variable X and a data sample whose first two moments are estimated as $\hat{m}_1 = 8.5$ and $\hat{m}_2 = 125.3$. Suppose the analyst decides on the class of gamma distributions (see Section 3.8.7). Since the gamma distribution

has two parameters (α and β), two equations are needed to determine their values, given $\hat{m}_1$ and $\hat{m}_2$ above. Using the formulas for the mean and variance of a gamma distribution in Section 3.8.7, we note the following relations connecting the first two moments of a gamma distribution, m_1 and m_2, and its parameters, α and β, namely,

$$m_1 = \alpha\beta$$
$$m_2 = \alpha\beta^2(1 + \alpha)$$

Substituting the estimated values of the two moments $\hat{m}_1$ and $\hat{m}_2$ for m_1 and m_2 above yields the system of equations

$$\hat{\alpha}\hat{\beta} = 8.5$$
$$\hat{\alpha}\hat{\beta}^2(1 + \hat{\alpha}) = 125.3$$

whose unique non-negative solution is

$$\hat{\alpha} = 1.3619$$
$$\hat{\beta} = 6.2412$$

and this solution completes the specification of the fitted distribution. Note that the same solution can be obtained from an equivalent system of equations, formulated in terms of the gamma distribution mean and variance rather than the gamma moments.

7.3.2 Maximal Likelihood Estimation Method

This method postulates a particular class of distributions (e.g., normal, uniform, exponential, etc.), and then estimates its parameters from the sample, such that the resulting parameters give rise to the *maximal likelihood* (highest probability or density) of obtaining the sample. More precisely, let $f(x;\theta)$ be the postulated pdf as function of its ordinary argument, x, as well as the unknown parameter θ. We mention that θ may actually be a set (vector) of parameters, but for simplicity we assume here that it is scalar. Finally, let $(x_1, \ldots, x_N)$ be a sample of independent observations. The maximal likelihood estimation (MLE) method estimates θ via the *likelihood function* $L(x_1, \ldots, x_N;\theta)$, given by

$$L(x_1, \ldots, x_N;\, \theta) = f(x_1;\, \theta)\, f(x_2;\, \theta)\, \cdots\, f(x_N;\, \theta).$$

Thus, $L(x_1, \ldots, x_N;\theta)$ is the *postulated* joint pdf of the sample, and is viewed as a function of both the (known) sample, $(x_1, \ldots, x_N)$, as well as the (unknown) parameter, θ. A maximal likelihood estimator, $\hat{\theta}$, maximizes the function $L(x_1, \ldots, x_N;\theta)$ (or equivalently, the log-likelihood function, $\ln L(x_1, \ldots, x_N;\theta)$) over θ, for a given sample, $(x_1, \ldots, x_N)$.

As an example, consider the exponential distribution with parameter $\theta = \lambda$, and derive its maximum likelihood estimate, $\hat{\theta} = \hat{\lambda}$. The corresponding maximal likelihood function is

$$L(x_1, \ldots, x_N;\lambda) = \lambda e^{-\lambda x_1} \lambda e^{-\lambda x_2} \ldots \lambda e^{-\lambda x_N} = \lambda^N e^{-\lambda \sum_{i=1}^{N} x_i}$$

and the log likelihood function is

$$\ln L(x_1, \ldots, x_N; \lambda) = N \ln \lambda - \lambda \sum_{i=1}^{N} x_i.$$

The value of λ that maximizes the function $\ln L(x_1, \ldots, x_N; \lambda)$ over λ is obtained by differentiating it with respect to λ and setting the derivative to zero, that is,

$$\frac{d}{d\lambda} \ln L(x_1, \ldots, x_N; \lambda) = \frac{N}{\lambda} - \sum_{i=1}^{N} x_i = 0.$$

Solving the above in λ yields the maximal likelihood estimate

$$\hat{\lambda} = \frac{N}{\sum_{i=1}^{N} x_i} = \frac{1}{\bar{x}},$$

which is simply the sample rate (reciprocal of the sample mean).

As another example, a similar computation for the uniform distribution Unif(a,b) yields the MLE estimates $\hat{a} = \min\{x_i : 1 \le i \le N\}$ and $\hat{b} = \max\{x_i : 1 \le i \le N\}$.

7.4 ARENA *INPUT ANALYZER*

The Arena *Input Analyzer* functionality includes fitting a distribution to sample data. The user can specify a particular class of distributions and request the *Input Analyzer* to recommend associated parameters that provide the best fit. Alternatively, the user can request the *Input Analyzer* to recommend both the class of distributions as well as associated parameters that provide the best fit. Table 7.2 displays the distributions supported by Arena and their associated parameters.

Table 7.2
Arena-supported distributions and their parameters

Distribution	Arena name	Arena parameters
Exponential	*EXPO*	*Mean*
Normal	*NORM*	*Mean, StdDev*
Triangular	*TRIA*	*Min, Mode, Max*
Uniform	*UNIF*	*Min, Max*
Erlang	*ERLA*	*ExpoMean, k*
Beta	*BETA*	*Beta, Alpha*
Gamma	*GAMM*	*Beta, Alpha*
Johnson	*JOHN*	*G, D, L, X*
Log-normal	*LOGN*	*LogMean, LogStdDev*
Poisson	*POIS*	*Mean*
Weibull	*WEIB*	*Beta, Alpha*
Continuous	*CONT*	*P1, V1, ...* [a]
Discrete	*DISC*	*P1, V1, ...*

[a] The parameters *P1*, *P2*, ... are cumulative probabilities.

Note, however, that the parameters in Table 7.2 may differ from those in the corresponding distributions of Sections 3.7 and 3.8.

We next illustrate distribution fitting via the *Input Analyzer*. Suppose the analyst would like to fit a uniform distribution to the data of Table 7.1. To this end, the *Repair_Times.dft* file (see Section 7.2) is opened from the *File* menu of the *Input Analyzer*. The *Fit* menu displays all Arena-supported distributions (see Table 7.2), and upon selection of a particular distribution, the *Input Analyzer* computes the associated best-fit parameters and displays the results. In our case, Figure 7.2 depicts the output for the best-fit uniform distribution, including the following items:

- Graph of the best-fit distribution (pdf) superimposed on a histogram of the empirical data
- Parameters of the best-fit distribution
- Statistical outcomes of the goodness-of-fit tests employed (see Section 7.5)
- Summary information on the empirical data and the histogram constructed from it.

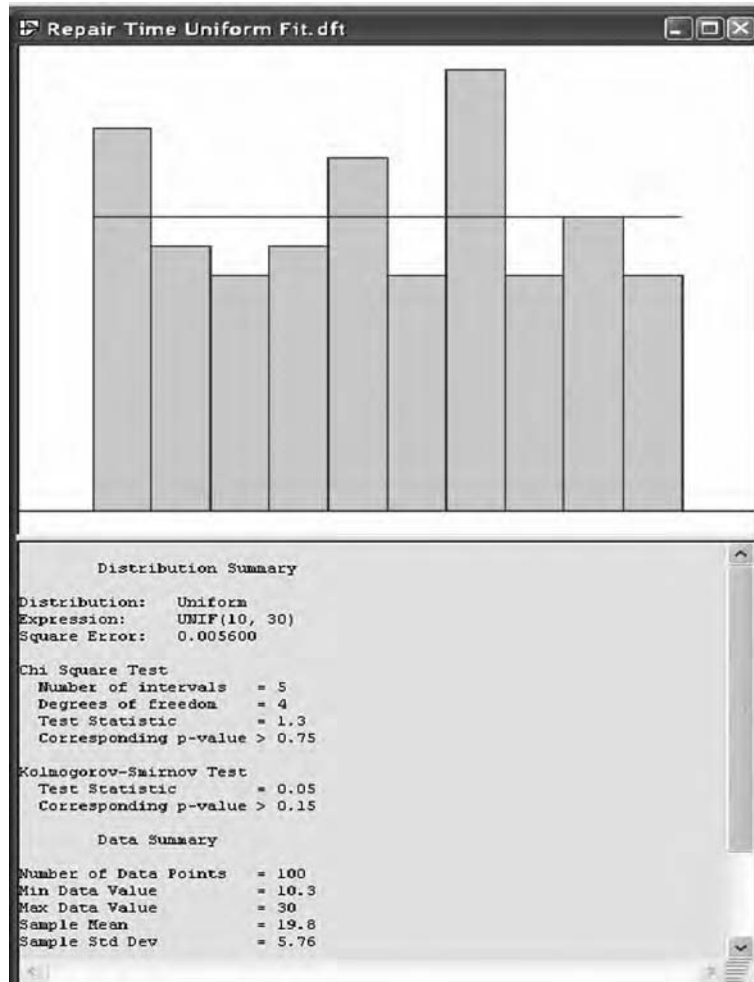


Figure 7.2 Best-fit uniform distribution for the repair time data of Table 7.1.

In our case (best-fit uniform distribution), the associated parameters are just the minimum and maximum of the sample data. The *Curve Fit Summary* option of the *Window* menu provides a detailed summary of any selected fit.

Next, suppose that the analyst changes her mind and would like instead to find the best-fit beta distribution for the same data. Figure 7.3 depicts the resulting *Input Analyzer* output.

Note that the state space of the corresponding random variable, generated via the expression $10 + 20$ (Beta(0.988, 1.02)) is $S = [10, 30]$ rather than the standard

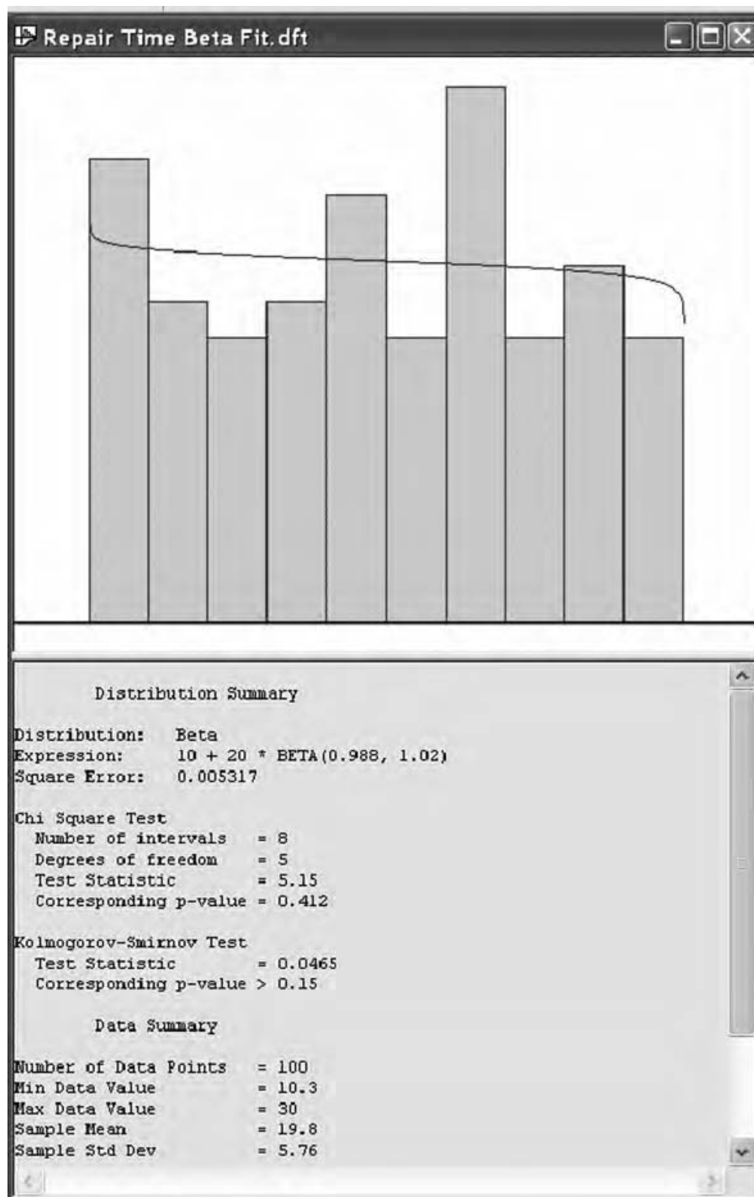


Figure 7.3 Best-fit beta distribution for the repair time data of Table 7.1.

state space $S = [0, 1]$. Thus, the beta distribution was transformed from the interval [0, 1] to the interval [10, 30] by a linear transformation in the expression above.

In a similar vein, Figure 7.4 depicts the results of fitting a gamma distribution to a sample of manufacturing lead-time data (not shown). Note the *Square Error* field appearing in each summary of the distribution fit. It provides an important measure, e^2, of the goodness-of-fit of a distribution to an empirical data set, defined by

$$e^2 = \sum_{j=1}^{J} \left[\hat{p}_j - p_j\right]^2,$$

where J is the number of cells in the empirical histogram, $\hat{p}_j$ is the relative frequency of the j-th cell in the empirical histogram, and p_j is the fitted distribution's (theoretical)

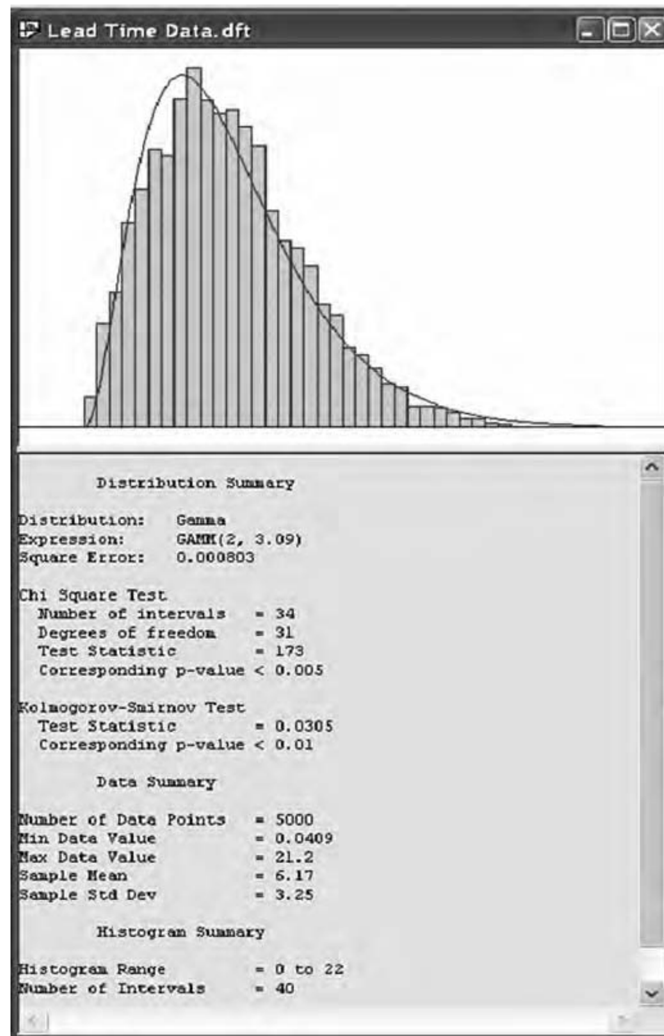


Figure 7.4 Best-fit gamma distribution for a sample of lead time data.

```
Lead Time Data.dft - Fit All Summary
========================================================
Fit All Summary
Data File: C:\Ben\Arena_book\Old\Ch-7_Fig-7-4_Lead Time Data.dft


Function        Sq Error
-----------------------
Normal          0.00162
Triangular      0.00844
Uniform         0.0247
Weibull         0.0497
Poisson         0.056
Lognormal       -1.#J
Erlang          0.0198
Exponential     0.0198
Gamma           0.0799
Beta            0.107
```

Figure 7.5 *Fit All Summary* report for a sample of lead-time data.

probability of the corresponding interval. Obviously, the smaller the value of e^2 is, the better the fit.

Finally, suppose the analyst would like the *Input Analyzer* to determine the best-fit distribution over *all distribution classes* supported by Arena as well as the associated parameters. To this end, the *Fit All* option is selected from the *Fit* menu. In this case, the *Fit All Summary* option may be selected from the *Window* menu to view a list of all classes of fitted distributions and associated square errors, e^2, ordered from best to worst (see Figure 7.5).

7.5 GOODNESS-OF-FIT TESTS FOR DISTRIBUTIONS

The goodness-of-fit of a distribution to a sample is assessed by a statistical test (see Section 3.11), where the null hypothesis states that the candidate distribution is a sufficiently good fit to the data, while the alternate hypothesis states that it is not. Such statistical procedures serve in an advisory capacity: They need not be used as definitive decision rules, but merely provide guidance and suggestive evidence. In many cases, there is no clear-cut best-fit distribution. Nevertheless, goodness-of-fit tests are useful in providing a quantitative measure of goodness-of-fit (see Banks et al. [1999] and Law and Kelton [2000]).

The *chi-square test* and the *Kolmogorov–Smirnov test* are the most widely used tests for goodness-of-fit of a distribution to sample data. These tests are used by the Arena *Input Analyzer* to compute the corresponding test statistic and the associated *p*-value (see Section 3.11), and will be reviewed next.

7.5.1 CHI-SQUARE TEST

The *chi-square* test compares the empirical histogram density constructed from sample data to a candidate theoretical density. Formally, assume that the empirical

sample $\{x_1, \ldots, x_N\}$ is a set of N iid realizations from an underlying random variable, X. This sample is then used to construct an empirical histogram with J cells, where cell j corresponds to the interval $[l_j, r_j)$. Thus, if N_j is the number of observations in cell j (for statistical reliability, it is commonly suggested that $N_j > 5$), then

$$\hat{p}_j = \frac{N_j}{N}, \quad j = 1, \ldots, J$$

is the relative frequency of observations in cell j. Letting $F_X(x)$ be some theoretical candidate distribution whose goodness-of-fit is to be assessed, one computes the theoretical probabilities

$$p_j = \Pr\{l_j \leq X < r_j\}, \quad j = 1, \ldots, J.$$

Note that for continuous data, we have

$$p_j = F_X(r_j) - F_X(l_j) = \int_{l_j}^{r_j} f_X(x)\, dx,$$

where $f_X(x)$ is the pdf of X, while for discrete data, we have

$$p_j = F_X(r_j) - F_X(l_j) = \sum_{x=l_j}^{r_j} p_X(x),$$

where $p_X(x)$ is the pmf of X. The chi-square test statistic is then given by

$$\chi^2 = \sum_{j=1}^{J} \frac{(N_j - Np_j)^2}{Np_j}. \tag{7.1}$$

Note that the quantity Np_j is the (theoretical) expected number of observations in cell j predicted by the candidate distribution $F_X(x)$, while N_j is the actual (empirical) number of observations in that cell. Consequently, the j-th term on the right side of (7.1) measures a relative deviation of the empirical number of observations in cell j from the theoretical number of observations in cell j. Intuitively, the smaller is the value of the chi-square statistic, the better the fit. Formally, the chi-square statistic is compared against a critical value c (see Section 3.11), depending on the significance level, α, of the test. If $\chi^2 < c$, then we accept the null hypothesis (the distribution is an acceptably good fit); otherwise, we reject it (the distribution is an unacceptably poor fit). The chi-square critical values are readily available from tables in statistics books. These values are organized by degrees of freedom, d, and significance level, α. The degrees of freedom parameter is given by $d = J - E - 1$, where E is the distribution-dependent number of parameters estimated from the sample data. For instance, the gamma distribution Gamm(α, β) requires $E = 2$ parameters to be estimated from the sample, while the exponential distribution Expo(λ) only requires $E = 1$ parameter. An examination of chi-square tables reveals that for a given number of degrees of freedom, the critical value c increases as the significance level, α, decreases. Thus, one can trade off test significance (equivalently, confidence) for test stringency.

As an example, consider the sample data of size $N = 100$, given in Table 7.1, for which a histogram with $J = 10$ cells was constructed by the *Input Analyzer*, as shown in

Table 7.3
Empirical and theoretical statistics for the empirical histogram

Cell number j	Cell Interval $[l_j, r_j)$	Number of Observations N_j	Relative Frequency $\hat{p}_j$	Theoretical Probability p_j
1	[10,12)	13	0.13	0.10
2	[12,14)	9	0.09	0.10
3	[14,16)	8	0.08	0.10
4	[16,18)	9	0.09	0.10
5	[18,20)	12	0.12	0.10
6	[20,22)	8	0.08	0.10
7	[22,24)	13	0.13	0.10
8	[24,26)	10	0.10	0.10
9	[26,28)	10	0.10	0.10
10	[28,30)	8	0.08	0.10

Figure 7.1. The best-fit uniform distribution, found by the *Input Analyzer*, is depicted in Figure 7.2.

Table 7.3 displays the associated elements of the chi-square test. The table consists of the histogram's cell intervals, number of observations in each cell, and the corresponding empirical relative frequency for each cell. An examination of Table 7.3 reveals that the histogram ranges from a minimal value of 10 to a maximal value of 30, with the individual cell intervals being (10, 12), (12, 14), (14, 16), and so on. Note that the fitted uniform distribution has $p_j = 0.10$ for each cell j. Thus, the χ^2 test statistic is calculated as

$$\chi^2 = \frac{(13-10)^2}{10} + \cdots + \frac{(8-10)^2}{10} = 3.6.$$

A chi-square table shows that for significance level $\alpha = 0.10$ and $d = 10 - 2 - 1 = 7$ degrees of freedom, the critical value is $c = 12.0$; recall that the uniform distribution Unif (a, b) has two parameters, estimated from the sample by $\hat{a} = \min\{x_i : 1 \le i \le N\} = 10$ and $\hat{b} = \max\{x_i : 1 \le i \le N\} = 30$, respectively. (When the uniform distribution parameters a and b are known, then $d = 10 - 1 = 9$. In fact, the Arena *Input Analyzer* computes d in this manner.) Since the test statistic computed above is $\chi^2 = 3.6 < 12.0$, we accept the null hypothesis that the uniform distribution Unif(10, 30) is an acceptably good fit to the sample data of Table 7.1.

It is instructive to follow the best-fit actions taken by the *Input Analyzer*. First, it calculates the square error

$$e^2 = \sum_{j=1}^{10} \left[\hat{p}_j - p_j\right]^2 = (0.13 - 0.10)^2 + \cdots + (0.08 - 0.10)^2 = 0.0036.$$

Next, although the histogram was declared to have 10 cells, the *Input Analyzer* employed a χ^2 test statistic with only 6 cells (to increase the number of observations in selected cells). It then proceeded to calculate it as $\chi^2 = 2.1$, with a corresponding p-value of $p > 0.75$, clearly indicating that the null hypothesis cannot be rejected even at significance levels as large as 0.75. Thus, we are assured to accept the null hypothesis of a good uniform fit at a comfortably high confidence.

7.5.2 Kolmogorov-Smirnov (K-S) Test

While the chi-square test compares the empirical (observed) histogram pdf or pmf to a candidate (theoretical) counterpart, the *Kolmogorov-Smirnov* (*K-S*) test compares the empirical cdf to a theoretical counterpart. Consequently, the chi-square test requires a considerable amount of data (to set up a reasonably "smooth" histogram), while the K-S test can get away with smaller samples, since it does not require a histogram.

The K-S test procedure first sorts the sample $\{x_1, x_2, \ldots, x_N\}$ in ascending order $x_{(1)}, x_{(2)}, \ldots, x_{(N)}$, and then constructs the empirical cdf, $\hat{F}_X(x)$, given by

$$\hat{F}_X(x) = \frac{\max\{j : x_{(j)} \leq x\}}{N}.$$

Thus, $\hat{F}_X(x)$ is just the relative frequency of sample observations not exceeding x. Since a theoretical fit distribution $F_X(x)$ is specified, a reasonable measure of goodness-of-fit is the largest absolute discrepancy between $\hat{F}_X(x)$ and $F_X(x)$. The K-S test statistic is thus defined by

$$KS = \max_x\{|\hat{F}_X(x) - F_X(x)|\}. \tag{7.2}$$

The smaller is the observed value of the K-S statistic, the better the fit.

Critical values for K-S tests depend on the candidate theoretical distribution. Tables of critical values computed for various distributions are scattered in the literature, but are omitted from this book. The Arena *Input Analyzer* has built-in tables of critical values for both the chi-square and K-S tests for all the distributions supported by it (see Table 7.2). Refer to Figures 7.2 through 7.4 for a variety of examples of calculated test statistics.

7.6 MULTIMODAL DISTRIBUTIONS

Recall that the *mode* of a distribution is that value of its associated pdf or pmf at which the respective function attains a maximal value (a distribution can have more than one mode). A *unimodal* distribution has exactly one mode (distributions can be properly defined to have at least one mode). In contrast, in a *multimodal* distribution the associated pdf or pmf is of the following form:

- It has more than one mode.
- It has only one mode, but it is either *not monotone increasing* to the left of its mode, or *not monotone decreasing* to the right of its mode.

Put more simply, a multimodal distribution has a pdf or pmf with multiple "humps." One approach to input analysis of multimodal samples is to separate the sample into mutually exclusive unimodal subsamples and fit a separate distribution to each subsample. The fitted models are then combined into a final model according to the relative frequency of each subsample.

As an example, consider a sample of N observations of which N_1 observations appear to form a unimodal distribution in interval I_1, and N_2 observations appear to form a unimodal distribution in interval I_2, where $N_1 + N_2 = N$. Suppose that the theoretical distributions $F_1(x)$ and $F_2(x)$ are fitted separately to the respective subsamples. The combined distribution to be fitted to the entire sample is defined by

$$F_X(x) = \frac{N_1}{N} F_1(x) + \frac{N_2}{N} F_2(x). \tag{7.3}$$

The distribution $F_X(x)$ in Eq. 7.3 is a legitimate distribution, which was formed as a probabilistic mixture of the two distributions, $F_1(x)$ and $F_2(x)$.

EXERCISES

1. Consider the following downtimes (in minutes) in the painting department of a manufacturing plant.

37.2	10.9	3.9	1.6	17.3
69.8	2.7	30.8	24.3	61.0
73.7	14.6	24.7	15.4	26.1
14.8	29.1	6.9	7.4	71.2
10.1	45.4	54.7	9.7	99.5
99.5	16.0	4.5	20.8	30.2
30.2	41.3	21.2	31.3	2.6
61.0	2.6	70.4	7.5	20.1
26.1	20.1	6.9	32.6	7.4
71.2	7.7	39.0	43.3	9.7

   a. Use the chi-square test to see if the exponential distribution is a good fit to this sample data at the significance level $\alpha = 0.05$.
   b. Using Arena's *Input Analyzer*, find the best fit to the data. For what range of significance levels α can the fit be accepted? (*Hint*: revisit the concept of p-value in Section 3.11.)
2. Consider the following data for the monthly number of stoppages (due to failures or any other reason) in the assembly line of an automotive assembly plant.

12	14	26	13	19
15	18	17	14	10
11	11	14	18	13
9	20	19	9	12
17	20	25	18	12
16	20	20	14	11

   Apply the chi-square test to the sample data of stoppages to test the hypothesis that the underlying distribution is Poisson at significance level $\alpha = 0.05$.
3. Let $\{x_i\}$ be a data set of observations (sample).
   a. Show that for the distribution Unif(a, b), the maximal likelihood estimates of the parameters a and b are $\hat{a} = \min\{x_i\}$ and $\hat{b} = \max\{x_i\}$.
   b. What would be the estimates of a and b obtained using the method of moments?
   c. Demonstrate the difference between the two sets of estimates for a and b using a data set of 500 observations from a uniform distribution generated by the Arena *Input Analyzer*.

4. The Revised New Brunswick Savings bank has three tellers serving customers in their Highland Park branch. Customers queue up FIFO in a single combined queue for all three tellers. The following data represent observed iid interarrival times (in minutes).

2.3	4.7	0.2	2.1	1.6
1.8	4.9	1.0	1.6	1.0
2.7	1.0	1.9	0.5	0.5
0.1	0.7	3.0	3.6	0.6
5.5	6.6	1.1	0.3	1.4
3.0	2.0	2.8	1.4	2.1
1.8	4.1	0.2	4.7	0.5
0.1	1.7	1.3	0.5	2.2
4.7	4.7	0.5	2.6	2.9
2.5	0.7	0.3	0.1	1.2

The service times are iid exponentially distributed with a mean of 5 minutes.

a. Fit an exponential distribution to the interarrival time data in the previous table. Then, construct an Arena model for the teller queue at the bank. Run the model for an 8-hour day and estimate the expected customer delay in the teller's queue.
b. Fit a uniform distribution to the interarrival time data above. Repeat the simulation run with the new interarrival time distribution.
c. Discuss the differences between the results of the scenarios of 4.a and 4.b. What do you think is the reason for the difference between the corresponding mean delay estimates?
d. Rerun the scenario of 4.a with uniformly distributed service times, and compare to the results of 4.a and 4.b. Use MLE to obtain the parameters of the service time distribution.
e. Try to deduce general rules from the results of 4.a, 4.b, and 4.d on how variability in the arrival process or the service process affects queue performance measures.

This page intentionally left blank

Chapter 8

Model Goodness: Verification and Validation

A model of any kind (physical, analytical, simulation, etc.) is a simplified representation of some system under study, and as such, a model would rarely capture precisely system characteristics. Consequently, a model cannot always be expected to predict accurately system performance. Therefore, the cardinal goal of modeling is merely to construct an acceptably "correct" model. The notion of model correctness (often referred to as model *goodness*) is necessarily relative rather than absolute, and its precise meaning will vary from case to case. Suffice it to say that model goodness should be interpreted as an operational notion that requires the model to produce "sufficiently accurate" performance predictions for the system under study. Again, the notion of sufficient accuracy of model predictions depends on the individual cases under consideration, and the task of defining accuracy criteria is often left to the analyst. In many cases, model accuracy cannot be exactly gauged (e.g., when the system under study does not exist), so that any assessment of model goodness becomes merely an educated guess. Since the determination of model goodness is rarely a crisp or algorithmic procedure, the enterprises of modeling and goodness assessment are necessarily "best effort" activities.

This chapter focuses on goodness assessment, and specifically on its two components: *verification* and *validation*. Verification assesses the *correctness* of the formal representation of the *intended* model (in our case, a computer simulation program), by inspecting computer code and test runs, and performing consistency checks on their statistics. Validation assesses how *realistic* the modeling assumptions are, by comparing model performance metrics (predictions), obtained from model test runs, to their counterparts in the system under study (obviously, validation is possible only if the latter exists). Note that a model might exhibit satisfactory goodness even though its underlying assumptions are quite unrealistic. Conversely, an ostensibly realistic model may reveal its shortcomings by exhibiting poor predictive power. It is worth noting that a complete and sure verification and validation of models is rarely possible. Consequently, verification and validation activities must be alert to circumstantial evidence of coding errors and faulty logic. Note carefully that *absence of evidence is*

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

not evidence of absence. Thus, verification and validation are best-effort activities that conclude when the analyst is reasonably sure of model goodness; however, the true state of "correctness" is rarely known conclusively.

As described in Chapter 1, the activities in a simulation project call for constructing a model *after* a thorough analysis of the system at hand. Once the conceptual model is encoded as a computer program, a debugger is often used to verify the program (recall that Arena provides an interactive command-line debugger as described in Chapter 6). The model is further verified through analysis of its predictions. More specifically, verification consists in the main of the following activities:

1. Inspecting simulation program logic (code "walk-through").
2. Performing simulation test runs and inspecting sample path trajectories. In particular, in a visual simulation environment (like Arena), the analyst inspects both code printouts as well as graphics to verify (as best one can) that the underlying program logic is correct.
3. Performing simple consistency checks, including sanity checks (quick "back-of-the-envelope" plausibility checks), as well as more sophisticated checks of theoretical relationships among predicted statistics. Queueing-theoretic relations are a common example.

Once the analyst is satisfied with the verification stage, validation activities can get under way. In practice, verification and validation need not be strictly sequential. Since both are based in part on simulation test runs, some of their associated activities can be conducted concurrently. However, certain verification activities (e.g., code inspection) should be conducted before any further validation.

The rest of this chapter describes a set of procedures for verification and validation of simulation models, mainly via examples.

8.1 MODEL VERIFICATION VIA INSPECTION OF TEST RUNS

This section provides a set of guidelines for verifying that a simulation model correctly implements its design specifications. Although some of these guidelines are useful in verifying general computer code, special emphasis will be placed on Arena facilities that support model verification.

8.1.1 Input Parameters and Output Statistics

Be sure to formulate a set of input parameters and output statistics, such that the former permit ready experimentation with key parameter values, and the latter capture all essential system performance metrics. The Arena standard output includes entity counts (the running current values or final values). Such statistics are extremely helpful in checking that the model has no "dead-ends" (model components that "trap" transactions by erroneously providing no transfer mechanism out of them). Such dead-ends can be discovered by inspecting statistics to verify that a proper flow of entities is maintained through selected model components.

8.1.2 Using a Debugger

An interactive run controller or debugger is an indispensable feature of any simulation tool (the Arena run controller is reviewed in Chapter 5). It allows the modeler to trace a set of variables as their values evolve over time or track entities as they proceed through Arena modules in the course of model runs. The Arena run controller is capable of providing the entire run history (sample path realization) in a search for faulty model logic. It also has some convenient run-time features, such as stopping model runs when prescribed conditions are satisfied.

8.1.3 Using Animation

To the extent possible, animation should be liberally used to verify model logic. Making test runs to observe the model's animated evolution (e.g., entity flows), coupled with inspection of running statistics and variable trace values, is a particularly effective verification method.

8.1.4 Sanity Checks

Some basic statistics produced by the model under study should be analyzed to see if they are reasonable or plausible. For example, if the arrival rate at a system is known, one may wish to check whether the *observed* number of arrivals (or departures) is in line with their expected values. If the observed values look suspiciously out of line, then a more concerted debugging effort can be launched.

8.2 MODEL VERIFICATION VIA PERFORMANCE ANALYSIS

At some point in the modeling enterprise, the modeler should conduct more sophisticated consistency analysis beyond sanity checks. The vast majority of Arena-based simulation models consist of a queueing component, and such models call for a *performance analysis* study of a queueing system, which involves the computation of appropriate performance measures and verification of certain relations among them. Due to the pervasiveness of queueing models in simulation modeling, the analyst needs to acquire a modicum of knowledge of queueing theory. This section overviews some elements of performance analysis of queueing systems and associated queueing theory that support model verification.

8.2.1 Generic Workstation as a Queueing System

The queueing-modeling paradigm plays a key role in the analysis and design of manufacturing systems, telecommunications systems, transportation, and other areas. Queueing theory is a well-developed field that has produced an enormous body of research on queueing systems and queueing networks. The reader is referred to Gross

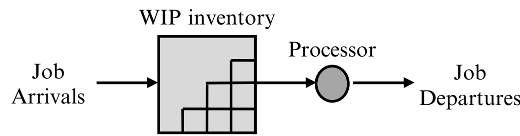


Figure 8.1 A simple workstation.

and Harris (1974), Cooper (1990), and Kleinrock (1975), to name a few. Here, we review briefly the formulation and analysis of queueing systems commonly used to model manufacturing and service systems.

Consider a generic queueing example from the domain of manufacturing. Recall that a *workstation* on the factory floor is a location where work is carried out by a server (machines or humans), possibly subject to random failures. Figure 8.1 depicts a schematic workstation, where jobs arrive to be processed or machined. Excess jobs are kept in a buffer (*work-in-process*, abbreviated as *WIP*) until their processing can begin. Upon completion, jobs depart for another destination (possibly another workstation).

When studying workstations in manufacturing environments, we routinely model them as queueing systems. The flow of incoming jobs, either singly or in batches, forms the *arrival stream*. The time that a job is delayed for processing at the workstation server is its *service time*. If, however, an incoming job cannot start processing right away (because the server or servers are busy), then it is held in a *buffer*, and in due time will be removed from the buffer and assigned a server. In a workstation model, the buffer capacity may be finite or infinite, although in real-life workstations it is, of course, finite. A job that finds the buffer full on arrival is usually *rerouted* to another workstation or simply assumed to be *lost*. The order in which jobs queue up in the buffer is called a *queueing discipline*. A workstation model may have additional wrinkles, such as server failure and repairs, routinely modeled via random uptimes and downtimes.

8.2.2 Queueing Processes and Parameters

A waiting line necessarily develops in every service-providing facility, whenever it cannot momentarily cope with its current workload (brought in by customers, jobs, demands, or more generally, transactions, all of which will be used interchangeably). A queueing system that models a service facility is mathematically characterized by its customer arrival process, service process, number of servers, service discipline, and queue (buffer) capacity. The arrival process is routinely defined in terms of the probability law governing the time intervals between consecutive arrivals (called *interarrival times*). Accordingly, let A_i be the interarrival time between job $i - 1$ and job i. It is often assumed that the A_i are iid (independent, identically distributed) random variables. The arrival rate, denoted by λ, is the expected number of job arrivals per unit time, and is given by $\lambda = 1/\mathrm{E}[A_i]$ for all i, that is, the arrival rate is the reciprocal of the expected interarrival time. The service time is the time that the server devotes to a particular job. Let X_i be the service time of job i. Again, it is often assumed that the X_i are iid random variables with service rate $\mu = 1/\mathrm{E}[X_i]$ for all i, that is, the service rate is the reciprocal of the expected service time.

8.2.3 Service Disciplines

In a multiserver facility, the servers are also often assumed to be statistically independent and identical, with a common service time distribution (iid servers). The customer waiting room (buffer) usually has finite capacity; however, a very large buffer may be considered infinite for modeling purposes. When more than one server is available for service, a server is usually selected at random with equal probabilities. Recall that the order in which jobs are moved from the buffer to seize a server and start service is the service discipline or queueing discipline, the most common of which are listed below:

- *FIFO* (*first in, first out*), also known as *FCFS* (*first come, first served*), is the most common discipline. Here, jobs are served in their order of arrival.
- *LIFO* (*last in, first out*), also known as *LCFS* (*last come, first served*), serves the most recent arrival first.
- *SIRO* (*service in random order*) selects the next job in the buffer randomly, each with equal probability.
- *RR* (*round robin*) is associated with a (fixed) service time, often referred to as the *time quantum*. Jobs are served cyclically, one quantum at a time, until attaining their requisite service time.
- *PS* (*priority service*) assumes that jobs have priorities associated with them, and selects a job with the highest service priority. If several are present, then a secondary discipline may be used to select among highest-priority jobs, such as FIFO, LIFO, SIRO, and so on. Jobs with the same priority are said to belong to the corresponding *priority class*.

A standard notation has been developed to specify a queueing system succinctly. It employs slash-separated symbols, representing an arrival process, a service process, the number of servers, and the queue capacity (understood to be infinity, when omitted), in this order. The symbol M (shorthand for *Markov*, see Section 3.9.4) stands for an exponential distribution, D for a deterministic value, and GI for a general distribution, all in the corresponding iid arrival or service process. Thus, $M/M/1$ specifies a queue with iid exponential distributions for interarrival and service times, a single server, and infinite buffer capacity; similarly, $M/D/k/C$ specifies a queue with iid exponential interarrival times, deterministic service times, k servers, and a capacity C, of which the first k positions are occupied by servers.

8.2.4 Queueing Performance Measures

A queueing system is studied to glean understanding of its behavior and to estimate its performance measures (metrics) of interest. Common performance measures follow:

- Average number of jobs in the queue (buffer only)
- Average number of jobs in the system (buffer and servers)
- Average job waiting time (buffer-only delay)
- Average job sojourn time (buffer and service delays)
- Server utilization (fraction of time a server is busy)
- Throughput (output rate, namely, departure rate from the system)

One general way of evaluating these measures is to first compute analytically (if possible) or to estimate (by simulation) the steady-state probability distribution of the

number of jobs in the system. Thus, if the system has capacity K and P_n denotes the steady-state probability of having n jobs in the system, then the average number of jobs in the system, $\bar{N}_S$, is given by

$$\bar{N}_S = \sum_{n=0}^{K} n\, P_n \tag{8.1}$$

where K may be finite or infinite. In a single-server system, the average number of jobs in service is $1 - P_0$ (the probability that the system is not empty of customers), and therefore, the average number of jobs waiting in the queue, $\bar{N}_q$, becomes

$$\bar{N}_q = \bar{N}_S - (1 - P_0). \tag{8.2}$$

In general, $1 - P_0 = \rho$ is the server utilization for any single-server queueing system with finite or infinite queue capacity. For infinite capacity systems, utilization can be expressed as the ratio of the input rate and the service rate, that is, $\rho = \lambda/\mu$. When $\rho \geq 1$, then the server is said to be *exhausted*. In this case, the server is not able to keep up with the incoming workload, and consequently the number of jobs in the system grows without bound in the long run. This situation is colorfully referred to as "system explosion," but in queueing theory terminology, the system is said to be *unstable*. For stability to hold and long-run measures to exist, we must have $\rho < 1$, or equivalently, $P_0 > 0$. Clearly, a finite-capacity system is always stable, because the number of jobs in it is bounded by its capacity. Note that we are typically interested in stable systems in the context of simulation modeling.

8.2.5 Regenerative Queueing Systems and Busy Cycles

The fundamental relation $\rho = \lambda/\mu$ can be justified more rigorously using the following so-called *regeneration argument* for the *GI/GI/*1 queue, that is, for iid interarrival times and iid service times. Any such stable queueing system goes through an endless sequence of cycles, called *busy cycles*. Figure 8.2 illustrates busy cycles in a queue.

Each busy cycle consists of an *idle period* (during which the queue is empty and the server is idle), followed by a *busy period* (during which the queue is not empty and the server is busy serving jobs). When the system starts an idle period, the next event will definitely be a job arrival, and the (random) time until this event occurs is independent of the history of the workstation up until that point. Thus, in the *GI/GI/*1 queue, the system history *regenerates* itself (statistically) at time points inaugurating idle periods, in the sense that system histories over distinct regeneration cycles

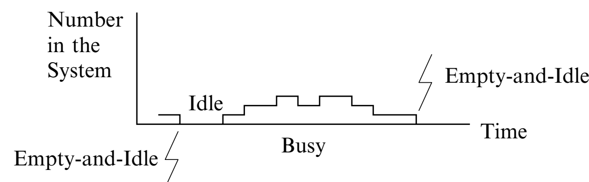


Figure 8.2 Regeneration cycles consisting of successive idle and busy periods.

(intervals consisting of successive idle and busy periods) are iid. The corresponding stochastic processes are therefore called *regenerative* processes or *renewal* processes (see Section 3.9.3). For example, the stochastic process of the number of jobs in the system is regenerative (refer again to Figure 8.2).

Let T_B be the length of the busy period, and let T_C be the length of the regeneration cycle. In a system with infinite capacity, all jobs arriving during a regeneration cycle are served during the associated busy period. Thus, the number of incoming and outgoing jobs over a regeneration cycle (as well as corresponding averages) must coincide, that is,

$$\lambda\, \mathrm{E}[T_C] = \mu \mathrm{E}[T_B]$$

resulting in the relation

$$\frac{\lambda}{\mu} = \frac{\mathrm{E}[T_B]}{\mathrm{E}[T_C]} = \rho. \tag{8.3}$$

The right side in Eq. 8.3 is precisely the fraction of time the server is busy over a regeneration cycle, that is, $\rho = \lambda/\mu = \Pr\{\text{server is busy}\}$.

The length of the busy period and the regeneration cycle are also important measures of interest. When interarrival times are Expo(λ), the idle time in the regeneration cycle is also Expo(λ), because it is the residual interarrival time (from the time instant the server goes idle). Given the memoryless property of the exponential distribution (see Section 3.8.4), the following expression can be written:

$$\Pr\{\text{server is idle}\} = 1 - \rho = \frac{1/\lambda}{\mathrm{E}[T_C]}, \tag{8.4}$$

which readily yields

$$\mathrm{E}[T_C] = \frac{1/\lambda}{1 - \rho}. \tag{8.5}$$

From Eqs. 8.3 and 8.5, we obtain

$$\mathrm{E}[T_B] = \frac{\mathrm{E}[X]}{1 - \rho} \tag{8.6}$$

where X is the service time with $\mathrm{E}[X] = 1/\mu$. The quantity $\mathrm{E}[T_B]$ merits further elaboration. Each job inaugurating a busy period arrives at an empty system. However, new jobs (the "offspring" of the first job) may arrive during its service time, and each of those may have its own "offspring." Thus, $\mathrm{E}[T_B]$ can be viewed as the expected time to serve the "family" of jobs "spawned" by the first job in the busy cycle. A similar analysis of servers subject to various failures can be found in Altiok (1997).

8.2.6 Throughput

The throughput (output or departure rate) of a queueing system can be expressed in terms of utilization. The server works at the rate of μ jobs per unit time (while busy), yielding an average throughput of

$$\bar{o} = \mu(1 - P_0). \quad (8.7)$$

However, for queues with infinite capacity, Eq. 8.7 reduces to λ due to long-run job flow conservation in stable queueing systems (see also Eq. 8.3).

8.2.7 Little's Formula

Consider any steady-state storage system, where units arrive, spend some time in the system, and eventually depart. A general steady-state relation of great practical importance is *Little's formula*,

$$\bar{N} = \lambda\bar{W} \quad (8.8)$$

where $\bar{N}$ is the average number of jobs in the system and $\bar{W}$ is the average time a job spends in the system (mean flow time). In particular, Little's formula holds for any queueing subsystem (one should, however, draw the boundaries of the subsystem under consideration with some care). Thus, for the entire queueing system, $\bar{N}_S = \lambda\bar{W}_S$; similarly, for the buffer alone, $\bar{N}_b = \lambda\bar{W}_b$. Interestingly, Little's formula is valid for any queue capacity. Thus, for a queue with finite capacity (and therefore with a possible loss stream), the arrival rate, λ, in Eq. 8.8 is simply the *effective* arrival rate (the expected rate of arrivals that manage to enter the system, excluding any lost jobs), and not the *offered* arrival rate (which includes lost jobs).

In effect, Little's formula expresses a conservation law, although this is not immediately evident from Eq. 8.8. While a formal proof is fairly intricate, a heuristic plausibility argument (more accurately, an interpretation of Eq. 8.8) can be advanced as follows. Consider some target customers traversing the system. Then, the right-hand side of Eq. 8.8 is the expected number of customers that arrive while the target customer sojourns in the system, while the left-hand side of Eq. 8.8 is the expected number of customers that accumulate in the system. Equation 8.8 asserts that these expectations are equal.

Little's formula is an important tool for verifying queueing simulations due to its simplicity and generality. It may be used to discover modeling errors or coding bugs, where customers are erroneously "marooned" in the system, or are inadvertently created or destroyed.

8.2.8 Steady-State Flow Conservation

A widely used conservation relation for single-server queueing systems with finite capacity (buffer size plus service positions), K, is

$$\lambda(1 - \pi_K) = \mu(1 - P_0) \quad (8.9)$$

where π_K is the steady-state probability that an arriving job finds the buffer full on arrival, and P_0 is the steady-state probability that the system is empty (see Gross and Harris [1998]). Thus, Eq. 8.9 is a flow conservation equation stating that in steady state, the effective arrival rate into the system equals the departure rate from the system (throughput). Note carefully that probabilities of the form π_K (steady-state probability of a job finding k jobs already in the system on arrival) and P_k (steady-state probability of k jobs in the system) are distinct quantities with the following operational meaning:

- π_K is the long-run *fraction of jobs* that find (on arrival) k jobs in the system. Such quantities are referred to as *customer-average probabilities* or *arrival point probabilities*. Note carefully that the number of jobs in the system is computed only at the (random) times of arrival and excludes the arriving job. For this reason, this type of system state is referred to as the state *embedded* at arrival times, or as the state *seen* by arriving jobs.
- P_k is the long-run *fraction of time* that the system has precisely k jobs in it. Such quantities are referred to as *time average probabilities* or *arbitrary time probabilities*.

8.2.9 PASTA Property

Generally, customer averages and time averages are *not* equal (embedding the state at random times can introduce a "bias" into a customer average as compared to the corresponding time average). While generally, $\pi_k \neq P_k$, there are cases when they are equal (Melamed and Whitt [1990a,b], Melamed and Yao [1995]). The most common case where the equality

$$\pi_k = P_k$$

holds is known as *PASTA* (*Poisson Arrivals See Time Averages*)—in other words, when the arrival stream is a Poisson process. Intuitively, this can be attributed to the memoryless property of the iid exponentially distributed interarrival times (see Section 3.8.4), or equivalently, to the independent increments of the Poisson process (see Section 3.9.2).

8.3 EXAMPLES OF MODEL VERIFICATION

In this section we employ the theory reviewed in Section 8.2 to demonstrate practical verification methods. We present verification examples in two systems: a single workstation and a system consisting of two workstations in tandem.

8.3.1 Model Verification in a Single Workstation

Consider a workstation (represented as a single-server queue), whose Arena model is exhibited in Figure 8.3.

The workstation houses a single machine (server). Job processing times are exponentially distributed with a mean of 4 hours (i.e., with processing rate $\mu = 0.25$), while job interarrival times are uniformly distributed between 1 and 8 hours. The workstation buffer has a finite capacity of four jobs (excluding any job being processed), so that jobs arriving at a full buffer are lost.

Jobs (entities) are generated in the *Create* module, called *Create Arrivals*, and their arrival times are stored in the *ArrTime* attribute of each incoming entity in the *Assign* module called *Assign Arrival Time*. An incoming job entity then enters the *Decide* module, called *Is Buffer Full?*, to check if there is space available in the buffer. If no space is available, the job entity proceeds to the *Record* module, called *Count Rejected*, to increment the count of rejected units, and is then disposed of. Otherwise, the job entity enters the *Seize* module, called *Seize Server*, where it attempts to seize the

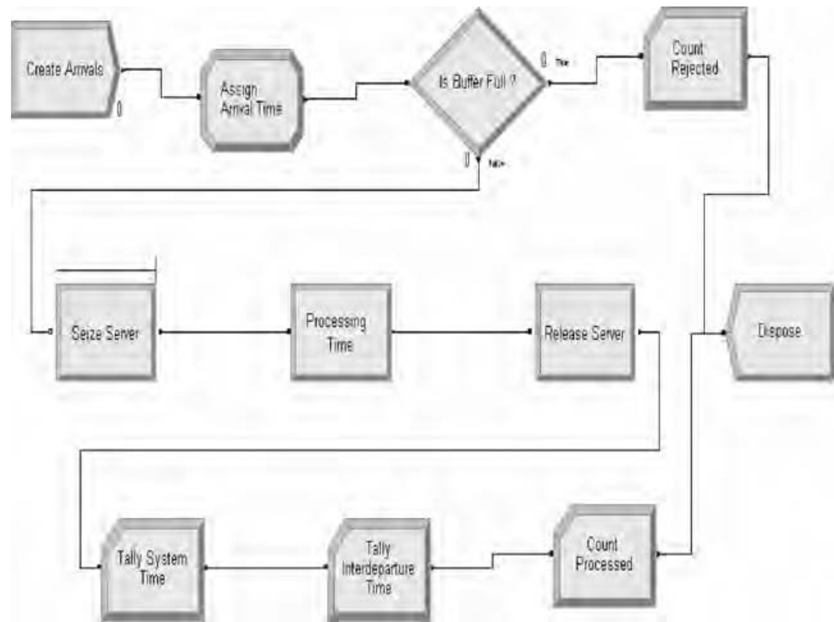


Figure 8.3 Arena model of a single workstation model.

resource *Server*, possibly waiting for its turn in the associated queue, *Server_Queue*. Once the job entity succeeds in seizing the server, it proceeds to the *Delay* module, called *Processing Time*, and stays there for the requisite processing time. On service completion, it enters the *Release* module, called *Release Server*, where it releases the *Server* resource. Finally, the job entity traverses three consecutive *Record* modules to tally its system time (module *Tally System Time*) and the time since the previous departure (module *Tally Interdeparture Time*), and to update the number of jobs completed (module *Count Processed*). The job entity is then disposed of and leaves the system.

Figure 8.4 displays summary statistics generated by a simulation run of the single-workstation model for 100,000 hours.

We now proceed to verify various performance measures of the model, starting with throughput (note that hats, or carets, denote estimated values). For a given replication r, a throughput estimator, $\hat{o}_1(r)$, for Eq. 8.7 is given by

$$\hat{o}_1(r) = \frac{D(r)}{T(r)}$$

where $D(r)$ is the number of jobs processed to completion during replication r, and $T(r)$ is the length (duration) of replication r.

Next, an examination of Figure 8.4 verifies that replication 1 ran for $T(1) = 100{,}000$ hours, and the *Counter* section reveals that $D(1) = 20{,}661$ jobs were processed to completion, while 1578 jobs were rejected. Thus, the estimated throughput for replication 1 is given by

$$\hat{o}_1(r) = \frac{20,661}{100,000} = 0.2066 \text{ jobs/hour.}$$

4:00:20PM **Queues** January 2, 2006

WorkStation Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Server_Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	6.6325	0.246276679	0	61.2068

Other	Average	Half Width	Minimum	Maximum
Number Waiting	1.3705	0.056543971	0	4.0000

4:03:22PM **Resources** January 2, 2006

WorkStation Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

SERVER

Usage	Value			
Total Number Seized	20,662.00			
Scheduled Utilization	0.8392			
Number Scheduled	1.0000	(Insufficient)	1.0000	1.0000
Number Busy	0.8392	0.009298799	0	1.0000
Instantaneous Utilization	0.8392	0.009298799	0	1.0000

4:04:26PM **User Specified** January 2, 2006

WorkStation Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Tally

Between	Average	Half Width	Minimum	Maximum
Interdeparture Time	4.8393	0.039080331	0.00053841	42.2518

Interval	Average	Half Width	Minimum	Maximum
System Time	10.6939	0.290070384	0.00076275	65.5984

Counter

Count	Value
Processed	20,661.00
Rejected	1,578.00

Figure 8.4 Summary statistics report for the model of Figure 8.3.

On the other hand, the *Resources* section shows that utilization was estimated as $\hat{u} = 0.8392$, and by the system description, while the workstation was busy, it processed jobs at the rate of $\mu = 0.25$ jobs per hour. Using Eq. 8.7, an alternative estimate of the throughput is given by

$$\hat{o}_2(r) = \mu \times \hat{u} = 0.25 \times 0.8392 = 0.2098 \text{ jobs/hour.}$$

Furthermore, yet another alternative estimate of the throughput using the mean inter-departure time from the *Tally* section is given by

$$\hat{o}_3(r) = \frac{1}{\text{E[interdeparture time]}} = \frac{1}{4.8393} = 0.2066 \text{ jobs/hour.}$$

Finally, recall from Eq. 8.9 that the effective input rate should be the same as the effective output rate, which is simply the throughput. Let $1 - \pi_K$ be the probability that an arriving job succeeds in entering the workstation. The *Counter* section shows that the number of jobs that succeeded in entering the workstation was 20,661 out of a total of 20,661 + 1,578 jobs that arrived at the workstation. We then estimate

$$1 - \hat{\pi}_K = \frac{20661}{20661 + 1578} = 0.9290.$$

Since from Eq. 8.9 the effective arrival rate is the throughput, it follows that a fourth estimate of the throughput is simply

$$\hat{o}_4(r) = \lambda(1 - \hat{\pi}_K) = \frac{1}{4.5}\, 0.9290 = 0.2065.$$

The fact that the four throughput estimates, $\hat{o}_1(r)$, $\hat{o}_2(r)$, $\hat{o}_3(r)$, and $\hat{o}_4(r)$, are quite close, even though they were computed in different ways, is reassuring. This fact would tend to increase our confidence that the computed throughput is correct. Furthermore, the estimate $\hat{o}_2(r)$ supports the correctness of the utilization value, and the estimate $\hat{o}_4(r)$ supports the correctness of the probabilities that the system is full (and not full) at arrival instances. Consequently, these verification facts support the contention that the model is correct.

Another common verification check is based on Little's formula in Eq. 8.8. Note that the formula asserts that there are two ways to estimate the mean number of jobs in the system. One way is simply to compute the average number in the system, $\bar{N}_S$. The other way is to form the product of the estimated (effective) arrival rate λ_{eff} and the average time a job spends in the system, $\bar{W}_S$. Now, from the *Queues* section in Figure 8.4, the estimated mean number of jobs in the buffer alone is $\hat{N}_b = 1.3705$, while the estimated mean delay per job in the buffer is $\hat{W}_b = 6.6325$. Applying Little's formula (Eq. 8.8) to the buffer alone results in the relation

$$\bar{N}_b = \lambda_{eff} \times \bar{W}_b.$$

Estimating each side separately yields

$$\hat{N}_b = 1.3705$$

and

$$\hat{\lambda}_{eff} \times \hat{W}_b = 0.2065 \times 6.6325 = 1.3693.$$

Note that $\hat{N}_b$ and $\hat{\lambda}_{eff} \times \hat{W}_b$ are approximately equal, thereby providing additional confidence in the goodness of the estimates of $\bar{N}_b$, λ_{eff}, and $\bar{W}_b$.

In a similar vein, Little's formula for the entire system can be written as

$$\bar{N}_S = \lambda_{eff} \times \bar{W}_S$$

where

$$\hat{N}_S = \hat{N}_b + \hat{u} = 1.3705 + 0.8392 = 2.2085$$

and

$$\hat{\lambda}_{eff} \times \hat{W}_S = \hat{\lambda}_{eff} \times \left(\hat{W}_b + \frac{1}{\mu}\right) = 0.2065 \times (6.6325 + 4) = 2.1956.$$

Again the two computations yield acceptably close values. In fact, they should come closer and closer as the simulation run length is increased.

The previous example exhibits multiple verification tests, some of which were overlapping. In practice, it usually suffices to perform a set of nonoverlapping verification tests. Of course, the more extensive the verification effort, the higher is the modeler's confidence in the model. Note carefully, however, that verification does not conclusively prove program correctness. Rather, it merely provides circumstantial evidence for model correctness, indicating the absence of credible evidence to the contrary.

8.3.2 Model Verification in Tandem Workstations

In this example, we consider a system of two workstations in tandem: an assembly workstation followed by a painting workstation, as shown in Figure 8.5. Thus, the output of the first workstation is funneled as input to the second.

Suppose that neither workstation has buffer capacity limitations, so that all arriving jobs can enter the first workstation and eventually proceed to the second. Assume further that the arrival stream consists of two types of jobs: *type 1* and *type 2*, each undergoing its own assembly and painting operations. Job interarrival times are exponentially distributed with means 4 hours for type 1 and 10 hours for type 2. For type 1 jobs, assembly times (in hours) are distributed according to the Tria(1, 2, 3) distribution, while painting times are deterministic, lasting 3 hours. For type 2 jobs, assembly times (again in hours) are distributed according to the Tria(1, 3, 8) distribution, while painting times are deterministic, lasting 2 hours. We simulate this scenario for 100,000 hours to estimate resource utilizations and mean flow times by job type, and average WIP (work-in-process) levels in each buffer.

The Arena model corresponding to Figure 8.5 is depicted in Figure 8.6. The model has two *Create* modules, one for each job type. Arrival times, job types, and processing times are all specified in the respective *Assign* modules (*Assign Type 1 Parameters* and

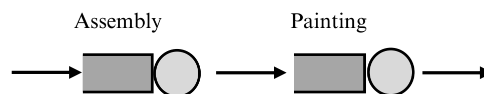


Figure 8.5 Two workstations in tandem.

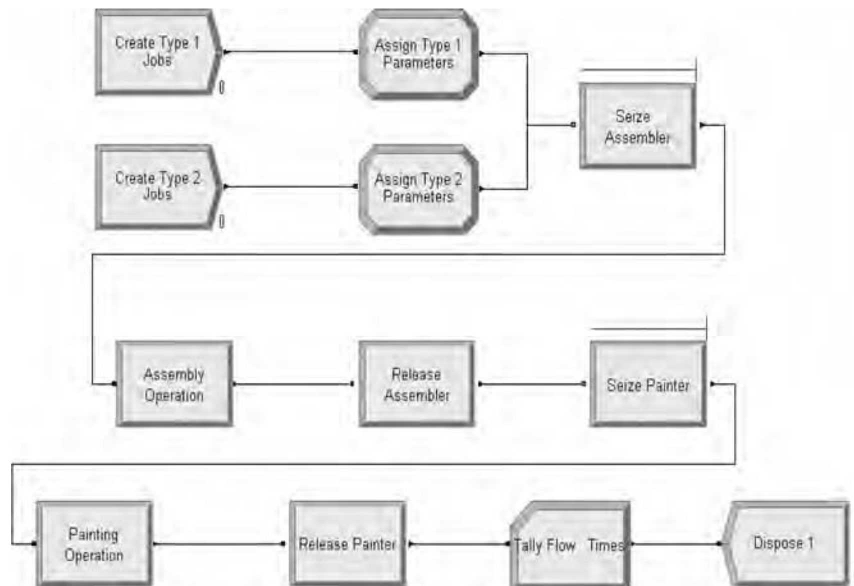


Figure 8.6 Arena model of the two-workstation flow line.

Assign Type 2 Parameters). The dialog box of the first *Assign* module is displayed in Figure 8.7.

Each job first seizes the *Assembler* resource via the *Seize* module, called *Seize Assembler*, whose dialog box is displayed in Figure 8.8.

Jobs wait in the queue *Assembler_Queue* while the resource *Assembler* is busy. The dialog box for this resource is displayed in Figure 8.9. Note the *Resource State* field in Figure 8.9. The option *Type* instructs the corresponding *Seize* module to set the state of its *Assembler* resource according to the type of the job entity being processed. Recall that the job attribute *Type* is set to the type of the job, namely, 1 or 2. When a job entity seizes the resource *Assembler*, the resource's state is assigned that job's type. It therefore

Figure 8.7 Dialog box for the *Assign* module for *type 1* jobs.

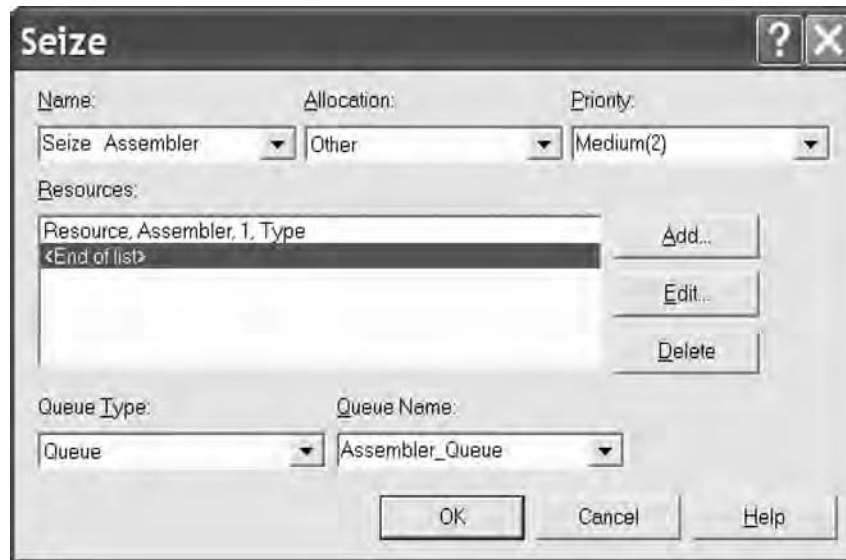


Figure 8.8 Dialog box for module *Seize Assembler*.

Figure 8.9 Dialog box for resource *Assembler*.

becomes possible, in principle, to collect resource statistics by job type, such as the percentage of time each job type keeps resource *Assembler* busy. However, this approach requires a definition in the *StateSet* module for each resource that aims to collect statistics by entity type. The resultant spreadsheet view of the *Resource* module is displayed in Figure 8.10.

Since each resource has a user-defined *StateSet* module entry, each such entry has to be defined as a separate row, as shown in Figure 8.11.

Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Initial State	Failures	Report Statistics
1	Assembler	Fixed Capacity	1	0.0	0.0	0.0	Assembler States		0 rows	☑
2	Painter	Fixed Capacity	1	0.0	0.0	0.0	Painter States		0 rows	☑

Figure 8.10 Dialog spreadsheet of the *Resource* module with *StateSet* module names.

StateSet - Advanced Process

	Name	States
1	Assembler States	2 rows
2	Painter States	2 rows

States

	State Name	AutoState or Failure
1	Processing 1	
2	Processing 2	

Figure 8.11 Dialog spreadsheet of the *StateSet* module (left) and its States dialog spreadsheet (right).

In Figure 8.11, the dialog spreadsheet at the left shows that the resources *Assembler* and *Painter* each have two states, while the dialog spreadsheet at the right indicates that the user-assigned resource state names are *Processing 1* and *Processing 2*. Each resource state will be assigned according to the type of job that seizes it.

After leaving module *Seize Assembler*, the job entity proceeds to the *Delay* module, called *Assembly Operation*, whose dialog box is displayed in Figure 8.12.

Figure 8.12 Dialog box of the *Delay* module *Assembly Operation*.

Its *Delay Time* field stipulates that the job's delay time (assembly time) is supplied by the job entity's attribute called *Assembly Time*. When the assembly operation is completed (i.e., the associated delay is concluded), the job entity proceeds to the *Release* module, called *Release Assembler*, to release the resource *Assembler*. The job entity then proceeds to carry out the painting operation analogously via the sequence of modules *Seize Painter*, *Painting Operation*, and *Release Painter*.

Finally, the job entity proceeds to the *Record* module, called *Tally Flow Times*, to tally its flow time, measured from the point of arrival at the assembly workstation (that

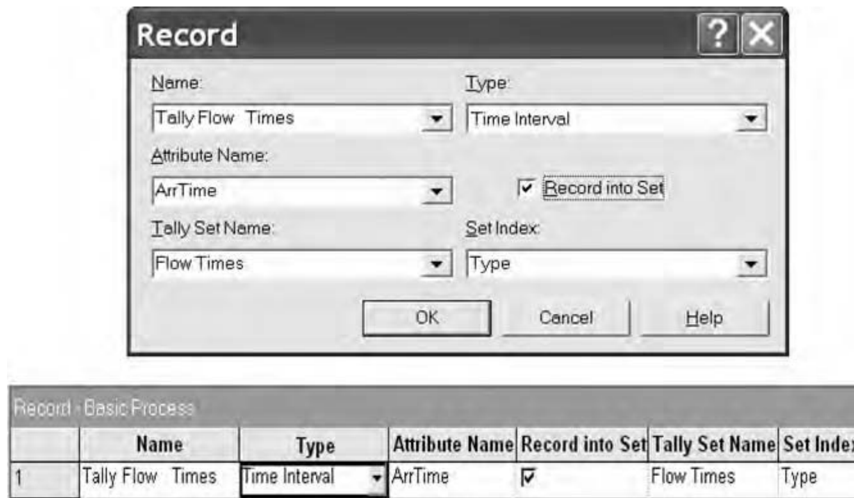


Figure 8.13 Dialog box (top) and spreadsheet view (bottom) for the *Record* module tallying flow times.

arrival time is stored in the job entity's *ArrTime* attribute). Figure 8.13 displays the dialog spreadsheet for this *Record* module (top), as well as the spreadsheet view of *Record* modules (bottom). Note that this *Record* module uses the *Time Interval* option in the *Type* field to collect job flow times. Furthermore, to indicate that flow times are to be tallied separately for each job type, the *Record into Set* check box is checked in the dialog box. Additionally, the *Tally Set Name* field specifies the tally set name, *Flow Times*, and the *Set Index* field is set to the *Type* attribute of incoming jobs to indicate flow-time tallying by job type. The tally set *Flow Times* is defined in the *Set* module from the *Basic Process* template panel, as shown in the associated dialog spreadsheets of Figure 8.14.

Figure 8.14 Dialog spreadsheet of the *Set* module (left) and its *Members* dialog spreadsheet (right).

The *Members* field on the left-side dialog spreadsheet specifies two separate tallies (the button labeled *2 rows*). Clicking that button pops up the right-side dialog box, which specifies the names of the two tallies of set *Flow Times* as *Type 1 FT* and *Type 2 FT* (these correspond to flow-time tallies of type 1 jobs and type 2 jobs, respectively).

Finally the *Statistic* module can be used to produce *Frequency* statistics representing estimated state probabilities of each resource. Figure 8.15 displays the corresponding dialog spreadsheet.

Figures 8.16 and 8.17 display the output statistics from a replication of the model depicted in Figure 8.6 of duration 100,000 hours.

Statistic - Advanced Process

	Name	Type	Frequency Type	Resource Name	Report Label	Output File	Categories
1	Assembler_States	Frequency	State	Assembler	Assembler_States		0 rows
2	Painter_States	Frequency	State	Painter	Painter_States		0 rows

Figure 8.15 Dialog box of the *Statistic* module with *Frequency* statistics.

The *Resources* section lists the utilization of each resource, while the *Frequencies* section displays the state probabilities for each resource, including resource utilization by job type. The *Queues* section contains statistics on delay times and average queue sizes (average WIP levels) for each resource queue. Finally, the *User Specified* section displays flow time statistics for each job type.

We begin our verification analysis with workstation utilization by job type. The arrival rates of type 1 and type 2 jobs at the assembly workstation are 0.25 and 0.1, respectively. Further, the mean assembly times are $(1 + 2 + 3)/3 = 2$ and $(1 + 3 + 8)/3 = 4$ for types 1 and 2 jobs, respectively (recall that processing times in each workstation are assumed to have a triangular distribution). Thus, the total workstation utilization of the assembly workstation is $\mu = 0.9$, with partial utilizations by job type,

$$u(1) = 2/4 = 0.5$$

and

$$u(2) = 4/10 = 0.4$$

where $u = u(1) + u(2)$. The simulation results from Figure 8.16 estimate the utilization of the assembly workstation as $\hat{u} = 0.889$. The two values, $u = 0.9$ and $\hat{u} = 0.889$, are close, but would be even closer if we were to run the simulation longer. As a rule, models should be run longer under heavy traffic (high utilization). The individual workstation utilizations by job type can be verified for the painting workstation in a similar manner.

Next, we proceed to verify the flow times of the two job types simultaneously. Little's formula (Eq. 8.8) may be applied to the entire system, resulting in the relation

$$\hat{N}_S = \lambda \times \hat{W}_S$$

where $\hat{N}_S$ is the simulation estimate of mean total jobs in the system, λ is the assumed total arrival rate of all jobs at the system, and $\hat{W}_S$ is the simulation estimate of the combined mean flow time through the system of jobs of all types.

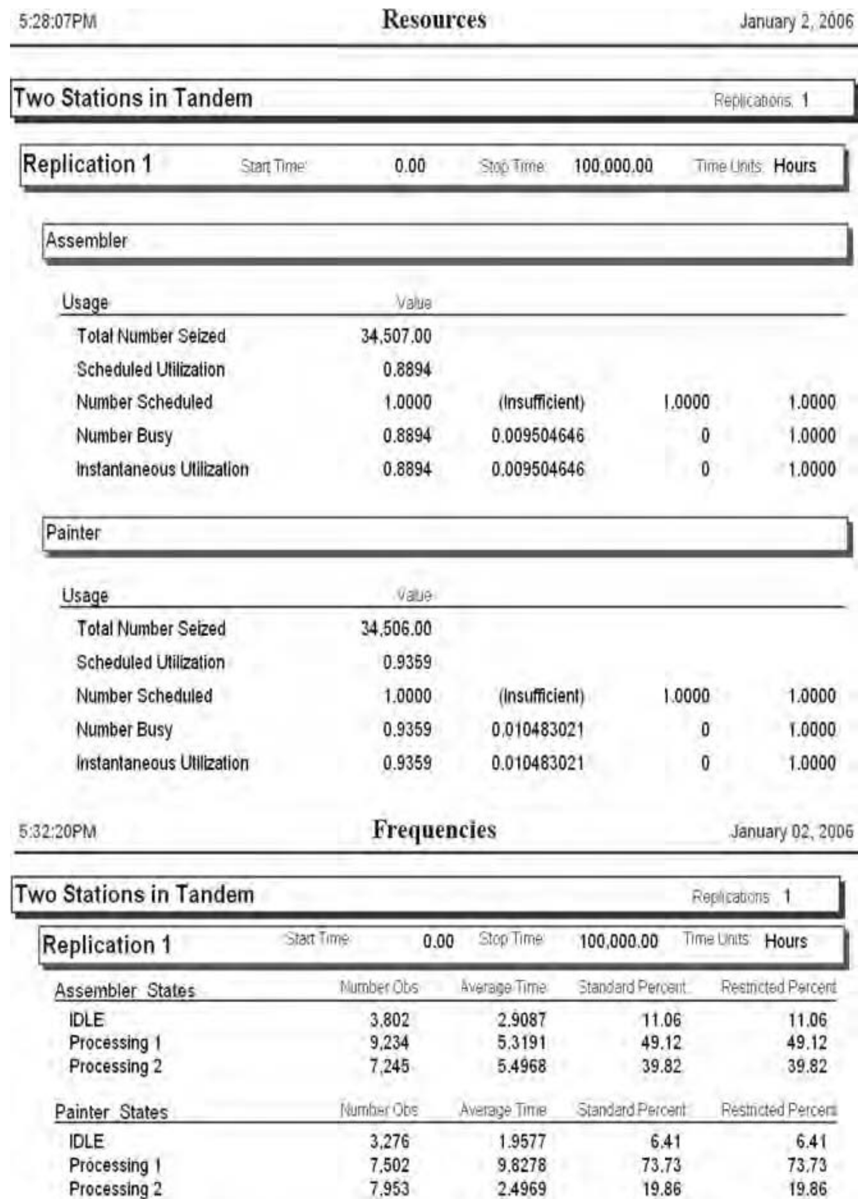
A theoretical verification of Little's formula is carried out by the following computation:

- The total arrival rate is $\lambda = 0.25 + 0.1 = 0.35$.
- The mean flow time of type 1 jobs was estimated by simulation as

$$\hat{W}_S(1) = 29.1943.$$

- The mean flow time of type 2 jobs was estimated by simulation as

$$\hat{W}_S(2) = 28.4181.$$



5:28:07PM **Resources** January 2, 2006

Two Stations in Tandem Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Assembler

Usage	Value			
Total Number Seized	34,507.00			
Scheduled Utilization	0.8894			
Number Scheduled	1.0000	(Insufficient)	1.0000	1.0000
Number Busy	0.8894	0.009504646	0	1.0000
Instantaneous Utilization	0.8894	0.009504646	0	1.0000

Painter

Usage	Value			
Total Number Seized	34,506.00			
Scheduled Utilization	0.9359			
Number Scheduled	1.0000	(Insufficient)	1.0000	1.0000
Number Busy	0.9359	0.010483021	0	1.0000
Instantaneous Utilization	0.9359	0.010483021	0	1.0000

5:32:20PM **Frequencies** January 02, 2006

Two Stations in Tandem Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Assembler States	Number Obs	Average Time	Standard Percent	Restricted Percent
IDLE	3,802	2.9087	11.06	11.06
Processing 1	9,234	5.3191	49.12	49.12
Processing 2	7,245	5.4968	39.82	39.82

Painter States	Number Obs	Average Time	Standard Percent	Restricted Percent
IDLE	3,276	1.9577	6.41	6.41
Processing 1	7,502	9.8278	73.73	73.73
Processing 2	7,953	2.4969	19.86	19.86

Figure 8.16 *Resources* and *Frequencies* reports from a replication of the model in Figure 8.6.

- The estimated combined mean flow time over all job types (weighted by the relative arrival rates) is

$$\hat{W}_S = (0.25/0.35) \times 29.1943 + (0.1/0.35) \times 28.4181 = 28.9725.$$

- The mean number of jobs in the system is estimated from the simulation run as the sum of average buffer sizes plus workstation utilizations over all job types, namely,

5:41:37PM **Queues** January 2, 2006

Two Stations in Tandem Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Assembler_Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	11.8275	1.50862	0	100.35

Other	Average	Half Width	Minimum	Maximum
Number Waiting	4.0814	0.450163015	0	37.0000

Painter_Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	11.8530	2.48641	0	81.6583

Other	Average	Half Width	Minimum	Maximum
Number Waiting	4.0900	0.777431795	0	29.0000

5:44:41PM **User Specified** January 2, 2006

Two Stations in Tandem Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Other

None	Average	Half Width	Minimum	Maximum
Type 1 FT	29.1943	2.43419	4.0365	126.85
Type 2 FT	28.4181	2.95947	3.2739	121.82

Figure 8.17 *Queues* and *User Specified* reports from a replication of the model in Figure 8.6.

$$\hat{N}_S = \sum_k \hat{N}_b(k) + \sum_k \hat{u}(k) = 4.0814 + 4.0900 + 0.8894 + 0.9359 = 9.9967.$$

Thus, we find that

$$\hat{N}_S = 9.9967 \text{ and } \lambda \times \hat{W}_S = 0.35 \times 28.9725 = 10.1404.$$

It follows that Little's formula is satisfied with acceptable precision (less than 1.5% relative error). Again, the preceding two quantities can be expected to agree more closely as the length of the simulation run increases.

8.4 MODEL VALIDATION

Validation activities are critical to the construction of credible models. The standard approach to model validation is to collect data (parameter values, performance metrics, etc.) from the system under study, and compare them to their model counterparts (see Law and Kelton [2000] and Banks et al. [2004]).

As previously discussed in Section 7.2, the data collection effort of input analysis can provide the requisite data from the system under study. Data collected are classified into input values and corresponding output values. For instance, consider a machine that processes jobs arriving at a given rate, and suppose that we are interested in delay times experienced by jobs in the buffer. Raw input data might then consist of job interarrival times, processing times, and buffer delays, which may be obtained by observing the system over, say, a number of days. In a similar vein, a model is constructed and run to produce simulation data (that serve as input data sets) from which the corresponding output data (performance metrics) are then estimated.

Table 8.1 displays schematically the structure of the correspondence between a generic input data set and the corresponding output metrics over a period of N time units (e.g., days). Each set of output metric values constitutes a sample of random values. In our case, the output metrics consist of estimated mean delay times; that is, we collect two sets (samples) of output data:

1. A sample $\{\bar{D}_1, \ldots, \bar{D}_N)$ of observed average delays collected from the real-life system under study over days $i = 1, \ldots, N$
2. A sample $\{\hat{D}_1, \ldots, \hat{D}_N\}$ of estimated mean delays collected from runs of the simulation model over days $i = 1, \ldots, N$

Thus, for the real-life system under study, $O_i = \bar{D}_i$, $i = 1, \ldots, N$, while for the simulation runs, $O_i = \hat{D}_i$, $i = 1, \ldots, N$. More precisely, model validation seeks to determine the goodness-of-fit of the model to the real-life system under study (in our case we statistically compare the average and mean delay times). Recasting this check in terms of hypothesis testing (see Section 3.11), we wish to determine whether the two delay metrics are statistically similar (null hypothesis) or statistically different (alternative hypothesis). To this end, define a sample consisting of the differences

$$G_i = \hat{D}_i - \bar{D}_i, \; i = 1, \ldots, N,$$

which are approximately normally distributed with mean μ_G and variance σ_G^2. The hypothesis testing assumes the form

Table 8.1

Generic input and output data

Day	Input Data Set	Output Metric Set
1	I_1	O_1
2	I_2	O_2
$\vdots$	$\vdots$	$\vdots$
N	I_N	O_N

$$\begin{cases} H_0: & \mu_G = 0 \\ H_1: & \mu_G \neq 0 \end{cases}$$

Then, under the null hypothesis, the statistic

$$t_{N-1} = \frac{\bar{G} - \mu_G}{S_G/\sqrt{N}}$$

is distributed according to a t distribution with $N - 1$ degrees of freedom, where $\bar{G}$ and S_G are the sample mean and sample standard deviation of the sample $\{G_1, \ldots, G_N\}$, respectively.

For a prescribed significance level, α, the corresponding confidence interval satisfying

$$\Pr(t_1 \leq t_{N-1} \leq t_2) = 1 - \alpha$$

is equivalent to

$$\Pr(\bar{G} - t_{\alpha/2,N-1}S_G/\sqrt{N} \leq \mu_G \leq \bar{G} + t_{1-\alpha/2,N-1}S_G/\sqrt{N}) = 1 - \alpha$$

where $t_1 = t_{\alpha/2,N-1}$ and $t_2 = t_{1-\alpha/2,N-1}$. If the above confidence interval contains 0, then H_0 cannot be rejected at significance level α, indicating that the test supports model validity. If, however, the confidence interval does not contain 0, then the test suggests that the model is not valid.

EXERCISES

1. *Car wash*. A car-wash facility has five self-serve washing booths. Cars arrive according to a Poisson process at the rate of one car every 10 minutes. A customer's choice of any one of the booths is equally likely. When all the booths are busy, cars join a single queue and are then served in FCFS manner. Washing times are iid uniformly distributed between 20 and 40 minutes. The facility is open from 7:00 A.M. to 6:00 P.M. every day.
   a. Develop an Arena model for the car-wash facility, simulate it for 1 year, and estimate the following performance measures:
      - Booth utilization
      - Throughput of each booth
      - Throughput of the system
      - Average number of busy booths
      - Average number of cars in the queue
      - Average delay per car in the queue
      - Average system time per car
      - Probability that an arriving car finds all booths busy
      - Probability that an arriving car finds at least one idle booth
   b. Verify your results for each of the above performance measures.
2. *Credit card operation*. The customer service department of the World Express credit card company is a 24-hour/day operation with a single service representative that handles customer telephone calls. Call interarrival times are iid exponentially distributed with a rate of four calls per hour. The handling times per customer are iid uniformly distributed between 5 and 15 minutes. As soon as five callers queue

up for service, any additional customers are lost. Exactly 40% of the calls are referred to a single specialist for further service, whose durations are distributed iid Tria(10, 18, 30) minutes. The specialist can accommodate all referred callers in a buffer, and then serves them in FCFS manner.

a. Develop an Arena model for the customer service department, and simulate it for 1 month (30 days), and estimate the following performance measures:
   - Utilization of the service manager and the specialist
   - Average number of customers in the two queues
   - Mean delays in each queue
   - Probability that a caller is rejected

b. Verify your results for each of the above performance measures.

3. *Machine failure and repair.* The machining department of a tool manufacturer has a shop floor with 25 identical machines producing various types of tools. Machines are operated 24 hours a day continually, except when they are down. Failed machines are repaired by any one of six technicians in the repair shop, and repair times are distributed iid Tria(7.5, 15, 36) hours. After a machine is repaired, it continues to operate until it fails again. Times to failure are iid exponentially distributed with mean 60 hours. When more than six machines are down, each is kept in a queue to be repaired in FIFO manner.

   a. Develop an Arena model for the shop floor, simulate it for 365 days, and estimate the following performance measures:
      - Average rate of machine failure on the shop floor
      - Average waiting time for repair by a technician
      - Average time spent in the repair shop
      - Average number of down machines
      - Average number of busy repair technicians

   b. Verify your results for each of the above performance measures.

This page intentionally left blank

Chapter 9

Output Analysis

Recall that a Monte Carlo model is governed by a probability law that determines its random behavior. Except for very simple cases rarely encountered in practice, that law is too complicated to write down, and consequently, we cannot analytically derive system statistics either. Rather, we develop a simulation program that encapsulates that probability law by scheduling and processing random events. Each run of the simulation program, called a *replication* in simulation parlance, produces a sample system history from which various statistics are estimated via *output analysis*.

Output analysis is the modeling stage concerned with designing replications, computing statistics from them and presenting them in textual or graphical format. Thus, as its name suggests, output analysis focuses on the analysis of simulation results (output statistics). It provides the main value-added of the simulation enterprise by trying to understand system behavior and generate predictions for it. The main issues addressed by output analysis follow:

- *Replication design*. A good design of simulation replications allows the analyst to obtain the most statistical information from simulation runs for the least computational cost. In particular, we seek to minimize the number of replications and their length, and still obtain reliable statistics.
- *Estimation of performance metrics*. Replication statistics provide the data for computing point estimates and confidence intervals for system parameters of interest (see Section 3.10). Critical estimation issues are the size of the sample to be collected and the independence of observations used to compute statistics, particularly confidence intervals. Recall that estimates are used at the validation stage to test the goodness-of-fit of a given simulation model to empirical data (see Chapter 8). The observed goodness-of-fit would impact the analyst's confidence in the predictive power of the simulation model.
- *System analysis and experimentation*. Statistical estimates are used in turn to understand system behavior and generate performance predictions under various scenarios, such as different input parameters (*parametric analysis*), scenarios of operation, and so on. Experimentation with alternative system designs can elucidate their relative merits and highlight design trade-offs.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

This chapter addresses generic output analysis issues, such as data collection for various statistics under various simulation regimes (terminating or steady state), estimation of performance measures, and testing of statistical hypotheses. Finally, it surveys the output analysis facilities supported by Arena via a detailed working example.

9.1 TERMINATING AND STEADY-STATE SIMULATION MODELS

Simulation models can be classified into two main classes based on their time horizon: *terminating* models and *steady-state* models. Each class gives rise to different statistical issues relating to their output analysis.

9.1.1 Terminating Simulation Models

A *terminating* simulation model has a natural termination time for its replications that is inherent in the system. In such models, the modeler is interested in short-term system dynamics and statistics within the system's natural time horizon. An example of a terminating model is a bank that opens daily at 8:00 A.M. and closes at 4:00 P.M. Here, the modeler might be interested in short-term performance measures, such as the daily maximal waiting time, or the daily maximal number of customers waiting to be served. Since each business day starts afresh at 8:00 A.M. with a new customer arrival stream, successive days cannot be juxtaposed to form a longer replication, say, of 1 week. It simply does not make sense to extend the replication beyond the natural termination time (in this case, beyond 4:00 P.M.).

In terminating simulation models, the number of replications is the critical parameter of the associated output analysis, since it is the only means of controlling the sample size of any given estimator. Recall that the sample size affects directly the estimator's variance, and consequently, its statistical accuracy (see Section 3.10).

9.1.2 Steady-State Simulation Models

A *steady-state* simulation model has no natural termination time for its replications, and could be potentially run "forever." In such models, the modeler is interested in long-term dynamics and statistics. An example of a steady-state model is an ATM (automatic teller machine) that remains open 24 hours a day, and is subject to failures. Here, the modeler might be interested in long-term (steady-state) performance metrics such as the long-term fraction of time (probability) the machine is up, or the mean economic loss incurred per day by foregoing ATM fees when the ATM is down.

In this book, we are more interested in steady-state simulation models than in terminating ones. For further information, see Law and Kelton (2000), Nelson (1992), Banks et al. (2004), and Kleijnen (1987).

Note that the initial state of a system tends to exert a bias on replication statistics, at least for a while. As an example, consider again an ATM at a busy airport, which almost always has a line of customers, day or night. Clearly, an empty waiting line is not a "typical" state in the sense that its long-term probability (fraction of time the line is

empty) is quite small. Thus, if started with an empty waiting line, the simulated queue statistics will be uncharacteristically low for a while, but will eventually tend to their long-term values. In simulation parlance, the simulated initial "atypical" history is called *transient state*, as opposed to the simulated "typical" history that evolves later, which is called *steady state*. The reader is cautioned that this terminology is technically a misnomer, and should be regarded as merely suggestive. In fact, the state of almost all systems is highly dynamic, and over time a stochastic system evolves and changes continually. What is actually meant is that *state statistics*, computed over progressively longer time periods, exhibit *transient-state* behavior or *steady-state* behavior, respectively. More precisely, the transient-state regime is characterized by statistics that vary as a function of time, while the steady-state regime prevails when statistics stabilize and do not vary over time. In between these two regimes, there is typically a transition period when the system approaches the steady-state regime—a period characterized by small and generally decreasing variability of the statistics over time. For all practical purposes, the system may be considered to be *approximately* in steady state during that transition period, whereas true steady state is attained, with few exceptions, only asymptotically, as time tends to infinity. Steady-state simulations seek, however, to estimate such long-term steady-state statistics.

In steady-state simulation, only long-term statistics are of interest, but initial system conditions tend to bias its long-term statistics. Therefore, it makes sense to start statistics collection *after* an initial period of system *warm-up*, namely, after the biasing effect of the initial conditions decays to insignificance. In fact, if statistics *are* collected during warm-up, they should be discarded after warm-up, and their collection restarted at that point. In Arena, the warm-up period can be declared in the *Replication Parameters* tab of the *Setup*... option of the *Run* menu. Alternatively, if we do not wish to suspend initial statistics collection, then each replication length should be increased to reduce the biasing effect of initial conditions. This approach, while feasible, is not attractive, as it incurs an additional (and unnecessary) computational cost.

As an example, consider a workstation where jobs arrive with exponential interarrival times of mean 1 hour, and a fixed unit processing time of 0.75 hours. Figure 9.1 depicts the throughput of the workstation and the average job delay in the buffer as functions of time.

Note the high variability exhibited by both statistics in the time interval from 0 to 330 hours, followed by markedly more stable statistics thereafter. The system thus appears to be distinctly in transient state up until time $t = 330$ or so, and to approach steady state

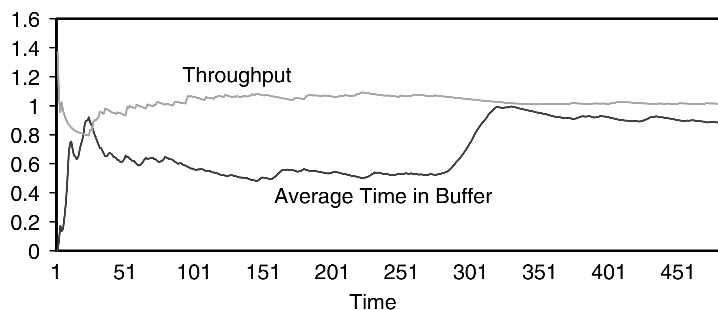


Figure 9.1 Workstation throughput and average job delay as functions of time.

thereafter. Indeed, the throughput tends to the arrival rate when computed over longer and longer time intervals. The transient behavior of the average job delay seems to be in effect longer than that of the throughput. It is, therefore, advisable to run the system without statistics collection until perhaps time $t = 400$, and to start collecting steady-state statistics thereafter.

Steady-state models have a number of important issues associated with their output analysis.

- Since steady-state models have no natural termination time, how does one select a replication length?
- Since a warm-up period is advantageous, how does one select warm-up length?

The guiding principle of replication length is stabilization of the statistics of interest. In practice, one selects a convenient time increment, and records the statistics collected from the initial time until the end of each increment (note that the corresponding collection intervals are progressively longer). The replication is stopped when statistics at the end of several successive increments are sufficiently close (e.g., within a specific ε, to be determined by the analyst). Another approach is to evaluate the two sides of a known relation (e.g., Little's formula, Eq. 8.8, applied to a queueing subsystem). The replication would be terminated when both sides of the relation yield sufficiently close values.

In a similar vein, the length of a warm-up period is determined by observing experimentally when the time variability of the statistics of interest is largely eliminated. However, unlike replication length, there is no point in waiting until the statistics are perfectly stabilized (in that case, the system is already in steady state). Note that warm-up is employed just to shorten the time it takes the statistics to stabilize. In practice, a statistics plot over time, such as Figure 9.1, can be extremely helpful in determining the length of warm-up, simply by inspection. It should also be mentioned that this issue can be rendered moot by just increasing the overall replication length and foregoing warm-up altogether.

Generally, a replication can be terminated by time, number of observations, or triggered by a logical condition. In Arena, terminating a replication at a prescribed time is achieved by specifying a value for the *Replication Length* field in the *Replication Parameters* tab of the *Setup...* option of the *Run* menu. The *Terminating Condition* field on the same tab can also be used to terminate replications, typically utilizing event counts, such as a prescribed number of departures or failures. For instance, Arena might terminate a replication as soon as the value of the current counter in a *Record* module exceeds a prescribed value specified in the aforementioned *Terminating Condition* field.

9.2 STATISTICS COLLECTION FROM REPLICATIONS

Recall that each replication produces a random history (sample path), from which various statistics are computed (see Section 3.10). These statistics are estimates of various parameters of interest (probabilities, means, variances, etc.). More formally, suppose we are interested in a *parameter* θ of the system (e.g., the mean flow times of jobs through a workstation or their blocking probability on trying to enter a finite buffer). The simulation will then be programmed to produce an *estimator*, $\hat{\Theta}$, for the true but unknown parameter, θ, which evaluates to some *estimate*, $\hat{\Theta} = \hat{\theta}$. Note carefully the distinct meanings of the related entities θ, $\hat{\Theta}$, and $\hat{\theta}$:

- θ is a deterministic but unknown parameter (possibly vector-valued).
- $\hat{\Theta}$ is a *variate* (random variable) estimator of θ.
- $\hat{\theta}$ is some realization of $\hat{\Theta}$.

For each replication r, the estimator $\hat{\Theta}$ yields a separate estimate, $\hat{\Theta}(r) = \hat{\theta}(r)$. Furthermore, $\hat{\Theta}(r)$ is some function of the history of replication r. For example, suppose that θ is the (unknown) mean of flow times through a workstation. An estimator $\hat{\Theta}$ of θ might then be chosen as the sample mean (see Section 3.10) of job flow times, $\{X_1(r), \ldots, X_n(r)\}$, where $X_j(r)$ is the j-th job flow time observed during replication r and n is the number of jobs that departed from the workstation. In short, $\hat{\Theta}$ is the customer average $\bar{X}(r)$ observed in the course of replication r. As another example, suppose that θ is the (unknown) machine utilization. An estimator $\hat{\Theta}$ of θ is chosen as the fraction of time the workstation is busy processing in the course of replication r. In short, $\hat{\Theta}$ is the time average of the corresponding history $\{Y_t(r) : A(r) \le t \le B(r)\}$, where

$$Y_t(r) = \begin{cases} 1, & \text{if machine is busy} \\ 0, & \text{if machine is idle} \end{cases}$$

is the indicator function of the machine status at time t during replication r, and $[A(r),B(r)]$ is the time interval of replication r.

9.2.1 Statistics Collection Using Independent Replications

Although output analysis can extract information from a single replication, in a typical situation a simulation program is run n times with independent initial random number generator (RNG) seeds (see Chapter 4). These simulation runs constitute n independent replications that produce a random sample $\{\hat{\theta}(1), \ldots, \hat{\theta}(n)\}$ of estimates of θ, drawn from the underlying estimator $\hat{\Theta}$; recall that "random" means here that the sample is drawn from iid (independent and identically distributed) experiments. Output analysis would then use this random sample to form a *pooled* estimate for θ, based on all n replications, in order to increase the statistical reliability of estimating θ (see Section 9.3). Examples include the sample mean, sample variance, and confidence intervals.

As an example, consider again the sequence of job flow times generated in replication r, and denote the j-th job flow time by $X_j(r)$ and its realization by $x_j(r)$. Suppose the estimator of the mean flow time is the sample mean of the flow times in each replication, yielding the estimates

$$\bar{x}(r) = \frac{1}{l(r)} \sum_{j=1}^{l(r)} x_j(r), \quad r = 1, \ldots, n \tag{9.1}$$

where $l(r)$ is the number of flow time observations recorded in replication r. Note carefully that within each replication r, the individual flow times $X_j(r)$, $j = 1, \ldots, l(r)$ are *dependent* variates, whereas the corresponding estimates $\bar{x}(r)$, $r = 1, \ldots, l(r)$ across replications are realizations of the *independent* variates, $\bar{X}(r)$, $r = 1, \ldots, n$. This independence is due to the fact that distinct replications using independent streams of random numbers are statistically independent, and consequently, so are the statistics computed from them. As we shall see, independent estimators are necessary in order to

take advantage of the *central limit theorem* (see Section 3.8.5) to obtain the distribution of pooled estimators and construct from them confidence intervals for the true parameter to be estimated (see Section 9.4).

9.2.2 Statistics Collection Using Regeneration Points and Batch Means

In the previous section we assumed that each replication yields precisely one estimate for the statistic under consideration. For example, each replication of a workstation model was assumed to yield one estimate for the mean flow time, one estimate for the probability of blocking, and so on. However, replications can be time consuming. In addition, each replication warm-up period represents additional overhead, if not outright "wastage."

One obvious way to speed up estimation is to collect several estimates for the same statistic from any given replication. Note that such an estimation strategy would automatically reduce "warm-up" overhead. We would like these multiple estimates to be drawn from a sequence of estimators that are independent (or approximately so), both *across* replications (as is the case for independent replications), as well as *within* replications. In fact, if we can obtain independent estimates within replications, it would suffice to run just one replication, albeit a long one. To this end, one must identify when estimators within replications are indeed independent or approximately so.

A case in point is the class of *regenerative (renewal) processes* (see Section 3.9.3). Recall that such processes have specific time points (usually random), $T_1, T_2, \ldots$, such that the partial process histories over the regenerative (renewal) intervals $[T_j, T_{j+1})$ are iid. Consequently, statistical estimates collected over distinct regenerative intervals are also iid. In simple cases, one can actually identify such renewal intervals. For example, in a queueing system with independent interarrival times and service times independent of arrivals, the random arrival times of jobs at an idle system are regenerative points within a replication. Generally, however, identifying regeneration points in a complex system that are justified by theoretical arguments is difficult or impossible for all practical purposes.

A practical way of collecting multiple estimates from a single replication is the *batch means* method. As the name suggests, the main idea is to group observations into batches, and collect one estimate from each batch, provided the *batch means* are iid or approximately so. Note carefully that if the estimates are markedly dependent, then the central limit theorem (see Section 3.8.5) cannot be justifiably invoked to construct confidence intervals for the true parameters, based on pooled estimators. In batch means estimation, a replication generates a system history (sample path realization) from which statistical estimates are formed depending on whether the time index of the history is discrete or continuous.

Suppose the observed history is a discrete sample $\{x_1, \ldots, x_n\}$, for example, the flow times of n jobs. In this case, the total of n observations is divided into m batches of size k each ($n = mk$). This results in m batches (subsamples)

$$\{x_{11}, \ldots, x_{1k}\}, \{x_{21}, \ldots, x_{2k}\}, \ldots, \{x_{m1}, \ldots, x_{mk}\}.$$

From each batch $j = 1, \ldots, m$, a separate estimate $\hat{\theta}_j$ is formed from the k observations $\{x_{j1}, \ldots, x_{jk}\}$ of that batch only. The replication thus yields a set of estimates

$\{\hat{\theta}_1, \ldots, \hat{\theta}_m\}$. The batch size, k, is selected to be large enough so as to ensure that the corresponding estimators are iid or approximately so.

Suppose the observed history is a *continuous* sample $\{x_t: a \le t \le b\}$, such as the number of jobs in the buffer during time interval $[a,b]$. In this case, the interval $[a,b]$ is divided into m batches of length $c = (b - a)/m$ each, resulting in m subhistories,

$$\{x_t: t \in [a, a + c]\}, \{x_t: t \in [a + c, a + 2c]\}, \ldots, \{x_t: t \in [a + (m - 1)c, b]\}.$$

From each batch $j = 1, \ldots, m$, a separate estimate $\hat{\theta}_j$ is formed from the observations $\{x_t: t \in [a + (j - 1)c, a + jc]\}$ of that batch only. The replication thus yields again a set of estimates $\{\hat{\theta}_1, \ldots, \hat{\theta}_m\}$. The batch length, c, is selected to be long enough so as to ensure that the corresponding estimators are iid or approximately so.

9.3 POINT ESTIMATION

Consider a generic steady-state replication r, during which the simulation program collects some statistics in order to estimate their steady-state values. Since the replication is fixed in this section, we will simplify the notation by suppressing the replication index.

9.3.1 Point Estimation from Replications

Suppose the replication collects a sequence of n variates, $\{X_1, \ldots, X_n\}$, yielding a corresponding sample of observations, $\{x_1, \ldots, x_n\}$. The estimator for the mean value parameter is the sample mean

$$\bar{X} = \frac{1}{n}\sum_{j=1}^{n} X_j. \tag{9.2}$$

The sample mean is classified as a *point estimator*, because it estimates a scalar. In a similar vein, when the realizations $\{x_1, \ldots, x_n\}$ are substituted into Eq. 9.2, the resulting sample mean is similarly referred to as a *point estimate*. For example, the sample values might represent the buffer delay experienced by successive job arrivals. Recall that in this context, the average is classified as a *customer average* statistic for the obvious reason that each index j in Eq. 9.2 corresponds to customer j, and the averaging is carried out over a sequence of customer-oriented variates.

Suppose we are interested in a continuous-time stochastic process $\{X_t: 0 \le t \le T\}$ over some time interval $[0, T]$, yielding a corresponding sample of observations, $\{x_t, : 0 \le t \le T\}$. The estimator for the mean value parameter in this case is the point estimator

$$\bar{X} = \frac{1}{T}\int_0^T X_t \, dt \tag{9.3}$$

referred to as a *time average* statistic, because the variates involved are indexed by time. In fact, time averages of the form in Eq. 9.3 constitute the continuous analog of Eq. 9.2. Again, when the realizations $\{x_t: 0 \le t \le T\}$ are substituted into Eq. 9.3, the resulting time average is similarly referred to as a *point estimate*.

A common example of time-continuous variates in the queueing context is the total number of jobs, N_t, in a queueing system (buffer and server) at time t. Another important continuous-time stochastic process is the server utilization process $\{U_t: 0 \le t \le T\}$, defined via an indicator variate as

$$U_t = \begin{cases} 0, & \text{if } N_t = 0 \\ 1, & \text{if } N_t > 0. \end{cases} \tag{9.4}$$

The utilization statistic is the time average

$$\bar{U} = \frac{1}{T} \int_0^T U_t \, dt. \tag{9.5}$$

which is an estimator of the probability that the server is busy. Clearly, $U_t = 1$ when the server is busy, while $U_t = 0$ when the server is idle. It follows that the integral is just the *length of time* (in [0, *T*]) during which the server is busy, and the utilization is the *fraction of time* the server is busy. More generally, the probability of any event expressed in terms of a time-continuous stochastic process is estimated by the time average of the corresponding event indicator variates. For more examples of customer averages and time averages, see Section 2.3.1.

9.3.2 Point Estimation in Arena

The *Statistic* module allows the user to obtain estimates for *Tally* statistics (system-defined and user-defined customer averages), and *Time Persistent* statistics (system-defined and user-defined time averages), although Arena automatically collects statistics for queue lengths, delays, and utilizations. Both *Tally* and *Time Persistent* statistics permit user access to a number of Arena variables, such as *TAVG*, *DAVG*, and so on. For example, let *Some_Tally_Stat* and *Some_Time_Persistent_Stat* be the names of statistics declared in the *Statistic* module. The user may request computation of the dynamic values of the running averages *TAVG*(*Some_Tally_Stat*) and *DAVG*(*Some_Time_Persistent_Stat*) of the corresponding variables at any time during a simulation run. Observe that *Time Persistent* statistics allow the user to compute probabilities as expectations of indicator functions of events. For example, the probability that the queue *Some_Queue* has more than four waiting jobs can be estimated by setting the *Type* field of the *Statistic* module to *Time-Persistent*, and entering the predicate (logical condition) *NQ*(*Some_Queue*) > 4 in the corresponding *Expression* field. For a full listing of Arena variables, refer to the *Arena Variables Guide*.

As another example of point estimation in Arena, consider a workstation subject to failure, where jobs arrive with exponential interarrival times of mean 1 hour, and have a fixed processing time of 0.75 hours. Specifically, the workstation goes through up/down cycles as follows: It fails randomly (while busy) with exponentially distributed time-to-failure of mean 20 hours, and is then repaired with uniformly distributed repair times between 1 and 5 hours. Clearly, the system can be expected to exhibit higher waiting times than the corresponding system without failures, since the introduction of failures increases the probability that the machine is not available to arriving jobs. Based on analytical calculations, we expect the throughput to be one job per hour (same as the

Table 9.1
Results of five replications of a workstation subject to failures

Replication Number	Throughput	Average Job Delay	Probability of Down State
1	1.0096	3.9111	0.1144
2	0.9858	3.9307	0.1133
3	1.0002	3.4373	0.1038
4	1.0176	3.4243	0.1116
5	0.9999	3.5828	0.1147

arrival rate), the downtime probability to be 0.1125, and the average job delay in the buffer to be 4.11 hours per job (see Altiok [1997], Chapter 3). The results of five replications of the workstation are displayed in Table 9.1.

Note that the estimates vary across replications (the underlying estimator is a random variable!). Any of the values can be used to estimate the true (unknown) parameter, but how confident can the modeler be about their accuracy? Intuitively, the accuracy should improve by forming a pooled estimate of the sample mean, that is, by averaging over all five replication estimates. However, we still lack quantitative information on the confidence to be ascribed to any estimate. The next section addresses the confidence issue via interval estimation.

9.4 CONFIDENCE INTERVAL ESTIMATION

Confidence interval estimation quantifies the confidence (probability) that the true (but unknown) statistical parameter falls within an interval whose boundaries are calculated using appropriate point estimates (see Section 3.10). Standard statistical procedures for confidence interval estimation assume that the underlying sample consists of iid observations. Recall from Section 9.2.1 that one way of obtaining iid observations is to generate multiple replications whose random number streams are independent. Recall further from Section 9.2.2 that in a steady-state simulation setting, one can also use the batch means method to obtain multiple estimates from a single replication. For more information, see Alexopoulos and Seila (1998) and Nelson (1992).

9.4.1 Confidence Intervals for Terminating Simulations

In this section we illustrate confidence interval estimation of an (unknown) parameter θ in a terminating simulation. Following the setting in Section 9.1, we assume that n independent fixed-length replications of the model were run, and produced a sample $\{\hat{\theta}(1), \ldots, \hat{\theta}(n)\}$, where $\hat{\theta}(r)$ is the point estimate of θ produced by replication r. The pooled point estimator for θ, is the sample mean across replications

$$\bar{\Theta} = \frac{1}{n}\sum_{r=1}^{n}\hat{\Theta}(r) \qquad (9.6)$$

and the corresponding pooled estimate $\bar{\theta}$ is obtained from Eq. 9.6 by substituting $\hat{\Theta}(r) = \hat{\theta}(r)$ on its right-hand side for each $r = 1, \ldots, n$. Note carefully that the estimator $\bar{\Theta}$ in Eq. 9.6 is a random variable with mean μ and variance σ^2/n. Thus, increasing the number of replications, n, would decrease the variance of $\bar{\Theta}$, and consequently, increase our confidence in the point estimate value $\bar{\theta}$, produced by $\bar{\Theta}$.

However, we aim to further quantify this confidence by computing (at least approximately) the probability of events of the form

$$\Pr\{\hat{\Theta}_1 \le \theta \le \hat{\Theta}_2\} = 1 - \alpha \tag{9.7}$$

where the estimators $\hat{\Theta}_1$ and $\hat{\Theta}_2$ define a (random) confidence interval $[\hat{\Theta}_1, \hat{\Theta}_2]$ for θ, and α is the probability that the confidence interval *does not* include θ (α is a small probability, typically around 0.05). Recall from Section 3.10 that Eq. 9.7 specifies a confidence interval at confidence level (or significance level) α. To be able to obtain confidence interval estimates from Eq. 9.7, two conditions should hold:

1. The distribution of $\bar{\Theta}$ is known, at least approximately.
2. The distribution of $\bar{\Theta}$ can be expressed in terms of the unknown parameter θ.

To satisfy condition 1, observe that the sample $\{\hat{\theta}(1), \ldots, \hat{\theta}(n)\}$ was drawn from a set of iid random variables, $\{\hat{\Theta}(1), \ldots, \hat{\Theta}(n)\}$, with common mean μ and common variance σ^2. Under the central limit theorem (see Section 3.8.5), the random variable $\sum_{r=1}^{n} \hat{\Theta}(r)$ in Eq. 9.6 is approximately normally distributed with mean $n\mu$, and variance $n\sigma^2$. From properties of the normal distribution, it follows that $\bar{\Theta} \sim \text{Norm}(\mu, \sigma^2/n)$ (approximately), and the approximation improves as the sample size, n, tends to infinity.

To satisfy condition 2 we shall assume that the estimators $\hat{\Theta}(r)$ are unbiased for θ, so that we can write

$$\mu = \mathrm{E}[\bar{\Theta}] = \mathrm{E}[\hat{\Theta}(r)] = \theta \tag{9.8}$$

(or at least approximately so, if the bias is tolerable).

In practice, it is convenient to express confidence interval probabilities of the form in Eq. 9.7, not only in terms of $\bar{\Theta}$, but also in terms of the standardized normal random variable

$$Z = \frac{\bar{\Theta} - \mu}{\sigma/\sqrt{n}} \sim \text{Norm}(0, 1) \tag{9.9}$$

and its quantiles z_α, where $\Pr\{Z \le z_\alpha\} = \alpha$. The reason is that tables for z_α as function of α are widely available in books and in computer programs, and there is no need to have separate tables for other values of μ and σ^2. To see how this works, see Figure 9.2. Note that $z_{\alpha/2} = -z_{1-\alpha/2}$ and the probability is written as follows:

$$\Pr\{-z_{1-\alpha/2} \le Z \le z_{1-\alpha/2}\} = 1 - \alpha \tag{9.10}$$

Next, substitute Z from Eq. 9.9 into Eq. 9.10. Since $\mu = \theta$, simple algebra then yields the final representation of a confidence interval for the mean value at level α as

$$\Pr\{\bar{\Theta} - z_{1-\alpha/2}\sqrt{\sigma^2/n} \le \mu \le \bar{\Theta} + z_{1-\alpha/2}\sqrt{\sigma^2/n}\} = 1 - \alpha, \tag{9.11}$$

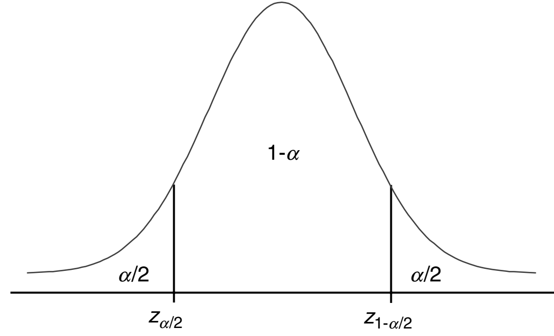


Figure 9.2 Confidence interval estimation using the standardized normal distribution.

which is often abbreviated as

$$\bar{\Theta} \pm z_{1-\alpha/2}\sqrt{\sigma^2/n}. \tag{9.12}$$

The notation in Eq. 9.12 highlights the fact that the confidence interval for the mean value is symmetric about the (unbiased) point estimator, $\bar{\Theta}$, with half-width $z_{1-\alpha/2}\sqrt{\sigma^2/n}$. Note that for a fixed α, the half-width is a measure of the accuracy of the associated confidence interval estimator: The narrower it is, the smaller its variance. But since the quantile $z_{1-\alpha/2}$ and the variance σ^2 are fixed values, it is clear from Eq. 9.11 that the accuracy of a confidence interval can be enhanced only by increasing the number of replications, n. This analysis conforms to the old adage of "no free lunch": To increase the statistical accuracy of estimates, one must pay in the coin of additional computation.

Finally, the confidence interval in Eq. 9.11 assumes that the variance σ^2 is known. Unfortunately, this is often not the case, and σ^2 needs to be estimated from the sample $\{\hat{\Theta}(1), \ldots, \hat{\Theta}(n)\}$ by the sample variance (see Section 3.10),

$$S_{\hat{\Theta}}^2 = \frac{1}{n-1}\sum_{r=1}^{n}[\hat{\Theta}(r) - \bar{\Theta}]^2.$$

Substituting the sample variance above into Eq. (9.9) results in a new random variable

$$T_{n-1} = \frac{\bar{\Theta} - \mu}{S_{\hat{\Theta}}/\sqrt{n}} \tag{9.13}$$

distributed according to the Student t distribution with $n - 1$ degrees of freedom (see Section 3.8.8). This distribution has a slightly larger variance than the corresponding standard normal distribution, Norm(0,1), but it converges to it as n tends to infinity. The quantile of the Student t distribution with n degrees of freedom at significance level α is denoted by $t_{n,\alpha}$, namely, $\Pr\{T_n \le t_{n,\alpha}\} = \alpha$. The confidence interval in this case becomes

$$\bar{\Theta} \pm t_{n-1,1-\alpha/2}\sqrt{S_{\hat{\Theta}}^2/n}. \tag{9.12}$$

For a given significance level α, the increased variance of the Student t distribution relative to the standard normal distribution will in turn tend to result in confidence intervals in Eq. 9.14 that are slightly wider than their normal counterparts in Eq. 9.12.

This is due to the added randomness in estimating the underlying variance by the estimator $S^2_{\hat{\Theta}}$. For a detailed discussion of output analysis in terminating simulations, refer to Law (1980) and Law and Kelton (2000).

9.4.2 Confidence Intervals for Steady-State Simulations

The methodology for estimating confidence intervals from replications of a terminating simulation (see Section 9.4.1) carries over to the present case of steady-state simulation. However, steady-state replications tend to be longer than their terminating counterparts, and consequently take longer to compute. On the other hand, longer replications often allow the application of the batch means method.

Consider next confidence interval estimation for some mean $\theta = \mu$ in a batch means setting with a single replication consisting of m batches (regardless of whether the underlying history was discrete or continuous). The corresponding set of m estimators, $\{\hat{\Theta}_1, \ldots, \hat{\Theta}_m\}$, gives rise to a sample mean

$$\bar{\Theta} = \frac{1}{m}\sum_{j=1}^{m} \hat{\Theta}_j$$

and sample variance

$$S^2_{\hat{\Theta}} = \frac{1}{m-1}\sum_{j=1}^{m} [\hat{\Theta}_j - \bar{\Theta}]^2.$$

Finally, the confidence interval for $\theta = \mu$ at significance level α is

$$\bar{\Theta} \pm t_{m-1,\,1-\alpha/2}\sqrt{S^2_{\hat{\Theta}}/m}.$$

For other approaches to the construction of confidence intervals, including methods producing a desired half-width, refer to Banks et al. (2004), Brately et al. (1987), Fishman and Yarberry (1997), Law (1977), Law and Kelton (1982), and Law and Kelton (2000).

9.4.3 Confidence Interval Estimation in Arena

Standard Arena output (see Sections 5.4 and 5.5) provides 95% batch means confidence intervals for each replication. These confidence intervals are computed for both *Tally* and *Time Persistent* statistics in terms of half-widths under the *Half Width* column heading. If, however, the estimated batch means are significantly *dependent* or the underlying sample history is too short to yield a sufficient number of batches, that column will display the message *(Insufficient)* to indicate that the data are not appropriate or inadequate for confidence interval estimation.

Arena also supports the computation of confidence intervals from multiple replications as a SIMAN summary report. The analyst can request this report by selecting the *Setup...* option in the *Run* menu, clicking on its *Reports* tab, and finally selecting the option *SIMAN Summary Report(.out file)* in the *Default Report* field. For Arena model name *some_model*, the report is placed in file *some_model.out*, and can be examined in any text editor.

9.5 OUTPUT ANALYSIS VIA STANDARD ARENA OUTPUT

In this section we review in some detail the output analysis facilities provided by the standard Arena output report via a working example, while Section 9.6 covers the Arena *Output Analyzer* and Section 9.7 covers the Arena *Process Analyzer*. The reader is encouraged to consult the Arena manual for more information.

9.5.1 WORKING EXAMPLE: A WORKSTATION WITH TWO TYPES OF PARTS

Consider a single-machine finishing operation in a workstation that processes two types of parts, denoted by G_1 and G_2. We make the following assumptions:

- Parts of type G_1 arrive according to iid exponential interarrival time distributions with common mean 2 hours, and each part has a fixed processing time of 1 hour.
- Parts of type G_2 arrive according to iid exponential interarrival time distributions with common mean 4 hours, and each part has a fixed processing time of 1.4 hours.
- All parts are processed in FIFO order and all part types have equal service priorities.

We wish to simulate the finishing operation for 10,000 hours in order to understand the behavior of the number of parts in the workstation buffer and the buffer delay for each type of parts. An Arena model for our simple system is depicted is in Figure 9.3.

The model consists mostly of modules from the Arena *Basic Process* template panel. There are two *Create* modules, one for each part type, so that each of these modules generates a distinct arrival stream. Each arriving part entity then proceeds to the corresponding *Assign* modules, called *Assign Operation Time of G_1* and *Assign Operation Time of G_2*, where the part type is saved in its *Type* attribute, the arrival time is saved in its *ArrTime* attribute, and the operation time is saved in its *Operation*

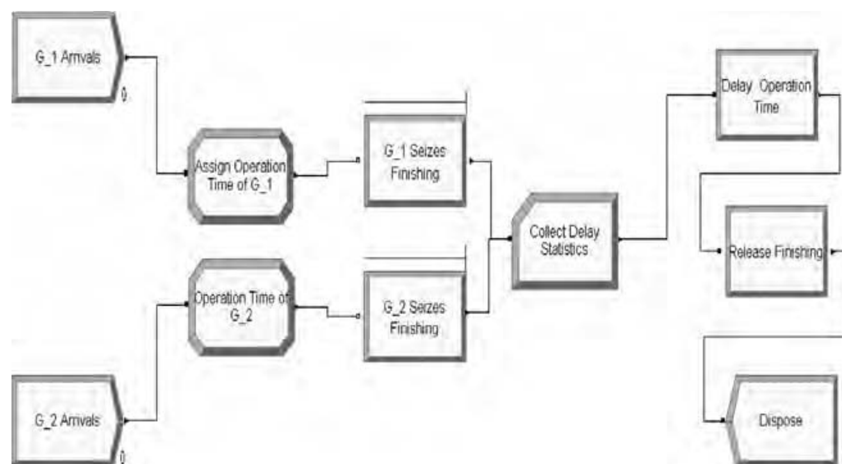


Figure 9.3 Arena model of a finishing operation with two types of parts.

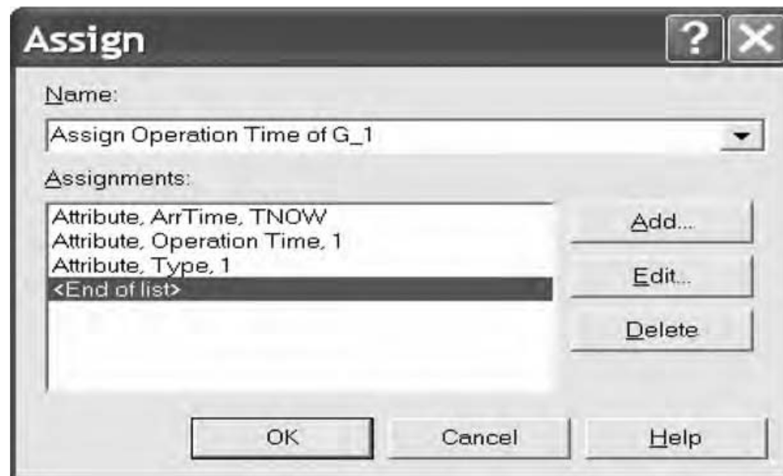


Figure 9.4 Dialog box for the *Assign* module for part *G_1*.

Time attribute. Figure 9.4 displays the dialog box for the *Assign* module *Assign Operation Time of G_1*.

Next, each part entity enters a *Seize* module, called *G_1 Seizes Finishing* or *G_2 Seizes Finishing*, respectively, and attempts to seize the server of the finishing operation; however, if the server is busy, the current arrival is queued (FIFO) at its associated *Seize* module. Figure 9.5 displays the dialog box for the *Seize* module *G_1 Seizes Finishing*, which aims to seize the common resource (machine) called *Finishing Operation*.

Note that the model defines multiple *Seize* modules, all of which aim to seize the same server. Following our naming conventions, the queue in the upper *Seize* module is called *G_1_Q* and the one in the lower *Seize* module is called *G_2_Q*. Eventually, each part entity seizes the common resource *Finishing Operation* from the

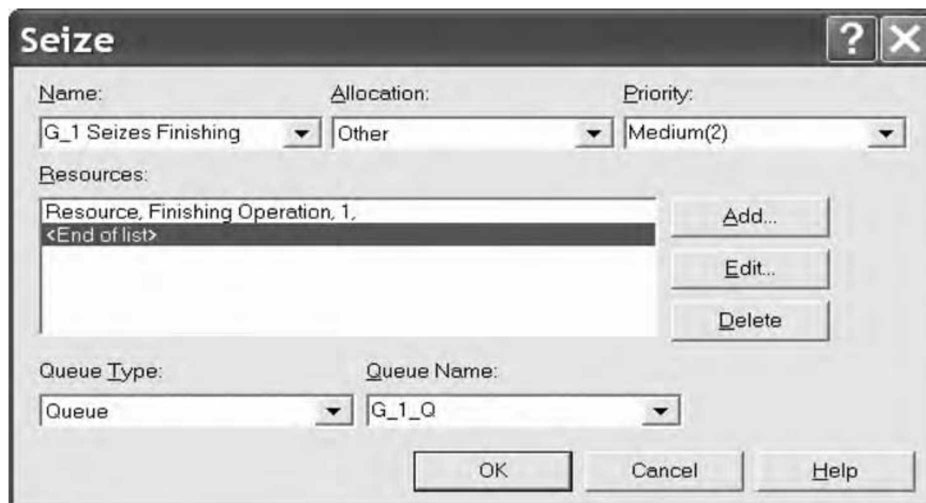


Figure 9.5 Dialog box for the *Seize* module *G_1 Seizes Finishing*.

corresponding *Seize* module. If the common resource is unavailable upon their arrival, then part entities of type G_1 wait in queue *G_1_Q*, while part entities of type G_2 wait in queue *G_2_Q*. Departing part entities of any type enter the *Record* module, called *Collect Delay Statistics*, to tally queue delays (the tallying mechanism will be discussed in the next section). Part entities departing from the *Record* module enter the *Delay* module, called *Delay Operation Time*, where they are detained for the processing time specified in their *Operation Time* attribute. After processing is completed, the part entities proceed to the *Release* module, called *Release Finishing*, to release the resource *Finishing Operation*, and finally enter the *Dispose* module, where they are disposed of.

9.5.2 Observation Collection

Figure 9.6 displays the dialog box of the *Record* module, called *Collect Delay Statistics*, which tallies statistics of type *Time Interval*. Such statistics tally the time difference between the arrival time of an entity at the *Record* module and the time stored in a prescribed attribute of that entity. In our case, the latter is the *ArrTime* attribute, which stores the part entity's arrival time at the finishing operation, so that the *Record* module may tally queue delays for each part type.

Observe that the *Record into Set* checkbox in Figure 9.6 is checked, and that its *Tally Set Name* field specifies a *Set* module name (*Queue Delays*). This has the effect of instructing the *Record* module to produce a separate delay average for each part type, rather than a pooled average across all types. In order to record the corresponding statistics separately, the modeler must create the *Queue Delays* tally set in the *Set* module (from the *Basic Process* template panel), and Arena records the respective statistics into separate set members, based on their *Type* attribute specified in the *Set Index* field of the *Record* module. Figure 9.7 displays the dialog spreadsheets associated with the *Set* module *Queue Delays*.

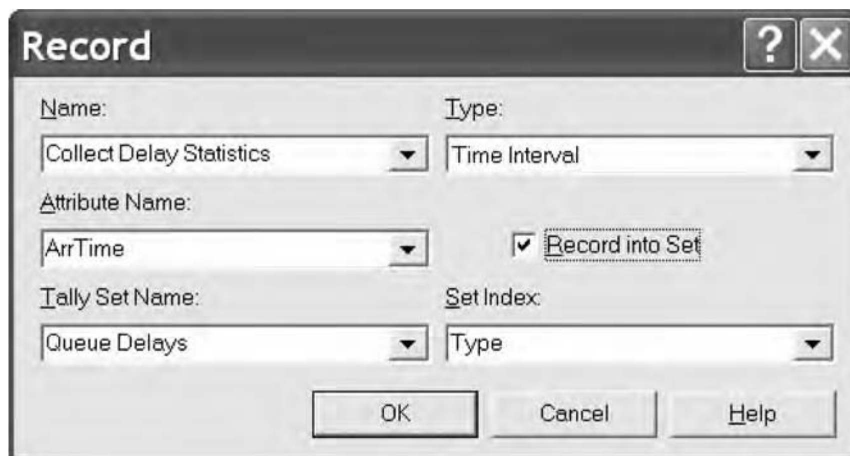


Figure 9.6 Dialog box of the *Record* module *Collect Delay Statistics*.

Set - Basic Process

	Name	Type	Members
1	Queue Delays	Tally	2 rows

Members

	Tally Name
1	G_1 Delay
2	G_2 Delay

Figure 9.7 Dialog spreadsheets for the *Set* module to tally delay times for each part type (left) and its members (right).

Statistic - Advanced Process

	Name	Type	Tally Name	Tally Output File
1	G_1 Delay	Tally	G_1 Delay	G1_Delay
2	G_2 Delay	Tally	G_2 Delay	G2_Delay

Figure 9.8 Dialog spreadsheet of the *Statistic* module with a *Tally* statistic for each part-type delay.

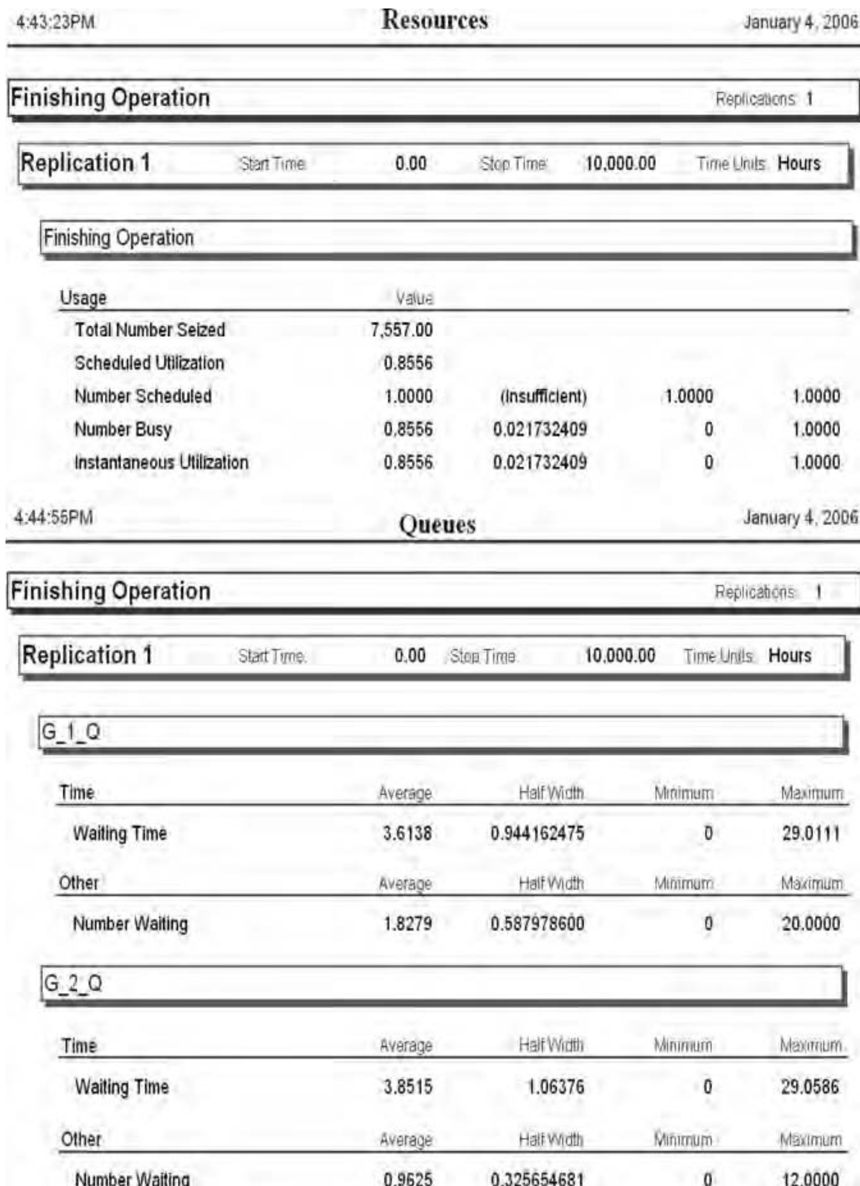
This set was selected as type *Tally* (in a drop-down list in column *Type*), and consists of two members (the button labeled *2 rows* in the *Members* column), one for type G_1 parts and the other for type G_2 parts. We have also declared the two delay *Tally* statistics in the *Statistic* module (even though we did not have to), whose dialog spreadsheet is displayed in Figure 9.8 for model completeness.

9.5.3 Output Summary

A replication of the Arena model for the finishing operation of Figure 9.3 was run for 10,000 hours. Figure 9.9 displays the resulting summary statistics.

An examination of the input data in Section 9.5.1 reveals that the finishing process was busy processing parts of type G_1 with partial utilization $\rho_1 = 1/2 = 0.5$, and parts of type G_2 with partial utilization $\rho_2 = 1.4/4 = 0.35$. Thus, the total utilization of the finishing workstation is $\rho = 0.85$. Therefore, we expect the probability of the finishing machine being in the *Busy* state to be around 0.85. The estimated probability in Figure 9.9 is actually 0.8556 over a replication of length 10,000 hours.

Columns 2, 3, and 4 in Table 9.2 (Finishing Machine *Busy*, G_1 Buffer Delay, and G_2 Buffer Delay) display the behavior of the estimates of three performance measures as functions of increasing replication length. Observe how the estimates appear to converge to respective limiting values as the simulation length increases (convergence is indicated by the fact that the values appear to stabilize and change very little for the higher range of replication lengths). Since we know the true value of machine utilization, we can take advantage of this knowledge in deciding on the *smallest* replication length that gives rise to sufficiently accurate estimates (high accuracy is indicated here by low variability in the estimates as function of replication length). We naturally seek the smallest value, since we would like to reduce the computational effort as much as possible. We point out that an insufficient replication length is characterized by apparent nonconvergent values of the estimates produced. As a general rule, a longer replication



4:43:23PM **Resources** January 4, 2006

Finishing Operation Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 10,000.00 Time Units: Hours

Finishing Operation

Usage	Value			
Total Number Seized	7,557.00			
Scheduled Utilization	0.8556			
Number Scheduled	1.0000	(Insufficient)	1.0000	1.0000
Number Busy	0.8556	0.021732409	0	1.0000
Instantaneous Utilization	0.8556	0.021732409	0	1.0000

4:44:55PM **Queues** January 4, 2006

Finishing Operation Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 10,000.00 Time Units: Hours

G_1_Q

Time	Average	Half Width	Minimum	Maximum
Waiting Time	3.6138	0.944162475	0	29.0111

Other	Average	Half Width	Minimum	Maximum
Number Waiting	1.8279	0.587978600	0	20.0000

G_2_Q

Time	Average	Half Width	Minimum	Maximum
Waiting Time	3.8515	1.06376	0	29.0586

Other	Average	Half Width	Minimum	Maximum
Number Waiting	0.9625	0.325654681	0	12.0000

Figure 9.9 Output statistics for the finishing operation model of Figure 9.3.

has a better chance of experiencing rare events, which may significantly affect the accuracy of the observed statistics.

9.5.4 Statistics Summary: Multiple Replications

Standard Arena reports provide statistics for each replication separately, but no pooled statistics across replications. However, pooled reports can be accessed by the

Table 9.2

Behavior of three performance measures as functions of replication length

Replication Length	Finishing Machine *Busy*	G_1 Buffer Delay	G_2 Buffer Delay
10,000	0.8556	3.6138	3.8515
100,000	0.8453	3.2605	3.3124
1,000,000	0.8504	3.3341	3.3318
10,000,000	0.8502	3.3067	3.3050

Table 9.3

Outputs summary statistics based on 10 replications

Performance Measure	Average Value	Half-Width	Minimum Value	Maximum Value
Machine *Busy*	0.8544	0.0087	0.8315	0.8702
G_1 Buffer Delay	3.3115	0.3206	2.4500	4.1146
G_2 Buffer Delay	3.3463	0.3412	2.4648	4.1028

modeler by first selecting the *Setup...* option in the *Run* menu, and then clicking on the *Reports* tab, and finally selecting the option *SIMAN Output Report(.out file)* in the *Default Report* field.

Table 9.3 illustrates statistics supported by the *SIMAN Output Report*, corresponding to three performance aspects (machine utilization and part delays by type), based on 10 replications of length 10,000 hours each. In particular, it displays the half-width of 95% confidence intervals (at significance level $\alpha = 0.05$) for all performance measures for which statistics were requested in an *Outputs* report. Normally, these are constructed from a single replication using the batch means method (see Section 9.2.2). If, however, more than one replication is run, then Arena's standard output provides these same statistics for each replication separately. Additionally, an *Outputs* report would provide the corresponding *pooled* summary statistics based on all replications, as shown in Table 9.3, including half-widths of 95% confidence intervals, as described in Section 9.4. Note that the confidence interval estimates in Table 9.3 are tighter (for the same level of confidence) than those of Figure 9.9. This should hardly be surprising, since the former were based on considerably more information than the latter, that is, 10 replications in Table 9.3 compared to a single replication in Figure 9.9.

9.6 OUTPUT ANALYSIS VIA THE ARENA *OUTPUT ANALYZER*

The Arena *Output Analyzer* is a tool that supports statistical analysis of replication output data (output analysis). Such data are collected and stored in data files during a simulation run in accordance with any statistical element defined in the *Statistic* module or in *Record* modules (see Sections 5.4 and 5.5). The *Output Analyzer* then provides

options to manipulate, analyze, and display the data and related statistics. The options include the following:

- *Data transfer* (e.g., import/export of ASCII files)
- *Statistical analysis* (batching, correlogram, point estimation, and confidence interval estimation for means and standard deviations, and statistical tests for comparing parameters of different samples)
- *Graphing* data and statistics (plots and charts)

The reader is reminded at this juncture that the working example of Section 9.5 will also be used throughout this section.

9.6.1 Data Collection

During each replication, observations are collected and written continually to data files. This mechanism preserves a record of the observations for later inspection or for statistical analysis. For instance, file *G_1_Delay* contains buffer delays for parts of type G_1. Using the *Output Analyzer*, the analyst can obtain a human-readable version of the file by using the *Export* option from the *Data File* item of the *File* pull-down menu.

As an example, consider Table 9.4, which displays buffer delays of type G_1 parts, recorded during the first 50 hours of a replication.

Table 9.4

Tallied buffer delay observations for parts of type G_1

Simulation Collection Time	Observed Time in Queue G_1_Q Time
0.00000000e + 000	0.00000000e + 000
2.40000000e + 000	1.70769546e + 000
3.40000000e + 000	2.16722249e + 000
4.40000000e + 000	3.04687979e + 000
5.40000000e + 000	3.76552538e + 000
7.80000000e + 000	5.91574045e + 000
8.80000000e + 000	5.33219193e + 000
9.80000000e + 000	2.49312870e + 000
1.09922681e + 001	0.00000000e + 000
1.19922681e + 001	7.17802770e − 001
1.47079443e + 001	0.00000000e + 000
1.57079443e + 001	7.57823257e − 001
1.67079443e + 001	4.95948674e − 001
1.77505471e + 001	0.00000000e + 000
1.87505471e + 001	2.95582065e − 001
1.97505471e + 001	8.78091690e − 001
2.07505471e + 001	1.47883109e + 000
2.70131256e + 001	4.35622213e − 001
3.09594790e + 001	0.00000000e + 000
4.28340274e + 001	0.00000000e + 000
4.55615801e + 001	0.00000000e + 000
4.65615801e + 001	4.35670033e − 001
−1.00000000e + 000	−1.00000000e + 000

The exported data are arranged in two columns, so that each row entry consists of a pair of values. The first value (column 1) is the simulation time of the collected observation (time stamp), while the second value (column 2) is the observed value at that time. Note that time −1.0 in the last line of Table 9.4 stands for EOF (end of file). In terms of Arena variables, every part departure immediately triggers the recording of the corresponding pair of values as a row in the output file.

9.6.2 Graphical Statistics

Figure 9.10 displays statistics of type G_1 parts. This display consists of four panels, each depicting graphical statistics for buffer delays of type G_1 parts, collected over 1000 simulation hours. Starting at the upper left corner and proceeding clockwise, the graphs display the following:

- A plot (continuous-curve graph) of the sequence of delay times recorded
- A bar chart of the same
- A histogram of the collected delay times and its cdf
- The moving average of the collected delay times (legend label *Smoothed*), superimposed on the actual delay times (legend label *Raw Data*)

An examination of the histogram panel reveals some probability mass in the histogram tail. This is due to the bursty nature of Poisson arrivals: When a burst of parts arrives (this can be visualized as a cluster of arrivals on the timeline), then burst

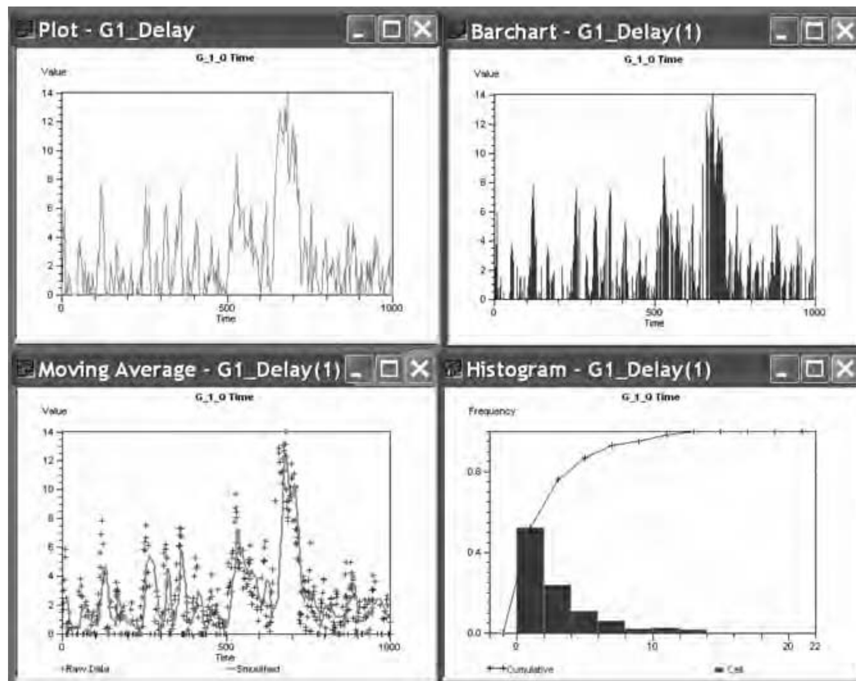


Figure 9.10 Graphical statistics for parts of type G_1.

members at its tail end incur higher-than-average delay times. These unusually long delays leave their mark in all four panels. For example, the plot, bar chart, and moving average panels exhibit sharp upward fluctuations corresponding to burst arrivals, while the histogram panel quantifies this effect in the shape of its tail.

9.6.3 Batching Data for Independent Observations

The *Output Analyzer* provides a number of statistical analysis tools as part of the *Analyze* pull-down menu. For example, using the *Batch/Truncate Obs'ns* option in the *Analyze* menu, the user can batch observations in the style of the batch means method to estimate confidence intervals without relying on the Arena standard output. As usual, batching implementation depends on the type of statistics observed: discrete sample (observation based) or continuous sample (time based) as described in Section 9.2.2. For instance, suppose we wish to construct a 95% confidence interval for the mean buffer delay of type G_1 parts using 500,246 observations over a replication length of 1,000,000 hours. Figure 9.11 displays the requisite *Batch/Truncate* dialog box for observations grouped into batches of size 1000.

For every batching action, the Arena *Output Analyzer* produces a summary report that displays the batching plan parameters and the resultant estimated batch covariance. In particular, Panel 9.1 displays the batching results specified in Figure 9.11.

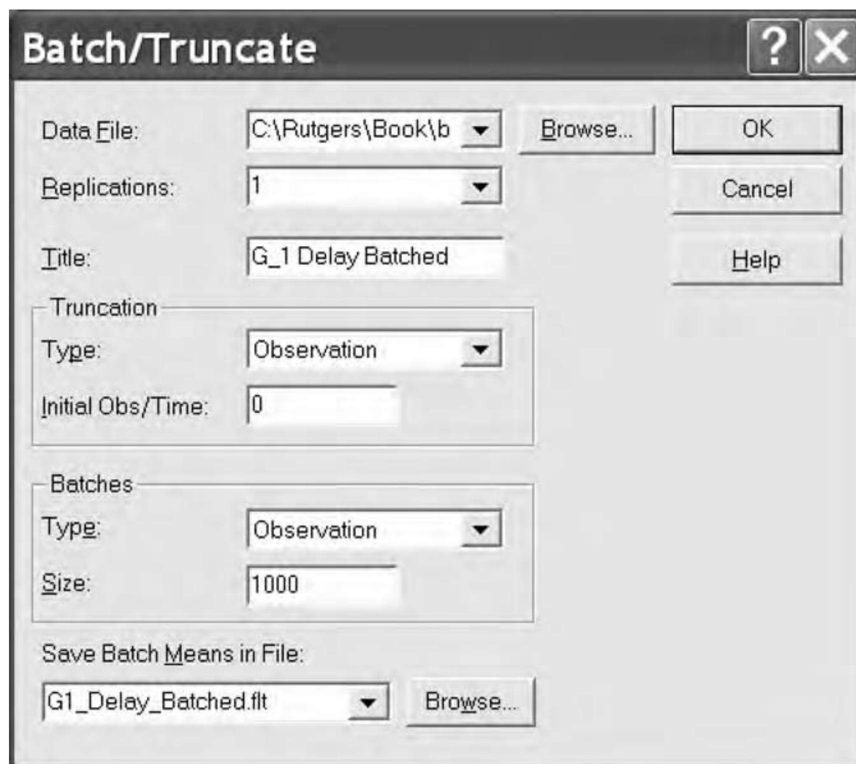


Figure 9.11 *Batch/Truncate* dialog box for delay times of G_1 type parts.

```
Batch/Truncate Summary
G_1 Delay Batched
    Batched observations stored in file :     G1_Delay_Batched.flt
        Initial Observations Truncated :     0
                     Number of Batches :     500
       Number of Observations Per Batch :     1000
    Number of Trailing Obs'ns Truncated :     246
 Estimate of Covariance Between Batches :     -0.03458
```

Panel 9.1 *Batch/Truncate* summary report for delay times of G_1 type parts.

The report in Panel 9.1 shows that we chose *not* to truncate any observations from the transient period. All in all, 500 batches of 1000 observations each were generated with 246 trailing observations (for the last incomplete batch). As a check on whether a batch size of 1000 suffices to yield approximately statistically independent batches, the report also provides an estimate of the covariance between batches. (Recall from Section 3.6 that covariance or correlation values close to 0 suggest that batches are uncorrelated, and consequently, approximately independent. The lack of correlation suggests, but does not imply, statistical independence.) Arena actually performs a statistical test (see Section 3.10) to determine whether the null hypothesis that the covariance is 0 can be rejected at a given significance level α. In our case, the covariance estimate is -0.03458, which is sufficiently close to zero to conclude that the null hypothesis cannot be rejected at significance level $\alpha = 0.05$. If, however, Arena found that the null hypothesis must be rejected, it would then issue a message such as

Covariance equal to 0 *rejected in favor of Covariance* > 0 *at* 0.05 *level*.

Such a message indicates that the batching plan should be modified. Generally, increasing the batch size should reduce the absolute value of batch covariance (or correlation).

9.6.4 Confidence Intervals for Means and Variances

Having verified that the batching plan appears statistically valid, we may now proceed to estimate the requisite confidence interval using the estimates obtained. To this end, the analyst may use the *Conf. Interval On Mean* option in the *Analyze* menu, and then select among two methods for computing confidence intervals for means: *Classical* or *Standardized Time Series*...

Figure 9.12 displays 95% classical confidence intervals for mean buffer delays for parts of type G_1 and type G_2, separately. Observe that these confidence intervals are considerably tighter than those in Table 9.3. Again, if more than one replication is available, then the user may elect to pool all the mean buffer delay estimates into a single sample, and use it to compute generally tighter confidence intervals. This can be achieved by selecting the *Lumped* option in the *Replications* field of the *Data File* dialog box.

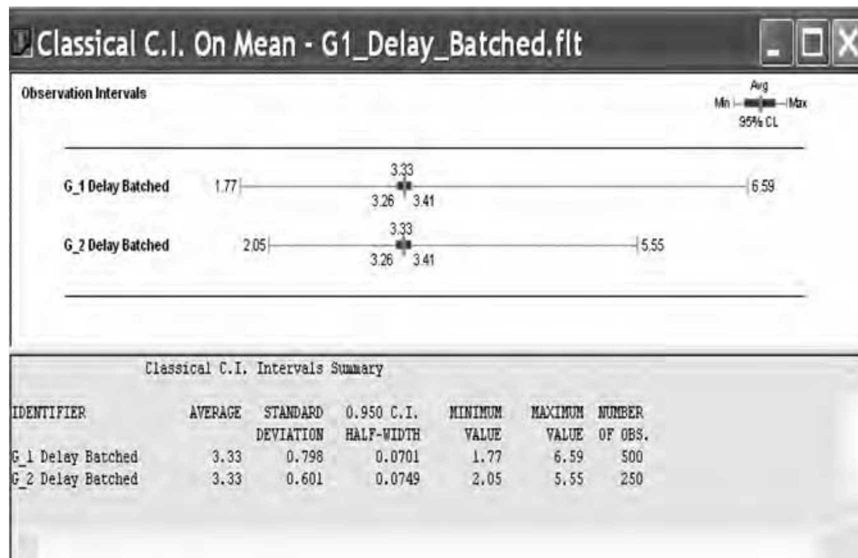


Figure 9.12 Confidence intervals for mean delays of parts of type G_1 and type G_2.

In a similar vein, the analyst can produce confidence intervals for the standard deviation, using the *Conf. Interval on Std Dev...* option in the *Analyze* menu.

9.6.5 Comparing Means and Variances

The *Analyze* menu also provides the options *Compare Means* and *Compare Variances* for comparing the means and variances, respectively, of two samples drawn from two populations, by testing them statistically for equality. For example, to test the null hypothesis that the means are equal, Arena sets up a confidence interval for their difference. Since under the null hypothesis the difference is zero, one accepts equality if the confidence interval includes 0, and rejects it, otherwise.

Figure 9.13 depicts the dialog boxes for comparing the mean delays of the two part types, G_1 and G_2, from their data files of observations. The test results are shown in Figure 9.14 via a confidence interval for the difference of the respective mean delays, at significance level $\alpha = 0.05$.

An examination of the graphic at the top as well as the numerical information at the bottom of Figure 9.14 reveals that the confidence interval for the difference does indeed include 0. Consequently, the null hypothesis of means' equality cannot be rejected at that significance level. This result is in line with expectations stemming from theoretical considerations. Indeed, queueing theory tells us that in a queueing system with multiple equal-priority job classes, a FIFO discipline, and a single server, the mean delay time of all job classes is the same. Of course, the average numbers of type G_1 and G_2 parts in the buffer are different due to their disparate arrival and service processes.

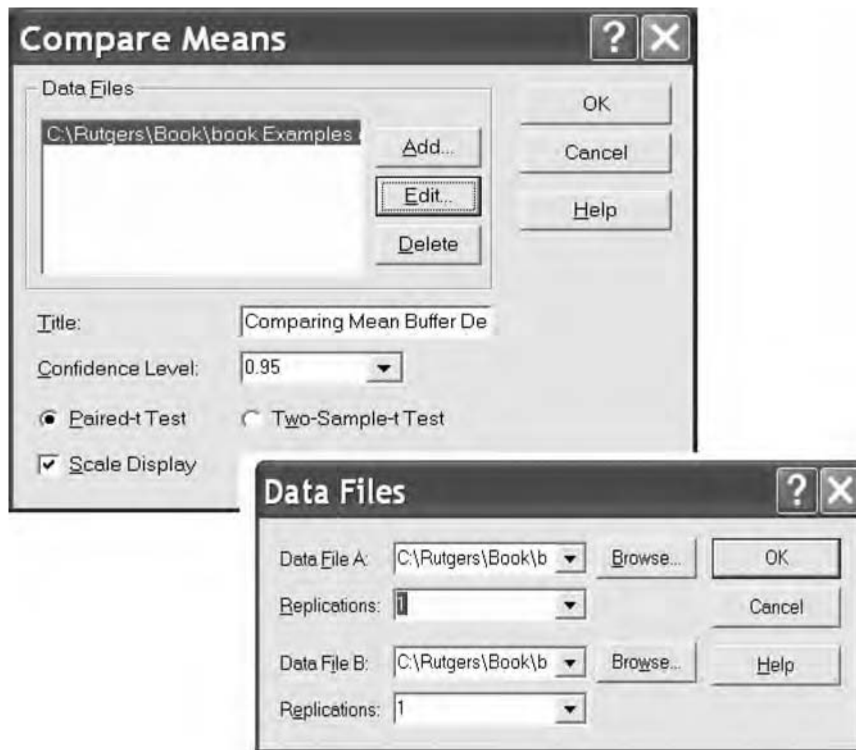


Figure 9.13 Dialog boxes for comparing mean buffer delays of type G_1 and type G_2 parts.

In a similar vein, the variances of two samples drawn from two populations can also be compared for equality, using the *Compare Variances* option of the *Analyze* menu. Again, Arena performs a statistical test of the null hypothesis that the true variances are equal by constructing a confidence interval, this time for the ratio of the sample

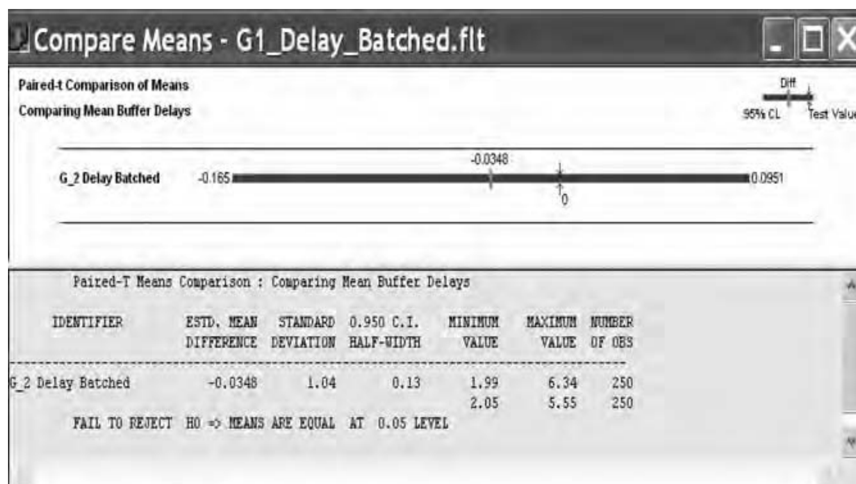


Figure 9.14 Test results for the equality of mean buffer delays of type G_1 and type G_2 parts.

variances of the two samples. Since under the null hypothesis the ratio is 1, one accepts the null hypothesis if the confidence interval includes 1, and rejects it, otherwise.

9.6.6 Point Estimates for Correlations

To gauge the statistical dependence among observations within a sample, the analyst can make use of the *Correlogram* option in the *Analyze* menu. This option computes the sample autocorrelation function (see Section 3.9), and displays the numerical values of the autocorrelation point estimates, as well as their graph, as function of the lag. (Recall that the value of a correlation coefficient ranges between -1 and $+1$.)

Figure 9.15 displays two correlograms for delays of type G_1 parts, based on the same underlying sample. The left-side correlogram was computed from simulation observations of successive delays. The correlogram at the right was computed from the sample means of batches of 1000 delays. The two correlograms are strikingly different: The successive delays are *strongly positively correlated*, while the sample means of the batches are *very nearly uncorrelated*. This example illustrates how a good batching plan can largely eliminate correlations among sample mean estimates, resulting in confidence intervals for the sample parameters that legitimately invoke the central limit theorem (see Section 3.8.5).

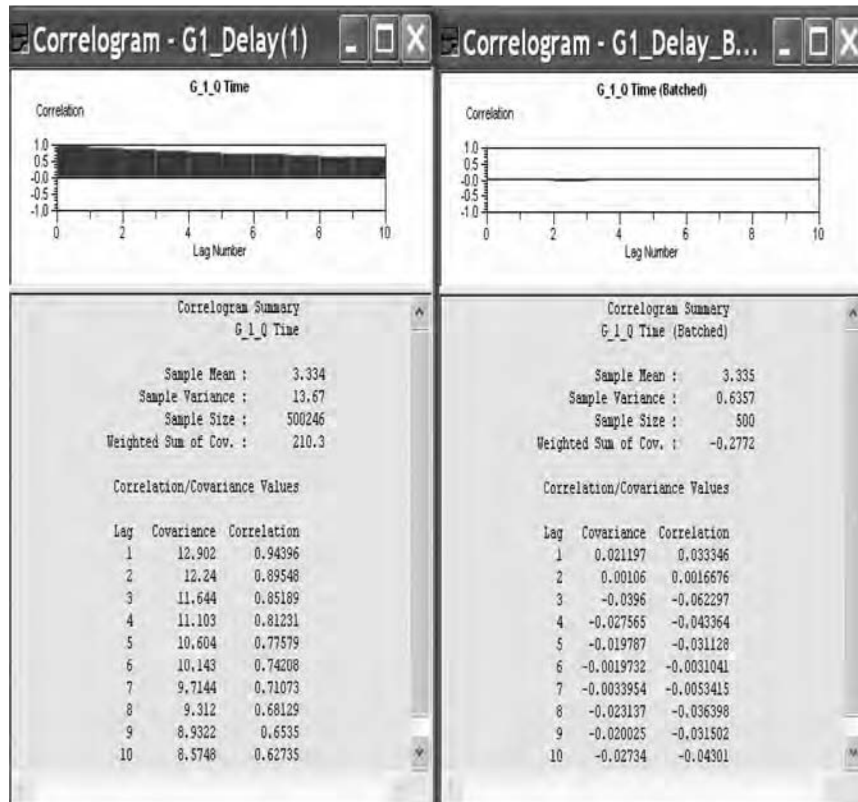


Figure 9.15 Results of correlation analysis for type G_1 parts.

9.7 PARAMETRIC ANALYSIS VIA THE ARENA *PROCESS ANALYZER*

The term *parametric analysis* refers to the activities of running a model multiple times with a different set of input parameters for each run, and then comparing the resultant performance measures. The purpose of parametric analysis is to understand the impact of parameter changes on system behavior (sensitivity analysis), often in the process of seeking the optimal configuration (parameter set) with respect to one or more performance measures or combination thereof.

In Arena parlance, input parameters are called *controls*, and the resultant performance measures are called *responses*. A collection of controls and responses for a given set of runs is referred to as a *scenario*, and a collection of scenarios is termed an Arena *project*.

The Arena *Process Analyzer* is a tool that supports parametric analysis of Arena models by allowing the modeler to create, run, and compare simulated scenarios, and thus observe the effect of prescribed controls on prescribed responses. Controls may consist of variables and resource capacities, while responses include both variables and statistics (note that only variables can serve both as controls and responses). User-defined categories of controls and responses are also allowed by using modules from the *Elements* template panel, such as *Resources*, *Variables*, *Tallies*, *Dstats*, *Counters*, and *Outputs*.

The *Process Analyzer* is accessible via the Windows *Programs* option or from the *Tools* menu in the Arena home screen. Once an Arena model is created, debugged, and validated, selecting the *Process Analyzer* option pops up a separate *Process Analyzer* window for each Arena project. To create scenarios, the modeler selects the *New* option in the *Process Analyzer File* menu to pop up a grid (table) panel for specifying Arena scenarios and their constituent controls and responses—one scenario per grid row.

Figure 9.16 displays a *Process Analyzer* window with an initial grid (identified by the heading *Scenario Properties*) and a panel to display charts below it. A new scenario specification is added either by selecting the *Scenario*. . . option in the *Insert* menu or by double-clicking below the grid as indicated by the Arena message. Each scenario specification consists of the following columns:

1. The *S* column is check-marked by Arena when a scenario is valid, and flagged when the scenario run is completed.
2. The *Name* field assigns a user-defined name to the current scenario.
3. The *Program File* field specifies a file name (*.pan* file), in which Arena saves user-defined scenarios. This file is created by Arena automatically whenever the underlying model is checked for correctness via the *Check Model* option in the Arena *Run* menu. The generated *.pan* file is placed in the same directory as the underlying *.doe* model file.
4. The *Reps* field specifies the number of replications to be run for the current scenario.
5. Additional columns for specifying controls and responses may be added to the grid by selecting the *Control*. . . or *Response* . . . option, respectively, in the *Insert* menu. These are inserted to the right of the existing columns in the order of creation. Arena then opens a dialog box with a tree of available controls or responses, and the modeler navigates the tree and selects the desired option.

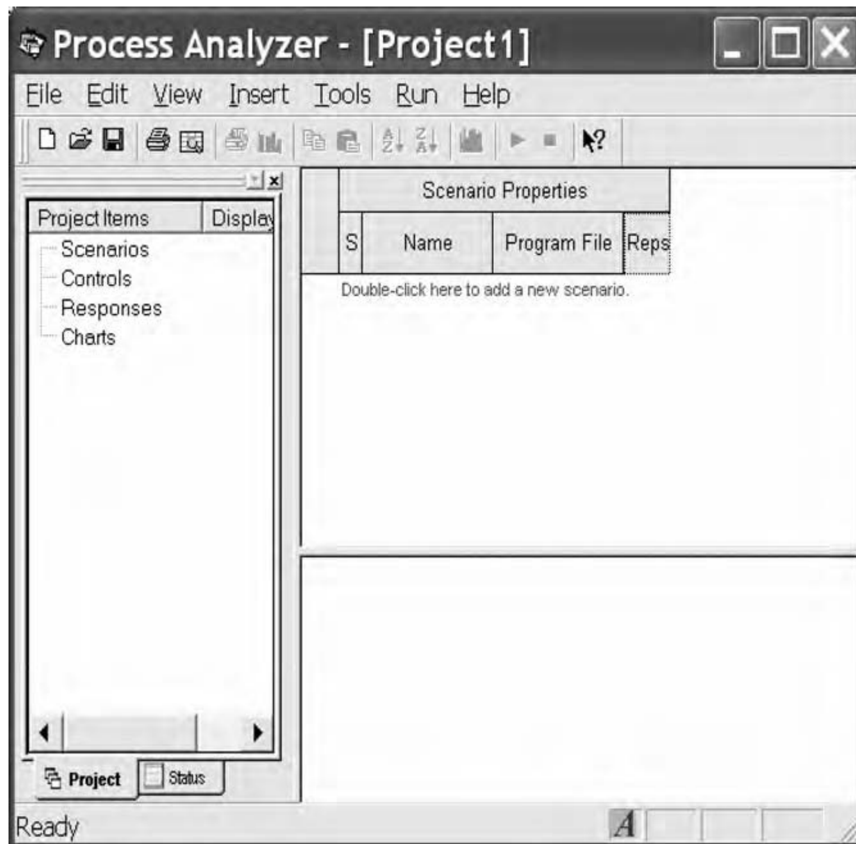


Figure 9.16 A *Process Analyzer* window with an initial grid.

To the left of the grid panel is the *Project* bar, used to display project information. The *Project* bar has two tabs at the bottom: the *Project* tab displays information on grid rows (scenarios), grid columns (controls and responses), and graphical charts, while the *Status* tab displays information on the current scenario being run.

Once the grid data are specified, the modeler may proceed to run the scenarios in the current *Process Analyzer* project. Individual scenarios can be run one by one by highlighting each requisite scenario in the grid, and then selecting the *Go* option in the *Run* menu. To run multiple scenarios, the modeler simply highlights the requisite set of scenarios (using the Ctrl key), and then launches the run. The scenarios will be run one after the other without user intervention. All *Process Analyzer* replications run in batch mode (without animation). As each scenario run is completed, the corresponding response values appear in the grid under the *Response* columns, replacing the initial dashes there. The resulting grid is handy for output analysis comparisons of project scenarios. Once a scenario run terminates, the modeler may visualize response statistics by creating charts. Charts can be defined using the *Chart*... option of the *Insert* menu.

The *Process Analyzer* facilitates sensitivity analysis, especially in complex systems where there may be many controls, responses, replications, and scenarios that have long execution times. As an example of a typical use of the *Process Analyzer*, consider a

single-machine workstation with random job arrivals and a finite buffer capacity. Jobs arriving at a full system are considered lost. We wish to perform a parametric analysis to understand the impact of the buffer capacity on system performance. To this end, we select the buffer capacity to be a control, and let the corresponding responses be buffer-related statistics, including the probability that the buffer is full, average WIP level, average delay in the buffer, as well as the system throughput.

The parametric analysis plan is as follows. In each successive scenario, the buffer capacity is increased by unity. In consequence, we expect the probability that the buffer is full to decrease as the buffer size increases, so that more jobs can enter the system. Therefore, the average WIP level and the throughput should increase. Figure 9.17 depicts a *Process Analyzer* window with a populated grid. The *Project* bar shows that five scenarios have been defined: one control (buffer capacity) and four responses (the statistics mentioned above). However, the X mark overlaying the spectacles icons under the *Project Items* column and the keyword *Hidden* under the *Display* column indicate that some grid rows and columns are defined but not displayed. More precisely, the fifth scenario and the second response are hidden in Figure 9.17, as evidenced by the absence of corresponding responses in the grid (hidden scenarios are not run). The modeler may hide and redisplay any scenario, control, or response by clicking the corresponding spectacles icon. An examination of the displayed responses confirms the expected behavior.

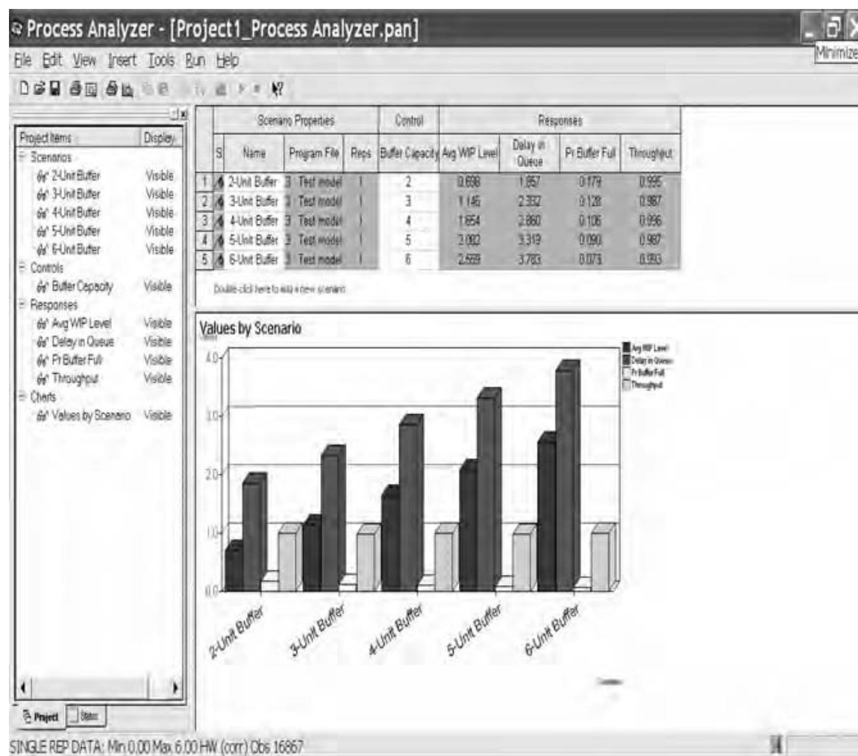


Figure 9.17 A *Process Analyzer* window with a populated grid.

EXERCISES

1. Revisit Exercise 2 in Chapter 6 (*Message queue in a database server*). Recall that the message queue has a finite capacity of 64 kb. Query interarrival times are iid with an exponential distribution of mean 2.5 seconds. Message sizes (in kilobytes) are distributed iid Tria(4, 12, 16). An arriving message will be lost if it cannot fit into the remaining capacity of the message queue. The queries are served on a *smaller-size-first* basis (which may not be too realistic, since long messages then tend to wait longer in the queue). The service time is proportional to message size, where each 4 kb portion of the message contributes 1 second to the service time.

   Performance measures of interest follow:

   - Average number of messages in the queue
   - Average delay per query in the queue
   - Utilization of the database server
   - Loss probability for a message upon arrival

   Perform the following tasks:

   a. Construct a revised Arena model, run it for 1000 seconds, and plot $\bar{W}_q$ and $\bar{N}_q$ over time.
   b. Identify a point in time when the system appears to settle into steady state.
   c. Run the model with stops at 1,000, 10,000, 100,000, and 1,000,000 seconds, and show the converging behavior of $\bar{W}_q$ and $\bar{N}_q$ as well as their relationship.
   d. Estimate the 95% confidence intervals for $\bar{W}_q$ and $\bar{N}_q$ produced by a run of 10,000 seconds. Use the Arena *Output Analyzer* and invoke the batch means method manually with your choice of batch size.
   e. Run 10 replications of 10,000 seconds each, and estimate the 95% confidence intervals for $\bar{W}_q$ and $\bar{N}_q$ across replications. Use the Arena *OUTPUTS* element (from the *Elements* template panel) and the SIMAN *Summary Report* (*.out file*) option in the *Default Report* field of the *Run Setup* dialog box (which pops up on selecting the *Setup*... option in the *Run* menu).
   f. Use the Arena *Process Analyzer* to study the impact of queue capacity on the loss probability by stepping the capacity from 16 kb to 128 kb in increments of 16 kb.

2. Revisit Exercise 1 in Chapter 6 (*Production line*). Recall that the line consists of two workstations in series: the first performs a *filling process* and the second performs a *capping process*. Both workstations have unlimited buffer space. A job in the real-life system is a batch of 10 containers to be filled and capped. However, to simplify modeling, here each batch will be treated as a single entity. Job interarrival times at the first workstation are iid exponentially distributed with mean 3.125 minutes. Upon completion of the filling process, the job joins the buffer of the capping process. Job filling-time distributions are iid Tria(1, 3, 5) minutes, while capping takes a fixed time of 3 minutes. Jobs depart from the system after the capping process is complete.

   Simulate the system for 10,000 minutes, and estimate the following statistics:

   a. Average job delay in each workstation buffer.
   b. Utilization of each workstation.
   c. Using the Arena *Output Analyzer* and the batch means method, estimate from a single replication the 95% confidence interval for the average job system time.

d. Compute a correlogram for the waiting times in the buffer of the first workstation for the individual waiting times.
e. Compute a correlogram for the waiting times in the buffer of the first workstation for the batched waiting times (choose an appropriate batch size).
f. Use the Arena *Process Analyzer* to study the impact of capping time on the mean job flow time by stepping the capping time from 1 minutes to 3 minutes in increments of 0.5 minutes.

3. The First New Brunswick Savings (FNBS) bank opens at 8 AM and closes at 4 PM on weekdays. Customers arrive at the bank according to iid exponentially distributed interarrival times with the following time-dependent rates:
   - 20 per hour between 8 A.M and 11 A.M
   - 35 per hour between 11 A.M and 1 P.M
   - 25 per hour between 1 P.M and 4 P.M

   Customers come to the bank for various services. Data collection shows that 50% of the customers come to withdraw money, 30% come to deposit money, and 20% come for other bank services. Histograms of the empirical data suggest the following service-dependent distributions of customer service times:
   - Money withdrawal: Unif(3, 5) minutes
   - Money deposit: Unif(4, 6)
   - Other services: Tria(5, 13, 18) minutes

   All service times are mutually independent and iid within each service. There are two tellers serving a single queue for both withdrawal and deposit services, and two bank officers serving another single queue for all other services.
   a. Develop an Arena model for the FNBS bank, run it for 1 day, and estimate customer mean waiting times. Note that this is a terminating simulation.
   b. Run 10 replications of the model, and obtain point estimates and 95% confidence intervals (manually) for the waiting times in each of the two queues. Repeat the estimation procedure using 20 replications, and compare the results to those of 10 replications.
   c. Run 10 replications of the model, and use the Arena *OUTPUTS* element (from the *Elements* template panel) to obtain a 95% confidence interval across replications. For this, you should use the SIMAN *Summary Report* (*.out file*) option in the *Default Report* field of the *Run Setup* dialog box (which pops up on selecting the *Setup…* option in the *Run* menu).
   d. Redo part b, using the Arena *Output Analyzer* to construct 95% confidence intervals for the waiting times in the two queues.
   e. Using the Arena *Output Analyzer*, test the hypothesis that the waiting times in the two queues have the same means.
   f. Use the Arena *Process Analyzer* to map the replicated mean waiting times and corresponding confidence intervals from part b to a graph of the 10 replications.

Chapter 10

Correlation Analysis

Correlation analysis is a modeling and analysis approach that straddles input analysis and output analysis. It consists of two activities:

- *Modeling correlated stochastic processes*. Generally, both autocorrelations within individual time series and cross-correlations across multiple time series are of interest (see Section 3.6). However, modeling autocorrelations preponderates.
- *Studying the impact of correlations on performance measures of interest via sensitivity analysis*. In such studies, the modeler may "increase" or "decrease" the magnitude of correlations in various correlated processes, and then observe the resulting performance measures (see Section 9.7). If the impact is marginal, modeling correlations can be dispensed with.

Thus, correlation analysis combines modeling (the input analysis of the first item) with analysis (the output analysis in the second item), especially in the context of simulation. As discussed later, correlation analysis is motivated by the impact of correlations on performance measures and the fact that ignoring them can lead to large errors in predicting system performance. Applications to manufacturing systems will be discussed later.

10.1 CORRELATION IN INPUT ANALYSIS

Correlation analysis as part of input analysis is simply an approach to modeling and data fitting. This approach insists on high-quality models incorporating temporal dependence, and strives to fit correlation-related statistics in a systematic way. To set the scene, consider a stationary time series $\{X_n\}_{n=0}^{\infty}$, that is, a time series in which all statistics remain unchanged under the passage of time. In particular, all X_n share a common mean, m_X, and common variance, σ_X^2. To fix the ideas, suppose that $\{X_n\}$ is to be used to model interarrival times at a queue (in which case the time series is non-negative). What characteristics of $\{X_n\}$ should be carefully modeled? We refer to a collection of such characteristics (both in a stochastic model as well as their counterparts in the empirical data that gave rise to the model) as a *signature*. As signatures are often just a set of statistical parameters (means, variances, and so on), the terms *signature* and

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

statistical signature will be used interchangeably. We can often (but not always) judge one signature to be stronger than another. Clearly, signatures become stronger under *inclusion*; for example, the statistical signature consisting of the mean *and* variance is obviously stronger than the signature consisting of the mean alone. Of course, in practice, the true statistical parameters in a signature are typically unknown; in this case, the modeler estimates the signature from empirical data (if available), or absent such data, the modeler may elect to go out on a limb and make an educated guess. In any event, it should be clear that a model fitted to a "strong" statistical signature should have a higher predictive power than one fitted to a "weaker" one.

The foregoing discussion suggests that the modeler should generally strive to fit a model to a strong statistical signature; more accurately, the modeler should strive to fit a model to as strong a signature as is practically feasible. On the other hand, the modeler should be careful not to "over-fit," in the sense that the resulting model contains too many parameters as compared to the data available. Note, however, that model "over-fitting" is relatively rare as compared to model "under-fitting," that is, producing a model by fitting an unduly weak signature. Ultimately, the model produced is a trade-off, incorporating multiple factors, such as the availability of sufficient empirical data, the goal of the modeling project, and perforce, the modeler's knowledge and modeling skill.

The purpose of this section is to encourage the reader to generally strive to fit a strong signature, which includes both the marginal distribution and the autocorrelation function (a statistical *proxy* for temporal dependence). To this end, we will consider a particular class of models, called *TES* models, that has favorable modeling properties. TES modeling will be described in the next section.

To clarify the strong signature recommended, we return to the time series of interarrival times, $\{X_n\}$, and assume that we have at our disposal a considerable number of empirical observations, $\hat{Y} = \{\hat{y}_n\}_{n=0}^{N}$ (we use the caret symbol to indicate empirical observations or statistics formed from them as estimates, in counterdistinction to theoretical constructs). Consider the following set of statistical signatures in ascending strength:

1. The mean, m_X, of the interarrival distribution. This is a "minimal" signature, since its reciprocal, $1/m_X$ is the arrival rate—a key statistic in queueing models.
2. The mean, m_X, and the variance, σ_X^2, of the interarrival distribution. The scatter information embedded in the variance can affect queueing statistics, when supplementing mean information.
3. Additional moments of the interarrival distribution, such as skewness and kurtosis (see Section 3.5).
4. The (marginal) distribution, F_X, of the interarrival distribution. Since a distribution determines all its moments, this signature subsumes all the above. In practice, we estimate F_X via an empirical histogram, $\hat{H}$, constructed from the empirical data, $\hat{Y}$.
5. The distribution, F_X, and the autocorrelation function, $\rho_X(\tau)$, of the interarrival time series. In practice, we estimate $\rho_X(\tau)$ by some estimate $\hat{\rho}_X(\tau)$, $\tau = 1, \ldots, T$, where $T < N$.

Clearly the last signature is stronger than any of the preceding ones. This signature is the focus of this section, and as such merits further elucidation.

The marginal distribution, F_X, is a first-order statistic of $\{X_n\}$. This means that it involves only a single random variable. It is important to realize that fitting a distribution is by itself a strong signature, since this would automatically fit *all* its moments. Recall that the autocorrelation function of a stationary time series $\{X_n\}$ with a finite variance is given by

$$\rho_X(\tau) = \frac{\mathrm{E}[X_n X_{n+\tau}] - m_X^2}{\sigma_X^2}, \quad \tau = 1, 2, \ldots . \tag{10.1}$$

The autocorrelation function, $\rho_X(\tau)$, is a second-order statistic, because its definition requires pairs of *lagged* random variables in $\{X_n\}$ (in addition to the first-order statistics of mean and variance). More importantly, the autocorrelation function is a convenient (partial) descriptor of *temporal dependence* within $\{X_n\}$, that is, it provides information on the "behavior" of random variables in $\{X_n\}$ that are separated by τ lags (time-index units). More specifically, $\rho_X(\tau)$ is just the correlation coefficient of the lagged random variables, $\{X_n\}$ and $\{X_{n+\tau}\}$ (recall that by stationarity, this value is the same for all n), and as such it provides a measure of the *linear dependence* prevailing among such lagged random variables. Simply put, it tells us to what extent the two are likely to behave as follows (see Section 3.6):

- Vary in the same direction, that is, if one random variable is large (small), to what extent is the other likely to be large (small) too on a scale of 0 to 1?
- Vary in opposite directions, that is, if one random variable is large (small), to what extent is the other likely to be small (large) too on a scale of -1 to 0?

This covariation information is an important aspect of temporal dependence, but by no means characterizes it. In fact, temporal dependence is fully characterized by all joint probabilities of random variables. Unfortunately, each such probability introduces a separate parameter into a prospective model, and fitting all of them is not a practical proposition. The autocorrelation function, in contrast, is a much cruder measure of temporal dependence (the statistic $\mathrm{E}[X_n X_{n+\tau}]$ can be deduced from the joint probabilities, but not vice versa), which focuses on linear dependence. However, it captures an important aspect of temporal dependence and introduces comparatively few parameters (its values for each lag considered), so that it qualifies as an effective *statistical proxy* for temporal dependence.

10.2 CORRELATION IN OUTPUT ANALYSIS

To motivate the need for modeling correlations, we shall demonstrate that autocorrelation can have a major impact on performance measures, and so cannot always be ignored merely for the sake of simplifying models. A simple example will illustrate how autocorrelation is germane to queueing statistics (applications to manufacturing systems will be discussed later). The example compares two related models, such that autocorrelation is present in one, but not in the other.

Consider a simple workstation (similar to an *M/M/*1 queue) with job arrival rate λ and processing (service) rate μ ($\mu > \lambda$), so that the system is stable with utilization $\mu = \lambda/\mu < 1$. It is known that in this simple system the steady-state mean sojourn time is $\mathrm{E}[S_{M/M/1}] = 1/(\mu - \lambda)$ (see Kleinrock [1975], chapter 3), and therefore the mean

waiting time in the buffer is $E[W_{M/M/1}] = 1/(\mu - \lambda) - 1/\mu$. Since all job arrivals and processing times are mutually independent, all corresponding autocorrelations and cross-correlations are identically zero.

Next, modify the job arrival process to convert the workstation to a *TES/M/*1 type of system (see Melamed [1991, 1993], Jagerman and Melamed [1992a, 1992b, 1994]). That is, we replace the Poisson arrival process by a so-called (autocorrelated) *TES process* (see Section 10.3 for a description). The merit of TES processes is that they simultaneously admit *arbitrary* marginal distributions and a variety of autocorrelation functions. In particular, we can select TES interarrival processes with the same interarrival time distribution as in the Poisson process (i.e., exponential with rate λ), but with *autocorrelated* interarrival times.

For a given TES process, let $E[W_{TES/M/1}]$ denote the corresponding steady-state mean waiting time in the buffer. To gauge the impact of autocorrelations in the job arrival stream on mean waiting times, we observe the *relative deviation*

$$d(\rho(1)) = \frac{E[W_{TES/M/1}] - E[W_{M/M/1}]}{E[W_{M/M/1}]} \tag{10.2}$$

as function of the lag-1 autocorrelation, $\rho(1)$, of the TES job arrival process. These relative deviations provide a measure of the errors incurred by a modeler who models the *TES*/*M*/1 queue as an *M*/*M*/1 queue, thereby ignoring autocorrelation in the arrival process. Note that the TES arrival process above becomes a Poisson process when its autocorrelations vanish; in that case, $d(\rho(1)) = 0$.

Table 10.1 displays the relative deviations of (10.2) for two representative cases: (1) $\mu = 1$ and $\lambda = 0.25$ (light traffic regime with utilization $u = 0.25$), and (2) $\mu = 1$ and $\lambda = 0.80$ (heavy-traffic regime with utilization $u = 0.8$). There is one exception: the column corresponding to $\rho(1) = 0$ displays the *exact* mean waiting times for the baseline *M*/*M*/1 system (with respect to which the deviations, $d(\rho(1))$, are computed elsewhere in the table). Note that the relative deviation compares apples to apples, since distributions of both the service processes and the interarrival time processes (and hence the corresponding arrival rates) remain unchanged in both systems, and only the magnitude of $\rho(1)$ is varied in the *TES*/*M*/1 system (it is, of course, always zero in the *M*/*M*/1 system).

The *TES*/*M*/1 mean waiting times were estimated by Monte Carlo simulation, since their analytic form is unknown. Note the dramatic *nonlinear* effect of the lag-1 autocorrelation on the relative deviation; in particular at $\rho(1) = 0.85$, the relative deviation for the case of light traffic is about 9,000%, while for the case of heavy traffic it climbs to over 23,000%! Clearly, a naïve use of an *M*/*M*/1 model that ignores autocorrelations

Table 10.1

Relative deviations of mean waiting times for *M/M/*1 and *TES/M/*1 systems

Machine Utilization, u	Lag-1 Autocorrelation of Job Interarrival Times, $\rho(1)$							
	−0.55	−0.40	−0.25	0	0.25	0.50	0.75	0.85
0.25	−35.6%	−31.5%	−24.6%	0.3317	61.2%	260.9%	2047%	9156%
0.80	4450%	246%	27%	3.9974	78.56%	503%	5315%	23463%

in its *TES/M/1* counterpart produces unacceptable errors and is misleadingly over-optimistic. For more information, see Livny et al. (1993). For additional studies of the performance impact of correlations in various random components of manufacturing systems, see Altiok and Melamed (2001).

10.3 AUTOCORRELATION MODELING WITH TES PROCESSES

Having established the importance of correlation modeling, we turn our attention to the issue of modeling correlations in empirical data, say, $\{z_1, \ldots, z_N\}$, assumed to come from a stationary process (see Section 3.9). The first step is to check whether the empirical time series has appreciable autocorrelations for a range of lags, $\tau = 1, \ldots, T$. To this end, we collect all pairs of observations $\{(z_1, z_{1+\tau}), (z_2, z_{2+\tau}), \ldots, (z_{1N-\tau}, z_N)\}$ for each given lag τ, and compute the corresponding sample correlation coefficient estimates, using Eq. 3.100, namely,

$$\hat{\rho}(\tau) = \frac{\sum_{i=1}^{N-\tau} (z_i - \bar{z})(z_{i+\tau} - \bar{z})}{\sum_{i=1}^{N} (z_i - \bar{z})^2}.$$

In practice, an inspection of the empirical autocorrelation function, $\hat{\rho}(\tau)$, will suffice to determine whether significant autocorrelations are present in the data (a rule of thumb is to model those autocorrelation lags with absolute values exceeding about 0.15 and ignore autocorrelation function tails that fall below this threshold).

Having established the presence of significant autocorrelations in the empirical data, the next problem faced by the modeler is how to construct a model incorporating the observed autocorrelations. This problem has two aspects:

1. How can one define a versatile family of stationary stochastic processes that can simultaneously model a broad range of marginal distributions and autocorrelation functions?
2. How does one go about fitting a particular process from such a family to empirical data obtained from field measurements?

The generality called for by the first aspect appears to be quite daunting. In particular, how can we "cook up" a stationary process with an arbitrarily prescribed marginal distribution? And even if we can, would we still have additional modeling flexibility to fit (or at least approximate) the empirical autocorrelation function? Fortunately, both questions can be answered in the affirmative for a large number of cases. Recent developments in the theory of time series modeling have yielded new methodologies that can do just that. These methodologies include the aforementioned *TES* (*transform-expand-sample*) class (Melamed [1991, 1997]), and the *ARTA* (*autoregressive to anything*) class (Cario and Nelson [1996]). Here we shall focus on TES processes.

The TES class has several modeling advantages:

- It is relatively easy (and fast) to generate a sequence of TES variates.
- Any marginal distribution can be *fitted* exactly.

- A broad range of autocorrelation functions can be approximated, including decaying, oscillating, and alternating autocorrelation functions.
- Fitting algorithms have been devised, but they require software support.
- The physical meaning of TES processes suggests how to change the characteristics of the associated autocorrelation function. This is very useful in sensitivity analysis of correlations—a main component of correlation analysis.

TES processes have been studied in some detail. The theory of TES processes was developed in Melamed (1991, 1997) and Jagerman and Melamed (1992a, 1992b, 1994); TES fitting is presented in Jelenkovic and Melamed (1995); TES-based forecasting is described in Jagerman and Melamed (1995); and software support for TES modeling is described in Hill and Melamed (1995). A detailed overview of TES processes can be found in Melamed (1993), and a description of TES applications in various domains appears in Melamed and Hill (1995). Here we shall present an overview of the TES class at an elementary level. For complete details, the previous references are recommended.

10.4 INTRODUCTION TO TES MODELING

The definition of a TES process involves two stochastic processes: an auxiliary process, called the *background process*, and a target process, called the *foreground process*. The two processes operate in lockstep in the sense that they are connected by a deterministic transformation. More specifically, the state of the background process is mapped to a state of the foreground process in such a way that the latter has a prescribed marginal distribution and a prescribed autocorrelation function (the fifth and strongest signature in Section 10.1). This strong signature is estimated from empirical data.

The introduction of two processes in lockstep (instead of just one) is needed for technical reasons to allow us to fit a TES process to the aforementioned strong signature. As we shall see, the background process is defined so as to yield random variables that are *always* uniform on [0, 1), and the foreground process is transformed to match a prescribed distribution and simultaneously approximate a prescribed autocorrelation function.

The definition of TES processes makes use of a simple mathematical operation, called *modulo-1 arithmetic*, which is just ordinary arithmetic restricted to the familiar fractions (values in the interval [0, 1), with the value 1 excluded). We use the notation $\langle x \rangle = x - \max\{\text{integer } n\colon n \leq x\}$ to denote the fractional value of any number x, and refer to this operation as *modulo-1 reduction*. Note that fractional values are defined for any positive or negative number. Three cases exemplify modulo-1 arithmetic:

- For zero, we simply have $\langle 0 \rangle = 0$.
- For positive numbers, we have the familiar fractional values. For example, $\langle 0.6 \rangle = \langle 1.6 \rangle = \langle 2.6 \rangle = \ldots = 0.6$.
- For negative values, the fractional part is the complementary value relative to 1. For example, $\langle -0.6 \rangle = \langle -1.6 \rangle = \langle -2.6 \rangle = \ldots = 1 - 0.6 = 0.4$.

Modulo-1 addition consists of operations of the form $\langle x + y \rangle$, and can be visualized geometrically as addition on a circle with circumference 1 (we refer to this circle as the *unit circle*). To illustrate, consider the unit circle depicted in Figure 10.1.

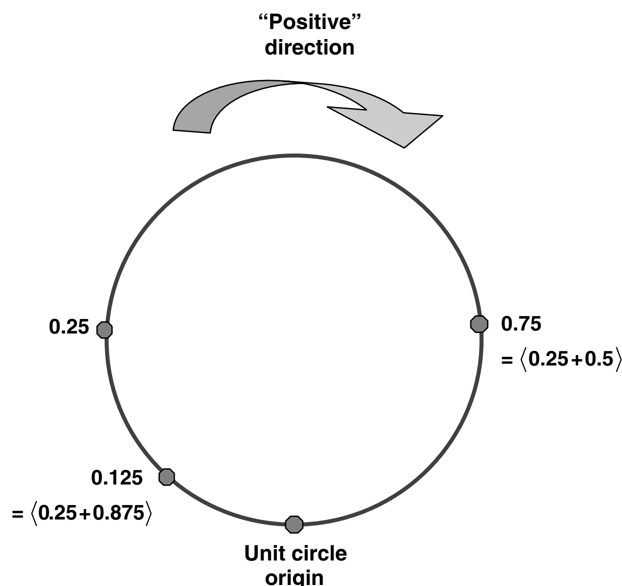


Figure 10.1 Visual representation of modulo-1 arithmetic.

In Figure 10.1, the unit circle is drawn with its origin at the bottom, and the "positive" direction is clockwise (the "negative" direction is counterclockwise). As in ordinary geometry, a point corresponds to a value interpreted as the clockwise "distance" from the origin (the length of the arc from the origin and clockwise to that point). For example, the value 0.25 is at 9 o'clock on the unit circle, and the clockwise arc from the origin spans one-quarter of the circle circumference. To perform the operation $\langle 0.25 + 0.5 \rangle$, we measure an arc whose length is half the circle conference from the point 0.25 and clockwise, yielding the point 0.75 at 3 o'clock on the unit circle. Finally, what happens if the sum exceeds 1? For example, consider $\langle 0.25 + 0.875 \rangle = \langle 1.125 \rangle = 0.125$. In this case, we sum precisely as before, except that now the second arc stretches past the origin, and its endpoint reaches the value 0.125. Thus, one needs only to take the fractional part and discard the integer part of the sum. Subtraction on the unit circle is analogous to addition, except that arcs of negative numbers are measured in the "negative" (counterclockwise) direction. No matter what addition we perform, the sum is always a point on the unit circle, and therefore a value in the range [0, 1).

A key mathematical result that allows us to endow TES processes with a uniform marginal distribution is the following somewhat surprising lemma:

Lemma of Iterated Uniformity (Melamed [1991]). If U is a uniform random variable on [0, 1) and V is *any* random variable independent of U, then $\langle U + V \rangle$ is also uniform on [0, 1), *regardless* of the distribution of V!

This lemma shows how to construct a large and versatile class of marginally uniform random processes. Let U_0 be a random variable with a uniform distribution on [0, 1). Let further $\{V_n\}_{n=1}^{\infty}$ be any sequence of iid random variables (with common density f_V), provided that each is also independent of U_0. The random variables V_n are called

innovations and their sequence is called an *innovation process*. We can now define a stochastic process

$$U_0, \langle U_0 + V_1 \rangle, \langle U_0 + V_1 + V_2 \rangle, \ldots, \langle U_0 + V_1 + \ldots, + V_n \rangle, \ldots,$$

and by the lemma of iterated uniformity, each of these random variables will be uniform on [0, 1) as required. Furthermore, for any prescribed distribution function, F, each could be further transformed into a foreground process,

$$F^{-1}(U_0),\ F^{-1}(\langle U_0 + V_1 \rangle),\ F^{-1}(\langle U_0 + V_1 + V_2 \rangle), \ldots,\ F^{-1}(\langle U_0 + V_1 + \ldots + V_n \rangle), \ldots,$$

and by the inverse transform method (see Section 4.2), each of the preceding random variables will have the prescribed distribution F. But we can achieve even more. Note that this construction is guaranteed to fit any distribution F, *regardless* of the innovation process $\{V_n\}_{n=1}^{\infty}$. We can now proceed to select an innovation process such that the previous transformed process fits (or approximates) a prescribed autocorrelation function, in addition to a prescribed marginal distribution. The freedom to select an appropriate innovation provides the basis for computer-aided TES modeling (Hill and Melamed [1995]).

The foregoing discussion has addressed a particular subclass of TES processes. General TES processes are classified into two main variants, TES$^+$ and TES$^-$, where the superscripts derive from the sign (plus or minus) of the associated lag-1 autocorrelations. In the next few subsections we formally define TES processes.

10.4.1 Background TES Processes

Let U_0 be an initial random variable with a uniform distribution on [0, 1), and let $\{V_n\}_{n=1}^{\infty}$ be an innovation sequence (any iid sequence of random variables, independent of U_0). A TES$^+$ background process, $\{U_n^+\}$, is defined by the recursive scheme

$$U_n^+ = \begin{cases} U_0, & n = 0 \\ \langle U_{n-1}^+ + V_n \rangle, & n > 0 \end{cases}, \tag{10.3}$$

while its TES$^-$ counterpart, $\{U_n^-\}$, is defined by

$$U_n^- = \begin{cases} U_n^+, & n \text{ even} \\ 1 - U_n^+, & n \text{ odd} \end{cases}. \tag{10.4}$$

The former variant is by far most commonly used in practice. A TES$^-$ sequence should be considered, however, when empirical sample paths or autocorrelation functions have an alternating (zigzag) appearance. Note that according to the lemma of iterated uniformity, all background TES processes, $\{U_n^+\}$ and $\{U_n^-\}$, are marginally *uniform* on [0, 1); however, each of them is generally an *autocorrelated* process.

In view of the foregoing discussion, background TES processes can be visualized as a *random walk on the unit circle*. Consider a simple case where the innovation variate is uniform on a subinterval [L, R) of the unit circle, so its density is a single step of length not exceeding 1, as depicted in Figure 10.2. The resulting process is called a *basic TES process*. In Figure 10.2, the unit circle lies at the bottom, in the two-dimensional plane, with the current TES$^+$ variate, U_n^+, at the 1 o'clock position.

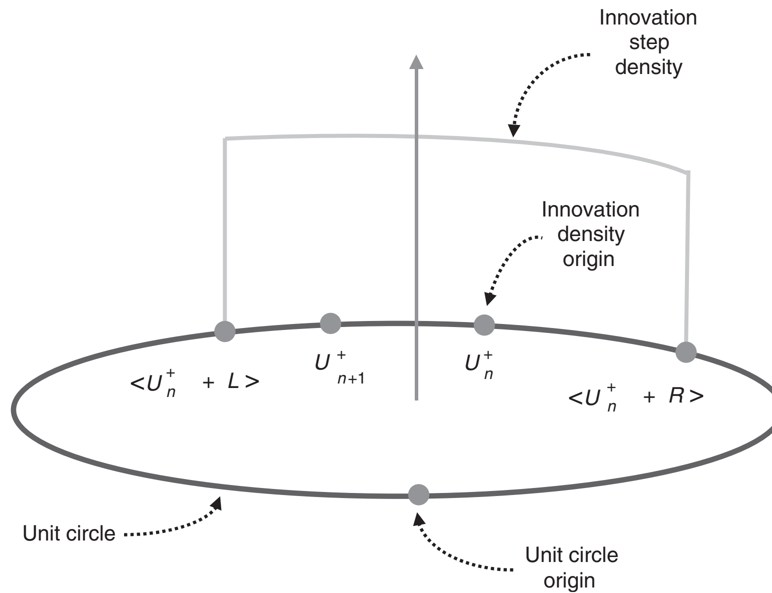


Figure 10.2 Visual representation of a basic TES$^+$ process.

A step-function innovation density was erected over the unit circle, with values in the third dimension (the "up" direction, indicated by the north-pointing arrow). The origin for the *support* of the innovation density (the region over which the density is positive) was set at the current TES variate, U_n^+. To obtain the next TES$^+$ variate, U_{n+1}^+, we sample a value on the unit circle from the innovation density, as a uniform value in the range $\langle U_n^+ + L \rangle$ to $\langle U_n^+ + R \rangle$ (the particular case considered in Figure 10.2 implies that $L < 0$, but this is just a special case). Indeed, because the origin of the innovation density was set at U_n^+, this procedure implies $U_{n+1}^+ = \langle U_n^+ + V_{n+1} \rangle = \langle U_n^+ + L + (R - L)W_{n+1} \rangle$, where $\{W_n\}$ is an iid sequence of uniform random variables on [0, 1), in agreement with Eq. 10.3.

A similar geometric interpretation applies to the TES$^-$ counterpart of the process in Figure 10.2. The only difference is that every odd time index n, we take the complement of the TES$^+$ variate with respect to 1, as specified by the odd case of Eq. 10.4.

Recall that by the lemma of iterated uniformity, all TES background processes are marginally uniform on [0, 1). In particular, for the TES$^+$ process of Figure 10.2 the uniformity property holds for any selection of L and R. However, the corresponding autocorrelation function *is* affected by such selections. The following list summarizes qualitatively the effect of L and R on the autocorrelation of a background TES process.

- The width of the innovation-density support has a major effect on the *magnitude* of the autocorrelations. The larger the support (namely, the larger the value of $R - L$), the smaller the magnitude (in fact, when $R - L = 1$, the autocorrelations vanish altogether). The intuitive reason for this behavior is that a larger support admits larger distances between successive TES iterates, resulting in a reduced autocorrelation magnitude. The smaller the support, the "closer" are U_n^+ and U_{n+1}^+. This fact increases the autocorrelation magnitude.

- The location of the innovation-density support affects the shape of the autocorrelation function. In fact, if the support is not symmetric about the origin, then the autocorrelation function assumes an oscillating form, and otherwise it is monotone decreasing. Moreover, the further the support is from the current background TES variate, the higher is the frequency of oscillation. The sample paths are also strongly affected. If the innovation density is symmetric about the origin (i.e., $L = -R$ in Figure 10.2), then the sample paths will not exhibit drift around the unit circle. In the nonsymmetric case, the sample paths are cyclical (but random) in appearance, with clockwise drift for $|L| < |R|$, and counterclockwise drift for $|L| > |R|$.

These properties of background TES processes are generally inherited by their foreground counterparts. Consequently, we illustrate the impact of the (L, R) pairs on the autocorrelation function of various basic TES (background) processes. Each of the Figures 10.3 through 10.7 displays a correlogram (graph of the autocorrelation function) with some statistics below it, including the autocorrelation estimates of the first 10 lags. The figures were generated using the Arena *Output Analyzer* (see Section 9.6).

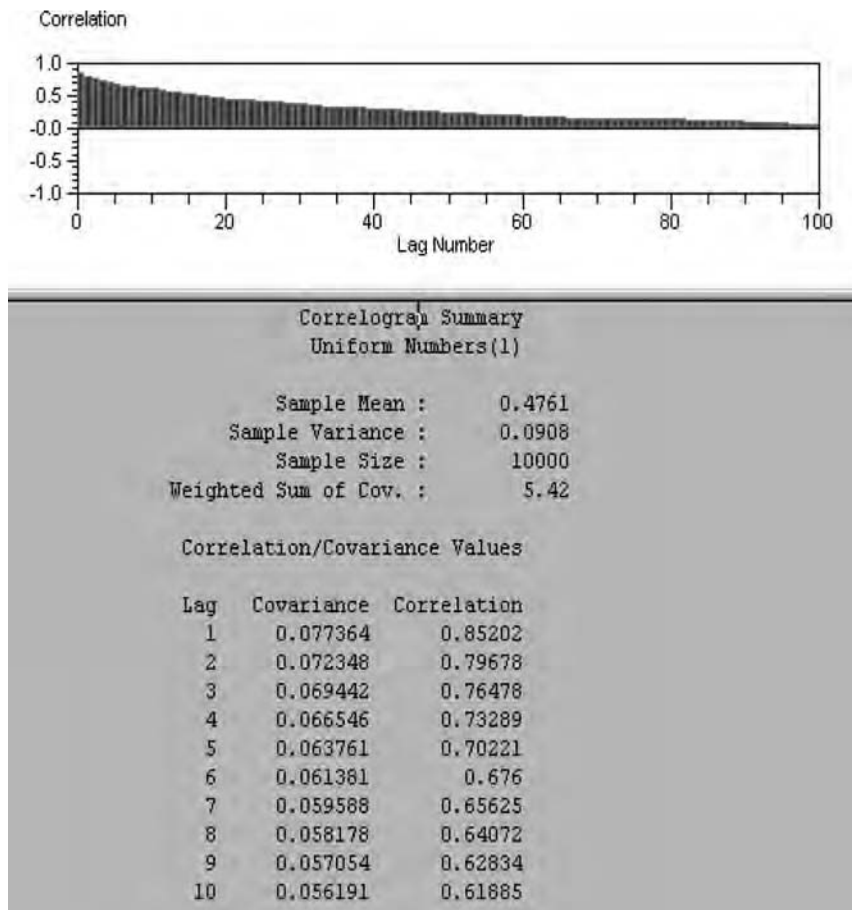
Figure 10.3 depicts the autocorrelation function of a basic TES$^+$ process with $(L, R) = (-0.05, 0.05)$. Note that the autocorrelation function is monotone decreasing with a moderate decay rate and large-magnitude autocorrelations in its lower lags. The monotone shape of the autocorrelation function is due to the symmetry of the innovation density about its origin ($|L| = |R| = 0.05$), while the large magnitude of the low-lag autocorrelations is a consequence of a narrow support for the innovation density (in this case, the support has width $R - L = 0.1$).

Figure 10.4 depicts the autocorrelation function of a basic TES$^+$ process with $(L, R) = (-0.2, 0.2)$, which corresponds to a considerably larger support of the innovation density. Note that in this case, the autocorrelation function is still monotone decreasing, but with a considerably higher decay rate and smaller-magnitude autocorrelations overall. The monotone shape of the function is again due to the symmetry of the innovation density about the origin ($|L| = |R| = 0.2$), while the smaller magnitude of the autocorrelations is a consequence of the much wider support of the innovation density ($R - L = 0.4$), as compared to Figure 10.3.

Figure 10.5 depicts the autocorrelation function of a basic TES$^+$ process with $(L, R) = (-0.01, 0.05)$, corresponding to a markedly nonsymmetric innovation density. Note carefully the oscillating pattern of the autocorrelation function. This is due to the fact that the process has clockwise drift on the unit circle. Indeed, the process is five times more likely to move to the "right" (clockwise direction on the unit circle of Figure 10.1) than to the "left" (counterclockwise direction on the unit circle of Figure 10.1).

Figure 10.6 depicts the autocorrelation function of a basic TES$^-$ process with $(L, R) = (-0.05, 0.05)$. Note carefully the alternating sign pattern of the autocorrelation function, with the highest-magnitude (negative) value at lag 1. A comparison of Figure 10.6 with Figure 10.3 (which have identical innovation densities) reveals that while the autocorrelation function in the former is monotone decreasing, the one in the latter only has a decaying envelope. In fact, it readily follows from Eq. 10.4 that the autocorrelation functions of any two background processes with identical innovation densities, $U^+ = \{U_n^+\}$ and $U^- = \{U_n^-\}$, are related by

$$\rho_{U^-}(n) = (-1)^n \rho_{U^+}(n), \quad n = 1, 2 \ldots. \tag{10.5}$$



Correlogram Summary
Uniform Numbers(1)

Sample Mean :	0.4761
Sample Variance :	0.0908
Sample Size :	10000
Weighted Sum of Cov. :	5.42

Correlation/Covariance Values

Lag	Covariance	Correlation
1	0.077364	0.85202
2	0.072348	0.79678
3	0.069442	0.76478
4	0.066546	0.73289
5	0.063761	0.70221
6	0.061381	0.676
7	0.059588	0.65625
8	0.058178	0.64072
9	0.057054	0.62834
10	0.056191	0.61885

Figure 10.3 Autocorrelation function of a basic TES$^+$ process with $(L, R) = (-0.05, 0.05)$.

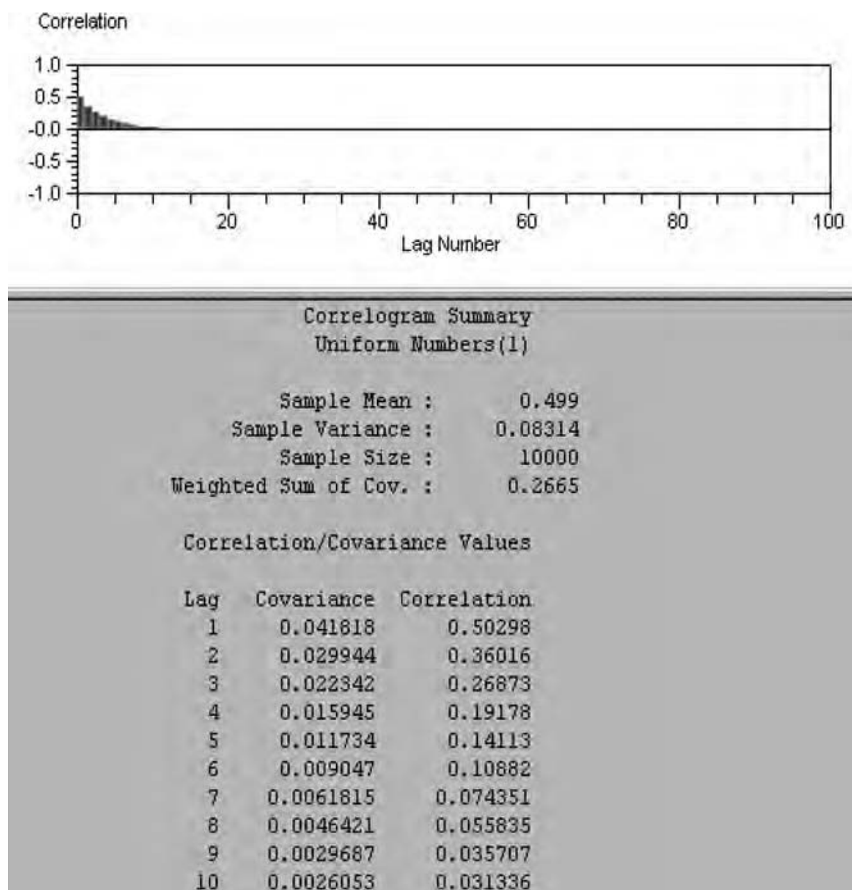
Thus, if we reflect the negative autocorrelations in Figure 10.6 onto the positive vertical axis, the result will be the autocorrelation function of Figure 10.3. Note, however, that the numerical values of autocorrelation magnitudes in the two figures are only approximately equal, since the relation in Eq. 10.5 is theoretical, while the numbers in the figure are estimated autocorrelations and thus contain experimental errors.

Finally, Figure 10.7 depicts the autocorrelation function of a basic TES$^-$ process with $(L, R) = (-0.01, 0.05)$. This figure exhibits a combined alternating and oscillating pattern.

10.4.2 Foreground TES Processes

A foreground TES process is obtained from a background TES process by a deterministic transformation, D, called a *distortion*. Thus, a foreground TES$^+$ process is of the form

$$X_n^+ = D(U_n^+), \tag{10.6}$$



Correlogram Summary
Uniform Numbers(1)

Sample Mean :	0.499
Sample Variance :	0.08314
Sample Size :	10000
Weighted Sum of Cov. :	0.2665

Correlation/Covariance Values

Lag	Covariance	Correlation
1	0.041818	0.50298
2	0.029944	0.36016
3	0.022342	0.26873
4	0.015945	0.19178
5	0.011734	0.14113
6	0.009047	0.10882
7	0.0061815	0.074351
8	0.0046421	0.055835
9	0.0029687	0.035707
10	0.0026053	0.031336

Figure 10.4 Autocorrelation function of a basic TES$^+$ process with $(L, R) = (-0.2, 0.2)$.

while a foreground TES$^-$ process is of the form

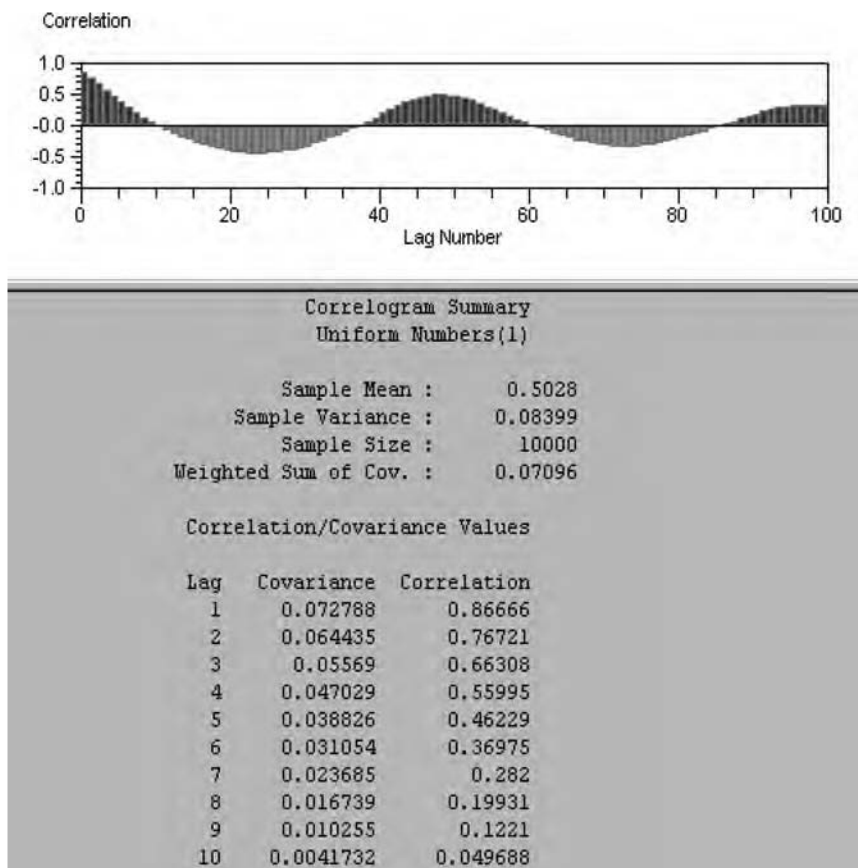
$$X_n^- = D(U_n^-). \tag{10.7}$$

The somewhat unusual term, *distortion*, refers to the fact that the uniform distribution is "special" in that it characterizes the marginal distribution of background TES processes; consequently, transformations of uniform random variables "distort" this special distribution in changing it to any other one. In fact, the uniform distribution is special in a probabilistic sense in that it gives rise to any other distribution via the inverse transform method (see Section 4.2): If X is a random variable with a prescribed distribution function F, and U is a uniform random variable on [0, 1], then X can be obtained from U by the operation

$$X = F^{-1}(U). \tag{10.8}$$

Indeed, X as defined in this equation *has precisely* the prescribed distribution, F.

As it happens, a simple distortion of the form in Eq. 10.8 is rarely adequate from a modeling standpoint, due to the peculiar geometry of the circle. To see why this is the



Correlogram Summary
Uniform Numbers(1)

Sample Mean : 0.5028
Sample Variance : 0.08399
Sample Size : 10000
Weighted Sum of Cov. : 0.07096

Correlation/Covariance Values

Lag	Covariance	Correlation
1	0.072788	0.86666
2	0.064435	0.76721
3	0.05569	0.66308
4	0.047029	0.55995
5	0.038826	0.46229
6	0.031054	0.36975
7	0.023685	0.282
8	0.016739	0.19931
9	0.010255	0.1221
10	0.0041732	0.049688

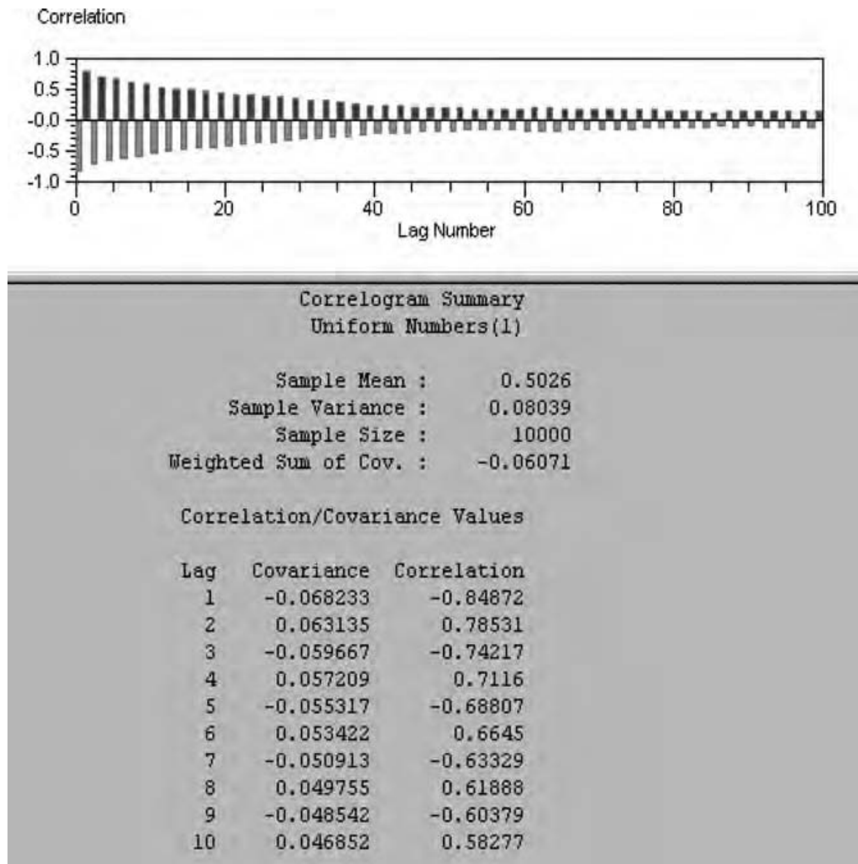
Figure 10.5 Autocorrelation function of a basic TES$^+$ process with $(L, R) = (-0.01, 0.05)$.

case, recall the geometric interpretation of modulo-1 arithmetic on the unit circle in Figure 10.1, and for each two points on it, $0 \leq x < 1$ and $0 \leq y < 1$, define the following two notions of distance:

- The *circular distance* $C(x) = \min\{\langle x - y \rangle, \langle y - x \rangle\}$ is the shortest distance (clockwise or counterclockwise) from x to y as measured on the *unit circle*.
- The *linear distance* $L(x) = |x - y|$ is the ordinary difference between x and y as measured on the *real line*.

Note carefully that the concepts of circular distance and linear distance are *distinct*, since the values they yield do not always coincide due to the different geometries of the circle and the real line. For example, $C(0.1, 0.9) = 0.2$, whereas $L(0.1, 0.9) = 0.8$. Keeping this in mind, consider a background TES$^+$ process, $\{U_n^+\}$, whose innovation variate is “small”—say, its density has a support of length 0.01. It follows that any circular distance between successive variates cannot exceed 0.01, that is, $C(U_n^+, U_{n+1}^+) \leq 0.01$.

As long as the process meanders away from the origin, the sample path of the process will have a “smooth” appearance, since the circular distance and the linear distance would then coincide. But what happens when the origin is crossed, clockwise



Correlogram Summary
Uniform Numbers(1)

Sample Mean : 0.5026
Sample Variance : 0.08039
Sample Size : 10000
Weighted Sum of Cov. : -0.06071

Correlation/Covariance Values

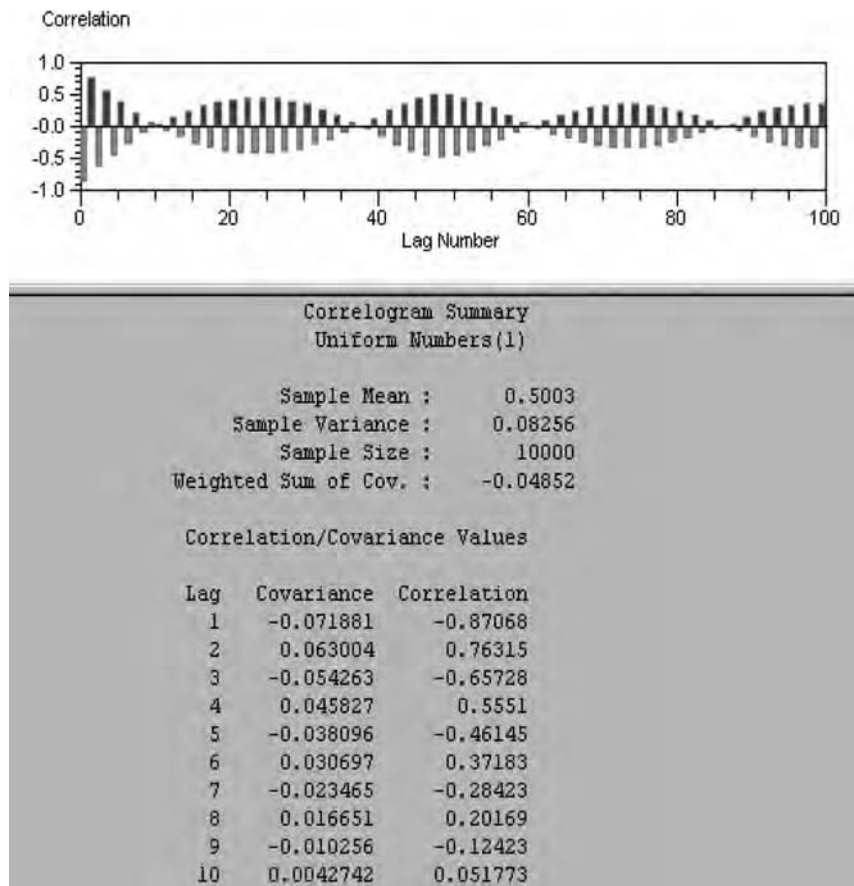
Lag	Covariance	Correlation
1	-0.068233	-0.84872
2	0.063135	0.78531
3	-0.059667	-0.74217
4	0.057209	0.7116
5	-0.055317	-0.68807
6	0.053422	0.6645
7	-0.050913	-0.63329
8	0.049755	0.61888
9	-0.048542	-0.60379
10	0.046852	0.58277

Figure 10.6 Autocorrelation function of a basic TES$^-$ process with $(L, R) = (-0.05, 0.05)$.

or counterclockwise? In that case the circular distance remains small (not exceeding 0.01) but the linear distance can be arbitrarily close to 1. In other words, the sample path when plotted on the real line would have an abrupt "jump." Furthermore, since both F and its inverse, F^{-1}, are monotone nondecreasing (this is a property inherent in distribution functions), these abrupt "jumps" are inherited by the foreground TES process. Consequently, although the marginal distribution and autocorrelation function of the resultant foreground process may well fit their empirical counterparts, the model's sample paths may be *qualitatively* different from the empirical data.

The foregoing discussion motivates the need for a mathematical means of "*smoothing*" the sample paths of background TES processes while simultaneously *retaining* their uniform distribution. Fortunately there is a simple way to achieve both goals, using a so-called *stitching transformation*, which maps the interval [0, 1] onto itself. The family of stitching transformations is defined by

$$S_\xi(u) = \begin{cases} \dfrac{u}{\xi}, & \text{if } 0 \le u \le \xi \\[2ex] \dfrac{1-u}{1-\xi}, & \text{if } \xi \le u \le 1 \end{cases} \tag{10.9}$$



Correlogram Summary
Uniform Numbers(1)

Sample Mean : 0.5003
Sample Variance : 0.08256
Sample Size : 10000
Weighted Sum of Cov. : -0.04852

Correlation/Covariance Values

Lag	Covariance	Correlation
1	-0.071881	-0.87068
2	0.063004	0.76315
3	-0.054263	-0.65728
4	0.045827	0.5551
5	-0.038096	-0.46145
6	0.030697	0.37183
7	-0.023465	-0.28423
8	0.016651	0.20169
9	-0.010256	-0.12423
10	0.0042742	0.051773

Figure 10.7 Autocorrelation function of a basic TES$^-$ process with $(L, R) = (-0.01, 0.05)$.

where ξ is a so-called *stitching parameter* lying in the interval [0, 1], which characterizes each stitching transformation. Note that Eq. 10.9 reduces to $S_0(u) = 1 - u$ for $\xi = 0$, and to $S_1(u) = u$ for $\xi = 1$. Typical stitching transformations are depicted in Figure 10.8. Equation 10.9 reveals that for $\xi = 0$, the stitching transformation, S_0, assumes the form of the dashed curve in Figure 10.8, while for $\xi = 1$ it becomes the identity (dotted curve in Figure 10.8). However, for $0 < \xi < 1$, S_ξ has a triangular shape with the apex at the point ξ (solid curve in Figure 10.8). In fact, all stitching transformations attain their maximal value of 1 at the point ξ, that is, $S_\xi(\xi) = 1$.

Why does a stitching transformation, S_ξ, with $0 < \xi < 1$, have a smoothing effect on the unit circle? Examine again the solid curve in Figure 10.8. Note that we may consider the x-axis as a one-dimensional representation of the unit circle (to this end, just imagine cutting the unit circle at its origin and unraveling it into a straight line of length unity). Note further that the solid curve satisfies the following facts:

- It is continuous on the interval [0, 1] on the real line.
- It assumes the same value (zero) at the extreme points 0 and 1.
- The points 0 and 1 coincide on the unit circle.

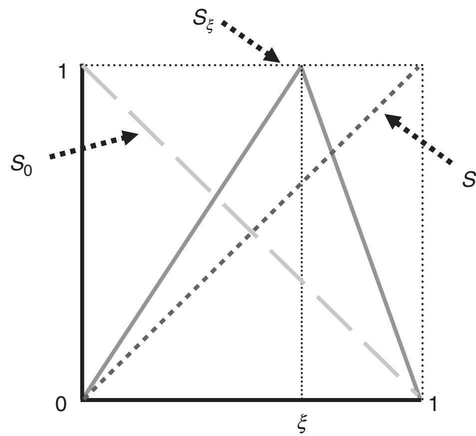


Figure 10.8 Stitching transformations for $\xi = 0$ (dashed curve), $\xi = 1$ (dotted curve), and $0 < \xi < 1$ (solid curve).

We therefore conclude that the solid curve is also continuous on the unit circle. This is evidently not the case for the other two curves (dashed and dotted): While both are continuous on the interval [0, 1] on the real line, neither is continuous on the unit circle. In fact, for the extremal cases, $\xi = 0$ and $\xi = 1$, the triangle degenerates into a sloped line and the values of the corresponding curves differ at 0 and 1. Thus, the solid curve is continuous on the unit circle, while the dashed and dotted curves are not. The effect of applying the solid curve to a background TES process is to "smooth" its sample paths whenever they "jitter" about the origin.

Experience shows that the majority of practical models call for the application of a stitching transformation, S_ξ, with $0 < \xi < 1$, in order to obtain a TES model with "realistic-looking" sample paths. Moreover, *all* stitching transformations enjoy the following valuable property: Each transforms a uniform random variable on [0, 1) to (another) random variable that is also uniform on [0, 1). See Melamed (1991) for a simple proof. Thus, the application of stitching not only "smoothes" sample paths, but also allows us to use the inverse transform method to fit any empirical distribution. Note, however, that while stitching does *not* change the (uniform) marginal distribution of background processes, it *does* affect their autocorrelations.

Figures 10.9 and 10.10 illustrate visually the effect of stitching on sample paths, by comparing the sample path of an unstitched background TES$^+$ process to a stitched counterpart. Observe the sharp drop in the magnitude of the unstitched variates in the left portion of Figure 10.9 (between time indices 70 and 80). This marked drop is due to the fact that the unstitched process moved on the unit circle in relatively small (random) steps, driven by clockwise drift. Eventually, the unstitched process "crosses" the origin in a clockwise direction (Figure 10.1), so that the magnitude of its variates changes abruptly from relatively large (around 1) to relatively small (around 0), thereby registering a "visual continuity" that is perceived as a sharp drop.

In contrast, the stitched process in Figure 10.10 passed through the origin of the unit circle at the same time index, descending gradually, without exhibiting a "visual discontinuity." Generally, stitching transformations in the range $0 < \xi < 1$ will moderate any sharp drops or sharp increases in corresponding unstitched processes, and the "smoothed" paths generated by them will often look more realistic. It is up to the

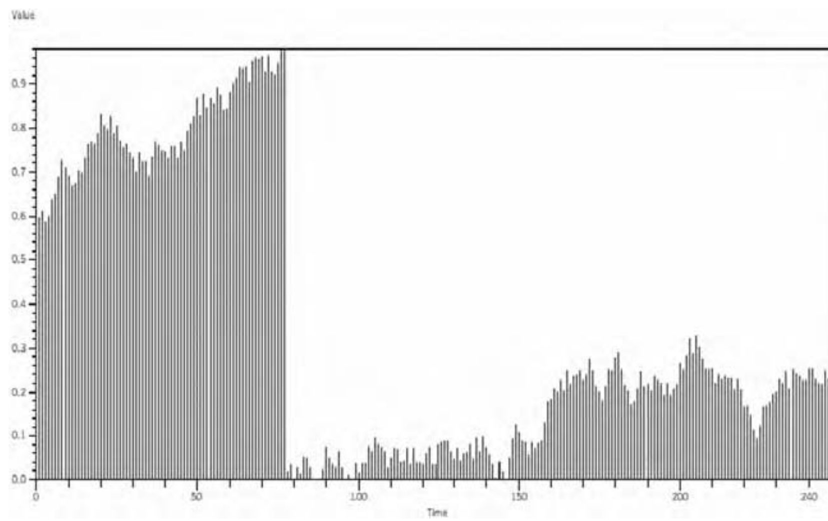


Figure 10.9 Sample path of a basic TES$^+$ background process with $(L, R) = (-0.04, 0.055)$ and without stitching ($\xi = 1$).

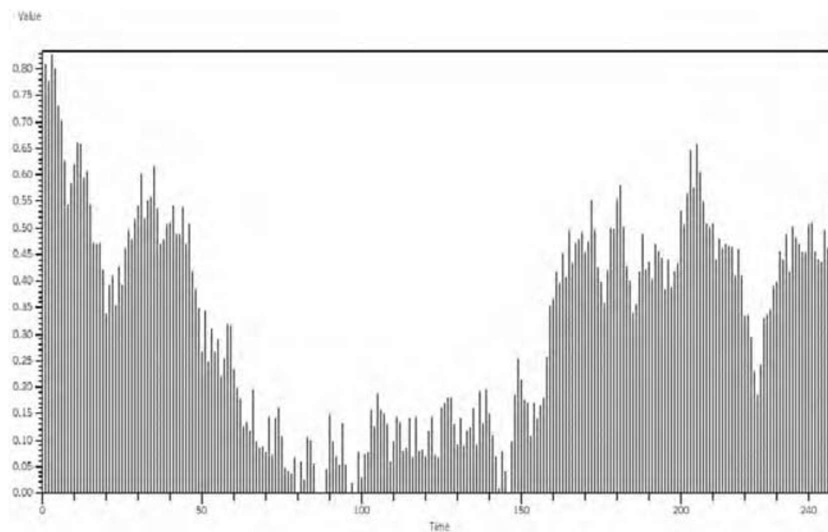


Figure 10.10 Sample path of a basic TES$^+$ background process with $(L, R) = (-0.04, 0.055)$ and with stitching ($\xi = 0.5$).

modeler to select a stitched or unstitched process, depending on the sample path behavior exhibited by the empirical time series to be modeled.

10.4.3 Inversion of Distribution Functions

The inversion of distribution functions via the inverse transform method is usually straightforward (see Eq. 10.8 and Section 4.2). In most cases, an analytical formula can

be derived for the inverse distribution function, although in some cases a numerical approximation must be devised. Here we demonstrate two examples of analytical inversions in widespread use.

An important example is the inverse distribution function of the exponential distribution

$$F_{exp}(x) = 1 - e^{-\lambda x}, \tag{10.10}$$

where λ is the rate parameter (see also Section 4.2.2). Substituting y for the left side in Eq. 10.10, and expressing x as a function of y yields the inverse exponential distribution function

$$F_{exp}^{-1}(y) = \frac{1}{\lambda}\ln(1-y). \tag{10.11}$$

In fact, the value $1 - y$ on the right side of Eq. 10.11 can be replaced by y if we wish to speed up the inversion (if Y is uniform on [0, 1], so is $1 - Y$).

Figures 10.11 and 10.12 display key statistics (sample path, histogram, and correlogram) of two foreground TES$^+$ processes with an exponential marginal distribution: the first one is an autocorrelated process, and the second one is an iid process. These processes were obtained by applying the inverse transform method of Eq. 10.11 with $\lambda = 1$ to basic TES$^+$ background processes. Thus, both have the *same* exponential marginal distribution of rate 1. However, the first basic TES$^+$ background process employs the pair $(L, R) = (-0.05, 0.05)$, while the second employs the pair $(L, R) = (-0.5, 0.5)$. Note carefully that both foreground processes have the same marginal distribution, as attested by their histograms. In contrast, the first foreground process exhibits significant autocorrelations, while the second has zero autocorrelations, as a consequence of its iid property (see the corresponding correlograms). In addition to the quantitative statistical differences, the corresponding sample paths exhibit qualitative differences as well. If the variates of these processes are interpreted as interarrival times, then the iid process in Figure 10.12 corresponds to a Poisson process (see Section 3.9.2), and thus the associated counting process of arrivals has independent increments. However, the autocorrelated process in Figure 10.11 has far "burstier" arrivals, even though their interarrival distribution is identical to that in Figure 10.12. Needless to say, offering these two arrival processes to the same service system would yield very different queueing statistics (e.g., waiting times in the first system would be much worse than in the second), a fact that again militates against ignoring temporal dependence in data, and supports the case for appropriate correlation analysis.

The example in Figure 10.12 is widely used in practical input analysis of empirical distributions, where $F = \hat{H}_\eta$ is derived from an empirical histogram density, $\hat{h}_\eta$, computed from an empirical sample $\eta = \{y_1, y_2, \ldots, y_N\}$ (see also Section 4.2.4). The distortion

$$D = \hat{H}_\eta^{-1}, \tag{10.12}$$

will fit any foreground TES process of the form presented in Eq. 10.6 or 10.7 to the empirical distribution, $\hat{H}_\eta$. A common estimator of the empirical distribution is the (normalized) histogram estimator, constructed as follows from an empirical time series $\eta = \{y_n\}$:

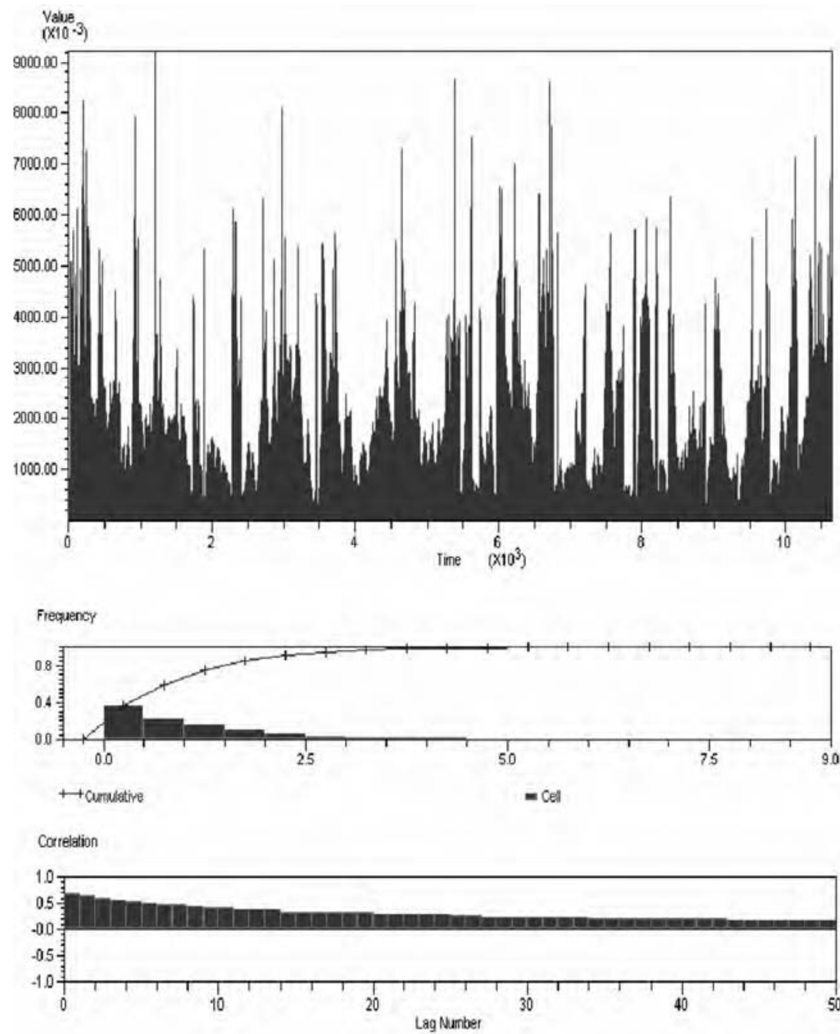


Figure 10.11 Sample path (top), histogram (middle), and correlogram (bottom) of an autocorrelated exponential basic TES$^+$ process with background parameters $(L, R) = (-0.05, 0.05)$.

1. Specify the histogram cells in the form of $J > 0$ nonoverlapping intervals $[l_j, r_j)$, $j = 1, 2, \ldots, J-1$ and $[l_J, r_J]$, which cover the range of values $\{y_n\}$.
2. Sort the observations, $\{y_n\}$, into the cells in such a way that y_n is placed in cell j whenever $l_j \le y_n < r_j$, $j = 1, 2, \ldots, J-1$ or $l_J \le y_n \le r_J$ (there is exactly one such cell by assumption).
3. For each cell $j = 1, 2, \ldots, J$, compute the relative frequency, $\hat{p}_j$, of that cell as the ratio of the number of observations in the cell to the total number of observations.

From the estimated relative frequencies, $\hat{p}_j$, it is easy to compute the histogram density as a step function that assumes the value $\hat{p}_j$ over the intervals $[l_j, r_j)$, $j = 1, 2, \ldots, J-1$, and $[l_J, r_J]$, whose widths are denoted by $w_j = r_j - l_j$. Letting the indicator function of any set A be defined by

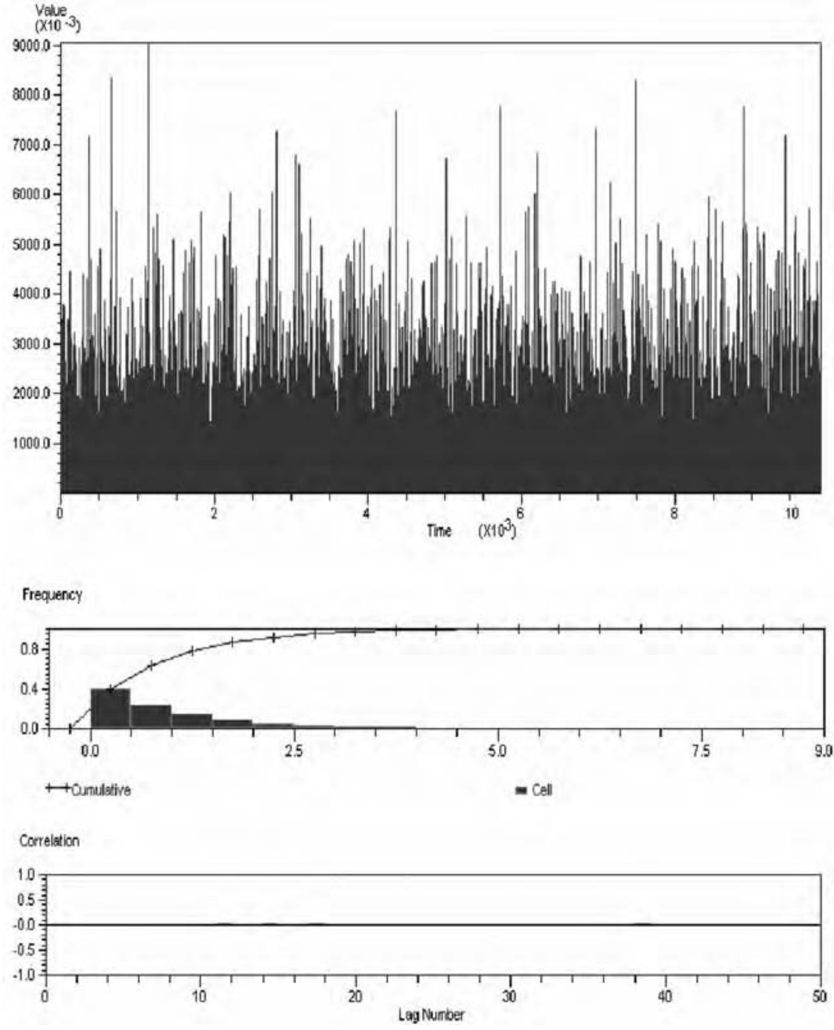


Figure 10.12 Sample path (top), histogram (middle), and correlogram (bottom) of an iid exponential basic TES$^+$ process with background parameters $(L, R) = (-0.5, 0.5)$.

$$1_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases},$$

we can readily express a histogram density as

$$\hat{h}_\eta(y) = \sum_{j=1}^{J-1} 1_{[l_j, r_j)}(y) \frac{\hat{p}_j}{w_j} + 1_{[l_J, r_J]}(y) \frac{\hat{p}_J}{w_J}. \tag{10.13}$$

A straightforward integration of Eq. 10.13 yields the histogram distribution function

$$\hat{H}_\eta(y) = \sum_{j=1}^{J-1} 1_{[l_j, r_j)}(y) \left[\hat{C}_{j-1} + (y - l_j) \frac{\hat{p}_j}{w_j} \right] + 1_{[l_J, r_J]}(y) \left[\hat{C}_{J-1} + (y - l_J) \frac{\hat{p}_J}{w_J} \right] \tag{10.14}$$

where $\hat{C}_0 = 0, \hat{C}_j = \sum_{i=1}^{j} \hat{p}_i,\ j = 1, 2, \ldots, J$, so that $\hat{C}_J = 1$. A modicum of algebra finally yields the formula for the inverse histogram distribution function

$$\hat{H}_\eta^{-1}(u) = \sum_{j=1}^{J} 1_{[\hat{C}_{j-1}, \hat{C}_j)}(u) \left[l_j + \left(u - \hat{C}_{j-1}\right) \frac{w_j}{\hat{p}_j} \right], \quad 0 \le u < 1. \tag{10.15}$$

Although a bit complex, the formula in Eq. 10.15 is not hard to implement on a computer.

10.5 GENERATION OF TES SEQUENCES

This section describes in detail the generation procedure for TES sequences in algorithmic form. It further illustrates the computer implementation of such algorithms in an Arena model.

For convenience, we separate the generation of TES$^+$ processes from that of TES$^-$ processes, even though the corresponding algorithms have large overlaps. We shall assume that the function mod1 implements modulo-1 reduction of any real number, and returns the corresponding fraction. Such a function is not always available in software platforms, and in Arena it is implemented by code in an *Assign* module. The following Algorithms 10.1 and 10.2 assume that TES modeling has already been carried out, and that the requisite parameters are available as inputs (explanatory comments are preceded by double slashes).

GENERATION OF TES$^+$ SEQUENCES

Inputs

1. *An innovation density* f_V // modeler choice
2. *A stitching parameter* ξ // modeler choice
3. *An inverse (marginal) distribution* F^{-1} // often $\hat{H}_\eta^{-1}$ of Eq. 10.15

Outputs

1. *A background TES*$^+$ *sequence* $\{U_n^+\}_{n=0}^{\infty}$
2. *A foreground TES*$^+$ *sequence* $\{X_n^+\}_{n=0}^{\infty}$

Algorithm

1. *Sample a value* $0 \le U_0 < 1$, *uniform on* [0, 1), // initial background variate
   and set $U_0^+ \leftarrow U_0$ *and* $n \leftarrow 0$ // more initializations
2. *Go to Step 6*. // go to generate the initial foreground variate
3. *Set* $n \leftarrow n + 1$ // bump up running index for next iteration
4. *Sample a value V from* f_V // sample an innovation variate, typically using
   // the inverse transform method
5. *Set* $U_n^+ \leftarrow \text{mod1}(U_{n-1}^+ + V)$ // compute a TES$^+$ background variate
   // using Eq. 10.3

6. *Set* $S \leftarrow S_\xi(U_n^+)$ // apply a stitching transformation, using Eq. 10.9
7. *Set* $X_n^+ \leftarrow F^{-1}(S)$ // compute a TES$^+$ foreground variate, by applying // an inverse distribution function
8. *Go to Step 3.* // loop for the next TES$^+$ variate generation

Generation of TES$^-$ Sequences

Inputs

1. *An innovation density* f_V // modeler choice
2. *A stitching parameter* ξ // modeler choice
3. *An inverse (marginal) distribution* F^{-1} // often $\hat{H}_\eta^{-1}$ of Eq. 10.15

Outputs

1. *A background TES$^-$ sequence* $\{U_n^-\}_{n=0}^\infty$
2. *A foreground TES$^-$ sequence* $\{X_n^-\}_{n=0}^\infty$

Algorithm

1. *Sample a value* $0 \le U_0 < 1$, *uniform on* $[0, 1)$ // initial background variate
   and set $U_0^- \leftarrow U_0^+ \leftarrow U_0$ *and* $n \leftarrow 0$ // more initializations
2. *Go to Step 7.* // go to generate the initial foreground variate
3. *Set* $n \leftarrow n + 1$ // bump up running index for next iteration
4. *Sample a value V from* f_V // sample the current innovation, typically // using the inverse transform method
5. *Set* $U_n^+ \leftarrow \text{mod}1(U_{n-1}^+ + V)$ // compute a TES$^+$ background variate, // using Eq. 10.3
6. *If n is even, set* $U_n^- \leftarrow U_n^+$ // compute a TES$^-$ background variate
   If n is odd, set $U_n^- \leftarrow 1 - U_n^+$ // using Eq. 10.4
7. *Set* $S \leftarrow S_\xi(U_n^-)$ // apply a stitching transformation // using Eq. 10.9
8. *Set* $X_n^- \leftarrow F^{-1}(S)$ // compute a TES$^-$ foreground variate, by // applying an inverse distribution function
9. *Go to Step 3.* // loop for the next TES$^-$ variate generation

Combining TES Generation Algorithms

We next exhibit an Arena program that combines the implementation of Algorithms 10.1 and 10.2 for any basic TES process (both TES$^+$ and TES$^-$) with an exponential distribution. This implementation generates a TES process arrival stream (a stream of entities with a TES interarrival process); however, TES processes with other functionalities (service times, demand sizes, time-to-failure and repair times, etc.) can also be constructed analogously. Accordingly, the following TES parameters are assumed to be inputs.

1. A pair of parameters L and R, such that $0 \leq L < R < 1$, which determines the basic innovation density

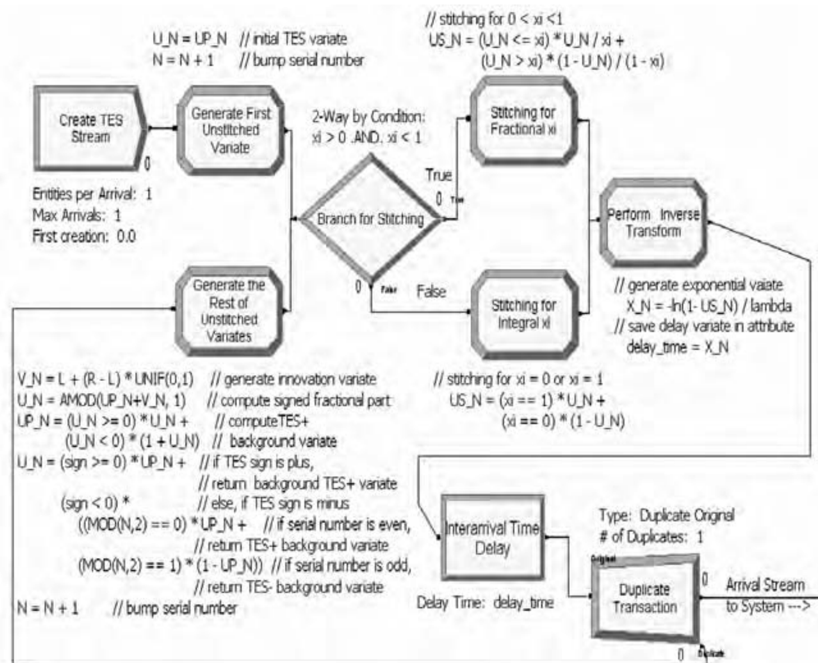
$$f_V(x) = \begin{cases} \frac{1}{R-L}, & L \leq x < R \\ 0, & otherwise \end{cases}$$

2. A stitching parameter $0 \leq \xi \leq 1$ (0.5 is typical)
3. An inverse of an exponential distribution function, $F^{-1}(u) = -\frac{1}{\lambda}\ln(1-u)$, for a given rate parameter, $\lambda > 0$

Figure 10.13 depicts the corresponding annotated Arena model (top) and all user-defined variables (bottom). The bottom portion of Figure 10.13 displays the spreadsheet view of the associated *Variable* module, which lists model parameters and variables and their initial values, if any (those requiring initialization are identified by a *1 rows* button label under the *Initial Values* column). The variable *sign* designates the generated process to be either TES$^+$(*sign* ≥ 0) or TES$^-$(*sign* < 0). The variables *L*, *R*, and *xi* are self-explanatory. The variable *lambda* is the rate parameter of the requisite exponential distribution, and *N* is the serial number of the current TES variate (initially 0). The remaining variables are related to TES theory: *UP_N* corresponds to U_n^+ (initialized to a value between 0 and 1 in the *Variable* module), *V_N* corresponds to V_n, *U_N* corresponds to an unstitched TES background variate (U_n^+ or U_n^- as the case may be), *US_N* corresponds to a stitched TES variate ($S_\xi(U_n^+)$ or $S_\xi(U_n^-)$ as the case may be), and *X_N* corresponds to a TES foreground variate (X_n^+ or X_n^- as the case may be).

The top portion of Figure 10.13 depicts the Arena model, annotated by key associated field names and values as well as associated SIMAN code fragments with comments preceded by double slashes. The model acts as a generator of a TES model (either TES$^+$ or TES$^-$) for an interarrival process in the following manner:

1. The *Create* module, called *Create TES Stream*, creates a single entity at time 0. That entity will later generate TES interarrival-time variates.
2. The entity proceeds to the *Assign* module, called *Generate First Unstitched Variate*, to set the initial value of *U_N*.
3. All other TES variates will be generated in the module, called *Generate the Rest of Unstitched Variates*, in subsequent iterations. Note carefully that the Arena function *AMOD* returns the signed fraction of a real number, and that the requisite modulo-1 reduction is carried out by adding 1 to negative fractions returned by it. The *sign* variable determines the type of TES process to be generated (TES$^+$ or TES$^-$).
4. The entity proceeds next to perform a stitching operation via the *Decide* module, called *Branch for Stitching*. From there it branches to one of the *Assign* modules, called *Stitching for Fractional xi* or *Stitching for Integral xi*, depending on the value of ξ (a single assignment would produce a compilation error, because the Arena implementation of assignments triggers a division by 0). The appropriate stitching transformation is carried out in these *Assign* modules.
5. Whichever branch was taken, the entity then proceeds to the *Assign* module, called *Perform Inverse Transform*, to compute the requisite exponential variate in *X_N*. The entity also assigns *X_N* to its attribute, called *delay_time*.



Variable - Basic Process

	Name	Rows	Columns	Clear Option	Initial Values	Report Statistics
1	sign			System	1 rows	
2	L			System	1 rows	
3	R			System	1 rows	
4	xi			System	1 rows	
5	lambda			System	1 rows	
6	N			System	1 rows	
7	UP_N			System	1 rows	
8	V_N			System	0 rows	
9	U_N			System	0 rows	
10	US_N			System	0 rows	
11	X_N			System	0 rows	

Figure 10.13 Arena model implementing Algorithms 10.1 and 10.2 for basic TES processes with exponential marginal distributions (top), and its *Variable* module spreadsheet (bottom).

6. At this point the generation of the next TES foreground variate is complete (note that no simulation time has elapsed in the course of any variate generation). The entity immediately proceeds to the *Delay* module, called *Interarrival Time Delay*, where it is delayed for a time interval equal in length to the value of the generated TES foreground variate stored in its *delay_time* attribute.
7. Finally, the entity proceeds to the *Separate* module, called *Duplicate Transaction*, where it spawns precisely one duplicate of itself. The entity itself proceeds to enter the system and to transact its business there. The duplicate entity loops back to the *Assign* module, called *Generate the Rest of Unstitched Variates*, to generate the next TES variate and the corresponding arrival.

Note that the modeler needs to guard against a numerical problem resulting from evaluating the logarithmic value $-\ln(0) = \infty$ in module *Perform Inverse Transform*. This can happen (though quite rarely), because the stitched variable *US_N* in the argument can assume the value 1. One way of handling this problem is to insert a *Decide* module before module *Perform Inverse Transform* and check there the value of *US_N*. When the condition *US_N == 1* is found to hold, the entity is sent back to module *Generate the Rest of Unstitched Variates* to re-sample another TES variate (hopefully one resulting in *US_N* < *1*). However, this solution might result in an infinite loop, say, in a deterministic TES process (this happens, for example when $L = R = 0$, and the interval $[L, R)$ degenerates to the single point 0). A more robust solution is to use the inserted *Decide* module to branch to the current *Perform Inverse Transform* module (if *US_N* < *1*), or to an additional *Assign* module (if *US_N == 1*), in which case a large value is assigned there to *X_N* (in lieu of infinity).

The above discussion illustrates the kind of numerical problems that might be encountered by the modeler, when programming a random number generation scheme. Such numerical problems depend on the underlying distribution, and consequently, their handling must be left to the modeler's discretion.

10.6 EXAMPLE: CORRELATION ANALYSIS IN MANUFACTURING SYSTEMS

Traditional stochastic models of machines subject to failures assume that system uptimes (time to failure) and downtimes (repair time) are mutually independent random variables. Consider the case when failures are bursty (positively autocorrelated), and estimate the corresponding relative errors of mean waiting times when a time-to-failure renewal process replaces a TES process of the same distribution. Assume that job arrivals are Poisson with rate $\lambda = 1.5$; the processing time is fixed at $x = 0.25$ or $x = 0.31$; failures are operation dependent (see Section 11.6) with time to failure being exponentially distributed with rate $\delta = 0.05$; and repair times are random with mean $\bar{y} = 15$ and squared coefficient of variation $\gamma_Y^2 = 1.5$. Assume further that processing resumes on the interrupted job upon repair completion.

In the case where the times to failure are independent, the workstation can be modeled as an *M/G/*1 queueing system where the processing time is, in fact, the process completion time consisting of all the failures experienced by a job on the machine. Consequently, the mean job waiting time is given by the modified Pollaczek–Khintchine (P-K) formula (Altiok [1997]),

$$\mathrm{E}[W_{M/G/1}] = \frac{\lambda\bar{c}^2(\gamma_C^2 + 1)}{2(1 - \lambda\bar{c})}, \tag{10.16}$$

where $\bar{c} = x(1 + \delta\bar{y})$ is the average process completion time, $\gamma_C^2 = \dfrac{x\delta\bar{y}_2}{\bar{c}^2}$ is the squared coefficient of variation of the process completion time, and $\bar{y}_2$ is the second moment of repair times. The probability that the machine is occupied (processing or down) is $\Pr\{B\} = \lambda\bar{c}$, and for stability we assume $\Pr\{B\} < 1$.

Table 10.2 displays the relative deviations

$$d(\rho(1)) = \frac{\mathrm{E}[W_{TES/M/1}] - \mathrm{E}[W_{M/G/1}]}{\mathrm{E}[W_{M/G/1}]}$$

Table 10.2

Relative deviations of mean waiting times in a workstation with failures/repairs

Pr{*B*}	Lag-1 Autocorrelation of Time to Failure, $\rho(1)$							
	0.00	0.14	0.36	0.48	0.60	0.68	0.74	0.99
0.66	15.8	11.4%	51.3%	90.1%	260%	543.7%	3232%	4115%
0.81	36.3	11%	151.3%	229.8%	564.7%	766%	3429%	7800%

as function of $\Pr\{B\}$, where $\Pr\{B\} = 0.66$ for $x = 0.25$, and $\Pr\{B\} = 0.81$ for $x = 0.31$, except for the column under the zero-autocorrelation heading, which displays the corresponding mean waiting times.

Monte Carlo simulations reveal that the relative errors are again nonlinear (Altiok and Melamed [2001]). In particular, at $\rho(1) = 0.74$ the relative deviation exceeds 3000% (one order of magnitude) when $\Pr\{B\} = 0.66$ or $\Pr\{B\} = 0.81$. Incidentally, since the arrival rates are unchanged in both rows, Table 10.2 also provides the relative deviations in the average work-in-progress (WIP) levels in the systems compared, due to Little's formula (see Section 8.2.7). Clearly, careless modeling of failures can lead to large prediction errors for both waiting times and WIP levels.

EXERCISES

1. Let the probability density function (pdf) of the innovation process sequence V be

$$f_V(x) = e^{-x}, \ x \geq 0.$$

   a. Generate a sample path of the iid random sequence $\{V_n\}_{n=1}^{30}$.
   b. Letting $U_0 = 0.1$ and $\xi = 1$ (no stitching), generate a sample path of the corresponding autocorrelated background sequences $\{U_n^+\}_{n=1}^{30}$ and $\{U_n^-\}_{n=1}^{30}$.
   c. Repeat (b) for $\xi = 0.5$, and compare the appearance of the corresponding sample path plots (you can use the graphing options of the Arena *Output Analyzer*).
   d. Using the Arena *Output Analyzer*, estimate the autocorrelation function of each random sequence in 1.b and 1.c for the first five lags.

2. Let the pdf of the innovation process sequence V be

$$f_V(x) = \frac{1}{R - L}, \ L \leq x < \mathrm{R}.$$

   a. Letting $U_0 = 0.6$, generate a sample path of the autocorrelated random sequences $\{U_n^+\}_{n=1}^{30}$ and $\{U_n^-\}_{n=1}^{30}$, for $\xi = 0.5$ and each of the following values of (L, R) pairs:
      - $(L, R) = (-0.1, 0.1)$
      - $(L, R) = (-0.3, 0.3)$
      - $(L, R) = (-0.2, 0.4)$
      - $(L, R) = (-0.3, 0.1)$
      - $(L, R) = (-0.5, 0.5)$.

b. Using the Arena *Output Analyzer*, estimate the autocorrelation function of each random sequence for the first five lags.

3. Consider a workstation processing jobs with a mean processing time of 25 minutes and a squared coefficient of variation of 2.0. Processing times are assumed to be iid gamma distributed. The interarrival times form a TES process with innovation pdf $f_V(x) = \frac{1}{R - L}$, $L \leq x < \mathrm{R}$, and stitching parameter $\xi = 0.5$. The marginal distribution of interarrival times is Unif(20, 60) minutes.

a. Letting $U_0 = 0.5$, generate five TES$^+$ arrival processes with the (L, R) pairs of 2.a, and offer each to the workstation. Run the corresponding simulation model for 24 hours, and estimate the mean system time of jobs. Describe qualitatively the impact of the innovation densities on mean system times.

b. Use the Arena model of Figure 10.13 to generate TES$^-$ processes, and repeat part 3.a using a TES$^-$ arrival process with the same parameters.

c. Compare qualitatively the results of 3.a and 3.b for each (L, R) pair.

This page intentionally left blank

Chapter 11

Modeling Production Lines

Manufacturing facilities in a host of industrial sectors are structured as a series of production stages. Jobs in a variety of forms are transferred from one stage to another to be processed in a prescribed order; these jobs eventually leave the system as finished or semifinished products. Such systems are referred to as *production lines*.

The flexibility afforded by computer-controlled machinery enables production lines to handle a broad range of operations. Various operations permit the deployment of a sequence of intelligent workstations on the shop floor for processing or assembling various products. This chapter discusses simulation modeling of workstation sequences in the production line context.

11.1 PRODUCTION LINES

Consider the representation of a generic manufacturing facility depicted in a rather abstract form in Figure 11.1. The manufacturing facility is a production line composed of manufacturing stages consisting of workstations with intervening buffers to hold product flowing along the line. Each workstation contains one or more machines, one or more operators (possibly robots), and a work-in-process (WIP) buffer. Upon process completion at a workstation, departing jobs join the WIP buffer at the next workstation, provided that space is available; otherwise, such jobs are typically held at the current workstation until space becomes available in the next buffer.

Many practical models may be formulated as variations on the generic production line of Figure 11.1, or with additional wrinkles. For example, a model may call for one or more repetitions of a certain process or a set of processes, or some of the workstations may process jobs in batches. In other cases, the transfer of jobs from one workstation to another is of central importance, so that transportation via vehicles or conveyors is modeled in some detail (see Chapter 13 for a detailed treatment of this subject). Eventually, jobs depart from the system, and such departures can occur, in principle, from any workstation. In general, production lines employ a *push regime*, where little attention is given to the finished-product inventory. The manufacturing line simply produces (pushes) as much product as possible under the assumption that all finished products are to be used. Otherwise, when the accumulation of finished products

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

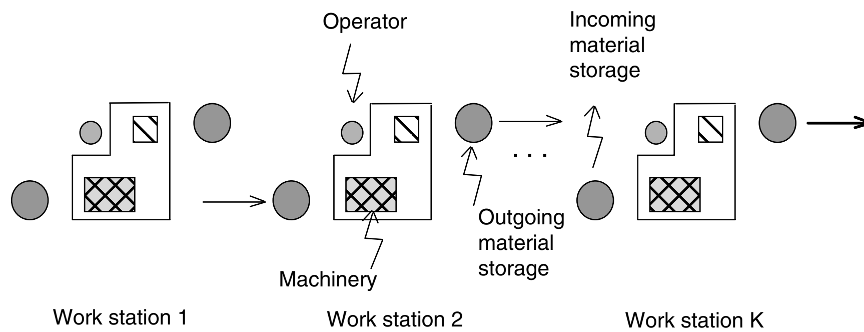


Figure 11.1 A generic production line.

becomes excessive, the manufacturing line may stop producing, at which point the push regime is switched to a *pull regime* (the process only produces in response to specific demands; see the examples in Chapter 12). Push and pull types of manufacturing systems are studied in detail in Altiok (1997) and Buzacott and Shanthikumar (1993).

More generally, storage limitations in workstations give rise to a *bottleneck* phenomenon, involving both *blocking* and *starvation*. The sources of this phenomenon are space limitations and cost considerations, which impose explicit or implicit target levels for storage between stages in a production line. Space limitations (finite buffers) in a downstream workstation can, therefore, cause stoppages at upstream workstations– a phenomenon known as *blocking*. Blocking policies differ on the exact timing of stoppage. One policy calls for an immediate halt of processing of new jobs in the upstream workstation as soon as the downstream buffer becomes full. The upstream workstation is then forced to be idle until the downstream buffer has space for another job, at which point the upstream workstation resumes processing. This type of blocking is often called *communications blocking*, since it is common in communications systems (see Chapter 14). Another policy calls for processing the next job, but holding it (on completion) in the upstream machine until the downstream buffer can accommodate a new job. This type of blocking policy is called *production blocking*, since it often occurs in manufacturing context. Various types of blocking mechanisms are discussed in detail in Perros (1994).

It is important to realize that blocking tends to propagate backwards to successive workstations located upstream in the production line. Similarly, some workstations may experience idleness due to lack of job flow from upstream workstations. This phenomenon is called *starvation* for obvious reasons. It is further important to realize that starvation tends to propagate forward to successive workstations located downstream in the production line. Blocking and starvation are, in fact, the flip sides of a common phenomenon and tend to occur together—a *bottleneck workstation* can be identified, which separates a production line into two segments, such that upstream workstations experience frequent blocking and downstream workstations experience frequent starvation.

Another type of (forced) idleness in production lines is caused by failures. Usually, a failed workstation is taken in for repair as soon as possible and resumes operation once repair is completed; the alternating periods of operation and failure are referred to as *uptimes* and *downtimes*, respectively. Clearly, a failed workstation can become a bottleneck workstation, causing blocking in upstream workstations and starvation in

downstream workstations. In the presence of failures, one needs to devise procedures for handling interrupted jobs (those being processed when a failure strikes). For instance, an interrupted job may need to be reprocessed from scratch or may merely need to resume processing. In some cases, however, the interrupted job is simply discarded.

11.2 MODELS OF PRODUCTION LINES

Productivity losses are potentially incurred whenever machines are idle (blocked or starved) due to machine failures or bottlenecks stemming from excessive accumulation of inventories between workstations. Furthermore, production lines are rarely deterministic; their randomness is due to variable processing times, as well as random failures and subsequent repairs. Such randomness makes it difficult to control these systems or to predict their behavior. A mathematical model or a simulation model is then used to make such predictions.

Design problems in production lines are primarily resource allocation problems. These problems include *workload allocation* and *buffer capacity allocation* for a given set of workstations with associated processing times. Generally, design problems are quite difficult to solve in manufacturing systems. This is due in part to the combinatorial nature of such problems.

Performance analysis of production lines strives to evaluate their performance measures as function of a set of system parameters. The most commonly used performance measures follow:

- Throughput
- Average inventory levels in buffers
- Downtime probabilities
- Blocking probabilities at bottleneck workstations
- Average system flow times (also called manufacturing *lead times*)

Using these measures to analyze manufacturing systems can reveal better designs by identifying areas where loss of productivity is most harmful. For a thorough coverage of design, planning, and scheduling problems in production and inventory systems, see Altiok (1997), Askin and Standridge (1993), Buzacott and Shanthikumar (1993), Elsayed and Boucher (1985), Gershwin (1994), Johnson and Montgomery (1974), and Papadopoulos et al. (1993), among others.

11.3 EXAMPLE: A PACKAGING LINE

Consider a generic packaging line for some product, such as a pharmaceutical plant producing a packaged medicinal product, or a food processing plant producing packaged foods or beverages. The line consists of workstations that perform the processes of *filling*, *capping*, *labeling*, *sealing*, and *carton packing*. Individual product units will be referred to simply as *units*.

We make the following assumptions:

1. The filling workstation always has material in front of it, so that it never starves.
2. The buffer space between workstations can hold at most five units.

3. A workstation gets blocked if there is no space in the immediate downstream buffer (manufacturing blocking).
4. The processing times for filling, capping, labeling, sealing, and carton packing are 6.5, 5, 8, 5, and 6 seconds, respectively.

Note that these assumptions render our packaging line a push-regime production line. To keep matters simple, no randomness has been introduced into the system, that is, our packaging line is deterministic.

It is worthwhile to elaborate and analyze the behavior of the packaging line under study. The first workstation (filling) drives the system in that it feeds all downstream workstations with units. Clearly, one of the workstations in the line is the slowest (if there are several slowest workstations, we take the first among them). The throughput (output rate) of that workstation then coincides with the throughput of the entire packaging line. Furthermore, workstations upstream of the slowest one will experience excessive buildup of WIP inventory in their buffers. In contrast, workstations downstream of the slowest one will always have lightly occupied or empty WIP inventory buffers. Thus, the slowest workstation acts as a bottleneck in our packaging line. Of course, this behavior holds for any deterministic push-regime production line.

11.3.1 An Arena Model

Figure 11.2 depicts an Arena model for the packaging line, in which Arena modules represent each process and units are transferred from process to process. The model logic strives to maintain a sufficient number of units (Arena entities) in the filling station, so as to keep it busy for as long as possible. To achieve this goal, a total of 30 unit entities are created at time 0, and their arrival time attribute, *ArrTime*, is assigned the current value of *TNOW* in the *Assign Arrival Times* module. These are promptly fed into the filling process buffer (still at time 0). Prior to exiting the system from the *Tally* module, called *Interdeparture Time*, unit entities are not disposed of, but are sent back to the *Assign Arrival Times* module. Note carefully that this is merely a modeling device to avoid starvation in the filling station, in compliance with the first assumption above. Starting with the filling process, unit entities go through the modules associated with the filling, capping, labeling, sealing, and packing processes in that order, and finally proceed to statistics collection modules. The model logic will be discussed in some detail in the next section.

11.3.2 Manufacturing Process Modules

Each production line process is modeled in Figure 11.2 by a *Process* module from the *Basic Process* template panel. Every process was declared to have a queue in front of it, as evidenced by the T-bar graphics in Figure 11.2. Recall that queue parameters can be examined in spreadsheet view in the *Queue* module from the *Basic Process* template panel. As an example, consider the *Process* module called *Filling Process*, whose dialog box is displayed in Figure 11.3. This *Process* module has a queue called *Filling Process.Queue* in front of a resource called *Filler*. The *Action* field in the *Filling Process* dialog box is *Seize Delay*, so that unit entities wait in the queue *Filling Process.Queue* whenever resource *Filler* is busy. As soon as resource *Filler* becomes available, it is

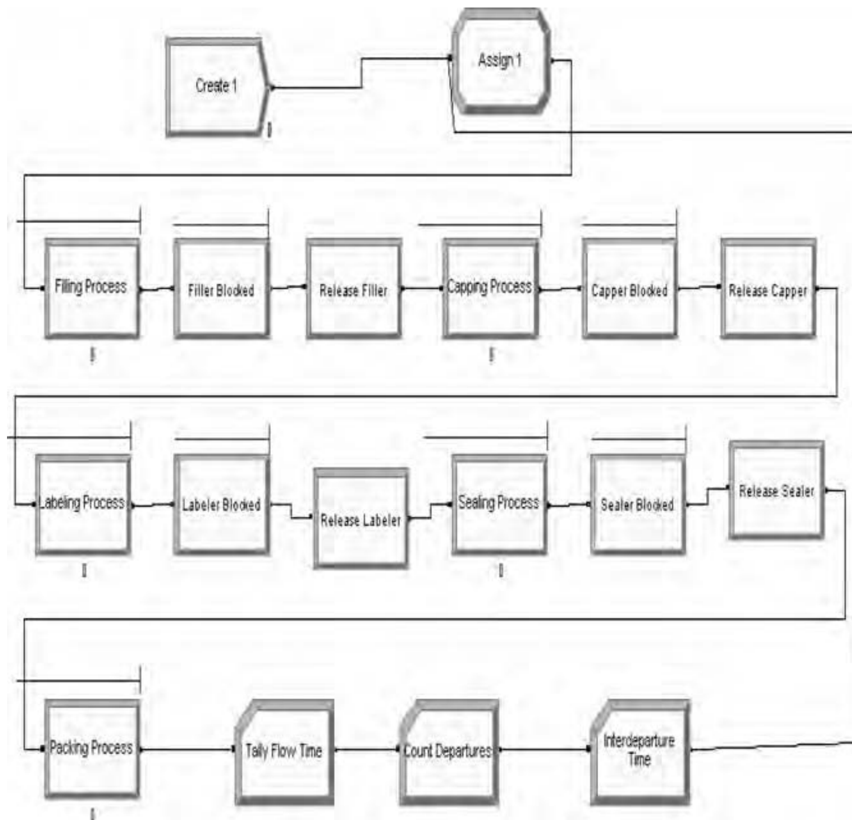


Figure 11.2 Arena model for the packaging line system.

seized by the first unit entity (if any) in the queue *Filling Process.Queue*, and a processing time delay of 6.5 seconds is then implemented in the associated *DELAY* block.

On processing completion, the unit entity proceeds to the next *Hold* module, called *Filler Blocked*, which implements blocking as necessary (this is explained in detail in Section 11.3.3). Only when space is available in the capping process that follows the filling process, will the unit entity be permitted to proceed to the *Release* module, called *Release Filler*, where resource *Filler* is released, thereby completing the filling process. In fact, the sequence of *Process*, *Hold*, and *Release* modules is repeated for each of the first four processes (the fifth and last process, *Packing Process*, does not experience blocking).

11.3.3 MODEL BLOCKING USING THE *HOLD* MODULE

Recall that the *Hold* module is used to hold entities in an associated queue until a condition is satisfied or until a future event takes place. In particular, such modules may be used to implement production blocking.

The Arena model of Figure 11.2 employs four *Hold* modules to this end (observe the corresponding queue T-bar graphics in Figure 11.2). As an example, Figure 11.4 displays the dialog box for the *Hold* module, called *Filler Blocked*, associated with the *Filling Process* module.

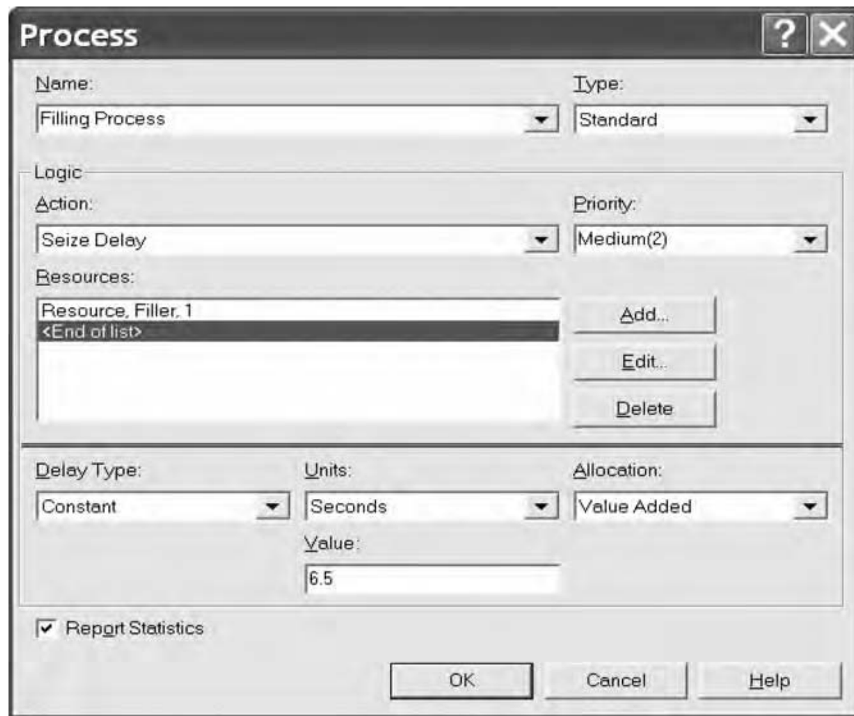


Figure 11.3 Dialog box of the *Process* module *Filling Process*.

The *Condition* field in the *Hold* module dialog box was set to scan for the condition

$$NQ(Capping\ Process.Queue) < 5.$$

In other words, it tests whether buffer space is available in the capping process—the manufacturing stage that immediately follows the filling process. If this condition is true, the corresponding unit entity leaves the filling process and joins the capping buffer. Otherwise, that unit entity becomes blocked and remains in the *Hold* module, or more specifically, in the associated queue (in our example, *Filler Blocked.Queue*) until the condition is satisfied. As soon as the unit entity leaves the *Hold* module, it enters the *Release* module, called *Release Filler*, and joins queue *Capping Process.Queue*. Clearly, the resource *Filler* must perforce be busy whenever a unit entity resides in the queue *Filler Blocked.Queue*. Furthermore, since the capacity of resource *Filler* was declared to be one unit, there can be at most one unit entity in the queue *Filler Blocked.Queue* at any point in time. It follows that

$$\Pr\{NQ(Filler\ Blocked.Queue) > 0\} = \frac{\text{Time } Filler\ Blocked.Queue \text{ is occupied}}{\text{Total simulation time}},$$

which also happens to be the average number of unit entities in *Filler Blocked.Queue*. This fact will later be used in obtaining blocking probabilities for any process *some_process* simply by defining (in the *Statistic* module) a *Time Persistent* statistic for the expression *NQ(some_process.Queue)*.

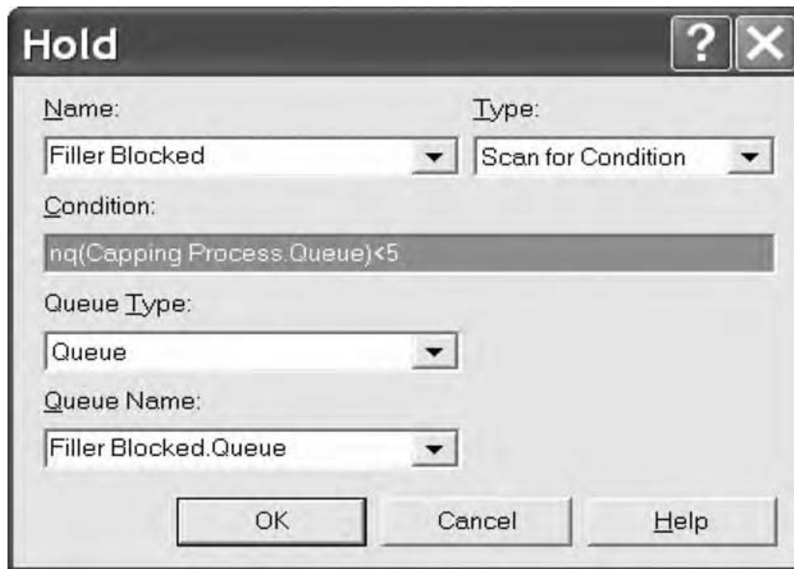


Figure 11.4 Dialog box of the *Hold* module associated with the filling process.

Note that a *Release* module follows each *Hold* module for each process. Consequently, the time spent by a unit entity in the *Hold* module's blocking queue is measured by Arena as part of the preceding *Process* module's resource busy time. This point will be revisited in Section 11.3.6 when we study the model's effective process utilizations.

11.3.4 Resources and Queues

Recall that the *Resource* module in the *Basic Process* template panel provides a spreadsheet view of all resources in the model, as illustrated in Figure 11.5.

Similarly, the *Queue* module provides a spreadsheet view of all queues in the model, including blocking queues, as illustrated in Figure 11.6.

Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Failures	Report Statistics
1	Filler	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
2	Capper	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
3	Packer	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
4	Sealer	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
5	Labeler	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑

Figure 11.5 Dialog spreadsheet of the *Resource* module.

Queue - Basic Process

	Name	Type	Shared	Report Statistics
1	Filling Process.Queue	First In First Out	☐	☑
2	Capping Process.Queue	First In First Out	☐	☑
3	Sealing Process.Queue	First In First Out	☐	☑
4	Packing Process.Queue	First In First Out	☐	☑
5	Capper Blocked.Queue	First In First Out	☐	☑
6	Sealer Blocked.Queue	First In First Out	☐	☑
7	Labeling Process.Queue	First In First Out	☐	☑
8	Labeler Blocked.Queue	First In First Out	☐	☑
9	Filler Blocked.Queue	First In First Out	☐	☑

Figure 11.6 Dialog spreadsheet of the *Queue* module.

The column under the *Type* heading in Figure 11.6 is used to specify a service discipline for each queue via an option from a pull-down menu in each selected entry. Service discipline options are *FIFO*, *LIFO*, *Lowest Attribute Value*, and *Highest Attribute Value*. The last two options are implemented using a particular attribute of unit entities in the queue. A common example is priority-based service using a *Priority* attribute in entities. Another example is the *SPT* rule, which admits into service the job with the shortest processing time. An implementation of this rule would require the modeler to declare an attribute, say *Processing_Time*, and to select the options *Lowest Attribute Value* or *Highest Attribute Value* in the *Type* column. This selection would automatically create an additional column in Figure 11.6, labeled *Attribute Name*, in which the user would enter the appropriate attribute name (in our case, *Processing_Time*). Modeling a finite capacity for a queue can be implemented analogously to blocking (see Section 11.3.3). Entries in the column under the *Shared* heading are checked if a queue is shared by more than one *Seize* module or *SEIZE* block.

11.3.5 Statistics Collection

Figure 11.7 displays the dialog box of the *Record* module, called *Tally Flow Time*, which tallies the time elapsed since the time stored in the unit entity attribute *ArrTime*. In particular, if we arrange for *ArrTime* to be a unit entity attribute that stores its arrival time in the system and set the *Type* field to the *Time Interval* option, and if unit entities proceed from the *Record* module *Tally Flow Time* to be disposed of, then this *Record* module will tally unit entity flow times through the system.

Figure 11.8 displays the dialog spreadsheet of the *Statistic* spreadsheet module for the packaging line model. Rows 1–4 and row 9 collect *Frequency* statistics (steady-state probability estimates) for the states of the filling, capping, sealing, packing, and labeling resources, respectively. The probability of any prescribed event can be similarly specified via an expression, provided that the *Type* field is set to the *Time Persistent* option. For example, the probability that the filling resource queue has two or more jobs in its buffer can be specified by the expression.

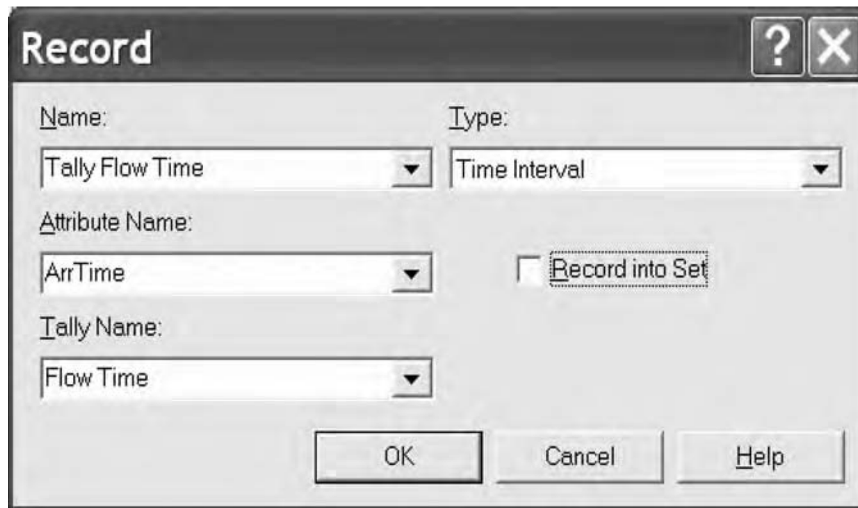


Figure 11.7 Dialog box of the *Record* module *Tally Flow Time*.

$$NQ\ (Filling_R_Q) >= 2.$$

Row 5 collects an *Output* type statistic specified by an expression that defines the reciprocal of the time average of the variable, called *Interdeparture Time*, which computes interdeparture times of unit entities from the system. Recalling that the expression is computed *after* the replication terminates, it follows that this *Output* statistic provides an estimate of the system throughput.

Finally, rows 6 through 10 collect blocking probabilities of the filling, capping, sealing, and labeling processes, respectively (recall that the packing process does not experience blocking). Note that each associated expression collects a time average of the size of the blocking queue in the corresponding *Hold* module. Since each blocking queue can hold at most one job, the computed time average is just the respective blocking probability.

11.3.6 Simulation Output Reports

The packaging line model was simulated for 100,000 seconds. Figure 11.9 displays the *Queues* report of the *Reports* panel for this model.

Note that each queue has *Waiting Time* and *Number Waiting* statistics, including the blocking queues of the *Hold* modules. The queue *Filling Process.Queue* had a maximum of 29 units (recall that 30 unit entities were placed there initially), and an average of 15.64 units. The queue *Capping Process.Queue* was nearly full most of the time, holding an average of 4.96 units with an average delay of 39.9 seconds. The labeling queue *Labeling Process.Queue* was also full most of the time holding an average of 4.99 units. In contrast, the sealing and packing queues were empty all the time. The *Queues* report also includes hold (blocking) queue statistics. These will be discussed, however, later in the context of blocking probabilities.

Statistic - Advanced Process

	Name	Type	Expression	Report Label	Output File	Frequency Type	Resource Name	Report Label
1	Filler States	Frequency		Filler States		State	Filler	Filler States
2	Capper States	Frequency		Capper States		State	Capper	Capper States
3	Sealer States	Frequency		Sealer States		State	Sealer	Sealer States
4	Packer States	Frequency		Packer States		State	Packer	Packer States
5	Throughput	Output	1/TAVG(Interdeparture Time)	Throughput		Value		Throughput
6	Filler is Blocked	Time-Persistent	nq(Filler Blocked.Queue)	Filler is Blocked		Value		Filler is Blocked
7	Capper is Blocked	Time-Persistent	nq(Capper Blocked.Queue)	Capper is Blocked		Value		Capper is Blocked
8	Sealer is Blocked	Time-Persistent	nq(Sealer Blocked.Queue)	Sealer is Blocked		Value		Sealer is Blocked
9	Labeler States	Frequency		Labeler States		State	Labeler	Labeler States
10	Labeler is Blocked	Time-Persistent	nq(Labeler Blocked.Queue)	Labeler is Blocked		Value		Labeler is Blocked

Figure 11.8 Dialog spreadsheet of the *Statistic* module.

Figure 11.10 displays the *Resources* report of the *Reports* panel for the packaging line model. The *Inst Util* column in the report shows that the resources *Capper*, *Filler*, and *Labeler* were 100% occupied. This is expected of the *Filler* resource, since the system was designed that way. The *Capper* and *Filler* resources did not experience idleness, while the *Sealer* and *Packer* resources were busy with 75% and 62% utilization, respectively. The *Resources* report has additional detailed statistics similar to the *Queues* report, which are not shown here.

Figure 11.11 displays the *User Specified* report of the *Reports* panel for the packaging line model. Recall that these statistics are collected as a result of declarations in the

4:02:22PM **Queues** February 10, 2006

Packaging Line Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Seconds

Capper Blocked.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	2.9997	(Correlated)	0.5000	3.0000

Other	Average	Half Width	Minimum	Maximum
Number Waiting	0.3743	(Correlated)	0	1.0000

Capping Process.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	39.8641	(Correlated)	0	40.0000

Other	Average	Half Width	Minimum	Maximum
Number Waiting	4.9860	(Correlated)	0	5.0000

Filler Blocked.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	1.5000	(Correlated)	1.0000	1.5000

Other	Average	Half Width	Minimum	Maximum
Number Waiting	0.1868	(Correlated)	0	1.0000

Filling Process.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	124.97	(Correlated)	0	188.50

Other	Average	Half Width	Minimum	Maximum
Number Waiting	15.6448	(Correlated)	0	29.0000

4:02:22PM	Queues			February 10, 2006
Packaging Line				Replications 1
Replication 1	Start Time 0.00	Stop Time 100,000.00		Time Units Seconds
Labeler Blocked.Queue				
Other	Average	Half Width	Minimum	Maximum
Number Waiting	0	(Insufficient)	0	0
Labeling Process.Queue				
Time	Average	Half Width	Minimum	Maximum
Waiting Time	39.9557	(Correlated)	0	40.0000
Other	Average	Half Width	Minimum	Maximum
Number Waiting	4.9951	(Correlated)	0	5.0000
Packing Process.Queue				
Time	Average	Half Width	Minimum	Maximum
Waiting Time	0	0.000000000	0	0
Other	Average	Half Width	Minimum	Maximum
Number Waiting	0	(Insufficient)	0	0
Sealer Blocked.Queue				
Other	Average	Half Width	Minimum	Maximum
Number Waiting	0	(Insufficient)	0	0
Sealing Process.Queue				
Time	Average	Half Width	Minimum	Maximum
Waiting Time	0	0.000000000	0	0
Other	Average	Half Width	Minimum	Maximum
Number Waiting	0	(Insufficient)	0	0

Figure 11.9 *Queues* report for the packaging line model.

Statistic and *Record* modules. The *Tally* section in Figure 11.11 provides interdeparture time and flow-time statistics. The mean interdeparture time for the run was estimated at 8 seconds (with all observations having the same value, 8). In spite of its apparent triviality, this statistic contributes to the verification of the simulation model—a topic that will be treated in more detail in Section 11.4. The mean flow time from the *Filling*

4:10:32PM	Resources	February 10, 2006

Packaging Line Replications 1

Replication 1 Start Time 0.00 Stop Time 100,000.00 Time Units Seconds

Resource Detail Summary

Usage

	Inst Util	Num Busy	Num Sched	Num Seized	Sched Util
Capper	1.00	1.00	1.00	12,505.00	1.00
Filler	1.00	1.00	1.00	12,511.00	1.00
Labeler	1.00	1.00	1.00	12,499.00	1.00
Packer	0.75	0.75	1.00	12,497.00	0.75
Sealer	0.62	0.62	1.00	12,498.00	0.62

Figure 11.10 *Resources* report for the packaging line model.

Process module to the *Interdeparture Time* module was estimated at 239.78 seconds, based on 12,497 observations (unit entity departures). The minimal and maximal flow times observed were 30.5 seconds and 262.5 seconds, respectively. Note that the mean flow time is closer to the maximal rather than the minimal observation, showing that the majority of flow-time observations were skewed toward 262.5 seconds.

The *Counter* section indicates that a total of 12,497 departures from the system were observed over the replication interval.

The *Time Persistent* section provides estimates for blocking probabilities. Evidently, the system experienced acute blocking in *Capping Process* (blocking occurred 37% of the time) due to long labeling times. Blocking at *Capping Process* has propagated upstream to *Filling Process*, which experiences 18.7% blocking.

The *Output* section estimates the system throughput as 0.125 units per second (equivalently, one unit every 8 seconds), in agreement with the mean interdeparture time in the *Tally* section.

Figure 11.12 displays the *Frequencies* statistics of the *Reports* panel for the first replication of the packaging line model. The displayed statistics pertain to the probabilities of the *busy* and *idle* states for each resource. An immediate observation is that the estimated *busy* probabilities for the resources *Filler*, *Labeler*, *Packer*, and *Sealer* include the corresponding blocking probabilities, since a blocked resource is not released, and therefore, is considered *busy*. Thus, in order to estimate the effective utilization of a resource (the fraction of time that it performs "actual work"), its estimated blocking probability is subtracted from its estimated probability in the *busy* state. Table 11.1 displays the effective utilization of each process resource.

4:12:56PM **User Specified** February 10, 2006

Packaging Line Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Seconds

Tally

Between	Average	Half Width	Minimum	Maximum
Interdeparture Time	8.0000	0.000000000	8.0000	8.0000

Interval	Average	Half Width	Minimum	Maximum
Flow Time	239.78	(Correlated)	30.5000	262.50

Counter

Count	Value
Number of Finished Units	12,497.00

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Capper is Blocked	0.3743	(Correlated)	0	1.0000
Filler is Blocked	0.1868	(Correlated)	0	1.0000
Labeler is Blocked	0	(Insufficient)	0	0
Sealer is Blocked	0	(Insufficient)	0	0

Output

Output	Value
Throughput	0.1250

Figure 11.11 *User Specified* report for the packaging line model.

4:23:37PM		Frequencies		February 10, 2006
Packaging Line				Replications: 1
Replication 1	Start Time: 0.00	Stop Time: 100,000.00		Time Units: Seconds
Capper States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	29	3,446.62	99.95	99.95
IDLE	29	1.6552	0.05	0.05
Filler States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	1	100,000.00	100.00	100.00
Labeler States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	1	99,988.50	99.99	99.99
IDLE	1	11.5000	0.01	0.01
Packer States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	12,497	6.0000	74.98	74.98
IDLE	12,498	2.0018	25.02	25.02
Sealer States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	12,498	5.0000	62.49	62.49
IDLE	12,498	3.0013	37.51	37.51

Figure 11.12 *Frequencies* report for the packaging line model.

Table 11.1

Effective process utilization from *Filling* to *Packing*

Process	*Filling*	*Capping*	*Labeling*	*Sealing*	*Packing*
Utilization	0.813	0.625	0.999	0.625	0.750

11.4 UNDERSTANDING SYSTEM BEHAVIOR AND MODEL VERIFICATION

The generic packaging line model of Section 11.3 may be abstracted as a sequence of workstations (or servers) having fixed processing times per product unit, and with the proviso that the first workstation is never idle. Thus, the throughput of the system (production rate) coincides with that of the slowest workstation in the sequence.

In the example of Section 11.3, the labeling workstation (resource *Labeler*) is the slowest, with 8 seconds of unit processing time. Therefore, the utilization of the labeling workstation is expected to be 100%, since all upstream workstations have shorter (fixed) unit processing times. Since the filling and capping workstations are both faster than the labeling workstation, the labeling buffer is sure to fill up eventually, thereby blocking the capping workstation and later on the filling workstation, thereby giving rise to significant blocking probabilities in these workstations. Furthermore, every departure from the labeling workstation finds the downstream workstations (sealing

and carton packing) in the *idle* state, since they too have shorter processing times than the labeling workstation. In consequence, the labeling and sealing workstations are never blocked. Accordingly, a product unit departs from the labeling workstation every 8 seconds—which is also the interdeparture time in downstream machines—resulting in system throughput of $1/8 = 0.125$ units per second. In fact, the throughput of *each* workstation in the system should also be 0.125. Indeed, using Eq. 8.7, which expresses the throughput $\bar{o}$ as the ratio of utilization to average unit processing time, the throughput of the filling workstation is $\bar{o} = 0.813 / 6.5 = 0.125$, while the throughput of the capping workstation is $\bar{o} = 0.625 / 5 = 0.125$.

As one facet of simulation model verification, we next verify that the average flow time and the average number of product units obeys Little's formula (8.8), which is further discussed in Section 11.10. Define the subsystem S to be the sequence of processes from the *Filling Process* module to and including the *Packing Process* module (the raw material storage in front of *Filling Process* is included in S). The average flow time through S is measured from the arrival of a unit entity at queue *Filling Process.Queue* to the unit entity's arrival at module *Interdeparture Time*. The replication estimated this average flow time to be 239.79 seconds (see Figure 11.10). The average number of units in S, $\bar{N}_s$, is just the sum of the corresponding average buffer contents (see Figure 11.8) plus the average number of units in service (average time in busy state; see Figure 11.11), that is,

$$\bar{N}_s = 15.6448 + 4.9860 + 4.9551 + 1.0 + 0.9995 + 0.9999 + 0.6249 + 0.7498 = 29.66, \quad (11.1)$$

which is close to the theoretical expected value 30 (note that this is a closed system with 30 circulating unit entities). Thus, in this case, Little's formula becomes the relation

$$\bar{N}_s = \bar{o}\bar{F}_s \quad (11.2)$$

where $\bar{N}_s$ is the average number of units in S, $\bar{o}$ is the throughput of S as discussed above (the throughput of S equals the arrival rate at S in steady state), and $\bar{F}_s$ is the average flow time through S. Using the average flow time and the throughput in Figure 11.11 to calculate the right side of Eq. 11.2 yields

$$239.78 \times 0.125 = 29.9725. \quad (11.3)$$

Because the right sides of Eqs. 11.1 and 11.3 are approximately the same, this fact tends to verify the replication's estimates. In fact, the discrepancy between Eqs. 11.1 and 11.3 becomes progressively smaller, as the replication run length increases.

Next, we perform sensitivity analysis by studying the impact of buffer capacities on system behavior. Such analysis will aid us in gaining insight into blocking—a phenomenon stemming from mismatch in the service rates and finite buffer capacities. Figure 11.13 displays three performance measures as functions of simultaneously increasing buffer capacities in *Labeling Process* and *Capping Process*. From left to right, the plotted measures are the throughput of S, the WIP level at *Labeling Process*, and the average flow time through S, excluding the *Filling* buffer. (If flow times were to include time in the *Filling* buffer, then they would remain constant. Why?)

Interestingly, the throughput of S is largely unaffected, while the average WIP level at *Labeling Process* and the mean flow time through S increase as the buffer capacities are increased. To understand these observations, recall that the system under study has fixed processing times. Recall further that buffers are placed in production lines in order

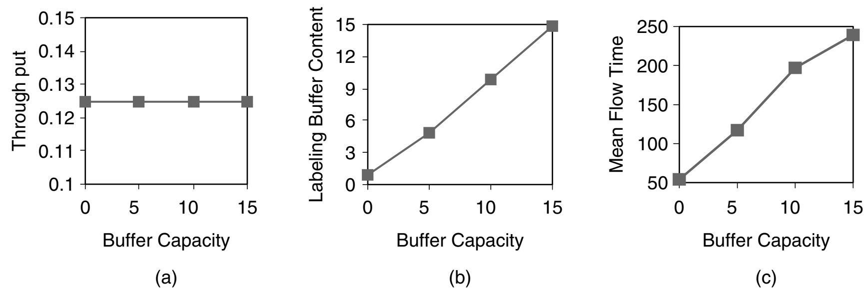


Figure 11.13 Impact of simultaneously increasing the capping and labeling buffer capacities on (a) throughput of *S*, (b) average number in labeling buffer, and (c) average flow time.

to absorb product flow fluctuations due to randomness in the system (e.g., random processing times, random downtimes, etc.). Such events introduce variability that tends to slow down product movement. But in systems with no variability (such as ours), the placement of buffers would have no impact on the throughput, as evidenced by Figure 11.13(a). However, placing larger buffers upstream of bottleneck workstations simply gives rise to larger WIP levels there, and consequently, to longer flow times. In contrast, buffers at workstations downstream of the bottleneck are always empty, regardless of capacity.

To summarize, our sensitivity analysis has revealed some valuable and somewhat unexpected design principles for deterministic production lines, namely, that *larger buffers actually can have an overall deleterious effect* on the system. On the one hand, increasing buffer sizes will not increase the throughput. On the other hand, such increases will result in larger on-hand inventories, thereby tying up precious capital in inventory. Worse still, it would take jobs longer to traverse the system. These effects can be economically detrimental (think of the penalty of longer manufacturing lead times). Reasonably sized buffers are necessary, however, to absorb the effects of variability in product flow, in the presence of randomness. In Section 11.6, we will discuss the beneficial effect of placing buffers when machine failures and repairs are introduced.

11.5 MODELING PRODUCTION LINES VIA INDEXED QUEUES AND RESOURCES

When models contain components with repetitive logic, then model construction can become tedious, and consequently, error-prone. A case in point is the packaging line model of Section 11.3, where component processes have analogous logic (although the parameters are different). More specifically, in each component process (*filling*, *capping*, *labeling*, *sealing*, and *carton packing*), units seize a resource, get processed, check if blocking occurs, and then release the corresponding resource. For models with such repetitive logic, Arena supports an efficient approach, called *indexing*, which largely eliminates the tedium of repetition. This approach exploits the fact that each object in Arena has an implicit integer ID number within its class; examples of object classes include such Arena constructs as queues, resources, and expressions. Thus, objects within a class can be viewed as object arrays, where individual objects are assigned an index for access and manipulation. An *index* is an integer that identifies the serial

number of the object in the array, and as such it varies from 1 to the size of the array (the number of objects created). Note that the array size is dynamic, and the modeler can enforce any particular order of objects in the array to facilitate indexing logic. To this end, the modeler can use the *Queues*, *Resources*, and *Expressions* elements from the *Elements* template panel, accessible from the *Template Attach* button in the *Standard* toolbar.

To illustrate the use of indexing in modeling repetitive components, we will modify the packaging line example of Section 11.3. Exploiting the Arena indexing feature, we first number all resources from 1 to 5, all resource queues from 1 to 5, and all blocking queues from 6 to 10, and then declare these objects and their indexes using the old Arena elements of *Resources* and *Queues*.

Figure 11.14 depicts the dialog box of the element called *Queues Element* (left) and the associated dialog box for a particular queue (right). Initially, *Queues Element* on the left is empty. The modeler clicks the *Add* button to pop up the dialog box on the right to define the relevant queue attributes for the queue selected in the *Name* field (here *Filler Queue*). The most important fields in the *Queues* dialog box are the index of the queue in the Arena queue array (specified in the *Number* field), and the queueing discipline of the queue (specified in the *Ranking Criterion* field). In addition, the modeler can select one of two types of specifications in the *Associated Block* field: either specify a (SIMAN) block with which the present queue will be associated, or enter the keyword *shared*, thereby signaling that the queue will be used in multiple *Hold* and *Seize* modules.

The assignment of user-specified index numbers to resources is similar. Figure 11.15 depicts the dialog box of the element called *Resources Element* (top) and the associated dialog box for a particular resource (bottom). Note that the *Resources* dialog box is handy for specifying additional information for a resource (in this case resource *Filler*). In addition to the resource index and name, the modeler can use the *Capacity or Schedule* field to enter either a static resource capacity or a schedule name governing a dynamic time-dependent capacity. In Figure 11.15, we use the *Resources* dialog box merely to assign an index number to resource *Filler*.

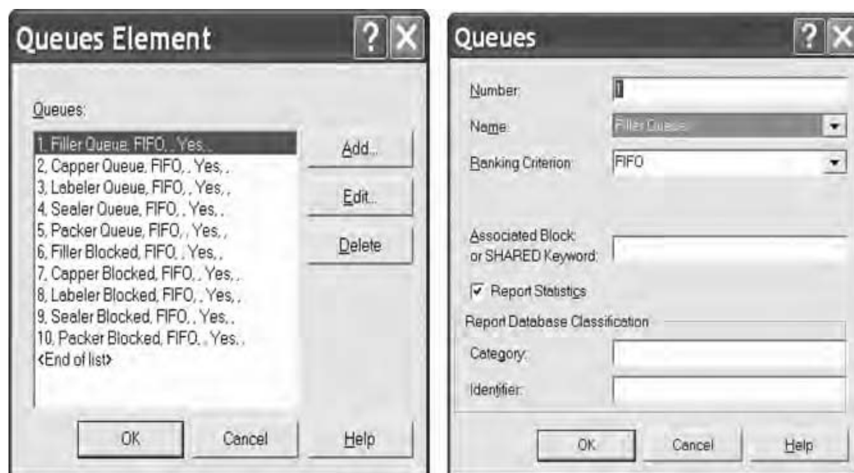


Figure 11.14 Dialog boxes of *Queues Element* (left) and *Queues* (right).

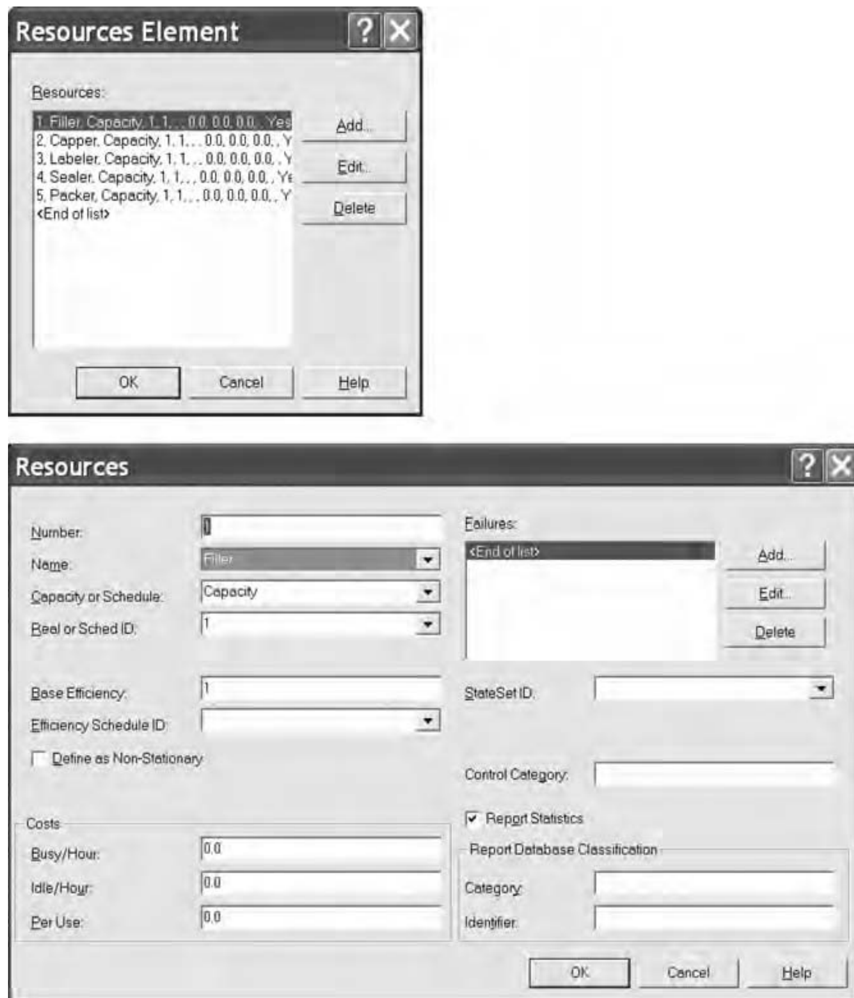


Figure 11.15 Dialog boxes of *Resources Element* (top) and *Resources* (bottom).

Finally, indexing of Arena expressions in the packaging line is illustrated in Figure 11.16, which depicts the dialog box of the element called *Expressions Element* (left) and the associated dialog box for a particular expression (right). Expression arrays may be vectors (one-dimensional) or matrices (two-dimensional). To specify a vector, the user enters its dimension in the *1-D Array Index* field. Similarly, to specify a matrix, the user enters its dimensions in the *1-D Array Index* and *2-D Array Index* fields. The actual elements of the expression array are entered in sequence column by column, one expression per line. Here, the entered expressions are constants that specify processing times and buffer capacities.

The equivalent version of the Arena model of Section 11.3, modified to incorporate indexing, is depicted in Figure 11.17, where all indexed queues, expressions, and resources are shown in this order under the corresponding headings on the right side of the figure. Starting at the upper left corner, the *Create* module, called *Create Jobs*, operates exactly as module *Create 1*, its counterpart in Figure 11.2 (the "jobs" alluded to

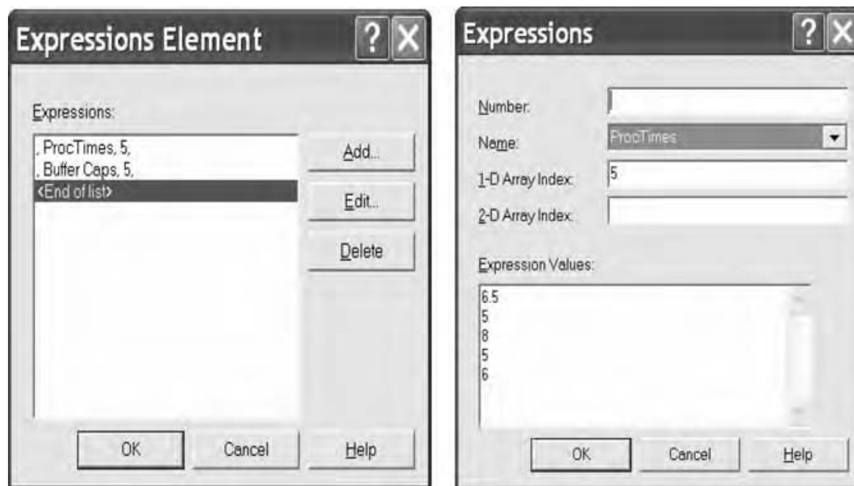


Figure 11.16 Dialog boxes of *Expressions Element* (left) and *Expressions* (right).

are, of course, product units). This module generates 30 unit entities at time 0, which circulate repeatedly in the model. However, the logic of routing product units through the packaging line workstations is implemented in a manner analogous to a *loop* in a procedural programming language. In our case, the unit entity attribute *Proc_Index* keeps track of the loop's running index.

When a unit entity emerges from module *Create Jobs*, it proceeds to the *Assign* module, called *Assign Process Index and Time*, where its *Proc_Index* attribute is initialized to 1 (as well as saving the current simulation time in attribute *Arr_Time* for

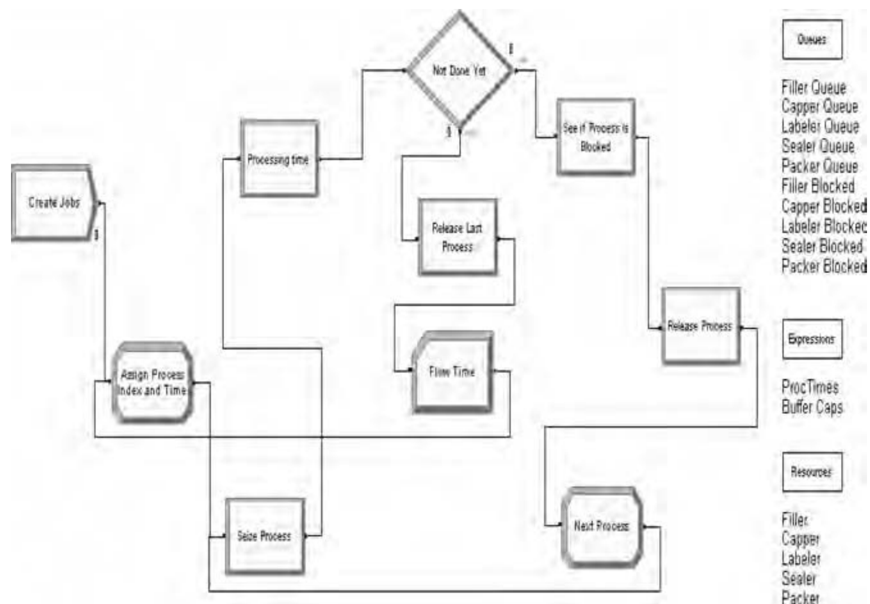


Figure 11.17 Arena model with indexing for the packaging line system.

later collection of total flow-time observations (at the last workstation implementing the packing process). Note that the packing line workstations model the resources *Filler*, *Capper*, *Labeler*, *Sealer*, and *Packer*, and these are indexed by 1, 2, 3, 4, and 5, respectively, as indicated by the list of resources under the *Resources* heading in Figure 11.17; the corresponding resource queues are indexed in the same way, as indicated by the list of queues under the *Queues* heading. Each unit entity will next be routed through the same sequence of modules, and its *Proc_Index* attribute will be incremented after each iteration so as to correspond to the next workstation to be visited. As will be seen, the appropriate service distribution for each workstation process will also be selected by this attribute.

The unit entity proceeds to the first module in the loop, namely, the *Seize* module called *Seize Process*. The dialog box for this module is displayed in Figure 11.18. Here, the *Resources* field indicates that the resource to be currently selected for seizing is specified by the current index number stored in attribute *Proc_Index* (initially the index is 1, which corresponds to resource *Filler*). The dialog box for the current resource in the loop is shown in Figure 11.19. Note that in this dialog box, the appropriate resource is selected by attribute (in the *Type* field), and the *Attribute Name* field specifies attribute *Proc_Index*.

Having seized the selected resource, the unit entity proceeds to the second module in the loop, namely, the *Delay* module called *Processing Time*. The dialog box for this module is displayed in Figure 11.20. Here, the *Delay Time* field indicates that a delay time for this module is the value at the current index position (saved in attribute *Proc_Index*) of the *ProcTimes* expression vector (recall that this expression vector was initialized previously, as illustrated in Figure 11.16).

Once the unit entity completes its delay and emerges from the *Processing Time* module, its processing at the current workstation is also complete. It now needs to make the following decision:

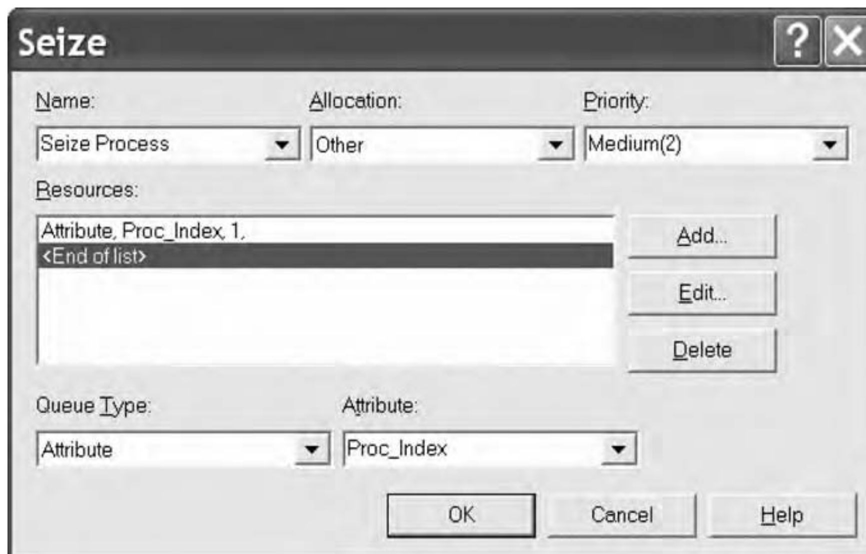


Figure 11.18 Dialog box of the first module (*Seize*) in the loop.

Figure 11.19 Dialog box of the current resource in the loop.

1. If this workstation (process) is not the last in the sequence, then proceed to the next workstation in the loop. Since the last process (packing) has index number 5, this is equivalent to checking whether *Proc_Index* < *5*.
2. Otherwise (i.e., *Proc_Index* == *5*), proceed to collect the system flow time (sojourn time in all five workstations) for this product unit, and then restart another sequence of processes.

To this end, the unit entity enters the third module in the loop, namely, the *Decide* module called *Not Done Yet*. Figure 11.21 displays the dialog box for this module. Note that this *Decide* module implements a two-way decision (in the *Type* field), controlled by a (logical) expression (in the *If* field) specified as *Proc_Index* < *5* (in the *Value* field). If the value of this expression is false (i.e., *Proc_Index* == *5* is true), then the unit entity processing is essentially complete, with only a few remaining housekeeping chores left.

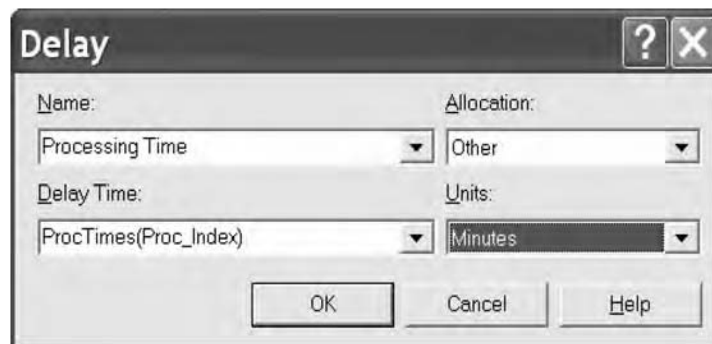


Figure 11.20 Dialog box of the second module (*Delay*) in the loop.

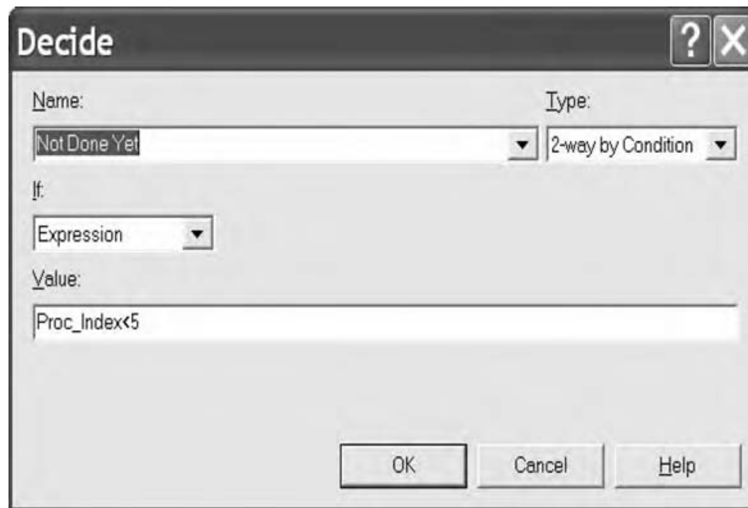


Figure 11.21 Dialog box of the third module (*Decide*) in the loop.

Specifically, the unit entity will proceed in this case to the *Release* module, called *Release Last Process*, to release the last resource, and will enter the *Record* module, called *Flow Time*, to collect the next system flow-time observation, before routing to the *Assign* module, called *Assign Process Index and Time*, for the next tour through all five workstations (new job processing). However, if *Proc_Index* < *5* is true, then the unit entity proceeds to the fourth module in the loop, namely, the *Hold* module called *See if Process is Blocked*; see the module's dialog box in Figure 11.22. In this module, the unit entity checks whether the next workstation (indexed by *Proc_Index+1*) is full (i.e., that transfer to the next process is blocked), by checking the logical expression in the *Condition* field,

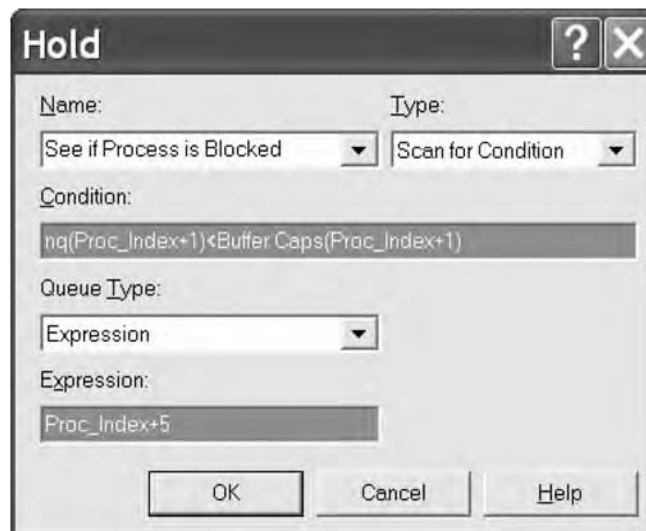


Figure 11.22 Dialog box of the fourth module (*Hold*) in the loop.

$$nq(Proc_Index + 1) < Buffer\ Caps(Proc_Index + 1).$$

Recall that *Buffer Caps* is an expression vector holding buffer capacities, which were previously initialized by the modeler. Note carefully that blocking queue indexes are specified by expressions (*Queue Type* field) whose values are given by *Proc_Index + 5* (*Expression* field). Thus, this *Hold* module is used for all processes, except for the last one; it holds a blocked unit entity at queue *Proc_index+5* until the next workstation has room to accommodate it.

As soon as the next workstation is found to have room (i.e., the next process is found to be nonblocking), the unit entity will move to the fifth module in the loop, namely, the *Release* module, called *Release Process*, to release the current resource (the one indexed by *Proc_Index*). The unit entity then proceeds to the sixth and last module in the loop, namely, the *Assign* module called *Next Process*, where the unit entity's attribute *Proc_Index* is incremented by 1, thereby pointing at the next workstation (process) to visit. Finally, the unit entity completes the loop by returning to the first module in the loop, namely, the *Seize* module, called *Seize Process*, to start the next loop iteration.

A comparison of run results of the equivalent packaging line models of Figures 11.2 and 11.17 shows that they produce identical statistics, as expected.

11.6 AN ALTERNATIVE METHOD OF MODELING BLOCKING

There are several ways of modeling blocking in Arena. In the packaging line example of Section 11.3, we chose to model blocking via the *Hold* module of the *Basic Process* template panel. An alternative way would be to treat each buffer as a resource. Figure 11.23 exemplifies this alternative modeling approach for a buffer in the packaging line model in Figure 11.2.

In Figure 11.23, the resource at the labeling workstation, which controls the number of units in the queue of *Labeling Process*, is named *Labeling Buffer*. The *Capacity* entry (number of servers) of this resource is set to 5 units in order to correspond to a buffer capacity of 5 in *Labeling Process*. An Arena implementation of this approach to modeling blocking is depicted in Figure 11.24.

In the previous model illustration our goal was to ensure that when the queue in *Labeling Process* is full, the *Capper* resource is not released for further capping. To this end, we arrange that each unit entity that has completed capping at *Capper* attempt to seize a unit capacity of *Labeling Buffer* before releasing the *Capper* resource. If no residual capacity of the resource *Labeling Buffer* is available, then the resource *Capper* gets blocked, thereby holding the unit entity until buffer space becomes available in the *Labeling Buffer* resource. As soon as this happens, the unit entity releases the *Capper* resource and moves on to *Labeling Process*, where it immediately joins *Labeling*

Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Failures
1	Labeling Buffer	Fixed Capacity	5	0.0	0.0	0.0		0 rows

Figure 11.23 Dialog spreadsheet of the *Resource* module for modeling blocking.

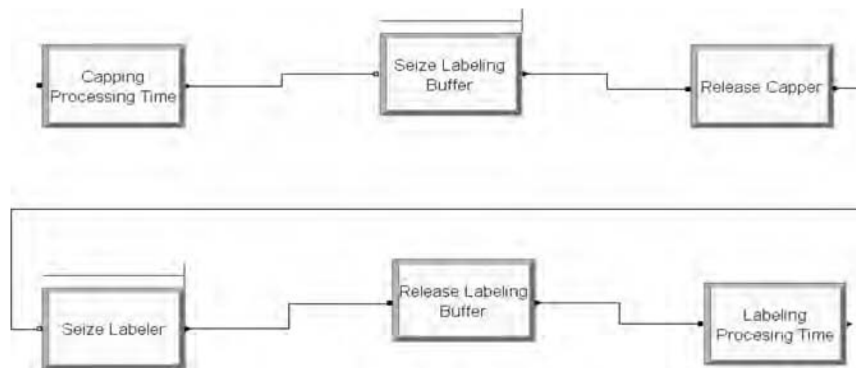


Figure 11.24 Alternate Arena modeling of blocking via *Resource* modules.

Process.Queue, and waits for its turn to seize the resource *Labeler*. During this wait, the unit entity keeps its unit space of the resource *Labeling Buffer*, and only releases that space when it succeeds in seizing the resource *Labeler*. Since there are only five units of resource available for seizing at *Labeling Buffer*, there will be at most five unit entities in the "physical" buffer *Labeling Process.Queue*.

Note that this approach to modeling blocking in Arena is costlier than the one used in modeling the packaging line in Section 11.3. There, only one module (*Hold*) was used to model blocking per process, whereas here three modules (*Seize*, *Release*, and *Resource*) are used for the same purpose. This increased complexity can become problematic. For example, the student version of Arena allows only a limited number of modules to be used in any given model.

11.7 MODELING MACHINE FAILURES

Process stoppages are unavoidable in manufacturing and service systems. In particular, machine failures of various kinds constitute an important source of idleness and variability in such systems. When modeling systems subject to failures, the modeler is usually interested in the long-run probabilities of down states, which translate operationally into the long-range fraction of time spent in those states. Certainly, efficient operation requires that downtimes be minimized, since these represent loss of production time. Simulation can help the modeler understand the impact of failures on system performance.

A stochastic process that models failures is characterized either by the statistical properties of the time intervals separating consecutive failures, or by the counting process that keeps track of the number of failures within a given period of time. These two characterizations are mathematically equivalent. However, the former characterization is commonly used to generate failures in simulation runs. In fact, failures can be modeled as a specialized arrival process of high priority, whose transactions simply preempt the server.

A typical failure scenario at a single workstation unfolds as follows. After a period of normal operation (uptime), a failure event occurs, the workstation stops its processing, and then experiences a downtime while undergoing a repair. (Sometimes an additional

delay for repair setup needs to be modeled, but this delay can usually be absorbed into a longer repair time.) Failures that occur while machines are actually processing jobs are called *operation dependent*. However, the new breed of highly computerized machines may fail at any time, regardless of machine status. Such failures are called *operation independent*, and may include machine malfunctions, startups, cleanups, adjustments, and other delays specific to a particular system. The two fundamental machine states, *Idle* and *Busy*, must then be augmented by additional failure and stoppage states, such as *Down*, *Cleaning*, *Adjustment*, and so on.

Arena admits any number of user-defined states in addition to the built-in ones (called *auto-states*). In particular, an Arena resource has four *auto-states*: *Idle*, *Busy*, *Failed*, and *Inactive*, and at any point in time a resource is in one of these states. A transition from one state to another is caused by an event occurrence. For instance, a failure-arrival event automatically causes the machine to undergo a transition to the *Failed* state. On the other hand, user-defined states and their transitions can be programmed entirely at the modeler's discretion. We are usually interested in the long-run probabilities of these states; such probabilities can be estimated via the *Frequency* option in the *Statistic* module.

The modeler can modulate the capacity of a resource over time by defining a time-varying capacity level in the *Schedule* module (see Section 5.8 for more details). In particular, to model forced downtimes (e.g., preventive maintenance), a resource can be inactivated by setting its capacity to zero, in which case the *Inactive* auto-state will have a non-zero probability.

Suppose that *Filling Process* in the packaging line model of Section 11.3 fails randomly and that it needs an adjustment after every 250 departures from the workstation. Assume that uptimes (times between a repair completion and the next failure, or time to failure) are exponentially distributed with a mean of 50 hours, while repair times are uniformly distributed between 1.5 hours to 3 hours. Also, the aforementioned adjustment time is uniformly distributed between 10 minutes to 25 minutes. Assume further that *Packing Process* can also experience random mechanical failures, and downtimes are triangularly distributed with a minimum of 75 minutes, a maximum of 2 hours, and a mode at 90 minutes. The corresponding uptimes are exponentially distributed with a mean of 25 hours. Finally, assume that random failures occur only while the machines are busy (operation-dependent failures). We shall refer to the modified packaging line model as the *failure-modified model*.

Figure 11.25 illustrates how failures in the failure-modified model are specified in a dialog spreadsheet for the *Failure* module from the *Advanced Process* template panel. The *Name* column in Figure 11.25 specifies failure names, while the *Type* column selects the type of failure arrivals: *Time* (for time-based arrivals) or *Count* (for count-based arrivals). For instance, random mechanical failures are normally time based, since

Failure - Advanced Process

	Name	Type	Up Time	Up Time Units	Count	Down Time	Down Time Units	Uptime in this State only
1	Random Failures_F	Time	EXPO(50)	Hours		UNIF(1.5 ,3)	Hours	Busy
2	Random Failures_P	Time	EXPO(25)	Hours		TRIA(75 ,90,120)	Minutes	Busy
3	Adjustment	Count			250	UNIF(10,25)	Minutes	

Figure 11.25 Dialog spreadsheet of the *Failure* module.

their arrivals call for interfailure time specification. In contrast, if a machine requires a cleanup action whenever it completes processing a prescribed number of units, then the cleanup stoppage is count based. Note that in Figure 11.25, the failure/stoppage named *Adjustment* is declared to be count-based.

For time-based failures, the *Up Time* column specifies the time interval to the next failure (after a repair completion), while the *Up Time Units* column specifies the corresponding time unit. Similarly, for count-based failures, the *Count* column specifies the number of entity departures to the next failure (250 for the *Adjustment* failure). Note that unused column entries are shaded depending on failure type.

The *Down Time* column specifies the length of downtimes, while the *Down Time Units* column specifies the corresponding time unit. Finally, the *Uptime in this State only* column is used for time-based failures to specify the state in which the resource must be for the failure to occur. For example, the time-based failures in Figure 11.25 can only occur when the underlying resources are in the *Busy* state.

Arena provides a mechanism for defining resource states and for linking them to failures/stoppages in the form of the *StateSet* spreadsheet module from the *Advanced Process* template panel. Figure 11.26 illustrates the use of the *StateSet* module for the packaging line model of Section 11.3. The left dialog spreadsheet of Figure 11.26 specifies a state set name (under the *Name* column) and the number of associated states (under the *States* column). For example, the first row specifies a state set, called *Filling States*, for the *Filler* resource with 4 states (the button labeled *4 rows*). Clicking that button pops up the dialog box to the right, which displays detailed state information. There, the column *State Name* displays user-defined or auto-state names, while the column *AutoState or Failure* indicates the association of each state name with the corresponding auto-state or user-defined failure name.

Finally, Figure 11.27 illustrates for the failure-modified model how the *Resource* module is used to associate resource states with failures and the action to be taken on failure occurrences. The bottom dialog spreadsheet of Figure 11.27 specifies in the first row that resource *Filler* (column *Name*) has 2 types of failures (the button labeled *2 rows* in column *Failures*). Clicking that button pops up the dialog spreadsheet at the top of Figure 11.27, which shows that resource *Filler* has two failures, called *Random Failures_F* and *Adjustment* (column *Failure Name*). The *Failure Rule* column specifies the requisite action to be taken on the unit entity (or entities) being processed when a time-based failure occurs (it does not apply to count-based failures). Actions are specified via options, the most common of which follow:

StateSet - Advanced Process

	Name	States
1	Filling States	4 rows
2	Capping States	2 rows
3	Labelling States	2 rows
4	Sealing States	2 rows
5	Packing States	3 rows

States

	State Name	AutoState or Failure
1	Idle	Idle
2	Busy	Busy
3	Adjust	Adjustment
4	Down	Random Failures_F

Figure 11.26 Dialog spreadsheets of the *StateSet* module (left) and resource *Filler* states (right).

Failures

	Failure Name	Failure Rule
1	Random Failures_F	Preempt
2	Adjustment	Wait

Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Initial State	Failures	Report Statistics
1	Filler	Fixed Capacity	1	0.0	0.0	0.0	Filling States		2 rows	☑
2	Capper	Fixed Capacity	1	0.0	0.0	0.0	Capping States		0 rows	☑
3	Packer	Fixed Capacity	1	0.0	0.0	0.0	Packing States		1 rows	☑
4	Sealer	Fixed Capacity	1	0.0	0.0	0.0	Sealing States		0 rows	☑
5	Labeler	Fixed Capacity	1	0.0	0.0	0.0	Labelling States		0 rows	☑

Figure 11.27 *Resource* dialog spreadsheet (bottom) and *Failures* dialog spreadsheet (top) in the failure-modified packaging line model.

- The *Preempt* option starts a downtime by suspending the resource immediately on failure arrival, so that the remaining processing of the current unit entity will resume once the downtime is over.
- The *Wait* option allows the current unit entity to finish processing, after which the resource is suspended and downtime begins.
- The *Ignore* option starts the downtime after the current unit entity finishes processing. However, only that portion of the downtime *following* the current unit entity completion is recorded (in contrast, the *Wait* option records the full downtime).

In our example, failure *Random Failures_F* will apply the *Preempt* option, while failure *Adjustment*, being count-based, will formally apply the *Wait* option, even though any other option could have been selected (recall that options do not apply to count-based failures).

The state probabilities of each resource can be obtained by requesting the collection of *Frequency* statistics in the *Statistic* module with the *State* option selected in its *Frequency Type* column. Figure 11.28 displays the resulting *Frequencies* report for the failure-modified model.

It is instructive to compare the performance of the original packaging line model of Section 11.3 with its failure-modified version. Everything else being the same, we expect the failure-modified model to show poorer performance than the original model. Indeed, a comparison of Figure 11.28 to Figure 11.12 reveals that the resources downstream of the *Filler* resource in the failure-modified model experience markedly increased idleness. The explanation of this outcome is straightforward. Since the *Filler* resource in the failure-modified model is shut down by random failures, it cannot produce as much as in the original model. Consequently, downstream processes are more frequently starved, leading to overall increased idleness in the system. In a similar vein, Figure 11.29 displays the resultant *User Specified* report for the failure-modified model.

A comparison of Figure 11.29 with Figure 11.11 reveals additional evidence that the failure-modified model performs at a lower level than the original one. Because *Filling Process* has increased idleness in the failure-modified model, its mean interdeparture

6:06:00PM **Frequencies** February 10, 2006

Packaging Line Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 1,000,000.00 Time Units: Seconds

Capper States	Number Obs	Average Time	Standard Percent	Restricted Percent
Busy	9,300	68.5537	63.75	63.75
Idle	9,300	38.9732	36.25	36.25

Filler States	Number Obs	Average Time	Standard Percent	Restricted Percent
Adjust	318	1,065.04	33.87	33.87
Busy	322	1,980.29	63.77	63.77
Down	3	7,888.19	2.37	2.37

Labeler States	Number Obs	Average Time	Standard Percent	Restricted Percent
Busy	321	2,073.88	66.57	66.57
Idle	321	1,041.38	33.43	33.43

Packer States	Number Obs	Average Time	Standard Percent	Restricted Percent
Busy	79,527	6.0145	47.83	47.83
Down	5	5,674.67	2.84	2.84
Idle	79,523	6.2034	49.33	49.33

Sealer States	Number Obs	Average Time	Standard Percent	Restricted Percent
Busy	79,615	5.3617	42.69	42.69
Idle	79,616	7.1987	57.31	57.31

Figure 11.28 *Frequencies* report for the failure-modified packaging line model.

time increases to about 12.5 seconds (as opposed to 8 seconds in the original model), and concomitantly, its system throughput is only about 0.08 (as opposed to 0.125 units per second in the original model). Thus, this example amply demonstrates the deleterious effects of machine failures and stoppages on system performance measures. The economic consequences of the resultant system performance must necessarily follow suit.

The impact of machine failures on system performance and the reduction of this impact by placing buffers between machines have been extensively studied. For further discussions on the subject, see Buzacott and Shanthikumar (1993), Gershwin (1994), Altiok (1997), and Papadopoulos et al. (1993). Analysis of manufacturing networks with infinite buffers is found in Whitt (1983).

11.8 ESTIMATING DISTRIBUTIONS OF SOJOURN TIMES

Frequency statistics are handy Arena constructs that simplify the task of estimating the distribution of the number of units in a queue or buffer. However, estimating the distribution of a sojourn time (total time spent in a system or subsystem) requires a bit more work. For instance, suppose we are interested in the distribution of the delay time D_p that a product unit spends in the packing buffer of the packaging line model of Section 11.3. More specifically, we wish to estimate the following delay-time probabilities:

6:08:04PM **User Specified** February 10, 2006

Packaging Line Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 1,000,000.00 Time Units: Seconds

Tally

Between	Average	Half Width	Minimum	Maximum
Interdeparture Time	12.5438	0.575661455	6.0000	10,488.55

Interval	Average	Half Width	Minimum	Maximum
Flow Time	376.28	17.36510	30.5000	11,965.81

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Capper is Blocked	0.2389	0.016142093	0	1.0000
Filler is Blocked	0.1194	0.018964448	0	1.0000
Labeler is Blocked	0.02794982	(Insufficient)	0	1.0000
Sealer is Blocked	0.02826982	(Insufficient)	0	1.0000

Output

Output	Value
Throughput	0.07972077

Figure 11.29 *User Specified* report for the failure-modified packaging line model.

$$\Pr\{D_p < 1\}, \Pr\{1 \le D_p < 3\}, \Pr\{3 \le D_p < 5\} \text{ and } \Pr\{ D_p \ge 5\}.$$

Figure 11.30 displays an Arena fragment for estimating these delay-time probabilities. Here, an arriving unit entity traverses the fragment from left to right in Figure 11.30. On arrival in the packing workstation, the first (*Assign*) module time stamps the current time (*Tnow*) in the unit entity's attribute *Delay Start Time*. The unit entity then enters the second (*Seize*) module and eventually succeeds in seizing the packing resource *Packer* there, possibly after waiting in its queue. At this point it enters the third (*Assign*) module, where its delay time in the packing buffer is computed by the expression

$$Delay\ in\ Pack_Q = Tnow - Delay\ Start\ Time$$

and stored in the unit's *Delay in Pack_Q* attribute. The computed delay times are tallied by adding the outcomes of the parenthesized conditions, which evaluate either to 1 or 0 (if true or false, respectively). For example, the variable *N1* counts the number of delay observations strictly less than 1, *N3* counts those that fall in the interval [1, 3), *N5* counts those that fall in the interval [3, 5), and *NM* counts those that are greater than or

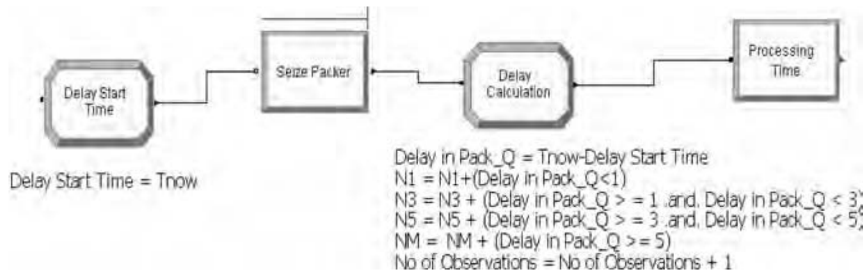


Figure 11.30 Arena fragment for estimating delay-time probabilities in the packing buffer.

equal to 5. Note that every incoming unit entity increments one and only one of the variables *N1*, *N3*, *N5*, and *NM*. Finally, the variable *No of Observations* tracks the total number of delay observations.

When the replication terminates, the requisite delay-time probabilities are computed via *Output* type statistics, defined in the *Statistic* module. Figure 11.31 displays the dialog spreadsheet for estimating the four requisite delay-time probabilities using suitable ratio expressions.

In a similar vein, one can also estimate other sojourn-time probabilities, such as the distribution of the manufacturing lead time (the time from the start of the first operation to the completion of the last one). The implementation of such sojourn-time estimation is left as an exercise to the reader.

11.9 BATCH PROCESSING

In typical manufacturing environments, it is quite common to encounter processes or machines that operate on a number of jobs (product units) as a group. The group size is usually a prescribed fixed number, but it may also be random. In many cases, the group of jobs becomes a permanent assembly and continues its manufacturing process as a single unit. In some cases, the converse occurs, and a group of jobs is split into the constituent individual members. In manufacturing parlance, job groups are said to be processed in *batches*. Examples of batch processes include painting, heat treatment, packaging, various assembly and montage operations in the auto and electronics industries, as well as a variety of operations such as sterilization, shaking, and centrifugation in chemical and pharmaceutical facilities and many others. The Arena constructs that support group processing functionality are the *Batch* and *Separate* modules. The *Batch* module supports both *permanent* and *temporary* batches. Conversely, the

Statistic - Advanced Process

	Name	Type	Expression	Report Label	Output File
1	P Delay LT 1	Output	N1/No of Observations	P Delay LT 1	
2	P Delay LT 3	Output	N3/No of Observations	P Delay LT 3	
3	P Delay LT 5	Output	N5/No of Observations	P Delay LT 5	
4	P Delay GE 5	Output	NM/No of Observations	P Delay GE 5	

Figure 11.31 Dialog spreadsheet of the *Statistic* module with *Output* statistics for estimating delay probabilities in the packaging buffer.

Separate module recovers the constituent individual members of temporary groups; permanent groups cannot be split into their constituent members.

To illustrate batch processing, we modify the failure-modified packaging line of Section 11.7 by incorporating batching and separating, as shown in Figure 11.32, and refer to this system as the *batch-modified* packaging line model. Let product unit interarrival times in the batch-modified model be uniformly distributed between 5 and 10 seconds. Assume that *Labeling Process* labels batches of five units at a time, following which the units proceed separately as individual units. Assume further that batches of 10 units are packed together in *Packing Process*. Each batch labeling time is 25 seconds, and each batch packing time is 30 seconds. To enable batching and separating, we need to increase the buffer capacity at *Packing Process* to accommodate batches. For simplicity, we just set all buffer capacities to infinity, thereby eliminating blocking. However, machine failures and stoppages are retained in the batch-modified model. We are interested in the mean flow time of unit entities from their arrival at queue *Filling Process.Queue* until their arrival at module *Interdeparture Time*.

Figure 11.33 displays the dialog box of the first *Batch* module following the capping operation. This *Batch* module, called *Batch 1*, collects unit entities into temporary batches (*Type* field) of size 5 (*Batch Size* field). Recall that labeling is then carried out for each batch as a whole. The *Rule* field specifies that batches will consist of any unit entity (option *Any Entity*), rather than restricting batching to unit entities with the same value of a prescribed attribute (*By Attribute* option). Next, the *Temporary* option in the *Type* field displayed in Figure 11.33 specifies that this batch is to be split later on (as opposed to the *Permanent* option). Finally, the *Save Criterion* field is used to select

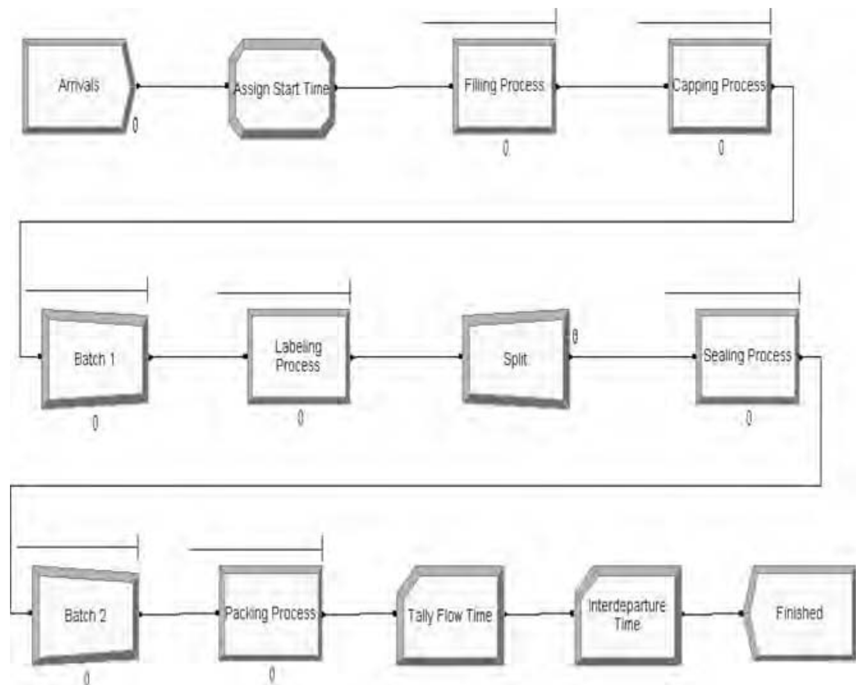


Figure 11.32 Arena model for the batch-modified packaging line system.

the set of attributes associated with each batch. For example, in the dialog box of Figure 11.33, the option *Last* was selected, indicating that batch attributes are inherited from the last unit entity of the batch (the one that completes the batch).

Following the labeling operation, each batch enters the *Separate* module called *Split*, whose dialog box is displayed in Figure 11.34. The *Split Existing Batch* option in the *Type* field stipulates that batches are to be split back into their individual constituent members. The other *Type* field option, *Duplicate Original*, is used to create duplicates of the entity that enters this module. The option *Retain Original Entity Values* selected in the *Member Attributes* field ensures that the original attributes of all batch members will be recovered upon splitting.

After the sealing operation is completed, the units are packed at *Packing Process* into carton boxes in batches of 10. Package entities then enter additional modules, where they

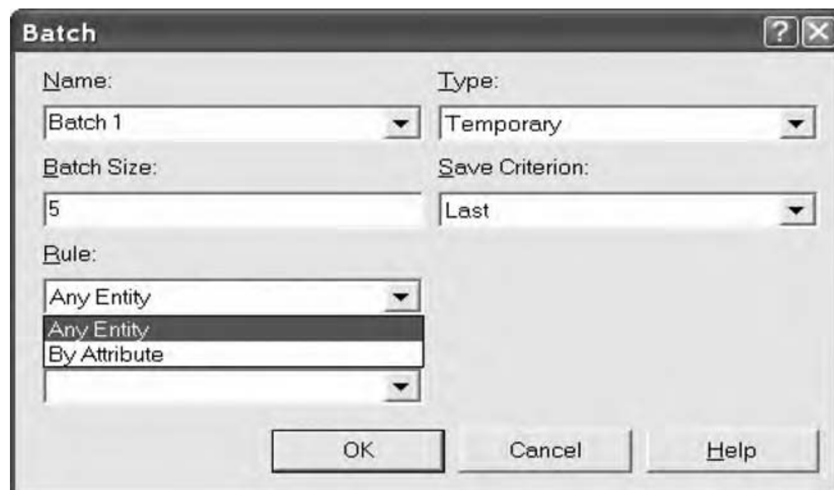


Figure 11.33 Dialog box of the *Batch* module *Batch 1*.

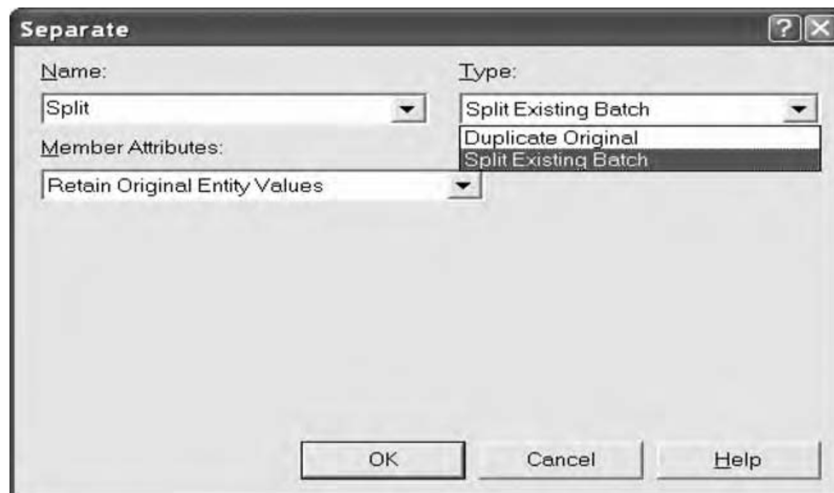


Figure 11.34 Dialog box of the *Separate* module *Split*.

are counted and their system flow time is tallied before being disposed of at module *Finished*.

The simulation *Resources* report for one replication of the batch-modified model is displayed in Figure 11.35. Finally, the corresponding *User Specified* report is displayed in Figure 11.36.

11.10 ASSEMBLY OPERATIONS

It is quite common in manufacturing environments to have assembly operations, where a number of parts from different buffers are assembled to produce a single unit of finished or semifinished product. Arriving matching part units are buffered in matching buffers in front of the assembly station as shown in Figure 11.37.

6:08:15PM **Resources** March 2, 2006

Packaging Line Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Seconds

Resource Detail Summary

Usage

	Inst Util	Num Busy	Num Sched	Num Seized	Sched Util
Capper	0.67	0.67	1.00	13,335.00	0.67
Filler	0.87	0.87	1.00	13,336.00	0.87
Labeler	0.67	0.67	1.00	2,666.00	0.67
Packer	0.40	0.40	1.00	1,332.00	0.40
Sealer	0.67	0.67	1.00	13,328.00	0.67

Figure 11.35 *Resources* report for the batch-modified model.

6:10:44PM **User Specified** March 2, 2006

Packaging Line Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Seconds

Tally

Between	Average	Half Width	Minimum	Maximum
Interdeparture Time	74.9941	0.235296642	65.0000	88.6404

Interval	Average	Half Width	Minimum	Maximum
Flow Time	91.9193	0.033092160	91.5000	97.9789

Output

Output	Value
Throughput	0.01333438

Figure 11.36 *User Specified* report for the batch-modified model.

The number of units taken from the matching buffers to the assembly operation may vary from one case to case. In all cases, however, the assembly operation can start only after all the matching units are present in the matching buffers.

Arena provides a module called *Match*, which can be used for assembly of matching parts. More generally, the *Match* module is used to synchronize the movement of various unit entities in a model. For instance, the arriving matching entities may wait for each other in buffers until a batch is complete, at which time each entity resumes moving along its logical path in the model.

The *Match* module and its dialog box are shown in Figure 11.38. Note that the module icon contains as many matching buffers (T bars) as the value in the *Number to Match* field in the dialog box. Furthermore, there are that many connection entry points as well as exit points in the *Match* module.

The *Match* module works as follows. As soon as each buffer has at least one entity in it, precisely one entity is picked from each matching buffer, and these are sent to the associated connecting destinations. Thus, this module can be used to synchronize entity movements. In particular, the *Match* module may be used to model assembly operations by connecting exactly one exit point to the designated assembly station (the corresponding entity represents the assembled unit product), and all others to a *Dispose* module. The synchronization point is specified by the *Type* field: the *Any Entities* option synchronizes any entities in the matching buffers, while the *Based on Attribute* option selects the synchronized entities as those that have the same value in the specified attribute. For example, if entities have a color attribute, then only entities with the same color will be matched. Thus, a batch might be declared if each matching buffer contains at least one red entity. However, only one red entity is then selected from each matching buffer. If various numbers of units are assembled from matching buffers, then a

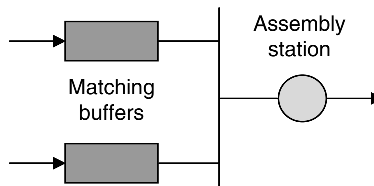


Figure 11.37 Schematic representation of an assembly operation.

Figure 11.38 A *Match* module icon (left) and its dialog box (right).

Batch module can be used first, and then the batched units can be assembled using a *Match* module.

11.11 MODEL VERIFICATION FOR PRODUCTION LINES

This section generalizes certain aspects of production lines that play an important role in understanding line behavior and verifying its logic.

Consider the production line depicted in Figure 11.39. Here, M_i denotes the i-th machine and B_i denotes the buffer between M_{i-1} and M_i with finite-capacity N_i (excluding the server positions in both machines). Let X_i be the random variable representing the processing time in M_i. We may assume that M_1 has a sufficient amount of raw material, so that it never starves. Another possibility is that M_1 has a buffer at which product units arrive in a particular manner (either randomly or according to a schedule).

Define the following probabilities:

$P_i(I)$ is the probability that M_i is idle.
$P_i(B)$ is the probability that M_i is blocked.
$P_i(D)$ is the probability that M_i is down.
$P_i(U)$ is the probability that M_i is up (producing).

Then, the utilization of M_i is given by

$$P_i(U) = 1 - P_i(I) - P_i(B) - P_i(D), \tag{11.4}$$

indicating that M_i is in the up state when it is neither idle, nor blocked, nor down.

While M_i is actually busy (with probability $P_i(U)$, it is producing at rate $1/\mathrm{E}[X_i]$ yielding a throughput $\bar{o}_i$ given by

$$\bar{o}_i = \frac{P_i(U)}{\mathrm{E}[X_i]}. \tag{11.5}$$

In the event that product units are not lost, the long-run flow rate (throughput) of units in a production line with K machines is the same at every machine, namely,

$$\bar{o}_1 = \bar{o}_2 = \cdots = \bar{o}_K. \tag{11.6}$$

Thus, workstation throughputs are identical, each equal to the line throughput, $\bar{o}_\ell$. If product units are lost at a particular workstation j due to scrapping or rejection (with some prescribed probability), then the throughput $\bar{o}_j$ should be adjusted accordingly. In addition, all throughputs of downstream workstations would be similarly

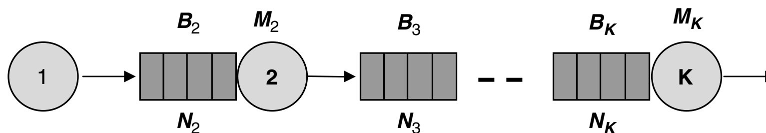


Figure 11.39 Schematic representation of a generic production line.

adjusted. Equations 11.4–11.6 can assist the modeler in verifying production line models.

Little's formula (Eq. 8.8) can also apply to the entire production line (see Section 11.4). Consider again the production line in Figure 11.39 with K workstations and $K - 1$ buffers, and let $\bar{F}_S$ be the average total time a product unit spends in the line. Let $\bar{N}_j$ denote the average contents in buffer B_j, and let $\bar{N}_S$ denote the average total number of units in the system. Little's formula is given by

$$\bar{N}_S = \bar{o}_\ell \bar{F}_S, \tag{11.7}$$

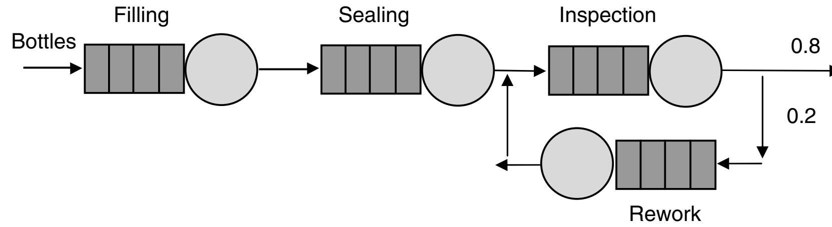
which was used in model verification in Section 11.4. On the other hand, $\bar{N}_S$ can be computed directly as

$$\bar{N}_S = \sum_{j=2}^{K} \bar{N}_j + \sum_{j=1}^{K} P_j(U) + \sum_{j=1}^{K-1} P_j(B), \tag{11.8}$$

where the first term on the right side is the average total buffer contents of the line (excluding the first machine that has no buffer and further excluding server positions), the second term is the average total number of units in service positions while the servers are busy, and the last term is the average total number of units in server positions while the servers are blocked. For verification purposes, the modeler should check the agreement of computations for Eqs. 11.7 and 11.8.

EXERCISES

1. *Three-stage production line*. Consider the filling and inspection system consisting of three stages in series shown in the next illustration. Bottles arrive with iid exponentially distributed interarrival times with a mean of 2 minutes. They enter a filling operation with an infinite buffer in front of it to accommodate arriving bottles. The filler fills a batch of five bottles at a time. Filling times are iid uniformly distributed between 3 and 5 minutes. The entire batch of 5 bottles then attempts to join the sealer queue (of 10-bottle capacity) as individual bottles. However, if the sealer queue cannot accommodate the entire batch, the filler becomes blocked until space becomes available in the sealer queue. The filler is subject to failures with iid exponential times to failure with a mean of 50 minutes, and with iid triangularly distributed downtimes with parameters (2, 5, 10) minutes. Failures may occur at any point in time. The sealing operation is performed on a batch of three bottles at a time. It takes precisely 3 minutes to seal all three bottles. The inspection queue has a finite capacity of 10 bottles. Inspection takes precisely 1.45 minutes per bottle. Experience shows that 80% of bottles pass inspection and depart from the system, while the rest are routed to a rework station to modify the fill volume. The rework station has an infinite-capacity queue and a single server. Rework times are iid uniform between 1 and 4 minutes. After rework, bottles are routed back to the inspection queue for another inspection. However, rework bottles have a higher priority than nonrework bottles. Note that a bottle may go through rework more than once.



a. Develop an Arena model for the production line, and simulate it for 1 year, assuming 365 days of work per year and 8 hours of work per day.
b. Estimate the following statistics:
   - Average WIP contents in each buffer
   - Server utilization for each workstation
   - Average flow time through the system
   - Distribution (histogram) of flow times with five intervals
   - State probabilities (busy, idle, down, blocked) of every resource in the system
   - Probability that the filling, sealing, and rework workstations are simultaneously blocked by the inspection station

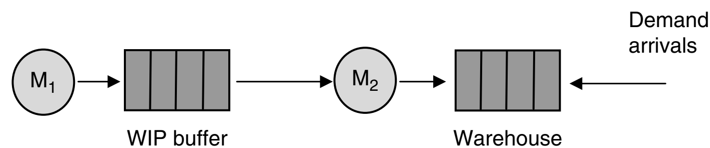
2. *Production system with inspection and rework.* Consider the following production system, which produces printed circuit boards in multiple stages with unlimited buffer capacities. Unit interarrival times are iid exponential with a mean of 2.2 hours. Arriving boards have two parts: part 1 and part 2. Upon arrival, parts separate and proceed to their respective processes separately in two branches. Part 1 units proceed along the first branch to a chemical disintegration process followed by a circuit layout 1 process, where disintegration times are iid uniform between 0.5 and 2 hours, and circuit layout times are iid uniform between 1 and 2 hours. Part 2 units proceed along the second branch to an electromechanical cleaning process followed by a circuit layout 2 process, and the corresponding processing times are iid exponentially distributed with means 1.4 and 1.5 hours, respectively. An assembly workstation then picks one unit from each branch and assembles them in an operation that lasts precisely 1.5 hours. Assembled units proceed to a testing station where batches of five units are tested simultaneously. Testing times are iid triangularly distributed with parameters 6, 8, and 10 hours. After testing, batches are split into individual units, which then depart from the system. After every 8-hour period, all processes shut down for a 1-hour maintenance operation. The following random stoppages are observed in each process.

Station Name	Failure Type	Time to Failure (hours)	Time to Repair (hours)
Disintegration	Random	Expo(1/20)	Unif(1, 2)
Cleaning	Random	Expo(1/30)	Beta(5, 1)
Layout 1	Random	Expo(1/25)	Unif(2, 5)
Layout 2	Random	Expo(1/25)	Beta(5, 1)
Testing	Adjustment	Every 200 units	Tria(1, 3, 4)

Recall that in Arena, the parameters of the beta distribution are in reverse order, and the parameter of the exponential distribution is the mean (not the rate).

a. Develop an Arena model for the production system, and simulate it for 1 year of operation.
b. Estimate the following statistics:
   - Utilization of each process
   - Average delay at each process
   - Average system flow time of assembled units
   - Distribution of the number of jobs in the testing buffer
   - Probability that the system is in maintenance
   - State probabilities of each process (idle, busy, failed, and maintenance)

3. *Two-stage manufacturing system with warehousing*. Consider the following two-stage manufacturing system, where a product is produced at stage 1 and packaged at stage 2.



Customers arrive at the warehouse with product demands and their orders are filled (possibly backordered) from warehouse inventory. When the reorder point at the warehouse is reached, machine M_2 starts producing using material from the buffer, which is fed by machine M_1.

Stage 1 consists of machine M_1 and an output WIP buffer capacity of 40 units. Machine M_1 always has material to process (never starves) and takes 1 hour to process a unit. Stage 2 consists of machine M_2 and a finished product warehouse. Machine M_2 takes 0.75 hours to process a unit. Customers arrive at the finished product warehouse according to a Poisson process at the rate of 1 per 9 hours. The demand size distribution is Disc({(0.3, 3), (0.3, 6), (0.4, 12)}), and excess customer demand is backordered.

Machine M_1 is blocked whenever a finished unit cannot enter the WIP buffer due to lack of space, and remains blocked until room becomes available in the WIP buffer. Machine M_2 strives to maintain a level of 60 units in the warehouse: Production is in progress at M_2 as long as the inventory level at the warehouse is below 60, and gets blocked when that inventory level reaches 60. (Note: warehousing will be covered in the next chapter in more detail in the context of supply chains.)

a. Develop an Arena model for the manufacturing system, and simulate it for 10 years.
b. Estimate the following statistics:
   - Actual machine utilizations (excluding any idleness)
   - Blocking probabilities for each machine
   - Average inventory levels in the WIP buffer and the warehouse
   - Average backorder level in the finished product warehouse
   - Probability of backordering upon customer arrival

This page intentionally left blank

Chapter 12

Modeling Supply Chain Systems

A *supply chain system* (*supply chain*, for short) is a network that mediates the flow of entities involved in a product life cycle, from production to vending (Simchi-Levi et al. [2003]). Such a network consists of nodes and arcs. Nodes represent suppliers, manufacturers, distributors, and vendors (e.g., retail stores), as well as their inventory facilities for storing products and transportation facilities for shipping them among nodes. Arcs represent routes connecting the nodes along which goods are transported in a variety of modes (trucking, railways, airways, and so on). From a sufficiently high vantage point, supply chains have essentially a feed-forward network structure, with upstream and downstream components arranged in *echelons* (stages), such that raw materials, parts, products, and so on flow downstream, payments flow upstream, and information flows in both directions. However, there are supply chains where entity flows may not be strictly directional because products may be returned for repair or refund. The key supply chain echelons are as follows:

1. The *supply echelon* feeds raw material or parts to the manufacturing operations.
2. The *production echelon* converts raw material and parts to finished product.
3. The *distribution echelon* consists of a *distribution network* (warehouses, distribution centers, and transportation facilities) that moves finished products to vendors.
4. The *vendor echelon* sells products to end-customers.

In practice, the complexity of supply chains spans a broad range from simple supply chains where each echelon is a single node, to complex ones where each echelon is itself a complicated network consisting of a large number of nodes and arcs. For example, the production echelon may be hierarchical such that upstream factories provide parts to a downstream factory that assembles the parts into more elaborate intermediate or final products. In a similar vein, a distribution network can consist of a large number of warehouses arranged hierarchically from distribution centers to wholesalers and retailers. In fact, supply chains may cross international boundaries and extend over multiple continents.

The importance of supply chains stems from the fact that they constitute a large chunk of economic activities (in the United States, supply chains contribute about 10% of gross domestic product). To improve their bottom line, gain competitive advantage, or just plain survive, companies are interested in effective and efficient supply chain

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

operations that meet customer expectations. To this end, the discipline of *supply chain management* (*SCM*) sets itself the mission of producing and distributing products in the right quantities to the right locations at the right time, while keeping costs down and customer service levels up. In essence, SCM (also known by the older term *logistics*) aims to fulfill its mission by searching for good trade-offs between system costs and customer satisfaction. Thus, SCM is concerned with the efficient and cost-effective integration and coordination of the following supply chain elements:

1. Suppliers, factories, warehouses, and stores that span the nodes of a supply chain
2. The transportation network linking these nodes
3. The information technology infrastructure that enables data exchange among supply chain nodes, as well as information systems and tools that support supply chain planning, design, and day-to-day operations
4. Methodologies and algorithms for controlling material flow and inventory management

SCM faces a number of challenging issues involving difficult decisions. Should the supply-chain decision making be centralized (single decision maker for the entire network that uses all available information) or distributed (multiple decision makers that use information local to their part of the network)? Are decision makers allowed to share information (transparency)? How can order-quantity variability be reduced? For example, it has been noted that lack of transparency in a supply chain gives rise to the so-called *bullwhip effect*, whereby the variability of orders increases as one moves up the supply chain. While these issues are company-oriented issues, supply chains also deal with customer-oriented issues involving customer satisfaction stemming from the frequency of stock-outs. For example, a customer's demand may not be fully satisfied from stock on hand. The shortage may be backordered or the sale may be lost. Either way, it is desirable to balance the cost of holding excessive inventory against the cost of not fully satisfying customer demand.

In studying such issues, modelers typically focus on the following key performance metrics, which eventually can be translated to monetary measures:

- Customer service levels (e.g., the fraction of satisfied customer demands, known as *fill rate*)
- Average inventory levels and backorder levels
- Rate and quantity of lost sales

To achieve good performance, supply chains employ inventory control policies that regulate the issuing of orders to replenish stocks. Such control policies utilize the concept of *level crossing* as follows: An (inventory) level is said to be *up-crossed* if it is reached or overshot from below, and *down-crossed*, if it is reached or undershot from above. Inventories can be reviewed continuously or periodically, and orders are typically placed when inventory levels fall below a prescribed threshold (called *reorder point*). Selected typical inventory control policies used in industry follow:

- (R,r) *inventory control policy.* The inventory has a *target level* R, and *reorder point* r. Replenishment of the inventory (i.e., product ordering) from a provider is suspended as soon as the inventory level up-crosses the target level, and remains suspended until it reaches the reorder point, whereupon replenishment is resumed. For example, when the provider is a production facility, suspension and resumption of replenishment mean the corresponding stopping and starting of production for the inventory.

- *Order up-to-R inventory control policy.* This is a special case of the (R, r) policy, where $r = R - 1$. In other words, replenishment resumes as soon as the inventory down-crosses the target level. This policy is also known as the *base-stock policy.*
- (Q, R) *inventory control policy.* The inventory has an *order quantity* Q, and *reorder point* R. Whenever the inventory level down-crosses R, an order quantity of Q is placed with the provider.

Typically, when an order is placed with a provider, there is a lag in time (called *lead time*) after which the order is received. The *lead-time demand* is the magnitude of demand that materializes during lead time, and is typically random and therefore can result in stock-outs. Consequently, to mitigate the uncertainty in lead-time demand, companies carry *safety stock,* which is extra inventory stocked to maintain good customer service levels. Companies may also elect to place new orders before previous ones are received. The *inventory position* is the inventory level plus inventory on order minus backorders. Ordering decisions are typically made based on inventory positions rather than inventory levels.

Recall that the production echelon of supply chains has already been treated in Chapter 11. In contrast, this chapter will focus on inventory management and material flow among echelons.

12.1 EXAMPLE: A PRODUCTION/INVENTORY SYSTEM

This section presents a generic model of a production/inventory system (Altiok [1997]) consisting of a production facility that is subject to failure, which supplies a warehouse with one type of product. This generic model illustrates how an inventory control policy regulates the flow of product between production and inventory facilities.

12.1.1 Problem Statement

Consider a production/inventory system where the production process (e.g., packaging) is comprised of three stages:

1. Filling each container unit (e.g., bottles)
2. Sealing each unit
3. Placing labels on each unit

For modeling purposes, the processing times of individual stages are combined into a single-stage processing time.

Figure 12.1 depicts a schematic diagram of the system. A raw-material storage source feeds the production process, and finished product units (units, for short) are stored in a warehouse. Customers arrive at the warehouse with product requests (demands), and if a request cannot be fully satisfied by on-hand inventory, the unsatisfied portion represents lost sale.

The following assumptions are made:

- There is always sufficient raw material in storage, so the process never starves.
- Product processing is carried out in lots of five units, and finished lots are placed in the warehouse. Lot processing time is uniformly distributed between 10 and 20 minutes.

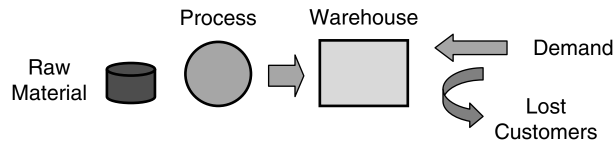


Figure 12.1 A generic production/inventory system.

- The production process experiences random failures that may occur at any point in time (see Chapter 11 for more details). Times between failures are iid exponentially distributed with a mean of 200 minutes, while repair times are iid normally distributed, with a mean of 90 minutes and a standard deviation of 45 minutes.
- The warehouse operations implement the (R, r) inventory control policy with target level $R = 500$ units and reorder point $r = 150$ units.
- The interarrival times between successive customers are iid uniformly distributed between 3 to 7 hours, and individual demand quantities are distributed uniformly between 50 and 100 units. For programming simplicity, demand quantities are allowed to be any real number in this range; however, for integer demand quantities one can use the Arena function *ANINT(UNIF(50, 100))*. On customer arrival, the inventory is immediately checked. If there is sufficient stock on hand, that demand is promptly satisfied. Otherwise, the unsatisfied portion of the demand is lost.
- The initial inventory is 250, so the production process is initially idle.

We are interested in the following performance measures:

1. Production process utilization
2. Downtime probability of the production facility
3. Average inventory level at the warehouse
4. Percentage of customers whose demand is not completely satisfied
5. Average lost demand quantity, given that it is not completely satisfied

The production/inventory problem described previously, though fairly elaborate, is still a gross simplification of real-life supply-chain systems. More realistic problems have additional wrinkles, including multiple types of products, multiple production stages, production setups, startups, cleanups, and so on.

12.1.2 Arena Model

Having studied the problem statement, we now proceed to construct an Arena model of the system under study. Figure 12.2 depicts our Arena model of the production/inventory system.

The model is composed of two segments:

- *Inventory management segment*. This segment keeps track of product unit entities. Entity definitions can be inspected and edited in the spreadsheet view of the *Entity* module from the *Basic Process* template panel. In this part of the model, the packaging process takes a unit of raw material from its queue, processes it as a batch of five, and adds the finished lot to the warehouse inventory, represented by the variable *Inventory.* Thus, when a lot entity completes processing, *Inventory* is incremented by five, and the simulation logic branches to one of two outcomes as follows: (1) If *Inventory*

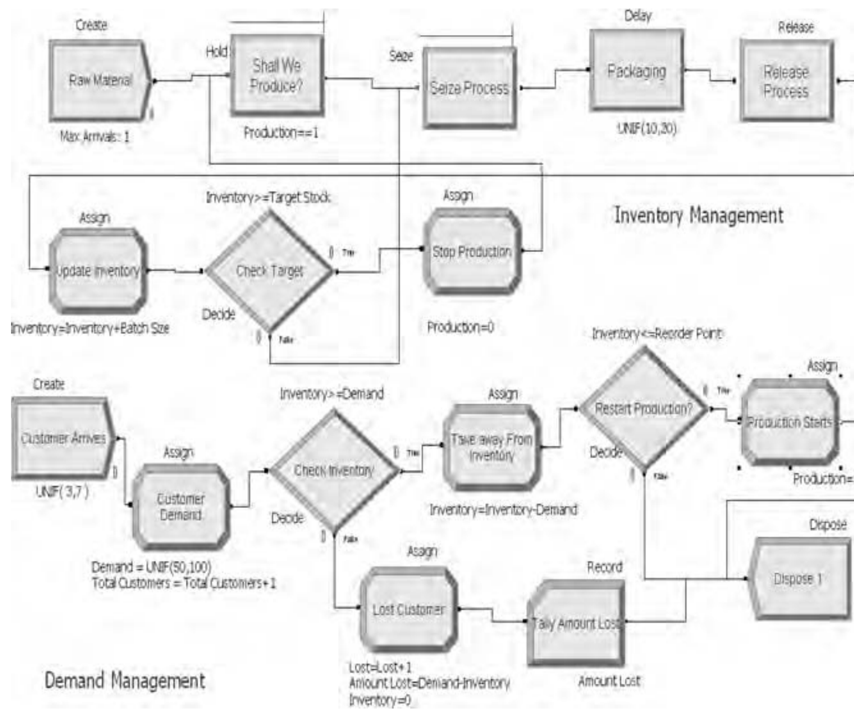


Figure 12.2 Arena model of the production/inventory system.

up-crosses the target level (variable *Target Stock*), then production stops until the reorder point (variable *Reorder Point*) is down-crossed again. (2) Processing of a new batch starts immediately when the reorder point is down-crossed (recall that we always have sufficient raw material, so the production process never starves by assumption).

- *Demand management segment*. This segment generates customers and their demands and adjusts variable *Inventory* upon customer arrival. It monitors the value of *Inventory*, and triggers resumption of suspended production when the reorder point is down-crossed. It also keeps track of lost demand (customers whose demand is not fully satisfied).

In addition, input and output data logic is interspersed in the two segments above. This logic consists of input/output modules (variables, resources, statistics, etc.) that set input variables, compute statistics, and generate summary reports.

In subsequent sections, we proceed to examine the Arena model logic of Figure 12.2 in some detail.

12.1.3 Inventory Management Segment

We begin with the inventory management portion of the logic. The *Create* module, called *Raw Material*, generates product units for the packaging (batching) operation. A *Hold* module, called somewhat whimsically *Shall We Produce?*, serves to control the start and stop of the operation by feeding product units into a sequence of *Seize*, *Delay*, and *Release* modules (called *Seize Process*, *Packaging*, and *Release Process*,

respectively, in the Arena model), all drawn from the *Advanced Process* template panel. The actual processing (packaging in our case) takes place at the *Delay* module, called *Packaging Process*, where the packaging time of a batch is specified as Unif(10, 20) minutes.

The *Create* module *Raw Material* was initialized by populating it with a single product entity (at time 0), and therefore the interarrival time is irrelevant. Furthermore, by setting the *Max Arrivals* field to 1, the *Create* module will deactivate itself thereafter. The product entity circulates in the model repeatedly, with each cycle representing a production cycle.

The circulating product entity then proceeds to the *Hold* module, called *Shall We Produce?*, to test if production is turned on. The state of production (*Off* = 0 or *On* = 1) is maintained in the variable *Production*, which is initially set to 0. Recall that the *Hold* module performs a gating function on an entity by scanning for the truth or falsity of a logical condition, as shown in the *Hold* dialog box in Figure 12.3. If the condition (in our case, *Production = 1*) is true, then the product entity proceeds to the next module. Otherwise, it waits in queue *Shall We Produce?.Queue* until the condition becomes true before being allowed to proceed. The circulating product entity then proceeds to queue *Packaging Queue* in the *Seize Process* module and seizes the server immediately (being the only one in contention for the *Packaging Process* resource).

Figure 12.4 depicts the dialog boxes of the *Seize* and *Release* modules, *Seize Process* and *Release Process*, which contain resource information. The *Seize* module has a queue called *Packaging Queue*, where product entities await their turn for one unit of resource *Packaging Process* to perform the packing operation at module *Packaging*. Once the resource becomes available, it is seized by the highest-ranking product (in this case, rank 1). While the current process is in progress, the *Seize* module bars any additional product entities from entering it. On service completion, the *Release* module (which never denies entry) releases one unit of resource *Packaging Process*, as indicated by the *Release* dialog box in Figure 12.4. Note that an entity may seize a number

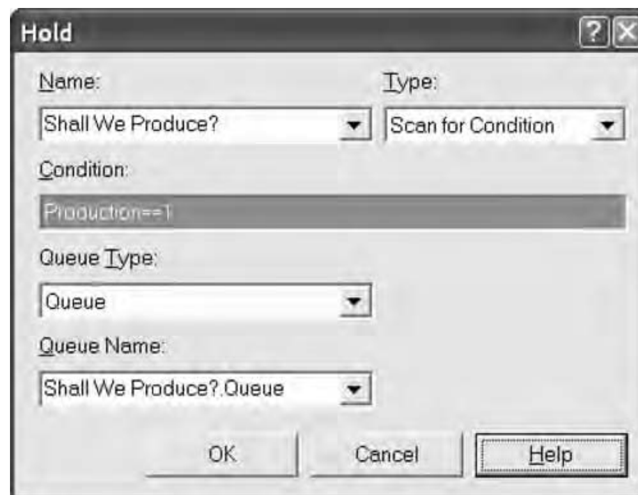


Figure 12.3 Dialog box of the *Hold* module *Shall We Produce?*.

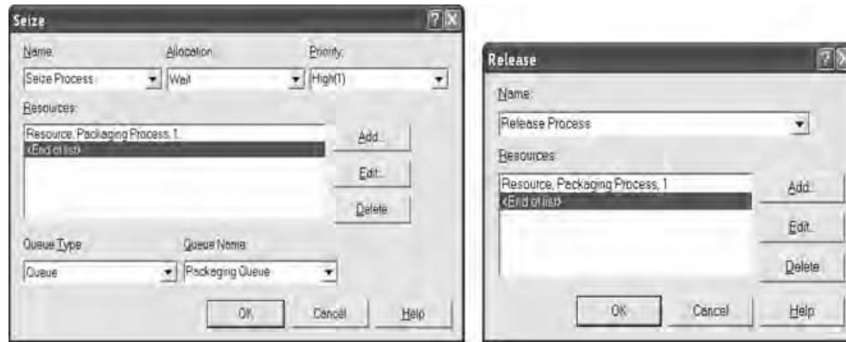


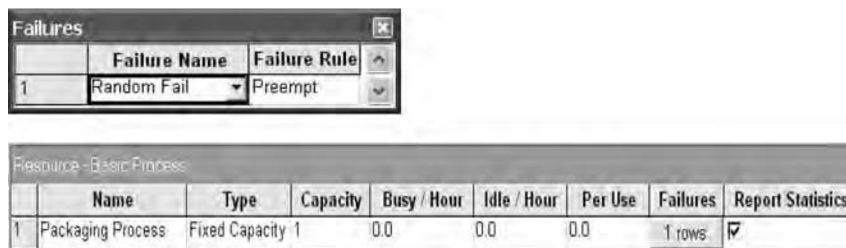
Figure 12.4 Dialog spreadsheets of the *Seize* module *Seize Process* (left) and *Release* module *Release Process* (right).

of resources (when available) simultaneously. Further, an entity may release a number of owned resources simultaneously.

A spreadsheet view of the *Resource* module with an entry for *Packaging Process* is shown at the bottom of Figure 12.5 with its *Failures* field (associating resources to failures) popped up (top). All failure/repair data (uptimes and downtimes) are specified in the *Failure* spreadsheet module (see Chapter 11), as illustrated in Figure 12.6.

The inventory level at the warehouse is maintained by the variable *Inventory*, initially set to 250. Whenever the circulating product entity (batch of five units) enters the *Assign* module, called *Update Inventory*, it adds a batch of five finished units (variable *Batch Size*) to the warehouse inventory by just incrementing *Inventory* by the batch size.

The circulating product entity then proceeds to the *Decide* module, called *Check Target*, whose dialog box is shown in Figure 12.7. Here, the circulating product entity tests whether the inventory target level has been up-crossed. The test can result in two outcomes:



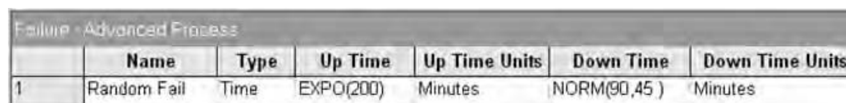
Failures

	Failure Name	Failure Rule
1	Random Fail	Preempt

Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	Failures	Report Statistics
1	Packaging Process	Fixed Capacity	1	0.0	0.0	0.0	1 rows	☑

Figure 12.5 Dialog spreadsheet of the *Resource* module showing resource *Packaging Process* (bottom) with its *Failures* dialog spreadsheet (top).



Failure - Advanced Process

	Name	Type	Up Time	Up Time Units	Down Time	Down Time Units
1	Random Fail	Time	EXPO(200)	Minutes	NORM(90,45)	Minutes

Figure 12.6 Dialog spreadsheet of the *Failure* module.

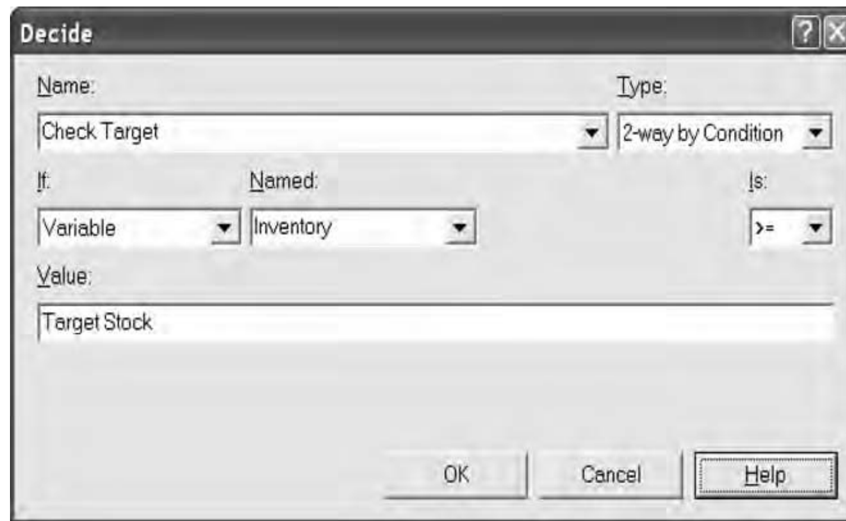


Figure 12.7 Dialog box of the *Decide* module *Check Target*.

- If the inventory target level has been up-crossed, then the circulating product entity moves on to the following *Assign* module, called *Stop Production*, and sets *Production* = *0* to signal that production is suspended.
- Otherwise, the circulating product entity does nothing.

Either way, the circulating product entity has completed its sojourn through the system and would normally be disposed of (at a *Dispose* module). However, since the *Packaging* module is never starved and no delay is incurred since its departure from the *Packaging* module, we can "recycle" the circulating product entity by always sending it back to the *Packaging* module to play the role of a new arrival. This modeling device is logically equivalent to disposing of a product entity and creating a new one. However, it is computationally more efficient, since it saves us this extra computational effort, so that the simulation will run faster. We recommend that model run optimizations by means of *circulating entities* (or any other logically correct modeling device) should be routinely sought to speed up simulation runs. The emphasis here is on "logically correct" optimizations. Note carefully that if the packaging operation were allowed to starve, then our previous computational "shortcut" would be invalid, and we would have to dispose of and create new entities throughout the simulation run.

12.1.4 Demand Management Segment

In this section, we continue with the demand management segment of the logic.

The source of customer demand at the warehouse is the *Create* module, called *Customer Arrives*, whose dialog box is depicted in Figure 12.8. The arrival pattern of customers is specified to be random with interarrival time distribution Unif(3, 7). Each arrival is an entity of type *Customer* (customer entity) with its private set of attributes (e.g., a demand quantity attribute, called *Demand*). On arrival, a customer entity first enters the *Assign* module, called *Customer Demand*, where its *Demand* attribute is

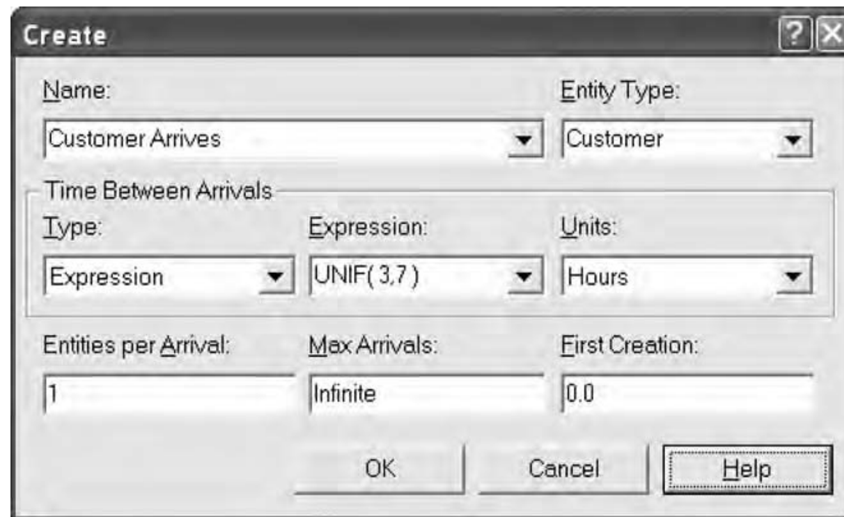


Figure 12.8 Dialog box of the *Create* module *Customer Arrives*.

assigned a random value from the Unif(50, 100) distribution. The customer entity then proceeds to the *Decide* module called *Check Inventory* to test whether the warehouse has sufficient inventory on hand to satisfy its demand. The test can result in two outcomes:

1. If the value of variable *Inventory* is greater or equal to the value of attribute *Demand*, then the current demand can be satisfied and the customer entity takes the *True* exit to the *Assign* module, called *Take Away From Inventory*, where it decrements the inventory by the demand amount. It next proceeds to the *Decide* module, called *Restart Production*, to test whether the *Reorder Level* variable has just been down-crossed. If it has, the customer entity proceeds to the *Assign* module, called *Production Start*, to set *Production* = *1*, which would promptly release the circulating product entity currently detained in the *Hold* module *Shall We Produce?*, effectively resuming the production process. Either way, the customer entity proceeds to be disposed of at the *Dispose* module, called *Dispose1*.
2. If the value of variable *Inventory* is strictly smaller than the value of attribute *Demand*, then the current demand is either partially satisfied or not at all. Either way, the customer entity proceeds to the *Assign* module, called *Lost Customer*, where it sets the *Inventory* variable to 0. It also updates the variable *Lost*, which keeps track of the number of customers to-date whose demand could not be fully satisfied (*Lost* = *Lost* + 1), and the attribute called *Amount Lost*, which keeps track of the demand lost by the current customer (*Amount Lost* = *Demand* – *Inventory*). The customer entity next enters the *Record* module, called *Tally Amount Lost*, to tally the lost quantity per customer whose demand was not fully satisfied. Finally, the customer entity proceeds to be disposed of at module *Dispose1*.

The user-defined variables involved in the model can be set or inspected by clicking on the *Variable* module of the *Basic Process* template panel. This yields the spreadsheet view of the module, as exemplified in Figure 12.9.

Variable - Basic Process

	Name	Rows	Columns	Clear Option	Initial Values	Report Statistics
1	Inventory			System	1 rows	☐
2	Batch Size			System	1 rows	☐
3	Target Stock			System	1 rows	☐
4	Production			System	1 rows	☐
5	Reorder Point			System	1 rows	☐
6	Demand			System	0 rows	☐
7	Lost			System	0 rows	☐
8	Total Customers			System	0 rows	☐

Figure 12.9 Dialog spreadsheet of the *Variable* module.

12.1.5 Statistics Collection

Figure 12.10 displays the dialog spreadsheet of the *Statistic* module for the production/inventory model. It includes a *Time-Persistent* type statistic, called *Stock on Hand*, for the *Inventory* variable, and another called *Production On*, for the expression *Production = 1*. The statistical outputs for *Inventory* consist of the average value, 95% confidence interval, and minimal and maximal values of the inventory level ever observed during the replication. The statistical output for *Production = 1* is the percentage of time this expression is true, that is, the probability that the packaging process is set to produce. (Keep in mind, however, that the production process may experience downtimes.) Recall that this method of time averaging expressions can be used to estimate any probability of interest, including joint probabilities. Statistics collection can also be affected by checking the requisite boxes under the *Report Statistics* column in the *Variable* module (see Figure 12.9).

Figure 12.10 also includes a *Frequency* type statistic, called *Process States*, which estimates the state probabilities of the packaging process, namely, the probabilities that the packaging process is *busy* or *down* (recall that, in fact, the packaging process is never idle).

The *Output* type statistic in Figure 12.10 is used to calculate the percentage of customers that experienced some lost demand. The corresponding *Expression* field there indicates that when the replication terminates, the variable *Lost* is divided by the variable *Total Customers* to yield the requisite percentage.

Figure 12.11 displays the dialog box of the *Record* module, called *Tally Amount Lost*. Whenever a customer entity enters this module, the expression (in this case, the variable) *Amount Lost* is evaluated and the resultant value is tallied. When the replication terminates, the output report will contain a *Tallies* section summarizing the average

Statistic - Advanced Process

	Name	Type	Expression	Report Label	Frequency Type	Resource Name	Report Label	Categories
1	Stock on Hand	Time-Persistent	Inventory	Stock on Hand				0 rows
2	Process States	Frequency			State	Packaging Process	Process States	0 rows
3	Production On	Time-Persistent	Production==1	Production On				0 rows
4	Lost Percentage	Output	Lost/Total Customers	Lost Percentage				0 rows

Figure 12.10 Dialog spreadsheet of the *Statistic* module.

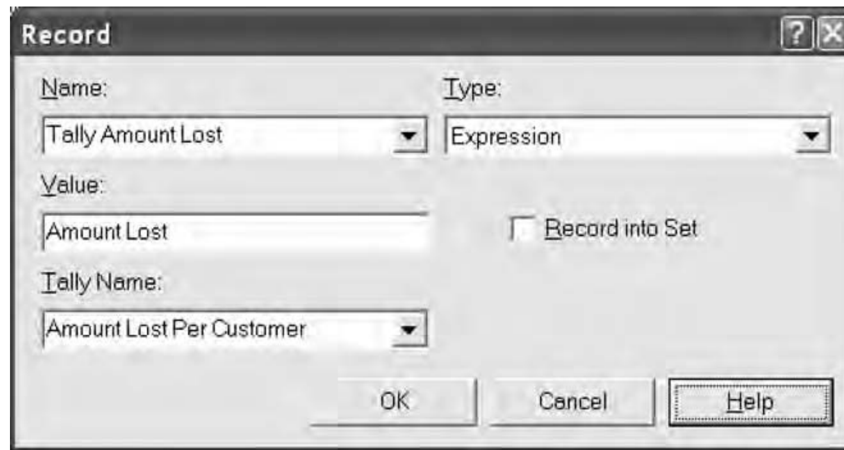


Figure 12.11 Dialog box of the *Record* module *Tally Lost Amount*.

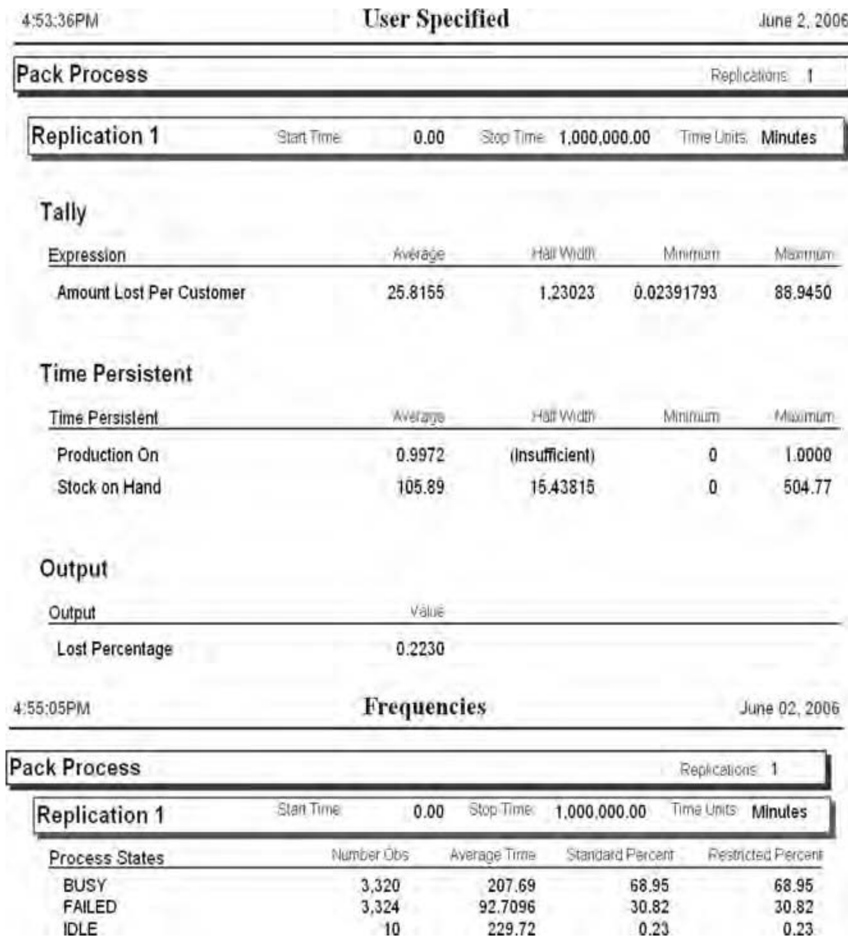
amount lost per customer. Note carefully that since we tally a loss only when it occurs, this average is conditional on the event that a customer actually experienced a loss. If we were to include in the tally a value of 0 (no loss) for every customer with a fully satisfied demand, then the resulting average would provide us with the unconditional average of lost demand per customer. Obviously, the conditional average loss must be larger than the unconditional average loss.

12.1.6 Simulation Output

Figure 12.12 displays reports of the results of a simulation run of length 1,000,000 minutes (slightly less than 2 years).

An inspection of the results of Figure 12.12 is quite instructive vis-a-vis the performance of the production/inventory model.

- The expression *Amount Lost Per Customer* in the *Tally* section indicates that the average lost demand per customer who experienced some loss was about 26 units. The *Half Width* column estimate the half-length of respective confidence intervals at the 95% confidence level.
- The *Stock on Hand* variable in the *Time Persistent* section shows that the inventory up-crossed occasionally its target level. However, this happened only rarely, since the *Production* variable in the *Time Persistent* section assumed the value *On* = *1* about 99.72% of the time. Most of the time the inventory level was lower, averaging around 106 units—well below the reorder level, $r = 150$.
- The *Output* statistic *Lost Percentage* in the *Output* section reveals that 22.3% of the customers had their demand either partially satisfied or not satisfied at all.
- Finally, the *Frequencies* section estimates the state probabilities (BUSY, FAILED, and IDLE) of the production process. These are, respectively, 68.95% (resource utilization), 30.82% (downtime probability), and 0.23% (idle probability). Note that the average uptime and the average downtime are very close to their theoretical values (in this case, the downtime probability is equal to the ratio of expected downtime to expected cycle length).



4:53:36PM **User Specified** June 2, 2006

Pack Process Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 1,000,000.00 Time Units: Minutes

Tally

Expression	Average	Half Width	Minimum	Maximum
Amount Lost Per Customer	25.8155	1.23023	0.02391793	88.9450

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Production On	0.9972	(Insufficient)	0	1.0000
Stock on Hand	105.89	15.43815	0	504.77

Output

Output	Value
Lost Percentage	0.2230

4:55:05PM **Frequencies** June 02, 2006

Pack Process Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 1,000,000.00 Time Units: Minutes

Process States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	3,320	207.69	68.95	68.95
FAILED	3,324	92.7096	30.82	30.82
IDLE	10	229.72	0.23	0.23

Figure 12.12 Simulation results for the production/inventory model.

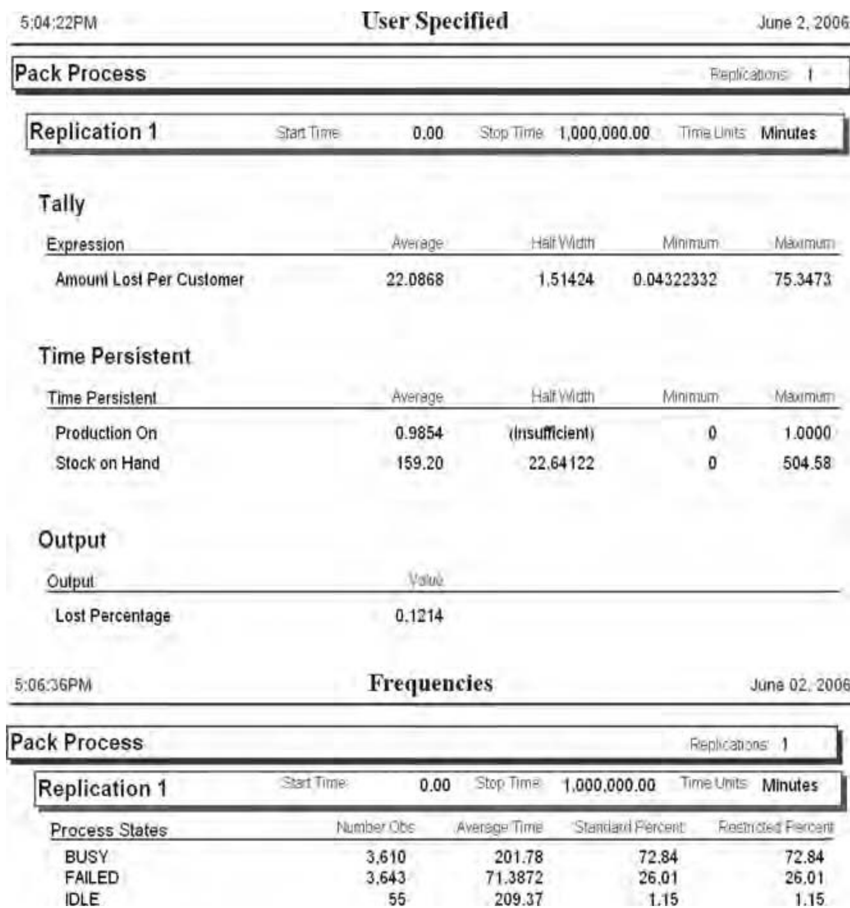
Much more information can be gleaned from the statistical output of Figure 12.12, but these items will not be pursued further (we encourage you to complete the analysis of the simulation results).

12.1.7 Experimentation and Analysis

In this section, we will experiment with selected model parameters, so as to improve the customer service level of *fill rate* (probability that the demand of an arriving customer is fully satisfied). Clearly, the fill rate is quite low. Our performance metric of interest is the complementary probability of partially or fully unsatisfied demands. Figure 12.12 estimates this probability as 22.3% (the value of *Lost Percentage* in the *Output* section). How can we modify the system to decrease this probability to an "acceptable" level? In inventory-oriented systems such as the one under consideration, the only way to increase the fill rate is to increase the level of inventory on hand. In our case, we may attempt to achieve this goal in two ways.

The first way of increasing the fill rate is to modify the original production/inventory system by investing more in maintenance activities. This will reduce downtimes and make the process more available for production. Figure 12.13 displays the estimated performance changes in the modified production/inventory model when the average repair time is reduced to 70 minutes with a standard deviation of 30 minutes. The reduction in repair time in Figure 12.13 represents an improvement of about 16% in maintenance activities, as compared with the downtime probabilities in Figure 12.12. The resulting reduction in the percentage of partially or fully unsatisfied demands has dropped significantly from 22.3% to 12.14%. This reduction represents a more than 45% improvement in the fill rate.

The second way of increasing the fill rate is to modify the original production/inventory system by increasing the reorder level parameter. The inventory level would then hit the reorder level sooner and production would resume earlier, thereby decreasing the chance of an inventory shortfall. Figure 12.14 displays the estimated performance changes in the modified production/inventory model when the reorder level parameter is doubled.


5:04:22PM **User Specified** June 2, 2006

Pack Process Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 1,000,000.00 Time Units: Minutes

Tally

Expression	Average	Half Width	Minimum	Maximum
Amount Lost Per Customer	22.0868	1.51424	0.04322332	75.3473

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Production On	0.9854	(Insufficient)	0	1.0000
Stock on Hand	159.20	22.64122	0	504.58

Output

Output	Value
Lost Percentage	0.1214

5:06:36PM **Frequencies** June 02, 2006

Pack Process Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 1,000,000.00 Time Units: Minutes

Process States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	3,610	201.78	72.84	72.84
FAILED	3,643	71.3872	26.01	26.01
IDLE	55	209.37	1.15	1.15

Figure 12.13 Simulation results for the production/inventory model with reduced downtimes.

5:12:36PM **User Specified** June 2, 2006

Pack Process Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 1,000,000.00 Time Units: Minutes

Tally

Expression	Average	Half Width	Minimum	Maximum
Amount Lost Per Customer	25.8540	1.55079	0.02697518	91.0525

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Production On	1.0000	(Insufficient)	0	1.0000
Stock on Hand	103.52	11.96894	0	462.74

Output

Output	Value
Lost Percentage	0.2198

Figure 12.14 Simulation results for the production/inventory model with doubled reorder level.

However, the second way of increasing the fill rate does not work well for our system, because the inventory level was below the reorder level most of the time, and consequently, the production process was busy most of the time anyway. The percentage of unsatisfied customers is marginally improved from 22.3% to about 21.98%, but we would like to do better. This analysis leads us to conclude that a significant improvement in the fill rate can be achieved only by improving the production process, rather than by modifying our inventory replenishment policy.

12.2 EXAMPLE: A MULTIPRODUCT PRODUCTION/INVENTORY SYSTEM

This section extends the production/inventory system studied in Section 12.1 from a single product to multiple ones. The new system again operates under the lost sales discipline.

12.2.1 Problem Statement

The extended system is depicted in Figure 12.15. Accordingly, the production facility produces product types 1, 2, and 3, and these are supplied to three distinct incoming customer streams, denoted by types 1, 2, and 3, respectively. The production facility produces batches of products, switching from production of one product type to another, depending on inventory levels. However, products have priorities in production, with product 1 having the highest priority and product 3 the lowest.

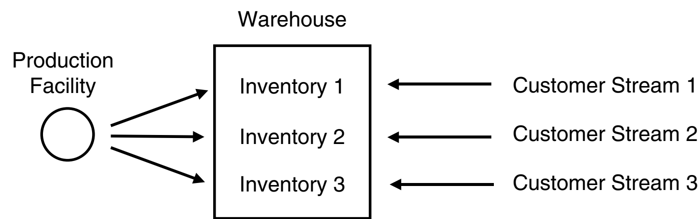


Figure 12.15 A production/inventory model with three products.

A raw-material storage feeds the production process, and finished units are stored in the warehouse. Customers arrive at the warehouse with product demands, and if a demand cannot be fully satisfied by inventory on hand, the unsatisfied portion represents lost sales. Each product has its own parameters as per Table 12.1. The following assumptions are made:

- There is always sufficient raw material in storage, so the production process never starves.
- Processing of each product type is carried out in lots of five units, and finished lots are placed in the warehouse. Lot processing time is deterministic as per Table 12.1.
- The production process experiences random failures, which may occur only while the production facility is busy. Times between failures are exponentially distributed with a mean of 200 minutes, while repair times are normally distributed, with a mean of 70 minutes and a standard deviation of 30 minutes (recall that if a negative repair time is sampled, another sample is generated until a non-negative value is obtained).

The warehouse implements a separate (R, r) inventory control policy for each product type. Note that this is actually a convenient policy when a resource needs to be shared among multiple types of products. For instance, when the production process becomes blocked, it may be assigned to another product. In our case, the production facility may switch back and forth between the two highest priority products a few times before it turns to product type 3. It should be pointed out that other production-switching policies may also be employed, such as switching after the current product unit is produced, rather than the current production run.

Assume that we want to simulate the system for 100,000 hours, and estimate the following statistics:

Table 12.1

Parameters of the production/inventory model with three product types

Product Type	Target Level	Reorder Point	Initial Inventory	Processing Time (hours)	Demand Interarrival Time (hours)	Demand Quantity
1	100	50	75	1	Exp(16)	U(4, 10)
2	200	100	150	0.6	Exp(8)	U(10, 15)
3	300	150	200	0.3	Exp(4)	U(20, 30)

- Production-facility utilization, by product
- Production-facility downtime probability
- Average inventory level, by product
- Percentage of customers whose demand is not completely satisfied, by product
- Average lost demand quantity given that it is not completely satisfied, by product

12.2.2 Arena Model

Since the current model is an extension of the earlier single-product version, its structure is similar to that of Figure 12.2. Accordingly, it has two main segments: inventory management and demand management. The inventory management segment is modified to model multiple inventory types with shared storage. It again circulates a product entity to produce batches of each product type, monitor inventories, and stop production when necessary. The demand management segment is modified to generate multiple types of customers and their demands. It also models the drawing down on inventories, and initiates production orders for each product type as necessary. In the next section, we examine the model logic of each segment in some detail.

12.2.3 Inventory Management Segment

Figure 12.16 depicts the modified inventory management segment that models the production facility. The *Create* module, called *Raw Material*, operates precisely as in the previous example: It creates a single product entity that controls the packaging operation (see Section 12.1.3) of all product types.

The circulating product entity is detained in the *Hold* module, called *Shall We Produce?*, until the following condition becomes true:

$$\textit{Production Order (1)} + \textit{Production Order (2)} + \textit{Production Order (3)} > 0,$$

as shown in the *Condition* field of the dialog box in Figure 12.17. Here, the quantities *Production Order(k)*, $k = 1, 2, 3$, are vector elements that assume each a value of 1 or 0 representing the status of pending orders by product type. More specifically, *Production Order(k)* $= 1$ indicates that a production order is present for product k, while *Production Order(k)* $= 0$ indicates absence thereof. When the logical expression in the *Condition* field is true, this indicates that *at least* one product order is present.

When multiple product orders are present, it is necessary to decide which product type to produce next. To this end, the circulating product entity proceeds to the *Search* module, called W*hich Product to Switch to?*, whose dialog box is displayed in Figure 12.18. An incoming entity into this module triggers a search. Using an index variable that ranges from the *Starting Value* field contents to the *Ending Value* field contents, the *Search* module can perform various searches over the specified range, guided by a logical condition specified in the *Search Condition* field.

Various types of search can be selected in the *Type* field. For example, the modeler can search a range of expressions for the first one with a prescribed value; it can also search an entity queue or entity batch for the first entity with a prescribed attribute

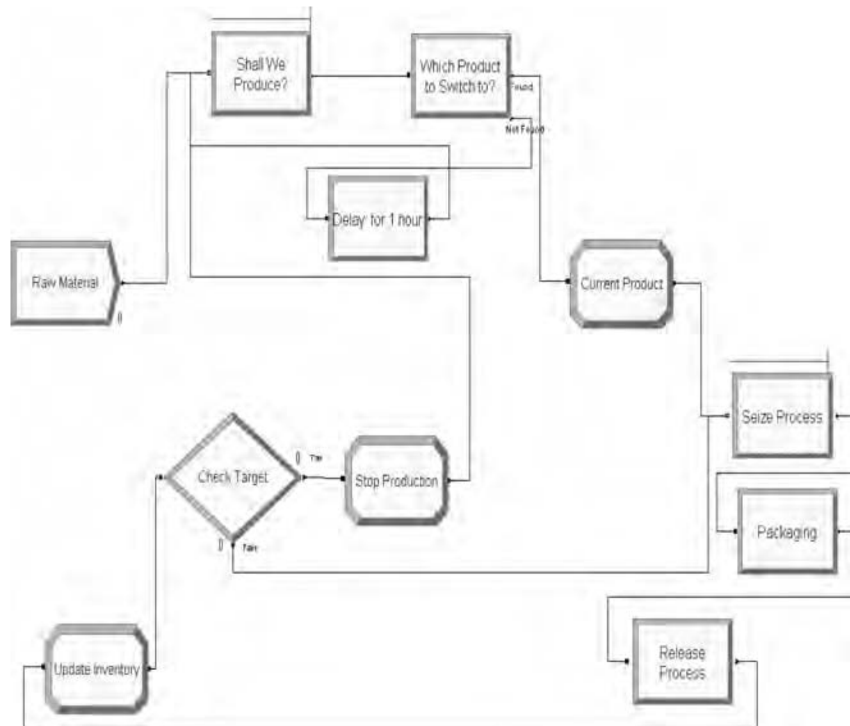


Figure 12.16 Arena model of the inventory management segment with three product types.

Figure 12.17 Dialog box of the *Hold* module *Shall We Produce?*.

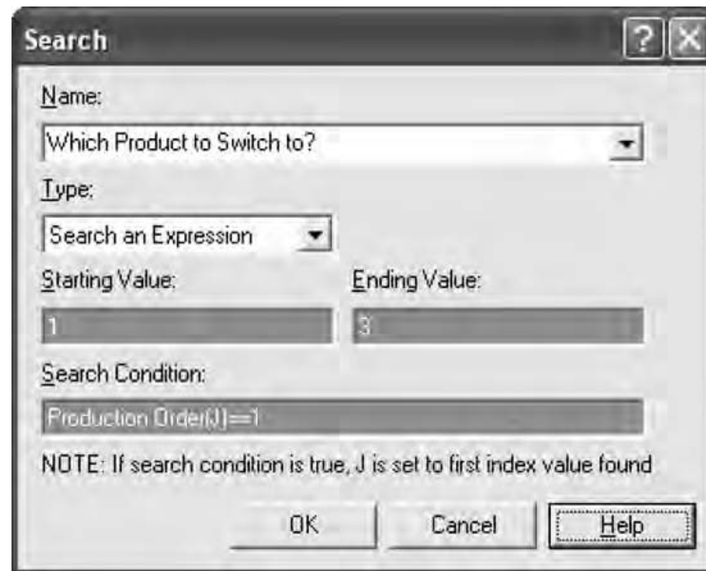


Figure 12.18 Dialog box of the *Search* module *Which Product to Switch to?*

value, or one that satisfies a prescribed condition. The search result is returned in the Arena variable *J*, as shown in the *Search Condition* field; a positive *J* value returns the result of a successful search, while a returned value of $J = 0$ signals a failed search.

In our case, the search is for the first product *k* in the range {1, 2, 3}, which satisfies the condition *Production Order(k) == 1*. Recall that in our model, product types have production priorities with lower-numbered products having higher priority. Consequently, the search for the highest-priority pending order starts with product type 1 and proceeds in ascending product type number.

Following search completion, the circulating product entity enters the *Assign* module, called *Current Product*, whose dialog box is displayed in Figure 12.19. Here, the circulating product entity notes which product type to switch production to. This is accomplished by setting the variable *Current Item* to variable *J*, which now contains the product type index returned from the just completed search. Note that it is appropriate here for *Current Item* to be a variable, since only one entity circulates in the inventory management segment. If multiple entities were to roam this segment, then *Current Item* would have to be an attribute in order to maintain the integrity of its value.

Next, the circulating product entity proceeds to traverse a standard *Seize-Delay-Release* sequence of three modules (*Seize Process*, *Packaging*, and *Release Process*, respectively), which model the production of the current product type, as indicated by the *Current Item* variable. While the *Seize* and *Release* modules are identical to their counterparts in Section 12.1, the *Delay* module implements here a product-dependent processing time (using the values from Table 12.1), as shown in Figure 12.20. The requisite processing times are stored in the vector *Processing Time*, indexed by product types.

When processing of the current batch is done, that batch should be placed in the warehouse. To this end, the circulating product entity proceeds to the *Assign* module, called *Update Inventory*, whose dialog box is shown in Figure 12.21. The inventory

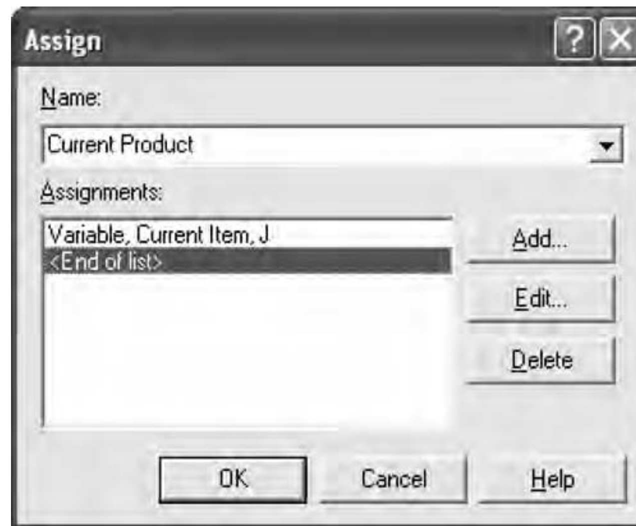


Figure 12.19 Dialog box of the *Assign* module *Current Product*.

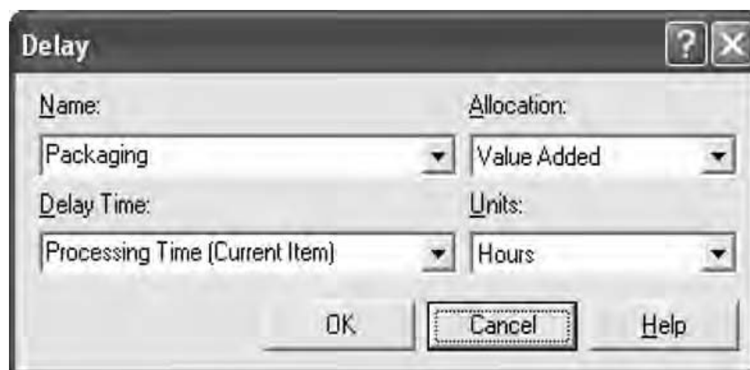


Figure 12.20 Dialog box of the *Delay* module *Packaging*.

level of each product is modeled by a vector, called *Inventory*, which is indexed by product type. In a similar vein, the batch size of each product type is stored in the vector *Batch Size* of the same dimension. To update the inventory level of the current product type, the corresponding inventory level is simply incremented by the just-produced batch size (see the assignment in the first line of the *Assignments* field).

Clicking the Edit. . .button in Figure 12.21 pops up the *Assignments* dialog box for that line, displayed in Figure 12.22. The corresponding *Assignments* expression is:

Inventory(Current Item) = Inventory(Current Item) + Batch Size(Current Item)

where the *Other* option in the *Type* field indicates an assignment to a vector element. Another style of assignment to vectors will be introduced in Section 12.2.4.

At this point, the circulating product entity needs to check whether the target value of the current product type has been reached, so it proceeds to the *Decide* module, called

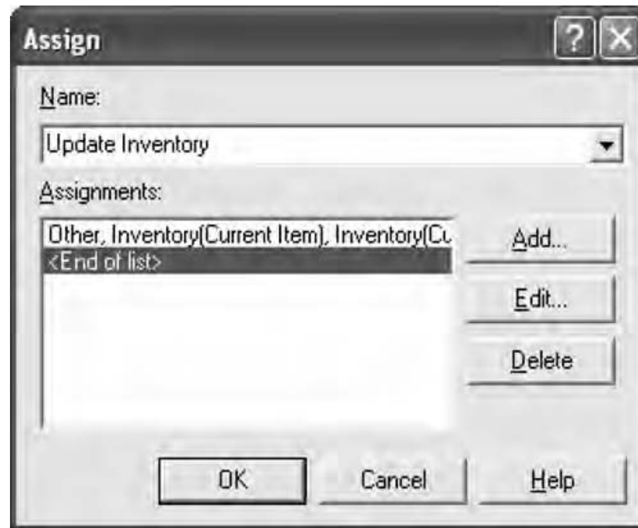


Figure 12.21 Dialog box of the *Assign* module *Update Inventory*.

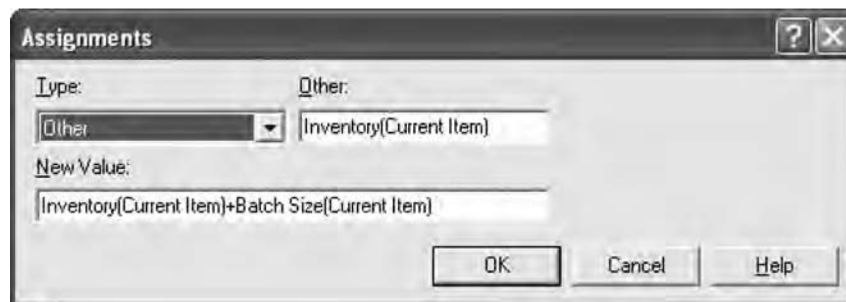


Figure 12.22 Dialog box of the *Assignments* field from the *Assign* module *Update Inventory*.

Check Target, whose dialog box is displayed in Figure 12.23. Note that inventory target values for product types 1, 2, and 3 are stored in the vector elements *Target Stock(k)*, $k = 1, 2, 3$, respectively, as evidenced by the logical expression in the *Value* field. The circulating product entity checks if the target level of the current product type has been reached. If it has, the product entity would take the exit branch from the *Check Target* module to the *Assign* module, called *Stop Production*, whose dialog box is shown in Figure 12.24. Here, the circulating product entity stops the production of the current production type by executing the assignment

$$Production\ Order(current\ item) = 0.$$

It then loops back to the *Hold* module *Shall We Produce?* (see Figure 12.16) to identify the next product type (if any) to which production is to be switched.

If, however, the target level of the current product type has not been reached, the circulating product entity would take the exit branch from the *Check Target* module to

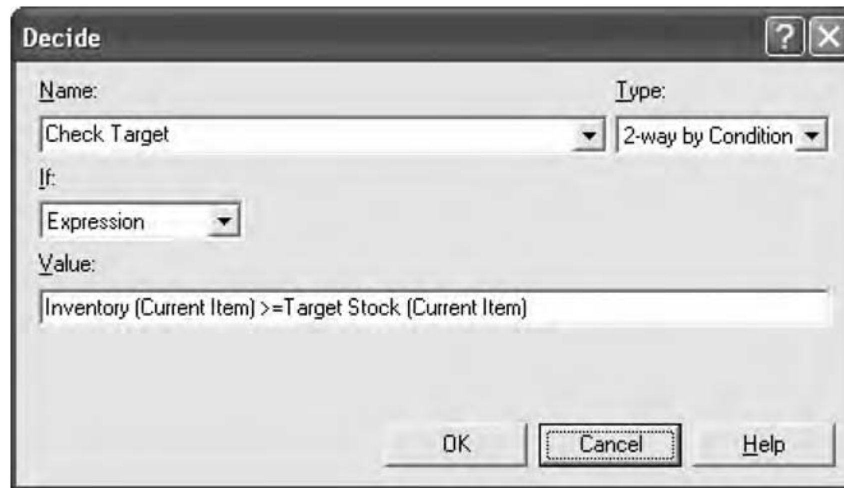


Figure 12.23 Dialog box of the *Decide* module C*heck Target*.

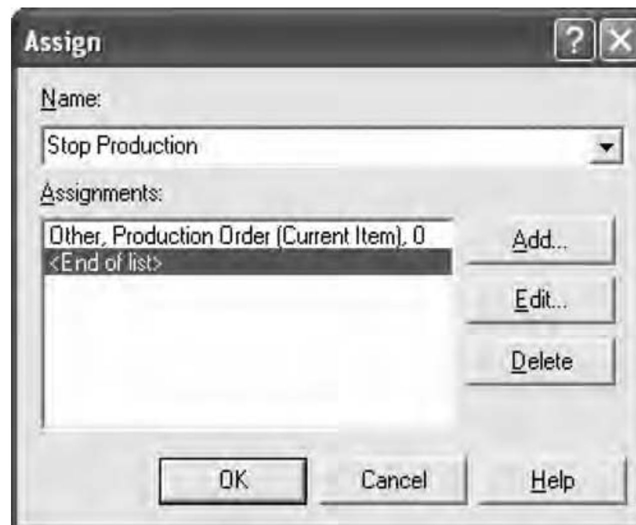


Figure 12.24 Dialog box of the *Assign* module *Stop Production*.

the *Seize* module *Seize Process* (see Figure 12.16) to process the next unit of the current product type.

Finally, recall that the production facility experiences failures only in the busy state. This constraint is specified in the *Failure* module dialog spreadsheet, as shown in Figure 12.25. Here, the *Uptime in this State Only* field is set to *BUSY* to indicate that the production facility does not "age," unless it is busy producing products. In contrast, leaving this field blank would allow failures to occur any time, even when the production facility is idle, since "aging" takes place continually.

Failure - Advanced Process

	Name	Type	Up Time	Up Time Units	Down Time	Down Time Units	Uptime in this State only
1	Random Fail	Time	EXPO(200)	Minutes	NORM(70,30)	Minutes	BUSY

Figure 12.25 Dialog spreadsheet of the *Failure* module.

12.2.4 Demand Management Segment

Figure 12.26 depicts the modified demand management segment, which models demand arrivals at the inventory facility. In this model, the arrival processes of the three product types are modeled by a tier of three *Create* modules on the left side.

Figure 12.27 depicts the dialog box of the *Create* module, which generates customer entities that carry demand for the first product type.

In a similar vein, the demand processes of the three product types are modeled by a tier of three *Assign* modules, where the demand-product type and quantity are assigned to an incoming customer entity. Figure 12.28 depicts the dialog box of the *Assign* module, which generates customer demand quantities for the first product type. In this module, a product type is assigned to the customer entity's attribute *Type*, and then a demand quantity is sampled and assigned to the customer entity's attribute *Demand*.

In addition, the model keeps track of the total number of arrivals of each customer type in a vector of counters, called *Total Customers*, by incrementing the corresponding counter. The vector *Total Customers* in the third assignment was declared and assigned in the corresponding *Assignments* module, whose dialog box is shown in Figure 12.29. Here, the *Type* field was set to the *Variable Array (1D)* option, although the modeler can alternatively use the *Other* option, as illustrated in Figure 12.22.

Next, the customer entity needs to check whether there is sufficient inventory in the warehouse to satisfy the requested quantity of the requisite product type. To this end, all customer entities proceed to the *Decide* module, called *Check Inventory*, whose dialog box is depicted in Figure 12.30.

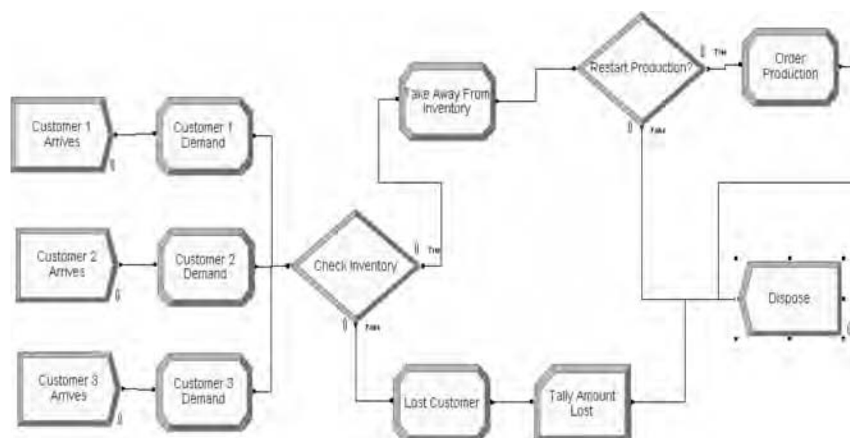


Figure 12.26 Arena model of the demand management segment for the production/inventory system with three product types.

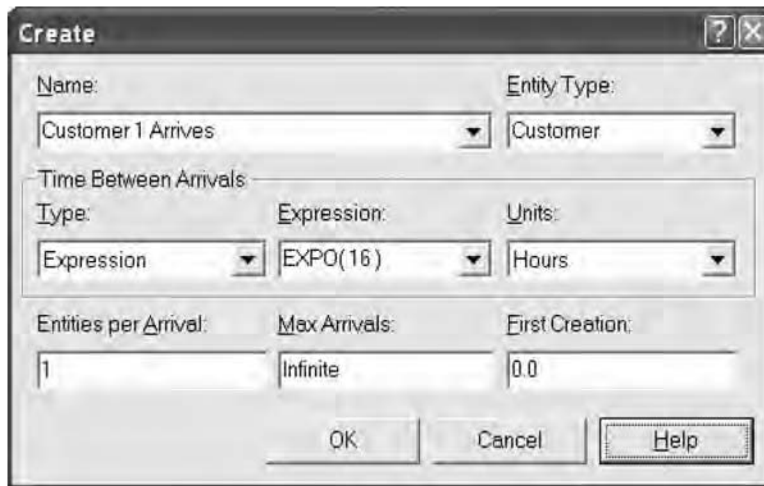


Figure 12.27 Dialog box of the *Create* module that generates type 1 customer entities.

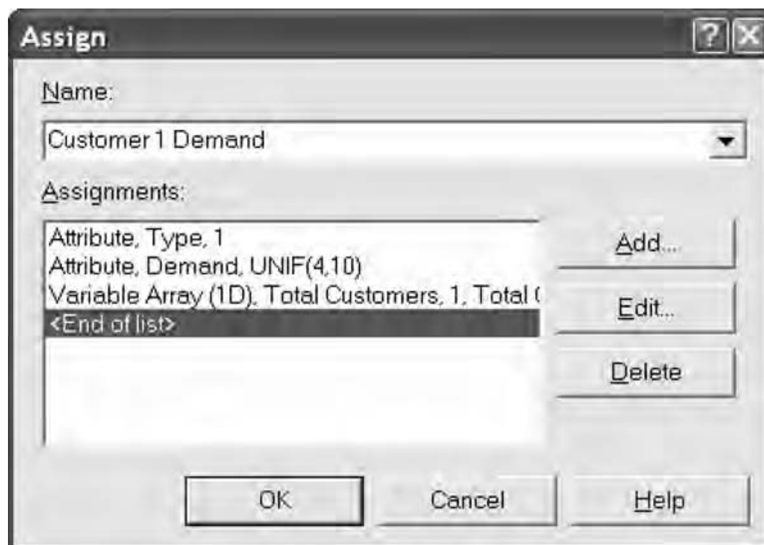


Figure 12.28 Dialog box of the *Assign* module that generates demand quantities for type 1 customer entities.

The availability of sufficient product is expressed by the logical expression in the *Value* field, keeping in mind that the product type information is stored in the *Type* attribute of the customer entity. If there is sufficient inventory in stock, then the demand of the current customer entity can be satisfied. In this case, the customer entity takes the exit branch to the *Assign* module, called *Take Away From Inventory*, where the level of the corresponding inventory type is decremented by the demand quantity. Figure 12.31 displays the dialog box of this module (bottom) and its associated *Assignments* dialog box (top), which pops up when the *Edit...* button is clicked.

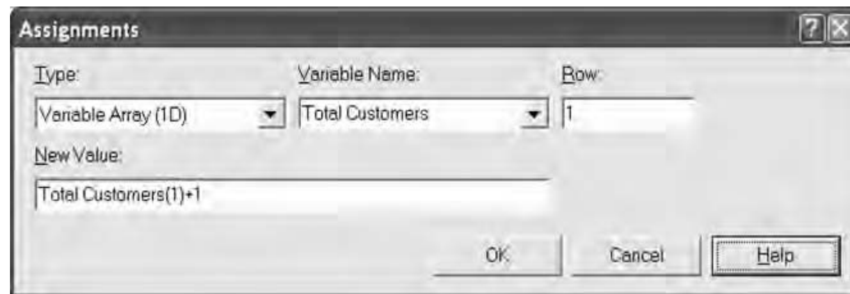


Figure 12.29 Dialog box of *Assignments* for declaring and assigning the *Total Customers* vector.

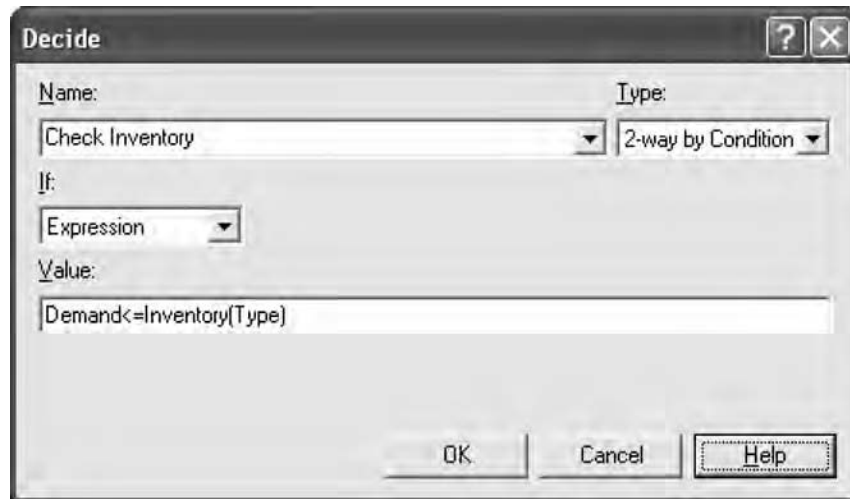


Figure 12.30 Dialog box of the *Decide* module *Check Inventory.*

The customer entity then enters the *Decide* module, called *Restart Production?*, to check whether the reorder level is subsequently down-crossed as shown in the logical expression in the dialog box of Figure 12.32. Note that the logical expression in the *Value* field compares two vector elements: the inventory level of the currently requested product type and the reorder level of the same product type. Two cases are then possible: (1) there was sufficient inventory on hand to satisfy the incoming demand, or (2) there was insufficient inventory on hand. We next describe the scenarios that result from each case.

Consider first the case of sufficient on-hand inventory. In this case, the customer entity proceeds to the *Assign* module, called *Order Production*, to initiate a production order, as shown in its dialog box (bottom) and associated *Assignments* module (top) in Figure 12.33. Note that production is initiated by simply assigning a value of 1 to the variable in vector element *Production Order(Type)*, which indicates shortage of that product type. Consequently, this shortage will be attended to in due time by the

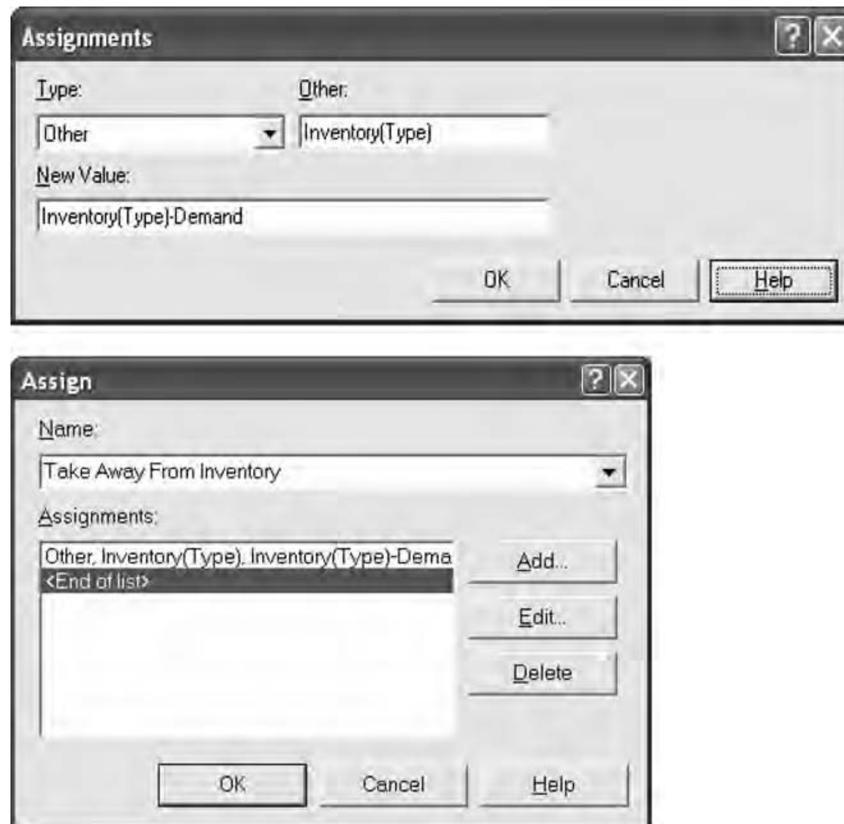


Figure 12.31 Dialog spreadsheets of the *Assign* module *Take Away From Inventory* (bottom) and its associated *Assignments* dialog box (top).

production facility that attends to production orders according to their priority structure. Note further that these variables may be set to 1 multiple times by customer entities entering this module during periods of shortage. While these assignments may be redundant, they are harmless in that the correctness of the model's state is maintained; moreover, the model's Arena logic is greatly simplified. Finally, having finished all its tasks in the model, the customer entity enters the dispose module, called *Dispose*, to be removed from the model.

Consider next the case of insufficient on-hand inventory, which results here in the customer demand being lost. The customer entity then takes the other exit branch from the *Decide* module *Check Inventory* to the *Assign* module *Lost Customer*, whose dialog box (bottom) and two associated *Assignments* modules (middle and top) are depicted in Figure 12.34. Three assignments are executed in the *Assignments* field of the *Assign* module. The first assignment (middle dialog box) increments by 1 the appropriate element of the vector *Lost*, in order to track the number of lost customers by the type of product requested. The second assignment (top dialog box) records the quantity lost in the *Amount Lost* attribute of the current customer entity. Finally, the

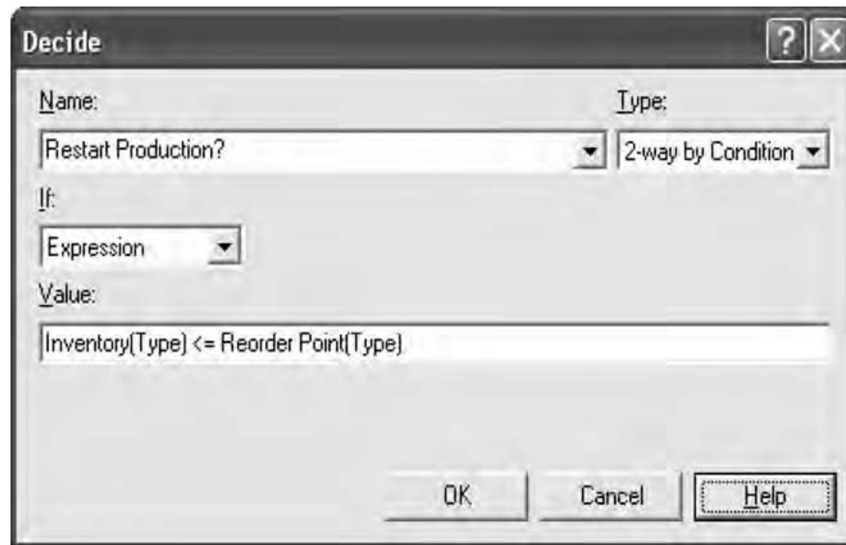


Figure 12.32 Dialog box of the *Decide* module *Restart Production?*.

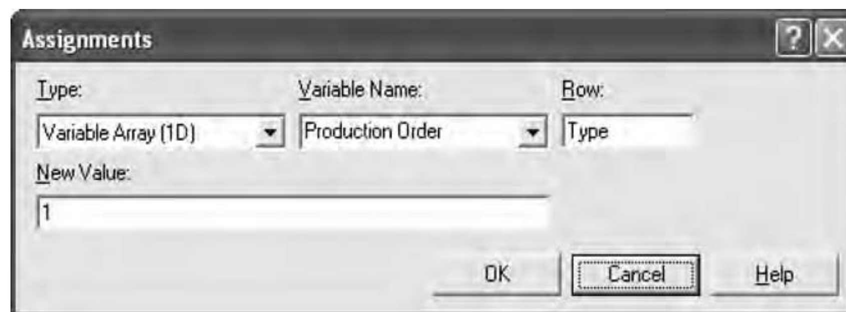


Figure 12.33 Dialog box of the *Assign* module *Order Production* (bottom) and its associated *Assignments* dialog box (top).

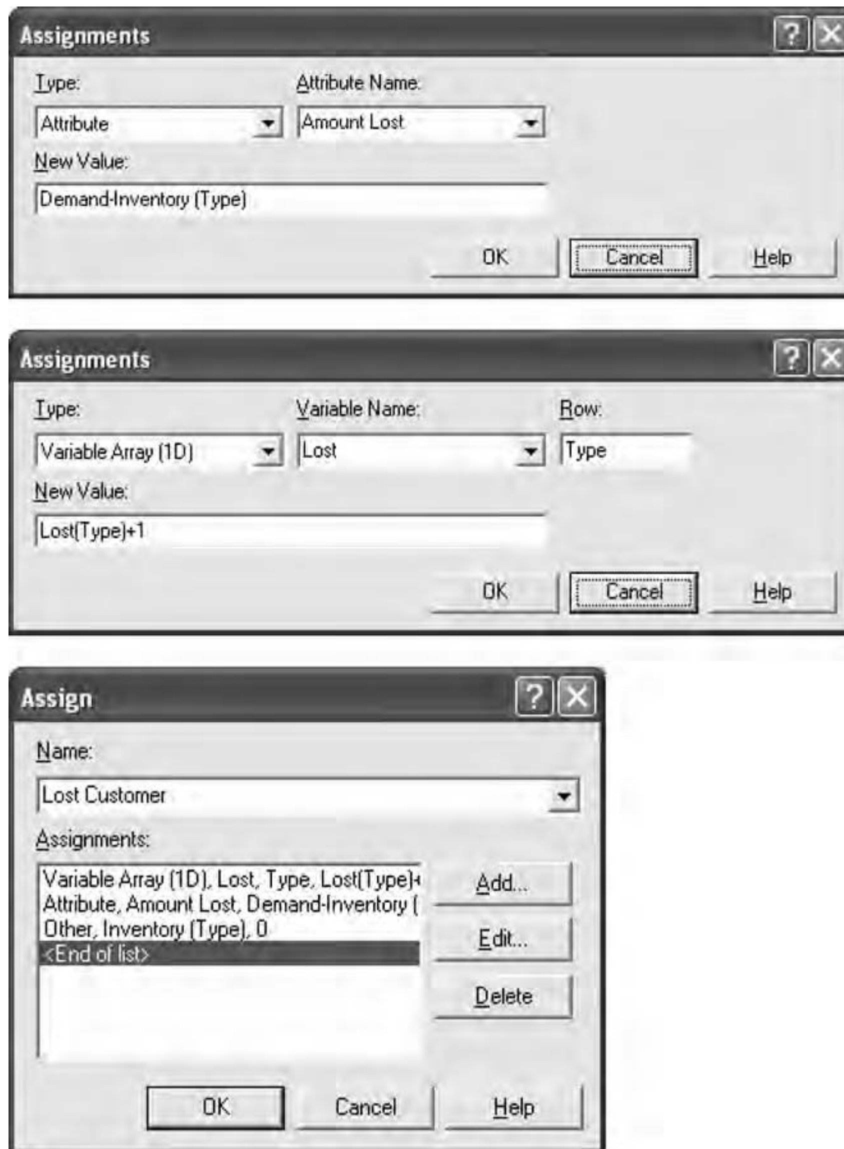


Figure 12.34 Dialog boxes of the *Assign* module *Lost Customer* (bottom) and its two associated *Assignments* modules (middle and top).

third assignment sets the inventory level of this product to 0. Note carefully that the order of assignments is important.

Next, the customer entity proceeds to the *Record* module, called *Tally Amount Lost*, to tally the current lost amount in the set *Lost Quantities*, as shown in the dialog box of Figure 12.35. Finally, the customer entity enters the dispose module, called *Dispose*, to be removed from the model.

Record

Name: Tally Amount Lost

Type: Expression

Value: Amount Lost

Record into Set

Tally Set Name: Lost Quantities

Set Index: TYpe

OK Cancel Help

Figure 12.35 Dialog box of the *Record* module *Tally Amount Lost*.

12.2.5 Model Input Parameters and Statistics

This section details the input parameters of the model and its statistics. Recall that all input parameters, including vector-valued ones, are declared in the *Variable* module from the *Basic Process* template.

Figure 12.36 displays the *Variable* dialog spreadsheet (bottom), as well as the initial values of the *Inventory* variable (top). The dimension of a variable is indicated by the number of rows indicated in the button labels under the *Initial Values* column in Figure 12.36. Accordingly, the label *0 rows* specifies no initial values, the label *1 rows*

Initial Values

1	75
2	150
3	200

Variable - Basic Process

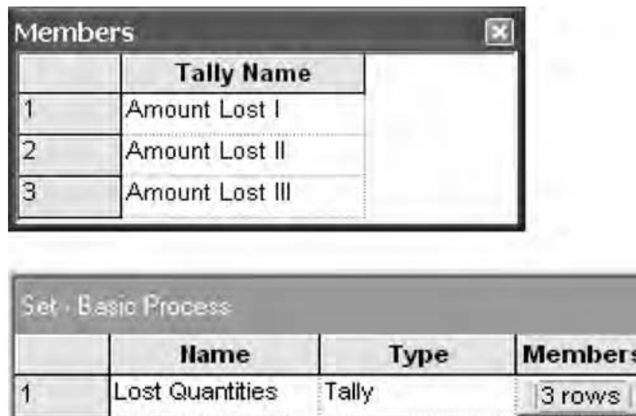
	Name	Rows	Columns	Clear Option	Initial Values	Report Statistics
1	Inventory	3		System	3 rows	
2	Batch Size	3		System	1 rows	
3	Target Stock	3		System	3 rows	
4	Production			System	1 rows	
5	Reorder Point	3		System	3 rows	
6	Total Customers	3		System	0 rows	
7	Current Item			System	0 rows	
8	Lost	3		System	0 rows	
9	Processing Time	3		System	3 rows	
10	Production Order	3		System	0 rows	

Figure 12.36 Dialog spreadsheet of the *Variable* module (bottom) and the *Initial Values* dialog spreadsheet of variable *Inventory* (top).

specifies a single initial value for an ordinary (one-dimensional) variable (or a vector of identical values), whereas a label of the form *n rows* (for $n > 1$) specifies a vector-valued variable with the indicated dimension. The *Clear Option* column specifies the simulation time (if any) to reset to the initial value(s) specified. Typically, the *System* option is selected to activate initialization only at the beginning of each replication. Statistics collection of one-dimensional variables is enabled by checking the corresponding boxes under the *Report Statistics* column. However, statistics of vector-valued variables can be collected through the *Statistic* module spreadsheet, and will be illustrated next.

Recall that we used the *Set* element to track lost demand by product type (1, 2, or 3) in the *Record* module called *Tally Amount Lost*. To this end, we declare a three-dimensional *Set*, called *Lost Quantities*, in the *Set* module spreadsheet (from the *Basic Process* template panel), as shown in Figure 12.37.

Finally, the specification of statistics collection can also be made in the *Statistic* module spreadsheet, as illustrated in Figure 12.38. Note that statistics collected on vector-valued variables are specified separately for each vector element. The *Stock on Hand* and *Production Order* statistics (by product type) are time averages, and as such of type



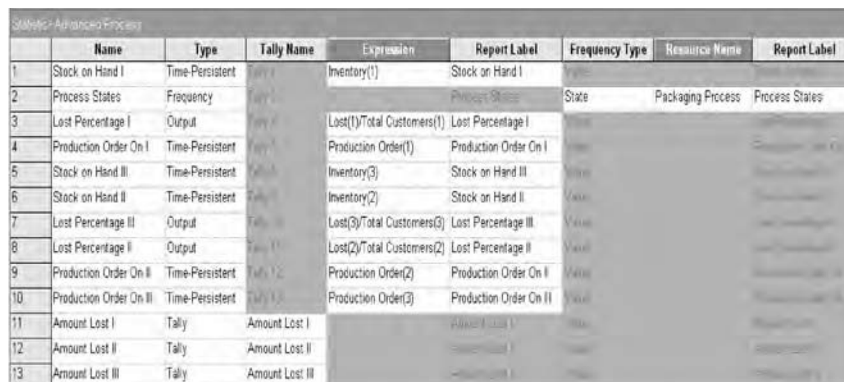
Members

	Tally Name
1	Amount Lost I
2	Amount Lost II
3	Amount Lost III

Set - Basic Process

	Name	Type	Members
1	Lost Quantities	Tally	3 rows

Figure 12.37 Dialog spreadsheet of the *Set* module (bottom) and the *Members* dialog spreadsheet for *Tally* set *Amount Lost* (top).



Statistic - Advanced Process

	Name	Type	Tally Name	Expression	Report Label	Frequency Type	Resource Name	Report Label
1	Stock on Hand I	Time-Persistent		Inventory(1)	Stock on Hand I			
2	Process States	Frequency				State	Packaging Process	Process States
3	Lost Percentage I	Output		Lost(1)/Total Customers(1)	Lost Percentage I			
4	Production Order On I	Time-Persistent		Production Order(1)	Production Order On I			
5	Stock on Hand III	Time-Persistent		Inventory(3)	Stock on Hand III			
6	Stock on Hand II	Time-Persistent		Inventory(2)	Stock on Hand II			
7	Lost Percentage III	Output		Lost(3)/Total Customers(3)	Lost Percentage III			
8	Lost Percentage II	Output		Lost(2)/Total Customers(2)	Lost Percentage II			
9	Production Order On II	Time-Persistent		Production Order(2)	Production Order On II			
10	Production Order On III	Time-Persistent		Production Order(3)	Production Order On II			
11	Amount Lost I	Tally	Amount Lost I					
12	Amount Lost II	Tally	Amount Lost II					
13	Amount Lost III	Tally	Amount Lost III					

Figure 12.38 Dialog spreadsheet of the *Statistic* module for specifying statistics collection.

Time-Persistent, while *Process States* statistics are of type *Frequency*. The *Lost Percentage* statistics are obtained after the simulation stops, and as such are of type *Output*. Finally, the *Amounts Lost* statistics are customer averages, and as such of type *Tally*.

12.2.6 Simulation Results

Figure 12.39 displays the results of a simulation run of length 100,000 hours. An inspection of the *Frequencies* report shows that the production facility is busy about 64% of the time, idle about 14% of the time, and down about 22% of the time.

5:27:12PM **Frequencies** July 28, 2006

Pack Process Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Process States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	19,486	3.2597	63.52	63.52
FAILED	18,560	1.1774	21.85	21.85
IDLE	1,212	12.0712	14.63	14.63

5:28:39PM **User Specified** July 28, 2006

Pack Process Replications: 1

Replication 1 Start Time: 0.00 Stop Time: 100,000.00 Time Units: Hours

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Production Order On I	0.1983	0.009739706	0	1.0000
Production Order On II	0.3763	0.012398047	0	1.0000
Production Order On III	0.6291	0.012227168	0	1.0000
Stock on Hand I	73.6690	0.617746143	0	105.00
Stock on Hand II	138.33	1.50297	0	204.99
Stock on Hand III	178.64	2.96831	0	305.00

Output

Output	Value
Lost Percentage I	0.00096448
Lost Percentage II	0.01017465
Lost Percentage III	0.06420694

Other

None	Average	Half Width	Minimum	Maximum
Amount Lost I	5.3889	(Insufficient)	0.4967	9.8767
Amount Lost II	10.2734	(Insufficient)	0.5134	15.2429
Amount Lost III	19.9212	0.477395206	0.01278375	29.9786

Figure 12.39 Simulation results for the multiproduct production/inventory system.

Note that if the system were to "age" continually (so that failures could occur at any state), then the downtime probability would increase and the idle-time probability would decrease. However, the busy-time probability would not change (within simulation noise), since the work load does not change.

The fraction of time that a product type was pending varies in accordance to the priority structure of product types: about 20% for type 1, 38% for type 2, and 63% for type 3. As expected, the lowest-priority product type received significantly less attention from the production facility.

The statistics of stock on hand by product type reveal that the average inventory levels of all product types are below their reorder points, which indicates that the production facility is having a hard time keeping up with demand. This is largely due to failures as evidenced by the high downtime probability. Further evidence for this phenomenon is furnished by the minimum stock level statistics, all of which hit 0, indicating episodes of stock-out. Stock-outs are quantified by the percentage of loss entries in the *Output* section and the amount lost by product type in the *Other* section of the report.

Similarly to the previous example, system performance may be improved in a number of ways. Again, enhanced maintenance of the production facility would decrease its downtime. Finally, to avoid excessive neglect of orders with low-priority product types, the number of production switches among product types could be restricted.

12.3 EXAMPLE: A MULTIECHELON SUPPLY CHAIN

This section extends the production/inventory system studied in Section 12.1 by adding several echelons to it. Specifically, this example studies a four-echelon supply chain consisting of a supplier, manufacturing plant, distribution center, and retailer. The system uses so-called *installation stock policies* (replenishment policies based only on local information at the supplied installation). Note that such policies utilize only inventory position data (inventory on hand, outstanding orders, and backorders). In contrast, so-called *echelon stock policies* require additional information in the form of inventory positions at the current installation, as well as at all *downstream* echelons.

12.3.1 PROBLEM STATEMENT

Consider a single-product, multiechelon supply chain consisting of a retailer (*R*), distribution center (*DC*), manufacturing plant (*P*), and supplier (*S*). The manufacturing plant interacts with two buffers: an input buffer (*IB*) storing incoming raw material, and an output buffer (*OB*) storing outgoing finished product. The system is depicted schematically in Figure 12.40.

The retailer faces a customer demand stream, and to manage inventory, it uses a continuous-review (Q_R, R_R) control policy, based only on information at the retailer. Recall that under this policy, a replenishment of quantity Q_R is ordered from the distribution center whenever the inventory position (inventory on-hand plus outstanding orders minus backorders) down-crosses level R_R. If the distribution center has sufficient inventory on hand, then the order lead time consists only of transportation delay. Otherwise, it experiences additional delays due to additional transportation delays

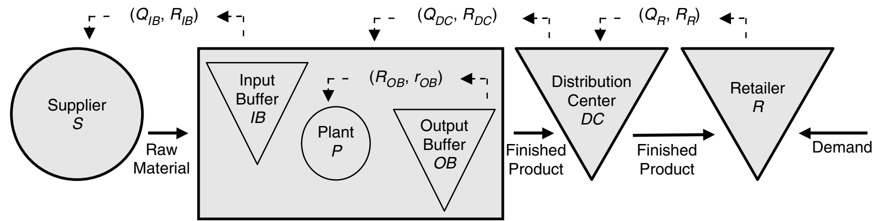


Figure 12.40 Multiechelon supply chain system.

and possible stock-outs upstream of the distribution center. Any excess demand at the retailer that cannot be immediately satisfied from on-hand inventory is lost.

The demand stream at the distribution center consists of orders from the retailer. However, unlike the retailer, unsatisfied demand is backordered. In a similar vein, the distribution center replenishes its stocks from the output buffer (*OB*) of the upstream plant, based on a (Q_{DC}, R_{DC}) continuous-review inventory policy. The unsatisfied portions of orders placed with the plant are backordered.

The plant's manufacturing policy is a continuous-review (R_{OB}, r_{OB}) policy. The plant manufactures one product unit at a time, having consumed one unit of raw material from the input buffer. Note that shortages of raw material in the input buffer (starvation) will cause production stoppages. The input buffer, in turn, orders from an external supplier assumed to have unlimited inventories at all times, so the raw-material lead time is limited to transportation delay, and the plant's orders are always fully satisfied. The corresponding inventory control policy is a continuous-review (Q_{IB}, R_{IB}) policy. Table 12.2 displays the values of the model's inventory control parameters.

The system is subject to the following assumptions:

1. The retailer faces customer demand that arrives according to a Poisson process. The demand quantity of each arrival is one product unit, and all unsatisfied demands are lost.
2. For modeling generality and versatility, we assume that the manufacturing time distribution of a product unit and all transportation time distributions are of the Erlang type (see Section 3.8.7). Specifically, all transportation delays are drawn from the $\mathrm{Erl}(k = 2, \lambda = 1)$ distribution, while the manufacturing time distribution is $\mathrm{Erl}(k = 3, \lambda = 5)$.
3. At all echelons, orders are received in the order they were placed (no overtaking takes place). In fact, an order shipment is launched only after the previous one is received at its destination.
4. When a stock-out occurs in the DC or plant and the unsatisfied portion of the order is backordered, order fulfillment (shipment) is deferred until the full order becomes available. In brief, there is no shipping of partial orders.

Table 12.2

Inventory control parameters

Input Buffer	Output Buffer	Distribution Center	Retailer
$R_{IB} = 10$	$R_{OB} = 30$	$R_{DC} = 10$	$R_R = 5$
$Q_{IB} = 13$	$r_{OB} = 10$	$Q_{DC} = 20$	$Q_R = 10$

We are interested in the following performance metrics:

- The long-run time averages of all inventory levels in the system
- The average number of backorders at the distribution center and the output buffer
- Customer service levels in each echelon

These performance metrics would guide the modeler in making suitable choices for parameters of the inventory control policies.

12.3.2 Arena Model

The system under study is more complex by far than its counterparts in the previous two examples. Accordingly, its Arena model is composed of five segments, each associated with an inventory-holding buffer in a system echelon. Each such buffer is subjected to the following events: order arrival, inventory updating, replenishment order triggering, and order shipment. Additional supply chain activities are modeled at the endpoints of the supply chain, namely, demand arrival on the downstream end, and product manufacturing on the upstream end. Figure 12.41 displays all the variables of the Arena model.

Next, we describe each model segment in some detail, starting with the extreme downstream echelon and moving upstream the supply chain.

12.3.3 Inventory Management Segment for Retailer

Figure 12.42 depicts the retailer inventory management segment of the Arena model. This segment generates the demand stream, handles demand fulfillment, and triggers replenishment orders from the distribution center.

The Poisson stream of customer arrivals with single-unit demand quantities are generated by the *Create* module, called *Customer Demand Arrival At Retailer*. On arrival, a customer entity first enters the *Record* module, called *Tally Retailer Demand*, to tally its demand.

The customer entity then proceeds to test whether the retailer has sufficient inventory on hand by entering the *Decide* module, called *Check Retailer Inventory*, whose dialog box is shown in Figure 12.43. The test has two possible outcomes. First, if the condition *Inventory_Retailer* $\geq$ *1* holds, then the customer entity takes the *True* exit to the *Assign* module, called *Take Away From Retailer Inventory*, where it decrements the on-hand inventory by 1. It then proceeds to another *Assign* module, called *Take Away From Retailer Inventory Position* to decrement by 1 the variable *InventoryPosition_Retailer*. Second, if the condition *Inventory_Retailer* = *0* holds, then the customer entity takes the *False* exit and the current demand is lost. In this case, the customer entity proceeds to the *Record* module, called *Tally Retailer Lost Sales*, to tally the lost demand.

Note that the two *Assign* modules make use of two variables to keep track of inventory information: *Inventory_Retailer*, which tracks the on-hand inventory level, fluctuating between 0 and $R_R + Q_R$, and *InventoryPosition_Retailer*, which tracks the inventory position, fluctuating between R_R and $R_R + Q_R$. The former is used to satisfy a customer's demand, while the latter is used when triggering a replenishment order. Recall that whenever the inventory position at the retailer down-crosses R_R, a

Variable - Basic Process

	Name	Rows	Colum	Clear Option	Initial Values
1	Inventory_Retailer			System	1 rows
2	Q_R			System	1 rows
3	Inventory_DC			System	1 rows
4	R_DC			System	1 rows
5	Order_Output			System	0 rows
6	Inventory_Output			System	1 rows
7	Production_Plant			System	0 rows
8	InventoryPosition_Input			System	1 rows
9	Inventory_Input			System	1 rows
10	InventoryPosition_Retailer			System	1 rows
11	R_R			System	1 rows
12	Order_DC			System	0 rows
13	InventoryPosition_DC			System	1 rows
14	Q_DC			System	1 rows
15	Order_Supplier			System	0 rows
16	R_I			System	1 rows
17	Q_I			System	1 rows
18	Backorder_DC			System	0 rows
19	BigR_Plant			System	1 rows
20	r_Plant			System	1 rows
21	Backorder_Output			System	0 rows
22	AvailableForBackorders_DC			System	0 rows
23	AvailableForBackorders_Output			System	0 rows
24	FullySatDemand_Retailer			System	0 rows
25	FullySatDemand_DC			System	0 rows
26	FullySatDemand_Output			System	0 rows
27	FullySatDemand_Input			System	0 rows

Figure 12.41 Dialog spreadsheet of the *Variable* module for the Arena model of the multiechelon supply chain system.

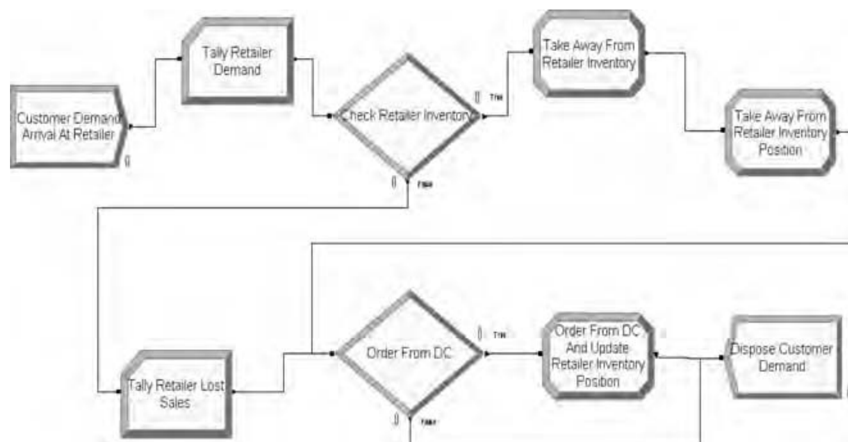


Figure 12.42 Arena model of the inventory management segment for the retailer.

Figure 12.43 Dialog box of the *Decide* module *Check Retailer Inventory.*

replenishment order of quantity Q_R is placed with the distribution center, and the inventory position is immediately updated to $R_R + Q_R$.

Both paths converge at the *Decide* module, called *Order From DC*, where the customer entity checks whether the variable *InventoryPosition_Retailer* has just down-crossed R_R. If it has, then the customer entity proceeds to the *Assign* module, called *Order From DC And Update Retailer Inventory Position*, and performs two assignments. The first assignment sets *Order_DC = 1*, which would promptly release a pending order entity currently detained in the *Hold* module, called *Shall We Release Retailer Order?*, in the distribution-center inventory management segment (see Figure 12.44). The second assignment sets *InventoryPosition_Retailer = InventoryPosition_ Retailer + Q_R* to immediately update the retailer inventory position. Either way, the customer entity proceeds to be disposed of in module *Dispose Customer Demand*.

12.3.4 Inventory Management Segment for Distribution Center

Figure 12.44 depicts the Arena model's inventory management segment for the distribution center. This model segment generates incoming retailer orders, updates distribution-center inventory levels, triggers replenishment orders from the output buffer at the manufacturing plant, sends shipments to the retailer, and updates the retailer inventory level.

The *Create* module, called *Demand Arrival To DC*, creates a single order entity at time 0, which later generates a pending order to be shipped from the distribution center to the retailer. The order entity enters the *Hold* module, called *Shall We Release Retailer Order?*, where it is held until the variable *Order_DC* equals 1, whereupon the order entity proceeds to the *Assign* module, called *Change Flag For Order From Retailer*, and sets *Order_DC* to 0. The order entity then proceeds to the *Separate* module, called *Separate 1*, where it duplicates itself. The order entity itself proceeds to enter the system

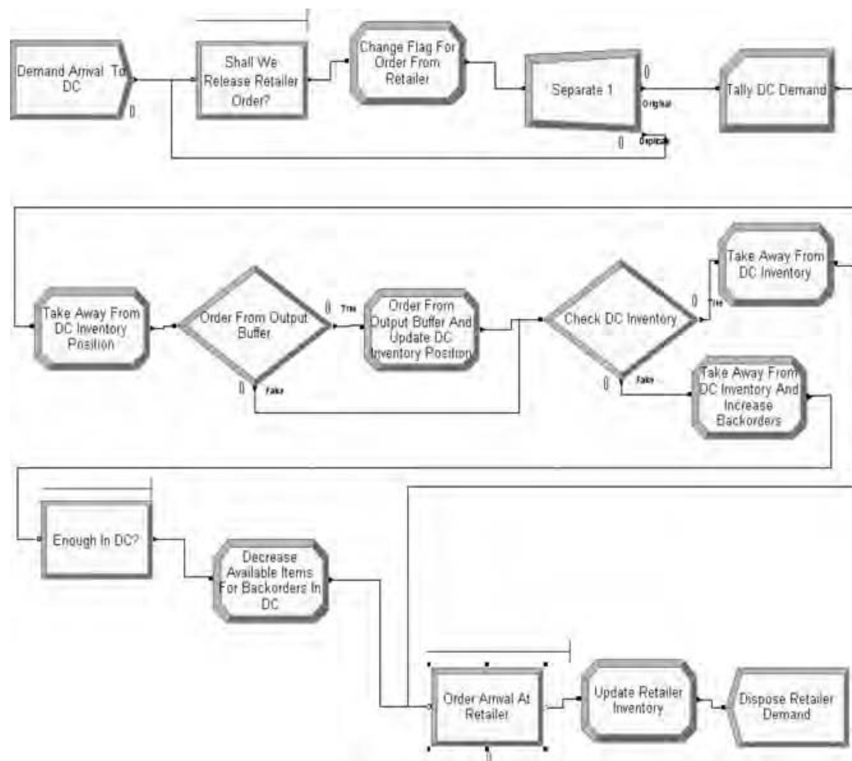


Figure 12.44 Arena model of the inventory management segment for the distribution center.

as a retailer order, while its duplicate loops back to the *Hold* module *Shall We Release Retailer Order?* to generate the next retailer pending order.

The retailer order entity is next tallied in the *Record* module, called *Tally DC Demand*, and then enters the *Assign* module, called *Take Away From DC Inventory Position*, where the variable *InventoryPosition_DC* is decremented by the value of the variable *Q_R*. The order entity then enters the *Decide* module, called *Order From Output Buffer*, to check whether the variable *InventoryPosition_DC* has down-crossed level R_{DC}. If it has, then the order entity proceeds to the *Assign* module, called *Order From Output Buffer And Update DC Inventory Position*, and performs two assignments. The first assignment sets *Order_Output* = *1*; this assignment promptly releases the order entity currently detained in the *Hold* module *Shall We Release DC Order?* in the output-buffer inventory management segment (see Figure 12.47). The second assignment increments the distribution-center inventory position variable *Inventory-Position_DC* by the value of the variable *Q_DC*.

The order entity next enters the *Decide* module, called *Check DC Inventory*, to test whether the retailer has sufficient inventory on hand. The test has two possible outcomes: (1) If the condition *Inventory_DC* ≥ *Q_R* holds, then the order entity takes the *True* exit to the *Assign* module, called *Take Away From DC Inventory*, where it decrements the on-hand inventory by *Q_R*. (2) If the condition *Inventory_DC* < *Q_R* holds, then the order entity takes the *False* exit. In this case, the demand is not fully satisfied, and should be backordered from the manufacturing plant. To this end, the

order entity proceeds to the *Assign* module, called *Take Away From DC Inventory And Increase Backorders*, where three assignments take place:

1. The backorder level is incremented by the shortage.
2. The attribute *UnsatisfiedPortionDemand_DC* of the order entity is assigned the unsatisfied portion of the demand.
3. The inventory level is decremented by 1.

Figure 12.45 depicts these assignments by displaying the dialog box of the *Assign* module *Take Away From DC Inventory And Increase Backorders* (bottom) and its associated *Assignments* dialog boxes (top).

The order entity next proceeds to the *Hold* module, called *Enough In DC?*, where it is detained until sufficient inventory accumulates in the distribution center to satisfy the shortage, as shown in the *Condition* field of the dialog box in Figure 12.46.

The variable *AvailableForBackorders_DC* is used to track the number of inventory product units on hand that are currently available to satisfy backorders during stock-out periods. That is, the order entity is detained in this *Hold* module until sufficient inventory becomes available to satisfy its unsatisfied portion. Note that multiple order entities may be simultaneously detained in this *Hold* module, but those will be satisfied in their order of arrival at this module, since the *Hold* module's queue discipline is FCFS. Now, if the condition *AvailableForBackorders_DC* $\geq$ *UnsatisfiedPortionDemand_DC* holds, then the order entity is released and proceeds to the *Assign* module, called *Decrease Available Items For Backorders In DC*, where the variable *AvailableForBackorders_DC* is decremented by the value of the variable *UnsatisfiedPortionDemand_DC*.

The retailer order entity is now ready for shipment to the retailer. Recall that we assumed that product units are processed sequentially in the transportation system, so as to preclude overtaking. To enforce this rule, we use the *Process* module, called *Order Arrival At Retailer*, to model the transportation delay from the distribution center to the retailer. The order entity next enters the *Assign* module, called *Update Retailer Inventory*, to update the retailer inventory level by incrementing variable *Inventory_Retailer* by *Q_R*. Keep in mind that the inventory position at the retailer was updated when the replenishment order was placed, but the inventory level at the retailer is only updated after the shipment is actually received there. Finally, the order entity proceeds to be disposed of in the *Dispose* module, called *Dispose Retailer Demand*.

12.3.5 Inventory Management Segment for Output Buffer

Figure 12.47 depicts the output-buffer inventory management segment of the Arena model. This model segment generates distribution-center orders, updates the output-buffer inventory level, triggers resumption of suspended manufacturing as necessary, sends shipments to the distribution center, and updates the distribution-center inventory level.

The logic of generating distribution center orders placed at the output buffer is virtually identical to the generation logic used in the previous segment, and therefore will not be described in detail. Recall that a pending order entity (bound for the distribution center) is detained in the *Hold* module, called *Shall We Release DC Order?*.

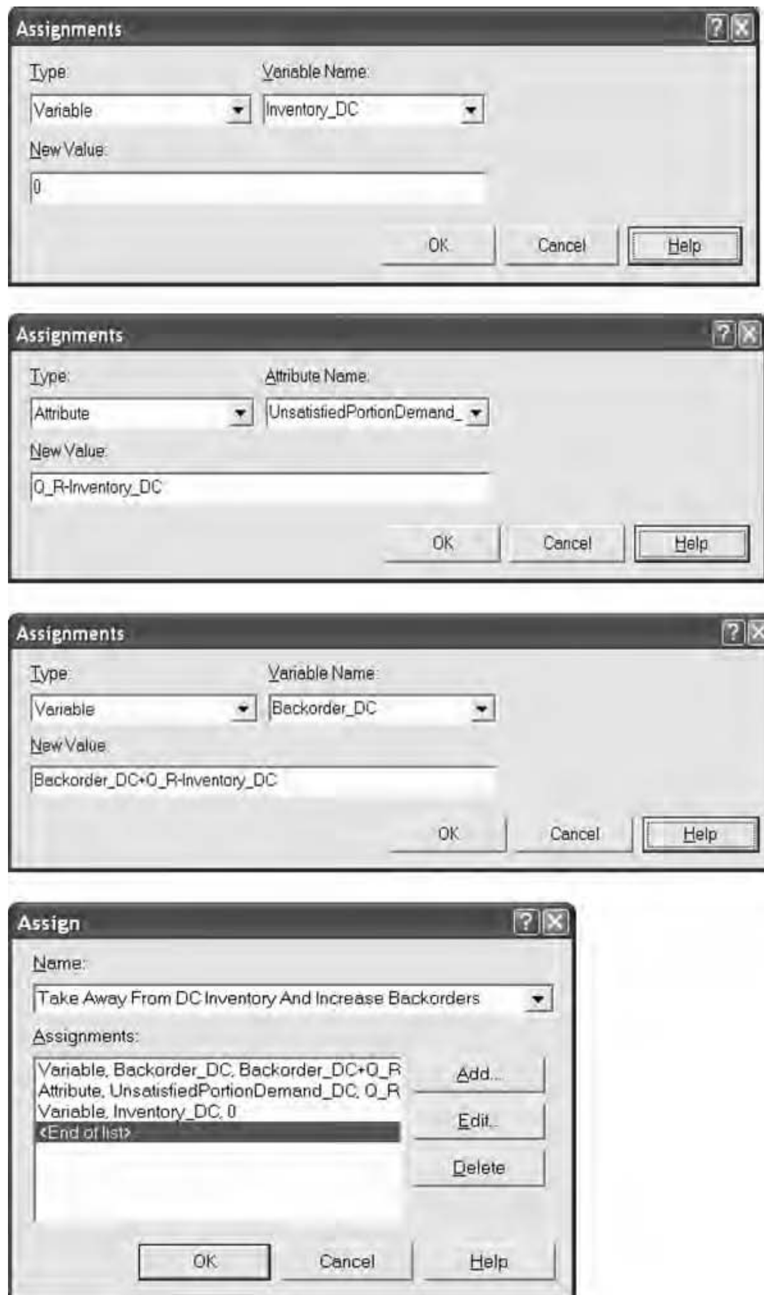


Figure 12.45 Dialog boxes of the *Assign* module *Take Away From DC Inventory And Increase Backorders* (bottom) and its associated *Assignments* dialog boxes (top).

This order entity is released once the variable *Order_Output* is set to 1. Similarly to the logic in the previous segment, the order entity proceeds to the *Separate* module, called *Separate 2*, where it duplicates itself. The order entity itself proceeds to enter the system as a distribution center order, while the duplicate loops back to the *Hold*

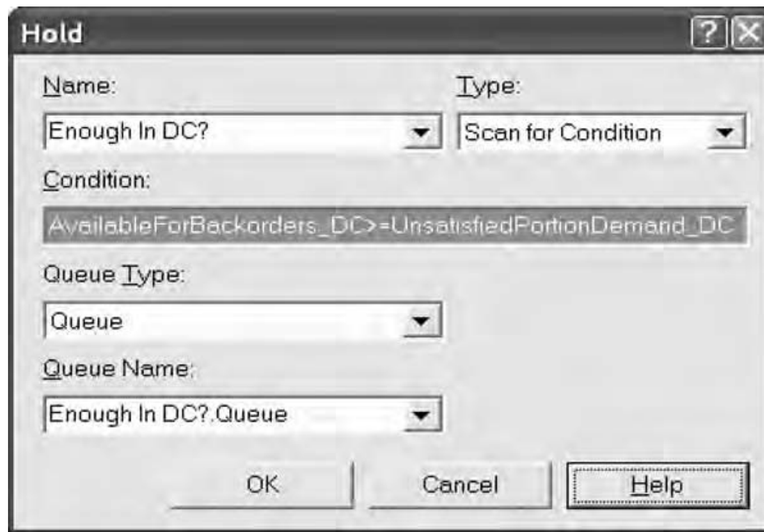


Figure 12.46 Dialog box of the *Hold* module *Enough In DC?*

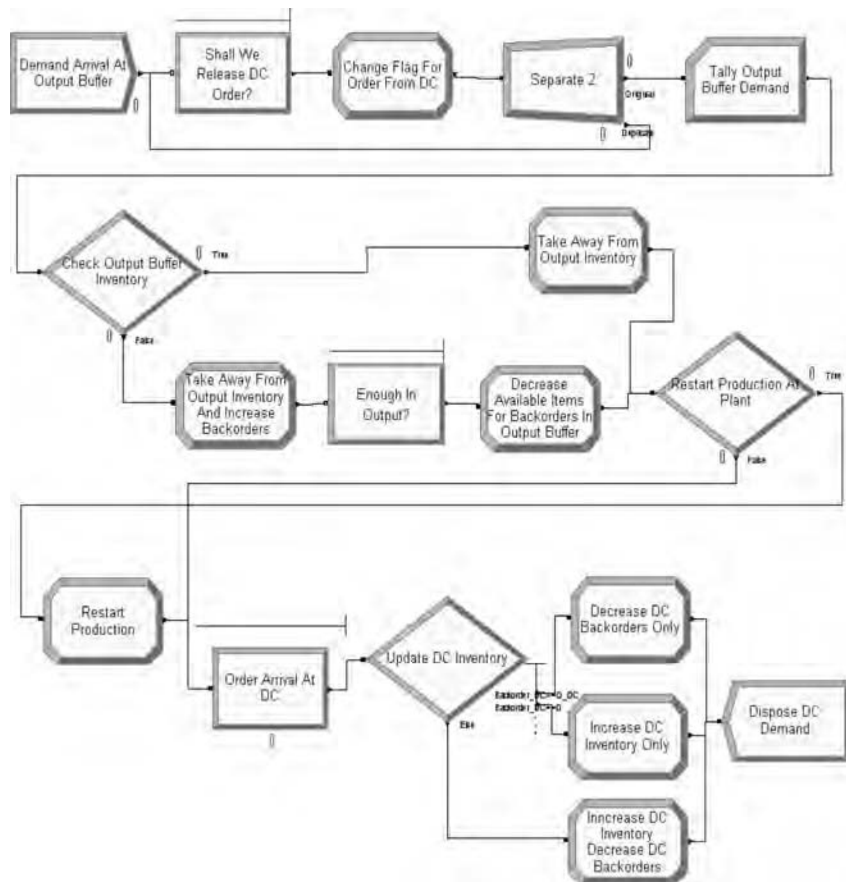


Figure 12.47 Arena model of the inventory management segment for the output buffer.

module, called *Shall We Release DC Order?*, to generate the next pending order of the distribution center.

The order entity next enters the *Record* module, called *Tally Output Buffer Demand*, to tally the order quantity. It then proceeds to the *Decide* module, called *Check Output Buffer Inventory*, to check whether the output buffer has sufficient inventory on hand to satisfy its demand. Again, two outcomes are possible. (1) If the condition *Inventory_Output* ≥ *Q_DC* holds, then the order entity takes the *True* exit to the *Assign* module, called *Take Away from Output Inventory*, where it decrements the on-hand inventory by *Q_DC*. (2) If the condition *Inventory_Output* < *Q_DC* holds, then the order entity takes the *False* exit. In this case, the demand is not fully satisfied and is backordered from the output buffer, and the order entity enters the *Assign* module, called *Take Away From Output Inventory And Increase Backorders*, to perform three assignments:

1. The backorder level variable *Backorder_Output* is incremented by the shortage amount.
2. The *UnsatisfiedPortionDemand_Output* attribute of the order entity is assigned the unsatisfied portion of the demand.
3. The inventory level *Inventory_Output* at the output buffer is set to 0.

The order entity next proceeds to the *Hold* module, called *Enough In Output?*, where it is detained until sufficient inventory accumulates in the output buffer, that is, until the condition

$$\mathit{AvailableForBackorders_Output} \geq \mathit{UnsatisfiedPortionDemand_Output}$$

becomes true. Once this happens, the order entity is released and proceeds to the *Assign* module, called *Decrease Available Items For Backorders In Output Buffer*, where the variable *AvailableForBackorders_Output* is decremented by the shortage portion, *UnsatisfiedPortionDemand_Output*.

The order entity then proceeds to the *Decide* module, called *Restart Production At Plant*, to check whether the variable *Inventory_Output* has just down-crossed the reorder level *r_Plant*. If it has, then the order entity enters the *Assign* module, called *Restart Production*, where it sets *Production_Plant = 1*, which would promptly release the pending production entity currently detained in the *Hold* module, called *Shall We Produce?*, in the input-buffer inventory/production management segment (see Figure 12.48), effectively resuming the production process.

The order entity is now ready for shipment to the distribution center. To this end, it proceeds to the *Process* module, called *Order Arrival At DC*, to model the transportation delay from the output buffer to the distribution center. The order entity next proceeds to the *Decide* module, called *Update DC Inventory*, where three outcomes are possible:

1. If the condition *Backorder_DC* ≥ *Q_DC* holds, then the order entity takes the exit for the *Assign* module, called *Decrease DC Backorders Only*, where it decrements *Backorder_DC* by *Q_DC* and increments *AvailableForBackorders_DC* by *Q_DC*.
2. If the condition *Backorder_DC = 0* holds, then the order entity takes the exit for the *Assign* module, called *Increase DC Inventory Only*, where it increments *Inventory_DC* by *Q_DC*.

3. If the condition $0 < Backorder_DC < Q_DC$ holds, then the order entity takes the exit for the *Assign* module, called *Increase DC Inventory Decrease DC Backorders*, where it sets *Inventory_DC* = *Inventory_DC* + *Q_DC* – *Backorder_DC*, increments *AvailableForBackorders_DC* by *Backorder_DC* by 1, and sets *Backorder_DC* to 0.

Finally, the order entity proceeds to be disposed of in the *Dispose* module, called *Dispose DC Demand*.

12.3.6 Production/Inventory Management Segment for Input Buffer

Figure 12.48 depicts the input-buffer production/inventory management segment of the Arena model. This model segment manages raw-material consumption and finished goods production by keeping track of a circulating control entity that modulates the suspension and resumption of production.

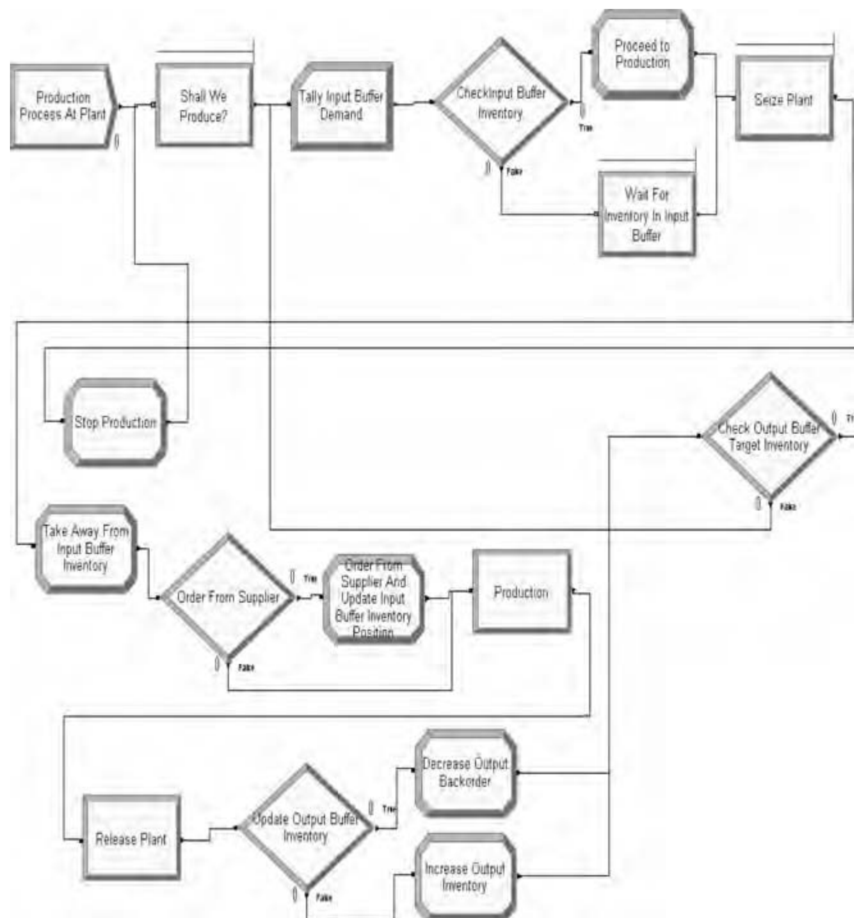


Figure 12.48 Arena model of the production/inventory management segment for the input buffer.

To manufacture a product unit, the plant removes one unit of raw material from the input buffer, processes it, and adds the resulting finished product to the output-buffer inventory and updates its level. If the target level is subsequently reached, then further production is suspended; production is re-started later when the reorder point is down-crossed. Production may also be stopped due to starvation resulting from depletion of raw material in the input buffer, until it is replenished from the supplier.

The *Create* module, called *Production Process At Plant*, generates a single control entity at time 0, which cycles in the segment such that each cycle represents a production cycle. The control entity first enters the *Hold* module, called *Shall We Produce?*, and is detained there until the production is allowed to re-start (recall that this happens when the inventory level in the output buffer down-crosses the reorder level there). Once production is allowed to resume, the control entity enters the *Record* module, called *Tally Input Buffer Demand*, to tally the next product unit.

The control entity then proceeds to the *Decide* module, called *Check Input Buffer Inventory*, to check if there is raw material in the input buffer. If true, it proceeds to the *Assign* module, called *Proceed to Production*, to update the number of satisfied demands. If false, starvation is in effect and the control entity takes the *False* exit to the *Hold* module, called *Wait For Inventory In Input Buffer*, until the condition *Inventory_Input* $\geq$ *1* becomes true. Once this condition holds, the control entity proceeds to the *Seize* module, called *Seize Plant*, where it immediately seizes the *Plant* resource. To model the consumption of one unit of raw material, the control entity enters the *Assign* module, called *Take Away From Input Buffer Inventory*, and decrements by 1 both the inventory on-hand variable *Inventory_Input* and the inventory position variable *InventoryPosition_ Input*.

Similarly to the previous segments, the control entity next proceeds to the *Decide* module, called *Order From Supplier*, to check whether the reorder point at the input buffer has been down-crossed. If it has, a raw material replenishment is promptly triggered by releasing the order entity currently detained in the *Hold* module, called *Shall We Release Plant Order?*, in the supplier inventory management segment (see Figure 12.49). The control entity next enters the *Assign* module, called *Order From Supplier And Update Input Buffer Inventory Position*, where it updates the input buffer

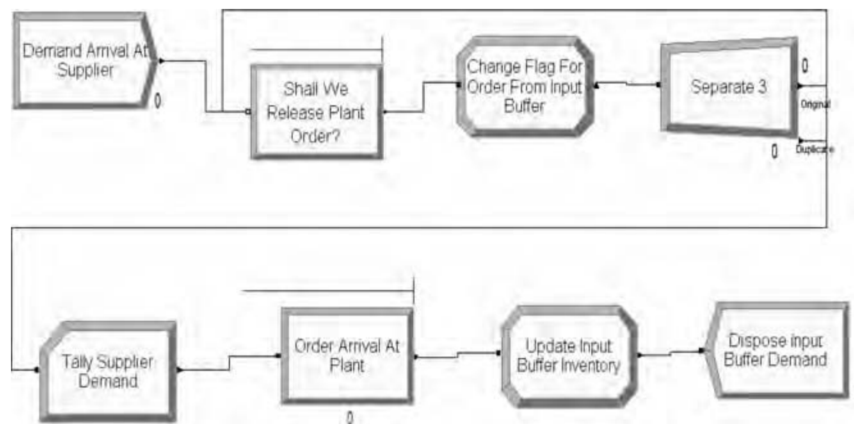


Figure 12.49 Arena model of the production/inventory management segment for the supplier.

inventory position (variable *InventoryPosition_Input*) and triggers a supplier order by setting the variable *Order_Supplier* to 1 of the variable *Q_Input*.

In all cases, the control entity eventually enters the *Delay* module, called *Production*, to model the manufacturing time delay of one product unit, following which it proceeds to the *Release* module, called *Release Plant*, where it releases the *Plant* resource. Before adding the finished product unit to inventory, the control entity enters the *Decide* module, called *Update Output Buffer Inventory*, to check if there are any pending backorders in the output buffer (i.e., whether the condition $Backorder_Output \geq 1$ holds). If there are pending backorders, the control entity takes the *True* exit for the *Assign* module, called *Decrease Output Backorder*, where a pending backorder is satisfied by decrementing the variable *Backorder_Output* by 1 and incrementing the variable *AvailableForBackorders_Output* by 1. Otherwise, if no backorders are pending, the control entity takes the *False* exit for the *Assign* module, called *Increase Output Inventory*, where the output buffer inventory level is updated by incrementing the variable *Inventory_Output* by 1.

The control entity then proceeds to the *Decide* module, called *Check Output Buffer Target Inventory*, to check whether the inventory level at the output buffer has reached its target level. If it has, then the control entity takes the *True* exit for the *Assign* module, called *Stop Production*, and sets $Production_Plant = 0$ to suspend production, after which it cycles back to the *Hold* module, called *Shall We Produce?*, to wait until the next production cycle is resumed. Otherwise, the control entity takes the *False* exit and cycles back to the *Record* module, called *Tally Input Buffer Demand*, to start the next production cycle.

12.3.7 Inventory Management Segment for Supplier

Figure 12.49 depicts the supplier inventory management segment of the Arena model. This model segment generates input buffer orders, sends shipments from the supplier to the input buffer, and updates the input buffer inventory level.

The logic of generating input buffer orders to the supplier is virtually identical to the generation logic used in the previous segments, and therefore will not be repeated. Note, however, that this model segment is a bit simpler than its counterparts; because the supplier always has sufficient inventory on hand, replenishment delays reduce to transportation delays.

12.3.8 Statistics Collection

Figure 12.50 displays the spreadsheet view of the *Statistics* module for the multi-echelon supply chain model. The spreadsheet includes *Time-Persistent* statistics of on-hand inventory levels at the retailer, distribution center, output buffer, and input buffer, as well as *Time-Persistent* statistics of backorder levels at the distribution center and output buffer. It also includes the *Time-Persistent* statistic of plant utilization, that is, the percentage of time the plant is busy producing. Finally, the *Output* statistics estimate customer service levels at each echelon in terms of the fill rate, namely, the probability (fraction) of orders that were satisfied from on-hand inventory, without experiencing backordering.

	Name	Type	Expression	Report Label	Output File
1	Retailer Average Inventory	Time-Persistent	Inventory_Retailer	Retailer Average Inventory	
2	DC Average Inventory	Time-Persistent	Inventory_DC	DC Average Inventory	
3	Output Average Inventory	Time-Persistent	Inventory_Output	Output Buffer Average Inventory	
4	Input Average Inventory	Time-Persistent	Inventory_Input	Input Buffer Average Inventory	
5	DC Average Backorder	Time-Persistent	Backorder_DC	DC Average Backorder	
6	Output Average Backorder	Time-Persistent	Backorder_Output	Output Buffer Average Backorder	
7	Input Buffer Customer Service Level	Output	FullySatDemand_Input/NC(Tally Input Buffer Demand)	Input Buffer Customer Service Level	
8	Output Buffer Customer Service Level	Output	FullySatDemand_Output/(NC(Tally Output Buffer Demand)/Q_DC)	Output Buffer Customer Service Level	
9	Retailer Customer Service Level	Output	FullySatDemand_Retailer/NC(Tally Retailer Demand)	Retailer Customer Service Level	
10	DC Customer Service Level	Output	FullySatDemand_DC/(NC(Tally DC Demand)/Q_R)	DC Customer Service Level	
11	Plant Utilization	Time-Persistent	Production_Plant==1	Plant Utilization	

Figure 12.50 Dialog spreadsheet of the *Statistic* module for the multiechelon supply chain system.

12.3.9 Simulation Results

The simulation study explored six supply-chain scenarios with varying demand rates. To attain reliable statistical outputs, each simulation run consisted of a single replication simulating 50,000,000 product-unit departures.

Simulation outputs for the six parameter settings of the demand arrival rate, λ, are displayed in Table 12.3. The computed statistics are average inventory levels, average backorder levels, and customer fill rates (table columns) at four echelons: input buffer, output buffer, distribution center, and retailer (table rows). The notation N/A stands for "not applicable."

An examination of Table 12.3 reveals that for a parameter setting of $\lambda = 1.0$, the average inventory level in the input buffer is 15.0203 and its fill rate is 99.88%. Furthermore, the average inventory level at the retailer is 8.5209 and its fill rate is 98.61%. This means that 1.39% of the demand at the retailer is lost. Varying the demand-arrival rate parameter for each given inventory shows an expected pattern: a parameter increase results in concomitant decreases of the corresponding average inventory levels and fill rates, and concomitant increases in the corresponding average backorder levels.

Note that as the system load increases, the customer service level deteriorates the most at the output buffer of the manufacturing plant. This is due to the fact that the plant has a fixed production rate that must cope with an increasing demand rate. Thus, the highest supply chain echelon (excluding its supplier) is most affected by the increasing load. This phenomenon is consistent with the bullwhip effect explained in the beginning of this chapter.

EXERCISES

1. *Inventory management*. A computer store sells and maintains an inventory of fax machines (units). On average, one unit is sold per day, but the store experiences a burst of customers on some days. A marketing survey suggests that customer interarrival times are iid exponentially distributed with rate 1 per day.

Table 12.3

Simulation outputs for a replication of multiechelon supply chain

	$\lambda = 1.0$			$\lambda = 1.1$		
	Inventory Level	Backorder Level	Fill Rate (%)	Inventory Level	Backorder Level	Fill Rate (%)
Input Buffer	15.0203	N/A	99.88	14.8334	N/A	99.87
Output Buffer	23.6452	0.0003	99.85	22.9043	0.0013	99.58
DC	23.0257	0.0000	100.00	22.8343	0.0000	100.00
Retailer	8.5209	N/A	98.61	8.3328	N/A	98.14
	$\lambda = 1.2$			$\lambda = 1.3$		
	Inventory Level	Backorder Level	Fill Rate (%)	Inventory Level	Backorder Level	Fill Rate (%)
Input Buffer	14.6480	N/A	99.86	14.4680	N/A	99.85
Output Buffer	22.0530	0.0052	98.88	21.0038	0.0199	97.23
DC	22.6296	0.0000	100.00	22.3842	0.0005	99.98
Retailer	8.1465	N/A	97.58	7.9646	N/A	96.97
	$\lambda = 1.4$			$\lambda = 1.5$		
	Inventory Level	Backorder Level	Fill Rate (%)	Inventory Level	Backorder Level	Fill Rate (%)
Input Buffer	14.2924	N/A	99.84	14.1222	N/A	99.82
Output Buffer	19.6205	0.0731	93.57	17.6510	0.2507	86.11
DC	22.0235	0.0031	99.90	21.3356	0.0169	99.56
Retailer	7.7828	N/A	96.26	7.5964	N/A	95.44

When the inventory on hand drops down to 5 units, an order of 20 units is placed with the supplier, and the lead time is uniformly distributed between 5 and 10 days. Unsatisfied customers check back with the store and wait for the order to arrive (backordering case). Keep in mind that another order is placed immediately upon order backorder arrival, should the inventory level fall below the reorder level after satisfying all backorders.

a. Develop an Arena model of the computer store, and simulate it for 2 years of daily (9:00 A.M. to 5:00 P.M.) operation.
b. What is the average stock on hand?
c. What is the average backorder level?
d. What percentage of the time does the store run out of stock? Note that this is the probability that a customer's order is not satisfied and is subsequently backordered.
e. What is the percentage of time that the stock on hand is above the reorder level?
f. Modify the customer arrival rate to 0.05 customers per hour during the first 4-hour period, and 0.25 customers per hour during the second 4-hour period of any day. Repeat items a through e for the modified system. Compare the performance measures of items a through e in the original and modified systems.

2. *Procurement auction*. Procurement auctions (also known as *reverse auctions*) constitute a common purchasing method designed to lower purchasing costs by inducing competition among suppliers. In this type of auction, there is a single buyer of goods, and multiple potential suppliers bidding to sell the requisite goods; winner selection is usually based on the lowest price. The procurer sends a "request for bid" to all available suppliers, and the suppliers respond immediately with bids consisting of two attributes, price and lead time. The lowest bid winner ships the ordered quantity, which reaches the procurer after the stated lead time. Consider a procuring company that operates as follows:
   - There are three independent suppliers, each of which generates iid prices from the Unif(10, 20) distribution, and iid lead times (in hours) from the Tria(2, 4, 5) distribution.
   - The company follows a (Q, R) inventory control policy, with order quantity $Q = 80$ and reorder point $R = 20$, where all shortages are backordered. An auction is initiated whenever the reorder point is reached.
   - Customer interarrival times (in hours) follow the exponential distribution with rate $\lambda = 1$.

   Demand quantities of arriving customers are iid and drawn from the discrete distribution

$$\Pr(D = d) = \begin{cases} 0.3, & d = 1 \\ 0.2, & d = 2 \\ 0.3, & d = 3 \\ 0.2, & d = 4. \end{cases}$$

   - The initial on-hand inventory is 50 units.

   Develop an Arena model of the company operations, and simulate it for 100,000 hours. Compute the following performance metrics of the system:

   a. Average price of winning bids
   b. Average lead time of winning bids
   c. Average on-hand inventory
3. *Multiple products with rapid switchover*. Consider a production/inventory system similar to the one described in Section 12.2. The production facility produces product types 1, 2, and 3, and these are moved to a warehouse to supply three distinct incoming customer streams, denoted by types 1, 2, and 3, respectively. A raw-material storage feeds the production process.

   The production facility produces products in single units, switching from the production of one product type to another, depending on inventory levels. However, product types have priorities in production, with product 1 having the highest priority and product 3 the lowest. Lower-priority products are subject to preemption by high-priority ones. More specifically, if the inventory level of a higher-priority product down-crosses its reorder level while the production of a lower-priority product is in progress, then the production process switches to the higher-priority product as soon as the current lower-priority product unit is completed. This policy provides a faster response to higher-priority products at the expense of more frequent setups. Clearly, this is a desirable policy in scenarios with low setup costs. However, in this problem setup times are assumed to be 0.

   Each product has its own parameters, as shown in the following table.

Product Type	Reorder Point r	Target Level R	Initial Inventory	Processing Times (minutes)	Demand Arrivals (hours)	Demand Quantity
1	10	30	5	15	Exp(2)	Unif(2, 8)
2	25	50	40	10	Exp(4)	Unif(6, 10)
3	50	80	30	5	Exp(6)	Unif(5, 10)

The system is subject to the following assumptions and operating rules:

- There is always sufficient raw material in storage, so the production process never starves.
- Lot processing time is deterministic as shown in the table above.
- The warehouse implements the (R, r) inventory control policy for each product (see Section 12.2).
- On customer arrival, the inventory is immediately checked. If there is sufficient stock on hand, the incoming demand is promptly satisfied. Otherwise, the unsatisfied portion of the demand is backordered.

Develop an Arena model for the system and simulate it for 1 year (24-hour operation) and estimate the following statistics:

a. Probability of an outstanding order, by product type
b. Production-facility downtime probability
c. Average inventory level, by product type
d. Percentage of customers whose demand is not completely satisfied, by product type
e. Average backorder level, by product type
f. Average backordered quantity per backorder, by product type

4. *Multiple products with finite raw material*. Consider the previous problem with the following extension. The system now has separate raw-material storage areas for each product type at the production facility. The production process uses one unit of raw material to process one unit of finished product. Raw-material storage is maintained using the (Q, R) policy with lead times. Ordering and lead-time data are given in the table below.

Product Type	Reorder Point	Order Quantity	Initial Inventory	Lead Time (days)
1	20	10	25	Unif(1, 2)
2	30	20	40	Unif(1, 3)
3	40	30	50	Unif(0.5, 4)

If the facility runs out of raw material while producing a product unit, it switches to the highest-priority product that needs attention and has raw material in its storage.

Develop an Arena model for the system and simulate it for 1 year (24-hour operation). Estimate the following statistics in addition to those in the previous problem:

a. Average raw material inventory levels, by product type
b. Average length of stock-out periods, by product type
c. Probability of stock-outs, by product type

5. *Optimal inventory policy*. Home Needs (HN) sells garden sprinklers through two stores. It supplies both stores from a single distribution center (DC), and the DC is supplied in turn from a single factory. Both stores operate around the clock

(24-hour business days), and the unsatisfied portion of customers demand at the stores is lost rather than backordered. All lead times are deterministic.

HN is interested in implementing order-up-to-*R* inventory control policies at its stores and DC. To this end, HN needs to forecast demand quantities arriving over a lead-time period (*lead-time demand*, for short), and excess inventory quantities needed to guard against excessive future stock-outs (*safety stock*). Let the lead-time demand forecast at the end of day *n* be given by

$$LTD_n = D_n \times LT,$$

where D_n is the daily demand observed on day *n*, and *LT* is a given deterministic lead time. Let the safety-stock forecast at the end of business day *n* be given by

$$SS_n = SSF \times LTD_n,$$

where *SSF* is a prescribed safety stock factor. Finally, let IP_n be the inventory position at the end of business day *n*.

At the end of each 24-hour business day, a review is performed at both stores and the DC, and an order is placed for each store, if necessary. The decision to place an order is based on the following rule at the end of each business day *n*:

- Do not order if $IP_n - LTD_n > SS_n$.
- Otherwise, place an order of quantity $Q_n = SS_n - IP_n + LTD_n$.

The system is subject to the following assumptions:

- HN observed that total daily customer demand at the first store is uniform between 5 and 30 sprinklers, and at the second store it is uniform between 10 and 40 sprinklers. Daily demand becomes known at the 8th hour into the business day, (recall that a business day is 24 hours long).
- The unsatisfied portion of the demand is lost.
- Except for the first day, if the daily demand at the DC is 0, then the *LTD* and *SS* forecasts carry over from the previous day.
- The factory operates as a single-stage process, and takes 2 minutes to produce each product unit.
- Orders from both stores take 1 hour to prepare and reach the DC.
- Lead times for orders are provided in the following table:

Origination/Destination	Lead Time (days)
DC to store 1	2
DC to store 2	2
Factory to DC	3 + manufacturing lead time

- The initial on-hand inventory at the stores and DC and the corresponding *SSF* parameters are given in the following table:

Location	Initial Inventory	*SSF*
Store 1	35	1
Store 2	50	1
DC	255	0.5

Develop an Arena model for the system, and estimate the minimal value of R that results in a 90% fill rate based on 20-year replications. Estimate the following statistics:

a. Percentages of fully satisfied and partially satisfied customers at each store and orders at the DC
b. Average inventory levels at each store and at the DC
c. Average lost portion of sales at each store

This page intentionally left blank

Chapter 13

Modeling Transportation Systems

Transportation systems affect many facets of daily life. A significant portion of a typical workday is spent traveling, including riding buses, trains, or airplanes, or more commonly, driving a car. Traffic congestion on terrestrial roads and flight lanes is now a familiar irritant, causing varying delays, pollution, and the occasional "road rage." While significant delays are typically encountered in metropolitan area highways and airports, it should be noted that maritime transportation has its own traffic patterns and delays due to weather conditions and tidal activity as well as equipment failures.

Monte Carlo simulation is an invaluable tool for studying transportation systems and solving their attendant problems (e.g., ATZ Consultants [1996], Soros Associates [1997]). Some common examples are listed below:

- Designing new traffic routes and alternate routes to satisfy demand for additional road capacity, or eliminating bottlenecks and congestion points in existing routes by appropriate placement of traffic lights and tollbooths.
- Designing traffic patterns on the factory floor, including transporters and conveyors, for efficient movement of raw material and product.
- Designing port facilities, such as berths and piers, and allocating vessels to berths. Such designs include material handling systems (loaders/unloaders, transporters, conveyors, and others) for containers and bulk-material transport.
- Designing new airports or adding runways to existing ones to satisfy demand for additional flight capacity. Such designs include air traffic patterns and routing, runway scheduling, and planning cargo operations.

Computer systems and telecommunication networks also have a transport component (information transfer and message transmission), but these will be treated separately in Chapter 14.

Generally, strategic decisions concerning transportation systems often involve identifying system capacities and eliminating bottlenecks so as to increase system throughput. The associated operations are extremely expensive, and therefore need to be studied carefully before any action is taken. Fortunately, simulation studies can be helpful and

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

even instrumental in identifying correct designs for large-scale transportation projects. Transportation-related simulation models employ entities to model the moving objects, such as cars, trucks, trains, planes, ships, tugboats, humans, and so on. Contention for transportation-oriented service by entities is modeled through resources, such as road pavement, traffic lights, tollbooths, airport taxiways and runways, and berths and piers, as well as material handling equipment. To make modeling more realistic, resources may experience stoppages for various reasons (failures, weather, strikes, etc.), which can further exacerbate delays in the system.

This chapter provides an overview of Arena facilities for modeling transportation systems. Their use is illustrated in a number of transportation-related examples.

13.1 *ADVANCED TRANSFER* TEMPLATE PANEL

Thus far, entity transfers were implemented via Arena connections (for instantaneous transfer) and *Delay* modules (for time-lapse transfer). The *Advanced Transfer* template panel provides additional mechanisms of time-lapse transfer of entities among sets of modules or geographic locations. This section briefly reviews the facilities provided by this template panel.

The *Advanced Transfer* template panel implements a worldview in which entities are transported among *Station* modules. The simplest transfer mechanism uses *Route* modules as dispatch points and *Station* modules as destination points. Additionally, the *Enter* and *Leave* modules may be used to transfer entities into and out of physical or logical locations. The *PickStation* module allows entities to select a destination *Station* module using a selection criterion, such as the minimal or maximal queue size, number of busy resource units, or an arbitrary expression. Alternatively, an entity can be endowed with an itinerary using the *Sequence* module to specify a sequence of *Station* modules (referred to as *Step* objects). The *Sequence* module also supports assignment of variables or attributes at each *Step* object.

Station and *Route* modules are the most fundamental components of the *Advanced Transfer* template panel. A *Station* module is used to designate a physical location in the model (e.g., a milling workstation that physically houses milling machines). It may also be used to designate a logical location in the model comprised of a set of modules that perform some logical function (e.g., inventory control logic). Entities are transferred among *Station* modules using *Route* modules, where the modeler specifies a *Station* module destination and duration for entity transfers. The destination *Station* module may be specified in an attribute, as an expression, as a *Station* module name, or as part of an itinerary defined in a *Sequence* module (see Section 13.5). Note that *Route* modules are not graphically connected to destination *Station* modules. Consequently, these modules arrange for entity transfer without animation.

The *Advanced Transfer* template panel also provides specialized transportation facilities for material handling in manufacturing-related systems. The *Transporter* module is used to model vehicles, such as trucks, forklifts, container carriers, and similar transport vehicles that move material in discrete parcels. These are collectively referred to as *transporters*. The *Conveyor* module is used to model continuous-mode conveyance facilities, such as conveyor belts. These are collectively referred to as *conveyors*.

In order to implement transporters or conveyors, the modeler specifies a transportation network consisting of locations (*Station* modules) and topology (distances among

locations). Network topology is specified in *Distance* modules (for transporters) and *Segment* modules (for conveyors). Thus, once the velocities of transporters and conveyors are known, Arena will automatically compute the corresponding travel times. When an entity gets hold of a transporter, the entity enters a *Transport* module for the duration of the transportation activity. In fact, the two (entity and transporter) move together as a group. When the group arrives at its destination *Station* module, the entity frees the transporter (possibly after a delay for unloading) and exits the *Transport* module. The transporter will stay at that *Station* module until requested again. Similarly, when an entity accesses a conveyor, it enters a *Convey* module for the duration of the conveyance activity, during which it occupies a number of cells on the conveyor. On arrival at its destination *Station* module, it exits the conveyor and releases the conveyor cells that it had previously occupied, and then exits the *Convey* module.

A number of modules regulate the operation of transporters and conveyors. The *Free* module is used to release a transporter engaged by an entity, while the *Exit* module is used to release a conveyor in a similar way. The *Activate* and *Halt* modules are used to start and stop transporters (e.g., to model scheduled and unscheduled maintenance), while the *Stop* and *Start* modules provide the analogous functionality for conveyors. The *Move* module is used to advance a transporter among stations. The assignment of various transportation facilities to requesting entities is handled by a separate set of modules. The *Allocate* and *Request* modules assign a transporter to an entity, with or without moving it, respectively. Similarly, the *Access* module is used to allocate conveyor cells to an entity.

In this chapter, we illustrate the use of most transporter-related modules in a number of examples. Because conveyor implementation is similar to that of transporters, we will not discuss such models here, but encourage the reader to experiment with them.

13.2 *ANIMATE TRANSFER* TOOLBAR

The *Animate Transfer* toolbar, shown in Figure 13.1, supports visualization and animation of various transportation devices, such as transporters and conveyors. This toolbar consists of 11 buttons as follows (from left to right):

- The *Storage* button is used to animate the contents of a storage facility similarly to the action of the *Queue* button in the *Animate* toolbar. Internally, the Arena modules *Store* and *Unstore* (from the *Advanced Process* template panel) provide the functionality of entities entering and departing a storage facility, and may be used to keep track of entities in a portion of the model. The number of entities in storage is accessible via the SIMAN variable *NSTO(storage_name)*, which may be inspected via the *Variable*

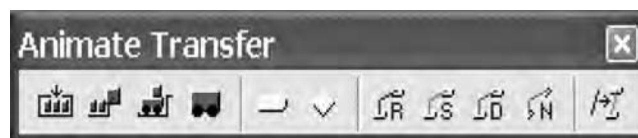


Figure 13.1 The Arena *Animate Transfer* toolbar.

button of the *Animate* toolbar. A graphical *Storage* T-bar graphically depicts the entities in a storage facility.

- The *Seize* button allows the modeler to define a so-called *seize area* to animate entities seizing a resource.
- The *Parking* button allows the modeler to define a so-called *parking area* to animate parking areas for transporters.
- The *Transporter* button allows the modeler to design a visual representation (picture) for a transporter.
- The *Station* button allows the modeler to specify an icon for a particular *Station* module.
- The *Intersection* button allows the modeler to specify an intersection in a network of *automated guided vehicles* (*AGV*s). These are transporter-type objects that must keep track of their positions in the system to avoid collisions. AGVs are not covered in this book.
- The *Route* button is used to specify the animation path for moving entities in the system.
- The *Segment* button is used to specify the animation path of a conveyor.
- The *Distance* button is used to specify the animation path of a transporter.
- The *Network* button is used to specify the animation path of an AGV. Unlike ordinary transporters, Arena endows AGVs with the capability of sensing each other to avoid collisions.
- The *Promote Path* button is used to promote a visual line to an animation path of a desired object.

The use of the *Animate Transfer* toolbar and various transportation-related modules will be illustrated in three examples in the sequel.

13.3 EXAMPLE: A BULK-MATERIAL PORT

Bulk materials are an important component of international trade, and their transportation is mediated by numerous seaports worldwide. Important bulk materials include iron ore, cement, bauxite, grain, oil, and coal. For analysis of port facilities, see Altiok (1998), White (1984), and Crook (1980).

This example illustrates bulk port operations, using the notions of station, entity routing among stations, entity pick-up and drop-off by another entity, and the control of entity movements using logical gating. It concerns a bulk material port, called Port Tamsar, at which cargo ships arrive and wait to be loaded with coal for their return journey. Cargo ship movement in port is governed by tug boats, which need to be assigned as a requisite resource. The port has a single berth where the vessels dock, and a single ship loader that loads the ships. A schematic representation of the layout of Port Tamsar is depicted in Figure 13.2.

Port Tamsar operates continually 24 hours a day and 365 days a year. The annual coal production plan calls for nominal deterministic ship arrivals at the rate of one ship every 28 hours. However, ships usually do not arrive on time due to weather conditions, rough seas, or other reasons, and consequently, each ship is given a 5-day grace period commonly referred to as the *lay period* (see Jagerman and Altiok [1999]). We assume that ships arrive uniformly in their lay periods and queue up FIFO (if necessary) at

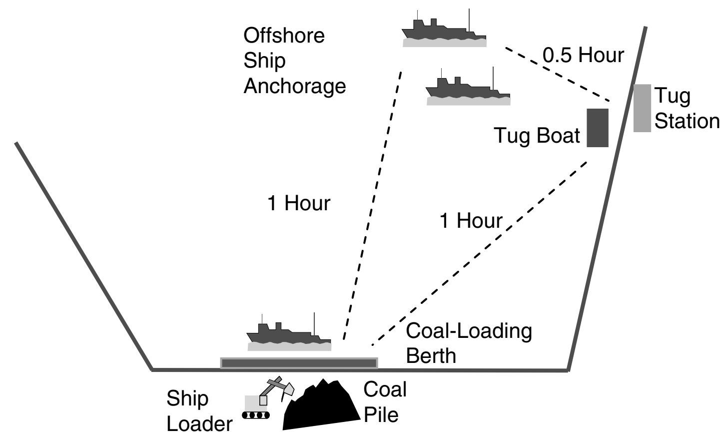


Figure 13.2 Layout of Port Tamsar.

an offshore anchorage location, whence they are towed into port by a single tug boat as soon as the berth becomes available. The tug boat is stationed at a tug station located at a distance of 30 minutes away from the offshore anchorage. Travel between the offshore anchorage and the berth takes exactly 1 hour. We assume that there is an uninterrupted coal supply to the ship loader at the coal-loading berth, and that ship loading times are uniformly distributed between 14 and 18 hours. Once a ship is loaded at the berth, the tug boat tows it away to the offshore anchorage, whence the boat departs with its coal for its destination. Departing vessels are accorded higher priority in seizing the tug boat.

An important environmental factor in many port locations around the world is tidal dynamics. Cargo ships are usually quite large and need deep waters to get into and out of port. Obviously, water depth increases with high tide and decreases with low tide, where the time between two consecutive high tides is precisely 12 hours. We assume that ships can go in and come out of port only during the middle 4 hours of high tide. Thus, the tidal window at the port is closed for 8 hours and open for 4 hours every 12 hours.

We wish to simulate Port Tamsar for 1 year (8760 hours) to estimate berth and ship loader utilization, as well as the expected port time per ship. We mention parenthetically that although a number of operating details have been omitted to simplify the modeling problem, the foregoing description is quite realistic and applicable to many bulk material ports and container ports around the world.

An Arena model of Port Tamsar consists of four main segments: ship arrivals, tugboat operations, coal-loading operations at the berth, and tidal window modulation. These will be described next in some detail along with simulation results.

13.3.1 Ship Arrivals

Ship arrivals are implemented in the Arena model segment depicted in Figure 13.3. Ship arrivals are generated deterministically by the *Create* module, called *Vessel Arrivals*, at the rate of one ship every 28 hours. On creation, a ship entity immediately proceeds to the *Delay* module, called *Lay Period*, where it is delayed uniformly between

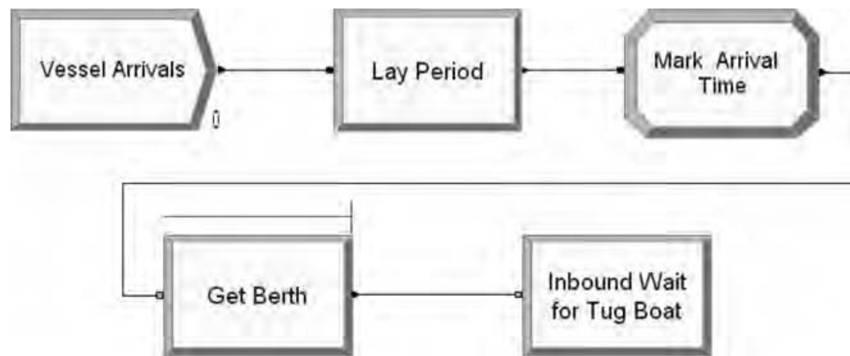


Figure 13.3 Arena model segment implementing ship arrivals at Port Tamsar.

0 and 120 hours to model an actual arrival within its lay period. The dialog boxes of modules *Vessel Arrivals* and *Lay Period* are displayed in Figure 13.4.

In due time, the ship entity enters the *Assign* module, called *Mark Arrival Time*, where its (actual) arrival time is stored in its *ArrTime* attribute. The ship entity then enters the *Seize* module, called *Get Berth*, whose dialog box is displayed in Figure 13.5.

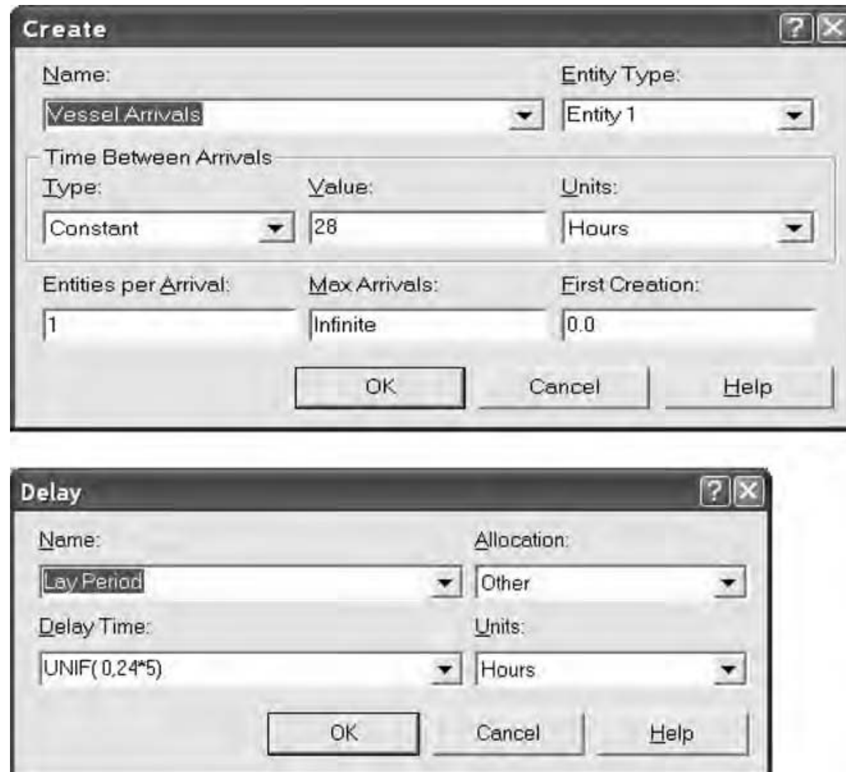


Figure 13.4 Dialog boxes of the *Create* module *Vessel Arrivals* and the *Delay* module *Lay Period*.

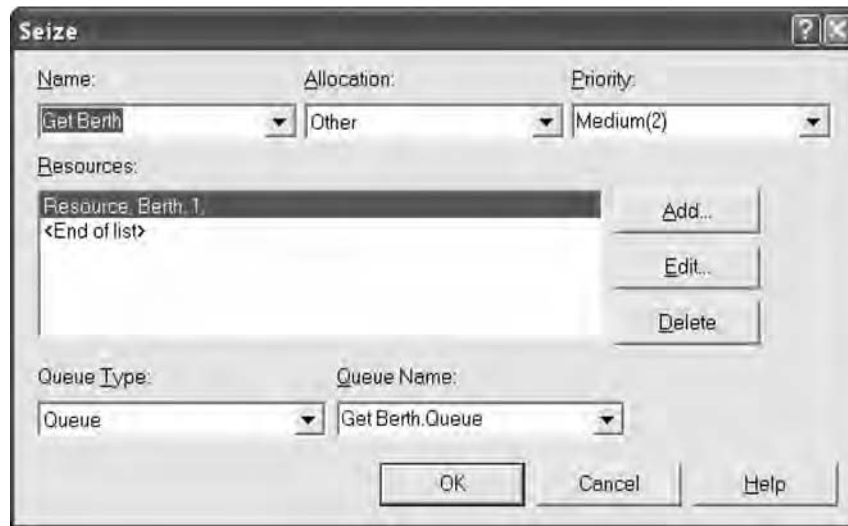


Figure 13.5 Dialog box of the *Seize* module *Get Berth*.

Here, the ship entity attempts to seize the berth, and waits in queue *Get Berth.Queue* while resource *Berth* is occupied.

Once a ship succeeds in seizing resource *Berth*, it enters the *Hold* module, called *Inbound Wait for Tug Boat*, whose dialog box is displayed in Figure 13.6. More specifically, the ship entity waits in the queue *Inbound Wait for Tug Boat.Queue* until the tugboat becomes available (note the *Infinite Hold* option in the *Type* field). The Tugboat segment will ensure that the tugboat constantly monitors that the queue *Inbound Wait for Tug Boat.Queue* has ships waiting to be towed into port and that the high-tide condition is satisfied.

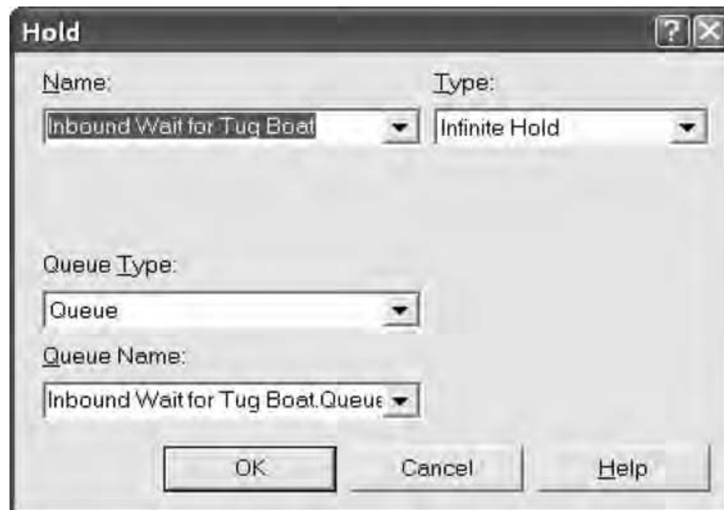


Figure 13.6 Dialog box of the *Hold* module *Inbound Wait for Tug Boat*.

13.3.2 Tug Boat Operations

Tug boat operations are implemented in the Arena segment depicted in Figure 13.7. At time 0, a single entity representing the tug boat is created by the *Create* module, called *Create Tug*, whose dialog box is displayed in Figure 13.8. The newly created tug boat entity then proceeds to the *Station* module, called *Tug Station*, whose dialog box is displayed in Figure 13.9.

Station modules are used in this model as entry points to a geographic location or to a set of modules representing a logical segment of the model. As will be seen later,

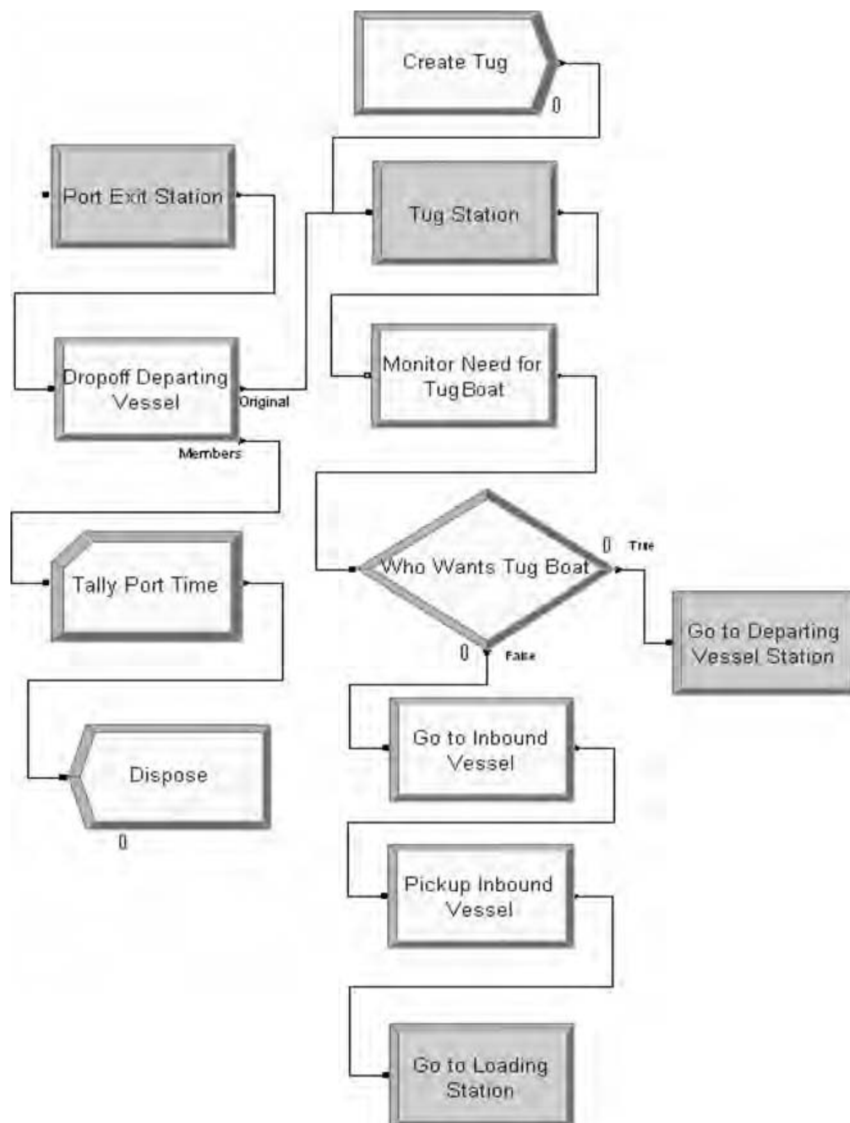


Figure 13.7 Arena model segment implementing tugboat operations at Port Tamsar.

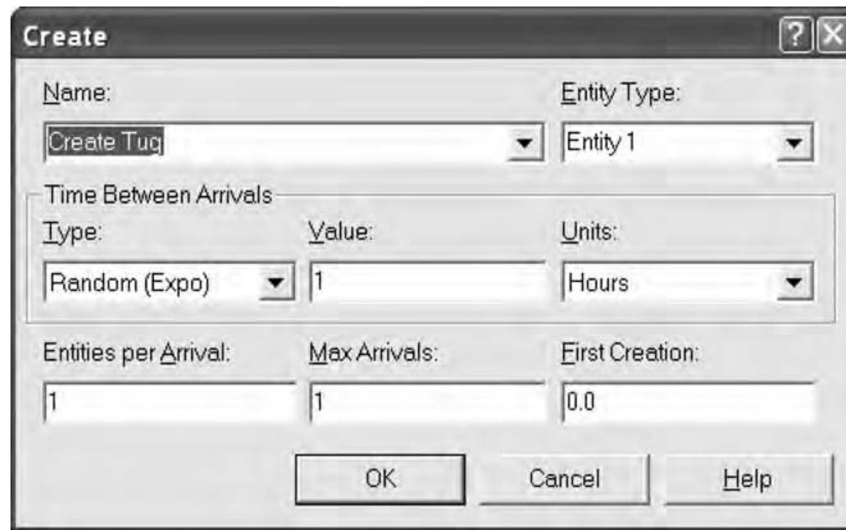


Figure 13.8 Dialog box of the *Create* module *Create Tug*.

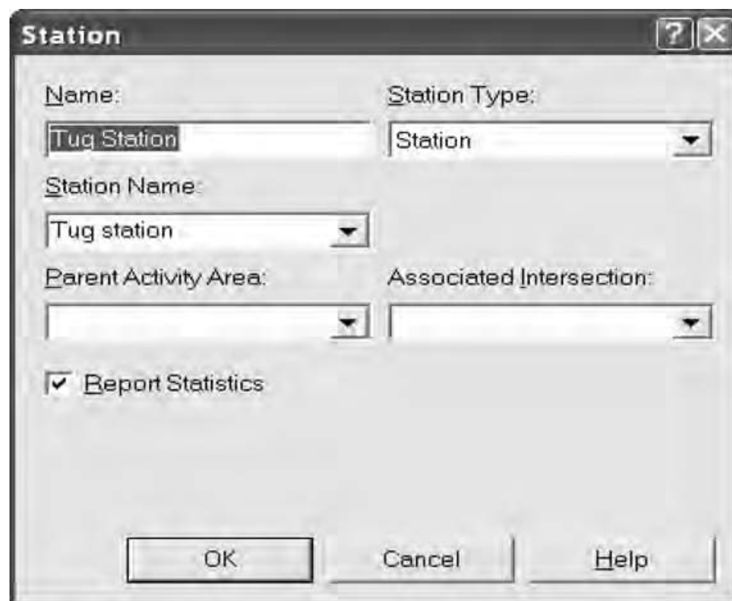


Figure 13.9 Dialog box of the *Station* module *Tug Station*.

Route modules (from the *Advanced Transfer* template panel) are used to shuttle entities among *Station* modules. The blank (unused) fields in Figure 13.9 are related to modeling automated guided vehicles (AGVs), which are outside the scope of this book.

Having entered module *Tug Station*, the tug boat entity proceeds to the *Hold* module, called *Monitor Need for Tug Boat*, whose dialog box is displayed in Figure 13.10. Here, the tugboat entity continually monitors the queues *Inbound Wait for TugBoat.Queue*

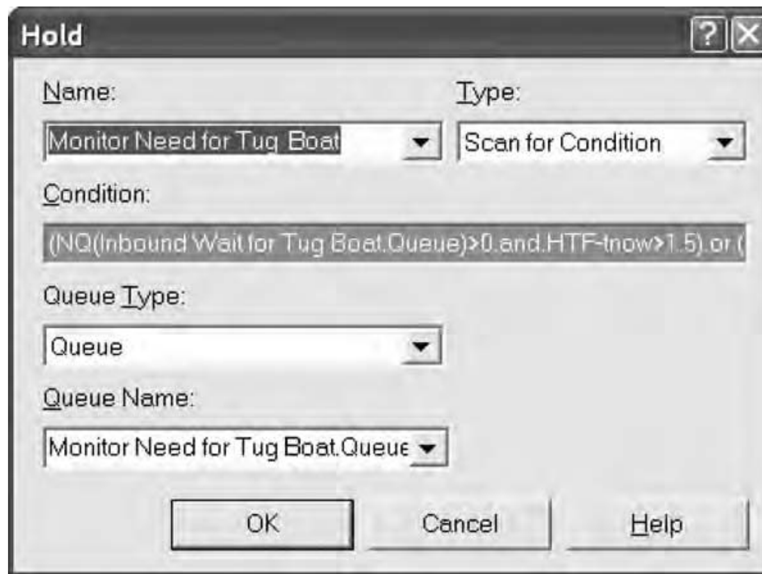


Figure 13.10 Dialog box of the *Hold* module *Monitor Need for Tugboat.*

and *Outbound Wait for TugBoat.Queue* for ships calling on its services (see the *Condition* field). It also ensures that there is enough high-tide time to get a ship out of the queues and tow it to its destination.

Note that it takes a total of 1.5 hours to tow a ship from queue *Inbound Wait for TugBoat.Queue* and 2 hours from queue *Outbound Wait for TugBoat.Queue*. The global variable *HTF* in the *Condition* field stores the end time of the next high tide assigned in the *Assign* module, called *Open* (this will be further discussed in Section 13.3.4). More formally, the *Hold* module monitors the condition

(NQ(Inbound Wait for TugBoat.Queue) > 0 .and. HTF–tnow > 1.5)

.or.

(NQ(Outbound Wait for TugBoat.Queue) > 0 .and. HTF–tnow > 2),

which monitors the residual high-tide period and scans for ship entities waiting for the tug boat in either of the *Hold* module queues housing inbound or outbound ships. While the scan condition is false (no ships are waiting to be towed), the tug boat entity resides in queue *Monitor Need for TugBoat.Queue*. However, once the scan condition becomes true, it will immediately proceed to the *Decide* module, called *Who Wants Tug Boat*, whose dialog box is displayed in Figure 13.11. In this *Decide* module the tug boat entity finds out whether it is needed by an inbound ship or by an outbound ship (which has a higher priority), and proceeds accordingly to the appropriate queue.

Decide modules are used in our model to dispatch entities to requisite destinations probabilistically or depending on prevailing conditions, as indicated by the option selected in the *Type* field. Having selected the option *2-way by Condition*, the *If* field can be used to enter a predicate (condition) involving variables, attributes, entity types or expressions, including random ones, whose specification is then entered in the *Value*

Figure 13.11 Dialog box of the *Decide* module *Who Wants Tug Boat*.

field. In our case, when the expression in the *Value* field is true, the tug boat entity will proceed to tow an outbound ship (by transferring to the *Route* module, called *Go to Departing Vessel Station*); otherwise, it will proceed to tow an inbound ship (by transferring to the *Delay* module, called *Go to Inbound Vessel*). Note carefully how higher towing priority is given to outbound ships, *by checking for their presence first*.

The dialog box for the former *Route* module, called *Go to Departing Vessel Station*, is displayed in Figure 13.12. The *Destination Type* and *Station Name* fields indicate that the tug boat will transfer to the *Station* module, called *Departing Vessel Station*, while the *Route Time* and *Units* fields specify a 1-hour transfer time.

In addition to the *Station* option in the *Destination Type* field, the modeler may select a transfer destination option of type *Sequential* (the destination is specified via a *Sequence* spreadsheet module (from *Advanced Transfer* template panel), *Attribute* (the destination is the value of an entering entity's specified attribute), or *Expression* (the destination is evaluated in an expression). Note that the *Route* module does not have an exit connection. Arena makes sure that the routed entity arrives at its destination at the end of its transfer time.

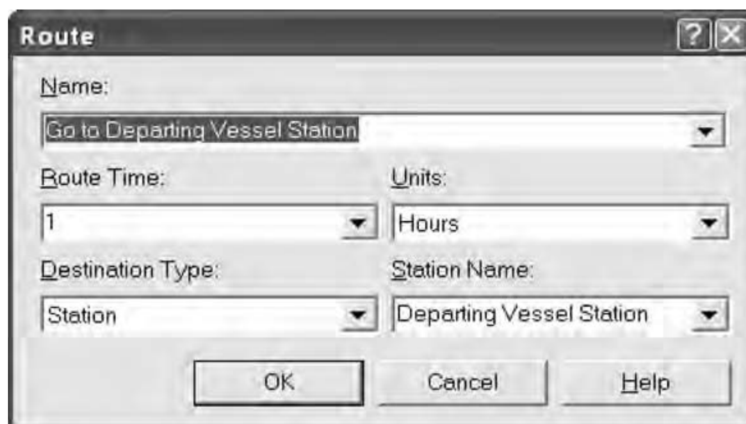


Figure 13.12 Dialog box of the *Route* module *Go to Departing Vessel Station*.

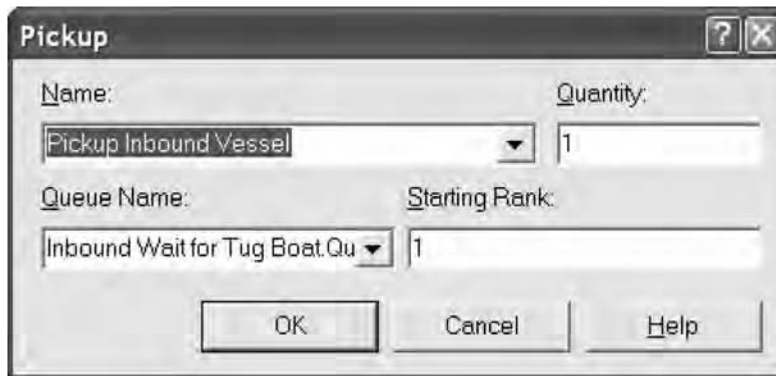


Figure 13.13 Dialog box of the *Pickup* module *Pickup Inbound Vessel*.

Recall that when a tow request is made by an inbound ship, the tug boat enters the *Delay* module, called *Go to Inbound Vessel*. Here, the tug boat is delayed for 30 minutes, which represents travel time to the anchorage area where the requesting ship resides in the queue *Inbound Wait for Tug Boat.Queue*.

The tug boat entity next enters the *Pickup* module (from the *Advanced Process* template panel), called *Pickup Inbound Vessel*, whose dialog box is displayed in Figure 13.13. A *Pickup* module is used by an incoming entity to pick up other entities residing in a queue. The number of entities to be picked up is specified in the *Quantity* field starting from the queue position specified in the *Starting Rank* field. The queue itself is specified in the *Queue Name* field.

The picking entity and the picked-up entities form a *group entity*, where the picked-up members form an internal queue and are identified by their rank (position) in it. Since the picking entity may make several pickups (at different times or places), picked-up members of the group maintain their ID via their rank. As will be seen later, rank information may be used in entity drop-off.

Next, the tug boat enters the *Route* module, called *Go to Loading Station*, to model the 1-hour towing operation of an incoming ship to the loading station on the berth. The corresponding self-explanatory dialog box is displayed in Figure 13.14.

The remainder of the Arena model segment of Figure 13.7 (starting with the module, called *Port Exit Station*), pertains to departing ships and will be explained in the next section when we discuss ship departures.

13.3.3 Coal-Loading Operations

Coal-loading operations are implemented in the Arena segment depicted in Figure 13.15. To commence a coal-loading operation, the tug boat towing an inbound cargo ship enters the *Station* module, called *Coal Loading Station*, whose relevant part of the dialog box is displayed in Figure 13.16.

The tug boat then proceeds immediately to the *Dropoff* module, called *Dropoff Inbound Vessel*, whose relevant part of the dialog box is displayed in Figure 13.17. Here, the towed ship is separated from the tugboat, or in other words, the tugboat drops off the vessel (hence the module's name). The *Dropoff* module is selected from the *Advanced Process* template panel, and is used in conjunction with the *Pickup* module.

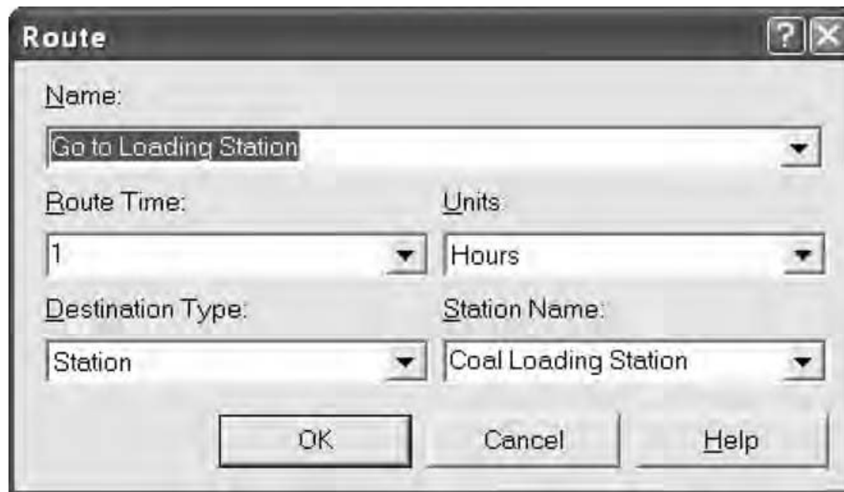


Figure 13.14 Dialog box of the *Route* module *Go to Loading Station*.

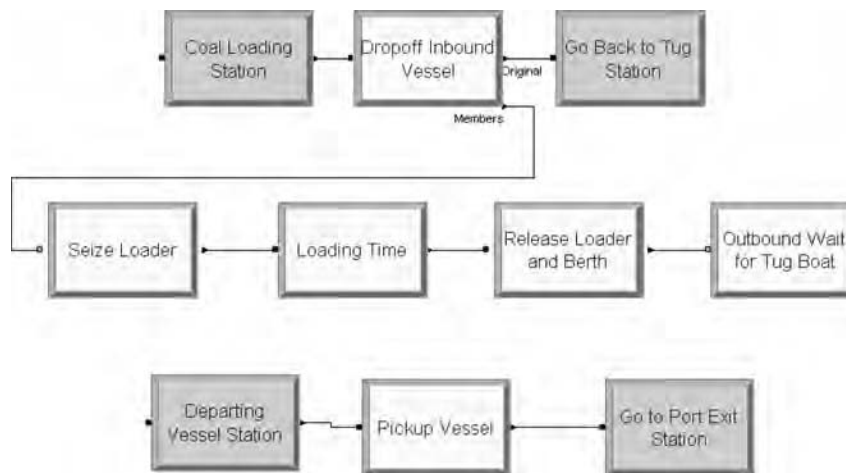


Figure 13.15 Arena model segment implementing coal loading operations at Port Tamsar.

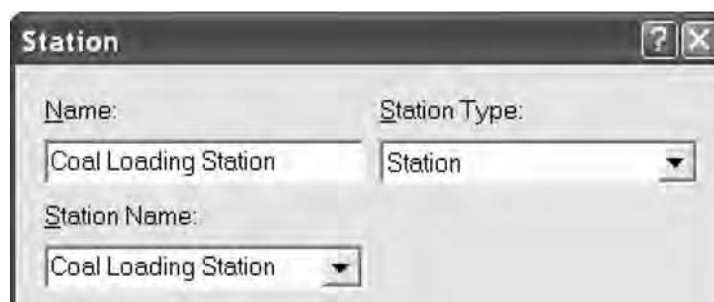


Figure 13.16 Dialog box of the *Station* module *Coal Loading Station*.

Figure 13.17 Dialog box of the *Dropoff* module *Dropoff Inbound Vessel*.

Entities entering a *Dropoff* module must be grouped entities, including the entity that picks up all other ones. The *Quantity* field specifies the number of entities to drop off, while the *Starting Rank* field specifies the rank of the entity in the group from which to start the drop-off. The *Member Attributes* field controls the attributes of the grouped entities. In Figure 13.17, the dropped-off entities are instructed to maintain their original attributes. The modeler can also stipulate that all new values or some of them be assigned to entity attributes via the options *Take All Representative Values* and *Take Specific Representative Values*, respectively.

Next, the picking entity and the picked-up entities exit the *Dropoff* module *Dropoff Inbound Vessel* and are routed to their destinations according to their specified connections. Consequently, the tug boat enters next the *Route* module, called *Go Back to Tug Station*, which routes the tug boat entity (after a 1-hour delay) to the *Station* module, called *Tug Station*, located in the tug boat operations segment of Figure 13.7. Simultaneously, the towed-ship entity proceeds to seize the loader by entering the *Seize* module, called *Seize Loader*, whose dialog box is displayed in Figure 13.18. Note that at this

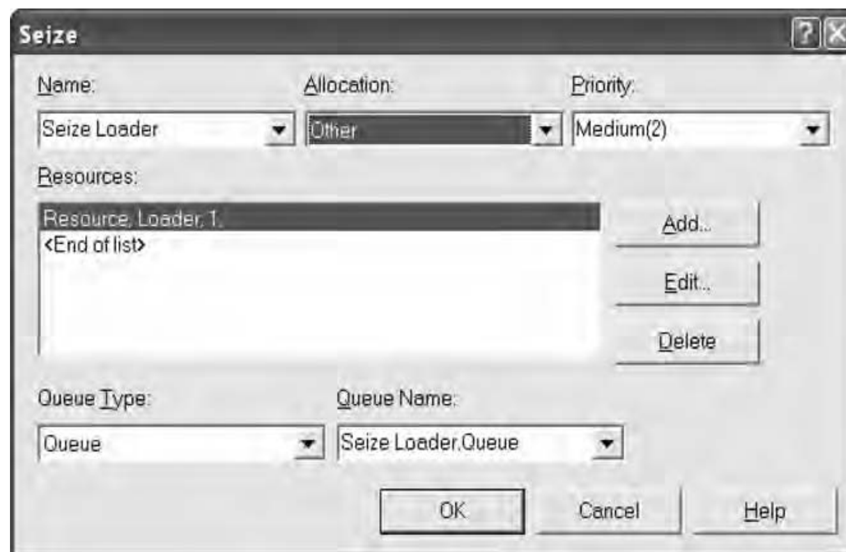


Figure 13.18 Dialog box of the *Seize* module *Seize Loader*.

time the loader is always available, since the previous departing ship (if any) must have already released it. Once the loader is seized, the ship entity proceeds to the *Delay* module, called *Loading Time*, to model a uniform loading time between 14 and 18 hours.

Following the loading delay, the ship entity releases the loader and berth resources simultaneously by entering the *Release* module, called *Release Loader and Berth*, whose self-explanatory dialog box is displayed in Figure 13.19.

Finally, the ship entity proceeds to the *Hold* module, called *Outbound Wait for Tug Boat*, to await its turn to be towed to the anchorage by the tugboat. The dialog box of this module is analogous to that of the *Hold* module, called *Inbound Wait for Tug Boat*, in the ship arrivals segment shown in Figure 13.3. As soon as the ship entity is placed in the queue *Outbound Wait for Tug Boat.Queue*, the tug boat entity will be dispatched in due time from module *Tug Station* in the tug boat operations segment of Figure 13.7 to module *Departing Vessel Station* in the coal-loading segment of Figure 13.15. It then enters the *Pickup* module, called *Pickup Vessel*, whose dialog box is displayed in Figure 13.20. The tug boat entity picks up the waiting ship entity

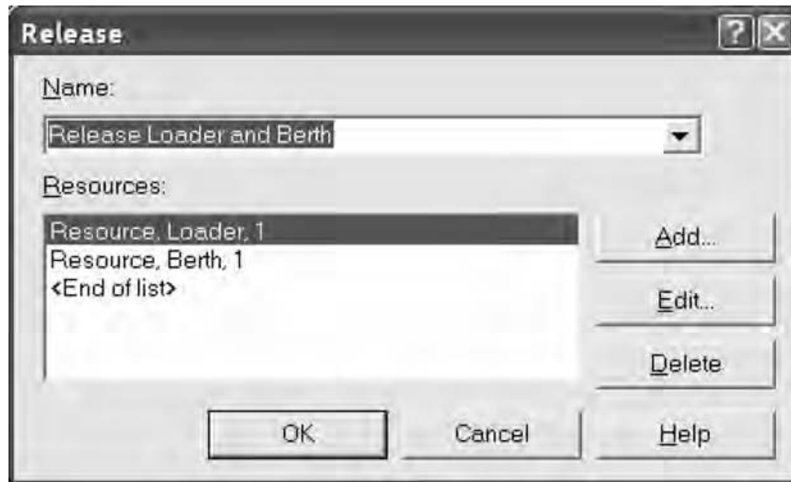


Figure 13.19 Dialog box of the *Release* module *Release Loader and Berth*.

Figure 13.20 Dialog box of the *Pickup* module *Pickup Vessel*.

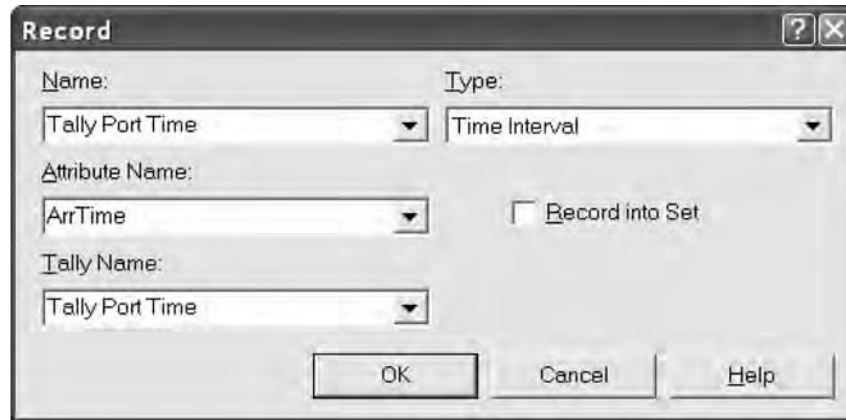


Figure 13.21 Dialog box of the *Record* module *Tally Port Time*.

and proceeds to the port exit by entering the *Route* module, called *Go to Port Exit Station*, which routes it 1 hour later to module *Port Exit Station* in the tug boat operations segment of Figure 13.7. Note that at this point the tug boat and ship entities left the coal-loading operations segment of Figure 13.15, and entered the tug boat operations segment of Figure 13.7. The tug boat entity then drops off the ship entity with its original attributes in the *Dropoff* module, called *Dropoff Departing Vessel*, and proceeds immediately to module *Tug Station* (both in the tug boat operations segment of Figure 13.7). The trip distance is assumed negligible, and therefore the trip is instantaneous.

The ship entity itself proceeds to the *Record* module, called *Tally Port Time*, whose dialog box is displayed in Figure 13.21, to record its sojourn time in the port (port time). To this end, the ship entity makes use of its *ArrTime* attribute that previously recorded its arrival time at the port, taking advantage of the fact that the drop-off retained its original attributes. Finally, the coal-loaded ship entity departs the port via a *Dispose* module.

13.3.4 Tidal Window Modulation

Tidal window modulation is implemented in the Arena segment depicted in Figure 13.22. This segment creates the tidal window and modulates its opening and closing, using the variable *Tidal Window* to represent the tidal state. A value of 0 codes for unfavorable tidal conditions (closed tidal window), while a value of 1 codes for favorable tidal conditions (open tidal window). Thus, all vessels can determine from the value of the variable *Tidal Window* whether they may move into or out of port.

A tidal window entity is created at time 0 in the *Create* module, called *Create Tidal Window*, whose dialog box is displayed in Figure 13.23. Note that since the *First Creation* field is 0.0 and the *Max Arrivals* field specifies precisely one arrival, the values of fields in section *Time Between Arrivals* are irrelevant.

The created tidal window entity first enters the *Assign* module, called *Close*, whose dialog box is displayed in Figure 13.24. Here, the variable *Tidal Window* is set to 0

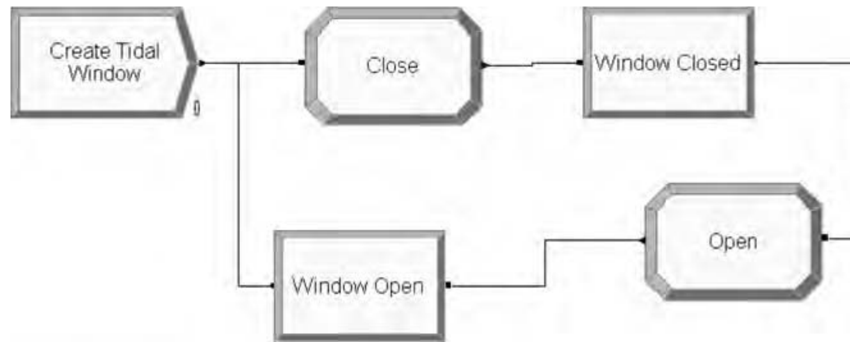


Figure 13.22 Arena model segment implementing tidal window modulation at Port Tamsar.

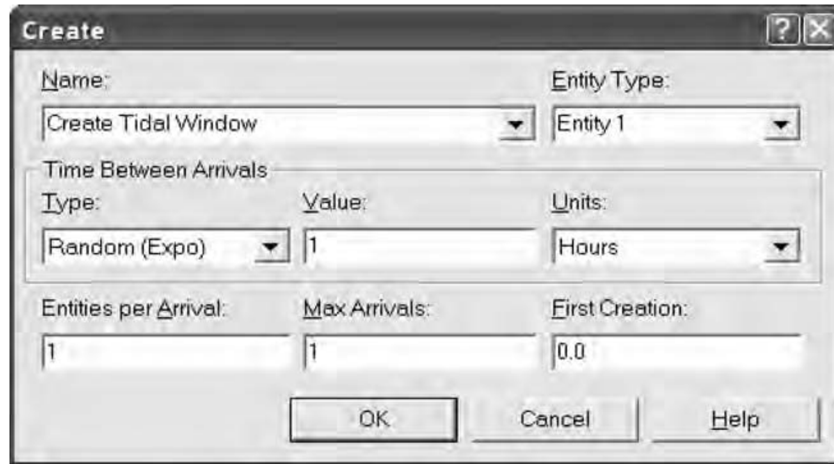


Figure 13.23 Dialog box of the *Create* module *Create Tidal Window.*

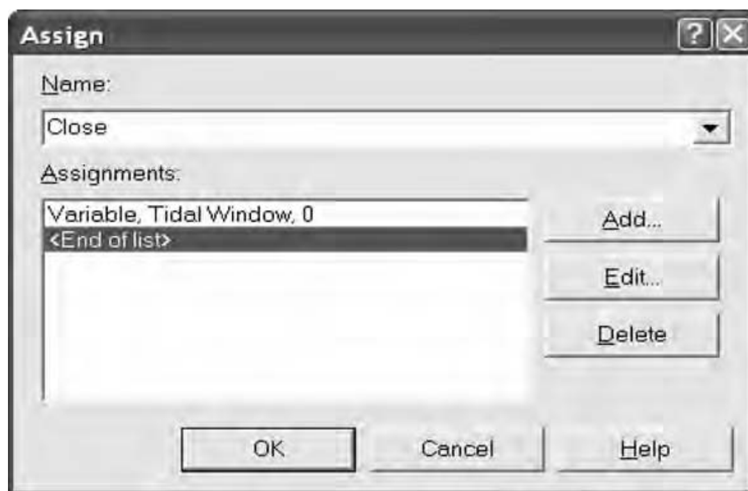


Figure 13.24 Dialog box of the *Assign* module *Close.*

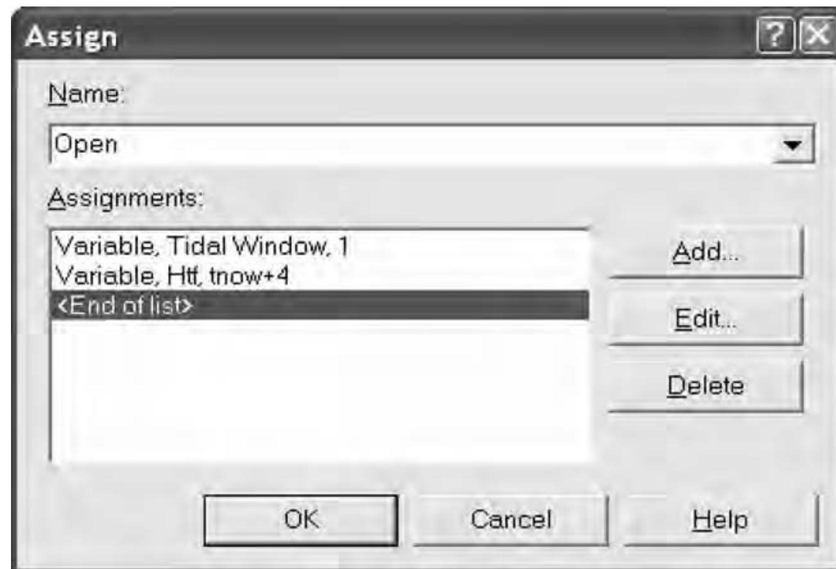


Figure 13.25 Dialog box of the *Assign* module *Open*.

(closed tidal window), after which the tidal window entity proceeds to the *Delay* module, called *Window Closed*, to maintain this tidal state for a duration of precisely 8 hours.

In a similar vein, the tidal window entity then enters the *Assign* module, called *Open*, whose dialog box is displayed in Figure 13.25. Here, the variable *Tidal Window* is set to 1 (open tidal window), and the end time of the high tide (i.e., *Tnow* + 4) is stored in variable *HTF*. The variable *HTF* is checked by the tug boat before attempting any ship towing.

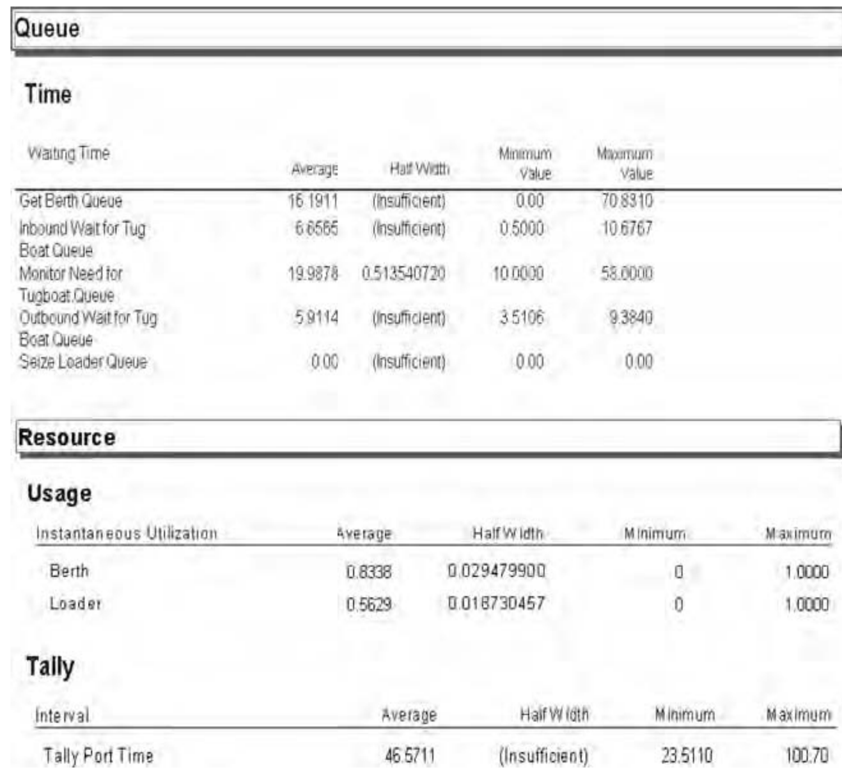
Next, the tidal window entity proceeds to the *Delay* module, called *Window Open*, to enable maritime traffic for 4 hours. The tidal window entity is then routed back to module *Close* to repeat the 12-hour tidal cycle.

13.3.5 Simulation Results for the Bulk Port Model

Port Tamsar was simulated for 1 year (8760 hours of continual operation), and the simulation results from one replication (obtained from the *Queue* option of the *Category Overview* section, the *Resource* option of the *Category by Replication* section, and the *User Specified* section) are displayed in Figure 13.26.

The waiting time of coal ships in queue *Get Berth.Queue* for the loading berth averaged about 16.19 hours. For outbound ships, the waiting time for the tug boat was about 5.91 hours. However, the corresponding waiting times for inbound coal ships averaged about 6.66 hours, due to their lower priority. The tug boat's average waiting time between towing assignments averaged some 19.99 hours.

The *Resource* section reveals that berth utilization was 83.38% and loader utilization was 56.29%. The discrepancy is due to the fact that a ship seizes the berth first from



Queue

Time

Waiting Time	Average	Half Width	Minimum Value	Maximum Value
Get Berth.Queue	16.1911	(Insufficient)	0.00	70.8310
Inbound Wait for Tug Boat.Queue	6.6565	(Insufficient)	0.5000	10.6767
Monitor Need for Tugboat.Queue	19.9878	0.513540720	10.0000	58.0000
Outbound Wait for Tug Boat.Queue	5.9114	(Insufficient)	3.5106	9.3840
Seize Loader.Queue	0.00	(Insufficient)	0.00	0.00

Resource

Usage

Instantaneous Utilization	Average	Half Width	Minimum	Maximum
Berth	0.8338	0.029479900	0	1.0000
Loader	0.5629	0.018730457	0	1.0000

Tally

Interval	Average	Half Width	Minimum	Maximum
Tally Port Time	46.5711	(Insufficient)	23.5110	100.70

Figure 13.26 Simulation results for the Port Tamsar model.

the anchorage location, and consequently, berth utilization is inflated by inbound waiting times for the tug boat as well as travel times to the berth. In contrast, loader times consist solely of loading times. Finally, the sojourn times of cargo ships in the port averaged 46.57 hours. Note that estimating tug boat utilization would require keeping track of all tug boat busy periods in order to calculate the requisite time average using *Time-Persistent* statistics.

The simulation results of Figure 13.26 suggest that the berth constitutes a bottleneck at Port Tamsar, which slows the system down, giving rise to over 16 hours of waiting at the anchorage (partially due to waiting for an open tidal window). From the port authority's point of view, long waiting times are financially deleterious, since ship owners may charge the port authority for excessive delays at the port (the so-called *demurrage* cost). To reduce the wait, ports may employ, in practice, multiple tug boats and loading berths to allow more ships to enter the port when the tidal window is open.

One way to attain this goal is to reduce the lay period, as this would reduce the variability of actual arrival times, and subsequently would also reduce mean waiting times at the anchorage. For example, Figure 13.27 displays simulation results for a lay period of 2 days (down from 5 days in Figure 13.26). Indeed, the average ship waiting time at the anchorage for berth availability (in queue *Get Berth.Queue*) consequently dropped to about 7.46 hours (down from about 16.19 hours in Figure 13.26). The results verify the well-known queueing principle that more regular arrivals give rise to shorter waiting times.

Queue

Time

Waiting Time	Average	Half Width	Minimum Value	Maximum Value
Get Berth.Queue	7.4632	(Insufficient)	0.00	34.4515
Inbound Wait for Tug Boat.Queue	6.4900	(Insufficient)	0.5000	10.8802
Monitor Need for Tugboat.Queue	18.5239	0.442091921	0.06159042	34.0000
Outbound Wait for Tug Boat.Queue	5.9884	(Insufficient)	3.5054	9.4596
Seize Loader.Queue	0.00	(Insufficient)	0.00	0.00

Figure 13.27 Simulation results for the Port Tamsar model with shorter lay periods.

13.4 EXAMPLE: A TOLL PLAZA

This example concerns a transportation system consisting of a toll plaza on the New Jersey Turnpike, and aims to study the queueing delays resulting from toll collection. The system under study is depicted in Figure 13.28.

The toll plaza consists of two exact change (EC) lanes, two cash receipt (CR) lanes, and one easy pass (EZP) lane. Arriving vehicles are classified into three groups as follows:

1. Fifty percent of all arriving cars go to EC lanes, and their normal service time distribution is Norm(4.81, 1.01). Recall that only the non-negative values sampled from this distribution are used by Arena (see Section 4.2).
2. Thirty percent of all arriving cars go to CR lanes, and their service time distribution is $5 + \text{Logn}(4.67, 2.26)$.
3. Twenty percent of all arriving cars go to EZP lanes, and their service time distribution is $1.18 + 4.29 \times \text{Beta}(2.27, 3.02)$.

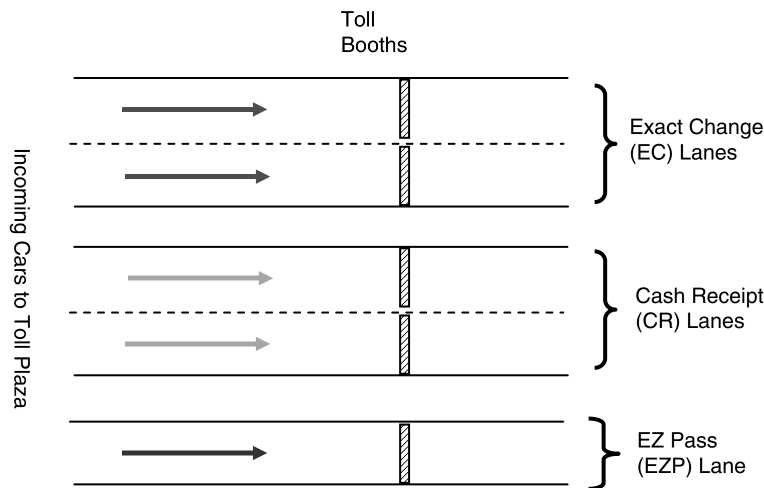


Figure 13.28 A toll plaza system on the New Jersey Turnpike.

To simplify matters, we assume that an incoming car always joins the shortest queue in its category (EC, CR, or EZP). We further assume that no *jockeying* between queues takes place. That is, once a car joins a queue in front of a tollbooth, it never switches to another queue.

Traffic congestion is distinctly nonstationary, varying widely by time of day. As expected, traffic is heavier during the morning rush hour (6 A.M.–9 A.M.) and the evening rush hour (4 P.M.–7 P.M.), and tapers off during off-peak hours. Table 13.1 summarizes vehicle interarrival time distributions over each 24-hour period.

The number of operating cash receipt booths varies over time. Since such booths must be manned, and therefore are expensive to operate, one of them is closed during the off-peak hours. Only during morning and evening rush hours do all cash receipt booths remain open.

Typical performance analysis objectives for the toll plaza system address the following issues:

- What would be the impact of additional traffic on car delays?
- Would adding another booth markedly reduce waiting times?
- Could some booths be closed during light traffic hours without appreciably increasing waiting times?
- What would be the impact of converting some cash receipt booths to exact change booths or to easy pass booths?
- How would waiting times be reduced if both cash receipt booths were to be kept open at all times?

Of course, additional issues may be specific to particular toll plazas under study, but in our case we wish to address the last issue in the list, using the performance metrics of average time to pass through the system and booth utilization.

An Arena model for the toll plaza system presented here is depicted in Figure 13.29. The model can be decomposed into the following segments: creation of car entities from the appropriate distributions over various time periods, dispatching a car to the appropriate tollbooth with the shortest queue, and serving incoming cars. To this end, we use the *Set* construct to facilitate modeling of module sets (model components) with analogous logic (e.g., multiple tollbooths). Consequently, the number of such components would be easily modifiable. The structure of the toll plaza model will be described next in some detail, along with simulation results.

Table 13.1

Interarrival time distributions by time of day

Time Period (hours)	Interarrival Time Distribution (seconds)
12 A.M.–6 A.M.	8 + Gamm(4.4, 4.12)
6 A.M.–9 A.M.	Tria(1.32, 1.57, 1.76)
9 A.M.–4 P.M.	2.64 + Weib(0.82, 4.5)
4 P.M.–7 P.M.	Tria(1.32, 1.57, 1.76)
7 P.M.–12 A.M.	4.2 + Gamm(0.87, 8.24)

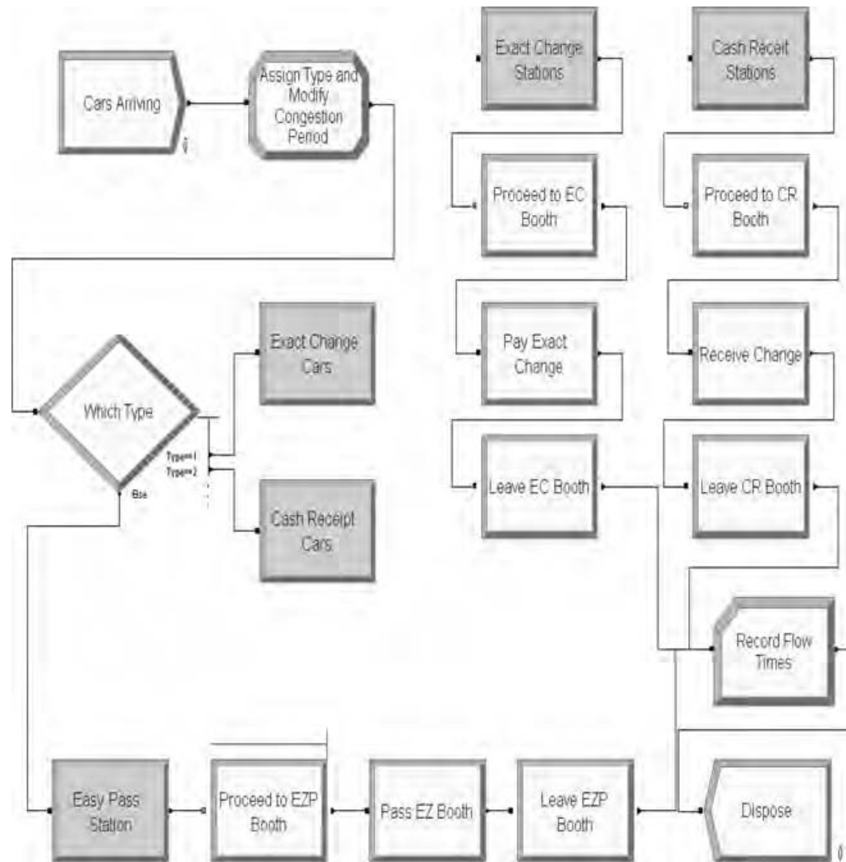


Figure 13.29 Arena model of the toll plaza system.

13.4.1 Arrivals Generation

Car arrivals are generated in the *Create* module, called *Cars Arriving*, whose dialog box is displayed in Figure 13.30. Here, the *Time Between Arrivals* section specifies time-dependent sampling distributions for car interarrival times via the expression *Int_Times(k)*. The variable *Int_Times* is a vector of sampling expressions, whose specification is displayed in Figure 13.31, and *k* is a variable representing an index between 1 and 5 for this vector.

Note that *k* and *Int_Times* correspond to entries in Table 13.1. In order to access the requisite distribution, the variable *k* is set to the appropriate row index by each arriving-car entity depending on the time of day of its arrival. This assignment takes place immediately following a car creation, when the car entity enters the *Assign* module, called *Assign Type and Modify Congestion Period*, whose dialog box is displayed in Figure 13.32. The assignment of *k* is made in the third line of the *Assignments* section, which partially displays the expression (the conditions in parentheses are in seconds and an asterisk denotes multiplication):

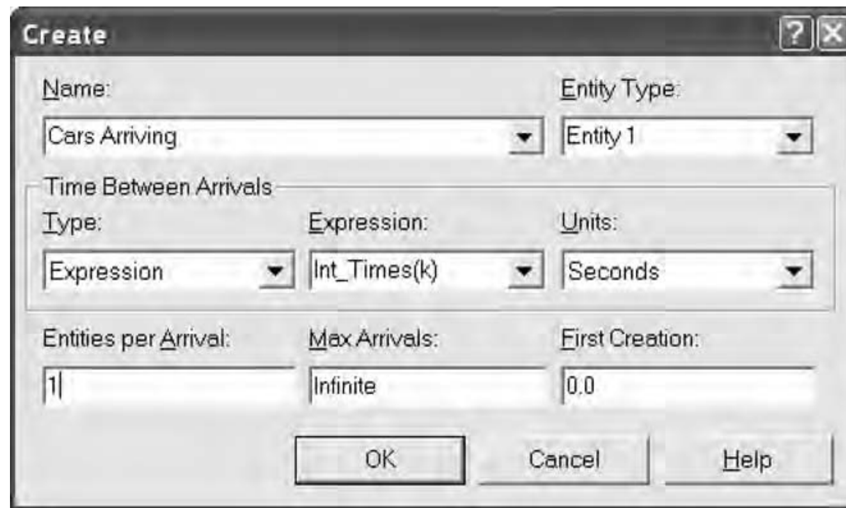


Figure 13.30 Dialog box of the *Create* module *Cars Arriving*.

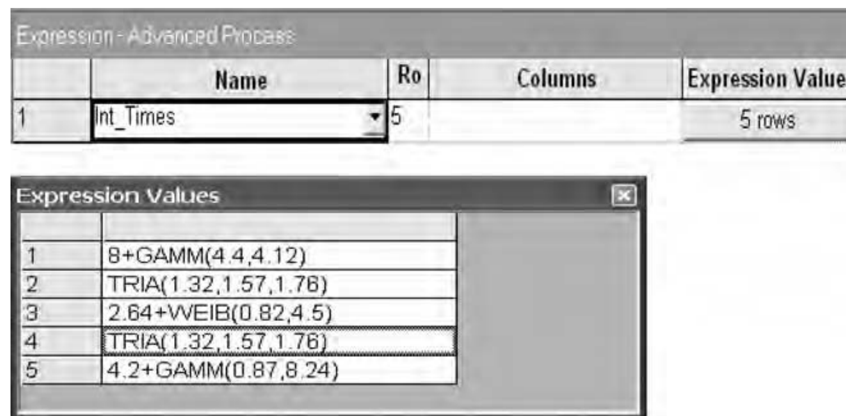


Figure 13.31 Dialog spreadsheet for specifying a vector of sampling distributions for time-dependent interarrivals in vector *Int_Times* (top) and the *Expression Values* dialog spreadsheet for specifying sampling expressions (bottom).

$$
\begin{aligned}
k = 1 * (t < 6 * 3600) &+ \\
2 * (t >= 6 * 3600.and.t < 9 * 3600) &+ \\
3 * (t >= 9 * 3600.and.t < 16 * 3600) &+ \\
4 * (t >= 16 * 3600.and.t < 19 * 3600) &+ \\
5 * (t >= 19 * 3600)&
\end{aligned}
$$

Note that the variable t is set to the simulation clock variable *Tnow* in the second row of the *Assignments* section. Consequently, precisely one predicate (condition) is true (evaluates to 1) at any given time *Tnow*, and all others are false (evaluate to 0), thereby yielding the requisite row index. In addition, the first line of the *Assignments* section samples each car's type number (1 = exact change, 2 = cash receipt, 3 = EZ pass) with the requisite probability (via the Arena *DISC* distribution), and assigns it to the car

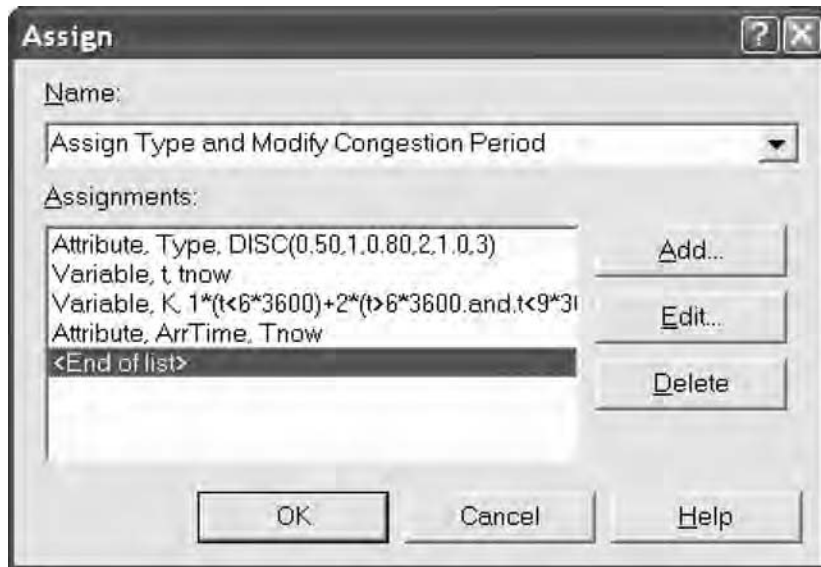


Figure 13.32 Dialog box of the *Assign* module *Assign Type and Modify Congestion Period*.

entity's *Type* attribute. The fourth line saves the car entity's arrival time, *Tnow*, in its *ArrTime* attribute (for future use in computing its system time).

Recall the discussion in Section 5.8 concerning the *Schedule* module, and note that we have a similar problem here. More specifically, changes in the interarrival time distributions in our model do not take effect at interval boundaries, but rather when the first car arrives in the new interval, and consequently will apply only to subsequent cars in this interval. If changes at interval boundaries were desired, then an alternative model would use a single entity circulating between *Assign* and *Delay* modules (one set per interval) to assign the requisite time-dependent distribution to a variable.

13.4.2 Dispatching Cars to Tollbooths

Once the assignments are made, the car entity proceeds to the *Decide* module, called *Which Type*, whose dialog box is displayed in Figure 13.33. This module dispatches a car entity to an appropriate tollbooth, in accordance with the car's *Type* attribute.

The notion of a location is implemented via the *Station* module (from the *Advanced Transfer* template panel), which may represent a single station (location) or a set of stations (multiple locations), each modeling a physical or a logical location in the model. In our case, the single EZP tollbooth is modeled by a single *Station* module, and the multiple EC and CR tollbooths are modeled as sets of *Station* modules, since the logic for each tollbooth type is analogous.

To illustrate, Figure 13.34 displays the dialog box of the *Station* module set representing EC tollbooths. To this end, the option *Set* is selected in the *Station Type* field (as opposed to the option *Station*, which defines a single *Station* module). The modeler then specifies the set member names in the *Station Set Members* field with the aid of the three buttons to its right.

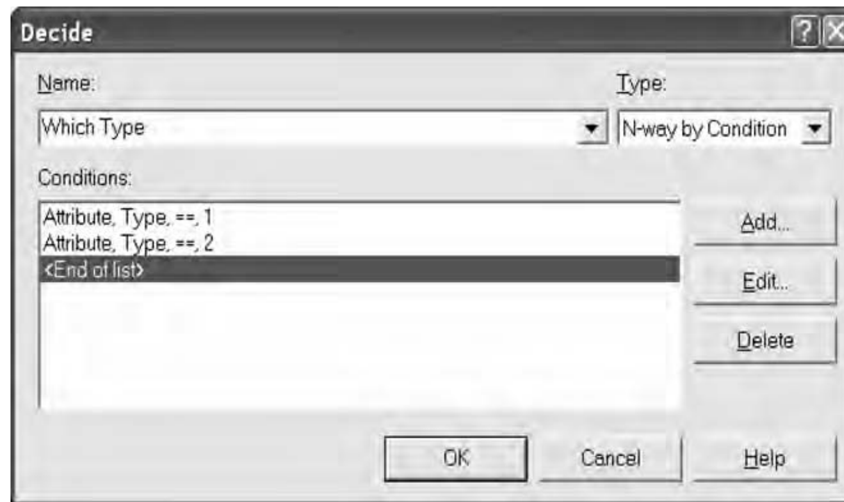


Figure 13.33 Dialog box of the *Decide* module *Which Type*.

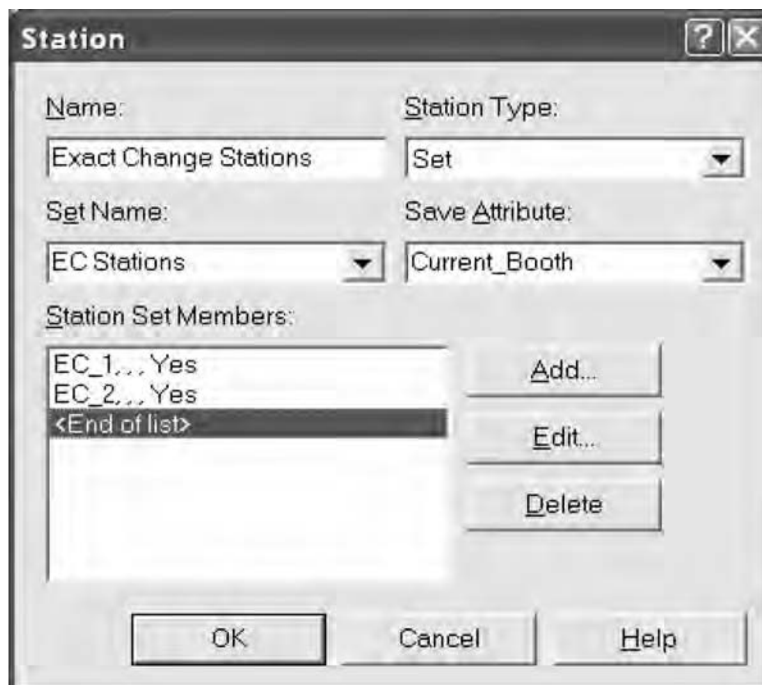


Figure 13.34 Dialog box of the *Station* module set for EC tollbooths.

In the *Station* module dialog box, the two EC tollbooths are named *EC_1* and *EC_2* (in a similar vein, the two CR tollbooths are named *CR_1* and *CR_2* in their *Station* module set). The name of this *Station* module set is specified in the *Set Name* field. The *Save Attribute* field is used to specify the attribute name in which to store the station ID

when an entity enters a particular member of the *Station* module set. This ID is just the rank (position) of the requisite *Station* name in the list of the *Station Set Members* field. In our case, the *Station* module ID is assigned elsewhere to attribute *Current_Booth* of the incoming car entity. Saving that ID can be useful, since it can be used to index the associated resource, queue, and so on.

While tollbooths are modeled by *Station* modules, dispatching cars to tollbooths by type is implemented in a *PickStation* module (from the *Advanced Transfer* template panel), which allows an entity to select a particular *Station* module among a set of such modules, based on some prescribed condition. Accordingly, module *Which Type* dispatches car entities of type 1 (EC), type 2 (CR), and type 3 (EZP), respectively, to the *PickStation* modules *Exact Change Cars*, *Cash Receipt Cars*, and *Station* module *Easy Pass Station*, based on their *Type* attribute.

Each car entity that enters its *PickStation* module selects a station from a set of stations (tollbooths), as illustrated in the dialog box of Figure 13.35 for EC cars (type 1). Here, the list of *Station* modules and queues to choose from is constructed by the modeler in the *Stations* section with the aid of the three buttons to its right. Each entry specifies a *Station* module (recall that *EC_1* and *EC_2* label the two EC tollbooths) and an associated queue (*ECSQ_1* and *ECSQ_2* label the queues in the EC tollbooths).

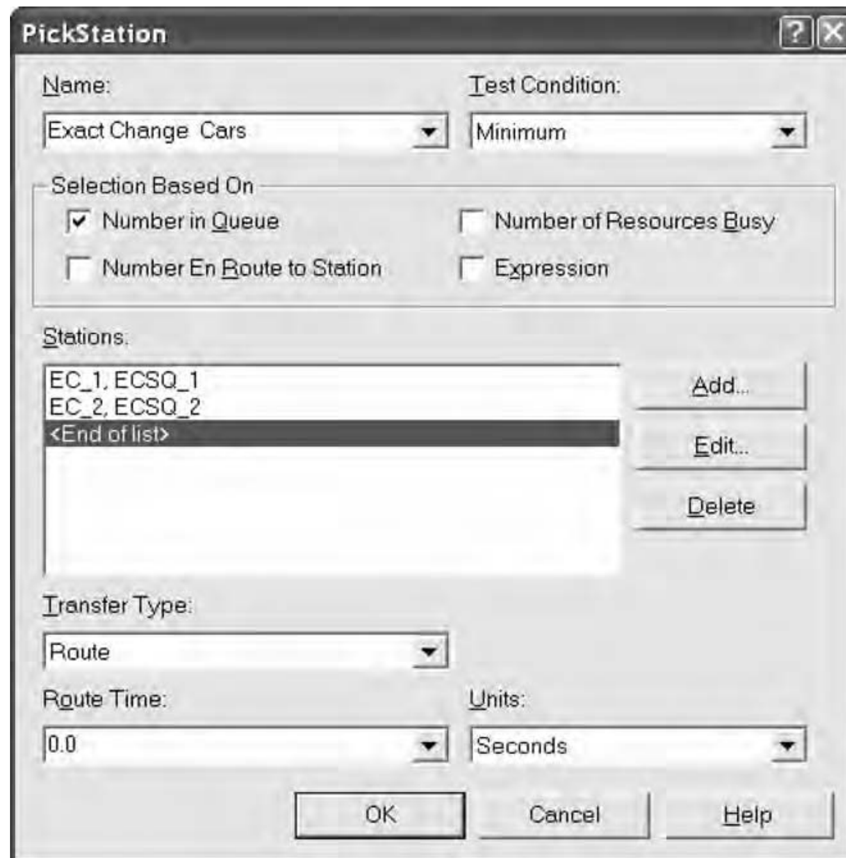


Figure 13.35 Dialog box of the *PickStation* module *Exact Change Cars*.

The selected option *Minimum* in the *Test Condition* field and the check-marked option *Number in Queue* in the *Selection Based On* section instruct the car entity to select a tollbooth with the shortest queue. The modeler can alternatively specify option *Maximum* and base tollbooth selection on other quantities, including an arbitrary expression. In case of ties, the first tollbooth satisfying the condition is selected. The *Transfer Type* field has been set to option *Route*, since it allows the modeler to specify a travel time in the *Route Time* and *Units* fields. Other *Transfer Type* options are *Connect* (an ordinary logical connection), *Transport* (for modeling transporters, such as trucks or forklifts), and *Convey* (for modeling conveyors). Transporters will be treated in a separate example in Section 13.5.

CR cars (type 2) require a different treatment of the associated *PickStation* module *Cash Receipt Cars*, whose dialog box is displayed in Figure 13.36. We again select option *Minimum* in the *Test Condition* field. Recall, however, that unlike EC booths, one of the CR booths (say, *CR_1*) is closed during off-peak hours, while the other one (say, *CR_2*) is always open. In order to account for this fact, we select the *Expression* option in the *Selection Based On* section, and specify a separate expression for each CR tollbooth in the *Stations* field. Let resource *CRS_1* and queue *CRSQ_1* be associated with *Station* module *CR_1*, and similarly, let resource *CRS_2* and queue *CRSQ_2* be associated with *Station* module *CR_2*.

Finally, recall that *MR(R)* and *NQ(Q)* are the Arena functions returning the current number of resource units in resource *R* and the current number of entities in queue *Q*,

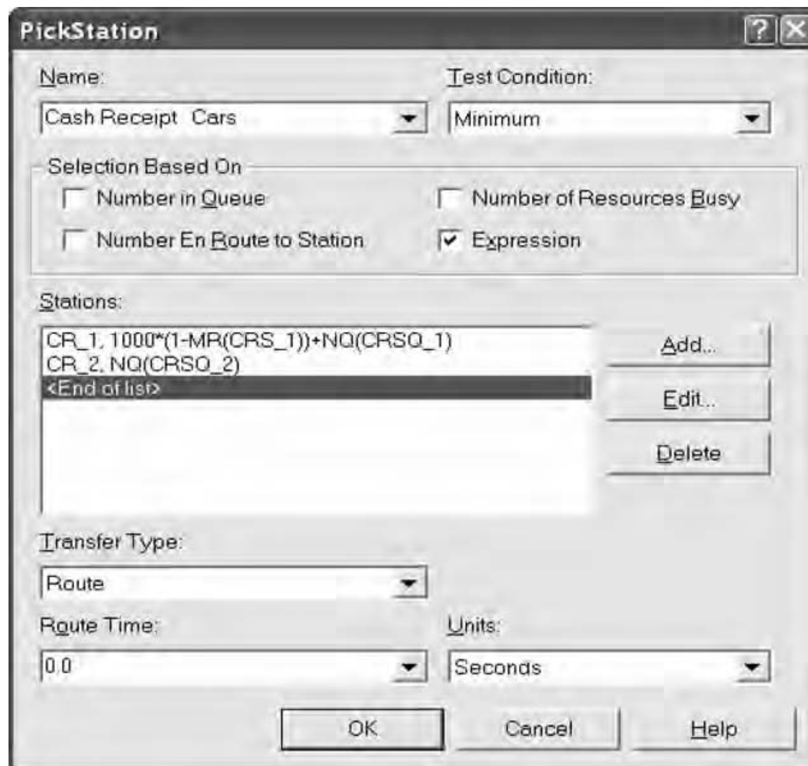


Figure 13.36 Dialog box of the *PickStation* module *Cash Receipt Cars*.

respectively, and suppose that queue *CRSQ_2* can accommodate less than 1000 cars (otherwise, replace 1000 by a sufficiently large number, which surely exists). Now, to ensure that the expressions in the *Stations* section are properly defined, consider the following two cases pertaining to the expression for *CR_1*:

1. If tollbooth *CR_1* is open, then *1 - MR(CRS_1) = 0*, and the expression for *CR_1* evaluates to *NQ(CRSQ_1)*. The tollbooth selection will be made based on the minimum of *NQ(CRSQ_1)* and *NQ(CRSQ_2)*, as required.
2. Conversely, if tollbooth *CR_1* is closed, then *1 - MR(CRS_1) = 1* and *NQ (CRSQ_1) = 0*, and the expression for *CR_1* evaluates to 1000. Since this number exceeds *NQ(CRSQ_2*) by assumption, tollbooth *CR_2* is sure to be selected, as required.

13.4.3 Serving Cars at Tollbooths

Once a car entity is sent to its respective tollbooth, it (possibly) queues up in the associated resource queue, and eventually pays the toll and leaves the toll plaza. In particular, cars departing from the *Station* module *Exact Change Stations* enter the *Seize* module, called *Proceed to EC Booth*, whose dialog box is displayed in Figure 13.37.

In the *Seize* module, the car entity attempts to seize the corresponding *Station* module's resource (tollbooth's server). But in order to refer to that resource, we need to specify a set of resources (servers) in the *Resources* field corresponding to the station (tollbooth) set. For example, for the EC tollbooths, the resources at station *EC_1* and *EC_2* are named *ECS_1* and *ECS_2*, respectively, and the corresponding queues are named *ECSQ_1* and *ECSQ_2*.

To specify these constructs, the requisite sets are created first. More specifically, to create a set of queues, the *Set* option is selected in the *Queue Type* field, and its name is

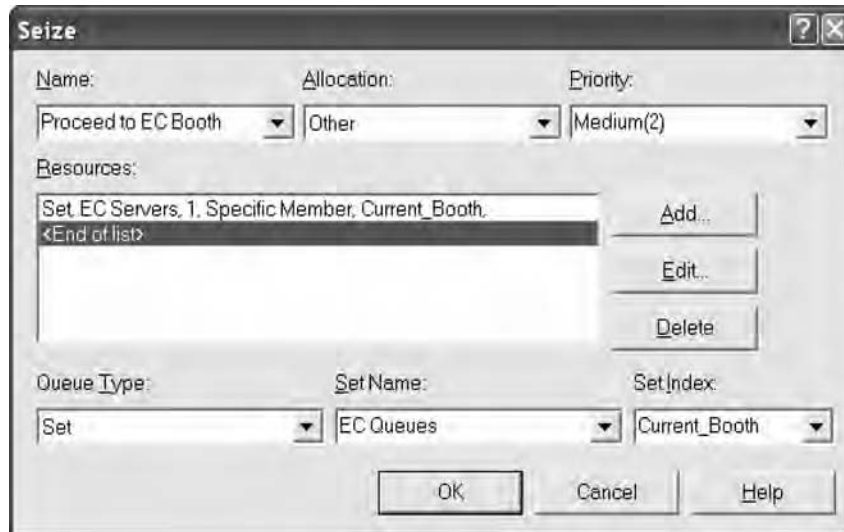


Figure 13.37 Dialog box of the *Seize* module *Proceed to EC Booth*.

entered in the *Set Name* field. Next, the *Add* button is used to create the requisite resource set, by popping up the *Resources* dialog box displayed in Figure 13.38. Here, the *Set* option is first selected in the *Type* field, and its name is entered in the *Set Name* field. Next, the *Specific Member* option is selected in the *Selection Rule* field, and the attribute name *Current_Booth* is entered in the *Set Index* field (recall that the latter will hold the appropriate ID of the resource to be seized). Other selection rules include random selection and cyclical selection.

To create resource members of the *EC Servers* set, the modeler uses the *Set* module from the *Basic Process* template panel to pop up the dialog spreadsheet at the bottom of Figure 13.39. More specifically, the bottom dialog spreadsheet pops up first,

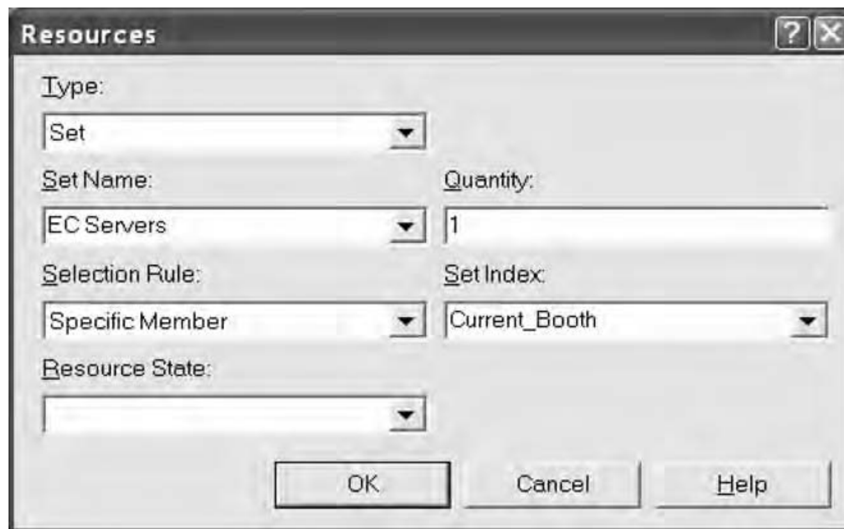


Figure 13.38 Dialog box of resource set *EC Servers*.

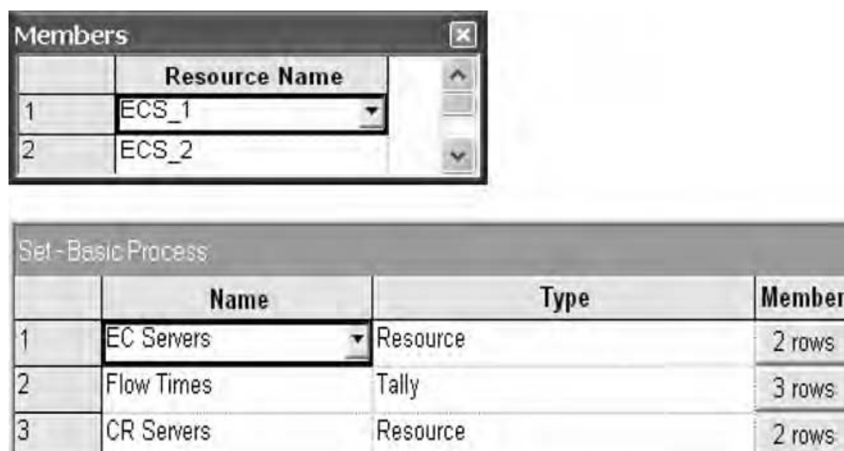


Figure 13.39 Dialog spreadsheet views of the *Set* module (bottom) and the *Members* dialog spreadsheet for specifying of resource set *EC Servers* (top).

with each row corresponding to a distinct set. Generally, if a requisite set is not yet created, then it can be created as a new row in the spreadsheet by double clicking on the text message *Double-click here to add a new row*. The new set is specified in the *Name* column, and its type selected in the *Type* column. Once the set is created and appears as a spreadsheet row, the modeler clicks the button under the *Members* column and pops up the dialog box at the top of Figure 13.39. New set members are then added as rows in the top spreadsheet, similarly to set addition.

The creation of queue sets and their members is analogous, except that the *Advanced Set* module (from the *Advanced Process* template panel) is used instead of the *Set* module. Figure 13.40 displays the spreadsheet views of the *Advanced Set* module (bottom) and the members of the queue set *EC Queues* (top), where the attribute *Current_Booth* holds the ID of the corresponding tollbooth queue.

Once a car entity succeeds in seizing its respective server resource (tollbooth), it enters a *Delay* module in its path for the duration of the toll payment activity. For example, Figure 13.41 displays the dialog box of the *Delay* module for toll payment by EC cars.

Recall that one CR tollbooth (specifically, *CR_1*) is closed during off-peak hours. This wrinkle is implemented by using the *Schedule* module spreadsheet from the *Basic*

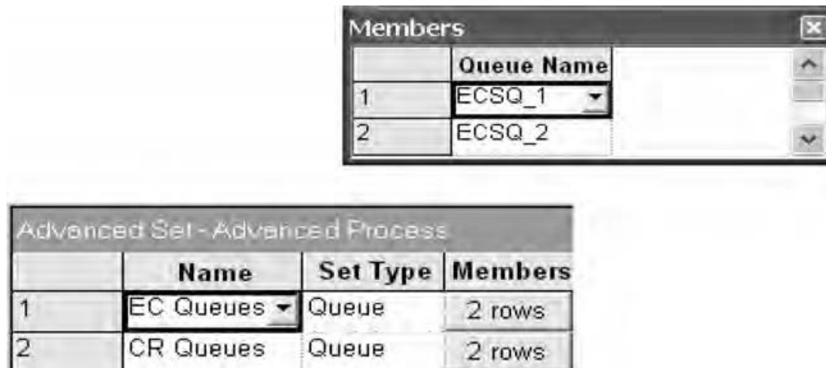


Figure 13.40 Dialog spreadsheet views of the *Advanced Set* module (bottom) and the *Members* dialog spreadsheet for specifying queue set *EC Queues* (top).

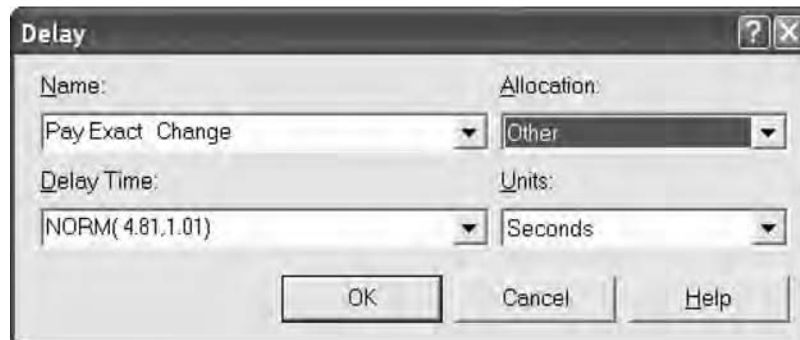
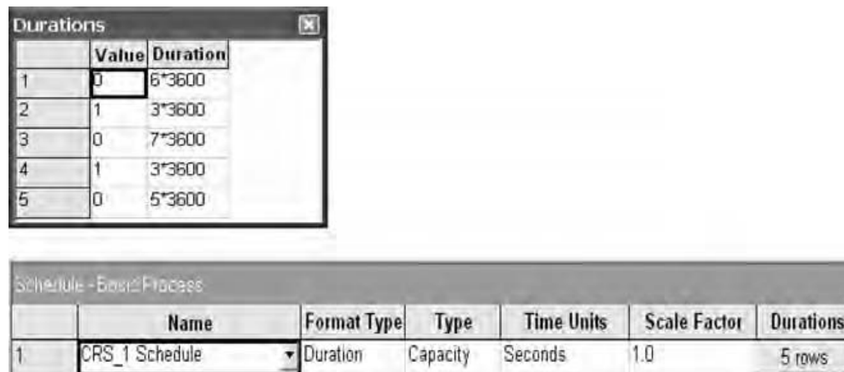


Figure 13.41 Dialog box of the *Delay* module *Pay Exact Change*.

Process template panel to pop up the corresponding note insertion below spreadsheet at the top of Figure 13.42. The association of the resource *CRS_1* with the schedule *CRS_1 Schedule* is specified in the *Resource* module (not shown), by selecting the *Based on Schedule* option in the *Type* column, and entering the name *CRS_1 Schedule* in the *Schedule Name* column.

On payment completion, each car entity proceeds to a corresponding *Release* module in its path to release the tollbooth resource (again using its *Current_Booth* attribute to index into the resource set). For example, Figure 13.43 displays the dialog box of the *Release* module for EC cars.

At this point the car entity has completed the toll paying activity, and its only remaining task in the model before being discarded at the *Dispose* module (called *Dispose*) is to tally its own flow time in the system. To this end, every car entity enters



Durations

	Value	Duration
1	0	6*3600
2	1	3*3600
3	0	7*3600
4	1	3*3600
5	0	5*3600

Schedule - Basic Process

	Name	Format Type	Type	Time Units	Scale Factor	Durations
1	CRS_1 Schedule	Duration	Capacity	Seconds	1.0	5 rows

Figure 13.42 Dialog spreadsheet of the *Schedule* module (bottom) and the *Durations* dialog spreadsheet for server *CRS_1* at tollbooth *CR_1* (top).

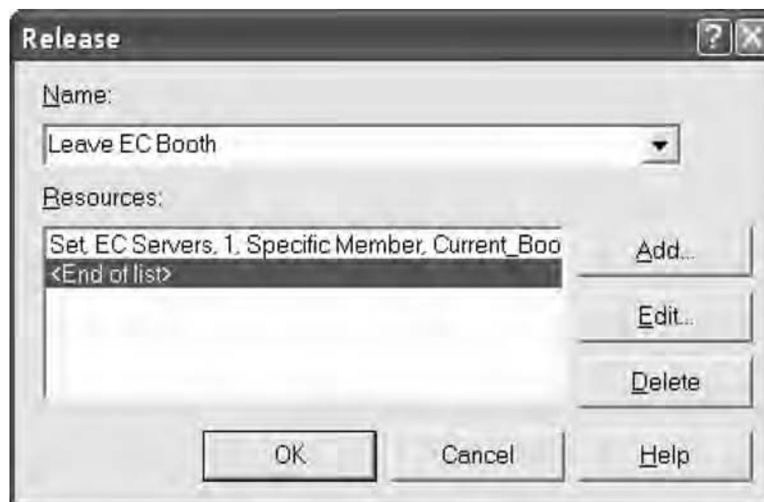


Figure 13.43 Dialog box of the *Release* module *Leave EC Booth*.

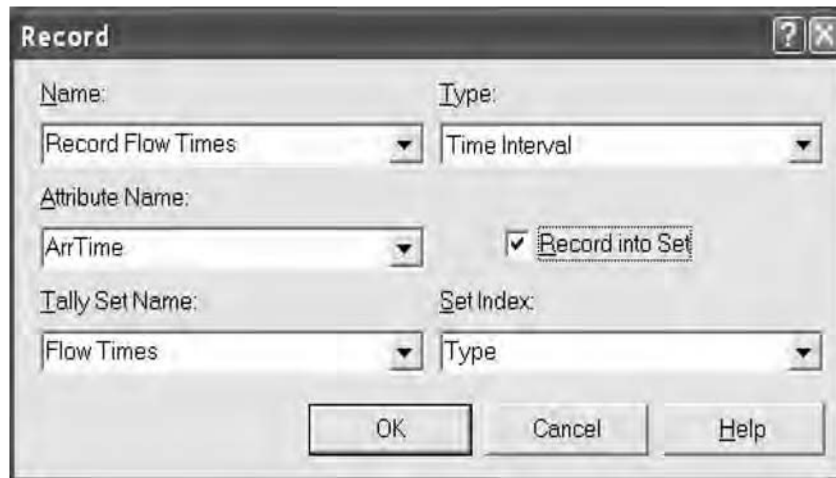


Figure 13.44 Dialog box of the *Record* module *Record Flow Times*.

the *Record* module, called *Record Flow Times*, whose dialog box is displayed in Figure 13.44. Here, the *Type* field is set to the *Time Interval* option, which records the elapsed time from the value stored in the car's attribute specified in the *Attribute Name* field. Since that attribute (*ArrTime*) was previously arranged to store the car's arrival time at the toll plaza, the requisite flow times are indeed collected.

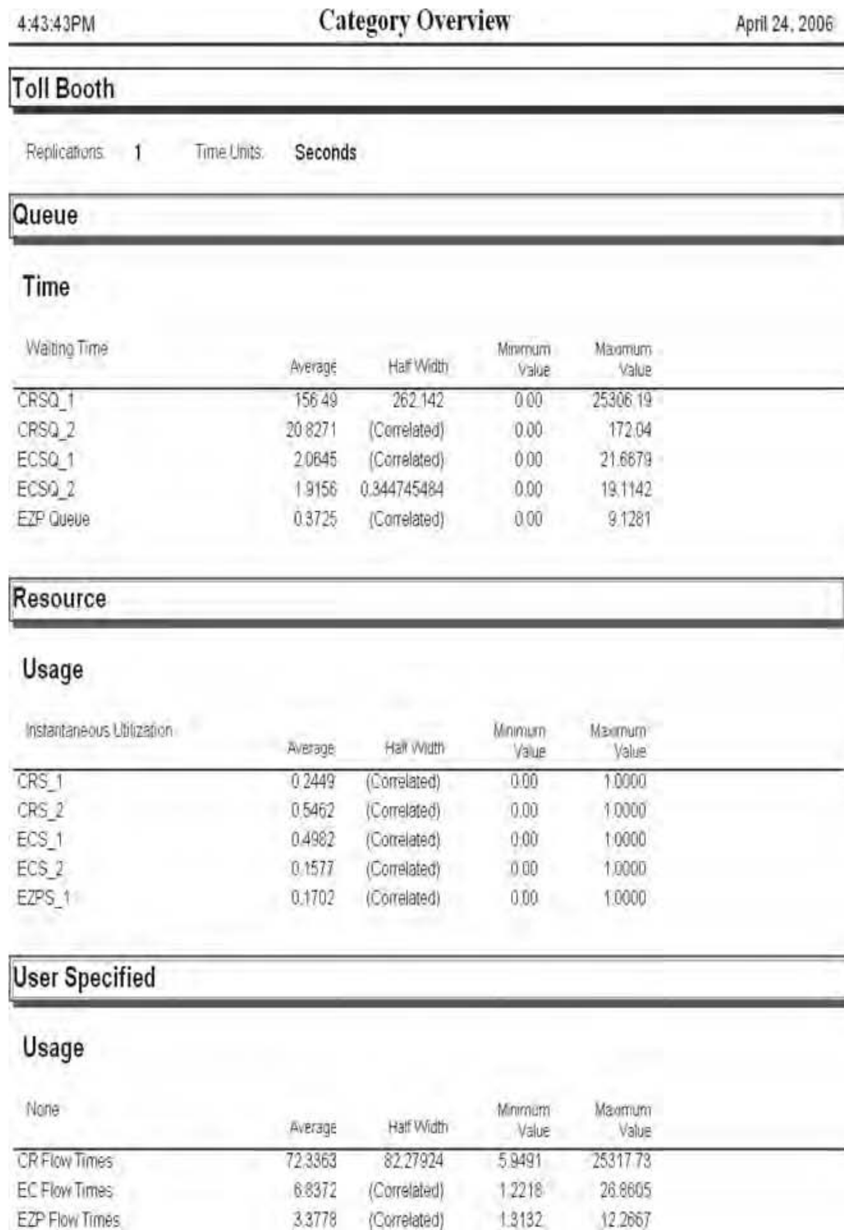
Note that the check box *Record into Set* is marked in Figure 13.44. Note further that a name is specified in the *Tally Set Name* field (*Flow Times*), and that the option selected in the *Set Index* field is the car entity's *Type* attribute. These data arrange to record flow times separately by car type—EC, CR, and EZP. Arena uses a set of three tally statistics to manage system flow-time collection by car type.

An overall examination of the toll plaza model in Figure 13.29 reveals considerable similarity and minor variations in the logic sequence of the three types of cars. The model uses *PickStation* modules for both EC and CR cars but not for EZP cars, since there is only one EZP tollbooth. Although this difference simplifies the model, it essentially limits it to a single EZP booth. There is considerable merit in making the model more general, by repeating the logic of the EC and CR paths for the EZP path. The added generality would allow the analyst to easily vary the number of EZP tollbooths, thereby increasing the experimental range of the model.

13.4.4 Simulation Results for the Toll Plaza Model

The toll plaza model was simulated for $24 \times 60 \times 60 = 86,400$ seconds (1 day), and the simulation results are displayed in Figure 13.45.

The *Queue* section displays delay statistics for each of the five tollbooth queues (two EC queues, two CR queues, and one EZP queue). As expected, CR queues have the longest delays, EC queues have shorter delays, and the EZP queue has a tiny delay. Note that queue *CRSQ_1* has a very large maximal delay, since our model shuts down tollbooth *CR_1* at the onset of the off-peak period, leaving all cars waiting there at


4:43:43PM **Category Overview** April 24, 2006

Toll Booth

Replications: 1 Time Units: Seconds

Queue

Time

Waiting Time	Average	Half Width	Minimum Value	Maximum Value
CRSQ_1	156.49	262.142	0.00	25306.19
CRSQ_2	20.8271	(Correlated)	0.00	172.04
ECSQ_1	2.0645	(Correlated)	0.00	21.6679
ECSQ_2	1.9156	0.344745484	0.00	19.1142
EZP Queue	0.3725	(Correlated)	0.00	9.1281

Resource

Usage

Instantaneous Utilization	Average	Half Width	Minimum Value	Maximum Value
CRS_1	0.2449	(Correlated)	0.00	1.0000
CRS_2	0.5462	(Correlated)	0.00	1.0000
ECS_1	0.4982	(Correlated)	0.00	1.0000
ECS_2	0.1577	(Correlated)	0.00	1.0000
EZPS_1	0.1702	(Correlated)	0.00	1.0000

User Specified

Usage

None	Average	Half Width	Minimum Value	Maximum Value
CR Flow Times	72.3363	82.27924	5.9491	25317.73
EC Flow Times	6.8372	(Correlated)	1.2218	26.8605
EZP Flow Times	3.3778	(Correlated)	1.3132	12.2667

Figure 13.45 Simulation results for the toll plaza model.

the time “stranded” until the onset of the next rush hour. Of course, in a realistic model, the modeler should include logic to empty queue *CRSQ_1* before closing tollbooth *CR_1*.

The *Resource* section displays server utilizations for each of the five tollbooth servers. Since tollbooth *CR_1* is closed during off-peak hours but tollbooth *CR_2* always remains open, the utilization of server *CRS_2* is understandably higher than

that of server *CRS_1*. Furthermore, since tollbooth *EC_1* is preferentially selected over tollbooth *EC_2* when neither is busy (it is listed as the first destination in the corresponding *PickStation* module), server *ECS_1* has a higher utilization than that of server *ECS_2*.

The *User Specified* section summarizes tally results of car flow times through the toll plaza by car type. Again, flow times of type CR are much longer than all others, mainly due to the closure of tollbooth *CR_1* during off-peak hours.

Note that we are dealing here with a terminating simulation (recall Section 9.1.1), with a modest replication length of merely 24 hours. Consequently, we can expect considerable variability in statistics across replications. The reader is urged to run multiple replications (say, five) of this model and observe the resulting variability in statistics, such as waiting and flow times. This would illustrate the need for running a sufficient number of replications to achieve reliable statistics. More specifically, the grand means of statistics across replications should stabilize, that is, the modeler needs to ensure that adding more replications does not appreciably change the aforementioned grand means.

The dramatic impact of the closure of tollbooth *CR_1* during off-peak hours raises a natural question: To what extent would the flow times of CR cars be improved, if tollbooth *CR_1* were to remain open at all times? It is easy to modify the Arena model of the toll plaza to reflect this improved operating rule: Merely set the capacity of server resource *CRS_1* to 1 at all times.

The simulation results of the improved toll plaza model are displayed in Figure 13.46. As expected, the improved toll plaza yields dramatically lower average flow time and maximal flow time for CR cars, and higher utilization of the tollbooths. The slight variations in the statistics of EC and EZP cars and tollbooths from Figure 13.45 to Figure 13.46 are simply due to randomness, since cars of distinct types do not interact in the absence of jockeying.

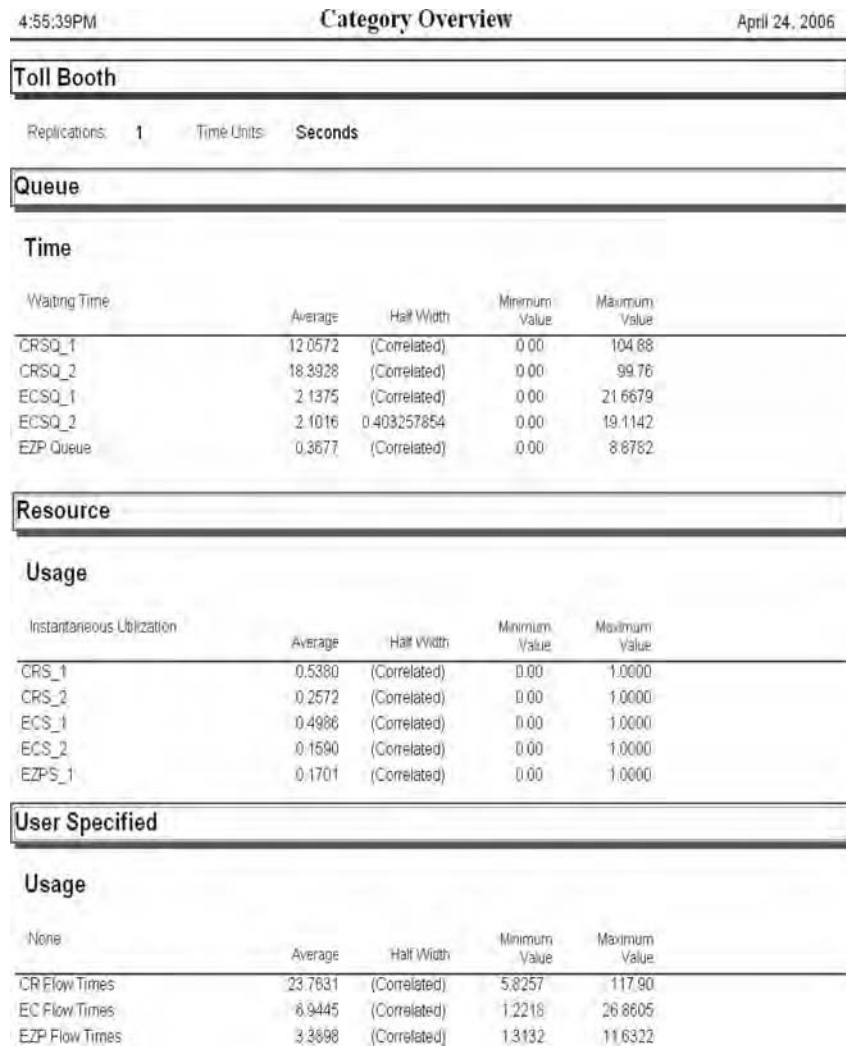
13.5 EXAMPLE: A GEAR MANUFACTURING JOB SHOP

Recall that transportation activities in manufacturing systems employ two modes of transportation: transporters to carry discrete loads (e.g., trucks, forklifts), and conveyors to carry a continuous stream of material or product (e.g., conveyor belts). The Arena facilities for these modes are analogous. Consequently, we focus in this example only on transporters moving among stations (in this case, workstations), thereby sequencing the operations in the requisite order.

Consider a job shop producing three types of gears, G1, G2, and G3, for three different types of cars. The job shop is spread out geographically on the factory floor and its layout consists of the following locations:

- An arrival dock
- A milling workstations with four milling machines
- A drilling workstations with three drilling machines
- A paint shop with two spray booths
- A polishing area with a single worker
- A shop exit

The distances among locations are given in Table 13.2.



4:55:39PM **Category Overview** April 24, 2006

Toll Booth

Replications: 1 Time Units: Seconds

Queue

Time

Waiting Time	Average	Half Width	Minimum Value	Maximum Value
CRSQ_1	12.0572	(Correlated)	0.00	104.88
CRSQ_2	18.3928	(Correlated)	0.00	99.76
ECSQ_1	2.1375	(Correlated)	0.00	21.6679
ECSQ_2	2.1016	0.403257854	0.00	19.1142
EZP Queue	0.3677	(Correlated)	0.00	8.8782

Resource

Usage

Instantaneous Utilization	Average	Half Width	Minimum Value	Maximum Value
CRS_1	0.5380	(Correlated)	0.00	1.0000
CRS_2	0.2572	(Correlated)	0.00	1.0000
ECS_1	0.4986	(Correlated)	0.00	1.0000
ECS_2	0.1590	(Correlated)	0.00	1.0000
EZPS_1	0.1701	(Correlated)	0.00	1.0000

User Specified

Usage

None	Average	Half Width	Minimum Value	Maximum Value
CR Flow Times	23.7631	(Correlated)	5.8257	117.90
EC Flow Times	6.9445	(Correlated)	1.2218	26.8605
EZP Flow Times	3.3898	(Correlated)	1.3132	11.6322

Figure 13.46 Simulation results for the improved toll plaza model.

Gear jobs arrive in batches of 10 units and their interarrival times are uniformly distributed between 400 and 600 minutes. Of arriving batches, 50% are of type G1, 30% are of type G2, and 20% are of type G3. A gear job arrives at the arrival dock and from there is dispatched to its specific (type-dependent) sequence of manufacturing operations. A sequence consists of a subset of milling, drilling, painting, and polishing operations. Table 13.3 displays the operations plan showing the sequence of operations and the associated processing times for each gear type. The layout of the job shop and operation sequences of gear types are depicted in Figure 13.47.

Gears are transported among locations by two trucks running at a constant speed of 100 feet/minute. Each truck can carry only one gear at a time. When a job is complete at a location, the gear is placed into an output buffer, a transport request is made for a truck, and the gear waits for the truck to arrive. Once a gear is transported to the next

Table 13.2

Distances among job shop locations

From Location	To Location	Distance (feet)
arrival dock	milling workstation	100
arrival dock	drilling workstation	100
milling workstation	drilling workstation	300
milling workstation	paint shop	400
milling workstation	polishing area	150
paint shop	polishing area	300
drilling workstation	paint shop	150
drilling workstation	polishing area	400
paint shop	arrival dock	250
polishing area	arrival dock	250
polishing area	shop exit	200
shop exit	arrival dock	550
shop exit	drilling workstation	500
shop exit	milling workstation	300
shop exit	paint shop	400
shop exit	polishing area	200

Table 13.3

Operations plan for gears by type

Gear Type	Operations Sequence	Processing Time (minutes)
G1	milling	35
	drilling	20
	painting	55
	polishing	15
G2	milling	25
	painting	35
	polishing	15
G3	drilling	18
	painting	35
	polishing	15

location, it is placed in a FIFO input buffer. Finally, when the polishing operation is completed, the finished gear departs from the job shop via the shop exit. Similar models of job shops can be found in Schriber (1990) and Kelton et al. (2004).

To analyze the performance of the job shop, we plan to run a simulation over 1 year of operation. The following statistics are of interest:

- Gear flow times (by type)
- Gear delays at operations locations
- Machine utilizations

Figure 13.48 depicts an Arena model for the gear shop, consisting of three main segments: gear batch arrivals, gear transportation, and gear processing. A segment-by-segment walkthrough of the model follows next.

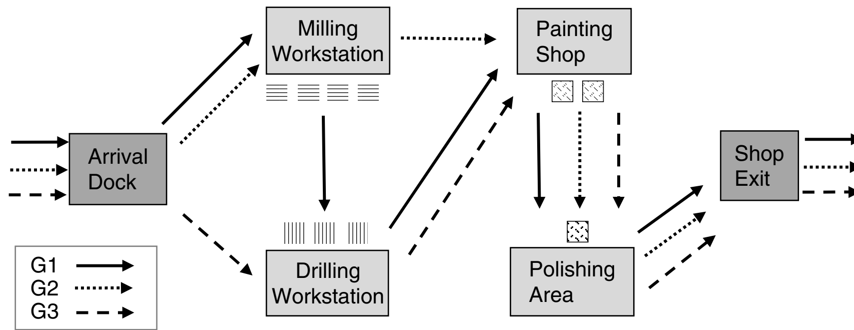


Figure 13.47 Layout of job shop and operation sequences by gear type.

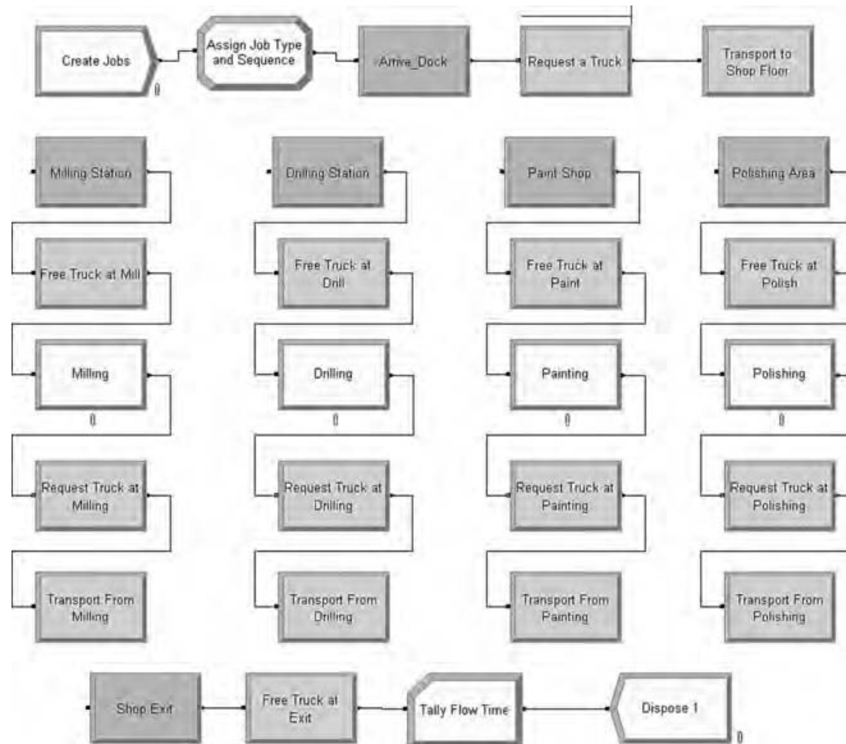


Figure 13.48 Arena model for the gear manufacturing job shop.

13.5.1 Gear Job Arrivals

Gear entities are created in the *Create* module, called *Create Jobs*, whose dialog box is displayed in Figure 13.49. The *Entities per Arrival* field indicates that gear jobs arrive in batches of 10, and the *Time Between Arrivals* section specifies batch interarrival times to be uniformly distributed between 400 and 600 minutes. Following arrival, each incoming gear entity proceeds as a separate entity.

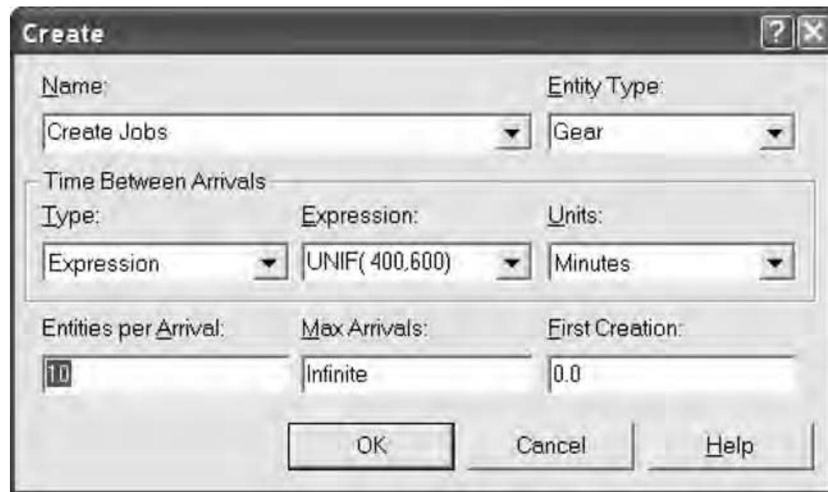


Figure 13.49 Dialog box of the *Create* module *Create Jobs*.

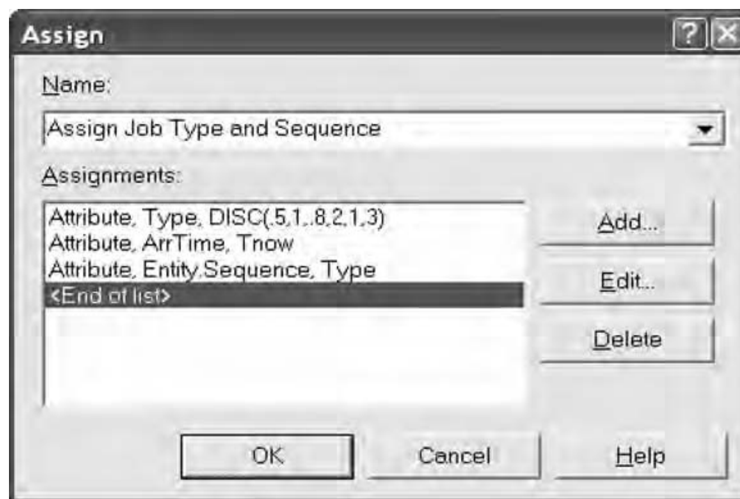
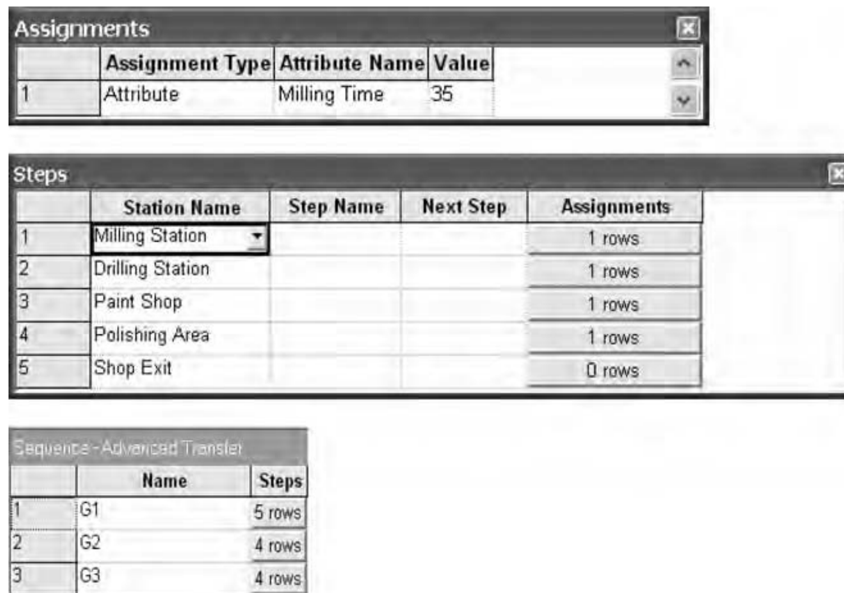


Figure 13.50 Dialog box of the *Assign* module *Assign Job Type and Sequence*.

An arriving gear entity next enters the *Assign* module, called *Assign Job Type and Sequence*, whose dialog box is displayed in Figure 13.50. Here, a gear entity is assigned a type by sampling it from a discrete distribution, and saving the type code (1, 2, or 3) in its *Type* attribute. In addition, the *ArrTime* attribute is assigned the value of the simulation clock, *Tnow*, for later use in computing the gear entity's flow time. Finally, the Arena attribute *Entity.Sequence* is assigned the value of the *Type* attribute. This attribute acts as an index that associates a gear type with the corresponding operations sequence.

The operations sequences for gear types are specified in the *Sequence* module from the *Advanced Transfer* template panel, whose dialog spreadsheet is displayed at the bottom of Figure 13.51. Three sequences (row entries) are defined here, one for each



Assignments

	Assignment Type	Attribute Name	Value
1	Attribute	Milling Time	35

Steps

	Station Name	Step Name	Next Step	Assignments
1	Milling Station			1 rows
2	Drilling Station			1 rows
3	Paint Shop			1 rows
4	Polishing Area			1 rows
5	Shop Exit			0 rows

Sequence - Advanced Transfer

	Name	Steps
1	G1	5 rows
2	G2	4 rows
3	G3	4 rows

Figure 13.51 Dialog spreadsheet of the *Sequence* module (bottom), the *Steps* dialog spreadsheet for specifying operations steps of type G1 gears (middle), and the *Steps Assignment* spreadsheet for specifying milling time assignment of type G1 gears (top).

gear type. Each sequence consists of a sequence name (*Name* column) and a series of steps (*Steps* column), listed in the order of processing.

To specify steps, the modeler clicks the button under the *Steps* column and pops up the *Steps* dialog spreadsheet. The five steps of type G1 gears processing are displayed in the middle spreadsheet of Figure 13.51. Each step is a row entry specifying the location name and associated values (under the *Assignments* column). Clicking the corresponding button pops up the associated *Assignments* dialog spreadsheet. The assignment of time for the milling time operation is exemplified in the top spreadsheet of Figure 13.51. Arena's internal sequencing mechanism is explained in the next section.

13.5.2 Gear Transportation

Job shop locations are modeled as *Station* modules. Accordingly, every gear entity proceeds to the *Station* module, called *Arrive_Dock*, to model its physical arrival at the job shop's arrival dock. From here, gear entities will be transported to the job shop floor to start the first step in their operations sequence.

To this end, a gear entity enters the *Request* module (from the *Advanced Transfer* template panel), called *Request a Truck*, whose dialog box is displayed in Figure 13.52. The *Transporter Name* field indicates a request for a *Fork Truck* transporter. If multiple transporters are available, the modeler can specify how to select one in the *Selection Rule* field. Such selections may be cyclical, random, preferred order (as listed in the *Transporter* module), smallest distance, largest distance, or a specific transporter. Here the selection rule requests the transporter nearest to the arrival dock. Furthermore, the *Save Attribute* field specifies that the ID of the selected transporter be saved in the *Truck_ID* attribute of the requesting gear entity. The saved ID will be used in due time

Figure 13.52 Dialog box of the *Request* module *Request a Truck*.

to free that particular truck. Since requesting transporters from multiple locations is a form of contention for resources, the *Priority* field allows the modeler to assign a priority to requests issued at multiple *Request* modules (here a high priority is assigned in order to clear the arrival dock as soon as possible). The *Entity Location* field indicates the location of the requesting entity, and the *Velocity* field specifies the transporter's velocity, which is 100 feet/minute in our case. Finally, gear entities requesting transportation at the same *Request* module are instructed in the *Queue Name* field to wait in the queue, called a *Truck.Queue*, until a transporter becomes available.

As soon as a gear entity grabs a truck, it proceeds to the *Transport* module, called *Transport to Shop Floor*, whose dialog box is displayed in Figure 13.53. The *Transporter Name* and *Unit Number* fields specify the type and ID of the selected transporter, which here is the truck whose ID is kept in the *Truck_ID* attribute of the requesting gear entity. The transporter/gear destination is specified in the *Entity Destination Type* field as the *By Sequence* option, indicating that the destination is determined by the gear entity's sequence number. This field may also specify a *Station* module name, using the *Station* option. It can also specify an attribute or expression. The gear entity and the transporter move as a grouped entity at a velocity of 100 feet/minute as specified in the *Velocity* field. Note that the velocity may depend on trip type, so that an empty truck and a loaded one can be made to move at different velocities.

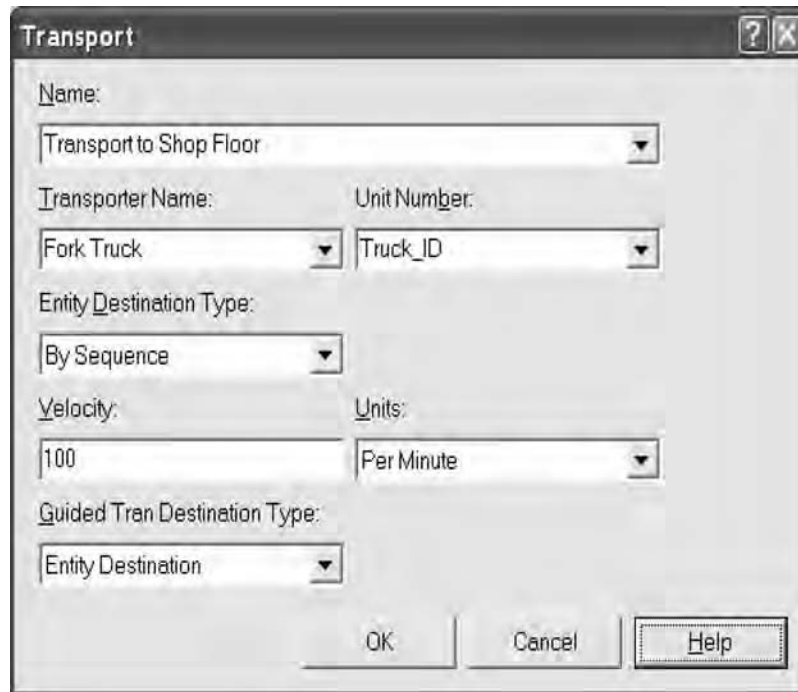


Figure 13.53 Dialog box of the *Transport* module *Transport to Shop Floor*.

13.5.3 Gear Processing

The gear processing segment encompasses sets of *Station* modules, each modeling an operation in the sequence, from milling to polishing. Since all sets have the same structure (except for names), we will only walk through the milling operation logic.

When a gear entity is transported to the milling operation, it enters the *Station* module, called *Milling Station*. It then proceeds to the *Free* module, called *Free Truck at Mill*, whose dialog box is displayed in Figure 13.54. Here, the *Transporter Name* and *Unit Number* fields specify the truck to be freed for use by other gear entities, using the *Truck_ID* attribute of the freeing gear entity.

The association between the transporter and the transported entity is severed: The freed transporter stays at the destination *Station* module until requested, while the gear entity moves on to the next module. In this case, it enters the *Process* module, called *Milling*, whose dialog box is displayed in Figure 13.55. The *Seize Delay Release* option in the *Action* field is used to model gear delays at this process. The resource seized is *Milling Machine* and the processing time is kept in the *Milling Time* attribute specified in the *Sequence* module of Figure 13.51. Furthermore, to model four milling machines at the milling workstation, resource *Milling Machine* has to be declared as having a capacity of four in the spreadsheet view of the *Resource* module. The capacity of other machine groups is similarly declared.

On completing the milling operation, the gear entity proceeds to the *Request* module, called *Request Truck at Milling*, whose dialog box is displayed in Figure 13.56. In this module, the gear entity requests transportation to the next operation, similarly to the

Figure 13.54 Dialog box of the *Free* module *Free Truck at Mill*.

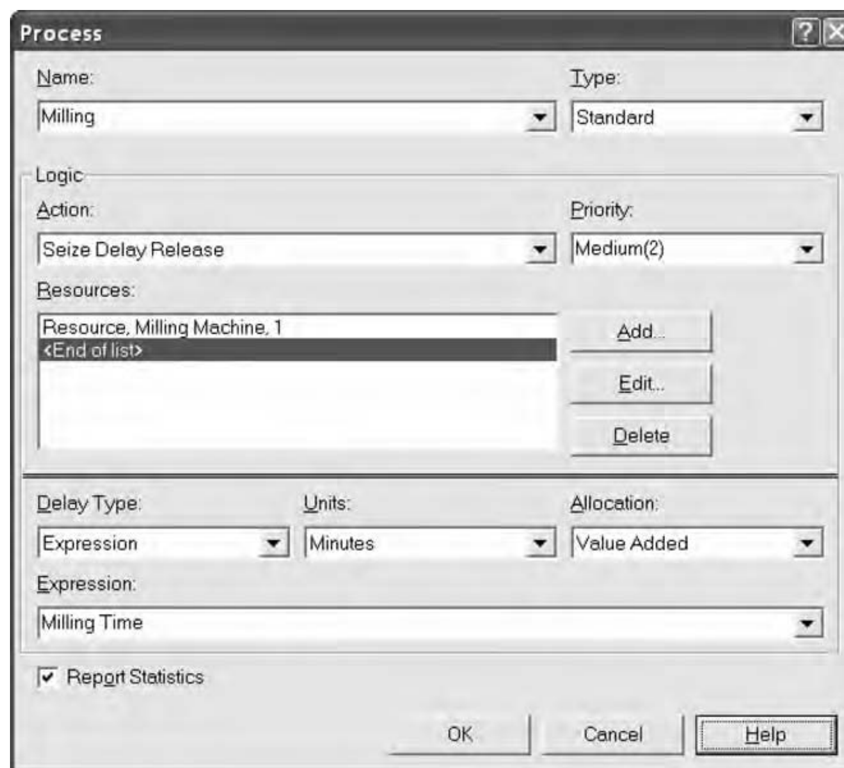


Figure 13.55 Dialog box of the *Process* module *Milling*.

first request from the arrival dock to the job shop floor (Figure 13.52). Here, it may have to wait in the queue, called *Request Truck at Milling.Queue*, which serves as the output buffer for the milling process. The transport operation takes place when a truck arrives and both gear and transporter enter the *Transport* module called *Transport From Milling*.

Gear entities move from one operation to another according to their specified sequences. It should be pointed out that Arena handles all sequencing details at run-time. The internal Arena attribute *IS* keeps track of each gear entity's step number in its

Figure 13.56 Dialog box of the *Request* module *Request Truck at Milling*.

sequence. Whenever a sequential transport is requested, Arena increments the *IS* attribute and indexes into the appropriate *Steps* module spreadsheet to determine the destination location and travel time. The *IS* attribute may also be modified by the modeler. Eventually the gear entity arrives at the *Station* module, called *Shop Exit*, which is always the last location in each operations sequence.

Next, the transporting truck is freed in the *Free* module, called *Free Truck at Exit*, and the finished gear entity is ready to record some statistics and then depart from the model at a *Dispose* module.

Figure 13.57 displays the dialog box of the *Record* module, called *Tally Flow Time*. Here, flow times are tallied with the aid of the *ArrTime* attribute of each finished gear entity. Note that these flow times are tallied by gear type, using the tally set mechanism. The *Tally Set Name* field indicates that tallies are to be entered in the *Flow Times* set. Each gear entity indexes into this set using its *Type* attribute, specified in the *Set Index* field. The *Flow Times* set is specified in the *Set* module spreadsheet from the *Basic Process* template panel. Figure 13.58 displays the *Set* spreadsheet and the members of the *Flow Times* set.

Arena computes travel times of transporters among *Station* modules based on their distances and transporter speeds. Figure 13.59 introduces *Fork Truck* transporters into the model and specifies their parameters in the *Transporter* module spreadsheet. These

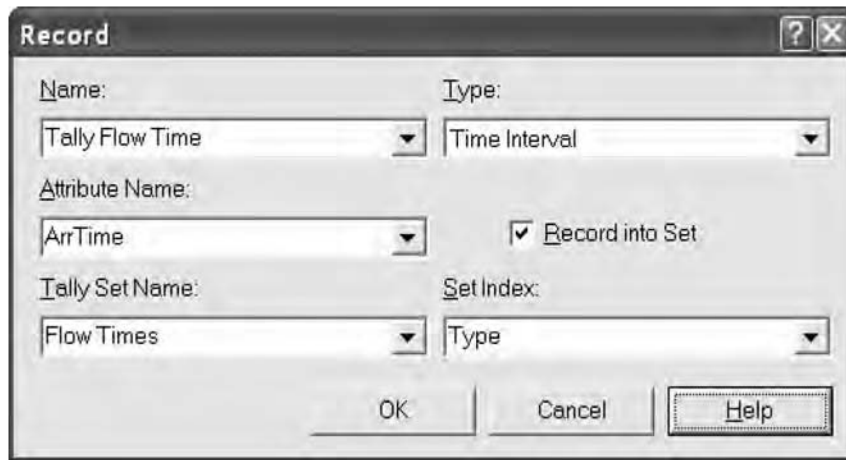


Figure 13.57 Dialog box of the *Record* module *Tally Flow Time*.

Members

	Tally Name
1	G1 Flow Time
2	G2 Flow Time
3	G3 Flow Time

Set - Basic Process

	Name	Type	Members
1	Flow Times	Tally	3 rows

Figure 13.58 Dialog spreadsheet of the *Set* module (bottom) and the *Members* dialog spreadsheet of the *Flow Times* set (top).

Initial Position Status

	Initial Position	Station Name	Initial Status
1	Station	Arrive_Dock	Active
2	Station	Arrive_Dock	Active

Transporter - Advanced Transfer

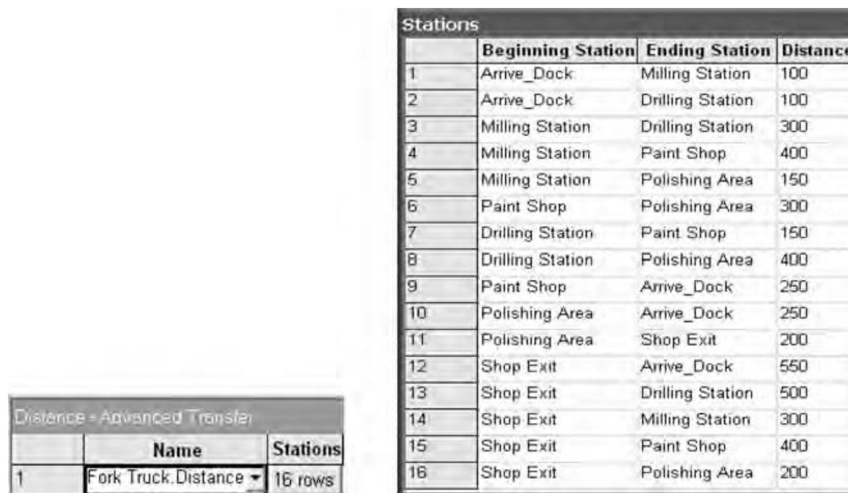
	Name	Number of Units	Type	Distance Set	Velocity	Units	Initial Position Status	Report Statistics
1	Fork Truck	2	Free Path	Fork Truck Distance	100	Per Minute	2 rows	☑

Figure 13.59 Dialog spreadsheet of the *Transporter* module (bottom) and the *Initial Position Status* dialog spreadsheet (top).

include columns for a *Name* field to specify the transporter set, a capacity field (*Number of Units*), a *Distance Set* field for specifying the name of a *Distance* module allowing the user to specify distances between pairs of *Station* modules, a *Velocity* and *Units* fields that specify the transporter speed (in our case, in feet per minute), and an *Initial Position Status* column of buttons, which pop up the *Initial Position Status* dialog spreadsheet (top of Figure 13.59). The latter is used to specify the location at which a transporter resides initially (at simulation time 0). In particular, the top dialog box in Figure 13.59 indicates that all *Fork Truck* transporters reside initially in the *Station* module called *Arrive_Dock*. Note that the transporter speed is the default speed. Arena allows the modeler to override this value and to further distinguish between the speed of an empty transporter (specified in a *Request* module) and a loaded transporter (specified in a *Transport* module).

Next, Figure 13.60 displays the dialog spreadsheet of the *Distance* module (left), as well as a corresponding *Stations* dialog spreadsheet (right), which pops up on clicking a button under the former's *Stations* column.

Finally, the collection of fork truck utilization (a *Time-Persistent* statistic) and flow-time statistics (*Tally* statistics) is specified in the *Statistic* spreadsheet module, as shown in Figure 13.61. Observe that the Arena variable *nt(transporter_name)* is used to collect transporter utilization, in our case, *nt(Fork Truck)*.



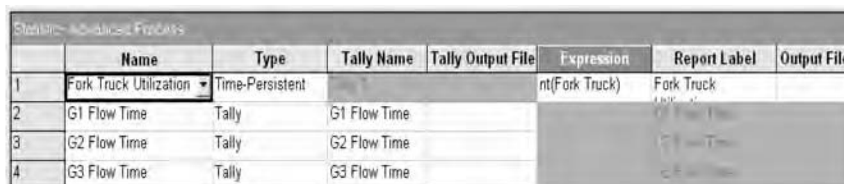
Distance - Advanced Transfer

	Name	Stations
1	Fork Truck.Distance	16 rows

Stations

	Beginning Station	Ending Station	Distance
1	Arrive_Dock	Milling Station	100
2	Arrive_Dock	Drilling Station	100
3	Milling Station	Drilling Station	300
4	Milling Station	Paint Shop	400
5	Milling Station	Polishing Area	150
6	Paint Shop	Polishing Area	300
7	Drilling Station	Paint Shop	150
8	Drilling Station	Polishing Area	400
9	Paint Shop	Arrive_Dock	250
10	Polishing Area	Arrive_Dock	250
11	Polishing Area	Shop Exit	200
12	Shop Exit	Arrive_Dock	550
13	Shop Exit	Drilling Station	500
14	Shop Exit	Milling Station	300
15	Shop Exit	Paint Shop	400
16	Shop Exit	Polishing Area	200

Figure 13.60 Dialog spreadsheet of the *Distance* module (left) and the *Stations* dialog spreadsheet (right).



Statistic - Advanced Process

	Name	Type	Tally Name	Tally Output File	Expression	Report Label	Output File
1	Fork Truck Utilization	Time-Persistent			nt(Fork Truck)	Fork Truck	
2	G1 Flow Time	Tally	G1 Flow Time				
3	G2 Flow Time	Tally	G2 Flow Time				
4	G3 Flow Time	Tally	G3 Flow Time				

Figure 13.61 Dialog spreadsheet of the *Statistic* module for collecting fork truck utilization and flow-time tallies.

13.5.4 Simulation Results for the Gear Manufacturing Job Shop Model

The job shop model was simulated for 1 year and the resulting output report is displayed in Figure 13.62. The *Time per Entity* section lists the statistics of gear waiting times for each operation. As expected, the average waiting time at the paint shop is very large as compared to the other operation locations, since spray times are quite long. The *Usage* section displays resource utilizations at individual operation locations. For

Job Shop

Replications: 1 Time Units: Minutes

Process

Time per Entity

Wait Time Per Entity	Average	Half Width	Minimum Value	Maximum Value
Drilling	0.6656	0.058478552	0.00	16.0000
Milling	14.6109	0.283086009	0.00	62.0000
Painting	44.8111	0.502319726	0.00	156.50
Polishing	3.0743	0.087881937	0.00	15.0000

Resource

Usage

Instantaneous Utilization	Average	Half Width	Minimum Value	Maximum Value
Drilling Machine	0.08984399	0.001432065	0.00	1.0000
Milling Machine	0.1248	0.001909548	0.00	1.0000
Paint Booth	0.4480	0.004240685	0.00	1.0000
Polishing worker	0.2988	0.002281493	0.00	1.0000

User Specified

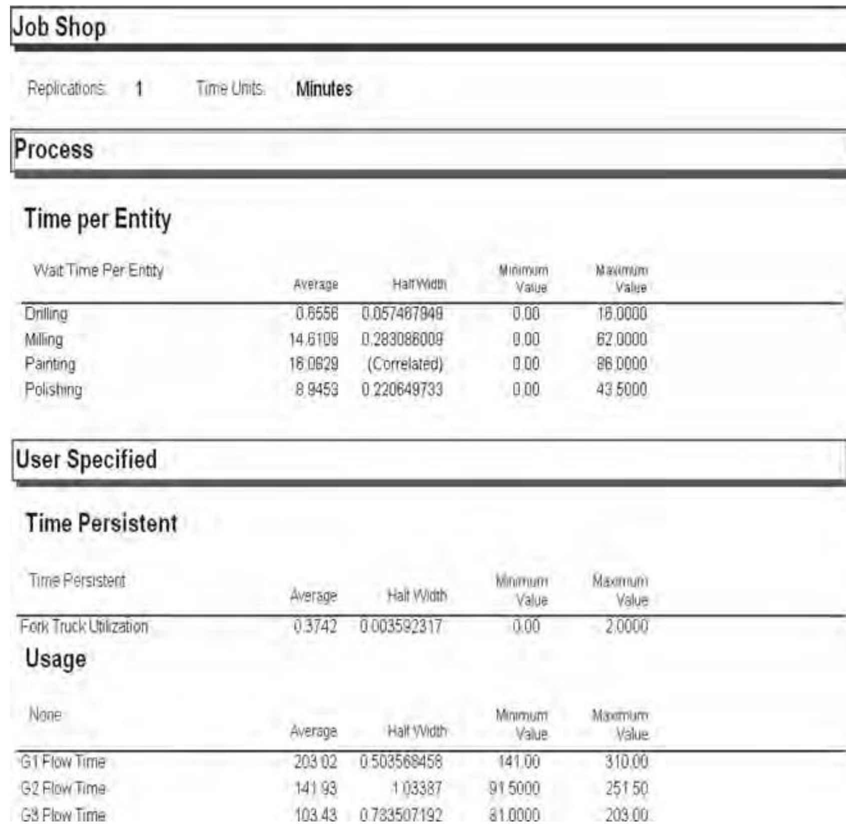
Time Persistent

Time Persistent	Average	Half Width	Minimum Value	Maximum Value
Fork Truck Utilization	0.3763	0.003317102	0.00	2.0000

Usage

None	Average	Half Width	Minimum Value	Maximum Value
G1 Flow Time	240.46	0.542674482	141.00	380.00
G2 Flow Time	153.20	1.50766	91.5000	326.00
G3 Flow Time	105.16	0.972138539	81.0000	224.00

Figure 13.62 Simulation results for the gear manufacturing job shop model.



Job Shop

Replications: 1 Time Units: Minutes

Process

Time per Entity

Wait Time Per Entity	Average	Half Width	Minimum Value	Maximum Value
Drilling	0.6556	0.057467949	0.00	16.0000
Milling	14.6109	0.283086009	0.00	62.0000
Painting	16.0629	(Correlated)	0.00	86.0000
Polishing	8.9453	0.220649733	0.00	43.5000

User Specified

Time Persistent

Time Persistent	Average	Half Width	Minimum Value	Maximum Value
Fork Truck Utilization	0.3742	0.003592317	0.00	2.0000

Usage

None	Average	Half Width	Minimum Value	Maximum Value
G1 Flow Time	203.02	0.503568458	141.00	310.00
G2 Flow Time	141.93	1.03387	91.5000	251.50
G3 Flow Time	103.43	0.733507192	81.0000	203.00

Figure 13.63 Simulation results for the modified gear manufacturing job shop model.

instance, the utilization of a drilling machine is about 0.0898. The *User Specified* section displays fork truck utilization statistics and flow-time statistics by gear type. The fact that the average flow times are much larger than the total processing time bears witness to excessive delays in resource queues.

To reduce the wait at the paint shop, we modify the job shop model by increasing the number of paint booths from two to three. The impact of this modification on gear delays and flow times is indicated in the simulation results of Figure 13.63. Clearly, the addition of a paint booth has significantly reduced the delay at the paint shop but slightly increased the delay at the polishing area, because speeding up an operation increases congestion downstream. The overall effect on gear flow times, however, is a slight reduction.

13.6 EXAMPLE: SETS VERSION OF THE GEAR MANUFACTURING JOB SHOP MODEL

This section exemplifies the use of Arena sets via a more advanced version of the previous job shop example, by taking advantage of the repetitive structure of the model to reduce its size (number of modules). To wit, note that the model logic for each

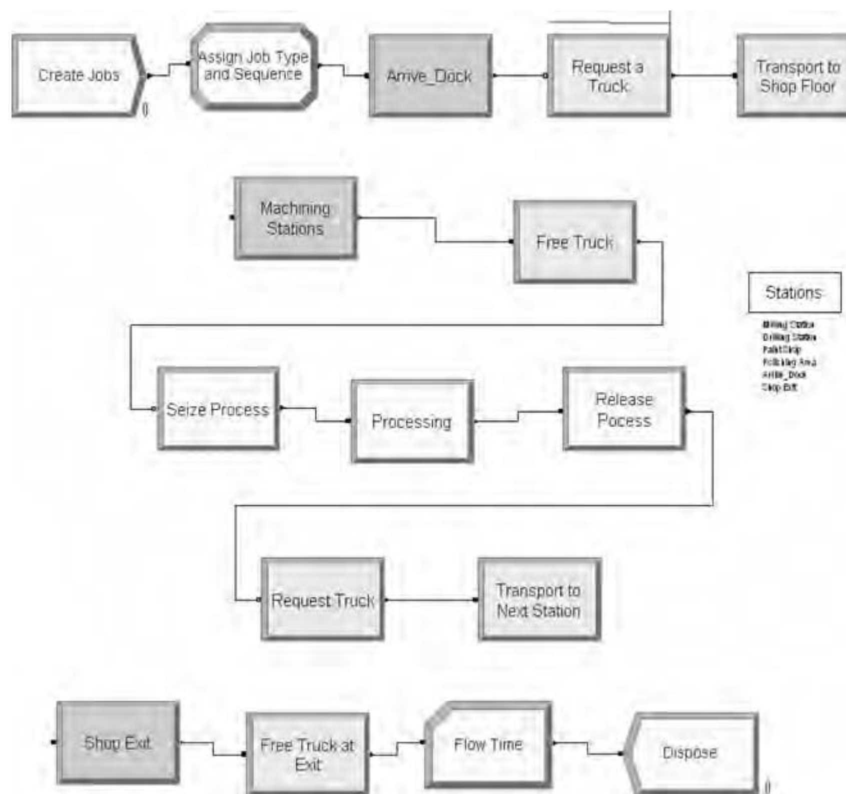


Figure 13.64 Arena model with sets for the gear manufacturing job shop example.

individual operation in Figure 13.48 is essentially the same. Consequently, it is possible in this model to create sets of queues and resources, effectively vectorizing the entire model. The resultant (smaller) vectorized model is depicted in Figure 13.64.

The logic until a job is transferred to the first operation location (first row of modules in Figure 13.64) and the logic of shop exit (last row of modules in Figure 13.64) are not part of the repetitive pattern, and therefore they carry over unchanged from Figure 13.48. However, the middle set of modules in Figure 13.64 has been redefined to incorporate sets, and is considerably smaller in size than its counterpart in Figure 13.48.

Observe that the logic of each operation location in Figure 13.48 consists of freeing a truck, processing, requesting a truck, and transporting to the next operation location. The middle part in Figure 13.64 captures the logic of all operation locations using the concepts of *set* and *sequence*. In effect, a job traverses (iterates) the middle set of modules multiple times according to its operations sequence. In each iteration, the middle set of blocks plays the role of the current operation location in the sequence, as explained later.

The first module in each iteration is the *Station* module, called *Machining Stations*, whose dialog box is displayed in Figure 13.65. The dialog box specifies a station set, called *Process Stations*, whose members consist of all operation locations (*Milling*

Figure 13.65 Dialog box of the *Station* module *Machining Stations*.

Station, *Drilling Station*, *Paint Shop*, and *Polishing Area*). Recall that as soon as a gear entity enters the *Transport* module, called *Transport to Shop Floor*, the gear entity is assigned the ID of the destination operation location, according to its operations sequence as specified in the *Sequence* module spreadsheet (see Figure 13.51). On arrival at the *Machining Stations* module, the destination ID is saved in the incoming gear entity's attribute *Entity.Station* for use in the current iteration.

Next, the gear entity proceeds to free its truck in module *Free Truck*, and then enters the *Seize* module, called *Seize Process*. Figure 13.66 displays the dialog box of this module (bottom) and the details of the machine resource to be seized (top). The *Seize Process* module plays the role of the *Seize* module of each workstation resource (one particular resource per iteration). On entry of a gear entity into this module, it selects a specific member of the resource set called *Machines*, based on its *Entity.Station* attribute. The *Machines* set consists of the resources *Milling Machine*, *Drilling Machine*, *Paint Booth*, and *Polishing Worker*. Note that the incoming job entity is queued in a corresponding member of the queue set called *Machine Queues*, again based on its *Entity.Station* attribute. The actual operation takes place in the *Delay* module, called *Processing*, whose dialog box is displayed in Figure 13.67.

Recall that in the previous version (without sets), individual operation times were assigned in the *Sequence* module spreadsheet and associated dialog boxes of Figure 13.51, and stored in separate gear entity's attributes (*Milling Time*, *Drilling Time*, *Spray Time*, and *Polishing Time*). In the current version (with sets), however, all these attributes are replaced by a single attribute, called *Processing Time*, of the incoming gear entity that is assigned the processing time of the current operation.

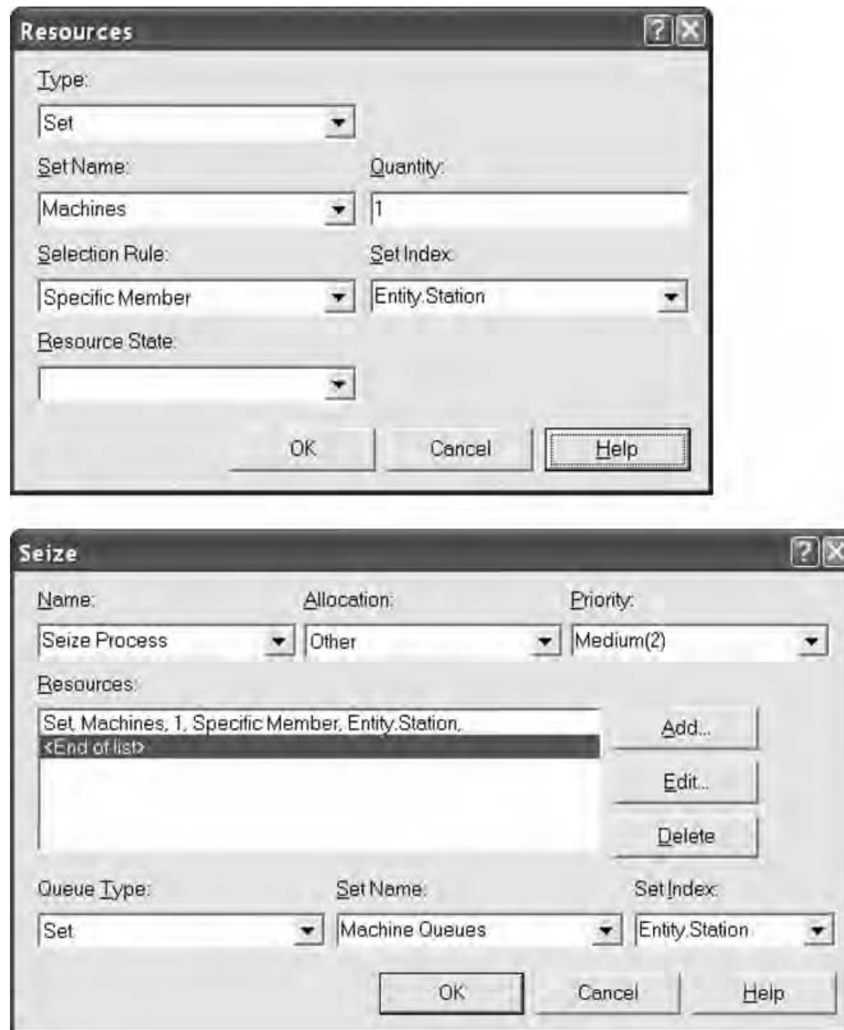


Figure 13.66 Dialog box of the *Seize* module *Seize Process* (bottom) and the *Resources* dialog box of the seized machine resource (top).

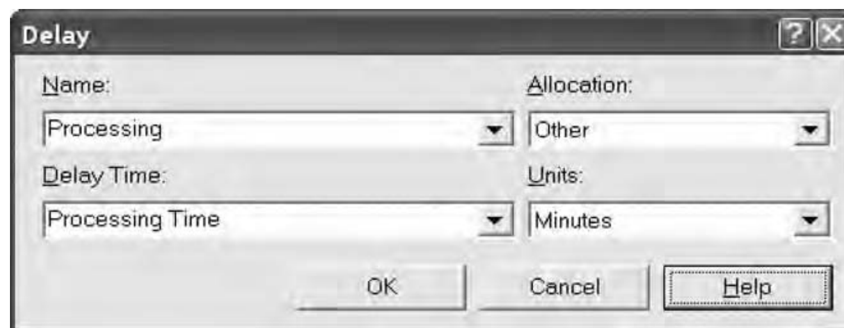


Figure 13.67 Dialog box of the *Delay* module *Processing*.

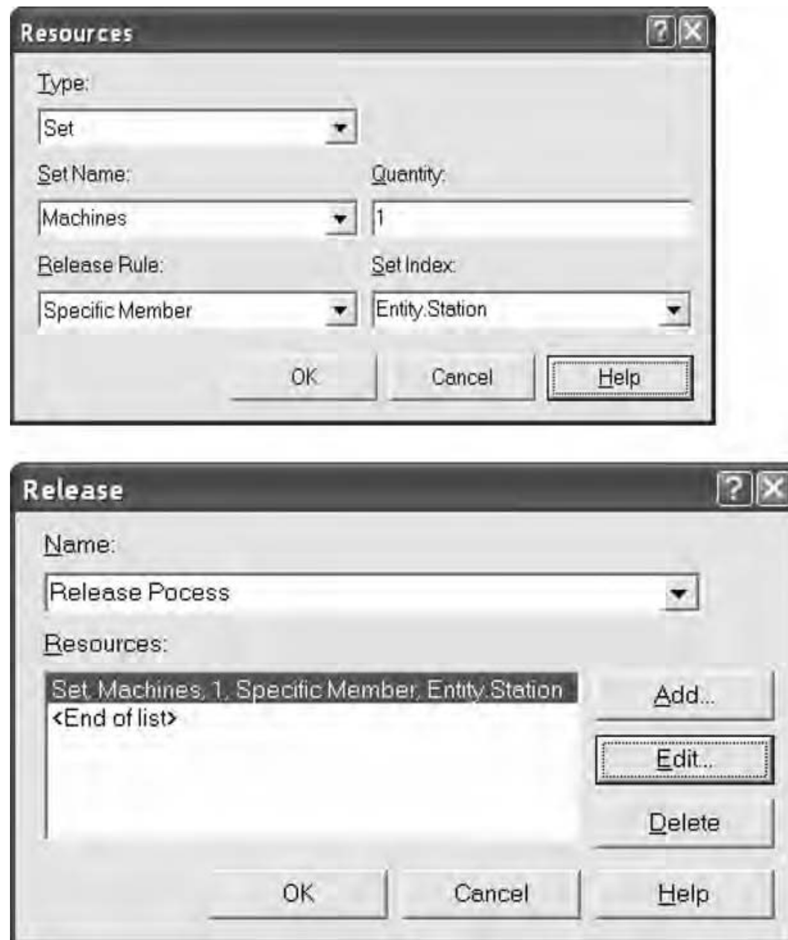


Figure 13.68 Dialog box of the *Release* module *Release Process* (bottom) and the *Resources* dialog box of the released machine resource (top).

Once the current operation is completed, the gear entity proceeds to the module, called *Release Process*, where it releases the specific resource it was holding. Figure 13.68 displayed the dialog box of this module and the released resource details.

The gear entity next proceeds to the *Request* module, called *Request Truck*, to be transported to the next destination in its operations sequence via the *Transport* module, called *Transport to Next Station*, precisely as in the previous model (see Figures 13.52 and 13.53). Finally, when all operations are completed, the gear entity proceeds to the *Station* module, called *Shop Exit*, as was done in the previous model.

The current model with sets adds three main elements to the previous model. First, the *Stations* element from the *Elements* template panel is used to enforce an explicit numbering of stations (operation locations), as shown in its dialog box depicted in Figure 13.69. Using the *Stations Element* dialog box to ensure a preferred numbering is recommended. This in turn ensures the correct referencing of set members (e.g., queues and resources) in model logic (e.g., the aforementioned iteration in the middle modules of Figure 13.64).

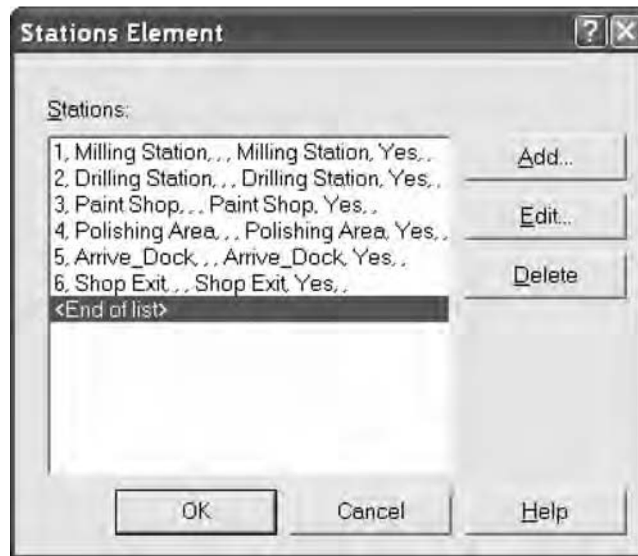


Figure 13.69 The *Stations Element* dialog box.

Members

	Resource Name
1	Milling Machine
2	Drilling Machine
3	Paint Booth
4	Polishing Worker

Set - Basic Process

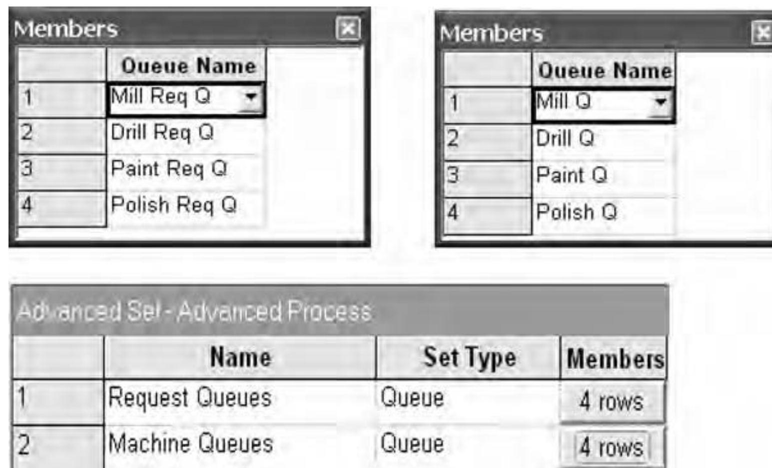
	Name	Type	Members
1	Machines	Resource	4 rows
2	Flow Times	Tally	3 rows

Figure 13.70 Dialog spreadsheet of the *Set* module (bottom) and the *Members* dialog spreadsheet for the *Machines* set (top).

Second, the *Set* module spreadsheet from the *Basic Process* template panel now includes the *Machines* resource set in addition to the *Flow Times* set. Figure 13.70 displays this spreadsheet (bottom) and the members of the *Machines* resource set (top).

Lastly, the current model also includes a new *Advanced Set* module spreadsheet from the *Advanced Process* template. Figure 13.71 displays this spreadsheet and the members of its sets.

Of course, running the gear manufacturing job shop model with sets produces identical results to those of its counterpart without sets. However, models with sets are more parsimonious in that they reduce the number of modules as evidenced by



Members

	Queue Name
1	Mill Req Q
2	Drill Req Q
3	Paint Req Q
4	Polish Req Q

Members

	Queue Name
1	Mill Q
2	Drill Q
3	Paint Q
4	Polish Q

Advanced Set - Advanced Process

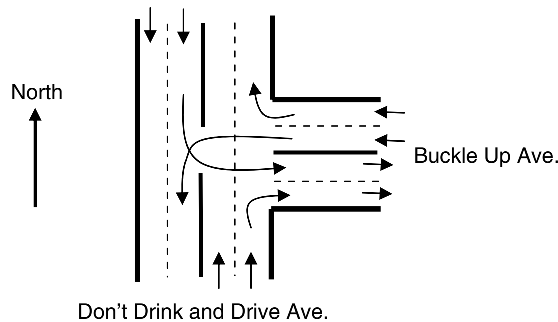
	Name	Set Type	Members
1	Request Queues	Queue	4 rows
2	Machine Queues	Queue	4 rows

Figure 13.71 Dialog spreadsheet of the *Advanced Set* module (bottom) and the *Members* dialog spreadsheet (top).

Figure 13.48 versus Figure 13.64. Finally, using sets renders models more intelligible and easier to modify.

EXERCISES

1. *Traffic light at road intersection*. Montgomery Township is considering installing a traffic light at the three-way intersection of *Don't Drink and Drive Ave*. and *Buckle Up Ave*., where accidents occur frequently. Assume that each direction of every avenue consists of 2 lanes. The layout of the intersection is shown in the next illustration.



Vehicles approach the intersection with the following iid interarrival time distributions and destinations:

- From the north: Interarrival times are iid Unif(2, 14) seconds. Twenty-five percent of the cars turn east, and 75% proceed south.
- From the south: Interarrival times are iid Unif(2, 15) seconds. Forty percent of the cars turn east, and 60% proceed north.
- From the east: Interarrival times are iid Unif(12, 18) seconds. Fifty percent of the cars turn north, and 50% turn south.

It takes 5 seconds to go through the intersection in any direction (provided the light is green, of course). The traffic light works in cycles in the following manner:

- First, the light for the north–south direction turns green, and the following scenario unfolds. During the first 45 seconds, non-turning traffic from the north and from the south can proceed, as well as right turns from the east and from the south. However, left turns from either the north or from the east are not allowed. During the next 25 seconds, the light remains green for southbound cars but becomes red for north-bound cars. During this time, left turns from the north are allowed, as well as right turns from the south and the east.
- Second, the light turns red in the north–south direction and then turns green in the easterly direction for 25 seconds. During this period, all traffic from the east and right turns from the south are allowed to proceed. However, note that left turns from the north are not allowed to proceed.

  a. Assuming time-homogenous (stationary) arrival rates (i.e., rates do not change over time), simulate the light-controlled behavior of the intersection for 5 days.
  b. For each car stream (lane), what is the average waiting time at the intersection?
  c. Keeping in mind that the *Schedule* module cannot be used with uniform interarrival times, describe how to model a system with a 20% increase in the arrival rates of all cars during the rush hours of 7 A.M.–9 A.M. in the morning and 5 P.M.–7 P.M. in the evening. Note that the Arena student version may not accommodate the modified system.

  Hint: You may model the passage of cars through the intersection using a *Process* module for each stream. The traffic light may be modeled as a single entity seizing these *Process* modules cyclically to stop the traffic.

2. *Bulk port operations*. Port Lasfar is an iron ore exporting port that operates 24 hours a day and 365 days a year. The annual export plan calls for nominal deterministic ship arrivals at the rate of one ship every 20 hours. However, ships usually do not arrive on time due to weather conditions, rough seas, or other reasons, and consequently, each ship is given a 7-day lay period (see Section 13.3). Assume that ships arrive iid uniformly in their lay periods. Ships vary in tonnage according to the following distribution:

$$Tonnage = \begin{cases} 45,000 & \text{with probability} \quad 0.30 \\ 57,000 & \text{with probability} \quad 0.50 \\ 65,000 & \text{with probability} \quad 0.20. \end{cases}$$

Arriving ships queue up FIFO (if necessary) at an offshore anchorage location, from where they are towed into port by a single tug boat as soon as the berth becomes available. The tug boat is stationed at a tug station located at a distance of 30 minutes away from the offshore anchorage. Travel between the offshore anchorage and the berthing area takes 1 hour.

There are two berths and two loaders at the port. An uninterrupted iron ore supply arrives at the port from a nearby mine. Whenever there is only one ship at the berths (only one berth is busy), both loaders load the ship together in half the time. However, as soon as another ship arrives at the empty berth, one of the loaders starts working on the new ship. In this case, the current ship continues its loading with only one loader. The loaders are identical and their rate of loading is

2000 tons per hour. Hence, the loading rate becomes 4000 tons per hour when both loaders work on the same ship. Assume no time is involved in shifting loaders. Once a ship is loaded at a berth, it relinquishes that berth (and the loader), and then requests tug boat service and waits until the tug boat arrives. Since there is only one tug boat, it does not matter whether it waits at the berth and then relinquishes the berth after the tug boat arrives or it releases it first and waits in a different area. The tug boat tows it away to the offshore anchorage, from where the boat departs with its iron ore for its destination. Higher priority is given to departing vessels in seizing the tug boat. Port Lasfar does not experience tidal activity or storms or any other stoppages.

a. Develop an Arena model for Port Lasfar, and simulate it for 1 year (8760 hours).
b. Estimate the following statistics:
   - Berth and ship loader utilization
   - Average ship port time
   - Average number of ships at the anchorage
   - Average ship delay at the anchorage

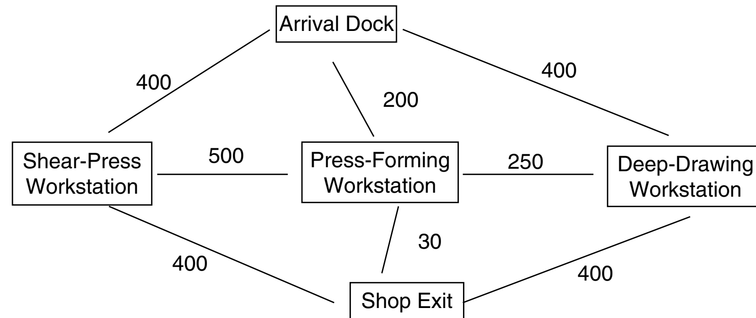
Hint: Since a loader may have to be separated from a ship in the middle of loading, you may approximate the loading time as consisting of increments of, say, 10 minutes, and load as much as you can in a 10-minute period and then check the status of the other berth. If you need to separate a loader, do it immediately; otherwise, continue for another 10 minutes of loading. Loading will continue until the entire ship is loaded. During this time, the model may experience periods with one loader working on the ship, and other periods with two loaders working on the ship.

3. *Job shop operations*. The production of various metal kitchen pots and utensils calls for sheet metal processing. KitchenWare Inc. (KWI) is a company that produces a number of metal kitchen products. KWI's shop floor has three major sheet metal processing areas. These are *shearing, press brake forming*, and *deep drawing*. Shearing is a process where roughly sized sheet metal pieces are cut from rolls of sheet metal. The company uses various sizes of sheet metal rolls. Press brake forming is a type of bending and forming operation. Also, deep drawing is an operation that makes a suitably deep sheet metal piece to make pots, pans, and food and beverage containers. The shop floor houses one shear press machine, two press forming machines, and one deep-drawing machine.

   KWI's products are categorized as pots or pitchers. These two categories of product arrive in equal proportions in a combined stream with uniform interarrival times between 15 and 25 seconds. Product types have individual process plans and processing times as follows:

Product Category	Process Plan	Unit Processing Times (seconds)
pots	shear press	10
	press forming	18
	deep drawing	30
pitchers	shear press	12
	press forming	24

The distances between locations (in feet) are given in the following diagram (distances are the same in each direction).



Part transportation among workstations is handled by three fork trucks, which run at a speed of 50 feet/second. Each truck can carry a batch of five items. When a job is completed at the shear-press or press-forming workstations, it is placed into an output buffer. As soon as five units of the same type accumulate in the output buffer, they are batched, and a request is made for a fork truck. Once a batch is taken to the press-forming workstation, it is split into its original items, and each item is placed into an input buffer. In contrast, in the deep-drawing workstation, batches are not split into individual units, but rather are processed in batches of five units at a time. After their last operation (press forming or deep drawing), batches of five items are taken to the shop exit from where they leave the shop floor in batches.

The press-forming process is subject to random failures at any point in time, with Tria(400, 600, 1200) seconds of uptime and Unif(1, 2) minutes of downtime.

a. Develop an Arena model of the shop floor of KWI and simulate it for a period of 30 days of continual operation.
b. Estimate the following statistics at each workstation:
   - Job flow times per type through the workstation
   - Machine utilization and average downtimes
   - Job delays in the queue of each process
   - Average time to batch five units at the shear-press workstation and each press-forming workstation
   - Average WIP levels at each workstation

4. *Job shop with sets*. Repeat Exercise 3 with a modified model using the concept of Arena sets, as illustrated in Section 13.6.

Chapter 14

Modeling Computer Information Systems

The last decade of the 20th century was witness to an enormous development of computer and communications systems. As processor speeds, memory sizes, and transmission rates were increasing at a seemingly exponential pace, they have progressively enabled ever more powerful computer-based systems. The Information Age is in full swing, symbolized most dramatically by the advent of the Internet and its explosive proliferation. Technology now pervades business, academia, and indeed our entire society. Computers collect and process vast collections of data. Fast communications networks mediate information exchange among computers and data sharing among users. It is not rare today to process a computer program on multiple processors simultaneously (parallel computing), or on a number of networked computers simultaneously (distributed computing).

Computer networks consist of computer nodes, called *hosts*. A common computer-network architecture is the *client/server* configuration, dating back to the 1980s. Here, a *client* is a host playing the role of a requester of services, while a *server* is a host providing the requisite service, and the two interact over a communications network. A host can be a client, a server, or both, depending on the system configuration. The client/server architecture is versatile, message based, and modular so as to improve the following aspects (Menasce et al. [1994]):

Usability—the ease with which a user can learn to operate, prepare inputs for, and interpret outputs from a system or component

Flexibility—the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed

Interoperability—the ability of two or more systems or components to exchange information and use it

Scalability—the ease with which a system or component can be "grown" to fit larger problems

In the 1990s, the cheaper desktop computer (personal computer, or PC) became the computer workstation of choice. Equipped with a graphical user interface (GUI), a fast

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

processor (CPU), considerable storage capacity, and multimedia devices, PCs pack substantial computational power at a reasonable price. Networked in client/server configurations, their power was augmented by fast communications, resulting in increased productivity.

The 1990s also witnessed the advent of the Internet and its protocols (e.g., TCP/IP) as a global communications infrastructure, and the concurrent advent of the World Wide Web (WWW, or simply, the Web) as the global information infrastructure. This information was organized as a web of interlinked pages, allowing users to "surf" among them with a click of the mouse. As the Internet and Web grew in leaps and bounds, enormous amounts of information were made accessible to an ever-increasing audience, and these developments ushered in a virtual technological revolution that swept the commercial, scientific, and educational domains. New applications that remove humans from the loop and thus reduce the cost of goods and services have become feasible. Such new Web-enabled applications include the following:

- *Electronic commerce* (*e-commerce*) has introduced an entirely new business paradigm that supports low-cost product marketing, and increased consumer convenience, resulting in lower product prices. In this paradigm, marketing products by companies as well as shopping by consumers are carried out electronically via Web pages. Online product catalogs, often with product images, permit consumers to browse, compare prices, place orders, and even pay electronically. An increasing volume of business has shifted to the Web, initially catalyzed by startup companies. These developments have been forcing many traditional companies (so-called "brick-and-mortar" companies) to follow suit. Electronic business (e-business) activities following this paradigm include business-to-business (B2B) for replenishing inventories by wholesalers and manufacturers as well as electronic auctions of all sorts of merchandise, business-to-consumers (B2C) for retail sales, and consumer-to-consumer (C2C) for direct sales among consumers.
- *The easy availability of information* on the Web has turned it into an enormous knowledge repository. Extensive searches for information over the Web have been enabled by search engines, which automate search procedures and rank the relevance of the results. These tools are becoming increasingly useful for scientists engaged in information finding and sharing, and are also widely used by ordinary people seeking more mundane information such as product prices, street addresses, or weather forecasts.
- *Distance learning* permits educational courses to be taught remotely, at the student's leisure, over the Internet. This technology can cut educational costs, since classrooms are largely obviated and class sizes can be almost unlimited.

The growing Internet/Web infrastructure has placed increasingly larger demands on servers as well as the networks connecting them. In particular, a crowded Internet can slow the delivery of responses (latency) to customer queries and transactions to a crawl, resulting in poor *quality of service* (*QoS*). The result is disappointed users and loss of business. Thus, performance evaluation of distributed client/server and web-based applications has become extremely important. For instance, electronic imaging systems require servers to perform computation-intensive image filtering operations. Likewise, database applications mediated by search engines can give rise to complex queries and searches that generate a large number of disk I/O operations. And extensive multimedia transmissions, particularly video, can rapidly tax the capacity of a communications network.

Assuming that the Web continues to grow at its current pace, applications will have to be designed with *capacity* issues in mind. For example, businesses often must formulate and answer "what-if" questions such as the following:

- What is the impact of a 20% increase in customer transactions on the response time of the system?
- How would the response time be changed if part of the workload were shifted from one hard disk to another?
- What good would it do to move a database to a remote host?

Such questions fall within the domain of *capacity planning*—the activity of predicting system performance for a given set of resource requirements in order to achieve a desired level of performance. For example, capacity planning of a client/server system might propose to allocate sufficient processor speeds and disk storage to ensure a QoS requirement, such as a sufficiently high probability of an adequately low response time. A capacity planning activity would launch a performance evaluation study via an analytical or simulation model to predict certain statistics of response times, and unearth any "performance bottlenecks" in the system. For further readings on capacity planning and performance evaluation of information systems, see Menasce and Almeida (1998), Cady and Howarth (1990), and Kant (1992), among many others.

In this chapter, we concentrate on simulation modeling of computer-based information systems, especially those configured in a client/server architecture. A number of examples will illustrate capacity planning for such systems.

14.1 CLIENT/SERVER SYSTEM ARCHITECTURES

Recall that in a client/server system, a client computer sends requests for specific services, while a server computer listens to client requests, processes them, and sends the response back to the client. The associated application program and its computational tasks are split between client and server so as to speed up response times by reducing communications overhead or achieving a degree of parallelism. Accordingly, the generally less-powerful client machine might be assigned some query preprocessing and post-processing tasks, while the generally more powerful server machine usually performs the core computational tasks. A client may request the services of multiple servers sequentially or in parallel. Examples of common servers include file servers and database servers, while examples of widely deployed client/server systems include transaction processing (TP) systems, such as online banking systems, Web-based online shopping software, inventory management software, and many others (see Gray and Reuter [1993], Hennessy and Patterson [1996], Menasce and Almeida [1998], Orfali et al. [1996]).

A client/server system is said to have an *n-tier architecture* depending on how clients and servers interact with each other. For instance, in a *two-tier architecture*, the client side utilizes a user interface that permits direct communication with the server host. Even though there may be more than one server, clients maintain direct connections with remote servers over the underlying communications network. Two-tier architectures seem to be appropriate for systems with up to about 100 clients.

Three-tier architectures emerged to overcome the limitations of two-tier architectures (Menasce and Almeida [1998]). In this architecture, a middle tier is added between the client side and the server side, which typically involves transaction processing, or

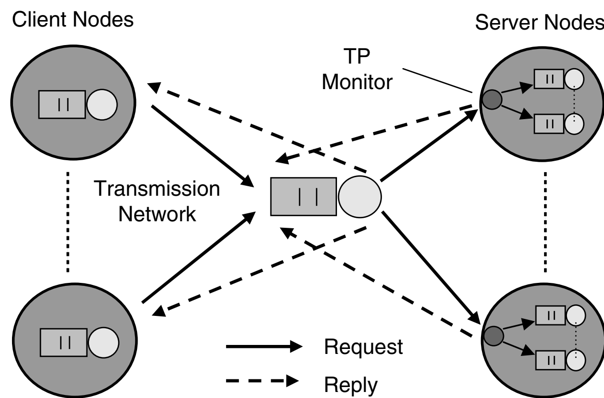


Figure 14.1 Schematic representation of a three-tier client/server system.

monitoring functionality. This middle tier may implement message queueing, task scheduling, prioritization of work in progress and application processing, or some combination of these. The middle tier thus interfaces the client side to the server side, but this interfacing is transparent to clients. The software implementing three-tier architectures is commonly known as *middleware*.

Figure 14.1 depicts a simple three-tier architecture in which the outer layers host computer processes and transaction queues while the middle layer monitors and routes transactions. Here, the TP monitor receives the transactions, queues them up for service, manages their path to completion, and finally, sends the reply back to the client.

For more information on client/server architectures and middleware, see, for example, Andrade et al. (1996) and Comer (1997). The elements of client/server systems will be described next.

14.1.1 Message-Based Communications

Clients and servers communicate via messages, which constitute client requests and server replies. Each message has origination and destination information, as well as a message body. A client message issues a request for a specific service from a server host (e.g., a database service). The reply from the server is packaged as another message, and sent back to the client. Thus, there are as many request types as there are services. However, in the course of serving a request, a server may itself act as a client by requesting services from other servers. For simplicity, we will not consider such secondary requests in this chapter.

Online banking is a common example involving message-based communications. Users might query their account status at an ATM (automatic teller machine) or access a secure website for the same. Such a query might be "show me the balance of my account" or "show me the last 15 cashed checks." The reply may be printed on paper or displayed on a computer monitor.

14.1.2 Client Hosts

Client hosts are computers that issue service requests initiated by computer programs or by customers connected to the host. As shown in Figure 14.2, requests emanating

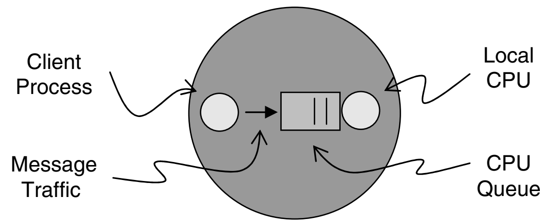


Figure 14.2 Schematic representation of a client host.

from a client host are generated by one or more client processes residing on the client host.

The request traffic stream generated by client processes may consist of a single request type, or more generally, a mix of types $k = 1, \ldots, N$, where the rate of the stream of type k requests is λ_k. For modeling purposes, it is common to merge (multiplex) all request streams into a single stream. In that case, the probability q_k that a request in the multiplexed stream is of type k is given by

$$q_k = \lambda_k / \sum_{j=1}^{N} \lambda_j, \quad k = 1, \ldots, N,$$

that is, as the relative rate of stream k in the multiplexed stream.

A generated request undergoes local preprocessing at the client host, and then is transmitted to a server host (usually over a communications network, unless the client and server are on the same host). On service completion, the reply is transmitted back to the client host and undergoes local post-processing, thereby completing the request/reply cycle. The *response time* of a request is the total time elapsed from the moment of submitting it (just before local pre-processing) until the moment the reply becomes available (just after local post-processing). Response times constitute the most critical performance measure in client/server systems.

14.1.3 Server Hosts

The bulk of request processing is typically performed at the server host. In addition to maintaining application code there, servers also call for a large amount of administrative operations. Figure 14.3 depicts a schematic representation of a typical server node.

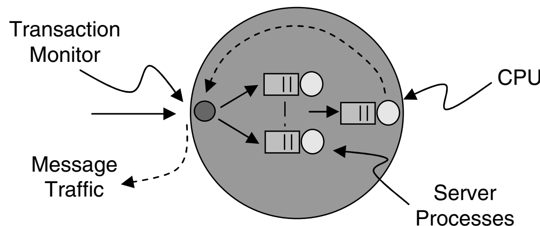


Figure 14.3 Schematic representation of a server host.

In its simplest form, a server node consists of one or more CPUs and a number of server processes that actually execute the code for the services requested by clients. The server processes and the CPU each have a message queue in front of them. Each server process is enabled to perform a specific set of services.

A *transaction processing* (*TP*) *monitor* process (*TP monitor*, for short) directs the incoming message traffic to the appropriate server process depending on the type of service requested. Since there may be other transactions (requests) awaiting their turn at the server process, incoming requests may have to be queued for service. When an incoming request eventually gets hold of the server process, it is immediately moved to the CPU queue for execution. Internally, such execution may follow a complex scenario. For example, the service of a request may generate read and/or write requests to local disks (I/O in computer jargon), or it may request information from a local or remote database. Eventually, the request transaction completes its service, and the server process sends the appropriate reply back to the monitor. The monitor then arranges to transmit the reply back to the client over the underlying network. The *elapsed time* of the request transaction at the server process is the total time elapsed from the moment it starts executing at the server's CPU until the moment it proceeds to the monitor for transmission back to the client. During the service time of a request the server process does not start serving another one.

A detailed model of the server host would have to account for the sequence of I/O requests and CPU processing, including service disciplines such as FIFO or round robin (see Section 8.2.3). This kind of detailed modeling will not be employed here. Rather, we will collapse all of this detail into a service time distribution for the sake of simplicity. Service distributions will be given as input parameters.

In reality, servers are considerably more complicated in the sense that they have software components to monitor each other and transmit client requests among themselves for secondary services. However, this complexity is transparent to the client. Since the objective here is to model client/server systems for Monte Carlo discrete-event simulation, we will be content to use this simplified representation in order to construct Arena models that can be simulated in the student version of Arena.

14.2 COMMUNICATIONS NETWORKS

The *communications network* (or simply, the *network*) connecting clients and servers plays a pivotal role in the performance of the client/server architecture. Its nodes include entrance and exit points that connect users to the network, as well as switches and routers that funnel traffic among nodes. A node in the network consists of transmission hardware and software, and network nodes interoperate and interact via transmission protocols. An example of a high-speed transmission protocol at the transport level is ATM (asynchronous transfer mode), which breaks up a message at the origination node into 53-byte packets (called *cells*) for transmission, and then assembles the cells back into messages at the destination node. Another example is Internet transmission, which is governed by a protocol suite, including TCP/IP.

Accurate models of communications networks can be quite complex depending on the level of detail incorporated by the modeler. In many cases, however, a detailed model is not necessary for obtaining adequate results. Accordingly, in this chapter, we

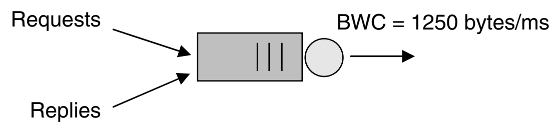


Figure 14.4 Single-server queue abstraction of a transmission node.

are content to model a communications network as a queueing network, with clients, servers, and transmission nodes.

Figure 14.4 abstracts a transmission node as a single-server queue. Incoming traffic is a multiplexed stream of messages comprised of a mix of client requests and server replies of various sizes. A buffer of given size precedes a transmission server operating at a given transmission speed (rate), known as the *bandwidth capacity (BWC)*. However, not all this capacity is available to messages. Accordingly, the *message transfer efficiency (MTE)* is the ratio of available BWC to total BWC, which is mostly in the range of 60 to 80% (the rest is consumed by transmission network overhead).

Each incoming message (request or reply) attempts to seize the transmission server, possibly queueing up in the buffer, and eventually gets transmitted to a destination node. The actual transmission time is proportional to the message size. For example, a BWC of 10 megabits per second (equivalent to 1250 bytes/msec) at 70% MTE will transmit a 1024-byte message in $1024/(1250^*0.7) = 1.17$ milliseconds. As soon as the message leaves the server, it is considered to have arrived at the destination node, which may be a server node, client node, or intermediate transmission node, since the service provided by the network is the transmission activity.

Because the buffer is finite, a communications network may experience transmission loss (unlike manufacturing systems with finite buffers, excess communications traffic is dropped rather than blocked). Communications models can incorporate lost messages and their statistics to obtain more realistic models.

14.3 TWO-TIER CLIENT/SERVER EXAMPLE: A HUMAN RESOURCES SYSTEM

Consider a human resources (HR) application, configured as a two-tier client/server system, consisting of four client nodes and one server node, with traffic flows as shown in the schematic representation of Figure 14.5. This HR application supports a number of services (request types) related to the company employees, using a database server that maintains an HR database (HRDB) of employee-related information. Database transactions types have the attributes displayed in Table 14.1.

Service requests belong to the following types:

1. A request of type *Add* adds a new employee with all his/her information (name, address, phone, expertise, etc.) to the HRDB in a message size of 1024 bytes, and the system returns a confirmation message of size 256 bytes.
2. A request of type *Delete* deletes an employee entry (with all the related information) from the HRDB. The system returns a confirmation message of size 512 bytes.
3. A transaction of type *Find* finds the complete employee information in the HRDB, based on partial data (e.g., the name alone). The system returns a reply message of size 512 bytes.

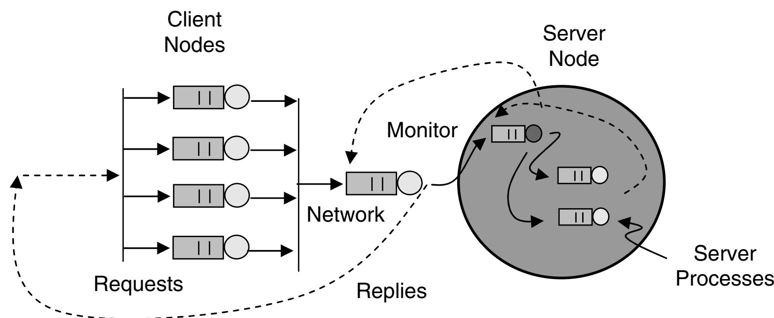


Figure 14.5 Schematic representation of the HR client/server system.

Table 14.1

Transaction attributes of the HR application

Request Type Number	Request Type Name	Service Requested	Request Size (bytes)	Reply Size (bytes)
1	*Add*	Add employee	1024	256
2	*Delete*	Delete employee	1024	512
3	*Find*	Find employee	256	512
4	*Search*	Search for employees	512	Disc({(0.2, 64), (0.3, 512), (0.3, 1024), (0.2, 2048)})

Table 14.2

Client-side service request profile

Client Node	Request Interarrival Distribution (messages/millisecond)	Request Type Distribution
1	Expo(1/30)	Disc({(0.2, 1), (0.15, 2), (0.4, 3), (0.25, 4)})
2	Expo(1/60)	Disc({(0.4, 1), (0.6, 2)})
3	Expo(1/120)	Disc({(0.6, 3), (0.4, 4)})
4	Expo(1/80)	Disc({(0.2, 1), (0.2, 2), (0.4, 3), (0.2, 4)})

4. A transaction of type *Search* searches the HRDB for all employees with common characteristics (e.g., same expertise, same department, etc.). The system returns a reply of random size; its (discrete) distribution is given in Table 14.1.

To characterize the traffic patterns in the system, we need to specify the arrival processes of all request types at each client node. Table 14.2 provides service request profiles. Note that the second column specifies the multiplexed request streams from each client node as Poisson processes with associated exponential interarrival time distributions. The type of any request in the corresponding multiplexed stream is specified by an associated discrete distribution in the third column.

The server node has a TP monitor that receives requests and routes them to the server process providing the requested service. There are two server processes, called *sp1* and

sp2; the former provides services of types *Add* and *Delete*, and the latter provides services of types *Find* and *Search*. The elapsed times for each service request type (including all CPU times and database-related times) are displayed in Table 14.3.

On service completion, replies to service requests are returned to the TP monitor, which in turn sends them back to their respective clients over the transmission network. The service time at the TP monitor is 1 millisecond per visit, while the BWC of the transmission network is 200 bytes/msec at 70% MTE.

Table 14.3

Elapsed times for service request types

Request Type Number	Request Type Name	Elapsed Time (milliseconds)
1	*Add*	10
2	*Delete*	8
3	*Find*	15
4	*Search*	Disc({(0.2, 10), (0.3, 25), (0.3, 32), (0.2, 45)})

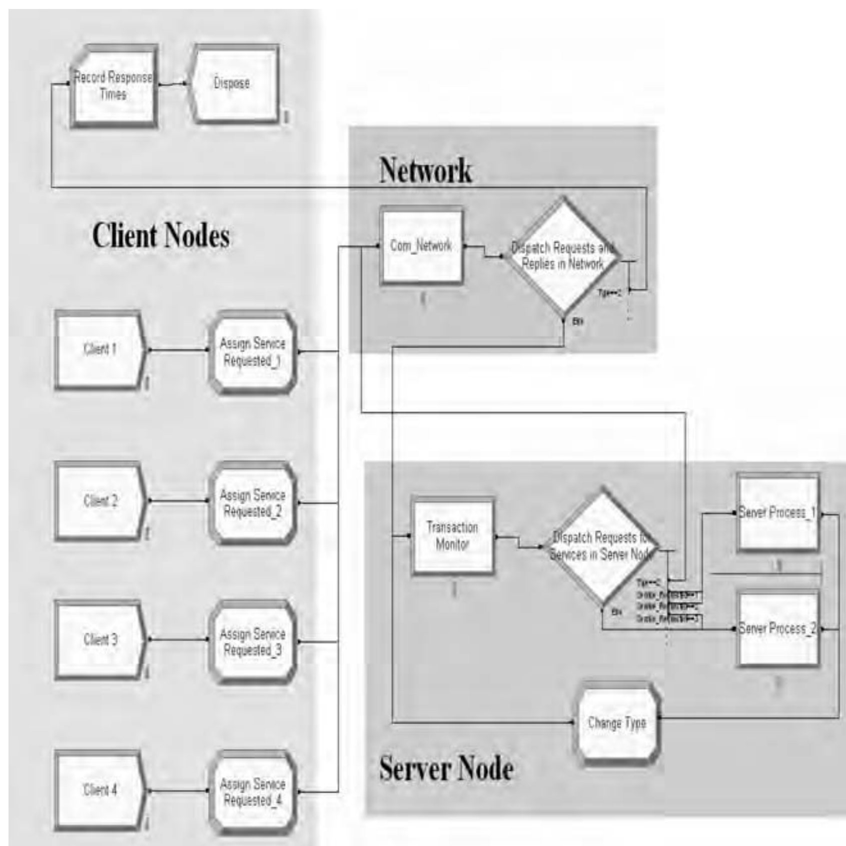


Figure 14.6 Arena model of the HR client/server system.

An Arena model of the system in Figure 14.5 is displayed in Figure 14.6. The Arena model consists of three labeled segments: client nodes, transmission network, and server node. We now proceed to describe each segment of the model, along with its simulation results.

14.3.1 Client Nodes Segment

Client nodes are modeled by the four *Create* modules, called *Client 1*, *Client 2*, *Client 3*, and *Client 4*, with the associated interarrival times specified in Table 14.2. Each *Create* module is connected to a corresponding *Assign* module, which sets the request entity attributes *ArrTime* (arrival time), *Service Requested* (request type number), and *Type* (1 codes for requests and 2 for replies).

The dialog box of the *Assign* module, called *Assign Service Requested_1* (for client node 1), is displayed in Figure 14.7. As usual, the arriving transaction entity's *ArrTime* attribute is set to *Tnow* (its arrival time). Its *Service_Requested* attribute is sampled from the discrete distribution specified in Table 14.2, with service coded by the integers 1, 2, 3, and 4; these integers correspond to the request type numbers in Tables 14.1 through 14.3. Finally, the *Type* field is set to 1 to indicate that it is a service request. Once the request is processed, the transaction entity would change its *Type* attribute to 2 to indicate its reply status.

14.3.2 Communications Network Segment

The transmission network segment provides a simplified model of the entire communications network. It consists of a *Process* module to model transmission delay, and a *Decide* module to model transaction routing.

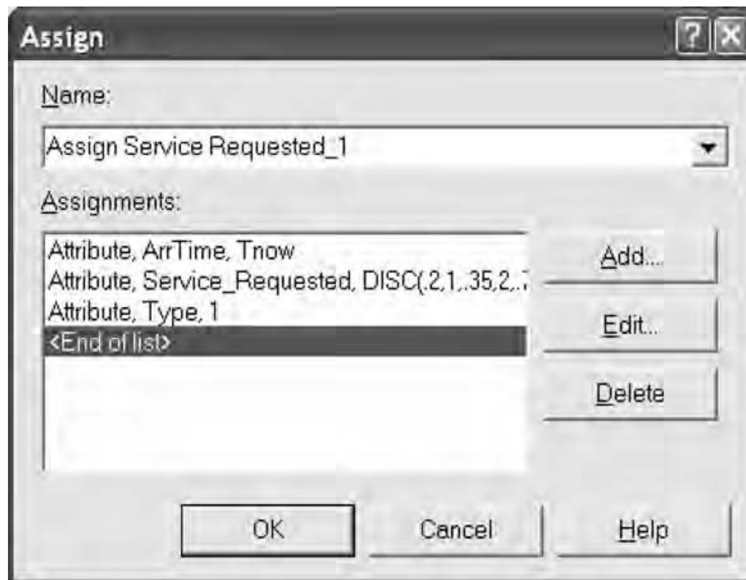


Figure 14.7 Dialog box of the *Assign* module *Assign Service Requested_1*.

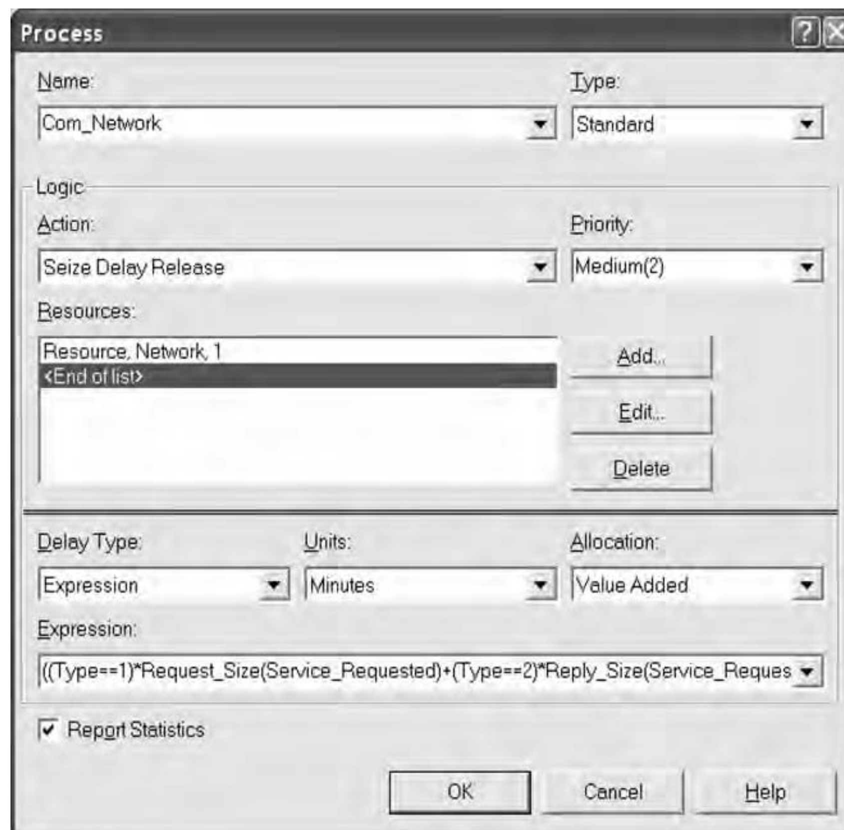


Figure 14.8 Dialog box of the *Process* module *Com_Network*.

Figure 14.8 displays the dialog box of the *Process* module, called *Com_Network*, which receives transaction entities from the two other segments. Here, the selected *Action* field option, *Seize Delay Release*, models the contention for the transmission resource, called *Network* (*Resources* field), and its seizure models the onset of actual transmission, which lasts until this resource's release.

Since both requests (on their way to the server node) and replies (on their way back to the client nodes) use this module for transmission, the corresponding message lengths (and subsequently transmission times) depend on both their service type and their transaction type (request or reply). For this reason, the *Expression* option was selected in the *Delay Type* field, and the *Expression* field specifies the delay time as

$$\begin{aligned}&((Type == 1)\ ^*\ Request_Size(Service_Requested) +\\&(Type == 2)\ ^*\ Reply_Size(Service_Requested))\\&(0.7\ ^*\ 200)\end{aligned}$$

In this expression, the first two lines determine the appropriate message size as a function of the attributes *Type* (request or reply) and *Service_Requested* (service type). Finally, in the third line, the message size is divided by the product of MTE (70%) and BWC (200 bytes/msec), thereby yielding the effective transmission rate. Arena

Expression - Advanced Process

	Name	Rows	Columns	Expression Values
1	Request_Size	4		4 rows
2	Reply_Size	4		4 rows

Expression Values

1	1024
2	1024
3	256
4	512

Figure 14.9 Dialog spreadsheet of the Expression module spreadsheet (left) and the *Expressions Values* dialog spreadsheet for specifying message sizes by type and service type (right).

does not support time units smaller than seconds, so for convenience, the model uses the minute unit as a millisecond unit.

Note that *Request_Size* and *Reply_Size* are used in the previous expression to retrieve the associated request size and reply size, respectively, as function of service type. Figure 14.9 displays the spreadsheet view of these expressions, using the *Expression* module from the *Advanced Process* template panel. The *Expression* module spreadsheet view at the left declares each of the two expressions, *Request_Size* and *Reply_Size*, to be a table with four rows (one for each service type) and one column (for the corresponding message size). The specification of the table corresponding to the expression *Request_Size* is displayed as an example on the right.

After its transmission delay at module *Com_Network*, each transaction entity proceeds to the *Decide* module, called *Dispatch Requests and Replies in Network*, to be dispatched to its requisite type-dependent destination. More specifically, the *Type* attribute is checked, and a service request (*Type* = *1*) is dispatched to the server node for service, while a reply (*Type* = *2*) is dispatched to the *Record* module, called *Record Response Times*, in the client nodes segment to tally its response time. The transaction entity then leaves the model via a *Dispose* module. In more realistic modeling scenarios, the replies would be routed back to their client node of origination to be processed, record their response times, and then be disposed of.

14.3.3 Server Node Segment

The server node segment processes service requests. It executes service requests brought in by request transactions, and dispatches them to the client nodes segment as reply transactions.

An incoming service request entity first enters the *Process* module, called *Transaction Monitor,* whose dialog box is displayed in Figure 14.10. Recall that the task of the TP monitor is to dispatch an incoming request to an appropriate server process, depending on the service type requested. All incoming request entities contend for the resource, called *Monitor*, which is governed by the now familiar *Seize Delay Release* option in the *Action* field. The delay time that models monitor processing time is precisely 1 millisecond, as indicated by the *Constant* option in the *Delay Type* field and the *Value* field.

Note carefully that the TP monitor handles both requests and replies — it dispatches requests to server processes and replies to the transmission network. The task of dispatching requests to the appropriate server process is carried out in the *Decide* module, called *Dispatch Requests for Services in Server Node*, whose dialog box is displayed in Figure 14.11.

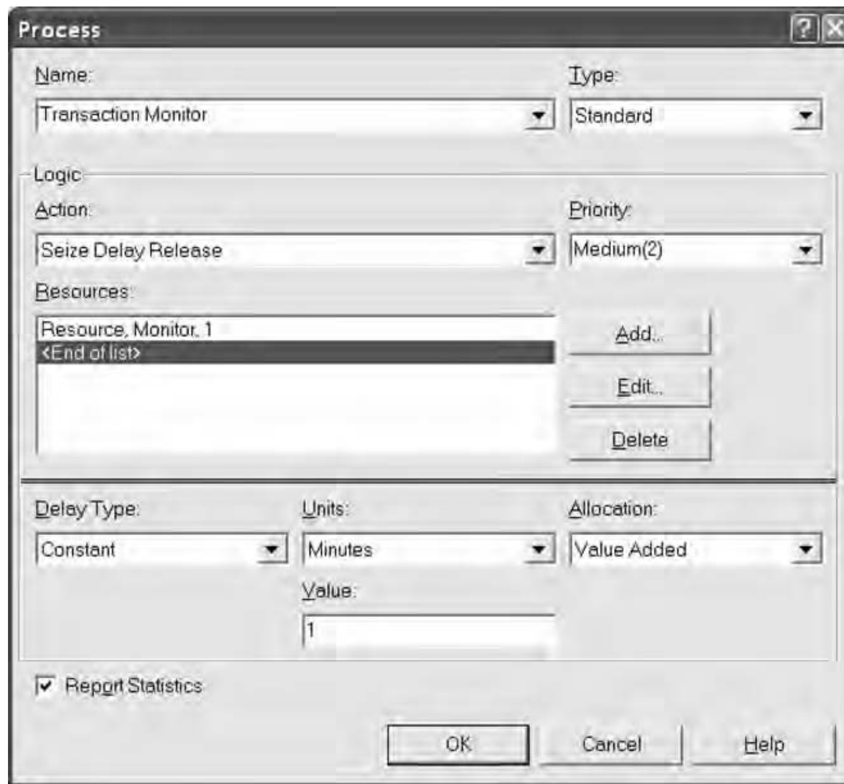


Figure 14.10 Dialog box of the *Process* module *Transaction Monitor.*

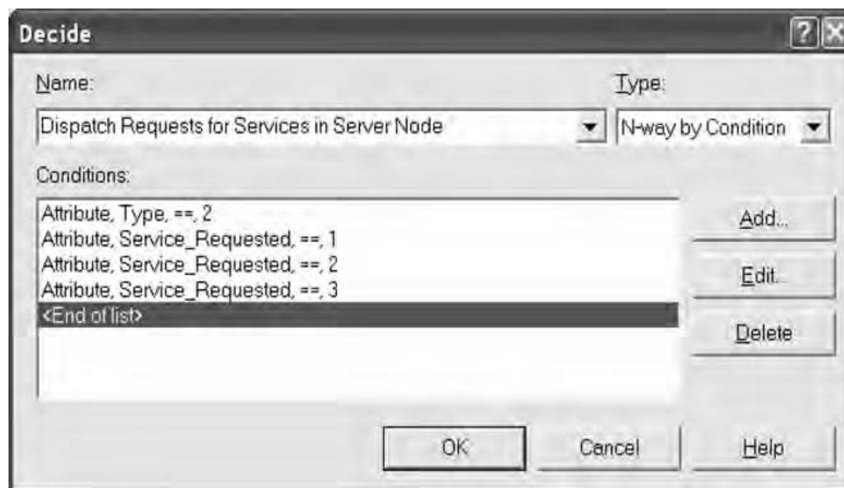


Figure 14.11 Dialog box of the *Decide* module *Dispatch Requests for Services in Server Node*.

The *N-Way by Condition* option in the *Type* field of the *Decide* module serves as a switch for dispatching incoming transaction entities. The dispatching decision computed in the *Conditions* field is as follows:

1. The *Type* attribute is checked first. If the predicate *Type == 2* is true, then this entity is a reply transaction to be sent to the transmission network. Otherwise, if *Type == 1* is true, then the entity is a request transaction, and the *Service_Requested* attribute is checked in the 4 items below.
2. If *Service_Requested == 1* is true (*Add* request), then the transaction entity is dispatched to server process 1.
3. If *Service_Requested == 2* is true (*Delete* request), then the transaction entity is dispatched to server process 1.
4. If *Service_Requested == 3* is true (*Find* request), then the transaction entity is dispatched to server process 2.
5. If *Service_Requested == 4* is true (*Search* request), then the transaction entity is dispatched to server process 2.

The server processes are modeled by the *Process* modules, called *Server Process_1* and *Server Process_2*, with respective resources *SP_1* and *SP_2*. The dialog box of module *Server Process_1* is displayed in Figure 14.12. Here again, the *Seize Delay Release* option was selected in the *Action* field, where incoming transactions entities contend for resource *SP_1*. Since the processing time is request dependent,

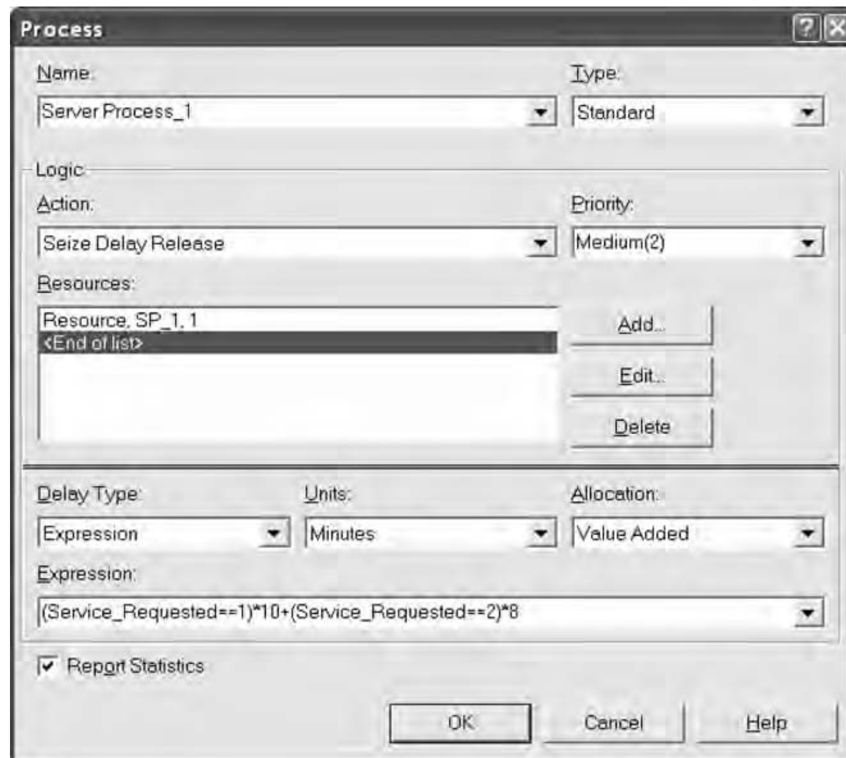


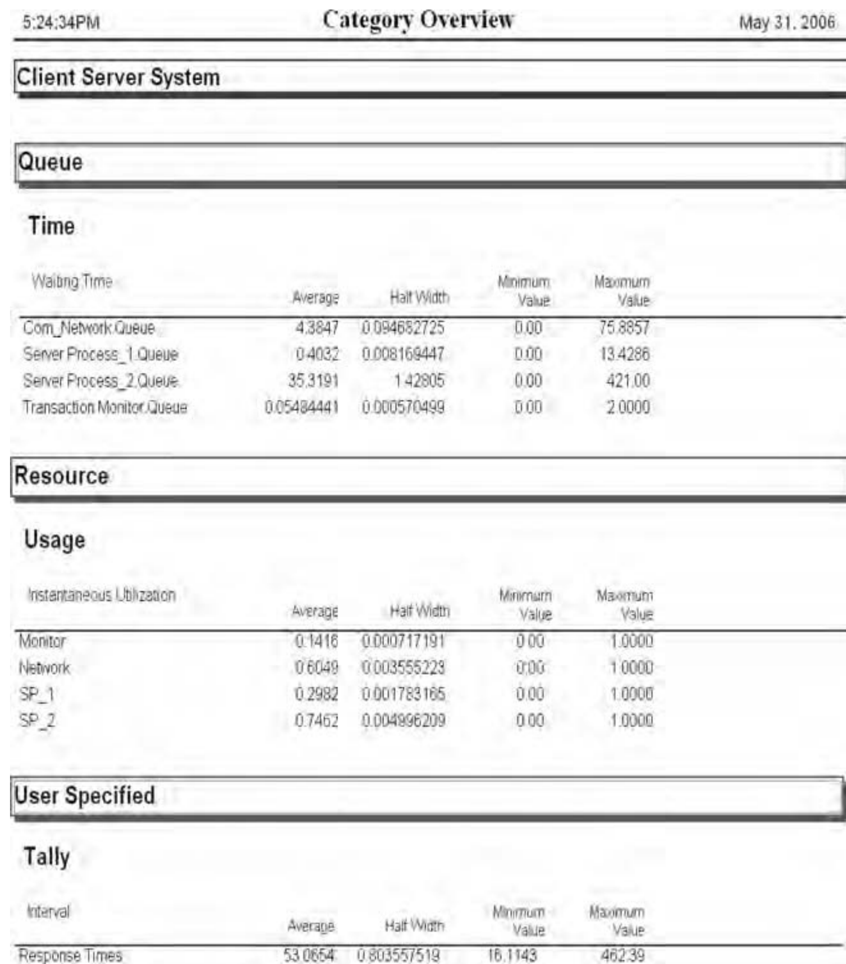
Figure 14.12 Dialog box of the *Process* module *Server Process_1*.

the processing delay is specified by an expression that assigns a 10-millisecond delay to *Add* requests (service type 1) and an 8-millisecond delay to *Delete* requests (service type 2). The dialog box of module *Server Process_2* is analogous.

Following service completion, the transaction entity enters the *Assign* module, called *Change Type*, where its *Type* attribute is changed from 1 (request) to 2 (reply). It then proceeds to the *Process* module, called *Transaction Monitor*, and after a brief processing delay, it enters the *Decide* module, called *Dispatch Requests for Services in Server Node*, which sends it to the transmission network segment for transmission back to the client nodes segment.

14.3.4 Simulation Results

The model of Figure 14.6 was simulated for 1 hour (3,600,000 milliseconds) of operation. The simulation results are displayed in Figure 14.13. An examination of the



5:24:34PM **Category Overview** May 31, 2006

Client Server System

Queue

Time

Waiting Time	Average	Half Width	Minimum Value	Maximum Value
Com_Network.Queue	4.3847	0.094682725	0.00	75.8857
Server Process_1.Queue	0.4032	0.008169447	0.00	13.4286
Server Process_2.Queue	35.3191	1.42805	0.00	421.00
Transaction Monitor.Queue	0.05484441	0.000570499	0.00	2.0000

Resource

Usage

Instantaneous Utilization	Average	Half Width	Minimum Value	Maximum Value
Monitor	0.1416	0.000717191	0.00	1.0000
Network	0.6049	0.003555223	0.00	1.0000
SP_1	0.2982	0.001783165	0.00	1.0000
SP_2	0.7462	0.004996209	0.00	1.0000

User Specified

Tally

Interval	Average	Half Width	Minimum Value	Maximum Value
Response Times	53.0654	0.803557519	16.1143	462.39

Figure 14.13 Simulation results for the Arena model of the HR client/server system.

simulation statistics reveals that the transmission network is busy 60% of the time, with an average buffer delay of 4.4 milliseconds. The TP monitor utilization of 14% shows that it is not congested.

Note that server process *SP_2* is more than twice busier than server process *SP_1*. This is due to the much longer elapsed times of services of type *Find* and *Search* as compared to services of type *Add* and *Delete*. Consequently, the average buffer delay at the queue of *SP_2* is over 35 milliseconds as compared to a fraction of a millisecond at the queue of *SP_1*.

The pooled response time of transactions varied between 16 milliseconds and 462 milliseconds with an average of 53.1 milliseconds and a tight confidence interval with a half-width of some 0.80 milliseconds. Clearly, some transactions experienced unacceptably long response times. To identify the system bottleneck, we recommend that response time statistics be collected and examined by service type (using the *Set* module to obtain statistics for each service type separately). Once the bottleneck is identified, additional service capacity can be added in the form of additional server processes and a modified allocation scheme of service types. For example, increasing the server capacity of resource *SP_2* from 1 to 2 would reduce the average pooled response time from 53.1 milliseconds to 36.3 milliseconds. Any further analysis is left to the reader.

14.4 THREE-TIER CLIENT/SERVER EXAMPLE: AN ONLINE BOOKSELLER SYSTEM

Consider a bookseller's e-business network, configured as a three-tier client/server system, as shown in Figure 14.14. The system consists of a cluster of two server nodes, linked by a transmission network and governed by TP middleware that dynamically balances the queue sizes of server processes in the cluster.

Each server node is depicted in Figure 14.14 by a circle that houses a number of server processes (in our case, two server processes per node). Server processes provide a number of services, and each server–process queue is a priority queue (requests are served FIFO within priority classes). The elapsed time for each service is random. The elapsed-time distributions are displayed in Table 14.4 along with priority information (smaller priority numbers indicate higher priorities).

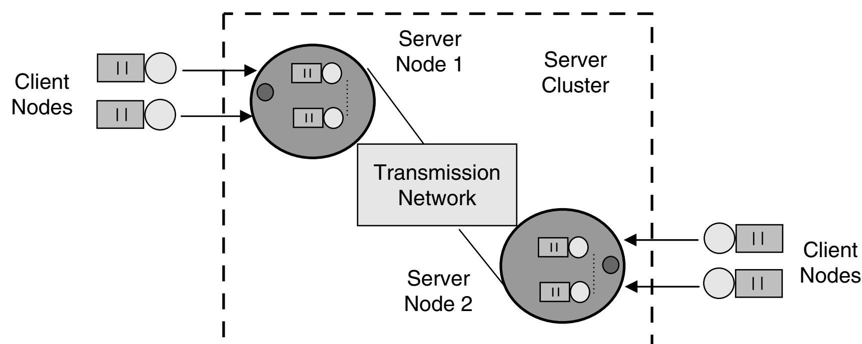


Figure 14.14 Layout of a bookseller's e-business network.

Table 14.4

Service data by server node

Service Number	Service Name	Service Priority	Elapsed Time at Server Node 1 (milliseconds)	Elapsed Time at Server Node 2 (milliseconds)
1	best sellers (BS)	5	Unif(10, 20)	Unif(18, 28)
2	new releases (NR)	4	Unif(13, 23)	Unif(21, 31)
3	book search (BE)	3	Unif(4, 14)	Unif(10, 20)
4	view cart (VC)	2	Unif(2, 12)	Unif(7, 17)
5	go to cashier (GC)	1	Unif(1, 11)	Unif(5, 15)

A number of client nodes are connected to each server node. However, when a service request arrives at a server node, it is not necessarily processed there. Rather, the TP monitor decides where it is to be processed by selecting a server process (anywhere in the system) with the minimal queue size. In reality, dynamic load balancing aims to equalize the queue workload (the total service time needed to serve all transactions in the queue, usually excluding the one in service). However, for the sake of simplicity, this model will balance only queue sizes.

Letting *SP_ij* denote server process j at node i ($i, j = 1,2$), Table 14.5 displays the allocation of services to each of the two server processes at each server node, and consequently, the workload distribution in the system.

The arrival streams of service requests are multiplexed into two arrival streams, one for each server node (see Figure 14.14). Request interarrival times are iid, exponentially distributed with service-dependent rates. Every request message is assumed to have a fixed size of 1024 bytes. The service discipline is FIFO within priority classes, that is, requests queue up for service at a server process according to their arrival times within their priority class (recall that smaller priority values represent higher priorities). Table 14.6 displays the interarrival time distributions of service requests and their mix at each server node.

Once a service request completes processing, a reply is sent back to the client node. Reply messages have a random size (1024 bytes or 8096 bytes). The reply size distribution is DISC({(0.45, 1024), (0.55, 8096)}). Finally, we are interested in the response times of service requests by type, and delays in server process queues.

In reality, the system's architecture is considerably more complicated; thus, Figure 14.14 is a simplified abstraction of the real system, suitable for the Arena student version. Nevertheless, the simplified model captures sufficient real-life detail to be useful.

Next, we will walk through an Arena model of the bookseller's e-business system. The model consists of three segments: request arrivals, the transmission network, and

Table 14.5

Allocation of services to server nodes

Client Node / Server Process	Service Name
node 1 / SP_11	BS, NR ,BE
node 1 / SP_12	VC, GC
node 2 / SP_21	BS, NR
node 2 / SP_22	BE, VC, GC

Table 14.6
Request interarrival time distributions and mix distributions by server node

	Node 1	Node 2
Distribution of Request Interarrival Times (milliseconds)	Expo(1/12)	Expo(1/18)
Distribution of Requests in Arrival Streams (percentages in mix):		
BS	0.20	0.23
NR	0.15	0.25
BE	0.30	0.25
VC	0.05	0.07
GC	0.30	0.20

server nodes (all server nodes have an identical structure). We now proceed to describe each segment of the model, along with its simulation results.

14.4.1 Request Arrivals and Transmission Network Segment

An Arena model of the request arrivals and transmission network segment is depicted in Figure 14.15. Client service requests are generated by the *Create* modules, called *Client Requests Node 1* and *Client Requests Node 2*, as two multiplexed streams.

Figure 14.16 displays the dialog box of the first *Create* module. Note that here and elsewhere, the base time unit of the model is seconds in lieu of milliseconds (again for convenience). Note further that the interarrival times are iid and exponentially distributed, so that the arrival streams constitute Poisson processes. Recall also that the multiplexing of Poisson processes yields aggregate Poisson process (see Section

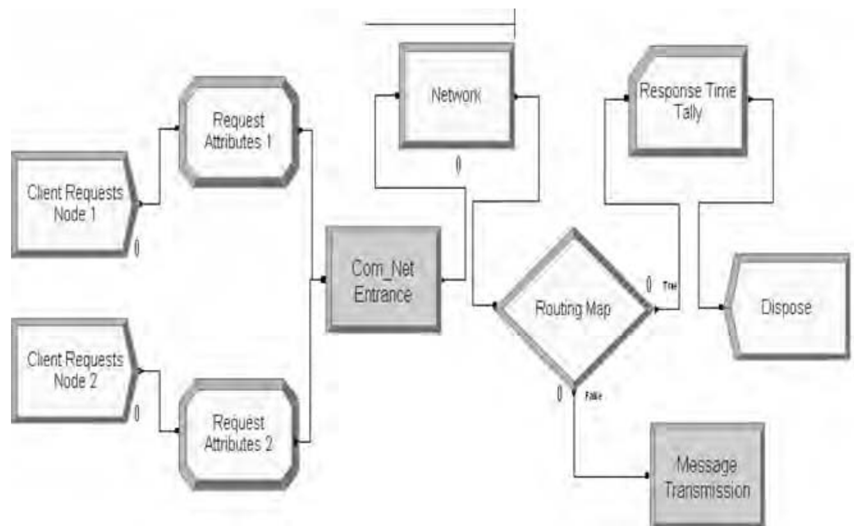


Figure 14.15 Arena model of the request arrivals and transmission network segment for the bookseller's e-business system.

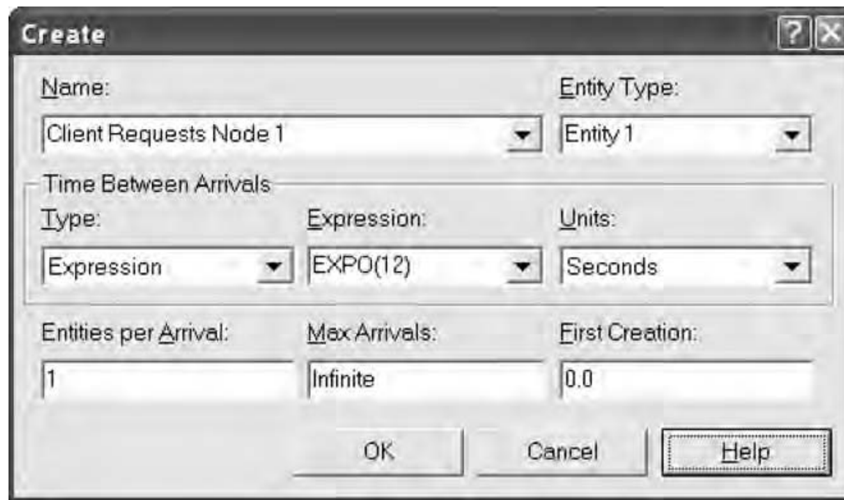


Figure 14.16 Dialog box of the *Create* module *Client Requests Node 1*.

3.9.2). Thus, each of the two *Create* modules may be validly defined to generate a combined Poisson stream of requests from multiple client nodes.

Generated request transaction entities proceed to the corresponding *Assign* module, called *Request Attributes 1* or *Request Attributes 2*, to assign values to a number of attributes. Figure 14.17 displays the dialog box of the second *Assign* module. Each arriving request entity is independently assigned a service type from a discrete distribution in attribute *Svc* (from Table 14.6), a corresponding priority (from Table 14.4) in attribute *Priority*, the arrival time (*Tnow*) in attribute *Arr_Time*, a message type (variable

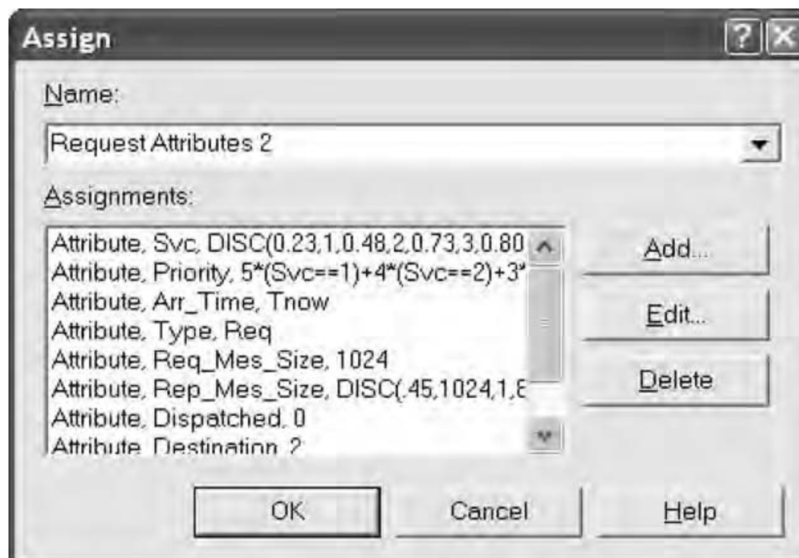


Figure 14.17 Dialog box of the *Assign* module *Request Attributes 2*.

Req for request or variable *Rep* for reply) in attribute *Type*, and the appropriate message sizes from a discrete distribution in attributes *Req_Mes_Size* and *Rep_Mes_Size*.

Finally, the attributes *Dispatched* and *Destination* maintain routing information for each transaction entity during its sojourn in the system. To understand their role in the model, recall that each transaction entity is affiliated on arrival with a particular server node, and this information is assigned to attribute *Destination* (1 codes for node 1, and 2 for node 2). These values subsequently serve as station numbers in the *Route* module called *Message Transmission*. When referencing station numbers in the model, the modeler should fetch the right numbers from the *Stations* element in the corresponding experiment file *filename.exp*. Stations are listed in ascending order and their index starts at 1. The *Dispatched* attribute maintains routing information of an arriving transaction entity. Initially (at the client node), the *Dispatched* attribute is assigned the value 0 (see Figure 14.17); subsequently, if the transaction entity is dispatched to receive service in another server node, then the *Dispatched* attribute is assigned the value 1. On each entry in a server node, the *Dispatched* attribute is examined by the node's TP monitor to determine whether the transaction entity was dispatched from another node. If it was, then the transaction entity will be served in the current node, and will not be dispatched again (otherwise, a transaction might be dispatched forever, possibly creating an "infinite loop" problem). The variable *Req* is initialized to 1 (to code for a request), while *Rep* is initialized to 2 (to code for a reply). Additionally, the variable *BWC* (representing the bandwidth capacity) is initialized to 1250 bytes/msec. These initializations are made in the *Variable* module spreadsheet.

14.4.2 TRANSMISSION NETWORK SEGMENT

The Arena model of the transmission network segment is depicted in Figure 14.15. Once the transaction entity leaves the *Assign* module (*Request Attributes 1* or *Request Attributes 2*), it enters the transmission network by proceeding to the *Station* module, called *Com_Net Entrance*, whose dialog box is displayed in Figure 14.18. While entities often enter a *Station* module from a *Route* module (via a "long jump"), here client nodes are directly connected to the *Station* module *Com_Net Entrance* to reduce the number of modules in the model. To model a transmission delay in the network, the transaction entity next enters the *Process* module, called *Network*, whose dialog box is displayed in Figure 14.19. Transmission is modeled as FIFO contention for the resource, called *Net_server*, via the usual *Seize Delay Release* option in the *Action* field to obtain network delay statistics per transaction entity. Since transmission times are size dependent, the transmission delay is computed as an expression utilizing the attributes *Req_Mes_Size* and *Rep_Mes_Size*, which hold the requisite message size. Thus, the expression in the *Expression* field specifies the transmission time as

$$((Type == Req)^{*}Req_Mes_Size + (Type == Rep)^{*}Rep_Mes_Size)/BWC/0.8.$$

Here the message size is selected by message type, and the requisite service time is obtained by dividing the message size by the effective bandwidth capacity $BWC^{*}0.80$, where 0.80 is the MTE parameter.

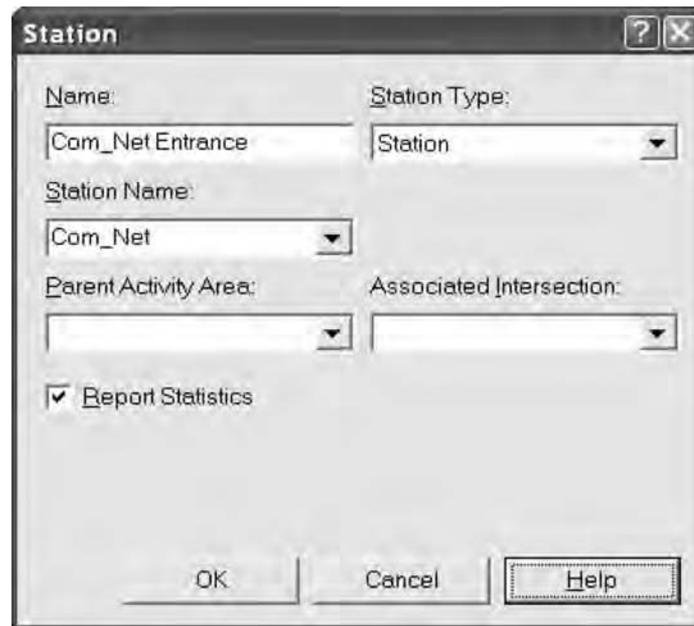


Figure 14.18 Dialog box of the *Station* module *Com_Net Entrance*.

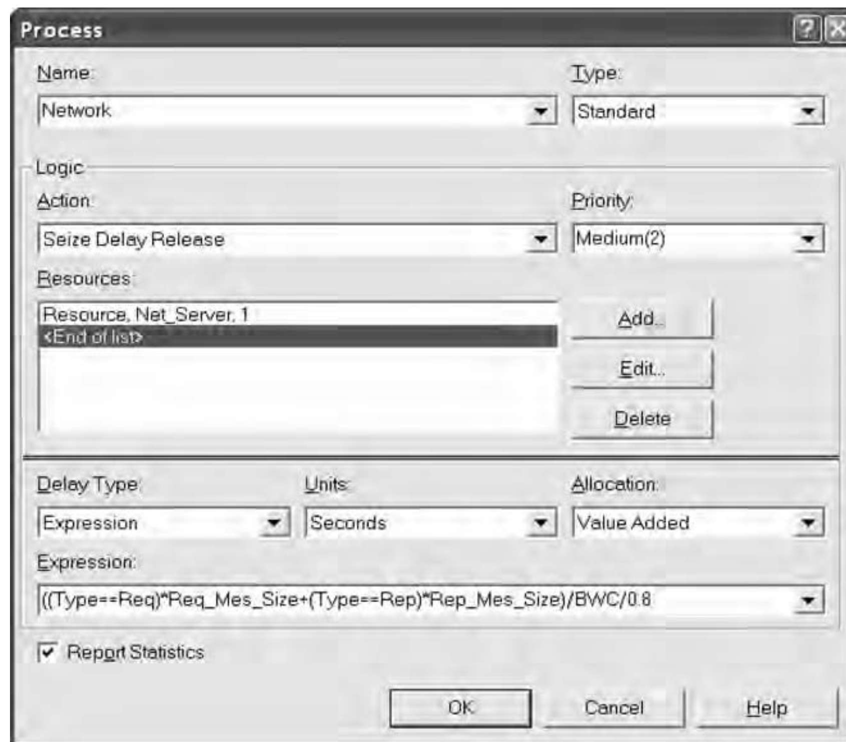


Figure 14.19 Dialog box of the *Process* module *Network*.

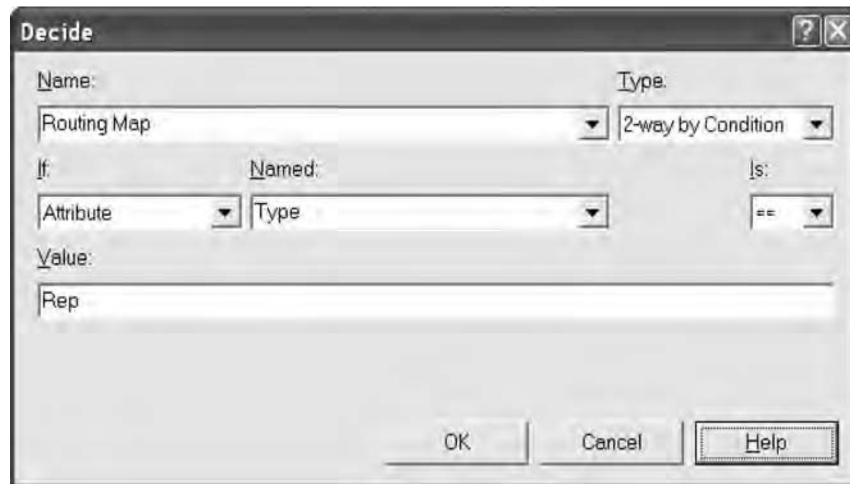


Figure 14.20 Dialog box of the *Decide* module *Routing Map*.

Since both requests and replies are transmitted across the network, transactions are routed to their destinations based on their type by the *Decide* module, called *Routing Map*, whose dialog box is displayed in Figure 14.20. The *2-Way by Condition* option in the *Type* field specifies two branches. If the entity is a reply transaction, then it is dispatched to the *Record* module, called *Response Time Tally*, to record its response time, from which it proceeds to a *Dispose* module. (We mentioned earlier that in reality, a reply transaction is dispatched back to its original client node before tallying its response time, but here we use a simplified model to economize on model size.) If the entity is a request transaction, then it is dispatched to the *Route* module called *Message Transmission*, whose dialog box is displayed in Figure 14.21. Here,

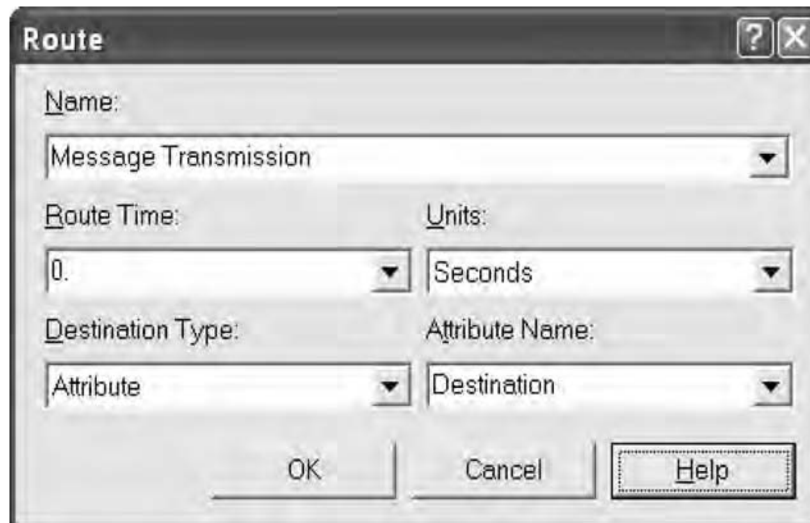


Figure 14.21 Dialog box of the *Route* module *Message Transmission*.

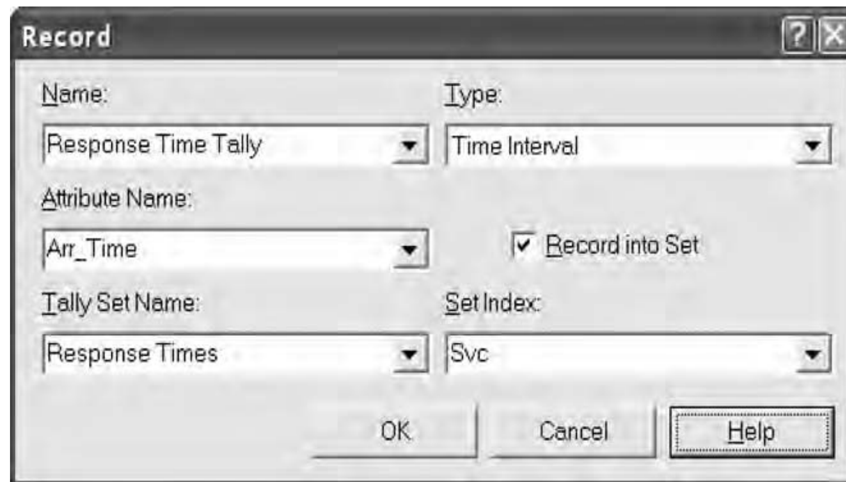


Figure 14.22 Dialog box of the *Record* module *Response Time Tally*.

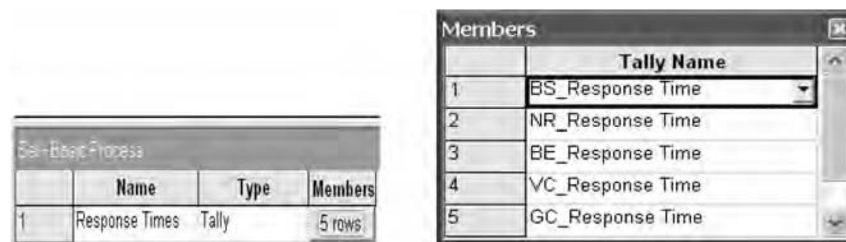


Figure 14.23 Dialog spreadsheet of the *Set* module (left) of the *Members* dialog spreadsheet for specifying the set *Response Times* (right).

a request transaction entity is routed to its destination station whose ID is stored in the *Destination* attribute of the entity. Note carefully that the zero routing time in the *Route Time* field does *not* represent transmission delay. The actual transmission delay is modeled in the *Process* module *Network* as explained earlier.

The *Record* module, called *Response Time Tally*, records transaction entities' response times into a tally set, called *Response Times*, whose dialog box is displayed in Figure 14.22. Note that response times are collected by service type in the tally set, called *Response Times*, and the *Set Index* field specifies the service type attribute *Svc* as an index to access the requisite tally.

The specification of the tally set is made in the *Set* module whose spreadsheet view is displayed in Figure 14.23 (left) and its *Members* dialog spreadsheet (right).

14.4.3 Server Nodes Segment

The remaining segment models server nodes, but since they are analogous, we will only review one of them. Accordingly, Figure 14.24 depicts an Arena model of server node 1.

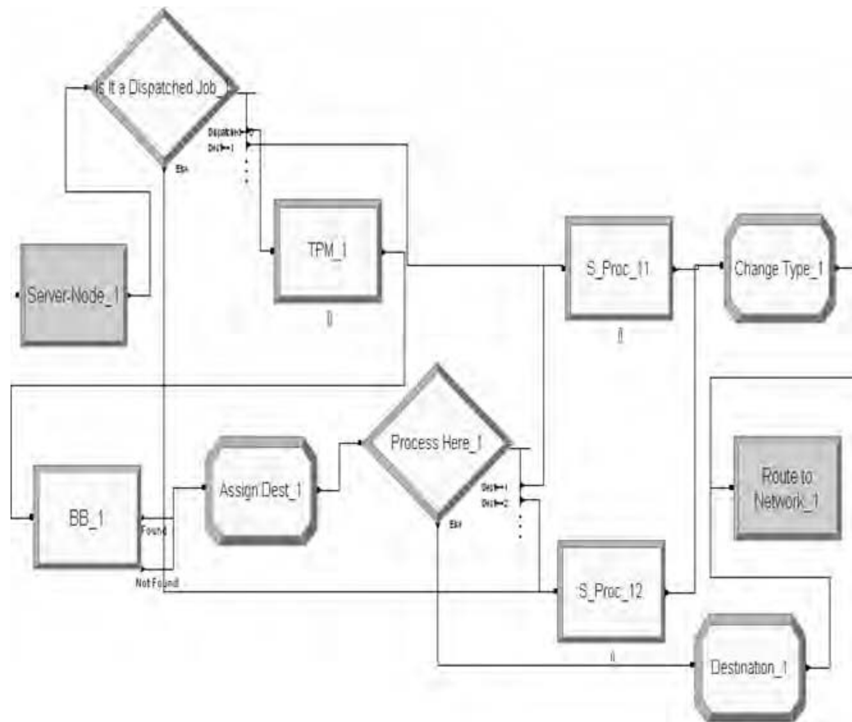


Figure 14.24 Arena model of server node 1 of the bookseller's e-business system.

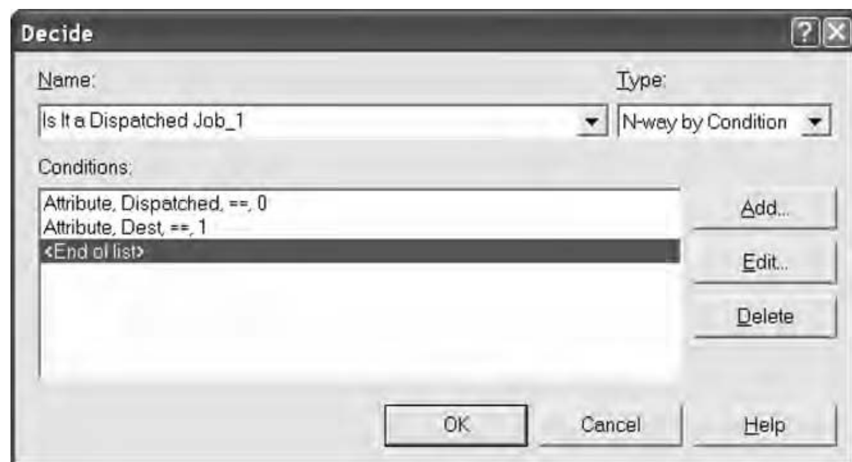


Figure 14.25 Dialog box of the *Decide* module *Is It a Dispatched Job_1*.

Transaction entities enter the server node at the *Station* module, called *Server-Node_1*, and then proceed to the *Decide* module, called *Is It a Dispatched Job_1*, whose dialog box is displayed in Figure 14.25. The role of this module is to examine incoming transaction entities and to separate those arriving directly from client nodes from those dispatched from server node 2 by the TP monitor. This separation is

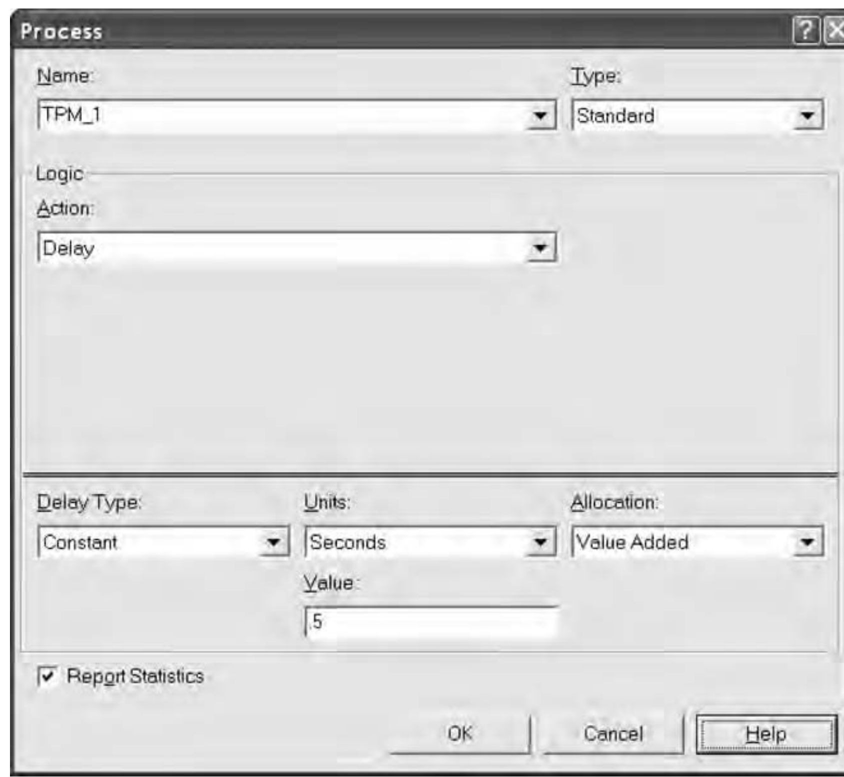


Figure 14.26 Dialog box of the *Process* module *TPM_1*.

necessary, since the corresponding logic sequences are different, as explained below (see also the discussion at the end of Section 14.4.1).

Suppose the value of the *Dispatched* attribute of the incoming transaction entity is zero. Then the entity is a request transaction arriving from a client node. This transaction entity is immediately routed to the corresponding TP monitor at the *Process* module, called *TPM_1*, whose dialog box is displayed in Figure 14.26. Note that this *Process* module represents the initial TP monitor processing, but does not model contention, since the *Action* field setting is the *Delay* option. The modeling justification for this choice is that the processing time of a routing decision is just 0.5 milliseconds, which is tiny relative to the means of interarrival times. Consequently, queueing delays here are highly unlikely and can be ignored for all practical purposes.

Following the initial TP monitor processing, a request transaction entity proceeds with the main routing processing by entering a *Search* module (from the *Advanced Process* template panel), which specifies a search for the optimal server process (one with the minimal queue size in the cluster to be assigned to this request transaction). Figure 14.27 displays the dialog box of the corresponding *Search* module, called *BB_1*, commonly known as the *bulletin board* in middleware parlance. Here, the *Type* field specifies a search type option (available options are *Search a Batch*, *Search a Queue*, or *Search an Expression*). The *Search Condition* field is used to specify a search criterion, while the *Starting Value* and *Ending Value* fields specify

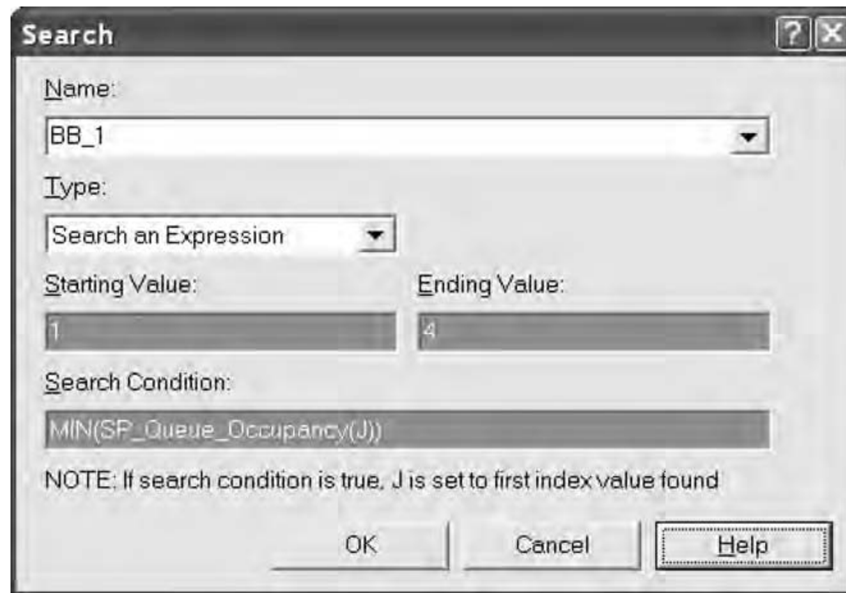


Figure 14.27 Dialog box of the *Search* module *BB_1*.

the search range; these values also determine if the range is searched in ascending (increasing) or descending (decreasing) order. Thus, a search succeeds if the search condition is satisfied for some item in the search range, and it fails otherwise. The search result is always an integer assigned to the system variable *J*, and its interpretation depends on the search type as follows:

- If the search fails, then *J* is set to 0.
- For the *Search a Batch* option, a batch (group) of entities is searched for compliance with the search condition over the range of ranks specified by the *Starting Value* and *Ending Value* fields. If the search succeeds, then *J* is set to the rank of the first entity found in the batch.
- For the *Search a Queue* option, a queue of entities is searched for compliance with the search condition over the range of ranks specified by the *Starting Value* and *Ending Value* fields. If the search succeeds, then *J* is set to the rank of the first entity found in the queue.
- For the *Search an Expression* option, a vector of expressions (such as in the *Expression Values* column of the *Expression* module spreadsheet or in the *Members* column of the *Advanced Set* module spreadsheet) is searched for compliance with the search condition over the range of indices specified by the *Starting Value* and *Ending Value* fields. If the search succeeds, then *J* is set to the index value of the first expression found in the module. See Figures 14.27 and 14.28 for an example.

The search procedure itself unfolds as follows. When an entity arrives in a *Search* module, the system variable *J* is set to the value in the *Starting Value* field. The search criterion in the *Search Condition* field is then checked. If true, the search ends and the variable *J* returns the search result. Otherwise, *J* is incremented (or decremented) by 1, and the search condition is rechecked as before. If the search reaches the value in the

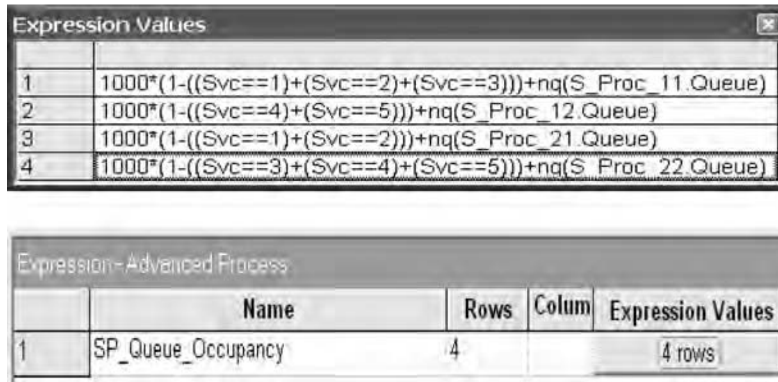


Figure 14.28 Dialog spreadsheet of the *Expression* module (bottom) and the *Expression Values* dialog spreadsheet for specifying the expression vector *SP_Queue_Occupancy* (top).

Ending Value field, without finding an entry that satisfies the search condition, the variable *J* is set to 0 to indicate that the search failed. For example, an ascending search of a queue for the entity with the highest *Age* attribute is specified by the search condition *MAX(Age)* over the rank range 1 to *NQ* (the name of the queue is specified in the *Queue Name* field). As another example, a descending expression search for reordering a product in an inventory is specified by the search condition

$$\textit{Inventory Level}(J) < \textit{Reorder Point}(J)$$

over the index range 8, ..., 2.

In our model, Figure 14.27 specifies an ascending expression search for a server process with the minimal occupancy, via the search condition expression

$$MIN(SP_Queue_Occupancy(J)),$$

where *SP_Queue_Occupancy* is an expression. As shown at the bottom of Figure 14.28, the expression vector *SP_Queue_Occupancy* is a vector of four rows, each holding a separate expression. Consequently, the search procedure will scan the rows (with the *J* variable holding the current row index) and return the row index with the minimal value. It can be seen that all rows are similarly structured. Each row returns the queue size of a specific server process, provided that the service request of a transaction entity belongs to a specific subset of services; otherwise, it returns a large number (1000), which is larger than any queue size. Consequently, if the transaction entity's service request (in its *Svc* attribute) is *not* in that subset, the *MIN* search condition for that row will deliberately fail. For instance, the second row returns the queue size of server process *S_Proc_12* for service types 4 and 5, and 1000 for all other services. When the search over the rows of *SP_Queue_Occupancy* is terminated, the variable *J* will contain the index of the destination server process with the smallest queue size, as required.

Next, the request transaction entity moves from the *Search* module to the *Assign* module, called *Assign Dest_1*, whose dialog box is displayed in Figure 14.29. Here, the system variable *J* is assigned to the transaction entity's attribute *Dest*, to be used in dispatching that transaction entity to the requisite server process (which may reside in either of the server nodes).

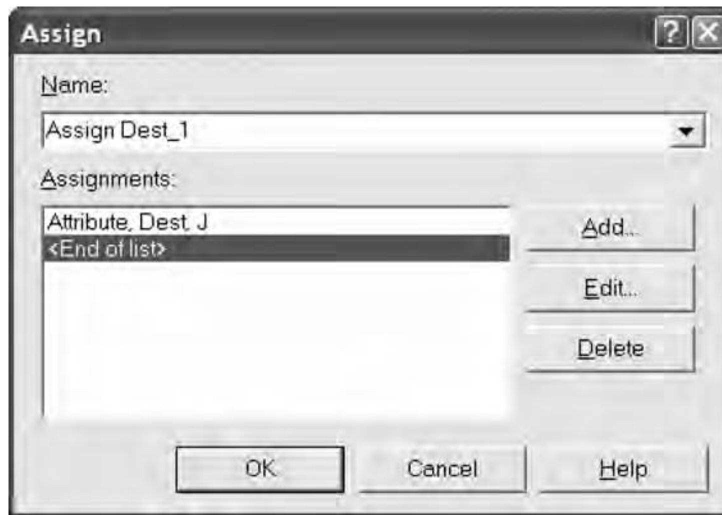


Figure 14.29 Dialog box of the *Assign* module *Assign Dest_1*.

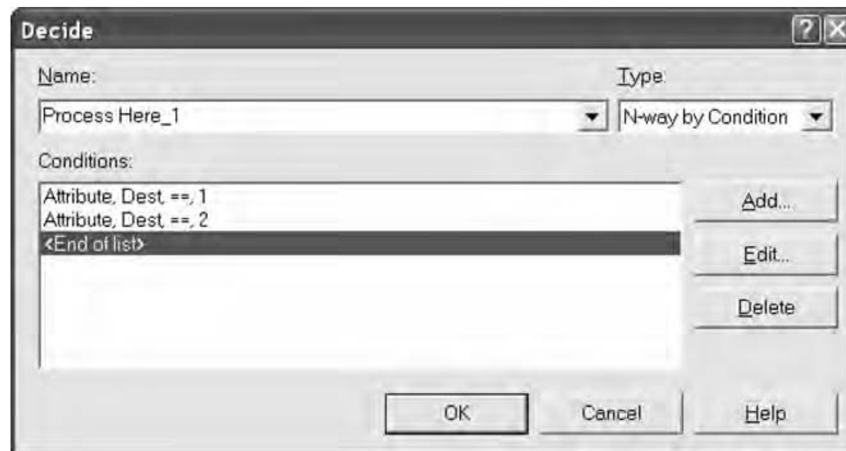


Figure 14.30 Dialog box of the *Decide* module *Process Here_1*.

The actual dispatching is carried out next in the *Decide* module, called *Process Here_1*, whose dialog box is displayed in Figure 14.30. If the value of the *Dest* attribute of a request transaction entity is 1 or 2, then this module dispatches it to *Process* module *S_Proc_11* (modeling *SP_11*) or to *Process* module *S_Proc_12* (modeling *SP_12*), respectively. However, if *Dest* > 2 is true, then this indicates that the requisite server process is in the other server node (in our case, *Server Node 2*), and this module dispatches the transaction entity to the *Assign* module, called *Destination_1*, where its *Destination* attribute is set to 2.

Figure 14.31 displays the dialog box of the *Process* module *S_Proc_11*. Contention for the server process resource *SP_11* is modeled by the *Seize Delay Release* option in the *Action* field. The service-dependent elapsed time is specified by the expression

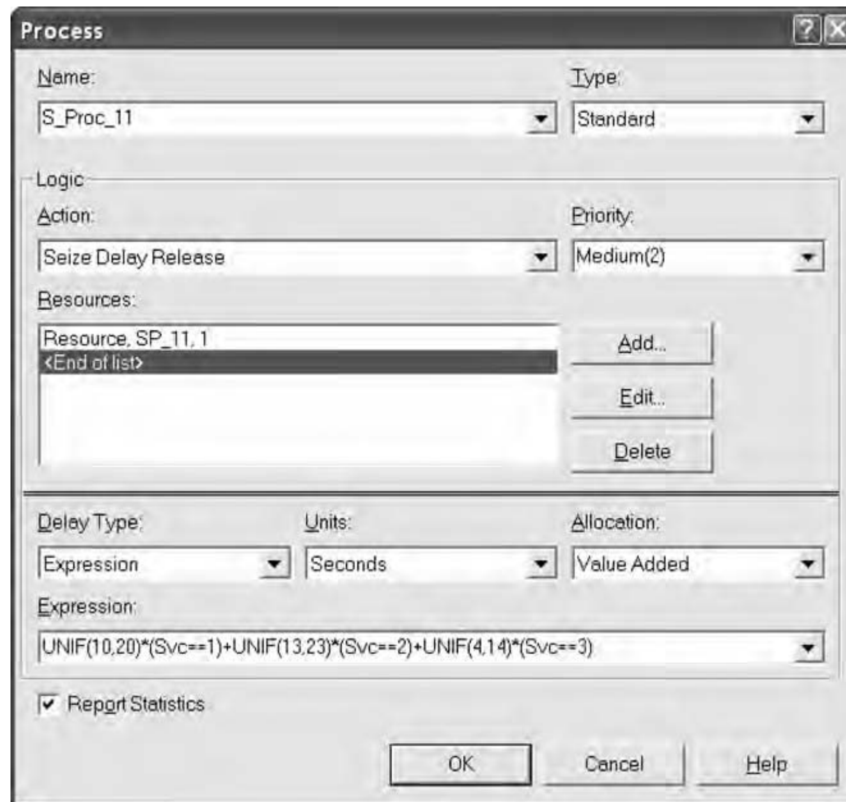


Figure 14.31 Dialog box of the *Process* module *S_Proc_11*.

$$UNIF(10, 20)\ ^{*}\ (Svc == 1) + UNIF(13, 23)\ ^{*}\ (Svc == 2) + UNIF(4, 14)\ ^{*}\ (Svc == 3),$$

which implements the relevant data from Table 14.4.

Recall that the transmission network queue is served in FIFO order, while the order of service in the server process queues is governed by a priority discipline (see Table 14.4 and Figure 14.17). These service disciplines are specified in the *Queue* module spreadsheet by selecting the appropriate options in the *Type* column and the *Attribute Name* column, as shown in Figure 14.32.

Recall that once the service request completes execution, the requesting transaction needs to be sent back to the origination client as a reply. To this end, the transaction entity first enters the *Assign* module, called *Change Type_1*, whose dialog box is displayed in Figure 14.33. The *Type* attribute is set here to the variable *Rep* to indicate a change of the transaction type from request to reply.

Next, to model the transmission to its associated client node, the reply transaction entity proceeds to the *Route* module, called *Route to Network_1*, whose dialog box is displayed in Figure 14.34. It is then routed in zero time to *Station* module *Com_Net Entrance*, from which it proceeds to the *Process* module *Network* for the actual network contention and transmission delay, as described in Section 14.4.2. Recall again that the seconds time unit actually stands for milliseconds.

Queue - Basic Process

	Name	Type	Attribute Name	Shared	Report Statistics
1	S_Proc_12.Queue	Lowest Attribute Value	Attribute 1	☐	☑
2	S_Proc_11.Queue	Lowest Attribute Value	Priority	☐	☑
3	S_Proc_21.Queue	Lowest Attribute Value	Priority	☐	☑
4	S_Proc_22.Queue	Lowest Attribute Value	Priority	☐	☑
5	Network.Queue	First In First Out	[illegible]	☐	☑

Figure 14.32 Dialog spreadsheet of the *Queue* module specifying the server process queues and transmission network queue properties.

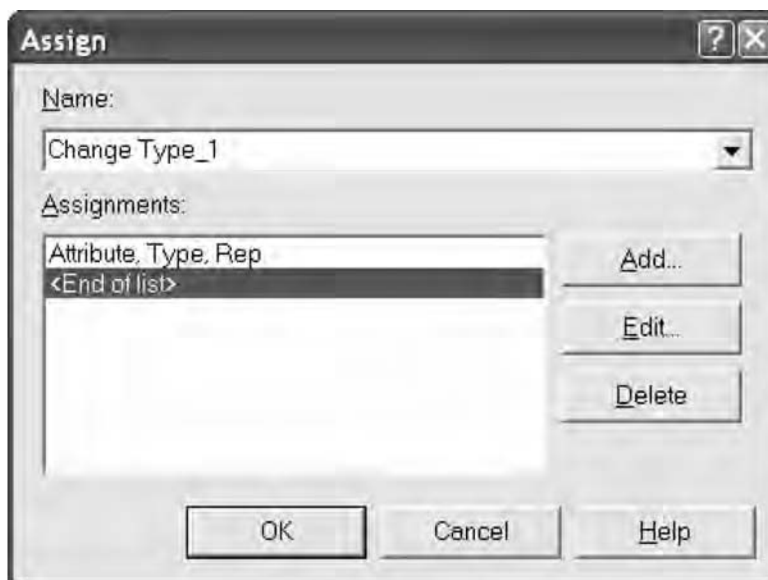


Figure 14.33 Dialog box of the *Assign* module *Change Type_1*.

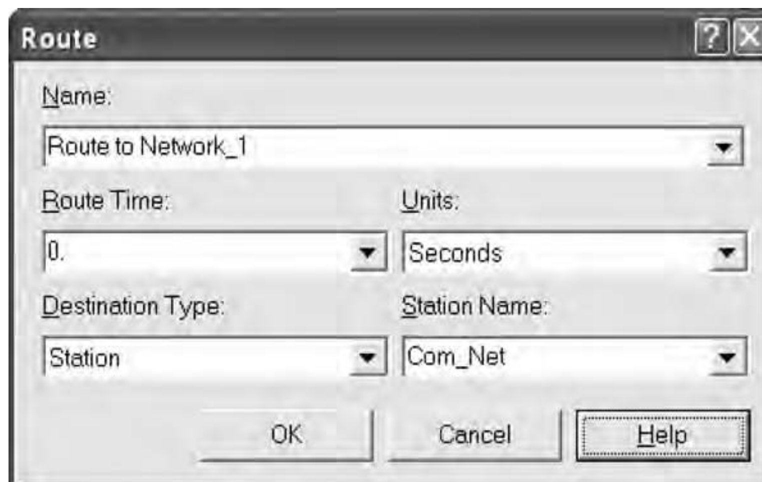
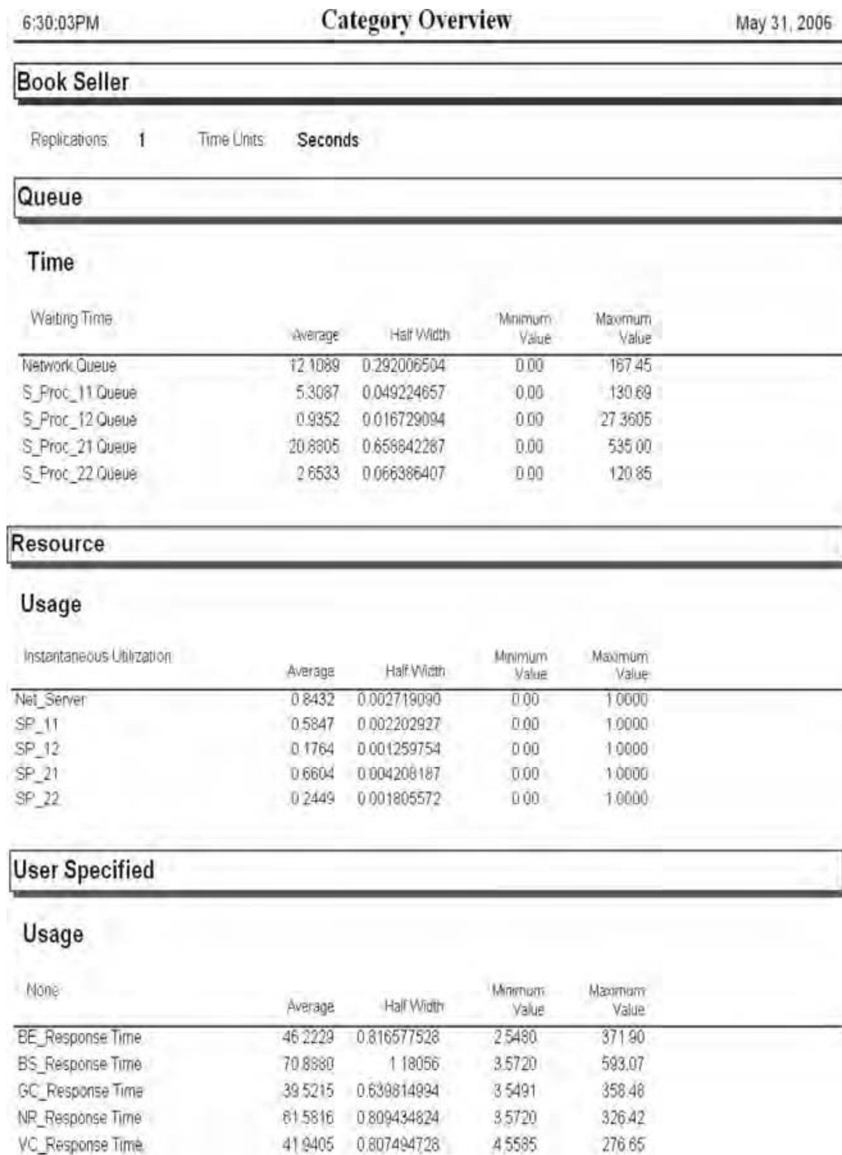


Figure 14.34 Dialog box of the *Route* module *Route to Network_1*.

14.4.4 SIMULATION RESULTS

The Arena model of the bookseller's e-business system was simulated for a 1-hour period (3,600,000 milliseconds) of operation. The simulation results are displayed in Figure 14.35.

Observe that the network resource *Net_Server* is busy about 84% of the time, resulting in average network delay of around 13 milliseconds across all transaction entities. These delays are quite insignificant, and entirely acceptable.



6:30:03PM **Category Overview** May 31, 2006

Book Seller

Replications: 1 Time Units: **Seconds**

Queue

Time

Waiting Time	Average	Half Width	Minimum Value	Maximum Value
Network.Queue	12.1089	0.292006504	0.00	167.45
S_Proc_11.Queue	5.3087	0.049224657	0.00	130.69
S_Proc_12.Queue	0.9352	0.016729094	0.00	27.3605
S_Proc_21.Queue	20.8805	0.658842287	0.00	535.00
S_Proc_22.Queue	2.6533	0.066386407	0.00	120.85

Resource

Usage

Instantaneous Utilization	Average	Half Width	Minimum Value	Maximum Value
Net_Server	0.8432	0.002719090	0.00	1.0000
SP_11	0.5847	0.002202927	0.00	1.0000
SP_12	0.1764	0.001259754	0.00	1.0000
SP_21	0.6604	0.004208187	0.00	1.0000
SP_22	0.2449	0.001805572	0.00	1.0000

User Specified

Usage

None	Average	Half Width	Minimum Value	Maximum Value
BE_Response Time	46.2229	0.816577528	2.5480	371.90
BS_Response Time	70.8880	1.18056	3.5720	593.07
GC_Response Time	39.5215	0.638814994	3.5491	358.48
NR_Response Time	61.5816	0.809434824	3.5720	326.42
VC_Response Time	41.9405	0.807494728	4.5585	276.65

Figure 14.35 Simulation results for the Arena model of the bookseller's e-business system.

The utilizations of server processes *SP_11* through *SP_22* vary widely. Note that each first server process at each server node (*SP_11* and *SP_21*) is much busier (over 58% utilization) than the second process in that node (*SP_12* and *SP_22*). This is due to the fact that service types with longer elapsed times are allocated to the first server process at each server node.

Delays in server process queues are fairly proportional to the corresponding server process utilization. The average delay in the queue of server process *SP_21* is about 21 milliseconds, while the average delay in server process *SP_22* is a mere 2.65 milliseconds. These delays directly affect the response times of customer requests. For example, best seller (BS) and new releases (NR) services have longer elapsed times than the other services. Consequently, the server processes providing these services are the busiest, and the corresponding response times (about 71 and 62 milliseconds, respectively) are significantly larger than those of the other services.

This bookseller's e-business example clearly shows the importance of workload allocation in client/server systems, and especially in mission-critical applications. Delays in server process queues and transmission network queues constitute major components of response times. Clearly, it is desirable to minimize these delays in order to reduce overall response times. A more balanced allocation of workload to servers would help reduce server process delays, and in turn reduce response times. On the other hand, to improve the transmission network performance, one may have to upgrade the network by adding additional equipment (routers, switches, etc.) to reduce network delays—a far more expensive proposition.

Exercises

1. *E-commerce and supply chains*. Consider BlueSky Books, a marketing and publishing company of a popular book (only one copy per customer order). The marketing activity centers on the website *BlueSky.com* operating 24 hours a day, which receives orders from customers and passes them on to the publisher. The publisher prints and maintains a small inventory of the book, and uses it to fulfill customer orders. Book requests are generated at a client host, and arrive with iid exponential interarrival times of rate 10,000 units per day. The CPU time per book order at the client node is exactly 1 second, and the size of each request message is precisely 2048 bytes. Book requests are sent to the server host through a communications (transmission) network with BWC of 2250 bytes/second and MTE of only 60%. The server host has a single server process that receives all book requests and processes them. The elapsed time per request is 4 seconds and includes charging the customer's credit card, transferring the proceeds to a bank account, and updating a local database with order information. On completion of processing, the server sends two messages of size 1024 bytes each over the transmission network—an acknowledgment message to the client and an order message to the publisher.

   The BlueSky publishing division has one press on its shop floor, and is guaranteed to have sufficient supplies of ink and paper on hand. The press can operate 24 hours a day and never experiences failures. The time it takes to print

a batch of 3000 books is iid Tria(2.5, 4, 6) hours. As soon as the publishing division receives an order from the website, it ships the order to the customer (and records an inventory reduction of one unit). The publishing division maintains a maximal inventory of 15,000 books, but when it drops down to 3000 books, the press is activated to replenish the inventory to 15,000 books, at which point the press is idled again. The shipping time for a book delivery is iid Unif(1, 3) days. Unsatisfied book orders are backordered. If there are outstanding backorders, they are satisfied first as soon as a new production batch is completed. The remaining books, if any, are added to the on-hand inventory.

a. Develop an Arena model for the BlueSky operation and simulate it for 15 days, starting with an initial inventory of 1000 books on hand.
b. Estimate the following statistics:
   - Average acknowledgment response time (the time a customer waits for an acknowledgment starting at the moment the order is submitted)
   - Average total response time (the time it takes the book to arrive at the customer's address from the moment of order submission), but only for customers whose demand was satisfied immediately, without a need for backordering
   - Average inventory level of books at the publishing division
   - Average backorder level

Hint: If you use a *Delay* module for the Unif(1, 3) days to delivery, then the student version of Arena might be inadequate, due to too many transaction entities in the model. Instead, include this (random) time in an expression of the total response time to be tallied (the expression should be entered into the *Tally* module for parsimony).

2. *Three-tier architecture in e-banking*. The First New Brunswick Savings (FNBS) bank has introduced e-banking to its customers. The e-banking infrastructure consists of a cluster of two server nodes linked to each other via a TP monitor, which dynamically balances the workload of each server in the cluster. The processing time at the TP monitor is 0.05 seconds. The e-banking program offers a number of services to its customers, as follows: account identification (AI), view account summary (VS), view last 15 withdrawals (VW), view last 15 deposits (VD), and view interim account report (VR). The elapsed times of services are server-node dependent. These are given in following table along with request priorities.

Request Type	Request Priority	Elapsed Time on Server Node 1 (seconds)	Elapsed Time on Server Node 2 (seconds)
AI	1	Unif(4, 8)	Unif(12, 16)
VS	2	Unif(10, 18)	Unif(15, 27)
VW	3	Unif(12, 22)	
VD	4	Unif(8, 15)	Unif(12, 16)
VR	5	Unif(15, 30)	Unif(22, 45)

Interarrival times of requests are iid exponentially distributed with a mean of 10 seconds. The probabilities of service request types arriving from any node are given in the next table.

Request Type	Probability of Request Type
AI	0.10
VS	0.15
VW	0.20
VD	0.20
VR	0.35

Once a service request is executed, a reply is sent back to the client. Request message sizes are fixed at 2048 bytes, whereas reply message sizes are iid Disc ({(0.3, 1024), (0.3, 2048), (0.4, 8096)}). Messages between client and server nodes are transported via a transmission network with a BWC of 2500 bytes per second and MTE of 70%. There are two server processes at each server node. The allocation of request types to server processes at each node is given in the following table.

Node/Server Process	Service Types
Node 1 / SP_1	AI, VS
Node 1 / SP_2	VW, VD, VR
Node 2 / SP_1	AI, VS, VW
Node 2 / SP_2	VD, VR

Server processes implement priority queueing disciplines with smaller priority values of requests given higher priority.

a. Develop an Arena model for the FNBS bank's e-banking system and simulate it for an 8-hour period.
b. Estimate the following statistics:
   - Server process utilizations
   - Transmission network utilization
   - Average pooled network delay
   - Average response time for each service request type
   - Average delays at server process queues

3. *Server node processes*. Consider a server node housing six server processes. Service requests arrive with iid exponentially distributed interarrival times of process-dependent rates. The CPU busy time (total time the CPU is busy processing a request) is a process-dependent constant. The following workload profile gives the arrival rates and service times for each server process.

Server Process	Arrival Rate (requests per second)	CPU Busy Time (seconds)
SP_1	5	0.06
SP_2	4	0.12
SP_3	8	0.02
SP_4	10	0.06
SP_5	12	0.01
SP_6	5	0.02

The server node houses two CPUs and a single CPU queue in front of them. As soon as service starts for a request at a server process, the request transaction

moves into the CPU queue, and after a possible delay in the queue, it is randomly dispatched to any of the available CPUs (all available CPUs are equally likely to be selected). Note that the server process does not start processing a new transaction until the processing of the current one is completed. The service discipline is round robin, that is, transactions at the CPU queue are served one time-quantum at a time. Assume that any service request requires only CPU processing. If the CPU busy time is completed within a quantum, then the transaction relinquishes the CPU and leaves the server process at the end of the last quantum. Otherwise, the transaction goes back to the end of the queue and waits for its next service quantum. Assume a service quantum of 0.005 seconds. Note that since there are six server processes and two CPUs, there can be at most four transactions waiting in the CPU queue.

a. Develop an Arena model for the server node, and simulate it for 1 hour.
b. Estimate the following statistics:
   - Server process utilizations
   - CPU utilization
   - Average waiting times in server process queues and the CPU queue
   - Average response time per server process (time interval from arrival at the server process until service is completed)

Hint: You can use the Arena *Set* module for resources and queues to build a concise Arena model.

This page intentionally left blank

Appendix A

Frequently Used Arena Constructs

This appendix provides a brief synopsis of the most frequently used Arena programming constructs (attributes, variables, and modules). For detailed information consult the Arena documentation and *Help* menu.

Throughout this appendix, we use the following notational conventions:

- ***Bold italic*** is used for Arena reserved keywords.
- *Normal italic* is used for syntactic elements to be replaced by user-provided names.
- **Bold** is used for Arena literals.
- Square brackets, [ ], denote optional items.
- Accessibility is denoted by **RO** (read-only), **RW** (read-write), and **W** (write).

A.1 FREQUENTLY USED ARENA BUILT-IN VARIABLES

Following is a subset of frequently used Arena built-in variables and their accessibility. For a complete list, refer to the Arena documentation and ***Help*** menu.

A.1.1 Entity-Related Attributes and Variables

- *Attribute Name* [(*Index 1, Index 2*)]: (**RW**) value of the entity attribute (if an array, then the parameters specify the element indices).
- *Entity.**Type*** [(*Entity Number*)]: (**RO**) type (or name) of a specific entity.
- *Entity.**Picture*** [(*Entity Number*)]: (**W**) entity's graphic (picture) to be used in animation.
- *Entity.**SerialNumber*** (**RO**): entity's automatically assigned identifier.
- *Entity.**Sequence*** [(*Entity Number*)]: (**RW**) station location number to which an entity is transferred next in a sequence.
- *Entity.**Jobstep*** [(*Entity Number*)]: (**RW**) index into entity's sequence.
- *Entity.**CurrentStation***: (**RO**) entity's current station location.
- ***AG*** (*Rank, Attribute Number*): (**RO**) value of attribute *Attribute Number* of the entity with the specified *Rank* in the active entity's group.
- ***NG*** [(*Entity Number*)]: (**RO**) number of entities in the group of representative *Entity Number*.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

A.1.2 Simulation Time Variables

- ***TNOW***: (**RW**) current simulation time.
- ***TFIN***: (**RW**) final simulation time.

A.1.3 Expressions

- ***ED*** (*Expression Number*): (**RO**) value of an expression specified by *Expression Number*.
- ***EXPR*** (*Expression Number* [*Index 1, Index 2*]): (**RO**) value of an expression specified by *Expression Number* and its indices.
- *Expression Name* [(*Index 1, Index 2*)]: (**RO**) value of an expression specified by its expression name and its indices.

A.1.4 General-Purpose Global Variables

- *Variable Name* [(*Index 1, Index 2*)]*:* (**RW**) value of a variable specified by *Variable Name* and its indices.

A.1.5 Queue Variables

- ***NQ*** (*Queue ID*): (**RO**) number of entities in the queue *Queue ID*.
- ***AQUE*** (*Queue ID, Rank, Attribute Number*): (**RO**) value of the attribute indexed by *Attribute Number* of the entity at the specified *Rank* in queue *Queue ID*.

A.1.6 Resource Variables

- ***MR*** (*Resource ID*): (**RW**) capacity units of the resource indexed by *Resource ID*.
- ***NR*** (*Resource ID*): (**RO**) number of busy units in the resource indexed by *Resource ID*.
- ***RESUTIL*** (*Resource ID*): (**RO**) instantaneous utilization (0, if ***NR***=0; 1, if ***NR***≥***MR***; ***NR***/***MR***, otherwise) of the resource indexed by *Resource ID*.
- ***STATE*** (*Resource ID*): (**RO**) current state (-1, if Idle; -2, if Busy; -3, if Inactive; -4, if Failed) of the resource indexed by *Resource ID*.

A.1.7 Statistics Collection Variables

- ***NC*** (*Counter ID*)): (**RO**) current value of the counter specified by *Counter ID*.
- ***DAVG*** (*Dstat ID*): (**RO**) average value of the dstat specified by *Dstat ID*.
- ***DMAX*** (*Dstat ID*): (**RO**) maximum value of the dstat specified by *Dstat ID*.
- ***DMIN*** (*Dstat ID*): (**RO**) minimum value of the dstat specified by *Dstat ID*.
- ***DSTD*** (*Dstat ID*): (**RO**) standard deviation of the dstat specified by *Dstat ID*.
- ***DTPD*** (*Dstat ID*): (**RO**) time period over which the dstat specified by *Dstat ID* has been collected.
- ***DHALF*** (*Dstat ID*): (**RO**) 95% confidence interval of the dstat specified by *Dstat ID*.

- ***DVALUE*** (*Dstat ID*): (**RO**) last recorded value of the dstat specified by *Dstat ID*.
- ***TAVG*** (*Tally ID*): (**RO**) average value of the tally specified by *Tally ID*.
- ***TMAX*** (*Tally ID*): (**RO**) maximum value of the tally specified by *Tally ID*.
- ***TMIN*** (*Tally ID*): (**RO**) minimum value of the tally specified by *Tally ID*.
- ***TNUM*** (*Tally ID*): (**RO**) number of values of the tally specified by *Tally ID*.
- ***TSTD*** (*Tally ID*): (**RO**) standard deviation of the tally specified by *Tally ID*.
- ***THALF*** (*Tally ID*): (**RO**) 95% confidence interval of the tally specified by *Tally ID*.
- ***TVALUE*** (*Tally ID*): (**RO**) last recorded value of the tally specified by *Tally ID*.

A.1.8 Transporter Variables

- ***IT*** (*Transporter ID, Unit Number*): (**RO**) status indicator (0, if Idle; 1, if Busy; 2, if Inactive) of the unit specified by *Unit Number* in the transporter specified by *Transporter ID*.
- ***MT*** (*Transporter ID*): (**RO**) number of active units in the transporter specified by *Transporter ID*.
- ***NT*** (*Transporter ID*): (**RO**) number of busy units in the transporter specified by *Transporter ID*.
- ***VT*** (*Transporter ID*): (**RW**) velocity of all units in the transporter specified by *Transporter ID*.

A.1.9 Miscellaneous Variables and Functions

- ***NSTO*** (*Storage ID*): (**RO**) number of entities stored in the storage specified by *Storage ID*.
- ***J***: (**RO**) search index variable.
- ***TF*** (*Table ID, X Value*): (**RO**) value of the variable *X Value* stored in the table specified by *Table ID*.

A.2 FREQUENTLY USED ARENA MODULES

Following is a subset of frequently used Arena modules, the associated template panel, and a brief explanation of module function and operation. The term *parameter* stands for the contents of the corresponding field in the module dialog box. For a complete list, refer to the Arena documentation and *Help* menu.

A.2.1 *Access* Module (*Advanced Transfer*)

Function: Used to allocate one or more cells of a conveyor to an entity for movement from one station to another.

Operation: When an entity arrives at this module, it waits until the appropriate number of contiguous cells on the conveyor become empty and align with the entity's station location. Once this condition is satisfied and the entity gains control of the requisite cells on the conveyor, it may be conveyed to the next station.

A.2.2 *Assign* Module (*Basic Process*)

Function: Used to assign values to variables, entity attributes, entity types, entity pictures, and other system variables.

Operation: Whenever an entity enters this module, one or more assignments are executed. An assignment can be made to entity attributes, entity type or entity picture and/or to global variables or other system variables. After new values are assigned, all entities exit the module from a single exit point. Assignments are added by filling out the sub-form ***Assignments***, which pops up on clicking the ***Add*** button on the module form.

A.2.3 *Batch* Module (*Basic Process*)

Function: Used to group entities into batches.

Operation: Entities arriving at the ***Batch*** module are placed in a queue until the required number of entities has accumulated. Once accumulated, the entities are grouped and replaced by a new representative entity, which inherits its attributes from batch members according to a rule specified in the ***Save Criterion*** parameter. The representative batch exits the module from a single exit point. Batches can be permanent or temporary.

A.2.4 *Create* Module (*Basic Process*)

Function: Used as a source to generates new entities, and release them into the model.

Operation: Entities are created using a schedule or based on inter-arrival times. Once created, the entities leave the module.

A.2.5 *Decide* Module (*Basic Process*)

Function: Used as a decision point in the model.

Operation: When an entity arrives at this module, a decision is made based on one or more conditions (deterministic outcome) or by chance (random outcome). The entity then leaves the module at an exit point, which is determined by the outcome. Conditions are based either on attribute values, or variable values, or expressions, or the entity type. When the value of parameter ***Type*** is **2-way by Chance** or **2-way by Condition**, the model has two exit points: one for **true** outcomes (located at the right side of the module) and one for **false** outcomes (located at the bottom of the module). When the value of parameter ***Type*** is **N-way by Chance** or **N-way by Condition**, there is an exit point for each outcome. An entity leaves the module from the computed exit point.

A.2.6 *Delay* Module (*Advanced Process*)

Function: Used to delay an entity by a specified amount of time.

Operation: When an entity arrives at this module, the time expression defined in the ***Delay Time*** parameter is evaluated and the entity remains in the module for that

time period. The time is then allocated to the entity's value added, non-value added, transfer, wait or other time as specified in the ***Allocation*** parameter.

A.2.7 *Dispose* Module (*Basic Process*)

Function: Used as the exit point of entities from a simulation model.

Operation: Entities arriving at this module are disposed of and removed from the model. Entity statistics may be recorded before the entity is disposed of by checking the ***Record Entity Statistics*** checkbox.

A.2.8 *Dropoff* Module (*Advanced Process*)

Function: Used to remove a specified number of entities from an entity's group and to send them to the next module in the model.

Operation: When an entity group arrives at this module, the specified number of the entities in the group, starting form the rank defined in the ***Starting Rank*** parameter, are removed from the group and sent to the module specified by model connections. The value of the user-defined group attributes and internal attributes of the representative entity of the group (designated at group formation time) may be copied to the dropped off entities based on a rule specified in the ***Member Attributes*** parameter. There are two exit points from this module. The original group of entities exit to the right of the module, while the dropped off entities exit at the bottom of the module.

A.2.9 *Free* Module (*Advanced Transfer*)

Function: Used to release the entity's most recently allocated transporter unit.

Operation: When the entity enters this module, it releases its most recently allocated transporter unit. If another entity is waiting in a queue to request or allocate the transporter, that transporter will be allocated to that entity. If there are no waiting entities at the time the transporter unit is freed, the transporter will wait idle at the freeing entity's station location, unless otherwise specified in the ***Transporter*** module.

A.2.10 *Halt* Module (*Advanced Transfer*)

Function: Used to change the status of a transporter unit to inactive.

Operation: When an entity enters this module it tries to halt the requisite transporter unit. If that unit is idle at the time, then its status is set immediately to inactive. If, however, that unit is busy at the time, then its status changes immediately to busy and inactive. If later on the entity that controls the halted unit proceeds to free it, then its status changes to inactive only at that point in time. Once a transporter unit has been halted, no entities can gain control of that unit until it is activated.

A.2.11 *Hold* Module (*Advanced Process*)

Function: Used to hold an entity in a queue to either wait for a signal, wait for a specified condition to become true, or to be held indefinitely.

Operation: When an entity arrives at this module and the value of the ***Type*** attribute is **Wait for Signal**, then a ***Signal*** module must be used to send the requisite signal that allows the entity to move on to the next module. If the value of the ***Type*** attribute is **Scan for Condition**, then the entity will remain at the module until the condition(s) defined in the ***Condition*** parameter become(s) true. On the other hand, if the value is **Infinite Hold**, then the hold period is indefinite, unless a ***Remove*** module is used to allow the entity to continue processing. The waiting queue for the entities can be specified in the ***Queue Type*** attribute as the module's internal queue or another queue.

A.2.12 *Match* Module (*Advanced Process*)

Function: Used to bring together (synchronize) a specified number of entities waiting in different queues at this module.

Operation: An entity arriving at this module is placed in one of up to five associated queues, based on its entry point, and remains in its queue until a match materializes (a match may be accomplished when there is at least one entity in each queue). At that point in time, one matching entity from each queue is removed, and all these entities exit the module simultaneously via their respective exit point. All exit points must be connected to some modules.

A.2.13 *PickStation* Module (*Advanced Transfer*)

Function: Used to select a particular station.

Operation: When an entity arrives at this module, a station is selected from a station group, based on the selection logic specified in the ***Selection Based On*** . . . section. The entity may then route, transport, convey, or connect to the selected station depending on the value of the ***Transfer Type*** parameter. The station selection process is based on the minimum or maximum value of a variety of system variables and expressions depending on the value of the ***Test Condition*** parameter.

A.2.14 *Pickup* Module (*Advanced Process*)

Function: Used to remove a number of consecutive entities from a given queue.

Operation: When an entity group arrives at this module, it removes a specified number of entities from a specified queue starting at a specified rank in the queue. The picked up entities are added to the end of the incoming entity group.

A.2.15 *Process* Module (*Basic Process*)

Function: Used as the main processing method for various functions such as delaying, seizing, and releasing resources and queuing.

Operation: Entities arriving at this module are processed differently, depending on the specified value for the ***Action*** parameter. Additionally, the user can choose the **submodel** option as the ***Type*** parameter, and specify a hierarchy of user-defined submodels and their logic. Process times and associated costs are allocated to incoming entities using the ***Allocation*** parameter, including **Value Added**, **Non-Value Added**, **Transfer**, **Wait**, and **Other**. All entities exit this module from a single exit point.

A.2.16 *ReadWrite* Module (*Advanced Process*)

Function: Used to read or write data from/to a specified source.

Operation: When an entity arrives at this module, data is read from an input file or the keyboard, or data values are assigned to a list of variables, attributes, or other expressions. The data can also be written to an output device, such as the screen or a file. When reading from or writing to a file, the ***ReadWrite*** logic varies according to the ***Type*** parameter for the ***Arena File Name*** parameter.

A.2.17 *Record* Module (*Basic Process*)

Function: Used to collect statistics in a particular location in the model.

Operation: When an entity arrives at this module, a single user-specified statistic (count or tally) is collected. The statistic type is selected in the ***Type*** parameter. Once the requested statistic is collected, the entity exits from a single exit point of the module.

A.2.18 *Release* Module (*Advanced Process*)

Function: Used to release units of a resource previously seized by an entity.

Operation: When an entity arrives at this module, it gives up control of resource unit(s) from a specified resource. Any entities waiting in queues for those resources will then contend for control of the released units.

A.2.19 *Remove* Module (*Advanced Process*)

Function: Used to remove a single entity from a specified position in a queue, and to send it to a designated module.

Operation: When an entity arrives at this module, it removes a specified entity from a specified queue and sends the removed entity to the next module. The removed entity is selected based on the entity's rank (position in the queue). The removed entity exits the module at the exit point labeled **Removed Entity** on the module icon, while the removing entity exits the module at the exit point labeled **Original** on the module icon. The removing entity is processed before the removed entity.

A.2.20 *Request* Module (*Advanced Transfer*)

Function: Used to assign a transporter unit to an entity and then to move the unit to the entity's location (in order to transport that entity).

Operation: When an entity arrives at this module, it is allocated a transporter unit (if none is available, the entity waits in this module until one becomes available). Once a transporter unit is allocated to the entity, that entity waits in this module until the transporter unit reaches the entity's location (specified in its ***Station*** attribute), and then the entity exits the module. A specific transporter unit may be defined using the ***Transporter Name*** parameter, or the selection may occur based on a rule in the ***Selection Rule*** parameter.

A.2.21 *Route* Module (*Advanced Transfer*)

Function: Used to transfer an entity to a specific station or the next station in the station sequence defined for that entity.

Operation: When an entity arrives at this module, its ***Station*** attribute is set by this module to a destination station. The entity is then sent to this destination station, and will arrive there after the time period specified in the ***Route Time*** parameter. If the value of the ***Destination Type*** parameter is **Sequential**, then the next station is determined by the entity's sequence and step within the sequence.

A.2.22 *Search* Module (*Advanced Process*)

Function: Used to search over entities in a queue or a batch, or an expression over a range of indices, and to return in the global variable ***J*** the first index for which the specified condition is true. In the first case, ***J*** returns the entity rank (in the queue or batch), while in the second case, ***J*** returns the first index for which the specified expression evaluated to true.

Operation: When an entity arrives at this module, the global variable ***J*** is set to a starting index and the search condition is then evaluated. If the search condition is satisfied, the search ends and the current value of ***J*** is retained. Otherwise, the value of ***J*** is incremented (or decremented) and the condition is re-evaluated. This process repeats until either the search condition is satisfied or the ending value is reached, in which case ***J*** is set to 0.

A.2.23 *Seize* Module (*Advanced Process*)

Function: Used to allocate units of one or more resources to an entity.

Operation: When an entity enters this module, it waits in a queue until all specified resources are available simultaneously. The entity can seize units of a particular resource or units of a member of a resource set.

A.2.24 *Separate* Module (*Basic Process*)

Function: Used to separate and recover the original members of a temporary batch of entities previously grouped in a ***Batch*** module. Also used to duplicate entities.

Operation: When a temporary batch entity enters this module, batch members are recovered and depart sequentially, whereas the temporary batch entity is disposed of.

The simulation clock is not advanced while batch members depart from this module. When used for entity duplication purposes, all entities inherit the incoming entity's attributes and leave the module before it.

A.2.25 *Signal* Module (*Advanced Process*)

Function: Used to send a signal to each ***Hold*** module in the model where the value of the ***Type*** parameter is **Wait for Signal**, in order to release the maximum specified number of entities.

Operation: When an entity arrives at this module, the ***Signal Value*** parameter of the module is evaluated as a signal code, which is then sent to each ***Hold*** module in the model in which the value of the ***Type*** parameter is **Wait for Signal**. On receipt of the signal, entities at ***Hold*** modules that are waiting for that signal are removed from their queues. The entity sending the signal then exits the module.

A.2.26 *Station* Module (*Advanced Transfer*)

Function: Used to define a station or a set of stations corresponding to a physical or logical location where processing occurs.

Operation: An entity arrives at this module directly from any of the modules where the entity transfer is initiated, even if the latter modules are not connected to the ***Station*** module. The entity may trigger statistics collection before exiting the module.

A.2.27 *Store* Module (*Advanced Process*)

Function: Used to add an entity to storage.

Operation: When an entity arrives at this module, the storage level is incremented, and the entity immediately moves to the next module in the model. The ***Unstore*** module may then be used to remove the entity from the storage.

A.2.28 *Transport* Module (*Advanced Transfer*)

Function: Used to transfer an entity controlling a transporter unit to a destination station.

Operation: When an entity arrives at this module, its ***Station*** attribute is set to the entity's destination station. The entity is then transported on a specified transporter unit to a destination station. After the time delay required for the transport elapses, the entity reappears in the model at the destination ***Station*** module.

A.2.29 *Unstore* Module (*Advanced Process*)

Function: Used to remove an entity from storage.

Operation: When an entity arrives at this module, the specified storage level is decremented and the entity immediately moves to the next module in the model.

A.2.30 *VBA* Block (*Blocks*)

Function: Used to insert VBA (Visual Basic for Applications) procedural user code into the model. The code is entered via the **Visual Basic Editor**.

Operation: For a ***VBA*** block with ID number *N*, the user provides VBA code for the corresponding ***VBA_Block_N_Fire*** event. When an entity enters a ***VBA*** block, Arena fires the corresponding event to execute the user-provided VBA code. Following execution of the event code, the entity exits the ***VBA*** block and proceeds to traverse the model in the usual way. For more information, see Appendix B.

Appendix B

VBA in Arena

Arena interoperates with two Microsoft technologies that are designed to enhance desktop applications. The first, *Active X Automation*, provides control facilities within and across applications via a framework platform that is accessible through a programming language. The second technology, *Visual Basic for Applications* (*VBA*) is a programming language, available in the Arena environment for writing procedural code to supplement Arena's visual programming facilities. Procedural programming implements complex algorithms far more efficiently than visual programming, but the latter is easier to use. Using VBA, the user can also exploit the integrative capabilities afforded by Microsoft technologies, and readily interact with such popular applications as Excel, Visio, etc., which are supported by Active X Automation.

The VBA programming environment can be accessed in Arena from the *Tools* menu by selecting the *Macro* option and then the *Show Visual Basic Editor* option. This action opens a separate window that hosts the VBA code, forms, debugging interface, and access to help. Here, the user can write various types of VBA code:

1. The user can provide event processing code that is executed in response to the occurrence of Arena events.
2. The user can also provide general-purpose code that executes in special Arena blocks, called *VBA* blocks. The VBA code in a *VBA* block is executed whenever an entity enters the block. Following execution, the entity exits the *VBA* block and proceeds to traverse the model in the usual way.
3. Finally, the user can write code to handle external events that occur in the form of real-time messages arriving at the Arena simulation across a network.

VBA blocks can be dragged from the *Blocks* template panel to the Arena canvas in the usual way. Double clicking a *VBA* block opens the *Visual Basic Editor*. The user uses this editor to write procedural code including event implementation. An *event* is a special routine that is called (executed) when a particular condition (event) occurs. To identify individual *VBA* blocks, Arena assigns them unique ID values as consecutive integers starting from 1. Accordingly, the event that is executed when an entity enters *VBA* block `N` is consistently named `VBA_Block_N_Fire`, where `N` is the ID number assigned by Arena to that *VBA* block. The distinctive font of the aforementioned event name will be used throughout this appendix to identify VBA-related code fragments.

This appendix overviews the salient features of Arena's VBA programming facilities, and provides some illustrative examples. It does not review VBA; rather, the reader is assumed to be sufficiently proficient in this programming language. For more information, the reader is referred to Kelton et al. (2004) and to Arena's *Help* facilities.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

B.1 ARENA'S OBJECT MODEL

An *object* is a self-contained programming construct consisting of two fundamental types of members, called *properties* and *methods*.

1. A *property* is an attribute of the object. Properties are stored in object fields but may be subject to access rules (read-only, write-only, or read-write). For example, an object may have a *Color* property, whose value is a color.
2. A *method* is an action that the object can perform. Methods may or may not have a return value. For example, a bank account object has *Deposit* and *Withdraw* methods that can change the bank account balance.

In *Object-Oriented Programming*, an object with properties and methods is said to belong to a *class*. An object is an instance of a class, while the corresponding class represents an abstraction of its instances, which all share the same definition of properties and methods. Each object has a *type*, and may belong to *collections*, which are ordered groupings of objects of the same type. The position of any object within a collection may change when an operation on the collection is carried out.

This overview will make use of the following categories of Arena objects:

1) *Model Window Objects*: this category consists of all items that can be placed or viewed on the Arena canvas (e.g., modules, animation objects, connectors, etc.).
2) *SIMAN objects*: this category provides access to data associated with simulation runs (e.g., variables, resource capacities, the simulation clock).

B.2 ARENA'S TYPE LIBRARY

Arena includes a type library (*Arena Type Library*) that contains descriptions of the Arena constructs, such as properties, methods, events, and constants, all of which are accessible through VBA. Such accessible information is said to be *exposed* to users and programs. The information in a type library serves as input to object browsers supplied by Microsoft® VBA and other development environments. The user can use such an object browser to view the descriptions for application installed on the system, Arena included. The *Visual Basic Editor* includes an *Object Browser* tool among other facilities.

To use the *Arena Type Library*, it must be referenced by the programming environment. In particular, the VBA code of an Arena document automatically references the *Arena Type Library*. The user references a library in the *Visual Basic Editor*, as follows:

1) Select the *References* option from the *Tools* menu.
2) In the *Available References* list, check the requisite library (e.g., *Arena nn.n Type Library*), and then click *OK*, where *nn.n* denotes an Arena version number.

Note: Arena automatically registers its type library with Windows the first time it runs. If you do not see the Arena type library in the *Available References* list, exit Arena and follow the steps again.

B.2.1 Resolving Object Name Ambiguities

VBA programs can reference multiple type libraries, which might sometimes declare items with the same name. For example, both Arena and Microsoft® Excel expose an object called `Application`. The resulting ambiguities must, of course, be resolved. Accordingly, VBA binds an ambiguous name to the library with the higher priority as defined in the *Project/Library* list in the *Object Browser*. The user can control the resolution procedure by changing the priority of libraries in the *Object Browser*, or by prefixing object types with the requisite library name as illustrated below.

```
Dim arenaObj As Arena.Application
Dim excelObj As Excel.Application
```

For a list of Arena object types, refer to the *Class* names in the *Classes* list of the *Object Browser*. If user code runs exclusively in the context of a VBA project associated with an Arena document, there is no need to prefix the names of Arena object types with `Arena`, because the Arena type library has a higher priority than other libraries that might declare conflicting names. VBA does not permit changes in the priority of the Arena type library when VBA is used within Arena, though other development environments may permit such priority changes.

B.2.2 Obtaining Access to the *Application* Object

Before VBA code can access objects in the Arena type library, access must be granted to the `Application` object, which represents an instance of Arena. An external program must typically create or retrieve an `Application` object before it can access other Arena objects from that instance. To this end, use the `CreateObject("Arena.Application")` VBA function to run a *new* instance of the Arena application, or the `GetObject("","Arena.Application")` VBA function to retrieve an *existing* instance of Arena. The `ThisDocument` object (to be explained in the following section) is the root object of an Arena model.

B.3 ARENA VBA EVENTS

Recall that a *simulation event* (*event* for short) is a procedure (subroutine) that is called to change the state of the model at a scheduled simulation time (see Chapter 2). Event execution is also referred to as *event calling* or *event firing*. To add VBA code to an event procedure, the user selects the `ModelLogic` object in the *Visual Basic Editor* and then selects the desired event. Arena's built-in VBA events fall into the following categories:

1) Arena-generated events that occur before a simulation run (for example, `DocumentOpen`, `DocumentSave`)
2) Arena-generated events that occur during a simulation run (for example, `RunBeginSimulation`, `RunEndSimulation`)
3) User-generated events that occur during a simulation run (for example, `VBA_Block_Fire`, `OnFileRead`)

Table B.1

Time sequencing of Arena VBA events

<table>
<tr><td>RunBegin</td><td>Simulation run is about to start.
Module data is available.
SIMAN data is not available.</td></tr>
<tr><td>RunBeginSimulation</td><td rowspan="6">Simulation run is in progress.
Simulation run data is available.
SIMAN data is available.</td></tr>
<tr><td>RunBeginReplication</td></tr>
<tr><td></td></tr>
<tr><td></td></tr>
<tr><td>RunEndReplication</td></tr>
<tr><td>RunEndSimulation</td></tr>
<tr><td>RunEnd</td><td>Simulation run is completed.
Module data is available.
SIMAN data is not available.</td></tr>
</table>

Table B.1 displays the time sequencing of these categories of events.

An Arena model can be in one of two states: *edit* state or *run* state. Before a run commences, the system is in the *edit* state. On initiating a run, Arena checks and initializes the model, translates the data supplied in its modules into a format required for simulation, and then sets the model to the *run* state. While running, simulation data (e.g., values of variables, resource states, statistics, etc.) can be accessed and modified via VBA user code. At the end of the run, Arena discards all simulation data and returns the model to the *edit* state.

The following is summary of commonly used Arena VBA events:

1) `ModelLogic_RunBegin` is called prior to checking the model. At this point the model can be modified (e.g., the value of module parameters), and these changes propagate to the simulation run. However, runtime simulation values (e.g., variables and entity attributes) cannot be changed. Arena then checks and initializes the model to the *run* state, but no entities are created as yet.
2) `ModelLogic_RunEnd` is called after the last replication is terminated. The VBA code cannot access any information from the just ended run.
3) `ModelLogic_RunBeginSimulation` is called prior to the first replication and executes exactly once during the simulation run. At this stage the user can read external data from other applications (e.g., Excel, Access). A user form can also be displayed in this event. Furthermore, entities can be created, resource capacities can be altered and variable values can be assigned.
4) `ModelLogic_RunEndSimulation` is called after the last replication is completed. At this point, runtime data (e.g., final statistics values, resource states, etc.) can be accessed via VBA user code. However, the user can code custom statistics and write their values to spreadsheets or databases.
5) `ModelLogic_RunBeginReplication` is called prior to each replication. After this event executes, Arena runs the simulation.
6) `ModelLogic_RunEndReplication` is called after each replication is completed. However, if the run is terminated prematurely, this event is not executed.

7) While an Arena model is running, it allows the user to activate VBA user code in a number of way:
   - `ModelLogic_VBA_Block_Fire` is called when an entity enters a VBA module in the model.
   - `ModelLogic_OnKeyStroke` is called whenever the user strikes a key (except the *Esc* key) during the simulation run.
   - `ModelLogic_OnClearStatistics` is called whenever statistics are reset (e.g., when the simulation completes the warm-up period, or when a replication is completed).

B.4 EXAMPLE: USING VBA IN ARENA

As an example of integrating VBA programming in Arena we will revisit the production/inventory system of Section 12.1. Recall that this model consists of two segments: demand management and production management. This example adds several features to this model: user forms, and writing and reading capabilities to and from Excel. These features facilitate the interaction of the user with an Arena simulation, adding both convenience and flexibility.

The user forms illustrated in this section are displayed whenever the user launches a simulation run or pauses it. The user then gets the opportunity to set or change model parameters for the impending run or when the run is in progress. This is particularly helpful for neophytes who wish to perform *what-if analysis* over multiple scenarios.

Recall that each Arena model includes a `ThisDocument` object embedded in the VBA project. This object provides a fundamental link between the VBA code in a project and the Arena model itself. When referenced from code in the VBA project, the `Model` property of `ThisDocument` returns an object of type `Arena.Model` from the *Arena type library.* In a similar vein, the `SIMAN` property of `ThisDocument` returns an object of type `Arena.SIMAN`.

Typically, the first lines of user code in a VBA event are:

```
Dim m as Arena.Model
Set m = ThisDocument.Model
```

In particular, the model's name is then accessible as `m.Name`. Note, however, that `ThisDocument` is not available to any code that is not part of the VBA project of an Arena model (e.g., external Visual Basic code or Microsoft® Excel macros).

B.4.1 CHANGING INVENTORY PARAMETERS JUST BEFORE A SIMULATION RUN

Suppose the user wishes to have the opportunity to set the reorder point and target level parameters in a production/inventory system just before a simulation run is launched. To this end, the user defines a user form in VBA, say `startForm`. Figure B.1 displays this form as a dialog box consisting of text boxes (fields) for editing the requisite parameters (called here *Reorder Level* and *Target Stock*) and a confirmation button, all in the usual Arena style.

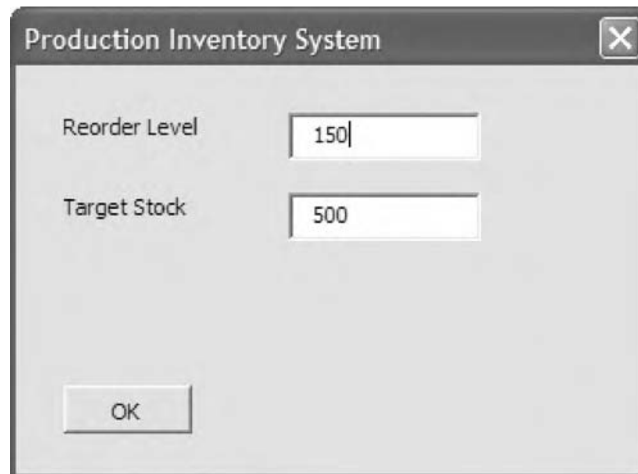


Figure B.1 Dialog box of `startForm`.

In order to fetch the initial parameter values (originally entered in Arena), the VBA user code references an Arena model object (say, `m`) and a SIMAN object (say, `s`). Both of these objects are defined in the `ModelLogic_RunBeginSimulation` event of the `ThisDocument` object, as illustrated by the code fragment below.

```
Dim m As Model
Dim s As SIMAN
Private Sub ModelLogic_RunBeginSimulation()
'RunBeginSimulation is used to access the SIMAN object properties
Set m = ThisDocument.Model
Set s = m.SIMAN
With startForm
   .Reorder_Box.value = s.VariableArrayValue(s.SymbolNumber
    ("Reorder Point"))
   .TargetStock_Box.value = s.VariableArrayValue(s.SymbolNumber
    ("Target Stock"))
   .Cancel_Button.Visible = False
   .Continue_Button.Visible = False
   .Show
End With
End Sub
```

The `SymbolNumber` method of the `SIMAN` object is used to map the name of a model element (e.g., variable, attribute) to its unique runtime integer ID. Since it is more mnemonic to refer to simulation elements by name rather than by ID, the method `SymbolNumber` is used to access Arena elements. While Arena lists its variables in the *VARIABLES* element, it stores them in the array property `VariableArrayValue`. Thus, an Arena variable can be accessed by using its unique integer ID (obtained by method `SymbolNumber`) to index into that array.

B.4.2 Changing Inventory Parameters during a Simulation Run

Suppose the user wishes to have the opportunity to change the reorder point and target level parameters during a simulation run, whenever the *Pause* button is clicked on Arena's *Run* toolbar (or *Run* menu). To this end, the user defines a user form in VBA, say `pauseForm`. Figure B.2 displays this form as a dialog box consisting of text boxes as before, but the `OK` button, is replaced by `Continue` and `Cancel` buttons.

The VBA code fragment following illustrates the use of the events `ModelLogic_RunPause()` and `ModelLogic_RunEnd()` in Arena.

```
Private Sub ModelLogic_RunPause()
'When the user hits the Pause Button
Set m = ThisDocument.Model
Set s = m.SIMAN
With pauseForm
   .Cancel_Button.Visible = True
   .Continue_Button.Visible = True
   .Ok_Button.Visible = False
   .Show
End With
End sub

Private Sub Continue_Button_Click()
Set m = ThisDocument.Model
startForm.hide
m.Go
End Sub
```

Figure B.2 Dialog box of `pauseForm`.

```
Private Sub Cancel_Button_Click()
Set m = ThisDocument.Model
startForm.hide
m.End
End Sub
```

The `End()` method ends the simulation run and returns the model to the *edit* state. Similarly, the `Go()` method checks the model at a non-running state, and then runs or resumes it.

The forms `startForm` and `pauseForm` can be readily expanded by adding to them the capability to set and change the starting inventory and the production batch size. The reader is encouraged to modify the previous forms accordingly.

B.4.3 Changing Customer Arrival Distributions Just before a Simulation Run

Suppose the user wishes to have the opportunity to change the customer inter-arrival distribution in the *Create* module, called *Customer Arrives*, from the uniform distribution Unif(3,7) to some other distribution. To this end, the user defines a user form in VBA, say `distForm`, which accesses this module's properties and changes its *Value* field just before the start of a simulation run.

To access the module, the user right-clicks the *Customer Arrives* module icon, and selects the *Properties . . .* option. This action pops up the module's *Object Properties* dialog box, shown in Figure B.3.

The user then changes the *Tag* field to *Customer Arrival*; *Tag* fields are routinely used to identify and locate specific modules or collections thereof. Accordingly, the following code fragment illustrates the use of the *Tag* field above in identifying the *Customer Arrives* module.

Figure B.3 *Object Properties* dialog box of the *Create* module *Customer Arrives*.

```
Private Sub UserForm_Initialize()
Set m = ThisDocument.Model
Set s = m.SIMAN
Dim createIndex As Long
Dim createModule As Module
'Find the Customer Arrival Module
createIndex = m.Modules.Find(smFindTag, "Customer Arrival")
Set createModule = m.Modules(createIndex)
CustomerArrival_Box.value = createModule.Data("Value")
Reorder_Box.value = s.VariableArrayValue(s.SymbolNumber
("Reorder Point") )
TargetStock_Box.value = s.VariableArrayValue(s.SymbolNumber
("Target Stock"))
End Sub
```

The `Find` method, with the constant `smFindTag` as first argument, returns the index of the first object in a collection of modules whose *Tag* field matches a given *Tag* value. The `createModule.Data` method returns the value of the *Value* field of the module previously retrieved by the `Find` method and the index it returned. A complete listing of Arena module field names may be found in the folder *C:\Program Files\Rockwell Software\Arena 10.0*, which contains a collection of *txt* files (e.g., *BasicProcess.txt*, *AdvancedProcess.txt*, etc.)

Figure B.4 displays the `distForm` user form with the modified customer inter-arrival time distribution in the *Customer Arrives* module (the *Continue* and *Cancel* buttons have been disabled in this example). The customer inter-arrival time distribution has been changed from Unif(3,7) hours to Unif(6,12) hours. To confirm that this change has been accepted, the user may inspect the *Customer Arrives* module dialog box.

Figure B.4 Dialog box of `distForm`.

B.4.3 Writing Arena Data to Excel via VBA Code

This section shows how to use VBA to record information about each customer's arrival time and demand quantity, write it into an Excel spreadsheet and save the spreadsheet as an Excel file. To this end, the original model is modified by inserting a *VBA* block into the demand management segment of the original model. Recall that whenever a customer entity enters a *VBA* block, it fires a VBA event and causes the code contained in that VBA block to execute.

The global declarations section of VBA defines variables that are visible to all procedures. The code fragment below declares global variables of type `Model` and `SIMAN` as well as Excel-related variables pointing to an Excel application, workbook, and worksheet.

```
'Global Arena variables
Dim g_Model As Model
Dim g_SIMAN As SIMAN

'Global Excel variables
Dim oXLApp As Excel.Application
Dim oXLWorkBook As Excel.Workbook
Dim oXLWorkSheet As Excel.Worksheet
```

Since Excel is an application external to Arena, it needs to be linked to Arena via a *reference* to the Excel library (Excel must of course be installed on the same computer). To specify such a reference, the user selects the *References* option from the *Tools* menu in the *Visual Basic Editor* to pop the dialog box of Figure B.5, and then the user checks in it the *Microsoft Excel 10.0 Object Library* item.

The VBA code fragment below demonstrates how to link Excel to Arena.

```
Private Sub ModelLogic_RunBeginSimulation()
Set g_Model = ThisDocument.Model
Set g_SIMAN = g_Model.SIMAN
'Start Excel and create a new spreadsheet
Set oXLApp = CreateObject("Excel.Application")
oXLApp.SheetsInNewWorkbook = 1
Set oXLWorkBook = oXLApp.Workbooks.Add
Set oXLWorkSheet = oXLWorkBook.ActiveSheet
With oXLWorkSheet
   .Activate
   .Name = "Customer_Info"
   .Cells(1,1).value = "Customer Number"
   .Cells(1,2).value = "Arrival Time"
   .Cells(1,3).value = "Demand Quantity"
End With
'Display the UserForm
startForm.Show
End Sub
```

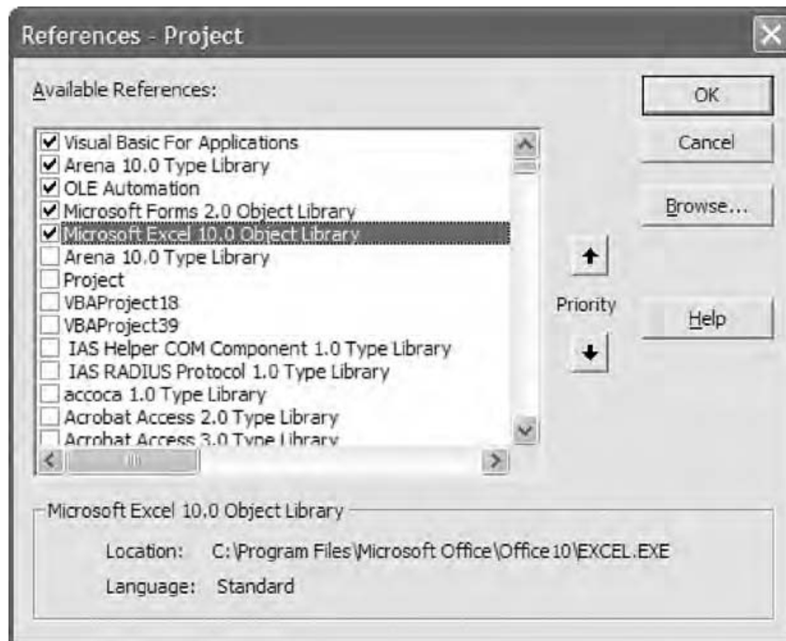


Figure B.5 The *References* dialog box for linking Excel to Arena.

This fragment implements the Arena `ModelLogic_RunBeginSimulation` event in order to interact with an Excel application at the start of a simulation run. The event code first sets global variables to point at the Arena model object and SIMAN object as explained before. It then uses the `CreateObject` method to start a new Excel application, and then creates one spreadsheet (referred to as workbook) and adds it to the worksheet collection using the `oXLApp.WorkBook.Add` method. Finally, the `oXLWorkSheet` variable is set to the active worksheet, and the first three columns of the first row of the Excel worksheet are set to the labels corresponding to the Arena attributes *Customer Number, Arrival Time,* and *Demand Quantity.* Once a simulation run is in progress, each incoming customer entity in the Arena model will have these attributes recorded in a new row of the worksheet, starting with row 2. The workbook itself will be saved as an Excel *.xls* file at the end of the simulation.

While the previous code fragment creates and initializes the Excel file, the recording of customer entity attributes to it occurs dynamically in the course of a simulation run, when each customer entity enters the appropriate *VBA* block (this block was added to the original Arena model). Figure B.6 depicts the modified model with that *VBA* block inserted in the demand management segment of the model logic.

Before recording the *Arrival Time* and *Demand Quantity* attributes of an incoming customer entity in Arena, the user needs to determine the index values of the `Customer_ArrTime` and `DemandQuantity` attributes into global VBA variables using the `SymbolNumber` function as discussed earlier. To this end, the user adds declarations of global variables to store these index values in the code section declaring Arena and Excel global variables, as illustrated next.

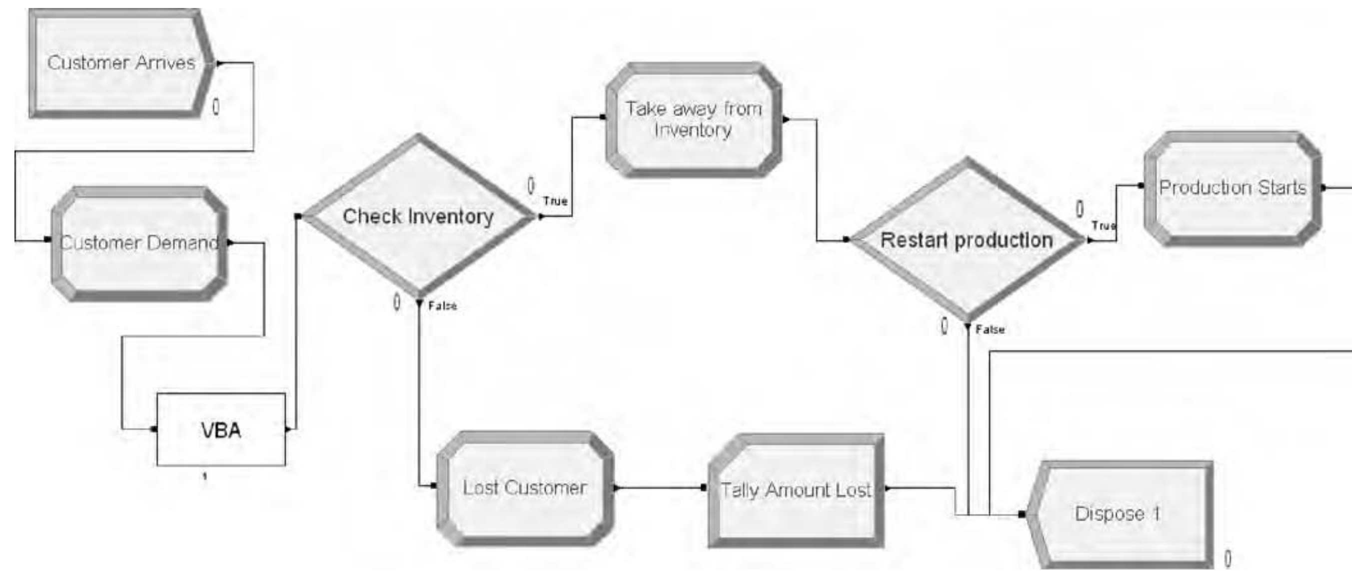


Figure B.6 The modified demand management segment with a *VBA* block for writing Arena data to Excel.

```
'Global Variables
Dim g_CustomerArrTime As Double
Dim g_DemandQ As Double
Dim g_CustomerNum As Long
```

The global variables above will be linked to their Arena counterparts by setting them to the appropriate index numbers, using the `symbolNumber` method as explained in Section B.4.1. To this end, the user should add the following additional VBA code fragment to the `RunBeginSimulation` event.

```
' Set the global variables that store the index
' of the SIMAN variable and attributes
g_CustomerArrTime = g_SIMAN.SymbolNumber("Customer_ArrTime")
g_DemandQ = g_SIMAN.SymbolNumber("Demand")
g_CustomerNum = g_SIMAN.SymbolNumber("Total Customers")
```

Note carefully that the arguments in quotes on the right-hand side above are the actual Arena names of the attributes and variables to be recorded in the Excel file. The actual recording takes place in the `VBA_Block_`*N*`_Fire` event associated with the recording *VBA* block (in our example, the requisite event is `VBA_Block_1_Fire`, since $N = 1$). When a customer entity enters that *VBA* block, it invokes the following VBA code stored in it.

```
Private Sub VBA_Block_1_Fire()
'Retrieve the Customer Arrival Time, Demand Quantity
'and Customer Number
Dim ArrTime, DemandQ As Double
Dim CustNum As Long
CustNum = g_SIMAN.VariableArrayValue(g_CustomerNum)
ArrTime = g_SIMAN.EntityAttribute(g_SIMAN.ActiveEntity,
g_CustomerArrTime)
DemandQ = g_SIMAN.EntityAttribute(g_SIMAN.ActiveEntity,
g_DemandQ)
'Write the values to the spreadsheet
With oXLWorkSheet
   .Cells(CustNum + 1, 1).value = CustNum
   .Cells(CustNum + 1, 2).value = ArrTime
   .Cells(CustNum + 1, 3).value = DemandQ
End With
End Sub
```

Here, `ArrTime` and `DemandQ` are local VBA variables used to retrieve information from the running simulation model via the global SIMAN object `g_SIMAN`. The `EntityAttribute` property of the SIMAN object is used to grant these local variables access to the active customer entity's attribute with index value stored in the global variables `g_CustomerArrTime` or `g_DemandQ`. The corresponding values are then written into Excel using the worksheet object as described earlier. Once the VBA code is executed in the *VBA* block, the customer entity leaves that block and traverses the Arena model in the usual way.

Recall that after the final customer entity is processed, the `RunEndSimulation` event is called. This event affords the user the opportunity to add VBA code to save the Excel file that contains the recorded data, as illustrated in the next VBA code fragment.

```
Private Sub ModelLogic_RunEndSimulation()
'save the spreadsheet
oXLApp.DisplayAlerts = False 'Don't prompt to overwrite
oXLWorkBook.SaveAs g_Model.Path & "Prod Inv System.xls"
oXLApp.Quit
Set oXLApp = Nothing
End Sub
```

Note that the Excel spreadsheet is saved to file *Prod Inv System.xls* in the current directory of the Arena model file, and that the `oXLApp` VBA object that corresponds to the Excel application object is set to `Nothing`, which terminates the Excel application.

B.4.4 Reading Arena Data from Excel via VBA Code

This section shows how to use VBA to read Arena data from Excel spreadsheets. Such data transfer will be illustrated by reading the *Reorder Level* and *Target Stock* parameters into an Arena model from an Excel workbook just prior to the start of a simulation run. To this end, the original model is modified by inserting a second *VBA* block into the inventory management segment of the original model, as shown in Figure B.7.

The user creates an Excel file, say *Model_Input.xls*, and populates a worksheet as shown in Figure B.8. Reading the Excel data is carried out in the `ModelLogic_RunBeginSimulation`, as shown in the VBA code fragment below.

```
Private Sub ModelLogic_RunBeginSimulation()
Set g_Model = ThisDocument.Model
Set g_SIMAN = g_Model.SIMAN
'Open Excel spreadsheet to read values from
Set oXLApp = CreateObject("Excel.Application")
g_ArenaDir = Mid(g_Model.FullName, 1, Len(g_Model.FullName) -
Len(g_Model.Name))
Set oXLWorkBook = oXLApp.Workbooks.Open(g_ArenaDir & "Model_Input.
xls")
Set oXLWorkSheet = oXLWorkBook.ActiveSheet
Set oXLRange = oXLWorkSheet.Range("A2:B2")
'Set the global variables that store the index of the SIMAN variable
g_ReorderPoint = g_SIMAN.SymbolNumber("Reorder Point")
g_TargetStock = g_SIMAN.SymbolNumber("Target Stock")
'Display the UserForm
startForm.Show
End Sub
```

Similarly to the previous example, an Excel application object is created to open the Excel file, and the target workbook and worksheet objects are set up for subsequent

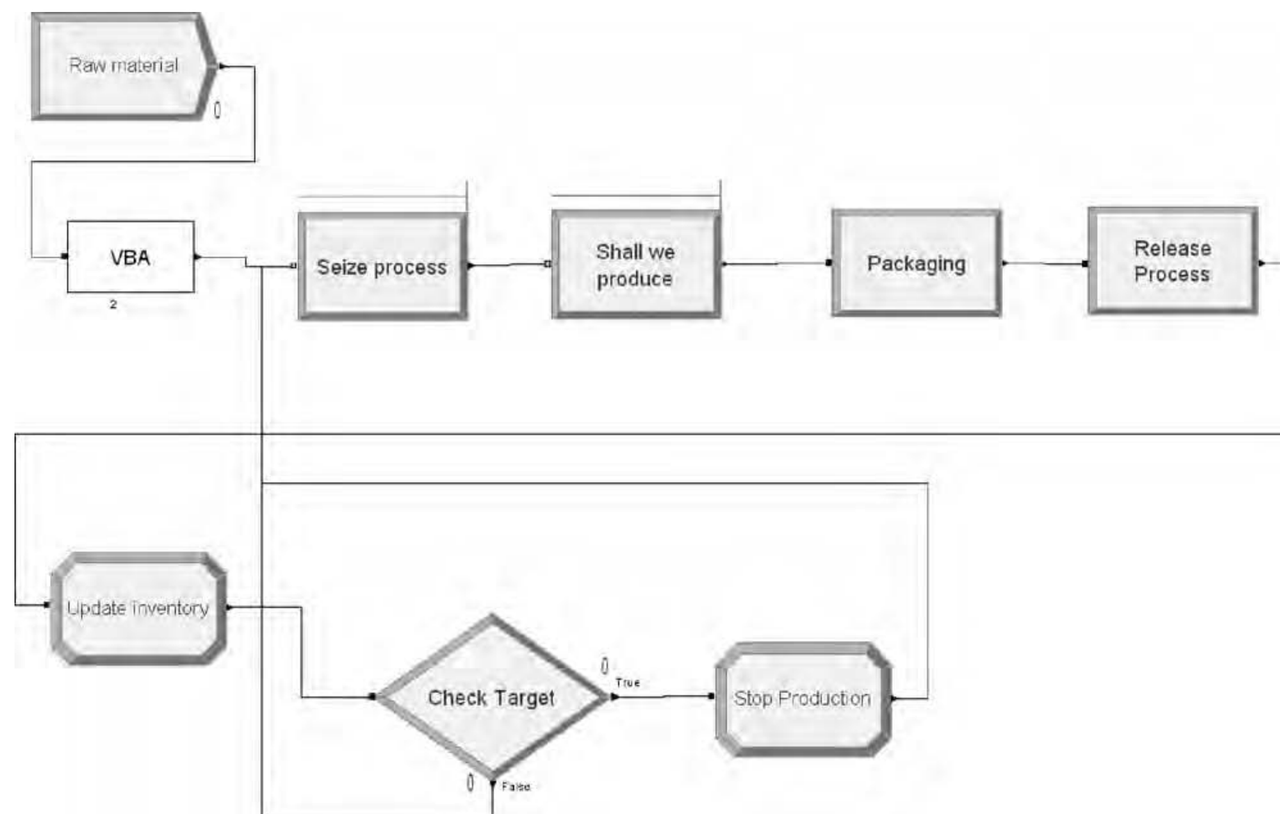


Figure B.7 The modified inventory management segment with a *VBA* block for reading Arena data from Excel.

	A	B
1	Reorder Level	Target Stock
2	100	500

Figure B.8 The Excel worksheet with data to be read into Arena.

reading of parameter values from the requisite cells into SIMAN variables. Here, the code defines a `Range` object, `oXLRange` (not defined in the previous example), where `A2:B2` is the cell address range that stores the parameter values to be transferred to SIMAN variables, using the appropriate index numbers for the *Reorder Level* and *Target Stock* parameters of the Arena model.

The actual reading of these parameter values is carried out in the *VBA* block with ID 2 (the *VBA* block with ID 1 was already assigned in the previous example, and that ID cannot be reused). *VBA* block 2 implements the `VBA_Block_2_Fire` event in the following VBA code fragment.

```
Private Sub VBA_Block_2_Fire()
'Transfer the values into SIMAN variables
g_SIMAN.VariableArrayValue(g_ReorderPoint) = oXLRange.Item(1, 1)
g_SIMAN.VariableArrayValue(g_TargetStock) = oXLRange.Item(1, 2)
End Sub
```

Note that unlike the previous example, the code above is executed precisely once, since only one entity is created in the inventory management segment of the model logic and that entity visits *VBA* block 2 just once. The `Item` property of the `Range` object is used to retrieve one value from a range of cells in the worksheet. Accordingly, `oXLRange.Item(1,1)` retrieves the contents of the cell in the first row and first column of the range, which in this case corresponds to cell `A2` in the Excel worksheet. In a similar vein, `Item(1,2)` corresponds to cell `B2` in the Excel worksheet. Both of these cell values are read into their respective SIMAN variables whose index values were obtained using the global variables `g_ReorderPoint` and `g_TargetStock`.

The code in the `ModelLogic_RunEndSimulation` event is essentially identical to its counterpart in the previous example, and therefore is not reproduced.

References

Alexopoulos, C. and A.F. Seila (1998), "Output Data Analysis," Chapter 7 in *Handbook of Simulation*, John Wiley and Sons, New York.

Altiok, T. (1997), *Performance Analysis of Manufacturing Systems*, Springer-Verlag, New York.

Altiok, T. (1998), "Factors Influencing Vessel Port Time in Bulk Ports," *Bulk Solids Handling*, Vol. 18, No. 1, 27–30.

Altiok, T. and B. Melamed (2001). "The Case for Modeling Correlation in Manufacturing Systems," *IIE Transactions*, Vol. 33, No. 9, 779–791.

Andrade, J., M. Carges, T. Dwyer, and S. Felts (1996), *The Tuxedo System*, Addison-Wesley, Reading, Massachusetts.

Askin, R.G. and C.R. Standridge (1993), *Modeling and Analysis of Manufacturing Systems*, John Wiley and Sons, New York.

ATZ Consultants (1996), *Capacity Expansion Study for Mineração Rio do Norte S.A., Brazil*, Technical Report, ATZ Consultants, New York.

Banks, J., J.S. Carson, B.L. Nelson, and D.M. Nicol (2004), *Discrete-Event System Simulation*, Prentice-Hall, Upper Saddle River, New Jersey.

Belanger, R.F., B. Donovan, K.L. Morse, S.V. Rice, and D.B. Rockower (1989), *MODSIM II Reference Manual*, CACI Products Company, La Jolla, California.

Benson, D. (1996), *Simulation Modeling and Optimization Using PROMODEL*, PROMODEL Corporation, Orem, Utah.

Brateley, P., B.L. Fox, and L.E. Schrage (1987), *A Guide to Simulation*, Springer-Verlag, New York.

Buzacott, J.A. and J.G. Shanthikumar (1993), *Stochastic Models of Manufacturing Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

CACI Products Company (1988), *COMNET II.5 User's Manual*, La Jolla, California.

Cady, J. and B. Howarth (1990), *Computer System Performance Management and Capacity Planning*, Prentice-Hall, Upper Saddle River, New Jersey.

Cario, M.C. and B.L. Nelson (1996), "Autoregressive to Anything: Time-Series Input Processes for Simulation," *OR Letters*, Vol. 19, 51–58.

Çinlar, E. (1975), *Introduction to Stochastic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey.

Comer, D.E. (1997), *Computer Networks and Internets*, Prentice-Hall, Englewood Cliffs, New Jersey.

Cooper R.B. (1990), *Introduction to Queueing Theory*, CEEP Press, Washington, D.C.

Crook, G.C. (1980), "Queueing Theory Data for Port Planning," in *Dock and Harbour Authority*, Foxlow Publications, London.

Dagpunar, J. (1988), *Principles of Random Variate Generation*, Clarendon Press, Oxford.

Devore, J.L. (1991), *Probability and Statistics for Engineering and the Sciences*, Brooks/Cole Publishing Company, Pacific Grove, California.

Elsayed, E.A. and T.O. Boucher (1985), *Analysis and Control of Production Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Evans, J.R. and D.L. Olsen (1998), *Introduction to Simulation and Risk Analysis*, Prentice-Hall, Upper Saddle River, New Jersey.

Feller, W. (1968), *An Introduction to Probability Theory and Its Applications*, John Wiley and Sons, New York.

Fishman, G.S. (1973), *Concepts and Methods in Discrete Event Simulation*, John Wiley and Sons, New York.

Fishman, G.S. and L.S. Yarberry (1997), "An Implementation of the Batch Means Method," *INFORMS Journal on Computing*, Vol. 9, 296–310.

Copyright 2007 by Academic Press, Inc. All rights of reproduction in any form reserved.

Gershwin, S.B. (1994), *Manufacturing Systems Engineering*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gray, J. and A. Reuter (1993), *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, California.

Gross, D. and C. Harris (1998), *Fundamentals of Queueing Theory*, 3rd ed., John Wiley and Sons, New York.

Hennessy, J. and D. Patterson (1996), *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, California.

Hill, J.R. and B. Melamed (1995), "TEStool: A Visual Interactive Environment for Modeling Autocorrelated Time Series," *Performance Evaluation*, Vol. 24, No. 1–2, 3–22.

Hoel, P.G., S.C. Port, and C.J. Stone (1971a), *Introduction to Probability Theory*, Houghton Mifflin, Boston, Massachusetts.

Hoel, P.G., S.C. Port, and C.J. Stone (1971b), *Introduction to Statistical Theory*, Houghton Mifflin, Boston, Massachusetts.

Jagerman, D.L. and B. Melamed (1992a), "The Transition and Autocorrelation Structure of TES Processes. Part I: General Theory," *Stochastic Models*, Vol. 8, No. 2, 193–219.

Jagerman, D.L. and B. Melamed (1992b), "The Transition and Autocorrelation Structure of TES Processes. Part II: Special Cases," *Stochastic Models*, Vol. 8, No. 3, 499–527.

Jagerman, D.L. and B. Melamed (1994), "The Spectral Structure of TES Processes," *Stochastic Models*, Vol. 10, No. 3, 599–618.

Jagerman, D.L. and B. Melamed (1995), "Bidirectional Estimation and Confidence Regions for TES Processes," *Proceedings of MASCOTS '95*, Durham, North Carolina, pp. 94–98.

Jagerman, D. and T. Altiok (1999), *Vessel Arrival Process and Queueing in Sea Ports Handling Bulk Materials*, Technical Report, Department of Industrial Engineering, Rutgers University, Piscataway, New Jersey.

Jelenkovic, P. and B. Melamed (1995), "Algorithmic Modeling of TES Processes," *IEEE Transactions on Automatic Control*, Vol. 40, No. 7, 1305–1312.

Johnson, M. and D.C. Montgomery (1974), *Operations Research in Production Planning, Scheduling and Inventory Control*, John Wiley and Sons, New York.

Kant, K. (1992), *Introduction to Computer System Performance Evaluation*, McGraw-Hill, New York.

Kelton, W.D., R.P. Sadowski, and D.T. Sturrock (2004), *Simulation with Arena*, 3rd ed., McGraw-Hill, New York.

Khoshnevis, B. (1994), *Discrete Systems Simulation*, McGraw-Hill, New York.

Kleijnen, J.P.C. (1987), *Statistical Tools for Simulation Practitioners*, Dekker, New York.

Kleinrock, L. (1975), *Queueing Systems*, Vol. 1, John Wiley and Sons, New York.

Law, A.M. (1977), "Confidence Intervals in Discrete Event Simulation: A Comparison of Replication and Batch Means," *Naval Research Logistics Quarterly*, Vol. 24, 667–678.

Law, A.M. (1980), "Statistical Analysis of the Output Data from Terminating Simulations," *Naval Research Logistics Quarterly*, Vol. 27, 131–143.

Law, A.M. and W.D. Kelton (1982), "Confidence Intervals for Steady-State Simulations, II: A Survey of Sequential Procedures," *Management Science*, Vol. 28, No. 5, 550–562.

Law, A.M. and W.D. Kelton (2000), *Simulation Modeling and Analysis*, McGraw-Hill, New York.

Livny, M., B. Melamed, and A.K. Tsiolis (1993), "The Impact of Autocorrelation on Queueing Systems," *Management Science*, Vol. 39, No. 3, 322–339.

Melamed, B. (1991), "TES: A Class of Methods for Generating Autocorrelated Uniform Variates," *ORSA Journal on Computing*, Vol. 3, No. 4, 317–329.

Melamed, B. (1993), "An Overview of TES Processes and Modeling Methodology," in *Performance Evaluation of Computer and Communications Systems*, edited by L. Donatiello and R. Nelson, 359–393, *Lecture Notes in Computer Science*, Springer-Verlag, New York.

Melamed B. and W. Whitt (1990a), "On Arrivals That See Time Averages," *Operations Research*, Vol. 38, 156–172.

Melamed B. and W. Whitt (1990b), "On Arrivals That See Time Averages: A Martingale Approach," *J of Applied Probability*, Vol. 27, 376–384.

Melamed B. and D. Yao (1995), "The ASTA Property," in *Advances in Queueing: Theory, Methods and Open Problems*, edited by J.H. Dshalalow, 195–224, CRC Press, Boca Raton, Florida.

Melamed, B. (1997), "The Empirical TES Methodology: Modeling Empirical Time Series," *Journal of Applied Mathematics and Stochastic Analysis*, Vol. 10, No. 4, 333–353.

Melamed, B. and J.R. Hill (1995), "A Survey of TES Modeling Applications," *SIMULATION*, Vol. 64, No. 6, 353–370.

Menasce, D.A. and V.A.F. Almeida (1998), *Web Performance, Metrics, Models, and Methods*, Prentice-Hall, Upper Saddle River, New Jersey.

Menasce, D.A., V.A.F. Almeida, and L.W. Dowdy (1994), *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*, Prentice-Hall, Upper Saddle River, New Jersey.

Morris, W.T. (1967), "On the Art of Modeling," *Management Science*, Vol. 13, No. 12.

Nelson, B.L. (1992), "Statistical Analysis of Simulation Results," Chapter 2 in *Handbook of Industrial Engineering*, 2nd ed., John Wiley and Sons, New York.

Orfali, R., D. Harkey, and J. Edwards (1996), *The Essential C/S Survival Guide*, John Wiley and Sons, New York.

Papadopoulos, H.T., C. Heavey, and J. Browne (1993), *Queueing Theory in Manufacturing Systems Analysis and Design*, Chapman and Hall, London.

Perros, H.G. (1994), *Queueing Networks with Blocking*, Oxford Press, New York.

Pritsker, A.A.B. (1986), *Introduction to Simulation and Slam II*, 3rd ed., Halsted, New York.

Rockwell Software (2005), *Arena Standard User Guide*, Rockwell Software, Sewickley, Pennsylvania.

Ross, S.M. (1993), *Introduction to Probability Models*, Academic Press, New York.

Schriber, T.J. (1990), *An Introduction to Simulation Using GPSS/H*, John Wiley, New York.

Simchi-Levi, D., P. Kaminsky, and E. Simchi-Levi (2003), *Designing and Managing the Supply Chain*, 2nd ed., McGraw-Hill/Erwin, New York.

Soros Associates (1997), *Capacity Planning and Analysis in Campagne des Bauxite du Guinea*, Technical Report, Soros Associates, Chicago.

Taylor, H.M. and S. Karlin (1984), *An Introduction to Stochastic Modeling*, Academic Press, New York.

White, R.P. (1984), "The Use of Waiting Line Theory in Planning Expansion in Port Facilities," *Dock and Harbour Authority*, Foxlow Publications, London.

Whitt, W. (1983), "Queueing Network Analyzer," *Bell Technical Journal*, Vol. 62, 2779–2815.

This page intentionally left blank

Index

A

Arena, 65
  attributes, 75
  entity, 65
  experiment file (*.exp*), 108
  expressions, 75
  Graphical user interface (GUI), 66, 107
  home screen, 66
  Input Analyzer, 130
  mapping to SIMAN, 109
  Menu bar, 67
  model checking, 110
  model construction, 107
  model file (*.doe*), 108
  model window canvas, 67
  Project bar, 67
  run control, 73, 112
  run tracing, 114
  standard output report, 177
  status bar, 67
  toolbars, 67
  transaction, 65
  variables, 75
Arena Blocks, 95
Autocorrelation function, 48, 197
Automatic guided vehicle (AGV), 119, 316

B

Batch processing, 253
  in Arena, 254
Blocking, 224
Bulletin board, 393
Button
  Animate Connectors, 112
  Animate Connectors, 118
  Break, 112
  Check, 110, 112
  Clock, 119
  Close, 111
  Command, 112
  Date, 119
  Distance, 316
  Edit, 111
  End, 111
  Fast Forward, 111
  Find, 111
  Global, 119
  Go, 110, 111
  Help, 120
  Histogram, 119
  Intersection, 316
  Level, 119
  Network, 316
  Next, 111
  Parking, 316
  Pause, 111
  Plot, 119
  Previous, 111
  Promote Path, 316
  Queue, 119
  Resource, 119
  Route, 316
  Segment, 316
  Seize, 316
  Start Over, 111
  Station, 316
  Step, 111
  Storage, 315
  Transporter, 316
  Variable, 119
  Watch, 112

C

Central Limit Theorem, 41, 174, 189
chart view, 67
Client/Server network, 369
  2-tier architecture, 371
  3-tier architecture, 371
  client hosts, 372
  elapsed time, 374
  flexibility, 369
  interoperability, 369
  n-tier architecture, 371
  scalability, 369
  server hosts, 373

Client/Server network, *(continued)*
transaction monitor, 374
transaction processing (TP), 371
transaction processing (TP) monitor, 372, 376, 380, 388, 392
usability, 369
Coefficient of skewness, 31
Coefficient of variation, 31
Communications network, 374
as a queueing network, 375
bandwidth capacity (BWC), 375
message transfer efficiency (MTE), 375
Confidence interval estimation, 173
for mean (unknown variance), 175
for mean (known variance), 174
in Arena, 176
steady-state simulation, 176
terminating simulation, 173
for means, 186
for variance, 187
Confidence level, 52
Correlation Analysis, 195
in Input Analysis, 195
in Output Analysis, 197
manufacturing example, 219
queueing example, 197
Correlation coefficient, 32
Correlogram, 189, 204
Covariance, 31
Critical value, 52
Cumulative distribution function (cdf), 28
Customer
average, 18
service level, 18

D

Data analysis, 125
Data collection, 5, 124
Data fitting
distribution, 128
multi-modal distributions, 137
parameters, 128
Data modeling, 127
Demurrage cost, 331
DES, 11
Distribution, 27
Bernoulli, 34
beta, 46
binomial, 34
discrete, 33
Erlang, 43
exponential, 39
F, 45
gamma, 42
Gaussian, 40
geometric, 35
Johnson, 40
joint, 29
lognormal, 41
mode, 137
multi-modal, 137
normal, 40
Poisson, 35
Rayleigh, 47
standard normal, 40
step (histogram), 37
Student's t, 44, 175
triangular, 38
uniform, 36
uni-modal, 137
Weibull, 47
Duplicating entities, 255

E

Error
type I, 51
type II, 51
Estimate, 50, 168
point, 171
pooled, 169
Estimator, 50, 168
confidence interval, 51
point, 171
point, 51
time average, 169
unbiased, 50
Event
dependence, 26
independence, 26
list, 12
list, 5
most imminent, 12
probabilistic, 25
scheduling, 12
Example model
2-tier Client/Server network, 375
3-tier Client/Server network, 384
assembly operations, 256
batch processing, 253
bulk port, 316
client nodes, 378
coal loading operations, 324
demand management, 270, 284
distribution center management, 297
entity-dependent service, 340, 353
estimating sojourn (flow) times, 251
gear manufacturing job shop, 346
generic packaging line, 225
hospital emergency room, 84
inventory management, 267, 278, 299, 303, 305
machine failures, 248
manufacturing blocking, 227, 246
multi-production/inventory system, 276
multi-echelon supply chain, 293

multiple arrival-stream generation, 334, 349
observation collection by entity type, 179
open-ended tracing, 114
production/inventory system, 265
request arrivals, 386
retailer inventory management, 295
server node, 380
server node, 391
ship arrivals, 317
single workstation, 69, 149
statistics collection, 272
tandem workstations, 153
tidal window modulation, 328
time-dependent arrival generation, 100
toll plaza, 332
tracing, 114
tracing selected blocks, 116
tracing selected entities, 117
transmission network, 378, 388
transporter-based transportation, 351
tugboat operations, 320
two tandem workstations, 78
workstation with 2 types of parts, 177
Expectation, 30

F

Failure
downtime, 247
forced downtime, 248
machine, 247
operation-dependent, 248
repair, 247
uptime, 247
Field
Conditions, 382
Destination Type, 323
Ending Value, 393
If, 322
Member Attributes, 326
Quantity, 324, 326
Queue Name, 324
Replication Length, 168
Resources, 340
Route Time, 323
Save Attribute, 337
Search Condition, 393
Selection Rule, 351
Set Index, 391
Starting Rank, 324, 326
Starting Value, 393
Station Name, 323
Station Set Members, 336
Stations, 338
Tally Set Name, 355
Terminating Condition, 168
Test Condition, 339
Transfer Type, 339
Transporter Name, 351, 352, 353
Unit Number, 352, 353
Units, 323
Value, 323
Velocity, 352
Frequency, 24

H

Hypothesis
alternative, 51
null, 51
testing, 51
Arena supported distributions, 130
distribution fitting, 131, 134

I

Input Analysis, 123
Input Analyzer, 124, 126, 130
Arena supported distributions, 130
distribution fitting, 134
Inventory policies
(***r,R***), 264
(***R,r***), 265
(***r,R***), 266

K

Kurtosis, 31

L

Lay period, 316
Lemma of Iterated Uniformity, 201
Little's formula, 148, 238, 259, 259

M

Menu
Analyze, 185, 187, 189
Arrange, 67
Fit, 131
Help, 120
Object, 67
Run, 67, 111
Tools, 67
Model
building, 5
computer, 2
descriptive, 3
goodness, 141
mathematical, 2
physical, 2
prescriptive, 3
validation, 6, 141, 161
verification, 2, 141
verification of single workstation, 150
verification of tandem workstations, 158
verification via inspection, 142
verification via performance analysis, 143

Modeling, 1
  analytical vs. simulation, 2
  cost and risks, 6
Modified P-K formula, 219
Module, 65
  Access, 315
  Activate, 315
  Advanced Set, 342
  Allocate, 315
  Assign, 153
  Batch, 253, 254
  Conveyor, 314
  Create, 100, 153, 334
  Decide, 75, 80, 322, 390, 392
  Delay, 156, 342
  Dispose, 72
  Dropoff, 324, 328
  Enter, 266, 314
  Entity, 85
  Expression, 394
  Failure, 248
  Free, 315, 353, 355
  Halt, 315
  Hold, 227, 268, 319, 321
  Leave, 314
  Match, 257
  Move, 315
  PickStation, 314, 338
  Pickup, 324, 327
  Process, 71, 80, 226, 379, 380, 388, 393
  Quene, 229
  Record, 73, 76, 156, 179
  Release, 229, 268, 343
  Request, 315, 351, 353
  Resource, 155, 229, 269
  Route, 314, 323, 328, 390
  Schedule, 100, 342
  Search, 393
  Seize, 154, 268, 353
  Separate, 254, 255
  Sequence, 314, 350
  Set, 157, 179, 341, 391
  Start, 315
  StateSet, 155, 249
  Station, 314
  Statistic, 73, 76, 157, 172, 180, 253, 272
  Stop, 315
  Store, 315
  Transport, 352, 354
  Transporter, 314
  Unstore, 315
Modulo-1 arithmetic, 200
  fractional value, 200
  unit circle, 200
Moments, 30

O

Option
  2-way by Condition, 322
  Batch/Truncate Obs'ns, 185
  Conf. Interval on Std Dev . . ., 187
  Compare Means, 187
  Compare Variances, 187
  Conf. Interval On Mean, 186
  Correlogram, 189
  Count, 77
  Expression, 77
  Fit All Summary, 134
  Lumped, 186
  N-Way by Condition, 382
  Schedule, 100
  Search a Batch, 394
  Search a Queue, 394
  Search an Expression, 394
  Setup . . ., 167
  Take All Representative Values, 326
  Take Specific Representative Values, 326
  Time Between, 77
  Time Interval, 77
Output Analysis, 165
Output Analyzer, 177, 182
  batching and truncating, 185
  confidence interval for mean, 186
  confidence interval for variance, 187
  confidence intervals, 186
  correlogram, 204
  data file, 183
  dependent batches, 186
  graphical statistics, 184
  point estimate for autocorrelation, 189
  summary report, 185
  test for comparing means, 187
  test for comparing variances, 188
Output Statistics, 76

P

Parameter estimation, 168
Parameter fitting, 128
  maximal likelihood method, 129
  method of moments, 128
Parametric Analysis, 165, 190
  capacity planning, 371
  quality of service (QoS), 370, 371
  response time, 371
Point estimation, 171
  for correlation, 189
  in Arena, 172
Probability, 24
Probability density function (pdf), 28
Probability mass function (pmf), 28
  conditional, 25
  law, 47
  value (*p*-value), 52, 136

Process Analyzer, 177, 190
  Controls, 190
Production line, 223
  behavior, 237, 258
  blocking, 224
  bottleneck workstation, 224
  design problems, 225
  performance analysis, 225
  pull regime, 224
  push regime, 223
  Sensitivity Analysis, 238
  starvation, 224
  verification, 238, 258
Project report, 9
p-value, 52, 136

Q

Queueing system, 144
  busy cycles, 146
  busy period, 146
  flow conservation, 148
  idle period, 146
  Little's formula, 148
  manufacturing workstation, 144
  PASTA property, 149
  performance analysis, 143
  performance measures (metrics), 145
  processes and parameters, 144
  regenerative argument, 146
  service disciplines, 145
  throughput, 147

R

Random number generator (RNG),
    15, 55
  linear congruential method, 55
  period, 56
  seed, 55
Regenearation points, 49
  acceptance, 52
  rejection, 52
Renewal points, 49
Replication, 15
Report
  automatic, 77
  Frequencies, 78
  pooled across replications, 181
  Processes, 78
  Queues, 78
  Replication Parameters, 168
  Resources, 78
  specified, 78
  User-Specified, 78
Resource
  autostates, 248
  buffer capacity allocation, 225
  workload allocation, 225
Run control, 111
  keyboard based, 112
  mouse based, 111

S

Sample
  autocorrelation function, 199
  correlation coefficient, 51
  mean, 51
  point, 25
  space, 25
  standard deviation, 51
  statistics, 18
  time average, 51, 171
  variance, 51, 175
Schedule, 100
Sensitivity analysis, 190, 195, 200
  example, 360
Significance level, 52
SIMAN, 65
  model file (*.mod*), 108
  output report (*.out* file), 176
Simulation
  clock, 12
  discrete event, 4, 11
  event, 4
  modeling, 1, 4
  Monte Carlo, 2, 23
  replication, 165
  state, 4
  steady state, 166
  steady state behavior, 167
  terminating, 166
  transient state behavior, 167
  warm-up, 167
  worldview, 4
Sojourn time
  estimation in Arena, 252
Standard deviation, 31
Starvation, 224
State
  trajectory, 11
  transition, 12
Statistical signature, 195
Statistical test
  Chi-Square, 134
  distribution fitting, 134
  for comparing means, 187
  for comparing variances, 187
  Kolmogorov-Smirnov, 137
Statistics
  counter, 76
  entity, 77
  frequency, 76
  output, 76
  tally, 76, 172
  time-persistent, 76, 172

Statistics collection
  Batch Means method, 170
  from independent replications, 169
  from regenerative replications, 170
  from replications, 168
Stochastic process, 47
  iid, 61
  Markov chain, 62
  TES processes, 215
  iid, 48
  Markov, 49
  Markov chain, 50
  Markov renewal, 50
  Poisson, 48
  regenerative, 49
  renewal, 49
  stationary, 48
  TES, 200
Stochastic process generation, 61
  general, 61
Stoppage, 247
Supply Chain Management, 264
  Bullwhip Effect, 264
  Echelons, 263
  Fill Rate, 264
Supply Chains, 263
Synchronization via *Match* module, 257
System
  production line, 223
  state, 11

T

Template panel, 65, 67
  Advanced Transfer, 68, 314
  Advanced Process, 68
  Basic Process, 68
  Blocks, 68
  Common, 68
  Elements, 68
  Reports, 68
  Support, 68
TES processes, 199
  background processes, 202
  background TES$^+$, 202
  background TES$^-$, 202
  basic processes, 202
  computer aided, 202
  distortion, 205
  foreground processes, 205
  foreground TES$^+$, 205
  foreground TES$^-$, 206
  generation, 215
  generation in Arena, 216
  generation of TES$^+$, 215
  generation of TES$^-$, 216
  geometric interpretation, 202
  initial random variable, 202
  innovation process, 201
  smoothing, 209
  stitched, 210
  stitching parameter, 209
  stitching transformation, 208
  unit circle, 202
  unstitched, 210
Time
  average, 18
  persistent, 172
Time dependent parameters, 100
Time dependent processes, 100
Toolbar
  Animate, 68, 118, 316
  Animate Transfer, 119, 315
  Basic View, 68
  Draw, 68
  Integration, 69
  Run Interaction, 69, 111, 118
  Standard, 68, 111

V

Variables
  TFIN, 75
  TNOW, 75
Variance, 31
Variate generation, 56
  discrete distribution, 59
  exponential distribution, 58
  Inverse Transform method, 57, 202, 206, 210, 211
  step distribution, 60, 212
  uniform distribution, 58